实验四 投影与消隐

1. **球体的投影效果：完成球体模型的透视投影曲线**
   (1) 构造双三次贝塞尔球面/球体模型（实验三的内容）
   (2) 对球体模型进行透视投影（如图 1）

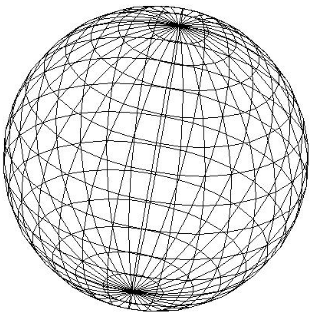

图 1 球体模型的透视投影

【实验过程及编码】

（一）实验步骤

（1）绘制线框球（实验三已完成内容）

（2）设计投影类 CProjection

（3）将实验三中的正交投影改为透视投影

（二）实验编码

（1）绘制线框球（实验三已完成内容）

在实验三中，线框球的绘制采用了正交投影。

（2）设计投影类 CProjection

➢ CProjection.h

```
class CProjection
{
public:
	CProjection(void);
	virtual ~CProjection(void);
	void SetEye(double R);//设置视点
	CP3 GetEye(void);//读取视点
	CP2 ObliqueProjection(CP3 WorldPoint);//斜投影
	CP2 OrthogonalProjection(CP3 WorldPoint);//正交投影
	CP2 PerspectiveProjection(CP3 WorldPoint);//透视投影
private:
	CP3 EyePoint;//视点
	double R, d;//视径和视距
};
```

➢ CProjection.cpp

```
CProjection::CProjection(void)
{
	R = 1200, d = 800;
```

```
    EyePoint.x = 0, EyePoint.y = 0, EyePoint.z = R;//视点位于屏幕正前方
}
CProjection::~CProjection(void)
{
}
void CProjection::SetEye(double R)//设置视径
{
    EyePoint.z = R;
}
CP3 CProjection::GetEye(void)//读取视点
{
    return EyePoint;
}
CP2 CProjection::ObliqueProjection(CP3 WorldPoint)//斜二测投影
{
    CP2 ScreenPoint;//屏幕坐标系二维点
   ScreenPoint.x = WorldPoint.x - 0.3536 * WorldPoint.z;
    ScreenPoint.y = WorldPoint.y - 0.3536 * WorldPoint.z;
    return ScreenPoint;
}
CP2 CProjection::OrthogonalProjection(CP3 WorldPoint)//正交投影
{
    CP2 ScreenPoint;//屏幕坐标系二维点
    ScreenPoint.x = WorldPoint.x;
    ScreenPoint.y = WorldPoint.y;
    return ScreenPoint;
}
CP2 CProjection::PerspectiveProjection(CP3 WorldPoint)
{
    CP3 ViewPoint;//观察坐标系三维点
    ViewPoint.x = WorldPoint.x;
    ViewPoint.y = WorldPoint.y;
   ViewPoint.z = EyePoint.z - WorldPoint.z;
    CP2 ScreenPoint;//屏幕坐标系二维点
    ScreenPoint.x = d * ViewPoint.x / ViewPoint.z;
    ScreenPoint.y = d * ViewPoint.y / ViewPoint.z;
    return ScreenPoint;
}
```

（3）将实验三中的正交投影改为透视投影

- CBezierPatch.h

在类声明中加入一个数据成员：

```
    CProjection projection;//投影
```

- CBezierPatch.cpp

<1> 绘制四边形网格的 **DrawFacet()** 函数中，将正交投影改为透视投影：

```
for(int nPoint = 0; nPoint < 4; nPoint++)
    //ScreenPoint[nPoint] = quadrP[nPoint];//正交投影
    ScreenPoint[nPoint] = projection.PerspectiveProjection(quadrP[nPoint]);//透视投影
```

