实验四 光照模型与纹理映射

1. 球体的纹理映射：完成球体模型的凹凸纹理映射效果

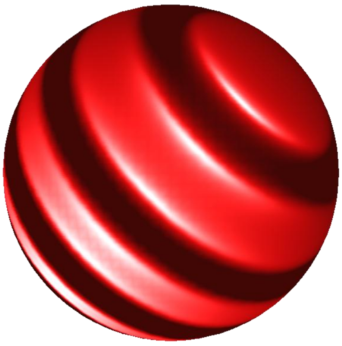

图 1 球体的正弦凹凸纹理映射

【实验过程及编码】

（一）实验步骤

（1）绘制具有消隐效果的线框球（实验四已完成内容）

（2）设计颜色类

（3）设计简单光照模型中的光源类

（4）设计简单光照模型中的材质类

（5）设计简单光照模型中的光照类

（6）设计 CZBuffer 类

（7）使用正弦函数扰动四边形网格顶点法向量

（8）初始化光照环境，绘制四边形小面

（二）实验编码

（1）绘制具有消隐效果的线框球

在实验四的实验内容中，已经完成了具有消隐效果的线框球的绘制。

（2）设计颜色类 CRGB

- CRGB.h

```
class CRGB
{
public:
    CRGB(void);
    CRGB(double red, double green, double blue, double alpha = 0.0);
    virtual ~CRGB(void);
    friend CRGB operator + (const CRGB &c0, const CRGB &c1);//运算符重载
    friend CRGB operator - (const CRGB &c0, const CRGB &c1);
    friend CRGB operator * (const CRGB &c0, const CRGB &c1);
    friend CRGB operator * (const CRGB &c, double scalar);
    friend CRGB operator * (double scalar, const CRGB &c);
    friend CRGB operator / (const CRGB &c, double scalar);
    friend CRGB operator += (CRGB &c1, CRGB &c2);
    friend CRGB operator -= (CRGB &c1, CRGB &c2);
    friend CRGB operator *= (CRGB &c1, CRGB &c2);
    friend CRGB operator /= (CRGB &c1, double scalar);
    void Normalize(void);//归一化到[0,1]区间
public:
    double red;//红色分量
    double green;//绿色分量
    double blue;//蓝色分量
    double alpha;//alpha 分量
};
```

- ➢ CRGB.cpp

```
CRGB::CRGB(void)
{
    red = 1.0;
    green = 1.0;
    blue = 1.0;
    alpha = 0.0;
}
CRGB::CRGB(double red, double green, double blue, double alpha)//重载构造函数
{
    this->red = red;
    this->green = green;
    this->blue = blue;
    this->alpha = alpha;
}
CRGB::~CRGB(void)
{
}
CRGB operator + (const CRGB &c0, const CRGB &c1)//"+"运算符重载
{
    CRGB color;
    color.red = c0.red + c1.red;
    color.green = c0.green + c1.green;
    color.blue = c0.blue + c1.blue;
    return color;
}
CRGB operator - (const CRGB &c0, const CRGB &c1)//"-"运算符重载
{
    CRGB color;
    color.red = c0.red - c1.red;
    color.green = c0.green - c1.green;
    color.blue = c0.blue - c1.blue;
    return color;
}
CRGB operator * (const CRGB &c0, const CRGB &c1)//"*"运算符重载
{
    CRGB color;
    color.red = c0.red * c1.red;
    color.green = c0.green * c1.green;
    color.blue = c0.blue * c1.blue;
    return color;
}
CRGB operator * (const CRGB &c, double scalar)//"*"运算符重载
{
    CRGB color;
    color.red = scalar * c.red;
    color.green = scalar * c.green;
    color.blue = scalar * c.blue;
    return color;
}
CRGB operator * (double scalar, const CRGB &c)//"*"运算符重载
{
    CRGB color;
    color.red = scalar * c.red;
    color.green = scalar * c.green;
    color.blue = scalar * c.blue;
    return color;
}
CRGB operator / (const CRGB &c, double scalar)//"/"运算符重载
{
    CRGB color;
    color.red = c.red / scalar;
    color.green = c.green / scalar;
    color.blue = c.blue / scalar;
    return color;
}
CRGB operator += (CRGB &c1, CRGB &c2)//"+="运算符重载
{
    c1.red += c2.red;
    c1.green += c2.green;
    c1.blue += c2.blue;
    return c1;
}
CRGB operator -= (CRGB &c1, CRGB &c2)//"-="运算符重载
{
    c1.red -= c2.red;
    c1.green -= c2.green;
    c1.blue -= c2.blue;
    return c1;
}
```

```
CRGB operator *= (CRGB &c1, CRGB &c2)//"*="运算符重载
{
    c1.red *= c2.red;
    c1.green *= c2.green;
    c1.blue *= c2.blue;
    return c1;
}
CRGB operator /= (CRGB &c1, double scalar)//"/="运算符重载
{
    c1.red /= scalar;
    c1.green /= scalar;
    c1.blue /= scalar;
    return c1;
}
void CRGB::Normalize(void)//归一化处理
{
    red = (red < 0.0) ? 0.0 : ((red > 1.0) ? 1.0 : red);
    green = (green < 0.0) ? 0.0 : ((green > 1.0) ? 1.0 : green);
    blue = (blue < 0.0) ? 0.0 : ((blue > 1.0) ? 1.0 : blue);
}
```

