# 附件1

**源代码**

1. **头文件**
2. **Matrix3.h**

#pragma once

//变换矩阵,初始为单位矩阵

struct Matrix3 {

double \_m[3][3];

Matrix3() {

makeIdentity();

}

void makeIdentity() {

for (int i = 0; i < 3; ++i)

for (int j = 0; j < 3; ++j)

\_m[i][j] = 0.0;

for (int i = 0; i <= 2; i++) {

\_m[i][i] = 1.0;

}

}

double\* operator[](int i)

{

return \_m[i];

}

Matrix3& makeTranslate(double dx, double dy);

Matrix3& makeRotate(double rad );

Matrix3& makeScale(double sx ,double sy );

Matrix3& makeC\_symmetry(double sx, double sy);

static Matrix3 translate(double dx, double dy);

static Matrix3 rotate(double rad );

static Matrix3 scale(double sx, double sy);

static Matrix3 C\_symmetry(double sx, double sy);

Matrix3 operator\* (Matrix3& other)

{

Matrix3 m;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

m.\_m[i][j] = 0;

for (int k = 0; k < 3; k++) {

m.\_m[i][j] += \_m[i][k] \* other.\_m[k][j];

}

}

}

return m;

}

Matrix3& operator\*= (Matrix3& other)

{

Matrix3 m = \*this \* other;

\*this = m;

return \*this;

}

};

1. **Matrix4.h**

#include "Vec3.h"

#pragma once

struct Matrix4 {

double \_m[4][4];

Matrix4() {

makeIdentity();

}

void makeIdentity() {

for (int i = 0; i < 4; ++i)

for (int j = 0; j < 4; ++j)

\_m[i][j] = 0.0;

for (int i = 0; i <= 3; i++) {

\_m[i][i] = 1.0;

}

}

double\* operator[](int i)

{

return \_m[i];

}

Matrix4& makeTranslate(double dx, double dy, double dz);

Matrix4& makeRotate\_x(double rad);//x轴旋转矩阵

Matrix4& makeRotate\_y(double rad);//y轴旋转矩阵

Matrix4& makeRotate\_z(double rad);//z轴旋转矩阵

Matrix4& makeScale(double sx, double sy, double sz);

Matrix4& makeC\_symmetry(double sx, double sy, double sz);

static Matrix4 translate(double dx, double dy, double dz);

static Matrix4 rotate\_x(double rad);

static Matrix4 rotate\_y(double rad);

static Matrix4 rotate\_z(double rad);

static Matrix4 scale(double sx, double sy,double sz);

static Matrix4 C\_symmetry(double sx, double sy, double sz);

Matrix4 operator\* (Matrix4& other)

{

Matrix4 m;

for (int i = 0; i < 4; i++) {

for (int j = 0; j < 4; j++) {

m.\_m[i][j] = 0;

for (int k = 0; k < 4; k++) {

m.\_m[i][j] += \_m[i][k] \* other.\_m[k][j];

}

}

}

return m;

}

Matrix4& operator\*= (Matrix4& other)

{

Matrix4 m = \*this \* other;

\*this = m;

return \*this;

}

Matrix4 makePrespective(double g\_fovy, double aspectRatio, double g\_zNear, double g\_zFar);

Matrix4 makeLookAt(Vec3& eye, Vec3& center, Vec3& up);

static Matrix4 Trans\_Matrix(Matrix4 arr);//转置

static Matrix4 Gaussian\_elimination(Matrix4 arr);//求逆

};

1. **Raster.h**

#pragma once

#include<math.h>

#include "Graphic.h"

#include "GeoDefine.h"

#include <utility>

#include "GeoDefin3D.h"

//链表的实现

typedef struct XET {

double x;

double dx, ymax;

XET\* next;

}AET, NET;

void LineBres(int x0, int y0, int xend, int yend);

void Polygon(PixelPoint\* pts, int num);

double backculling(Vec4& pt1, Vec4& pt2, Vec4& pt3);

void Z\_Buffer(Vertex3D pixelSet[]);

double computeA(Vec4& p1, Vec4& p2, Vec4& p3);

double computeB(Vec4& p1, Vec4& p2, Vec4& p3);

double computeC(Vec4& p1, Vec4& p2, Vec4& p3);

double computeD(Vec4& p1, Vec4& p2, Vec4& p3);

1. Render2D.h

#pragma once

#include "Matrix3.h"

#include <iostream>

#include "stdio.h"

#include "conio.h"

struct Dataset;

struct Layer;

struct Geometry;

class Render2D

{

public:

void init(double dx, double dy, double w, double h, int WWidth, int WHeight);

void setWindow(double left, double right, double bottom, double top);

void setViewport(int x, int y, int width, int height);

void translate(double x, double y);

void rotate(double angle);

void rotate(double x, double y, double angle);

void scale(double sx, double sy);

void scale(int x, int y, double d);//指定点缩放//指定点缩放

void render(Dataset& g\_Dataset);

void render(Layer\* pLayer );

void render(Geometry\* pGeometry );

void C\_symmetry(int x, int y); //关于原点对称

//bool ClipT(float p, float q, float\* u1, float\* u2);//判断区域

//void LiangLine(int xwmin, int ywmin, int xwmax, int ywmax, int x1, int y1, int x2, int y2);

Matrix3 m;

Matrix3 viewMatrix;

Matrix3 projectMatrix;

Matrix3 viewportMatrix;

Matrix3 getPixlToWorldMatrix;

1. **Render3D.h**

#pragma once

#include "Matrix4.h"

#include <iostream>

#include "stdio.h"

#include "conio.h"

struct Dataset;

struct Layer;

struct Geometry;

class Render3D

{

public:

void init(double dx, double dy, double dz, double w, double l, double h, int WWidth, int wlength, int WHeight);

void setWindow(double left, double back, double right, double bottom, double front, double top);

void setViewport(int x, int y, int z, int width, int length, int height);

void translate(double x, double y, double z);

void rotate\_x(double angle);

void rotate\_y(double angle);

void rotate\_z(double angle);

void rotate(double x, double y, double angle);

void scale(double sx, double sy, double sz);

void scale(int x, int y, int z, double d);//指定点缩放//指定点缩放

void C\_symmetry(int x, int y, int z); //关于原点对称

Matrix4 m;

Matrix4 g\_modelMatrix;//模型变换矩阵

Matrix4 g\_transformMatrix3D;//总变换矩阵

Matrix4 g\_viewMatrix;//视图变换矩阵

Matrix4 projectMatrix;

Matrix4 viewportMatrix;

};

1. **Vec3.h**

#pragma once

#include "Matrix3.h"

#include<math.h>

struct Vec3

{

Vec3()

{

\_v[0] = \_v[1] = \_v[2] = 0.0;

}

Vec3(double x, double y, double z)

