计算机图形学实验报告

NPR之卡通绘制小组

成员名单：贺心蕊、刘子渊、肖之屏（字母表顺序）

（一）选题原因

受到老师课上提到过的NPR的启发，看到了老师课上展示的静物速写NPR效果，我们觉得非常感兴趣，又询问过老师的意见，认为有一试的价值。一开始我们组初步草拟了三个题目：

1. 3D版Flappy Bird
2. 堆积木游戏
3. NPR非真实感绘制

老师审阅之后，给出了如下几条意见：

1. Flappy Bird的工作量会很大，效果不见得很好
2. 堆积木游戏的物理法则计算需要很多专业知识
3. NPR有实现的价值，可行性最强

综合以上因素，本组最终确定NPR为项目主题。

（二）项目构思

NPR，即non-photorealistic rendering，是上世纪90年代中期才开始逐渐成为研究热点的一个议题。真实感绘制一直是主流旋律。相比之下，非真实感的绘制其实还有很大的可发展余地。真实感绘制是追求像照片一样真实的绘制效果，而非真实感绘制则是反之。而事实上，很多时候，我们需要的恰恰是非真实感绘制，比如说3D动画，或者在宅界（动漫爱好者）流行的MMD（MikuMiku Dance，是一种使用各种人物模型制作的简易3D小短片，因为短，而且基本没有商用目的，反而容易见到各种非常炫酷的渲染效果）的渲染效果。

有学者指出，“真实感绘制”只是“非真实感绘制”的一种特殊形式。本组认为，任何形式的绘制都是在虚拟世界里用数据和函数架构出来的效果，就这一点上，非真实感绘制比真实感绘制具有更高层次上的抽象意义。

至于NPR的风格，就比较主流的而言，至少存在蜡笔风格、水墨画风格、白板风格、素描风格、水彩风格、卡通风格等几个不同的风格。

我们根据查得的资料进行了可行性和实用性的评估，认为，水墨画风格难度太大，几乎不可能做好（就像用计算机很难下好围棋一样，太灵活、太依赖感觉的东西就不好用本质上如此死板程序实现了）；水彩风格、蜡笔风格和白板风格实现难度不是特别大，但是应用范围不广，可扩展性不强；素描风格已经有许多研究者做出了比较好的效果，很难超越，而且，仍然是一种很难超越人类的技法（因为素描中的明暗对比，以及调子的上法等等实现细节都不是一成不变的）。

卡通渲染风格却不同。计算机的卡通渲染比人手工渲染的效果，从描边到上色都要更加均匀、细致。卡通渲染的应用范围也很广泛，3D动画，MMD，游戏，许多都用得上。在大部分可以使用真实感绘制的场景下，使用非真实感绘制不仅可行，而且还能取得特殊的效果。此外，我们发现，卡通效果的可扩展性比较强——形状，颜色，边线，等等，都可以有许多种截然不同的表达方式。一旦实现了一个框架，就像我们参加中期审查的时候所达到的程度那样，只需要修改小部分代码，加入几个函数，修改几个参数，修改几个函数，就可以达到截然不同、让人耳目一新的效果。

我们还发现卡通渲染当中也有许多不同的风格亚种。日式动漫的卡通渲染最为常见，我们的基本框架也是借鉴了日式动漫式渲染的算法。

我们小组的本意，是在卡通渲染的基础上，加入现代艺术（比如波普艺术、野兽派艺术）的元素——因为计算机图形学和传统意义上的艺术，有很多地方做着相同的事（都是使用一定的方法反映一个世界），它们本可以在更多的方面发生关联（比如互相借鉴），但是因为兼通二者的人才太少，使得这样的跨学科交流希望渺茫。

本小组的优势是在于成员有传统艺术方面知识比较渊博、艺术方面的专业素质较强的同学，也有编码能力过人、具有丰富的项目开发经验、组队合作经验丰富的同学，每个同学都愿意为这个项目奉献时间和精力。美中不足是全组的成员在参加图形学课程之前都没有相关方面的编程经验，许多设想不见得能够实现——最终也确实证明并未100%达到我们的心理预期，留下了一些遗憾。如果将来有时间更加深入掌握OpenGL这门编程语言，真正做到为我所用，我们将会想方设法完全实现出来我们最初的设想，了却这桩心愿。