（3）设计光源类 CLightSource

- CLightSource.h

```
#include"CRGB.h"
#include"P3.h"
class CLightSource
{
public:
    CLightSource(void);
    virtual ~CLightSource(void);
    void SetDiffuse(CRGB diffuse);//设置光源的漫反射光
    void SetSpecular(CRGB specular);//设置光源的镜面反射光
    void SetPosition(double x, double y, double z);//设置光源的位置
    void SetAttenuationFactor(double c0, double c1, double c2);//设置光强的衰减因子
    void SetOnOff(BOOL onoff);//设置光源开关状态
public:
    CRGB L_Diffuse;//漫反射光颜色
    CRGB L_Specular;//镜面反射光颜色
    CP3 L_Position;//光源位置
    double L_C0;//常数衰减因子
    double L_C1;//线性衰减因子
    double L_C2;//二次衰减因子
    BOOL L_OnOff;//光源开启或关闭
};
```

- CLightSource.cpp

```
CLightSource::CLightSource(void)
{
    L_Diffuse = CRGB(0.0, 0.0, 0.0);//光源的漫反射颜色
    L_Specular = CRGB(1.0, 1.0, 1.0);//光源镜面高光颜色
    L_Position.x = 0.0, L_Position.y = 0.0, L_Position.z = 1000.0;//光源位置直角坐标
    L_C0 = 1.0;//常数衰减系数
    L_C1 = 0.0;//线性衰减系数
    L_C2 = 0.0;//二次衰减系数
    L_OnOff = TRUE;//光源开启
}
CLightSource::~CLightSource(void)
{
}
void CLightSource::SetDiffuse(CRGB difuse)
{
    L_Diffuse = difuse;
}
void CLightSource::SetSpecular(CRGB specular)
{
    L_Specular = specular;
}
void CLightSource::SetPosition(double x, double y, double z)
{
    L_Position.x = x;
    L_Position.y = y;
    L_Position.z = z;
}
void CLightSource::SetOnOff(BOOL onoff)
{
    L_OnOff = onoff;
}
void CLightSource::SetAttenuationFactor(double c0, double c1, double c2)
{
    L_C0 = c0;
    L_C1 = c1;
    L_C2 = c2;
}
```

（4）设计材质类 CMaterial

➢ CMaterial.h

```
class CMaterial
{
public:
    CMaterial(void);
    virtual~CMaterial(void);
    void SetAmbient(CRGB c);//设置环境光的反射率
    void SetDiffuse(CRGB c);//设置漫反射光的反射率
    void SetSpecular(CRGB c);//设置镜面反射光的反射率
    void SetEmission(CRGB c);//设置自身辐射的颜色
    void SetExponent(double n);//设置高光指数
public:
    CRGB M_Ambient;//环境光的反射率
    CRGB M_Diffuse;//漫反射光的反射率
    CRGB M_Specular;//镜面反射光的反射率
    CRGB M_Emission;//自身辐射的颜色
    double M_n;//高光指数
};
```