{

\_v[0] = x;

\_v[1] = y;

\_v[2] = z;

}

Vec3(double x, double y)

{

\_v[0] = x;

\_v[1] = y;

\_v[2] = 1.0;

}

void set(double x, double y, double z)

{

\_v[0] = x;

\_v[1] = y;

\_v[2] = z;

}

double& x() { return \_v[0]; }

double& y() { return \_v[1]; }

double& z() { return \_v[2]; }

double& operator[](int i )

{

return \_v[i];

}

Vec3 operator\*(Matrix3& m)

{

Vec3 v;

for (int i = 0; i < 3; i++) {

v[i] = \_v[0] \* m[0][i] + \_v[1] \* m[1][i] + \_v[2] \* m[2][i];

}

return v;

}

Vec3& operator\*=(Matrix3& m)

{

Vec3 v = \*this \* m;

\*this = v;

return \*this;

}

Vec3 operator-()

{

Vec3 t;

for (int i = 0; i < 3; ++i)

{

t[i] = -\_v[i];

}

return t;

}

Vec3 operator-(Vec3& v)

{

Vec3 t;

for (int i = 0; i < 3; ++i)

{

t[i] = \_v[i] - v[i];

}

return t;

}

Vec3 operator+(Vec3& v)//加

{

Vec3 t;

for (int i = 0; i < 3; ++i)

{

t[i] = \_v[i] + v[i];

}

return t;

}

Vec3 operator+=(Vec3 &v)

{

Vec3 v1 = \*this + v;

\*this = v1;

return \*this;

}

Vec3 operator/(int m)//除法

{

Vec3 v;

for (int i = 0; i < 3; i++) {

v[i] = \_v[i] / m;

}

return v;

}

double operator\*(Vec3& v) //点积

{

double t;

t = \_v[0] \* v[0] + \_v[1] \* v[1] + \_v[2] \* v[2];

return t;

}

Vec3 operator^(Vec3& v) //叉积

{

Vec3 t;

t[0] = \_v[1] \* v[2] - \_v[2] \* v[1];

t[1] = -(\_v[2] \* v[0] - \_v[0] \* v[2]);

t[2] = \_v[0] \* v[1] - \_v[1] \* v[0];

return t;

}

void operator-=(Vec3& v)

{

for (int i = 0; i < 3; ++i)

{

\_v[i] -= v[i];

}

}

void operator\*=(double s) //标量乘法

{

for (int i = 0; i < 3; ++i)

{

\_v[i] \*= s;

}

}

void normalize() //归一化

{

double l = length();

for (int i = 0; i < 3; ++i)

\_v[i] /= l;

}

double length() //求长度

{

return sqrt(\_v[0] \* \_v[0] + \_v[1] \* \_v[1] + \_v[2] \* \_v[2]);

}

double \_v[3];

};

1. **Vec4.h**

#pragma once

#include "Matrix4.h"

struct Vec4

{

double \_v[4];

Vec4()

{

\_v[0] = \_v[1] = \_v[2] = \_v[3] = 0.0;

}

Vec4(double x, double y, double z, double k)

{

\_v[0] = x;

\_v[1] = y;

\_v[2] = z;

\_v[3] = k;

}

Vec4(double x, double y, double z)

{

\_v[0] = x;

\_v[1] = y;

\_v[2] = z;

\_v[3] = 1.0;

}

Vec4(Vec3& \_v3)

{

\_v[0] = \_v3[0];

\_v[1] = \_v3[1];

\_v[2] = \_v3[2];

\_v[3] = 1.0;

}

void set(double x, double y, double z, double k)

{

\_v[0] = x;

\_v[1] = y;

\_v[2] = z;

\_v[3] = k;

}

double& x() { return \_v[0]; }

double& y() { return \_v[1]; }

double& z() { return \_v[2]; }

double& w() { return \_v[3]; }

double& operator[](int i)

{

return \_v[i];

}

Vec4 operator\*(Matrix4& m)

{

Vec4 v;

for (int i = 0; i < 4; i++) {

v[i] = \_v[0] \* m[0][i] + \_v[1] \* m[1][i] + \_v[2] \* m[2][i] + \_v[3] \* m[3][i];

}

return v;

}

Vec4& operator\*=(Matrix4& m)

{

Vec4 v = \*this \* m;

\*this = v;

return \*this;

}

Vec4 operator\*(Vec4& v)

{

Vec4 t;

for (int i = 0; i < 4; i++)

{

t[i] = \_v[i] \* v[i];

}

return t;

}

Vec4 operator\*(double n)

{

Vec4 v;

for (int i = 0; i < 4; i++)

{

v[i] = \_v[i] \* n;

}

return v;

}

Vec4 operator+(Vec4& v)

{

Vec4 t;

for (int i = 0; i < 4; ++i)

{

t[i] = \_v[i] + v[i];

}

return t;

}

Vec4 operator+=(Vec4& v1)

{

for (int i = 0; i < 4; ++i)

{

v1[i] = \_v[i] + v1[i];

}

return v1;

}

Vec4 operator-(Vec4& v)

{

Vec4 t;

for (int i = 0; i < 4; ++i)

{

t[i] = \_v[i] - v[i];

}

return t;

}

double operator^(Vec4& v) //点积

{

double t;

t = \_v[0] \* v[0] + \_v[1] \* v[1] + \_v[2] \* v[2] + \_v[3] \* v[3];

return t;

}

Vec4 operator/(double m)//除法

{

Vec4 v;

for (int i = 0; i < 4; i++) {

v[i] = \_v[i] / m;

}

return v;

}

void normalize() //归一化

{

double l = length();

for (int i = 0; i < 4; ++i)

\_v[i] /= l;

}

double length() //求长度

{

return sqrt(\_v[0] \* \_v[0] + \_v[1] \* \_v[1] + \_v[2] \* \_v[2] + \_v[3] \* \_v[3]);

}

};

1. **源代码**
2. **Matrix3.cpp**

#include "Matrix3.h"

#include <math.h>

Matrix3& Matrix3::makeTranslate(double dx, double dy)

{

makeIdentity();

\_m[2][0] = dx;

\_m[2][1] = dy;

return \*this;

}

Matrix3& Matrix3::makeRotate(double rad)

{

makeIdentity();

double \_cos = cos(rad);

double \_sin = sin(rad);

\_m[0][0] = \_cos;

\_m[0][1] = \_sin;

\_m[1][0] = -\_sin;

\_m[1][1] = \_cos;

return \*this;

}

Matrix3& Matrix3::makeScale(double sx, double sy)

{

makeIdentity();

\_m[0][0] = sx;

\_m[1][1] = sy;

return \*this;

}

Matrix3& Matrix3::makeC\_symmetry(double sx, double sy)

{

makeIdentity();

\_m[0][0] = -1;

