**案例54-球体几何纹理映射算法**

文档编写：霍波魏

校稿/修订：孔令德

时间2019~2020

联系方式：QQ997796978

**说明：**本套案例由孔令德开发，原版本为Visual C++6.0，配套于孔令德的著作《计算机图形学-基于MFC三维图形开发》一书。孔令德计算机工程研究所的学生霍波魏在学习计算机图形学期间，对本套案例进行了升级并编写了学习文档。现在程序的编写和程序的解释都是基于Windows 10操作系统，使用Microsoft visual studio 2017平台的MFC（英文版）开发。

1. **案例描述**

本案例通过读入高度场位图制作一个文字凹凸纹理三维动画。

1. **知识点**

高度场凹凸纹理的和是使用8位灰度图像定义的。灰度图像中白色纹理表示高的区域，黑色纹理表示低的quyu。高度场中的和需要使用中心差分计算，相邻列的差得到，相邻行的差得到。



表面法矢量的扰动如图54-1所示

 +=

（a） 光滑表面 （b） 扰动函数 （c） 扰动后的表面

图54-1凹凸纹理映射

扰动后的法矢量的计算为：



1. **实现步骤**
2. 添加基础类与添加绘制球体的CSphere类。
3. 在CSphere类中计算顶点坐标、读入面表，绘制图形，调用CZBuffer类中的SetPoint函数绑定长方体的顶点纹理坐标，并在CZBuffer类中进行线性插值，在Phong函数中将扰动量和叠加到小面内各点的法矢量上，通过调用PhongShader来绘制光照文字凹凸球。
4. 在CLightSource类中对光源参数进行初始化，在CMaterial类中对材质属性进行初始化，在CLighting类中对光强进行计算。
5. 在CProjection类中进行透视变化。
6. 在CTestView中添加消息响应函数，在OnDraw中调用DoubleBuffer函数。
7. **主要算法**

1. CSphere类

public:

CSphere();

virtual~CSphere();

void ReadVertex();//读入顶点坐标

void ReadFace();//读入面坐标

void Draw(CDC\* pDC);//绘制球面

public:

CP3d \*V;//圆柱顶点一维数组

CT2 \*T;//圆柱纹理一维数组

CFace \*\*F;//面的二维数组

int N1, N2;//N1为经度区间,N2为纬度区间

int gAlpha, gBeta;//纬度角与经度角

CLighting \*pLight;//光照环境

CMaterial \*pMaterial;//物体材质

CProjection projection;

CTexture texture;

CSphere::CSphere()

{

}

CSphere::~CSphere()

{

if (pLight != NULL)

{

delete pLight;

pLight = NULL;

}

if (pMaterial != NULL)

{

delete pMaterial;

pLight = NULL;

}

if (V != NULL)

{

delete[]V;

V = NULL;

}

if (T != NULL)

{

delete[]T;

T = NULL;

}

for (int n = 0; n < N1; n++)

{

delete[] F[n];

F[n] = NULL;

}

delete[]F;

F = NULL;

}

void CSphere::ReadVertex()//点表

{

gAlpha = 4, gBeta = 8;//面片夹角

int r = 300;//r为球面半径

N1 = 180 / gAlpha, N2 = 360 / gBeta;//N1为纬度区域,N2为经度区域

V = new CP3d[(N1 - 1)\*N2 + 2];

T = new CT2[(N1 - 1)\*N2 + 2];

double gAlpha1, gBeta1;

//计算北极点坐标

V[0].x = 0, V[0].y = r, V[0].z = 0;

T[0].u = 0; T[0].v = 0;//闲置

//按行循环计算球体上的点坐标

for (int i = 0; i < N1 - 1; i++)

{

gAlpha1 = (i + 1)\*gAlpha\*PI / 180;

for (int j = 0; j < N2; j++)

{

gBeta1 = j \* gBeta\*PI / 180;

V[i\*N2 + j + 1].x = r \* sin(gAlpha1)\*sin(gBeta1);

V[i\*N2 + j + 1].y = r \* cos(gAlpha1);

V[i\*N2 + j + 1].z = r \* sin(gAlpha1)\*cos(gBeta1);