（三）实现情况

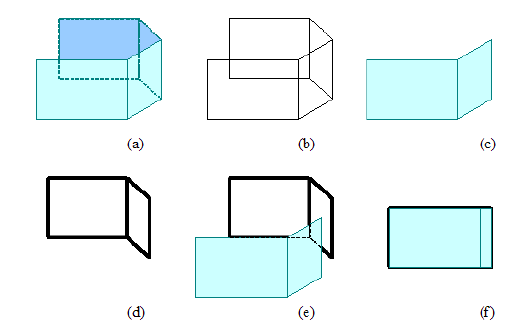
在期中之前，我们大致上实现了.obj文件的读取，日式卡通勾线算法和基本的单一色调日式卡通渲染。

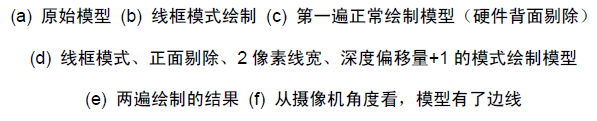
期末的时候，我们在期中的基础上实现了.mtl的读取，可以用卡通形式呈现物体的原有色调。此外，我们实现了类金属效果、热度图效果、带色调融合的原色渲染效果。

在期末之后，我们又实现了3D图形扭曲效果，和野兽派画风的贴图效果，更加接近最初的风格设想。涉及到以下的几种主要算法，以及一些自己编写的函数：

1. 描边算法

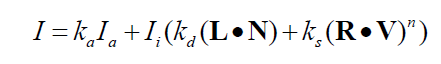
采用背面线框描边算法。





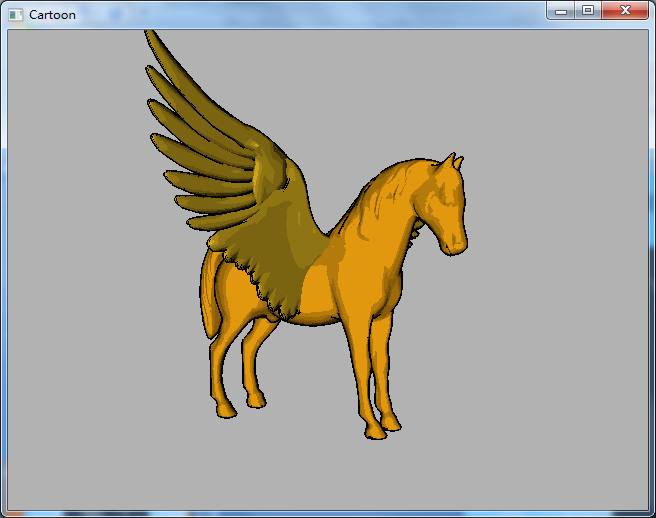
1. Cel-Shading明暗算法

Cel-shading 明暗算法是以Phong 局部光照模型为基础并加以改进得到的。在传统的Phong 局部光照模型中，反射光强度可以用以下算式表示：



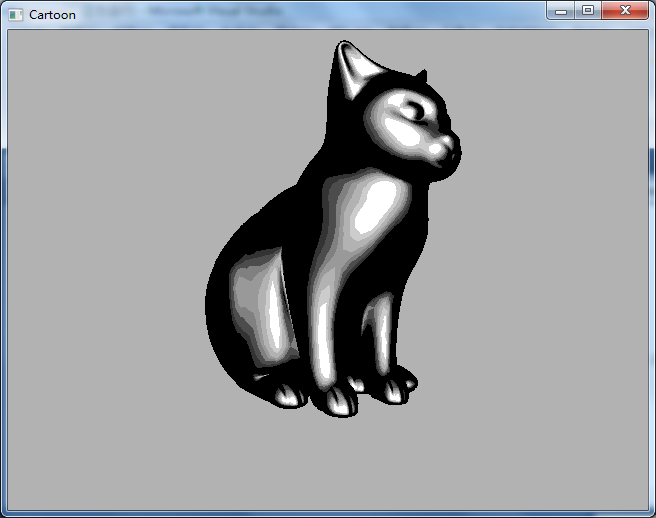
Cel-shading的主要思想是通过计算各个顶点法向量与光源方向的向量夹角（点乘）来决定改点的明暗程度，并且为了达到卡通渲染的效果，将这些值分段化，映射到32个不同明暗程度的区间上，以这些区间代表明暗程度的值最终决定颜色。