\_m[1][1] = -1;

\_m[2][2] = 1;

return \*this;

}

Matrix3 Matrix3::translate(double dx, double dy)

{

Matrix3 m;

m.makeTranslate(dx, dy);

return m;

}

Matrix3 Matrix3::rotate(double rad)

{

Matrix3 m;

m.makeRotate(rad);

return m;

}

Matrix3 Matrix3::scale(double sx, double sy)

{

Matrix3 m;

m.makeScale(sx, sy );

return m;

}

Matrix3 Matrix3::C\_symmetry(double sx, double sy)

{

Matrix3 m;

m.makeC\_symmetry(sx, sy);

return m;

}

1. **Matrix4.cpp**

#include "Matrix4.h"

#include <math.h>

#include "Define.h"

Matrix4& Matrix4::makeTranslate(double dx, double dy,double dz)

{

makeIdentity();

\_m[3][0] = dx;

\_m[3][1] = dy;

\_m[3][2] = dz;

return \*this;

}

Matrix4& Matrix4::makeRotate\_x(double rad)

{

makeIdentity();

double \_cos = cos(rad);

double \_sin = sin(rad);

\_m[1][1] = \_cos;

\_m[1][2] = \_sin;

\_m[2][1] = -\_sin;

\_m[2][2] = \_cos;

return \*this;

}

Matrix4& Matrix4::makeRotate\_y(double rad)

{

makeIdentity();

double \_cos = cos(-rad);

double \_sin = sin(-rad);

\_m[0][0] = \_cos;

\_m[0][2] = \_sin;

\_m[2][0] = -\_sin;

\_m[2][2] = \_cos;

return \*this;

}

Matrix4& Matrix4::makeRotate\_z(double rad)

{

makeIdentity();

double \_cos = cos(rad);

double \_sin = sin(rad);

\_m[0][0] = \_cos;

\_m[0][1] = \_sin;

\_m[1][0] = -\_sin;

\_m[1][1] = \_cos;

return \*this;

}

Matrix4& Matrix4::makeScale(double sx, double sy, double sz)

{

makeIdentity();

\_m[0][0] = sx;

\_m[1][1] = sy;

\_m[2][2] = sz;

return \*this;

}

Matrix4& Matrix4::makeC\_symmetry(double sx, double sy, double sz)

{

makeIdentity();

\_m[0][0] = -1;

\_m[1][1] = -1;

\_m[2][2] = -1;

return \*this;

}

Matrix4 Matrix4::translate(double dx, double dy,double dz)

{

Matrix4 m;

m.makeTranslate(dx, dy, dz);

return m;

}

Matrix4 Matrix4::rotate\_x(double rad)

{

Matrix4 m;

m.makeRotate\_x(rad);

return m;

}

Matrix4 Matrix4::rotate\_y(double rad)

{

Matrix4 m;

m.makeRotate\_y(rad);

return m;

}

Matrix4 Matrix4::rotate\_z(double rad)

{

Matrix4 m;

m.makeRotate\_z(rad);

return m;

}

Matrix4 Matrix4::scale(double sx, double sy,double sz)

{

Matrix4 m;

m.makeScale(sx, sy, sz);

return m;

}

Matrix4 Matrix4::C\_symmetry(double sx, double sy, double sz)

{

Matrix4 m;

m.makeC\_symmetry(sx, sy, sz);

return m;

}

Matrix4 Matrix4::makePrespective(double g\_fovy, double aspectRatio,

double g\_zNear, double g\_zFar)

{

//透视投影矩阵

const double S = 1 / (tan(DEG\_RAD(g\_fovy / 2)));

\_m[0][0] = S / aspectRatio;

\_m[1][1] = S;

\_m[2][2] = -(g\_zFar / (g\_zFar - g\_zNear));

\_m[2][3] = -1;

\_m[3][2] = -(g\_zFar \* g\_zNear) / (g\_zFar - g\_zNear);

\_m[3][3] = 0;

return \*this;

}

Matrix4 Matrix4::makeLookAt(Vec3& eye, Vec3& center, Vec3& up)

{

//观察变换矩阵

Vec3 x, y, z;

//z轴

z = eye - center;

z.normalize();

//x轴

x = up ^ z;

x.normalize();

//y轴

y = z ^ x;

y.normalize();

double dx = -x \* eye;

double dy = -y \* eye;

double dz = -z \* eye;

for (int i = 0; i < 3; ++i) //矩阵

\_m[i][0] = x[i];

for (int i = 0; i < 3; ++i)

\_m[i][1] = y[i];

for (int i = 0; i < 3; ++i)

\_m[i][2] = z[i];

\_m[3][0] = dx;

\_m[3][1] = dy;

\_m[3][2] = dz;

return \*this;

}

Matrix4 Matrix4::Trans\_Matrix(Matrix4 arr) {

Matrix4 aT;

for (int i = 0; i < 4; i++) {

for (int j = 0; j < 4; j++) {

aT[j][i] = arr[i][j];

}

}

return aT;

}

//求逆

Matrix4 Matrix4::Gaussian\_elimination(Matrix4 arr)

{

int i, j, k;

float W[4][8];

Matrix4 result;

float tem\_1, tem\_2, tem\_3;

// 对矩阵右半部分进行扩增

for (i = 0; i < 4; i++) {

for (j = 0; j < 8; j++) {

if (j < 4) {

W[i][j] = (float)arr[i][j];

}

else {

W[i][j] = (float)(j - 4 == i ? 1 : 0);

}

}

}

for (i = 0; i < 4; i++)

{

// 判断矩阵第一行第一列的元素是否为0，若为0，继续判断第二行第一列元素，直到不为0，将其加到第一行

if (((int)W[i][i]) == 0)

{

for (j = i + 1; j < 4; j++)

{

if (((int)W[j][i]) != 0) break;

}

if (j == 4)

{

//printf("这个矩阵不能求逆");

break;

}

//将前面为0的行加上后面某一行

for (k = 0; k < 2 \* 4; k++)

{

W[i][k] += W[j][k];

}

}

//将前面行首位元素置1

tem\_1 = W[i][i];

for (j = 0; j < 8; j++)

{

W[i][j] = W[i][j] / tem\_1;

}

//将后面所有行首位元素置为0

for (j = i + 1; j < 4; j++)

{

tem\_2 = W[j][i];

for (k = i; k < 8; k++)

{

W[j][k] = W[j][k] - tem\_2 \* W[i][k];

}

}

}

// 将矩阵前半部分标准化

for (i = 4 - 1; i >= 0; i--)

{

for (j = i - 1; j >= 0; j--)

{

tem\_3 = W[j][i];

for (k = i; k < 8; k++)

{

W[j][k] = W[j][k] - tem\_3 \* W[i][k];

}

}

}

//得出逆矩阵

for (i = 0; i < 4; i++)