T[i\*N2 + j + 1].u = gBeta1 / (2 \* PI)\*(texture.bmp.bmWidth - 1);

T[i\*N2 + j + 1].v = (PI - gAlpha1) / PI \* (texture.bmp.bmHeight - 1);

}

}

//计算南极点坐标

V[(N1 - 1)\*N2 + 1].x = 0, V[(N1 - 1)\*N2 + 1].y = -r, V[(N1 - 1)\*N2 + 1].z = 0;

T[(N1 - 1)\*N2 + 1].u = 0; T[(N1 - 1)\*N2 + 1].v = 0;//闲置

}

void CSphere::ReadFace()//读入面表

{

//设置二维动态数组

F = new CFace \*[N1];//设置行

for (int n = 0; n < N1; n++)

F[n] = new CFace[N2];//设置列

for (int j = 0; j < N2; j++)//构造北极三角形面片

{

int tempj = j + 1;

if (tempj == N2) tempj = 0;//面片的首尾连接

int NorthIndex[3];//北极三角形面片索引号数组

NorthIndex[0] = 0;

NorthIndex[1] = j + 1;

NorthIndex[2] = tempj + 1;

F[0][j].SetNum(3);

for (int k = 0; k < F[0][j].vN; k++)

F[0][j].vI[k] = NorthIndex[k];

}

for (int i = 1; i < N1 - 1; i++)//构造球面四边形面片

{

for (int j = 0; j < N2; j++)

{

int tempi = i + 1;

int tempj = j + 1;

if (tempj == N2) tempj = 0;

int BodyIndex[4];//球面四边形面片索引号数组

BodyIndex[0] = (i - 1)\*N2 + j + 1;

BodyIndex[1] = (tempi - 1)\*N2 + j + 1;

BodyIndex[2] = (tempi - 1)\*N2 + tempj + 1;

BodyIndex[3] = (i - 1)\*N2 + tempj + 1;

F[i][j].SetNum(4);

for (int k = 0; k < F[i][j].vN; k++)

F[i][j].vI[k] = BodyIndex[k];

}

}

for (int j = 0; j < N2; j++)//构造南极三角形面片

{

int tempj = j + 1;

if (tempj == N2) tempj = 0;

int SouthIndex[3];//南极三角形面片索引号数组

SouthIndex[0] = (N1 - 2)\*N2 + j + 1;

SouthIndex[1] = (N1 - 1)\*N2 + 1;

SouthIndex[2] = (N1 - 2)\*N2 + tempj + 1;

F[N1 - 1][j].SetNum(3);

for (int k = 0; k < F[N1 - 1][j].vN; k++)

F[N1 - 1][j].vI[k] = SouthIndex[k];

}

}

void CSphere::Draw(CDC \*pDC)//绘制球面

{

CZBuffer \*zbuf = new CZBuffer;//申请内存

zbuf->InitDeepBuffer(800, 800, 1000);//初始化深度缓冲器

zbuf->ReadGradient(texture.Imgx\_Gradient, texture.Imgy\_Gradient);

CPi3 Point3[3];//底面与顶面三角形顶点数组

CT2 Texture3[3];//底面与顶面三角形纹理数组

CVector3 Normal3[3];//底面与顶面三角形法矢量数组

CPi3 Point4[4];//侧面四边形顶点数组

CT2 Texture4[4];//侧面四边形纹理数组

CVector3 Normal4[4];//侧面四边形法矢量数组

for (int i = 0; i < N1; i++)

{

for (int j = 0; j < N2; j++)

{

CVector3 ViewVector(V[F[i][j].vI[0]], projection.ViewPoint);//面的视矢量

ViewVector = ViewVector.Normalize();//单位化视矢量

F[i][j].SetFaceNormal(V[F[i][j].vI[0]], V[F[i][j].vI[1]], V[F[i][j].vI[2]]);//计算小面片法矢量

F[i][j].fNormal.Normalize();//单位化法矢量

if (Dot(ViewVector, F[i][j].fNormal) >= 0)

{

if (3 == F[i][j].vN)//处理三角形面片

{

for (int m = 0; m < F[i][j].vN; m++)

{

projection.PerProject(V[F[i][j].vI[m]]);

Point3[m] = projection.ScreenP;