事实上这是个没有使用光源照明，直接将颜色计算出来“贴”到物体表面并且加以显示的过程。



1. 金属高光效果

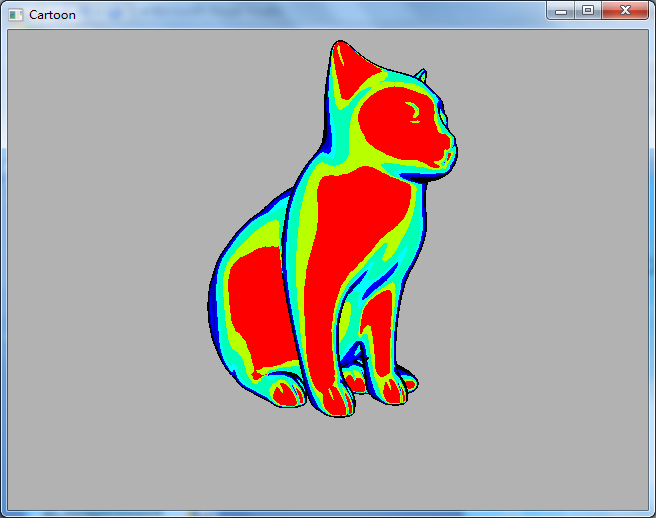
在光源的方向向量与面的法向量点乘，选出距离光源直射点较近区域，非高光区采取较高的阴影效果，高光区则是阴影效果与渐变shader的叠加。

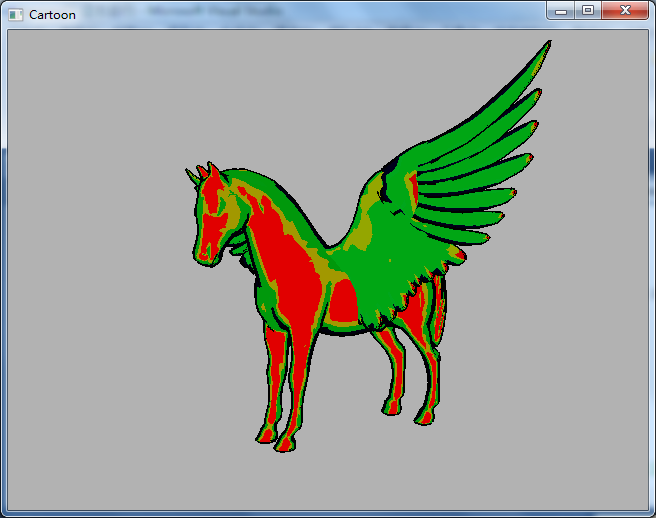


1. 热度图效果

将光源同时看做热源（通过调参也可以很容易地改热源的位置，但是尚未设置这项交互功能），被光源照射的强度越高红色越深，表示热度越高；背光源照射的强度越低蓝色越深，表示热度越低，形成热度由高到低颜色由红到绿再到蓝的渐变效果。

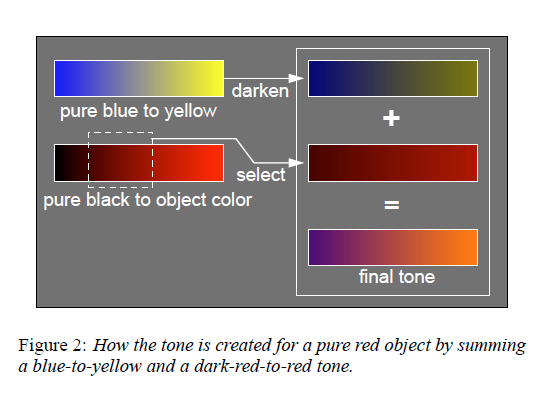
主要通过在shader中进行三个维度上的因变量控制在一定范围内的，自变量是法向量和光源向量的点乘乘积对应的shader里的值（即原本的“明暗程度”）的一元二次函数映射，形成渐变效果。在基础的Cel-Shading算法中添加函数映射即可。