{

for (j = 4; j < 8; j++)

{

result[i][j - 4] = W[i][j];

}

}

return result;

}

1. **MessageHandler.cpp**

#include <windows.h>

#include "MessageHandler.h"

#include "resource.h"

#include "Graphic.h"

#include "GeoDefine.h"

#include "GeometryFactory.h"

#include "Render2D.h"

#include "Render3D.h"

#include "DialogHelper.h"

#include "LayerImporter.h"

#include "cutter.h"

#include "GeoDefin3D.h"

#include "Vec4.h"

#include "Matrix4.h"

#include "Matrix3.h"

#include "Raster.h"

#include "Light.h"

enum OperationType {

otNone, otPan, otScaleOfCenter, otScaleOfFixedPoint,

otDrawRectangle, otDrawRectangleOutline,

otDrawLine, otDrawPolyline, otDrawPolygon, otDrawPolygonOutline,

otDrawCircle, otDrawEllipse, otCuttRectange, otCuttPolygon, ot3DRotate

};

bool g\_3Dmode = false; //是否为3D模式

OperationType g\_OperationType = otPan;//当前操作类型

Dataset g\_Dataset;//数据集

Layer\* g\_pLayer = NULL;

Render2D g\_Renderer;

Matrix4 g\_modeMatrix; //3D矩阵

Matrix4 g\_viewMatrix; //3D视图变换矩阵

Matrix4 g\_projectMatrix; //3D投影变换矩阵

Matrix4 g\_transformMatrix3D; //3D总变换矩阵

Mesh\* g\_pMesh = 0;

//void refreshMenuState();//更新菜单状态

Layer\* openLayer();

///处理菜单消息menuID

void handleMenuMessage(int menuID)

{

switch (menuID)

{

case ID\_PAN:

g\_OperationType = otPan;

setRubberMode(rmNone);//self:关闭橡皮线模式

setCursor(csHand);

break;

case ID\_2D\_DRAW\_RECT:

g\_OperationType = otDrawRectangle;

setRubberMode(rmRectangle);

setCursor(csArrow);

break;

case ID\_2D\_DRAW\_LINE:

g\_OperationType = otDrawLine;

setRubberMode(rmLine);

setCursor(csArrow);

break;

case ID\_2D\_DRAW\_POLYLINE:

g\_OperationType = otDrawPolyline;

setRubberMode(rmPolyline);

setCursor(csCross);

break;

case ID\_2D\_DRAW\_POLYGON:

g\_OperationType = otDrawPolygon;

setRubberMode(rmPolygon);

setCursor(csCross);

break;

case ID\_OPEN\_LAYER:

{

Layer\* pLayer = openLayer();

if (pLayer)

{

if (g\_Dataset.getLayerCount() == 1)//添加第一个图层时设置初始变换矩阵和比例尺

{

double dx = /\*-\*/pLayer->envelop.centerX();

double dy = /\*-\*/pLayer->envelop.centerY();

//void init(double dx, double dy, T pLayer->envelop.width(), T pLayer->envelop.height(), int getWindowWidth(), int getWindowHeight());

g\_Renderer.init(dx, dy, pLayer->envelop.width(), pLayer->envelop.height(), getWindowWidth(), getWindowHeight());

}

g\_Dataset.addLayer(pLayer);

refreshWindow();

}

break;

}

case ID\_MODE\_2D:

g\_3Dmode = false;

refreshWindow();

break;

case ID\_MODE\_3D:

g\_3Dmode = true;

refreshWindow();

break;

}

//refreshMenuState();//刷新菜单状态

}

///处理按键消息key

void handleKeyMessage(int key)

{

#pragma region 2D

double pan = 10.0;

double angle = DEG\_RAD(30);//self: 角度转化为弧度

switch (key)

{

case VK\_UP: // 上一行，上光标

if (isCtrlKeyPressed())

g\_Renderer.C\_symmetry(getWindowWidth() / 2, getWindowHeight() / 2); //对称

else

g\_Renderer.translate(0, pan);

break;

case VK\_DOWN:

g\_Renderer.translate(0, -pan);

break;

case VK\_LEFT:

if (isCtrlKeyPressed())

g\_Renderer.rotate(getWindowWidth() / 2, getWindowHeight() / 2, angle);

else

g\_Renderer.translate(-pan, 0);

break;

case VK\_RIGHT:

if (isCtrlKeyPressed())

g\_Renderer.rotate(getWindowWidth() / 2, getWindowHeight() / 2, -angle);

else

g\_Renderer.translate(pan, 0);

break;

default:

return;

}

#pragma endregion

#pragma region 3D

double rotate\_x = PI / 16;

double rotate\_y = PI / 16;

double rotate\_z = PI / 16;

double step = 10.0;

Matrix4 m;

switch (key)

{

case VK\_UP:

{

if (isShiftKeyPressed())

m.makeRotate\_x(-rotate\_x); //x轴旋转

else if (isCtrlKeyPressed())

m.makeTranslate(0, 0, -step);

else

m.makeTranslate(0, -step, 0);

}

break;

case VK\_DOWN:

{

if (isShiftKeyPressed())

m.makeRotate\_x(rotate\_x);

else if (isCtrlKeyPressed())

m.makeTranslate(0, 0, step);

else

m.makeTranslate(0, step, 0);

}

break;

case VK\_LEFT:

{

if (isCtrlKeyPressed())

m.makeRotate\_z(rotate\_z); //z轴旋转

else if (isShiftKeyPressed())

m.makeRotate\_y(rotate\_y); //y轴旋转

else

m.makeTranslate(-step, 0, 0);

}

break;

case VK\_RIGHT:

{

if (isCtrlKeyPressed())

m.makeRotate\_z(-rotate\_z);

else if (isShiftKeyPressed())

m.makeRotate\_y(-rotate\_y);

else

m.makeTranslate(step, 0, 0);

}

break;

default:

return;

}

g\_modeMatrix \*= m;

refreshWindow();

#pragma endregion

}

Geometry\* createGeometry(OperationType operationType, vector<PixelPoint>& pts);

///处理鼠标消息message

void handleMouseMessage(int message, int x, int y, int det)//int det = message == WM\_MOUSEWHEEL ? (short)HIWORD(wParam) : 0;

{

DPtToLPt(x, y, x, y);//窗口设备坐标转为窗口逻辑坐标

static Point2D pt1;//self: 2D点对象？

static bool dragging = false;

switch (message)

{

case WM\_LBUTTONDOWN:

{

if (g\_OperationType == otPan)//当前操作类型

{

pt1.x = x;

pt1.y = y;

dragging = true;

}

break;

}

case WM\_MOUSEMOVE:

{

if (dragging)

{

if (pt1.x != x || pt1.y != y)