Normal3[m] = CVector3(V[F[i][j].vI[m]]);

}

double tempi = i + 1, tempj = j + 1;//对三角形面片进行特殊处理

Texture3[0].u = gBeta \* (j) / 360.0\*(texture.bmp.bmWidth - 1); Texture3[0].v = (1.0 - gAlpha \* i / 180.0)\*(texture.bmp.bmHeight - 1);

Texture3[1].u = gBeta \* (j) / 360.0\*(texture.bmp.bmWidth - 1); Texture3[1].v = (1.0 - gAlpha \* tempi / 180.0)\*(texture.bmp.bmHeight - 1);

Texture3[2].u = gBeta \* tempj / 360.0\*(texture.bmp.bmWidth - 1); Texture3[2].v = (1.0 - gAlpha \* i / 180.0)\*(texture.bmp.bmHeight - 1);

zbuf->SetPoint(Point3, Normal3, Texture3, 3);//初始化

zbuf->CreateBucket();//创建桶表

zbuf->CreateEdge();//创建边表

zbuf->Phong(pDC, projection.ViewPoint, pLight, pMaterial);//填充三角形

zbuf->ClearMemory();

}

else//处理四边形面片

{

for (int m = 0; m < F[i][j].vN; m++)

{

projection.PerProject(V[F[i][j].vI[m]]);

Point4[m] = projection.ScreenP;

Normal4[m] = CVector3(V[F[i][j].vI[m]]);

Texture4[m] = T[F[i][j].vI[m]];

}

if (N2 - 1 == j)//消除图像纹理的接缝

{

Texture4[2].u = texture.bmp.bmWidth - 1;

Texture4[3].u = texture.bmp.bmWidth - 1;

}

zbuf->SetPoint(Point4, Normal4, Texture4, 4);//初始化

zbuf->CreateBucket();//创建桶表

zbuf->CreateEdge();//创建边表

zbuf->Phong(pDC, projection.ViewPoint, pLight, pMaterial);//填充四边形

zbuf->ClearMemory();

}

}

}

}

delete zbuf;

}

2.CZBuffer类

public:

CZBuffer();

virtual ~CZBuffer();

void ReadGradient(double \*\*x\_gra, double \*\*y\_gra);

void CreateBucket();//创建桶

void CreateEdge();//边表

void AddEt(CAET \*pNewEdge);//合并ET表

void ETOrder();//ET表排序

void Phong(CDC \*pDC, CP3d ViewPoint, CLighting \*pLight, CMaterial \*pMaterial);//Phong填充函数

void InitDeepBuffer(int Width, int Height, double Depth);//初始化深度缓冲器

CRGB Interpolation(double t, double t1, double t2, CRGB clr1, CRGB clr2);//颜色线性插值

CVector3 Interpolation(double t, double t1, double t2, CVector3 vec1, CVector3 vec2);//法矢线性插值

CT2 Interpolation(double t, double t1, double t2, CT2 tex1, CT2 tex2);//纹理线性插值

void SetPoint(CPi3 \*p, CVector3 \*n, CT2 \*t, int m);

void ClearMemory();//清理内存

void DeleteAETChain(CAET\* pAET);//删除边表

protected:

int PNum;//顶点个数

CPi3 \*P;//顶点数组

CT2 \*T;//纹理动态数组

CVector3 \*N;//顶点的法矢量动态数组

CAET \*pHeadE, \*pCurrentE, \*pEdge;//有效边表结点指针

CBucket \*pCurrentB, \*pHeadB;

double \*\*zBuffer;//缓深度冲区

int Width, Height;//缓冲区参数

double \*\*x\_gra;

double \*\*y\_gra;

CZBuffer::CZBuffer()

{

P = NULL;

N = NULL;

T = NULL;

pHeadE = NULL;

pCurrentB = NULL;

pEdge = NULL;

pCurrentE = NULL;

pHeadB = NULL;

zBuffer = NULL;

}

CZBuffer::~CZBuffer()

{

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

{

delete[] zBuffer[i];

zBuffer[i] = NULL;

}

if (zBuffer != NULL)

{

delete zBuffer;

zBuffer = NULL;

}

ClearMemory();