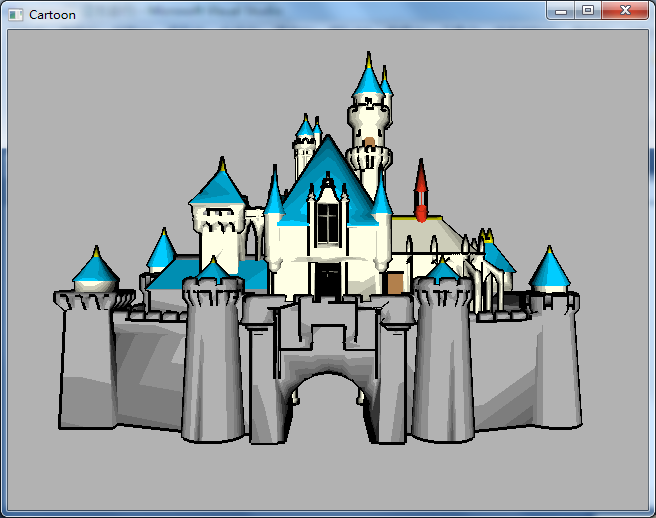
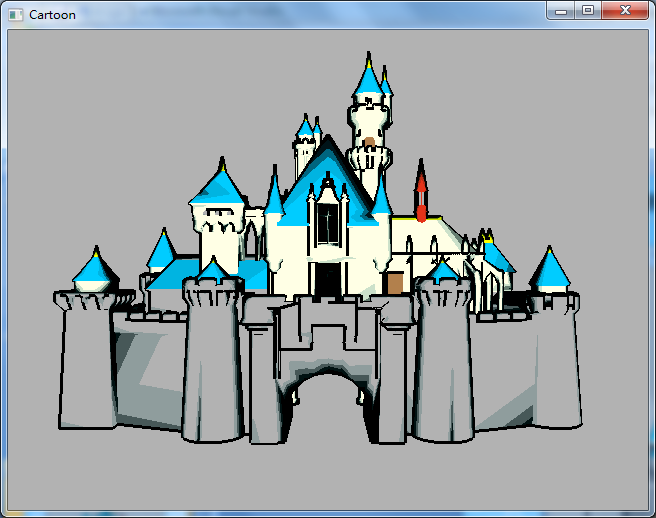


1. Tone-based Shading效果

Tone-based Shading效果在论文A Non-Photorealistic Lighting Model For Automatic Technical Illustration中提到。我们借鉴了这一项创意，但是没有使用他们的公式，而是自己编写了函数模型，参数也是自己调的。此外，论文中的Shading是连续变化的，而我们为了坚持卡通效果的基本立场，仍然采用分段明显的分明暗色块的效果。原理示意图如下：



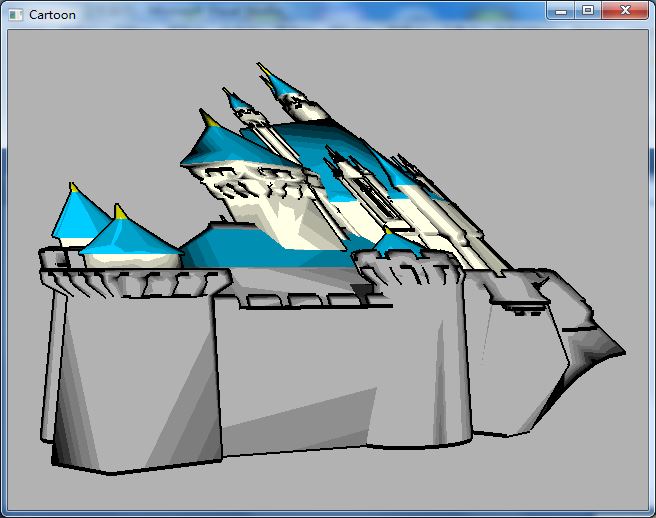
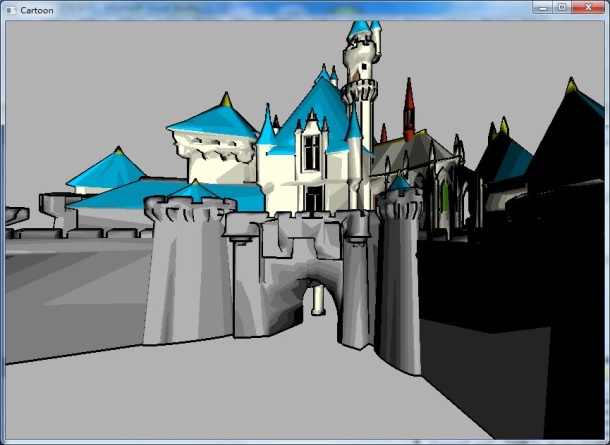
效果示意图如下：（左图为原卡通渲染效果，右为加了阴影：蓝、亮面：黄之后的渲染效果）