<2> 绘制控制网格的 **DrawControlGrid()**函数中，将正交投影改为透视投影：

```
for(int i = 0; i < 4; i++)
    for(int j = 0; j < 4; j++)
        //P2[i][j] = CtrPt[i][j];//正交投影
        P2[i][j] = projection.PerspectiveProjection(CtrPt[i][j]);//透视投影
```

- CTestView.cpp

若绘制一个具有倾斜角度（X 轴旋转 Alpha 角度、Y 轴旋转 Beta 角度）的静态透视投影线框球，在类 CTestView 类的构造函数 **CTestView()**中，加入如下语句：

```
Alpha = 60;  //X 轴旋转角度
tranUp.RotateX(Alpha);
tranDown.RotateX(Alpha);
Beta = 30;  //Y 轴旋转角度
tranUp.RotateY(Beta);
tranDown.RotateY(Beta);
```

2. **球体的消隐效果：使用背面剔除算法绘制球体的可见表面**

视点位于屏幕正前方，绘制球体的透视投影图（本次实验第一项内容），然后使用背面剔除算法绘制球体的可见表面，得到球体模型的消隐效果。

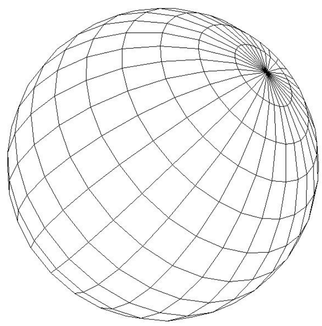

图 2 球体模型的消隐效果图

【实验过程及编码】

（一）实验步骤

（1）绘制具有透视投影效果的线框球（实验四第一项内容）

（2）设计三维向量类 CVector3

（3）绘制球体的可见网络

（三）实验编码

（1）绘制具有透视投影效果的线框球

在实验四的第一项实验内容中，已经完成了线框球的透视投影绘制。

（2）设计三维向量类 CVector3

- CVector3.h

```
class CVector3
{
public:
	CVector3(void);
	virtual ~CVector3(void);
	CVector3(double x, double y, double z);//绝对向量
	CVector3(const CP3 &p);
	CVector3(const CP3 &p0, const CP3 &p1);//相对向量
	double Magnitude(void);//计算向量的模
	CVector3 Normalize(void);//归一化向量
	friend CVector3 operator + (const CVector3 &v0, const CVector3 &v1);//运算符重载
	friend CVector3 operator - (const CVector3 &v0, const CVector3 &v1);
	friend CVector3 operator * (const CVector3 &v, double scalar);
	friend CVector3 operator * (double scalar, const CVector3 &v);
	friend CVector3 operator / (const CVector3 &v, double scalar);
	friend double DotProduct(const CVector3 &v0, const CVector3 &v1);//计算向量的点积
	friend CVector3 CrossProduct(const CVector3 &v0, const CVector3 &v1);//计算向量的叉积
private:
	double x,y,z;
};
```