{

g\_Renderer.translate(x - pt1.x, y - pt1.y);

pt1.x = x;

pt1.y = y;

refreshWindow();

}

}

break;

}

case WM\_MOUSEWHEEL:

{

if (!g\_3Dmode) {

if (isCtrlKeyPressed())

{

x = 0;

y = 0;

}

double d = det > 0 ? 0.9 : 1.0 / 0.9;//self:鼠标滚动条操作状态决定缩放状态

g\_Renderer.scale(x, y, d);

}

else

{

double d = det > 0 ? -10.0 : 10.0;

g\_viewMatrix \*= Matrix4::translate(0, 0, d);

}

refreshWindow();

break;

}

case WM\_LBUTTONUP:

case WM\_RBUTTONUP:

{

dragging = false;

if (getRubberMode() == rmNone) return;//非橡皮线模式，退出

int c = getRubberPointCount();

if (c < 2) return;//橡皮线顶点数<2 ,不能生成有效图形，退出

//橡皮线操作结束时才会创建图形对象，以下判断橡皮线操作是否结束

switch (g\_OperationType)

{

case otDrawLine:

case otDrawCircle:

case otDrawRectangle:

case otDrawRectangleOutline:

case otDrawEllipse:

if (message == WM\_RBUTTONUP)return;//鼠标右键弹起是Polygon或Polyline模式结束标志

break;

case otDrawPolyline:

case otDrawPolygon:

case otDrawPolygonOutline:

if (message == WM\_LBUTTONUP)return;//鼠标左键弹起是Line或Rectange模式结束标志

break;

default:

return;

}

vector<PixelPoint> pts(c);//pts为动态数组，数组大小为c

getRubberPoints(pts.data());//pts保存橡皮线顶点集合

//橡皮线操作结束，根据橡皮线顶点生成图形

Geometry\* pGeometry = createGeometry(g\_OperationType, pts);

if (pGeometry)

{

if (g\_OperationType == otCuttRectange || g\_OperationType == otCuttPolygon)

{

Cutter cutter(pGeometry->getEnvelop());

bool rs = cutter.cut(g\_pLayer);

delete pGeometry;

if (rs == false)return;

}

else

{

//g\_Editor.updataGeometry(pGeometry, g\_Renderer.getPixlToWorldMatrix());

g\_pLayer->addGeometry(pGeometry);

}

refreshWindow();

g\_pLayer->addGeometry(pGeometry);

refreshWindow();

}

}

break;

}

}

void display()

{

setYUp(true);//y轴向上

setOrig(0, getWindowHeight());

if (!g\_3Dmode)

{

g\_Renderer.render(g\_Dataset);

//g\_Renderer.renderSelectionSet(g\_Selection.getSelectionSet());//渲染选择集

}

else

{

int width = getWindowWidth(), height = getWindowHeight();

if (width == 0 || height == 0) return;

//模型矩阵\*视口矩阵\*投影矩阵，进行缩放变换到0-1之间

g\_transformMatrix3D = g\_modeMatrix \* g\_viewMatrix \* g\_projectMatrix \* Matrix4::scale(0.5, 0.5, 1.0) \* Matrix4::translate(0.5, 0.5, 0.0);

#pragma region vertex transform

//投影变换

vector<Vertex3D>& vertexSet = g\_pMesh->vertexSet;

static vector<Vertex3D> transformedVertexSet;

transformedVertexSet.resize(vertexSet.size());

//光照

Vertex3D pixelSet[3];

Vec4 eye(100, 100, 100, 0);//视点

//N = N0 + N1 + N2;

Matrix4 g\_normalMatrix = Matrix4::Trans\_Matrix(Matrix4::Gaussian\_elimination(g\_modeMatrix));

//N = N \* g\_normalMatrix;//法线矩阵：g\_modeMatrix求逆后转置即为法线矩阵

//光照模型

Light light;

Material M;

Vec4 L;//光向量：连接光源点和代求点

Vec4 V;//视点向量：连接视点和代求点

//Vec4 H;

for (int i = 0, c = vertexSet.size(); i < c; ++i)

{

transformedVertexSet[i].attribute=vertexSet[i].attribute;

Vec4(transformedVertexSet[i].attribute.normal) \*= g\_normalMatrix;//法线×法线矩阵

Vec4(transformedVertexSet[i].attribute.normal).normalize();

transformedVertexSet[i].pos = vertexSet[i].pos \* g\_transformMatrix3D;

Vec4 pt\_w = vertexSet[i].pos\*g\_modeMatrix;

Vec4 N = transformedVertexSet[i].attribute.normal;

//光强

L = light.pos - pt\_w;

V = eye - pt\_w;

//H = (L + V) / 2;

L.normalize();

V.normalize();

//H.normalize();

//计算顶点光强

for (int m = 0; m < 3; m++) {

//光照模型

transformedVertexSet[i].attribute.color[m] = max(0,light.ambient[m] \* M.k\_ambient[m] + (light.diffuse[m] \* M.k\_diffuse[m] \* (N^L)));// (light.specular[m] \* M.k\_shininess \* (H^N));

}

}

#pragma endregion

vector<Face>& faceSet = g\_pMesh->faceSet;

for (int i = 0, c = faceSet.size(); i < c; i++)

{

Face& face = faceSet[i];

//剪裁

#pragma region cull

int clipFlag[3] = { 0 };

for (int j = 0; j < 3; ++j)

{

Vec4& pos = transformedVertexSet[face[j]].pos;

double w = pos.w();

double x = pos.x();

double y = pos.y();

double z = pos.z();

if (w < 0 || x<0 || x>w || y<0 || y>w || z<0 || z>w)

{

clipFlag[j] = 1;

}

}

if (clipFlag[0] || clipFlag[1] || clipFlag[2])

{

continue; //有一个点不可见即删除整个三角形

}

#pragma endregion

#pragma region viewportTransform

Vertex3D pixelSet[3];

for (int j = 0; j < 3; ++j)

{

//视口变换

Vertex3D& v = transformedVertexSet[face[j]];

Vec4& pos = v.pos;

double r\_w = 1.0 / pos.w();

pixelSet[j].pos.x() = pos.x() \* r\_w \* (width - 1);

pixelSet[j].pos.y() = pos.y() \* r\_w \* (height - 1);

pixelSet[j].pos.z() = pos.z() \* r\_w;

pixelSet[j].attribute = v.attribute; //光栅化属性插值

}

#pragma endregion

#pragma region BackCulling

double v = backculling(pixelSet[0].pos, pixelSet[1].pos, pixelSet[2].pos);

if (v >= 0)

{

continue;

}

#pragma endregion

#pragma region Blanking

Z\_Buffer(pixelSet);

#pragma endregion

#pragma region rasterize

for (int j = 0; j < 3; ++j)