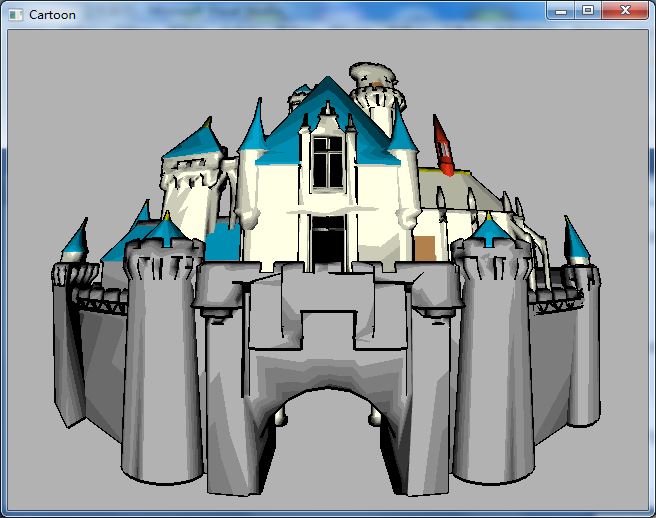
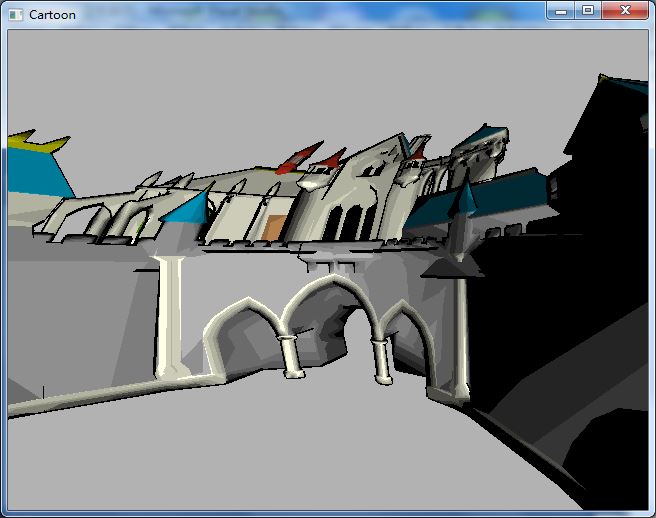
 

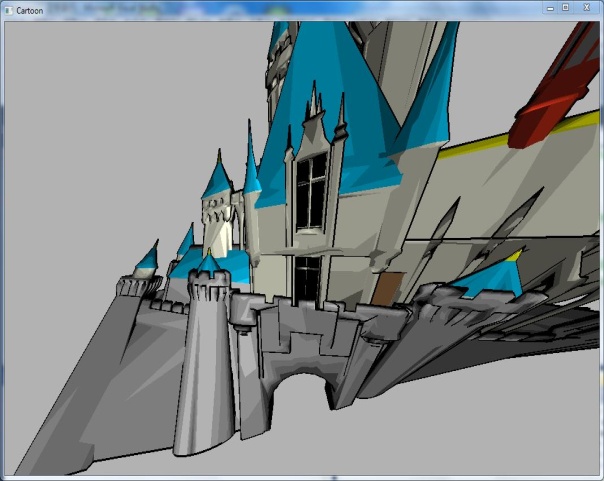
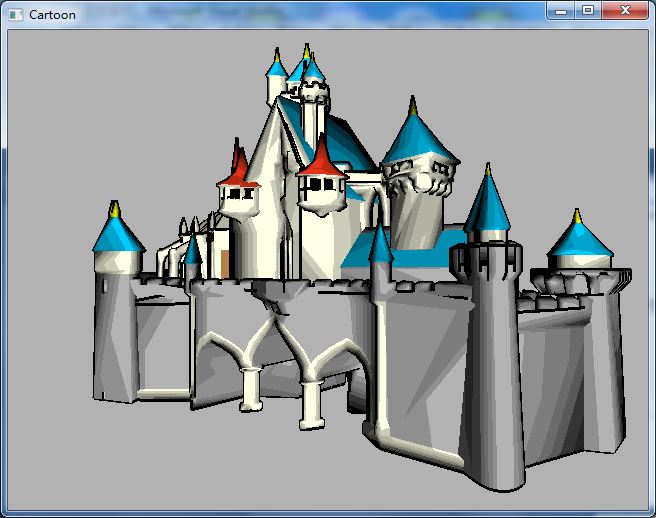
6） 三维模型扭曲变形效果