}

void CZBuffer::SetPoint(CPi3 \*p, CVector3 \*n, CT2 \*t, int m)

{

P = new CPi3[m];

N = new CVector3[m];

T = new CT2[m];

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

{

P[i] = p[i];

N[i] = n[i];

T[i] = t[i];

}

PNum = m;

}

void CZBuffer::CreateBucket()//创建桶表

{

int yMin, yMax;

yMin = yMax = P[0].y;

for (int i = 1; i < PNum; i++)//查找多边形所覆盖的最小和最大扫描线

{

if (P[i].y < yMin)

yMin = P[i].y;//扫描线的最小值

if (P[i].y > yMax)

yMax = P[i].y;//扫描线的最大值

}

for (int y = yMin; y <= yMax; y++)

{

if (yMin == y)//建立桶头结点

{

pHeadB = new CBucket;//建立桶的头结点

pCurrentB = pHeadB;//pCurrentB为CBucket当前结点指针

pCurrentB->ScanLine = yMin;

pCurrentB->pET = NULL;//没有链接边表

pCurrentB->pNext = NULL;

}

else//其他扫描线

{

pCurrentB->pNext = new CBucket;//建立桶的其他结点

pCurrentB = pCurrentB->pNext;

pCurrentB->ScanLine = y;

pCurrentB->pET = NULL;

pCurrentB->pNext = NULL;

}

}

}

void CZBuffer::CreateEdge()//创建边表

{

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

{

pCurrentB = pHeadB;

int j = (i + 1) % PNum;//边的第二个顶点，P[i]和P[j]构成边

if (P[i].y < P[j].y)//边的终点比起点高

{

pEdge = new CAET;

pEdge->x = P[i].x;//计算ET表的值

pEdge->yMax = P[j].y;

pEdge->k = (P[j].x - P[i].x) / (P[j].y - P[i].y);//代表1/k

pEdge->ps = P[i];//绑定顶点和颜色

pEdge->pe = P[j];

pEdge->ns = N[i];//绑定法矢量

pEdge->ne = N[j];

pEdge->ts = T[i];//绑定纹理

pEdge->te = T[j];

pEdge->pNext = NULL;

while (pCurrentB->ScanLine != P[i].y)//在桶内寻找该边的yMin

{

pCurrentB = pCurrentB->pNext;//移到yMin所在的桶结点

}

}

if (P[j].y < P[i].y)//边的终点比起点低

{

pEdge = new CAET;

pEdge->x = P[j].x;

pEdge->yMax = P[i].y;

pEdge->k = (P[i].x - P[j].x) / (P[i].y - P[j].y);

pEdge->ps = P[i];

pEdge->pe = P[j];

pEdge->ns = N[i];

pEdge->ne = N[j];

pEdge->ts = T[i];

pEdge->te = T[j];

pEdge->pNext = NULL;

while (pCurrentB->ScanLine != P[j].y)

{

pCurrentB = pCurrentB->pNext;

}

}

if (int(P[j].y) != P[i].y)

{

pCurrentE = pCurrentB->pET;

if (pCurrentE == NULL)

{

pCurrentE = pEdge;

pCurrentB->pET = pCurrentE;

}

else

{

while (pCurrentE->pNext != NULL)

{

pCurrentE = pCurrentE->pNext;

}

pCurrentE->pNext = pEdge;

}

}

}

}

void CZBuffer::Phong(CDC \*pDC, CP3d ViewPoint, CLighting \*pLight, CMaterial \*pMaterial)//填充多边形

{

double z = 0.0;//当前扫描线的z

double zStep = 0.0;//当前扫描线随着x增长的z步长

double A, B, C, D;//平面方程Ax+By+Cz＋D=0的系数

CVector3 V01(P[0], P[1]), V02(P[0], P[2]);

CVector3 VN = Cross(V01, V02);

A = VN.x; B = VN.y; C = VN.z;

D = -A \* P[0].x - B \* P[0].y - C \* P[0].z;

zStep = -A / C;//计算直线z增量

CAET \*pT1, \*pT2;

pT1 = NULL;

pT2 = NULL;

pHeadE = NULL;

for (pCurrentB = pHeadB; pCurrentB != NULL; pCurrentB = pCurrentB->pNext)