{

//像素坐标绘制可见的

Vec4& pt1 = pixelSet[j].pos;

Vec4& pt2 = pixelSet[(j + 1) % 3].pos;

//drawLine(pt1.x(), pt1.y(), pt2.x(), pt2.y());

//LineBres(pt1.x(), pt1.y(), pt2.x(), pt2.y());

}

#pragma endregion

}

}

}

///根据操作类型operationType和顶点集合pts创建合适的图形对象

Geometry\* createGeometry(OperationType operationType, vector<PixelPoint>& pts)

{

int count = pts.size();

if (count < 2) return NULL;

PixelPoint\* pData = pts.data();

///不同操作类型可以生成同一类型的图形，比如Rectange模式生成的图形也是多边形

switch (operationType)

{

case otDrawLine:

return GeometryFactory::createPolylineGeometry(pData, count);

case otDrawRectangle:

return GeometryFactory::creatRectangleGeometry(pData, count);

case otDrawRectangleOutline:

return GeometryFactory::creatRectangleOutlineGeometry(pData, count);

case otDrawPolyline:

return GeometryFactory::createPolylineGeometry(pData, count);

case otDrawPolygon:

return GeometryFactory::createPolygonGeometry(pData, count);

case otDrawPolygonOutline:

return GeometryFactory::createPolygonOutlineGeometry(pData, count);

case otDrawCircle:

return GeometryFactory::createCircleGeometry(pData, count);

case otDrawEllipse:

return GeometryFactory::createEllipseGeometry(pData, count);

}

return NULL;

}

//选择并打开图层

Layer\* openLayer()

{

char path[MAX\_PATH] = { 0 };

if (DialogHelper::selectSingleFile("", "shp", path, MAX\_PATH))

{

return LayerImporter::importShpLayer(path);

}

return NULL;

}

#pragma region less-used

double g\_fovy = 60, g\_zNear = 10, g\_zFar = 1600; //y方向张角，视点到近平面的距离，视点到远平面的距离

///处理窗口大小变化消息

void sized(int cx, int cy)

{

if (cx == 0 || cy == 0) return;

double aspectRatio = (double)cx / cy; //宽高比

g\_projectMatrix.makePrespective(g\_fovy, aspectRatio, g\_zNear, g\_zFar);

}

///初始化

void initialize()

{

g\_pLayer = new Layer();

g\_Dataset.addLayer(g\_pLayer);

g\_OperationType = ot3DRotate;

int width = getWindowWidth(), height = getWindowHeight();

if (width == 0 || height == 0) return;

double aspectRatio = (double)width / height;

g\_projectMatrix.makePrespective( g\_fovy, aspectRatio, g\_zNear, g\_zFar); //投影矩阵

//在世界坐标系中定义观察坐标系

Vec3 eye(0, 0, 400), center(0, 0, 0), up(0, 1, 0);

g\_viewMatrix.makeLookAt(eye, center, up);

//世界像素坐标到视口像素坐标

g\_pMesh = Box3D().toMesh();

vector<Vertex3D> vertexSet = g\_pMesh->vertexSet;

vertexSet.resize(vertexSet.size());

Face face;

Vertex3D pixelSet[3];

for (int i = 0; i < 3; i++) {

Vertex3D& v = vertexSet[face[i]];

Vec4& pos = v.pos \* g\_modeMatrix;

//转换为像素坐标

pixelSet[i].pos.x() = pos.x();

pixelSet[i].pos.y() = pos.y();

pixelSet[i].pos.z() = pos.z();

}

g\_modeMatrix.makeScale(100, 100, 100);

g\_modeMatrix \*= Matrix4::translate(0, 0, -100);

}

///销毁窗口时做清理工作

void destroy()

{

delete g\_pMesh;

}

#pragma endregion

#pragma region handleMessage

///所有消息的入口点

LRESULT handleMessage(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message)

{

case WM\_COMMAND://菜单消息

{

int menuID = LOWORD(wParam);

handleMenuMessage(menuID);

return FALSE;

}

case WM\_PAINT://绘制消息

display();

return FALSE;

case WM\_CREATE:

init((unsigned)hWnd);

initialize();

return TRUE;

case WM\_DESTROY:

destroy();

PostQuitMessage(0);

return TRUE;

case WM\_KEYDOWN://按键消息

{

int key = wParam;//获得按键代码

handleKeyMessage(key);

return FALSE;

}

case WM\_MOUSEMOVE://鼠标移动消息

case WM\_LBUTTONDOWN://鼠标左键按下消息

case WM\_LBUTTONUP://鼠标左键弹起消息

case WM\_RBUTTONUP://鼠标右键弹起消息

case WM\_RBUTTONDOWN://鼠标右键按下消息

case WM\_LBUTTONDBLCLK://鼠标双击消息

case WM\_MOUSEWHEEL://鼠标滚轮消息

{

int x = LOWORD(lParam);

int y = HIWORD(lParam);

int det = message == WM\_MOUSEWHEEL ? (short)HIWORD(wParam) : 0;

handleMouseMessage(message, x, y, det);

return FALSE;

}

case WM\_SIZE://窗口大小变化消息

{

UINT cx = LOWORD(lParam);

UINT cy = HIWORD(lParam);

sized(cx, cy);

return TRUE;

}

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

}

#pragma endregion

1. **Raster.cpp**

#include "Raster.h"

#include <windows.h>

#include<iostream>

#include "GeoDefin3D.h"

using namespace std;

void LineBres(int x0, int y0, int x1, int y1)//画直线

{

int iTag = 0;

int dx, dy, tx, ty, inc1, inc2, d, curx, cury,t,t1,t2;

setPixel(x0, y0, BLUE);

if (x0 == x1 && y0 == y1)

{

return;

}

dy = abs(y1 - y0);

dx = abs(x1 - x0);

if (dx < dy)//斜率大于1

{

iTag = 1;

swap(x0, y0);//关于y=x作对称

swap(x1, y1);

swap(dx, dy);

}

tx = ((x1 - x0) > 0) ? 1 : -1;

ty = ((y1 - y0) > 0) ? 1 : -1;

curx = x0;

cury = y0;

inc1 = 2 \* dy;

inc2 = 2 \* (dy - dx);

d = inc1 - dx;

while (curx != x1)

{

curx += tx;

if (d < 0)

{

d += inc1;

}

else

{

cury += ty;

d += inc2;

}

if (iTag)

{

setPixel(cury, curx, BLUE);

}

else

{

setPixel(curx, cury, BLUE);

}

}

}

void Polygon(PixelPoint\* pts, int num)

{

int y\_max = 0;

//选出最大的顶点所对应的y值

for (int i = 0; i < num; i++)

{

if (pts[i].y > y\_max)

y\_max = pts[i].y;

}

//初始化AET表

AET\* pAET = new AET;

pAET->next = NULL;