         将三维模型的顶点坐标依据不同曲面进行重新映射，相当于将模型不是投到平面成像，而是在特殊的曲面上进行三维呈现，达到整体上模型扭曲的效果。任意方向的扭曲变形相当于将给定矢量看为原来的Z轴方向，将扭曲后的原坐标系中的顶点坐标经过两次旋转得到新坐标，从而达到任意方向扭曲变形的整体效果。

以下分别为锥形变形效果，柱面变形效果，球面变形效果，马鞍面变形效果，波浪变形效果与扭曲效果。



说明：

成员分工

贺心蕊：原始obj模型渲染与初始边线生成；obj模型搜集；卡通渲染下的基于mtl文件的颜色设置；三维模型扭曲变形效果

刘子渊：obj文件读取与处理；改进的描边算法；Cel-shading明暗算法实现；金属高光效果

肖之屏：资料搜集；热度图效果；tone-based shading效果

（四）技术细节

1. 原始obj模型渲染与初始边线生成

功能：在进行卡通渲染之前进行原始obj模型与mtl模型的三维呈现，进行简单的边线显示。

代码实现：

全局变量：

typedef struct tagVECTOR{

float X, Y, Z; // Vector结构体

}

VECTOR;

VECTOR lightAngle; // 光线角度

GLfloat eye[3] = { 0.0, 0.0, 2.0 }; // 相机在世界坐标的位置

GLfloat at[3] = { 0.0, 0.0, 0.0 }; // 相机镜头对准的物体在世界坐标的位置

GLfloat up[3] = { 0.0, 1.0, 0.0 }; // 相机向上的方向在世界坐标中的方向

GLMmodel\* pmodel = NULL; //读取obj模型变量

GLdouble projection[16], modelview[16], inverse[16];

GLuint window, world, screen, command;

GLuint sub\_width = 256, sub\_height = 256;

static float angleX = 0.0, angleY = 0.0;

int rx = 0, ry = 0, rz = 0;

float px = 0, py = 0, pz = 0;

float scale = 1;

float lx = 1;

float shaderData[32][3];

bool outlineDraw = true; // 绘制外部轮廓的标志位

bool outlineSmooth = false; // 绘制反走样轮廓的标志位

float outlineColor[3] = { 0.0f, 0.0f, 0.0f }; // 设置轮廓线颜色

float outlineWidth = 3.0f; // 设置轮廓线宽度

GLfloat ambient[] = { 0.2, 0.2, 0.2, 1.0 }; // 材质的环境颜色

GLfloat diffuse[] = { 0.8, 0.8, 0.8, 1.0 }; // 材质的散射颜色

GLfloat specular[] = { 0.0, 0.0, 0.0, 1.0 }; // 材质的镜面反射颜色

GLfloat shininess = 65.0; // 镜面反射指数

GLfloat light\_position[]={ 1.0 , 1.0 , 1.0 , 1.0 }; // 光源位置

GLfloat light\_position2[] = {0.0, 0.0, 1.0, 1.0};

GLfloat shininess2 = 80.0;

GLuint shaderTexture[1];

函数及其功能

inline float Magnitude (VECTOR &V)

{

//功能：计算向量长度

}

void Normalize (VECTOR &V)

{

//功能：向量正规化

//计算向量长度，将向量各分量正规化

}

void inputKey(unsigned char key, int x1, int y1) {

//功能：获取键盘操作

//键盘左键：水平顺时针旋转

//键盘右键：水平逆时针旋转

//键盘上键：竖直顺时针旋转

//键盘下键：竖直逆时针旋转

// ’k’: 沿x轴移动

// ’h’: 沿-x轴移动

// ’u’: 沿y轴移动

// ’j’: 沿-y轴移动

// ’y’: 沿z轴移动

// ’i’: 沿-z轴移动

// ’n’: 按比例放大

// ’m’: 按比例缩小

// ’1’: 切换是否绘制边线

// ’2’: 切换是否进行反走样

// ’-’: 边线宽度变细

// ’k’: 边线宽度变粗

//窗口重新绘制

}

void reshape(int width, int height)