- CMaterial.cpp

```
CMaterial::CMaterial(void)
{
    M_Ambient = CRGB(0.2, 0.2, 0.2);//材质的环境反射率
    M_Diffuse = CRGB(0.8, 0.8, 0.8);//材质的漫反射率
    M_Specular = CRGB(0.0, 0.0, 0.0);//材质的镜面反射率
    M_Emission = CRGB(0.0, 0.0, 0.0, 1.0);//材质的辐射光
    M_n = 1.0;//高光指数
}
CMaterial::~CMaterial(void)
{
}
void CMaterial::SetAmbient(CRGB c)
{
    M_Ambient = c;
}
void CMaterial::SetDiffuse(CRGB c)
{
    M_Diffuse = c;
}
void CMaterial::SetSpecular(CRGB c)
{
    M_Specular = c;
}
void CMaterial::SetEmission(CRGB c)
{
    M_Emission = c;
}
void CMaterial::SetExponent(double n)
{
    M_n = n;
}
```

（5）设计光照类 CLighting

- CLighting.h

```
#include"CMaterial.h"
#include"CVector3.h"
#include"CLightSource.h"
class CLighting
{
public:
    CLighting(void);
    CLighting(int nLightNumber);
    virtual ~CLighting(void);
    void SetLightNumber(int nLightNumber);//设置光源数量
    CRGB Illuminate(CP3 ViewPoint, CP3 Point, CVector3 ptNormal, CMaterial* pMaterial);//计算光照
public:
    int nLightNumber;//光源数量
    CLightSource* LightSource;//光源数组
    CRGB Ambient;//环境光
};
```

- CLighting.cpp

```
CLighting::CLighting(void)
{
    nLightNumber = 1;
    LightSource = new CLightSource[nLightNumber];
    Ambient = CRGB(0.3, 0.3, 0.3);//环境光是常数
}
CLighting::CLighting(int nLightNumber)
{
    this->nLightNumber = nLightNumber;
    LightSource = new CLightSource[nLightNumber];
    Ambient = CRGB(0.3, 0.3, 0.3);
}
CLighting::~CLighting(void)
{
    if (LightSource)
    {
        delete[]LightSource;
        LightSource = NULL;
    }
}
void CLighting::SetLightNumber(int nLightNumber)
{
    if (LightSource)
        delete[]LightSource;
    this->nLightNumber = nLightNumber;
    LightSource = new CLightSource[nLightNumber];
}
CRGB CLighting::Illuminate(CP3 ViewPoint, CP3 Point, CVector3 ptNormal, CMaterial* pMaterial)
{
    CRGB ResultI = pMaterial->M_Emission;//材质自身发光为初始值
    for (int loop = 0; loop < nLightNumber; loop++)//检查光源开关状态
    {
        if (LightSource[loop].L_OnOff)//光源开
        {
            CRGB I = CRGB(0.0, 0.0, 0.0);// I代表“反射”光强
            CVector3 L(Point, LightSource[loop].L_Position);// L为光向量
            double d = L.Magnitude();// d为光传播的距离
            L = L.Normalize();//归一化光向量
            CVector3 N = ptNormal;
            N = N.Normalize();//归一化法向量
            //第1步，加入漫反射光
            double NdotL = max(DotProduct(N, L), 0);
            I += LightSource[loop].L_Diffuse * pMaterial->M_Diffuse * NdotL;
            //第2步，加入镜面反射光
            CVector3 V(Point, ViewPoint);//V为观察向量
            V = V.Normalize();//归一化观察向量
            CVector3 H = (L + V) / (L + V).Magnitude();//H为中值向量
            double NdotH = max(DotProduct(N, H), 0);
            double Rs = pow(NdotH, pMaterial->M_n);
            I += LightSource[loop].L_Specular * pMaterial->M_Specular * Rs;
            //第3步，光强衰减
            double c0 = LightSource[loop].L_C0;//c0为常数衰减因子
            double c1 = LightSource[loop].L_C1;//c1为线性衰减因子
            double c2 = LightSource[loop].L_C2;//c2为二次衰减因子
            double f = (1.0 / (c0 + c1 * d + c2 * d * d));//光强衰减函数
            f = min(1.0, f);
            ResultI += I * f;
        }
        else
            ResultI += Point.c;//物体自身颜色
    }
    //第4步，加入环境光
    ResultI += Ambient * pMaterial->M_Ambient;
    //第5步，光强归一化到[0,1]区间
    ResultI.Normalize();
    //第6步，返回所计算顶点的最终的光强颜色
    return ResultI;
}
```