//初始化NET表

NET\* pNET[1000];

for (int i = 0; i <= y\_max; i++)

{

pNET[i] = new NET;

pNET[i]->next = NULL;;

}

//建立NET表

for (int i = 0; i <= y\_max; i++)

{

for (int j = 0; j < num; j++)

{

if (i == pts[j].y)

{

//判断当前点的高低，使ymax、DX、DY的计算有变化

if (pts[(j - 1 + num) % num].y > pts[j].y) //前一个点y值>当前点y值

{

NET\* p = new NET;

p->x = pts[j].x;

p->ymax = pts[(j - 1 + num) % num].y;//与当前扫描线相交的活性边 的 最高点即为相邻顶点的y

double \_dx = pts[(j - 1 + num) % num].x - pts[j].x;

double \_dy = pts[(j - 1 + num) % num].y - pts[j].y;

p->dx = \_dx / \_dy; //x增量为斜率倒数

p->next = pNET[i]->next;

pNET[i]->next = p;

}

if (pts[(j + 1) % num].y > pts[j].y) //前一个点y值<当前点y值

{

NET\* p = new NET;

p->x = pts[j].x;

p->ymax = pts[(j + 1) % num].y;

double \_dx = pts[(j + 1) % num].x - pts[j].x;

double \_dy = pts[(j + 1) % num].y - pts[j].y;

p->dx = \_dx / \_dy;

p->next = pNET[i]->next;

pNET[i]->next = p;

}

}

}

}

for (int i = 0; i <= y\_max; i++)

{

//计算扫描线与多边形所有边的交点，并更新活性边表

NET\* p = pAET->next;

while (p)

{

p->x = p->x + p->dx; //更新x坐标

p = p->next;

}

//断表排序,不再开辟空间

//将所有交点按x坐标递增进行排序

AET\* tq = pAET;

p = pAET->next;

tq->next = NULL;

while (p)//顺着链表往下走

{

//找到第一个比它大的数字tq->next->next->x，则从p->next到tq->next都是比p->x小的

while (tq->next != NULL && tq->next->x <= p->x)

tq = tq->next;

//插入p到tq和tq->next之间

NET\* t = p->next;

p->next = tq->next;

tq->next = p;

p = t;

tq = pAET;//回到头

}

AET\* q = pAET;

p = q->next;

while (p)

{

if (p->ymax == i)

{

q->next = p->next;

delete p;

p = q->next;

}

else

{

q = q->next;

p = q->next;

}

}

//若NET中有新点，将其用插入法插入AET，按x递增的顺序排列

p = pNET[i]->next;

q = pAET;

while (p)

{

while (q->next && p->x >= q->next->x)

q = q->next;

//插入p

NET\* t = p->next;

p->next = q->next;

q->next = p;

p = t;

q = pAET;//回到头

}

//交点配对后区间填色

p = pAET->next;

while (p && p->next != NULL)

{

for (double j = p->x; j <= p->next->x; j++)

{

setPixel(j, i, GREEN);

}

p = p->next->next;//考虑端点情况

}

}

refreshWindow();

}

double backculling(Vec4& pt1, Vec4& pt2, Vec4& pt3)

{

double x1 = pt2[0] - pt1[0];

double y1 = pt2[1] - pt1[1];

double x2 = pt3[0] - pt1[0];

double y2 = pt3[1] - pt1[1];

double v = x1 \* y2 - y1 \* x2;

return v;

}

double computeA(Vec4& p1, Vec4& p2, Vec4& p3)

{

double A = p1.y() \* (p2.z() - p3.z()) + p2.y() \* (p3.z() - p1.z()) + p3.y() \* (p1.z() - p2.z());

return A;

}

double computeB(Vec4& p1, Vec4& p2, Vec4& p3)

{

double B = p1.z() \* (p2.x() - p3.x()) + p2.z() \* (p3.x() - p1.x()) + p3.z() \* (p1.x() - p2.x());

return B;

}

double computeC(Vec4& p1, Vec4& p2, Vec4& p3)

{

double C = p1.x() \* (p2.y() - p3.y()) + p2.x() \* (p3.y() - p1.y()) + p3.x() \* (p1.y() - p2.y());

return C;

}

double computeD(Vec4& p1, Vec4& p2, Vec4& p3)

{

double D = -p1.x() \* (p2.y() \* p3.z() - p3.y() \* p2.z()) - p2.x() \* (p3.y() \* p1.z() - p1.y() \* p3.z())

- p3.x() \* (p1.y() \* p2.z() - p2.y() \* p1.z());

return D;

}

void Z\_Buffer(Vertex3D pixelSet[])

{

double A, B, C, D;

A = computeA(pixelSet[0].pos, pixelSet[1].pos, pixelSet[2].pos);

B = computeB(pixelSet[0].pos, pixelSet[1].pos, pixelSet[2].pos);

C = computeC(pixelSet[0].pos, pixelSet[1].pos, pixelSet[2].pos);

D = computeD(pixelSet[0].pos, pixelSet[1].pos, pixelSet[2].pos);

//顶点光强

Vec4 I1 = pixelSet[0].attribute.color;

Vec4 I2 = pixelSet[1].attribute.color;

Vec4 I3 = pixelSet[2].attribute.color;

Vec4 I4, I5, Ip;

struct Attri

{

double x,z;

Vec4 I;

};

vector<Attri> interx(3);//初始化x扫描线交点容器

double MinValue = -1000000000;//深度缓存变量的最小值

double z, z4, z5, zp;

double x1, y1, z1, x2, y2, z2, x3, y3, z3;//顶点

double x4, x5, xa, ya, xb, yb, x;

x1 = pixelSet[0].pos.x(); y1 = pixelSet[0].pos.y(); z1 = pixelSet[0].pos.z();

x2 = pixelSet[1].pos.x(); y2 = pixelSet[1].pos.y(); z2 = pixelSet[1].pos.z();

x3 = pixelSet[2].pos.x(); y3 = pixelSet[2].pos.y(); z3 = pixelSet[2].pos.z();

//扫描线

int y\_max = 0;

//选出最大的顶点所对应的y值

for (int i = 0; i < 3; i++)

{

if (pixelSet[i].pos.y() > y\_max)

y\_max = pixelSet[i].pos.y();

}

for (int i = 0; i < y\_max; i++) //扫描线 y

{

interx.clear();//清空缓冲区

//interx1.clear();

for (int j = 0; j < 3; j++) //扫描线 x

{

xa = pixelSet[j].pos.x();

ya = pixelSet[j].pos.y();//当前点

double za= pixelSet[j].pos.z();

xb = pixelSet[(j + 1) % 3].pos.x();

yb = pixelSet[(j + 1) % 3].pos.y();//前一个点

double zb= pixelSet[(j + 1) % 3].pos.z();

