**案例48-简单阴影算法**

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**

本案例通过简单阴影算法绘制一个立方体阴影三维动画。

1. **知识点**

阴影是由于物体截断了光线而产生的，如果光源位于物体的一侧，阴影总是位于物体的另一侧，即与光源相反的一侧。在单点光源的照射下，阴影分为自身阴影和投影阴影。自身阴影是由于物体自身的遮挡而使光线照射不到他的某些表面产生的阴影；投影阴影是由于不透明物体遮挡光线使得位于物体另一侧的区域受不到光照而形成的阴影。

绘制自身阴影与投影阴影图形的算法如下：

1. 根据视点原来的观察位置，对物体实施隐面算法，使用正常的光照模型计算光强来绘制可见表面。
2. 将视点移到光源位置。从光源处向物体所有背光面投影光线，建立光线的参数方程，计算该光线与投影面(地面)的交点，使用深灰色填充交点所构成的阴影多边形，形成投影阴影。若选用简单光照模型，对于背光面，由于得不到光源的直接照射，只有环境光对其光强有贡献。
3. **实现步骤**
4. 添加基础类与添加绘制立方体的CCube类。
5. 在CProjection类中进行透视变化。
6. 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。
7. 在CCube类中计算顶点坐标、读入面表，绘制地面，绘制立方体及其自身阴影，计算光线与地面的交点，并绘制投影阴影。
8. 在绘制图形时调用CZBuffer类中的Gouraud函数进着色。
9. 在CTestView类中调用CCube类的绘制函数。
10. 在CTestView类的OnDraw函数中调用DoubleBuffer函数，并添加消息响应函数。
11. **主要算法**

1. CCube类

public:

CCube();

virtual~CCube();

void ReadCubeVertex();//读入立方体顶点

void ReadCubeFace();//读入立方体面表

void ReadGroundVertex();//读入地面顶点

void DrawCube(CDC \*pDC, CZBuffer \*zbuffer);//绘制立方体

void DrawGround(CDC \*pDC);//绘制地面

void DrawShadow(CDC \*pDC, CZBuffer \*zbuffer);//绘制阴影

CP3d CalculateCrossPoint(CP3d p0, CP3d p1);//计算光线与地面的交点

public:

CP3d VCube[8];//立方体点表

CFace FCube[6];//立方体面表

CP3d VGround[4];//地面顶点表

int N1, N2;//N1为纬度区间,N2为经度区间

CLighting \*pLight;//光照环境

CMaterial \*pMaterial;//物体材质

CProjection projection;

CCube::CCube()

{

}

CCube::~CCube()

{

if (pLight != NULL)

{

delete pLight;

pLight = NULL;

}

if (pMaterial != NULL)

{

delete pMaterial;

pMaterial = NULL;

}

}

void CCube::ReadCubeVertex()//读入立方体顶点坐标

{

//顶点的三维坐标(x,y,z),立方体边长为2a

double a = 160;

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

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

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

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

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

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

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

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

}

void CCube::ReadCubeFace()//读入立方体面表

{

//面的顶点数、面的顶点索引号与面的颜色

FCube[0].SetNum(4); FCube[0].vI[0] = 4; FCube[0].vI[1] = 5; FCube[0].vI[2] = 6; FCube[0].vI[3] = 7;

FCube[1].SetNum(4); FCube[1].vI[0] = 0; FCube[1].vI[1] = 3; FCube[1].vI[2] = 2; FCube[1].vI[3] = 1;

FCube[2].SetNum(4); FCube[2].vI[0] = 0; FCube[2].vI[1] = 4; FCube[2].vI[2] = 7; FCube[2].vI[3] = 3;

FCube[3].SetNum(4); FCube[3].vI[0] = 1; FCube[3].vI[1] = 2; FCube[3].vI[2] = 6; FCube[3].vI[3] = 5;

FCube[4].SetNum(4); FCube[4].vI[0] = 2; FCube[4].vI[1] = 3; FCube[4].vI[2] = 7; FCube[4].vI[3] = 6;

FCube[5].SetNum(4); FCube[5].vI[0] = 0; FCube[5].vI[1] = 1; FCube[5].vI[2] = 5; FCube[5].vI[3] = 4;

}

void CCube::ReadGroundVertex()//读入地面点表

{

int Length = 800, Width = 800, Depth = 160;

VGround[0].x = -Length; VGround[0].y = -Depth; VGround[0].z = -Width; VGround[0].c = CRGB(0.5, 0.5, 0.5);

VGround[1].x = -Length; VGround[1].y = -Depth; VGround[1].z = Width; VGround[1].c = CRGB(0.5, 0.5, 0.5);

VGround[2].x = Length; VGround[2].y = -Depth; VGround[2].z = Width; VGround[2].c = CRGB(0.5, 0.5, 0.5);