（6）设计深度缓冲器类 CZBuffer

① CPoint2 类

- CPoint2.h

```
#include"CRGB.h"
#include"CVector3.h"
class CPoint2
{
public:
   CPoint2(void);
   CPoint2(int x, int y);
   CPoint2(int x, int y, CRGB c);
   CPoint2(int x, int y, CVector3 n);
   virtual ~CPoint2(void);
   friend CPoint2 operator + (const CPoint2& pt0, const CPoint2& pt1);//运算符重载
   friend CPoint2 operator - (const CPoint2& pt0, const CPoint2& pt1);
   friend CPoint2 operator * (int scalar, const CPoint2& pt);
public:
   int x, y;//坐标
   CRGB c;//颜色
   CVector3 n;//法向量
};
```

- CPoint2.cpp

```
#include "CPoint2.h"
CPoint2::CPoint2(void)
{
      x = 0;
      y = 0;
      c = CRGB(0, 0, 0);
}
CPoint2::CPoint2(int x, int y)
{
      this->x = x;
      this->y = y;
      c = CRGB(0, 0, 0);
}
CPoint2::CPoint2(int x, int y, CRGB c)
{
      this->x = x;
      this->y = y;
      this->c = c;
}
CPoint2::CPoint2(int x, int y, CVector3 n)
{
      this->x = x;
```

```
    this->y = y;
    this->n = n;
}
CPoint2::~CPoint2(void)
{
}
CPoint2 operator + (const CPoint2 &pt0, const CPoint2 &pt1)//和
{
    CPoint2 point;
    point.x = pt0.x + pt1.x;
    point.y = pt0.y + pt1.y;
    return point;
}
CPoint2 operator - (const CPoint2 &pt0, const CPoint2 &pt1)//差
{
    CPoint2 point;
    point.x = pt0.x - pt1.x;
    point.y = pt0.y - pt1.y;
    return point;
}
CPoint2 operator * (const CPoint2 &pt, int scalar)//点和常量的积
{
    return CPoint2(pt.x * scalar, pt.y * scalar);
}
CPoint2 operator * (int scalar, const CPoint2 &pt)//点和常量的积
{
    return CPoint2(pt.x * scalar, pt.y * scalar);
}
CPoint2 operator / (const CPoint2 &pt, double scalar)//数除
{
    if (fabs(scalar) < 1e-4)
        scalar = 1.0;
    CPoint2 point;
    point.x = int(pt.x / scalar);
    point.y = int(pt.y / scalar);
    return point;
}
```

② CPoint3 类

➢ CPoint3.h

```
#include"CPoint2.h"
class CPoint3 : public CPoint2
{
    public:
    CPoint3(void);
```

```
        CPoint3(int x, int y, double z);
        virtual ~CPoint3(void);
        public:
        double z;
    };
```

- ➢ CPoint3.cpp

```
    CPoint3::CPoint3(void)
    {
    }
    CPoint3::CPoint3(int x, int y, double z) :CPoint2(x, y)
    {
        this->z = z;
    }
    CPoint3::~CPoint3(void)
    {
    }
```