{

//功能： 窗口重新设置

glViewport(0, 0, width, height); // 设置视口

glMatrixMode(GL\_PROJECTION); // 对投影矩阵应用随后的矩阵操作

glLoadIdentity(); // 将当前的用户坐标系的原点移到了屏幕中心

gluPerspective(60.0, (float)width/height, 1.0, 8.0); // 指定视景体在世界坐标系中的具体大小

glGetDoublev(GL\_PROJECTION\_MATRIX, projection);

glMatrixMode(GL\_MODELVIEW); // 对模型视景矩阵堆栈应用随后的矩阵操作

glLoadIdentity();

gluLookAt(eye[0], eye[1], eye[2], at[0], at[1], at[2], up[0], up[1],up[2]);

glGetDoublev(GL\_MODELVIEW\_MATRIX, modelview);

glClearColor(0.2, 0.2, 0.2, 0.0);

glEnable(GL\_DEPTH\_TEST); // 启用深度测试

glEnable(GL\_LIGHTING); // 启用灯源

glEnable(GL\_LIGHT0); // 启用0号灯

}

void display(void)

{

// 功能：显示视图

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

if (!pmodel) {

pmodel = glmReadOBJ("obj/flyhorse.obj"); // 默认obj读取

if (!pmodel) exit(0);

glmUnitize(pmodel); // 单位化模型并返回缩放比例因子

glmFacetNormals(pmodel); // 计算模型面的法向

glmVertexNormals(pmodel, 90.0); // 计算模型的平滑顶点法向

}

if (outlineSmooth) // 检查我们是否想要反走样线条

{

glHint (GL\_LINE\_SMOOTH\_HINT, GL\_NICEST); // 启用它们

glEnable (GL\_LINE\_SMOOTH);

}

else // 否则不启用

glDisable (GL\_LINE\_SMOOTH);

glEnable (GL\_TEXTURE\_1D); // 启用1维纹理映射

glBindTexture (GL\_TEXTURE\_1D, shaderTexture[0]); // 绑定纹理

glColor3f (1.0f, 1.0f, 1.0f);

glEnable (GL\_LIGHTING);

light\_position[0]=lx;

glMaterialfv(GL\_FRONT, GL\_AMBIENT, ambient); // 指定用于光照计算的当前材质属性。

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, diffuse);

glMaterialfv(GL\_FRONT, GL\_SPECULAR, specular);

glMaterialf(GL\_FRONT, GL\_SHININESS, shininess);

if (outlineDraw) //检查允许边线绘制

{

glEnable (GL\_BLEND); //启用渲染

glBlendFunc(GL\_SRC\_ALPHA,GL\_ONE\_MINUS\_SRC\_ALPHA); // 设置渲染模型

glPolygonMode (GL\_BACK, GL\_LINE); // 背面多边形绘制为线条

glClear(GL\_COLOR\_BUFFER\_BIT);

glLineWidth (outlineWidth); // 设置线宽

glColor3f (1.0f, 1.0f, 1.0f);

glCullFace (GL\_FRONT); // 正面多边形不进行绘制 ( NEW )

glDepthFunc (GL\_LEQUAL); // 改变模型深度

glColor3fv (&outlineColor[0]);

glColor3f (1.0f, 1.0f, 1.0f); //设置边线颜色

light\_position[0]=lx;

glMaterialfv(GL\_FRONT, GL\_AMBIENT, ambient); // 指定用于光照计算的当前材质属性。

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, diffuse);

glMaterialfv(GL\_FRONT, GL\_SPECULAR, specular);

glMaterialf(GL\_FRONT, GL\_SHININESS, shininess);

glLightfv( GL\_LIGHT0 , GL\_POSITION , light\_position ); //设置光源

glPushMatrix();

glRotatef(angleX, rx, 1, rz); // 旋转矩阵

glRotatef(angleY, 1, ry, rz);

glTranslatef(px, py, pz); // 平移矩阵

glmDraw(pmodel, GLM\_SMOOTH | GLM\_MATERIAL);

glPopMatrix();

glDepthFunc (GL\_LESS);

glCullFace (GL\_BACK);

glPolygonMode (GL\_BACK, GL\_FILL);

glDisable (GL\_BLEND);