VGround[3].x = Length; VGround[3].y = -Depth; VGround[3].z = -Width; VGround[3].c = CRGB(0.5, 0.5, 0.5);

}

void CCube::DrawCube(CDC \*pDC, CZBuffer \*zbuffer)//绘制立方体

{

CPi3 Point[4];//面的二维顶点数组

for (int nFace = 0; nFace < 6; nFace++)//面循环

{

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

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

FCube[nFace].SetFaceNormal(VCube[FCube[nFace].vI[0]], VCube[FCube[nFace].vI[1]], VCube[FCube[nFace].vI[2]]);

FCube[nFace].fNormal.Normalize();//单位化法矢量

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

{

for (int nVertex = 0; nVertex < FCube[nFace].vN; nVertex++)//顶点循环

{

projection.PerProject(VCube[FCube[nFace].vI[nVertex]]);//透视投影

Point[nVertex] = projection.ScreenP;

Point[nVertex].c = pLight->Illuminate(projection.ViewPoint, VCube[FCube[nFace].vI[nVertex]], FCube[nFace].fNormal, pMaterial);

}

zbuffer->SetPoint(Point, 4);//设置顶点

zbuffer->CreateBucket();//创建桶表

zbuffer->CreateEdge();//创建边表

zbuffer->Gouraud(pDC);//填充四边形

zbuffer->ClearMemory();

}

}

}

void CCube::DrawGround(CDC \*pDC)//绘制地面

{

CPi3 Point[4];//面的二维顶点数组

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

{

projection.PerProject(VGround[nVertex]);

Point[nVertex] = projection.ScreenP;

}

CBrush NewBrush;

NewBrush.CreateSolidBrush(RGB(100, 100, 100));

CBrush \*pOldBrush = pDC->SelectObject(&NewBrush);

pDC->BeginPath();

pDC->MoveTo(Round(Point[0].x), Point[0].y);//绘制左侧多边形

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

pDC->LineTo(Round(Point[i].x), Point[i].y);

pDC->LineTo(Round(Point[0].x), Point[0].y);

pDC->EndPath();

pDC->FillPath();//FillPath填充路径层

pDC->SelectObject(pOldBrush);

NewBrush.DeleteObject();

}

void CCube::DrawShadow(CDC \*pDC, CZBuffer \*zbuffer)//绘制阴影

{

CPi3 Point[4];//面的二维顶点数组

for (int nFace = 0; nFace < 6; nFace++)//遍历表面

{

//光源做为视点

CVector3 ViewVector(VCube[FCube[nFace].vI[0]], pLight->Light[0].L\_Position);//面的视矢量

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

FCube[nFace].SetFaceNormal(VCube[FCube[nFace].vI[0]], VCube[FCube[nFace].vI[1]], VCube[FCube[nFace].vI[2]]);

FCube[nFace].fNormal.Normalize();//单位化法矢量

if (Dot(ViewVector, FCube[nFace].fNormal) < 0)//绘制阴影

{

for (int nVertex = 0; nVertex < FCube[nFace].vN; nVertex++)//顶点循环

{

VCube[FCube[nFace].vI[nVertex]].c = CRGB(0.2, 0.2, 0.2);//阴影颜色

projection.PerProject(CalculateCrossPoint(pLight->Light[0].L\_Position, VCube[FCube[nFace].vI[nVertex]]));//计算该背光面上的顶点和光线连线和地面的交点

Point[nVertex] = projection.ScreenP;

}

zbuffer->SetPoint(Point, 4);//设置顶点

zbuffer->CreateBucket();//创建桶表

zbuffer->CreateEdge();//创建边表

zbuffer->Gouraud(pDC);//颜色渐变填充四边形

zbuffer->ClearMemory();

}

}

}

CP3d CCube::CalculateCrossPoint(CP3d p0, CP3d p1)//计算光线和地面的交点（第一个参数是光源，第二个参数是物体顶点）

{

CP3d p;

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

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

CVector3 VN = Cross(V01, V02);

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

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

double t; //计算直线参数方程的公共系数t

t = -(A\*p0.x + B \* p0.y + C \* p0.z + D) / (A\*(p1.x - p0.x) + B \* (p1.y - p0.y) + C \* (p1.z - p0.z));

p.x = p0.x + t \* (p1.x - p0.x);//代入参数方程计算交点坐标

p.y = p0.y + t \* (p1.y - p0.y);

p.z = p0.z + t \* (p1.z - p0.z);

p.c = CRGB(p1.c.red, p1.c.green, p1.c.blue);

return p;

}

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();

void Gouraud(CDC \*pDC);//填充

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

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

void SetPoint(CPi3 p[], 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

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;

pHeadE = NULL;

pCurrentB = NULL;

pEdge = NULL;

pCurrentE = NULL;

pHeadB = NULL;

zBuffer = NULL;