Vec4 Ia = pixelSet[j].attribute.color;

Vec4 Ib = pixelSet[(j + 1) % 3].attribute.color;

int y\_lower, y\_higher;

y\_lower = (ya < yb) ? ya : yb;//比较前一个点y值与当前点y值

y\_higher = (ya > yb) ? ya : yb;

//求交点

if (i < y\_lower || i >= y\_higher) continue; //无交点——>y超出范围

//x = xa + (i - ya) \* (xb - xa) / (yb - ya);//计算交点处x值

interx.resize(interx.size()+1);//存交点x值

interx.back().x= xa + (i - ya) \* (xb - xa) / (yb - ya);

interx.back().z = za + (i - ya) \* (zb - za) / (yb - ya);

interx.back().I = Ia + (Ib - Ia)\* (i - ya) / (yb - ya);

}

if (interx.size() == 0)

continue;

int s = interx.size();//交点个数

double \_x;

double z;

int a = 0, b = 1;

if (interx[0].x > interx[1].x)

a = 1, b = 0;

Ip = interx[b].I;

zp = interx[b].z;

Vec4 DI = (interx[b].I - interx[a].I) / (interx[b].x - interx[a].x);

double DZ = (interx[b].z - interx[a].z) / (interx[b].x - interx[a].x);

for (\_x = interx[a].x; \_x < interx[b].x; \_x++)

{

setPixel(\_x, i, zp, \_RGB(Ip[0] \* 255, Ip[1] \* 255, Ip[2] \* 255));

Ip += DI;

zp += DZ;

}

}

}

1. **Render2D.cpp**

#include "Render2D.h"

#include "GeoDefine.h"

#include "Graphic.h"

#include "Vec3.h"

#include "resource.h"

#include "GeometryFactory.h"

#include "DialogHelper.h"

#include "LayerImporter.h"

#include "Raster.h"

//void init(double dx, double dy, T pLayer->envelop.width(), T pLayer->envelop.height(), int getWindowWidth(), int getWindowHeight());

void Render2D::init(double dx, double dy, double w, double h, int WWidth, int WHeight) {

double s = std::min( WWidth / w , WHeight / h );

w = WWidth / s;

h = WHeight / s;

m.makeTranslate(w / 2 - dx, h / 2 - dy);

m \*= Matrix3::scale(s ,s );

}

void Render2D::setWindow(double left, double right, double bottom, double top)

{

}

void Render2D::setViewport(int x, int y, int width, int height)

{

}

void Render2D::translate(double x, double y) {

m \*= Matrix3::translate(x, y);

}

void Render2D::rotate(double angle)

{

m \*= Matrix3::rotate(angle);

}

void Render2D::rotate(double x, double y, double angle)

{

m \*= Matrix3::translate(-x, -y);

rotate(angle);

m \*= Matrix3::translate(x, y);

}

void Render2D::scale(double sx, double sy)

{

m \*= Matrix3::scale(sx, sy);

}

void Render2D::scale(int x, int y, double d) {

m \*= Matrix3::translate(-x, -y);

double sx, sy;

sx = sy = d; //d 比例系数

scale(sx, sy);

m \*= Matrix3::translate(x, y);

}//缩放

void Render2D::render(Dataset& g\_Dataset) {

for (int i = g\_Dataset.getLayerCount() - 1; i >= 0; --i)

{

render(g\_Dataset[i]);

}

}

void Render2D::render(Layer\* pLayer)

{

setPenColor(pLayer->layerColor);

for (int i = 0, size = pLayer->getGeometryCount(); i < size; ++i)

{

render( (\*pLayer)[i]);

}

}

void Render2D::render(Geometry\* pGeometryDef)

{

switch (pGeometryDef->getGeomType())//self注释：识别图形对象类型

{

case gtPolyline:

{

PolylineGeometry\* pGeometry = (PolylineGeometry\*)pGeometryDef;//转化为实际图形类型

const vector<Point2D>& pts = pGeometry->getPts();//获得图形顶点集合

for (int i = 0, ptsCount = pts.size(); i < ptsCount - 1; ++i)

{

Vec3 p1(pts[i].x, pts[i].y);

Vec3 p2(pts[(i + 1)].x, pts[(i + 1)].y);

p1 \*= m;

p2 \*= m;

//drawLine( p1.x(), p1.y(), p2.x(), p2.y());

LineBres(p1.x(), p1.y(), p2.x(), p2.y());

}

}

break;

case gtPolygon:

{

PolygonGeometry\* pGeometry = (PolygonGeometry\*)pGeometryDef;//转化为实际图形类型

const vector<Point2D>& pts = pGeometry->getPts();//获得图形顶点集合

size\_t ptsCount = pts.size();

vector <PixelPoint> \_pts(ptsCount);//drawPolygon函数参数是PixelPoint数组，\_pts用于接收Point2D转化的PixelPoint数组

for (int i = 0; i < ptsCount; ++i)

{

Vec3 p(pts[i].x, pts[i].y);

p \*= m;

\_pts[i].x = p.x();

\_pts[i].y = p.y();

}

//drawPolygon(\_pts.data(), ptsCount);

Polygon(\_pts.data(), ptsCount);

//Rasterpolygon(\_pts.data(), ptsCount);

}

break;

}

}

void Render2D::C\_symmetry(int x, int y)

{

m \*= Matrix3::C\_symmetry(x, y);

}

1. **Render3D.cpp**

#include "Render3D.h"

#include "GeoDefin3D.h"

#include "GeoDefine.h"

#include "Graphic.h"

#include "Vec4.h"

#include "resource.h"

#include "GeometryFactory.h"

#include "DialogHelper.h"

#include "LayerImporter.h"

void Render3D::setWindow(double left, double back, double right, double bottom, double front, double top)

{

}

void Render3D::setViewport(int x, int y, int z, int width, int length, int height)

{

}

void Render3D::translate(double x, double y, double z)

{

m \*= Matrix4::translate(x, y, z);

}

void Render3D::rotate\_x(double angle)

{

m \*= Matrix4::rotate\_x(angle);

}

void Render3D::rotate\_y(double angle)

{

m \*= Matrix4::rotate\_y(angle);

}

void Render3D::rotate\_z(double angle)

{

m \*= Matrix4::rotate\_z(angle);

}

void Render3D::scale(double sx, double sy, double sz)

{

m \*= Matrix4::scale(sx, sy, sz);

}

void Render3D::scale(int x, int y, int z, double d)

{

m \*= Matrix4::translate(-x, -y, -z);

double sx, sy, sz;

sx = sy = sz= d; //d 比例系数

scale(sx, sy, sz);

m \*= Matrix4::translate(x, y, z);

}

void Render3D::C\_symmetry(int x, int y, int z)

{

m \*= Matrix4::C\_symmetry(x, y, z);

}