- CVector3.cpp

```
CVector3::CVector3(void)
{
	x = 0.0,y = 0.0, z = 1.0;//指向 z 轴正向
}
CVector3::~CVector3(void)
{
}
CVector3::CVector3(double x, double y, double z)//绝对向量
{
	this->x = x;
	this->y = y;
	this->z = z;
}
CVector3::CVector3(const CP3 &p)
{
	x = p.x;
	y = p.y;
	z = p.z;
}
CVector3::CVector3(const CP3 &p0, const CP3 &p1)//相对向量
{
	x = p1.x - p0.x;
	y = p1.y - p0.y;
```

```
    z = p1.z - p0.z;
}
double CVector3::Magnitude(void)//向量的模
{
    return sqrt(x * x + y * y + z * z);
}
CVector3 CVector3::Normalize(void)//归一化为单位向量
{
    CVector3 vector;
    double magnitude = sqrt(x * x + y * y + z * z);
    if(fabs(magnitude) < 1e-4)
        magnitude  = 1.0;
    vector.x = x / magnitude;
    vector.y = y / magnitude;
    vector.z = z / magnitude;
    return vector;
}
CVector3 operator + (const CVector3 &v0, const CVector3 &v1)//向量的和
{
    CVector3 vector;
    vector.x = v0.x + v1.x;
    vector.y = v0.y + v1.y;
    vector.z = v0.z + v1.z;
    return vector;
}
CVector3 operator - (const CVector3 &v0, const CVector3 &v1)//向量的差
{
    CVector3 vector;
    vector.x = v0.x - v1.x;
    vector.y = v0.y - v1.y;
    vector.z = v0.z - v1.z;
    return vector;
}
CVector3 operator * (const CVector3 &v, double scalar)//向量与常量的积
{
    CVector3 vector;
    vector.x = v.x * scalar;
    vector.y = v.y * scalar;
    vector.z = v.z * scalar;
    return vector;
}
CVector3 operator * (double scalar, const CVector3 &v)//常量与向量的积
{
    CVector3 vector;
```

```
    vector.x = v.x * scalar;
    vector.y = v.y * scalar;
    vector.z = v.z * scalar;
    return vector;
}
CVector3 operator / (const CVector3 &v, double scalar)//向量数除
{
    if(fabs(scalar) < 1e-4)
        scalar = 1.0;
    CVector3 vector;
    vector.x = v.x / scalar;
    vector.y = v.y / scalar;
    vector.z = v.z / scalar;
    return vector;
}
double DotProduct(const CVector3 &v0, const CVector3 &v1)//向量的点积
{
    return(v0.x * v1.x + v0.y * v1.y + v0.z * v1.z);
}
CVector3 CrossProduct(const CVector3 &v0, const CVector3 &v1)//向量的叉积
{
    CVector3 vector;
    vector.x = v0.y * v1.z - v0.z * v1.y;
    vector.y = v0.z * v1.x - v0.x * v1.z;
    vector.z = v0.x * v1.y - v0.y * v1.x;
    return vector;
}
```

（3）绘制球体的可见网络

➢ CBezierPatch.cpp

在绘制四边形网格的 **DrawFacet()** 函数中，加入背面剔除算法，只绘制可见网格：

```
void CBezierPatch::DrawFacet(CDC* pDC)
{
    CP2 ScreenPoint[4];//二维投影点
    CP3 ViewPoint = projection.GetEye();//视点
    CVector3 ViewVector(quadrP[0], ViewPoint);// 面的视向量
    ViewVector = ViewVector.Normalize();//归一化视向量
    CVector3 Vector01(quadrP[0], quadrP[1]);//边向量
    CVector3 Vector02(quadrP[0], quadrP[2]);
    CVector3 Vector03(quadrP[0], quadrP[3]);
    CVector3 FacetNormalA = CrossProduct(Vector01, Vector02);//面法向量
    CVector3 FacetNormalB = CrossProduct(Vector02, Vector03);//面法向量
    CVector3 FacetNormal = (FacetNormalA + FacetNormalB);//面法向量
    FacetNormal = FacetNormal.Normalize();
    CPen pen(PS_SOLID, 1, RGB(0, 0, 0));
```

```
    CPen* pOldPen = pDC->SelectObject(&pen);
    if(DotProduct(ViewVector, FacetNormal) >= 0)//背面剔除算法
    {
        for(int nPoint = 0; nPoint < 4; nPoint++)
          ScreenPoint[nPoint] = projection.PerspectiveProjection(quadrP[nPoint]);//透视投影
        pDC->MoveTo(ROUND(ScreenPoint[0].x), ROUND(ScreenPoint[0].y));
        pDC->LineTo(ROUND(ScreenPoint[1].x), ROUND(ScreenPoint[1].y));
        pDC->LineTo(ROUND(ScreenPoint[2].x), ROUND(ScreenPoint[2].y));
        pDC->LineTo(ROUND(ScreenPoint[3].x), ROUND(ScreenPoint[3].y));
        pDC->LineTo(ROUND(ScreenPoint[0].x), ROUND(ScreenPoint[0].y));
    }
    pDC->SelectObject(pOldPen);
}
```