}

CZBuffer::~CZBuffer()

{

if (P != NULL)

{

delete[]P;

P = NULL;

}

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[], int m)

{

if (P != NULL)

{

delete[]P;

P = NULL;

}

P = new CPi3[m];

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

P[i] = p[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->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->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::Gouraud(CDC \*pDC)//填充多边形

{

double CurDeep = 0.0;//当前扫描线的深度

double DeepStep = 0.0;//当前扫描线随着x增长的深度步长

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;

DeepStep = -A / C;//计算扫描线深度步长增量

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->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;

}

}

CRGB ca, cb, cf;//ca、cb代边上任意点的颜色，cf代表面上任意点的颜色

ca = Interpolation(pCurrentB->ScanLine, pHeadE->ps.y, pHeadE->pe.y, pHeadE->ps.c, pHeadE->pe.c);

cb = Interpolation(pCurrentB->ScanLine, pHeadE->pNext->ps.y, pHeadE->pNext->pe.y, pHeadE->pNext->ps.c, pHeadE->pNext->pe.c);

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

double xleft, xright;//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)

{

xleft = pT1->x;

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

bInFlag = TRUE;

}

else

{

xright = pT1->x;

for (double x = xleft; x < xright; x++)//左闭右开

{

cf = Interpolation(x, xleft, xright, ca, cb);

if (CurDeep <= zBuffer[Round(x) + Width / 2][pCurrentB->ScanLine + Height / 2])//如果当前采样点的深度小于帧缓冲器中原采样点的深度

{

zBuffer[Round(x) + Width / 2][pCurrentB->ScanLine + Height / 2] = CurDeep;//使用当前采样点的深度更新深度缓冲器

pDC->SetPixelV(Round(x), pCurrentB->ScanLine, RGB(cf.red \* 255, cf.green \* 255, cf.blue \* 255));//绘制当前采样点

}

CurDeep += DeepStep;

}

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;

}

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;

}

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

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

CCube cube;

void CTestView::InitialLightingScene(void)

{

LightNum = 1;//光源个数

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

cube.pLight->Light[0].SetPosition(800, 800, 800);//设置光源位置坐标

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

{

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

cube.pLight->Light[i].L\_Specular = CRGB(1.0, 1.0, 1.0);//光源镜面高光颜色

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

cube.pLight->Light[i].L\_C1 = 0.0000001;//线性衰减系数

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

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

}

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

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

cube.pMaterial->SetDiffuse(CRGB(0.752, 0.606, 0.226));//材质对漫反射光的反射率

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

cube.pMaterial->SetEmit(CRGB(0.05, 0.05, 0.002));//材质自身发散的颜色

cube.pMaterial->M\_n = 30.0;//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)//绘制物体

{

CZBuffer \*zbuf = new CZBuffer;//申请深度缓冲内存

zbuf->InitDeepBuffer(1500, 1000, 1000);//初始化深度缓冲器

cube.DrawGround(pDC);

cube.DrawCube(pDC, zbuf);

cube.DrawShadow(pDC, zbuf);

delete zbuf;//释放内存

}

1. **实现效果**

简单阴影算法效果图如图48-1所示。

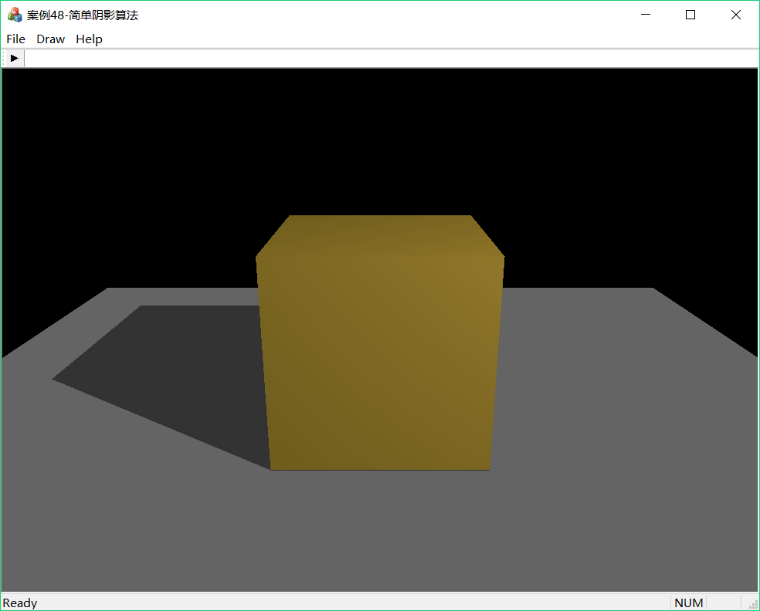


图48-1 简单阴影算法效果图