③ CZBuffer 类

- ➢ CZBuffer.h

```
    #include "CPoint3.h"
    #include "CLighting.h"//Blinn-Phong 模型
    class CZBuffer
    {
    public:
        CZBuffer(void);
        virtual ~CZBuffer(void);
        void InitialDepthBuffer(int nWidth, int nHeight, double zDepth);//初始化深度缓冲区
        void SetPoint(CP3 P0, CP3 P1, CP3 P2, CVector3 N0, CVector3 N1, CVector3 N2);// 三角形初始化
        void PhongShader(CDC* pDC, CP3 ViewPoint, CLighting* pLight, CMaterial* pMaterial);//光滑着色
    private:
        void SortVertex(void);//顶点排序
        void EdgeFlag(CPoint2 PStart, CPoint2 PEnd, BOOL bFeature);//边标记
        CVector3 LinearInterp(double t, double coorStart, double coorEnd, CVector3 normalStart, CVector3 normalEnd);//向量线性插值
    protected:
        CP3 P0, P1, P2;//三角形的浮点数顶点
        CPoint3 point0, point1, point2;//三角形的整数顶点坐标
        CPoint2* SpanLeft; //跨度的起点数组标志
        CPoint2* SpanRight;//跨度的终点数组标志
        int nIndex;//记录扫描线条数
        double** zBuffer;//深度缓冲区
        int nWidth, nHeight;//缓冲区宽度与高度
    };
```

