**案例49-立方体函数纹理映射算法**

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. **知识点**

我们通常认为函数纹理包含棋盘纹理和粗布纹理两种，这两种纹理不是通过读入位图生成的，而是通过函数来自动生成的纹理。

立方体中棋盘纹理是将每一个面按照指定的间距划分为若干个正方形，每一个正方形交替返回两种不同的颜色值，从而生成棋纹理。

立方体中粗布纹理的实现主要是依靠粗布纹理函数，公式见式(49-1)：

 （49-1）

式中A是取值范围在[0，1]之间的随机变量，q和p为频率系数。

1. **实现步骤**
2. 添加基础类与添加绘制立方体的CCube类。
3. 在CCube类中计算顶点坐标、读入面表，绘制图形。
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CZBuffer类中通过函数获取纹理颜色,并借助双线性插值获取网格面内每一点的纹理颜色。
6. 在CProjection类中进行透视变化。
7. 在CTestView中添加消息响应函数，在OnDraw中调用DoubleBuffer函数。
8. **主要算法**

1. CCube类

public:

CCube();

virtual~CCube();

void ReadPoint();//读入点表

void ReadFace();//读入面表

void Draw(CDC \*pDC);//绘制立方体表面

public:

CP3d V[8];//点表

CFace F[6];//面表

CProjection projection;

CLighting \*pLight;//光照环境

CMaterial \*pMaterial;//物体材质

CCube::CCube()

{

}

CCube::~CCube()

{

if (pLight != NULL)

{

delete pLight;

pLight = NULL;

}

if (pMaterial != NULL)

{

delete pMaterial;

pMaterial = NULL;

}

}

void CCube::ReadPoint()//点表

{

//顶点的三维坐标(x,y,z)

int a = 200;//立方体边长为2a

V[0].x = -a; V[0].y = -a; V[0].z = -a;

V[1].x = +a; V[1].y = -a; V[1].z = -a;

V[2].x = +a; V[2].y = +a; V[2].z = -a;

V[3].x = -a; V[3].y = +a; V[3].z = -a;

V[4].x = -a; V[4].y = -a; V[4].z = +a;

V[5].x = +a; V[5].y = -a; V[5].z = +a;

V[6].x = +a; V[6].y = +a; V[6].z = +a;

V[7].x = -a; V[7].y = +a; V[7].z = +a;

}

void CCube::ReadFace()//面表

{

//面的边数、面的顶点编号

F[0].SetNum(4); F[0].vI[0] = 4; F[0].vI[1] = 5; F[0].vI[2] = 6; F[0].vI[3] = 7;//前面顶点索引

F[0].t[0] = CT2(0, 0); F[0].t[1] = CT2(1, 0); F[0].t[2] = CT2(1, 1); F[0].t[3] = CT2(0, 1);//前面纹理坐标

F[1].SetNum(4); F[1].vI[0] = 0; F[1].vI[1] = 3; F[1].vI[2] = 2; F[1].vI[3] = 1;//后面顶点索引

F[1].t[0] = CT2(1, 0); F[1].t[1] = CT2(1, 1); F[1].t[2] = CT2(0, 1); F[1].t[3] = CT2(0, 0);//后面纹理坐标

F[2].SetNum(4); F[2].vI[0] = 0; F[2].vI[1] = 4; F[2].vI[2] = 7; F[2].vI[3] = 3;//左面顶点索引

F[2].t[0] = CT2(0, 0); F[2].t[1] = CT2(1, 0); F[2].t[2] = CT2(1, 1); F[2].t[3] = CT2(0, 1);//左面纹理坐标

F[3].SetNum(4); F[3].vI[0] = 1; F[3].vI[1] = 2; F[3].vI[2] = 6; F[3].vI[3] = 5;//右面顶点索引

F[3].t[0] = CT2(1, 0); F[3].t[1] = CT2(1, 1); F[3].t[2] = CT2(0, 1); F[3].t[3] = CT2(0, 0);//右面纹理坐标

F[4].SetNum(4); F[4].vI[0] = 2; F[4].vI[1] = 3; F[4].vI[2] = 7; F[4].vI[3] = 6;//顶面顶点索引

F[4].t[0] = CT2(0, 0); F[4].t[1] = CT2(1, 0); F[4].t[2] = CT2(1, 1); F[4].t[3] = CT2(0, 1);//顶面纹理坐标

F[5].SetNum(4); F[5].vI[0] = 0; F[5].vI[1] = 1; F[5].vI[2] = 5; F[5].vI[3] = 4;//底面顶点索引

F[5].t[0] = CT2(0, 0); F[5].t[1] = CT2(1, 0); F[5].t[2] = CT2(1, 1); F[5].t[3] = CT2(0, 1);//底面纹理坐标

}

void CCube::Draw(CDC\* pDC)//绘制立方体

{

CPi3 Point[4];//面的顶点坐标

CT2 Texture[4];//面的纹理坐标

CVector3 Normal4[4];

CZBuffer \*zbuf = new CZBuffer;

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

for (int nFace = 0; nFace < 6; nFace++)

{

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

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

F[nFace].SetFaceNormal(V[F[nFace].vI[0]], V[F[nFace].vI[1]], V[F[nFace].vI[2]]);