{

for (pCurrentE = pCurrentB->pET; pCurrentE != NULL; pCurrentE = pCurrentE->pNext)

{

pEdge = new CAET;

pEdge->x = pCurrentE->x;

pEdge->yMax = pCurrentE->yMax;

pEdge->k = pCurrentE->k;

pEdge->ps = pCurrentE->ps;

pEdge->pe = pCurrentE->pe;

pEdge->ns = pCurrentE->ns;

pEdge->ne = pCurrentE->ne;

pEdge->ts = pCurrentE->ts;

pEdge->te = pCurrentE->te;

pEdge->pNext = NULL;

AddEt(pEdge);

}

ETOrder();

pT1 = pHeadE;

if (pT1 == NULL)

return;

while (pCurrentB->ScanLine >= pT1->yMax)//下闭上开

{

CAET \* pAETTEmp = pT1;

pT1 = pT1->pNext;

delete pAETTEmp;

pHeadE = pT1;

if (pHeadE == NULL)

return;

}

if (pT1->pNext != NULL)

{

pT2 = pT1;

pT1 = pT2->pNext;

}

while (pT1 != NULL)

{

if (pCurrentB->ScanLine >= pT1->yMax)//下闭上开

{

CAET\* pAETTemp = pT1;

pT2->pNext = pT1->pNext;

pT1 = pT2->pNext;

delete pAETTemp;

}

else

{

pT2 = pT1;

pT1 = pT2->pNext;

}

}

CVector3 na, nb, nf;//na、nb代表边上任意点的法矢量，nf代表面上任意点的法矢量

na = Interpolation(pCurrentB->ScanLine, pHeadE->ps.y, pHeadE->pe.y, pHeadE->ns, pHeadE->ne);

nb = Interpolation(pCurrentB->ScanLine, pHeadE->pNext->ps.y, pHeadE->pNext->pe.y, pHeadE->pNext->ns, pHeadE->pNext->ne);

CT2 ta, tb, tf;//ta和tb代表边上任意点的纹理，tf代表面上任意点的纹理

ta = Interpolation(pCurrentB->ScanLine, pHeadE->ps.y, pHeadE->pe.y, pHeadE->ts, pHeadE->te);

tb = Interpolation(pCurrentB->ScanLine, pHeadE->pNext->ps.y, pHeadE->pNext->pe.y, pHeadE->pNext->ts, pHeadE->pNext->te);

BOOL bInFlag = FALSE;//区间内外测试标志，初始值为假表示区间外部

double xb, xe;//扫描线和有效边相交区间的起点和终点坐标

for (pT1 = pHeadE; pT1 != NULL; pT1 = pT1->pNext)

{

if (FALSE == bInFlag)

{

xb = pT1->x;

z = -(xb\*A + pCurrentB->ScanLine\*B + D) / C;//z=-(Ax+By+D)/C

bInFlag = TRUE;

}

else

{

xe = pT1->x;

for (double x = xb; x < xe; x++)//左闭右开

{

nf = Interpolation(x, xb, xe, na, nb);

tf = Interpolation(x, xb, xe, ta, tb);

CVector3 n1, n2;

n1 = CVector3(x\_gra[Round(tf.v)][Round(tf.u)], 0, 0);

n2 = CVector3(0, y\_gra[Round(tf.v)][Round(tf.u)], 0);

nf += n1 + n2;

CRGB c = pLight->Illuminate(ViewPoint, CP3d(Round(x), pCurrentB->ScanLine, z), nf, pMaterial);

if (z <= zBuffer[Round(x) + Width / 2][pCurrentB->ScanLine + Height / 2])//如果新采样点的深度小于原采样点的深度

{

zBuffer[Round(x) + Width / 2][pCurrentB->ScanLine + Height / 2] = z;//xy坐标与数组下标保持一致

pDC->SetPixelV(Round(x), pCurrentB->ScanLine, RGB(c.red \* 255, c.green \* 255, c.blue \* 255));

}

z += zStep;

}

bInFlag = FALSE;

}

}

for (pT1 = pHeadE; pT1 != NULL; pT1 = pT1->pNext)//边的连续性

pT1->x = pT1->x + pT1->k;