➢ CZBuffer.cpp

```
#define ROUND(d) int(d + 0.5)//四舍五入宏定义
CZBuffer::CZBuffer(void)
{
zBuffer = NULL;
}
CZBuffer::~CZBuffer(void)
{
for (int i = 0; i < nWidth; i++)
{
    delete[] zBuffer[i];
    zBuffer[i] = NULL;
}
if (zBuffer != NULL)
{
    delete zBuffer;
    zBuffer = NULL;
}
}
void CZBuffer::SetPoint(CP3 P0, CP3 P1, CP3 P2, CVector3 N0, CVector3 N1, CVector3 N2)
{
this->P0 = P0, this->P1 = P1, this->P2 = P2;
point0.x = ROUND(P0.x);
point0.y = ROUND(P0.y);
point0.z = P0.z;
point0.c = P0.c;
point0.n = N0;
point1.x = ROUND(P1.x);
point1.y = ROUND(P1.y);
point1.z = P1.z;
point1.c = P1.c;
point1.n = N1;
point2.x = ROUND(P2.x);
point2.y = ROUND(P2.y);
point2.z = P2.z;
point2.c = P2.c;
point2.n = N2;
}
void CZBuffer::PhongShader(CDC* pDC, CP3 ViewPoint, CLighting* pLight, CMaterial*
pMaterial)
{
double   CurrentDepth = 0.0;//当前扫描线的深度
CVector3 Vector01(P0, P1), Vector02(P0, P2);
CVector3 fNormal = CrossProduct(Vector01, Vector02);
```

```
double A = fNormal.x, B = fNormal.y, C = fNormal.z;//平面方程 Ax+By+Cz+D=0 的系数
double D = -A * P0.x - B * P0.y - C * P0.z;//当前扫描线随着 x 增长的深度步长
if (fabs(C) < 1e-4)
    C = 1.0;
double DepthStep = -A / C;//计算扫描线深度步长增量
SortVertex();
//定义三角形覆盖的扫描线条数
int nTotalLine = point1.y - point0.y + 1;
//定义 span 的起点与终点数组
SpanLeft = new CPoint2[nTotalLine];
SpanRight = new CPoint2[nTotalLine];
//判断三角形与 P0P1 边的位置关系，0-1-2 为右手系
int nDeltz = (point1.x - point0.x) * (point2.y - point0.y) - (point1.y - point0.y) * (point2.x - point0.x);//面法向量的 z 分量
if (nDeltz > 0)//三角形位于 P0P1 边的左侧
{
    nIndex = 0;
    EdgeFlag(point0, point2, TRUE);
    EdgeFlag(point2, point1, TRUE);
    nIndex = 0;
    EdgeFlag(point0, point1, FALSE);
}
else//三角形位于 P0P1 边的右侧
{
    nIndex = 0;
    EdgeFlag(point0, point1, TRUE);
    nIndex = 0;
    EdgeFlag(point0, point2, FALSE);
    EdgeFlag(point2, point1, FALSE);
}
for (int y = point0.y; y < point1.y; y++)//下闭上开
{
    int n = y - point0.y;
    for (int x = SpanLeft[n].x; x < SpanRight[n].x; x++)//左闭右开
    {
        CurrentDepth = -(A * x + B * y + D) / C;//z=-(Ax+By+D)/C
        CVector3 ptNormal = LinearInterp(x, SpanLeft[n].x, SpanRight[n].x, SpanLeft[n].n, SpanRight[n].n);
        ptNormal = ptNormal.Normalize();
        CRGB Intensity = pLight->Illuminate(ViewPoint, CP3(x, y, CurrentDepth), ptNormal, pMaterial);
        if (CurrentDepth <= zBuffer[x + nWidth / 2][y + nHeight / 2])//ZBuffer 算法
        {
            zBuffer[x + nWidth / 2][y + nHeight / 2] = CurrentDepth;
```

```
                pDC->SetPixelV(x, y, RGB(Intensity.red * 255, Intensity.green * 255,
Intensity.blue * 255));
            }
            CurrentDepth += DepthStep;
        }
    }
    if (SpanLeft)
    {
        delete[]SpanLeft;
        SpanLeft = NULL;
    }
    if (SpanRight)
    {
        delete[]SpanRight;
        SpanRight = NULL;
    }
    }
    void CZBuffer::EdgeFlag(CPoint2 PStart, CPoint2 PEnd, BOOL bFeature)
    {
    int dx = PEnd.x - PStart.x;
    int dy = PEnd.y - PStart.y;
    double m = double(dx) / dy;
    double x = PStart.x;
    for (int y = PStart.y; y < PEnd.y; y++)
    {
        CVector3 ptNormal = LinearInterp(y, PStart.y, PEnd.y, PStart.n, PEnd.n);
        if (bFeature)
            SpanLeft[nIndex++] = CPoint2(ROUND(x), y, ptNormal);
        else
            SpanRight[nIndex++] = CPoint2(ROUND(x), y, ptNormal);
        x += m;
    }
    }
    void CZBuffer::SortVertex(void)
    {
    CPoint3 pt[3];
    pt[0] = point0;
    pt[1] = point1;
    pt[2] = point2;
    for (int i = 0; i < 2; i++)
    {
        int min = i;
        for (int j = i + 1; j < 3; j++)
            if (pt[j].y < pt[min].y)
```

```
                min = j;
            CPoint3 pTemp = pt[i];
            pt[i] = pt[min];
            pt[min] = pTemp;
        }
        point0 = pt[0];
        point1 = pt[2];
        point2 = pt[1];
    }
    CVector3 CZBuffer::LinearInterp(double t, double tStart, double tEnd, CVector3 vStart, CVector3 vEnd)//向量线性插值
    {
        CVector3 vector;
        vector = (tEnd - t) / (tEnd - tStart) * vStart + (t - tStart) / (tEnd - tStart) * vEnd;
        return vector;
    }
    void CZBuffer::InitialDepthBuffer(int nWidth, int nHeight, double zDepth)//初始化深度缓冲
    {
        this->nWidth = nWidth, this->nHeight = nHeight;
        zBuffer = new double *[nWidth];
        for (int i = 0; i < nWidth; i++)
            zBuffer[i] = new double[nHeight];
        for (int i = 0; i < nWidth; i++)//初始化深度缓冲
            for (int j = 0; j < nHeight; j++)
                zBuffer[i][j] = zDepth;
    }
```