F[nFace].fNormal.Normalize();//单位化法矢量

if (Dot(ViewVector, F[nFace].fNormal) >= 0)//背面剔除

{

for (int nVertex = 0; nVertex < F[nFace].vN; nVertex++)//边循环

{

projection.PerProject(V[F[nFace].vI[nVertex]]);

Point[nVertex] = projection.ScreenP;

Texture[nVertex] = F[nFace].t[nVertex];

Normal4[nVertex] = F[nFace].fNormal;

}

zbuf->SetPoint(Point, Normal4, Texture, 4);//初始化

zbuf->CreateBucket();//创建桶表

zbuf->CreateEdge();//创建边表

zbuf->Phong(pDC, projection.ViewPoint, pLight, pMaterial);//纹理映射

zbuf->ClearMemory();

}

}

delete zbuf;

}

2.CZBuffer类

public:

CZBuffer();

virtual ~CZBuffer();

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

CRGB GetTextureColor(double u, double v);

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

CVector3 \*N;//顶点的法矢量动态数组

CT2 \*T;//纹理动态数组

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

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);

CRGB Textureclr = GetTextureColor(tf.u, tf.v);

pMaterial->SetDiffuse(Textureclr);//用纹理颜色作为材质的漫反射光反射率

pMaterial->SetAmbient(Textureclr);//用纹理颜色作为材质的环境光反射率

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;

}

CRGB CZBuffer::GetTextureColor(double u, double v)//获得纹理颜色

{

//棋盘纹理

/\*if(0==(int(floor(u\*8.0))+int(floor(v\*8.0)))%2)

return CRGB(0.1,0.1,0.1);

else

return CRGB(0.9,0.9,0.9);\*/

//粗布纹理

//u, v=[0, 1]; A=[0, 1]随机变量; p, q频率系数

double A = double(rand()) / RAND\_MAX; //A在[0，1]之间

double p = 50, q = 100;

double f = A \* (cos(p\*u) + cos(q\*v));//粗布纹理函数

f = (f < 0.0) ? 0.0 : ((f > 1.0) ? 1.0 : f);

return CRGB(f, f, f);

}

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::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;

}

}

1. **实现效果**

立方体函数纹理映射效果图有两种，其中棋盘纹理如图49-1，粗布纹理如图49-2。

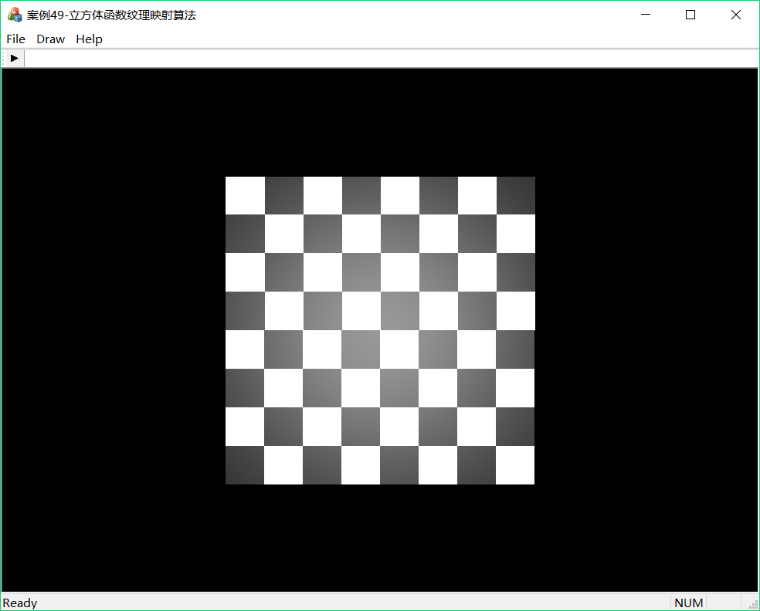


图49-1 立方体函数纹理映射之棋盘纹理效果图

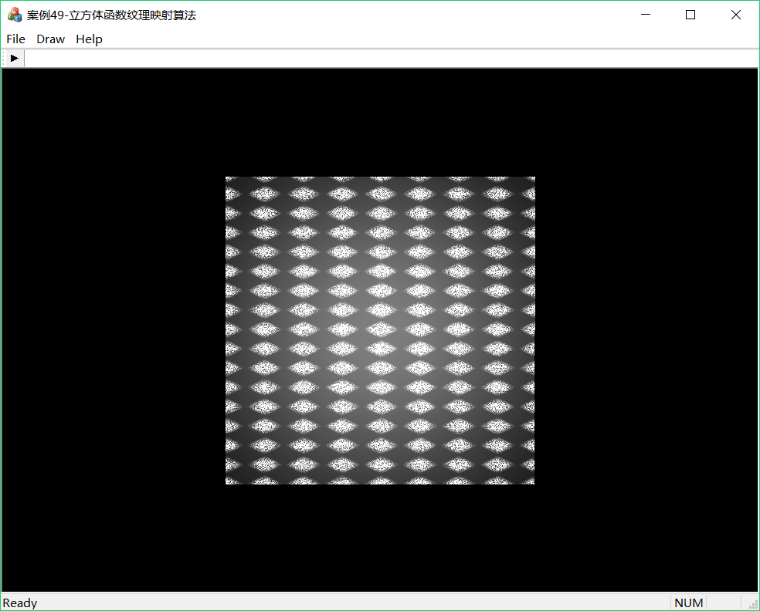


图49-2 立方体函数纹理映射之粗布纹理效果图