}

}

void CZBuffer::AddEt(CAET \*pNewEdge)//合并ET表

{

CAET \*pCE;

pCE = pHeadE;

if (pCE == NULL)

{

pHeadE = pNewEdge;

pCE = pHeadE;

}

else

{

while (pCE->pNext != NULL)

{

pCE = pCE->pNext;

}

pCE->pNext = pNewEdge;

}

}

void CZBuffer::ETOrder()//边表的冒泡排序算法

{

CAET \*pT1, \*pT2;

int Count = 1;

pT1 = pHeadE;

if (pT1 == NULL)

return;

if (pT1->pNext == NULL)//如果该ET表没有再连ET表

return;//桶结点只有一条边，不需要排序

while (pT1->pNext != NULL)//统计边结点的个数

{

Count++;

pT1 = pT1->pNext;

}

for (int i = 0; i < Count - 1; i++)//冒泡排序

{

CAET \*\*pPre = &pHeadE;

pT1 = pHeadE;

for (int j = 0; j < Count - 1 - i; j++)

{

pT2 = pT1->pNext;

if ((pT1->x > pT2->x) || ((pT1->x == pT2->x) && (pT1->k > pT2->k)))

{

pT1->pNext = pT2->pNext;

pT2->pNext = pT1;

\*pPre = pT2;

pPre = &(pT2->pNext);//调整位置为下次遍历准备

}

else

{

pPre = &(pT1->pNext);

pT1 = pT1->pNext;

}

}

}

}

CRGB CZBuffer::Interpolation(double t, double t1, double t2, CRGB clr1, CRGB clr2)//颜色线性插值

{

CRGB color;

color = (t - t2) / (t1 - t2)\*clr1 + (t - t1) / (t2 - t1)\*clr2;

return color;

}

CVector3 CZBuffer::Interpolation(double t, double t1, double t2, CVector3 vec1, CVector3 vec2)//矢量线性插值

{

CVector3 vector;

vector = (t - t2) / (t1 - t2)\*vec1 + (t - t1) / (t2 - t1)\*vec2;

return vector;

}

CT2 CZBuffer::Interpolation(double t, double t1, double t2, CT2 tex1, CT2 tex2)//纹理线性插值

{

CT2 texture;

texture = (t - t2) / (t1 - t2)\*tex1 + (t - t1) / (t2 - t1)\*tex2;

return texture;

}

void CZBuffer::InitDeepBuffer(int Width, int Height, double Depth)//初始化深度缓冲

{

this->Width = Width, this->Height = Height;

zBuffer = new double \*[Width];

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

zBuffer[i] = new double[Height];

for (int i = 0; i < Width; i++)//初始化深度缓冲

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

zBuffer[i][j] = Depth;

}

void CZBuffer::ReadGradient(double \*\*x\_gra, double \*\*y\_gra)

{

this->x\_gra = x\_gra;

this->y\_gra = y\_gra;

}

void CZBuffer::ClearMemory()

{

DeleteAETChain(pHeadE);

CBucket \*pBucket = pHeadB;

while (pBucket != NULL)//针对每一个桶

{

CBucket \* pBucketTemp = pBucket->pNext;

DeleteAETChain(pBucket->pET);

delete pBucket;

pBucket = pBucketTemp;

}

pHeadB = NULL;

pHeadE = NULL;

if (P != NULL)

{

delete[]P;

P = NULL;

}

if (N != NULL)

{

delete[]N;

N = NULL;

}

if (T != NULL)

{

delete[]T;

T = NULL;

}

}

void CZBuffer::DeleteAETChain(CAET\* pAET)

{

while (pAET != NULL)

{

CAET\* pAETTemp = pAET->pNext;

delete pAET;

pAET = pAETTemp;

}

}

3.CTestView类

public:

void DoubleBuffer(CDC\* pDC);//双缓冲绘图

void DrawObject(CDC\* pDC);//绘制物体

void InitialLightingScene(void);

protected:

int LightNum;//光源数量

BOOL bPlay;//动画开关

CTransform tran;//变换对象

CSphere sphere;

void CTestView::InitialLightingScene(void)