（7）初始化光照环境

```
  void CTestView::InitializeLightingScene(void)//初始化光照环境
  {
    //设置光源属性
    nLightSourceNumber = 1;//光源个数
    pLight = new CLighting(nLightSourceNumber);//一维光源动态数组
    pLight->LightSource[0].SetPosition(0, 0, 1000);//设置光源位置坐标
    for (int i = 0; i < nLightSourceNumber; i++)
    {
        pLight->LightSource[i].L_Diffuse = CRGB(1.0, 1.0, 1.0);//光源的漫反射颜色
        pLight->LightSource[i].L_Specular = CRGB(1.0, 1.0, 1.0);//光源镜面高光颜色
        pLight->LightSource[i].L_C0 = 1.0;//常数衰减因子
        pLight->LightSource[i].L_C1 = 0.0000001;//线性衰减因子
        pLight->LightSource[i].L_C2 = 0.00000001;//二次衰减因子
        pLight->LightSource[i].L_OnOff = TRUE;//光源开启
    }
    //设置材质属性
    pMaterial = new CMaterial;
    pMaterial->SetAmbient(CRGB(0.847, 0.10, 0.075));//环境反射率
    pMaterial->SetDiffuse(CRGB(0.852, 0.006, 0.026));//漫反射率
    pMaterial->SetSpecular(CRGB(1.0, 1.0, 1.0));//镜面反射率
    pMaterial->SetEmission(CRGB(0.0, 0.0, 0.0));//自身辐射的颜色
    pMaterial->SetExponent(10);//高光指数
  }
```

（8）使用正弦函数扰动四边形网格顶点法向量，绘制四边形小面

① 在 CBezierPatch 类中增加如下成员：

```
public:
    void SetScene(CLighting* pLight, CMaterial* pMaterial);//设置场景
private:
    CLighting* pLight;//光照
    CMaterial* pMaterial;//材质
    CVector3 quadrN[4];//四边形的顶点法向量
```

其中，SetScene()函数的定义如下：

```
void CBezierPatch::SetScene(CLighting* pLight, CMaterial* pMaterial)//设置场景
{
    this->pLight = pLight;
    this->pMaterial = pMaterial;
}
```

② CBezierPatch 类的 DrawFacet 函数调整为：

```
void CBezierPatch::DrawFacet(CDC* pDC, CZBuffer* pZBuffer)
{
CP3 ScreenPoint[4];//三维投影点
CP3 ViewPoint = projection.GetEye();//视点
CVector3 NewVector[4], PerturbationVector;//顶点法向量
for (int nPoint = 0; nPoint < 4; nPoint++)
{
    ScreenPoint[nPoint] = projection.PerspectiveProjection3(quadrP[nPoint]);//透视投影
    double Frequency = 50;
    double bump = sin((quadrP[nPoint].x + quadrP[nPoint].y + quadrP[nPoint].z) /
Frequency);//正弦函数扰动
    PerturbationVector = CVector3(bump, bump, bump);//扰动向量
    NewVector[nPoint] = quadrN[nPoint] + PerturbationVector;
}
pZBuffer->SetPoint(ScreenPoint[0], ScreenPoint[2], ScreenPoint[3], NewVector[0],
NewVector[2], NewVector[3]);
pZBuffer->PhongShader(pDC, ViewPoint, pLight, pMaterial);
pZBuffer->SetPoint(ScreenPoint[0], ScreenPoint[1], ScreenPoint[2], NewVector[0],
NewVector[1], NewVector[2]);
pZBuffer->PhongShader(pDC, ViewPoint, pLight, pMaterial);
}
```

其中，要在 CProjection 类中增加三维透视投影函数 PerspectiveProjection3()如下：

```
CP3 CProjection::PerspectiveProjection3(CP3 WorldPoint)//三维透视投影
{
  CP3 ViewPoint;//观察坐标系三维点
  ViewPoint.x = WorldPoint.x;
  ViewPoint.y = WorldPoint.y;
  ViewPoint.z = EyePoint.z - WorldPoint.z;
  ViewPoint.c = WorldPoint.c;
  CP3 ScreenPoint;//屏幕坐标系三维点
  ScreenPoint.x = d * ViewPoint.x / ViewPoint.z;
  ScreenPoint.y = d * ViewPoint.y / ViewPoint.z;
  ScreenPoint.z = (ViewPoint.z - d) * d / ViewPoint.z;//Bouknight 公式
  ScreenPoint.c = ViewPoint.c;
  return ScreenPoint;
}
```