{

LightNum = 1;//光源个数

sphere.pLight = new CLighting(LightNum);//一维光源动态数组

sphere.pLight->LightSource[0].SetPosition(800, 800, 800);//设置第一个光源位置坐标

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

{

sphere.pLight->LightSource[i].L\_Diffuse = CRGB(1.0, 1.0, 1.0); //光源的漫反射颜色

sphere.pLight->LightSource[i].L\_Specular = CRGB(0.8, 0.8, 0.8);//光源镜面高光颜色

sphere.pLight->LightSource[i].L\_C0 = 1.0;//常数衰减系数

sphere.pLight->LightSource[i].L\_C1 = 0.0001;//线性衰减系数

sphere.pLight->LightSource[i].L\_C2 = 0.00000001;//二次衰减系数

sphere.pLight->LightSource[i].L\_OnOff = TRUE;//光源开启

}

sphere.pMaterial = new CMaterial;//一维材质动态数组

sphere.pMaterial->SetAmbient(CRGB(0.247, 0.200, 0.075));//材质对环境光光的反射率

sphere.pMaterial->SetDiffuse(CRGB(0.6, 0.0, 0.0));//材质对漫反射光的反射率

sphere.pMaterial->SetSpecular(CRGB(1.0, 1.0, 1.0));//材质对镜面反射光的反射率

sphere.pMaterial->M\_Emit = CRGB(0.2, 0.0, 0.0);//材质自身发散的颜色

sphere.pMaterial->M\_n = 20.0;//高光指数

}

void CTestView::DoubleBuffer(CDC\* pDC)//双缓冲

{

CRect rect;//定义客户区矩形

GetClientRect(&rect);//获得客户区的大小

pDC->SetMapMode(MM\_ANISOTROPIC);//pDC自定义坐标系

pDC->SetWindowExt(rect.Width(), rect.Height());//设置窗口范围

pDC->SetViewportExt(rect.Width(), -rect.Height());//设置视区范围,x轴水平向右，y轴垂直向上

pDC->SetViewportOrg(rect.Width() / 2, rect.Height() / 2);//客户区中心为原点

CDC memDC;//内存DC

memDC.CreateCompatibleDC(pDC);//创建一个与显示pDC兼容的内存memDC

CBitmap NewBitmap, \*pOldBitmap;//内存中承载的临时位图

NewBitmap.CreateCompatibleBitmap(pDC, rect.Width(), rect.Height());//创建兼容位图

pOldBitmap = memDC.SelectObject(&NewBitmap);//将兼容位图选入memDC

memDC.SetMapMode(MM\_ANISOTROPIC);//memDC自定义坐标系

memDC.SetWindowExt(rect.Width(), rect.Height());

memDC.SetViewportExt(rect.Width(), -rect.Height());

memDC.SetViewportOrg(rect.Width() / 2, rect.Height() / 2);

rect.OffsetRect(-rect.Width() / 2, -rect.Height() / 2);

DrawObject(&memDC);//向memDC绘制图形

pDC->BitBlt(rect.left, rect.top, rect.Width(), rect.Height(), &memDC, -rect.Width() / 2, -rect.Height() / 2, SRCCOPY);//将内存memDC中的位图拷贝到显示pDC中

memDC.SelectObject(pOldBitmap);//恢复位图

NewBitmap.DeleteObject();//删除位图

}

void CTestView::DrawObject(CDC \* pDC)

{

sphere.Draw(pDC);

}

1. **实现效果**

球面几何纹理映射算法效果如图54-2所示。

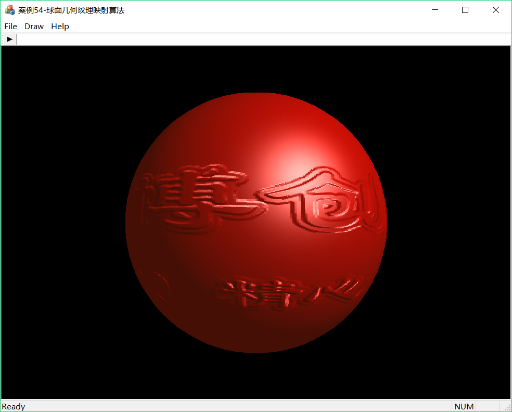


图54-2 球面几何纹理映射算法效果图