}

else //不绘制边线

{

light\_position[0]=lx;

glMaterialfv(GL\_FRONT, GL\_AMBIENT, ambient); // 指定用于光照计算的当前材质属性。

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, diffuse);

glMaterialfv(GL\_FRONT, GL\_SPECULAR, specular);

glMaterialf(GL\_FRONT, GL\_SHININESS, shininess);

glLightfv( GL\_LIGHT0 , GL\_POSITION , light\_position ); //设置光源

glPushMatrix();

glRotatef(angleX, rx, 1, rz); // 旋转矩阵

glRotatef(angleY, 1, ry, rz);

glTranslatef(px, py, pz); // 平移矩阵

glmDraw(pmodel, GLM\_SMOOTH | GLM\_MATERIAL);

glPopMatrix();

}

glDisable (GL\_TEXTURE\_1D);

glFlush ();

glutSwapBuffers();

}

void Load\_Obj(int value)

{

//功能：读取obj模型

//判断要读取的模型

//设定光照方向

//读取模型，面向量，法向量正规化

}

void redisplay\_all(void)

{

//功能：窗口重新绘制

}

主程序：

int main(int argc, char\*\* argv)

{

//功能：初始化

//打印操作说明

//进行模型、按键、菜单、纹理、光照等一系列初始化

}

2. 卡通渲染下的基于mtl文件的颜色设置

功能：在无颜色的卡通渲染下依据mtl文件中的内容进行颜色的绘制

代码实现：

全局变量：

char\* buf2 = new char [1000024]; //文件流缓冲区

class Matieral //记录mtl文件中的颜色

{

public:

char Name[100];

float color[3];

};

Matieral mtl[1000];

int mtl\_c = 0;

class mylog //记录obj文件中用到的颜色

{

public:

int num;

double color[3];

};

mylog Log[1000000];

int log\_c = 0;

函数及其功能：

GLvoid readMtlFile(FILE\* file)

{

//功能：读取mtl文件，提取颜色信息

//说明：mtl文件包括材料的各种信息。

1.环境反射Ka。

2.漫反射描述Kd

3.镜反射描述Ksb

4.滤光透射率Tf

5.光照模型描述

illum illum\_#

6.渐隐指数描述

d factor

7.反射指数描述

Ns exponent

8.清晰度描述

Sharpness value

9.折射值描述

// 扫描mtl文件，n开始的行进行新材料的记录

//提取漫反射参数作为颜色值记录。

}

GLvoid readObjFile(FILE\* file)

{

//功能：读取obj文件

//扫描文件时记录更换材料时需要的面元编号，更换的新颜色

}

void Draw (void)

{

//功能：模型绘制

//绘制多边形前检查面元编号，对于需要更换材料的情况进行颜色的重新设置。

}

BOOL ReadMesh ()

{

//功能：进行切换模型前每个文件的读取准备

//当前模型判断

//准备读取mtl文件

//准备读取txt文件

//准备读取obj文件

}

3.三维模型扭曲变形

功能：读取扭曲设置文件，依据文件要求进行三维模型的扭曲变形，相当于将模型不是投到平面成像，而是在特殊的曲面上进行三维呈现，达到整体上模型扭曲的效果。

全局变量：

char\* buf3 = new char [1000024];

bool Change[8] = {false, false, false, false, false, false, false, false};

/\*bool Change1 = false; //外凸锥面映射

bool Change2 = false; //内凹锥面映射

bool Change3 = false; //内凹柱面映射

bool Change4 = false; //内凹球面映射

bool Change5 = false; //正弦波浪映射

bool Change6 = false; //马鞍面映射

bool Change7 = false; //模型扭转映射

bool Change8 = false; //任意方向映射\*/

double Pos[3] = {1, 1, 1}; //映射方向

函数及其功能：

void Exchange()

{

if(Change[0]) //外凸锥面映射 Z = -sqrt(x \* x + y \* y)

{

vArray[vcnt].Pos.Z -= sqrt(vArray[vcnt].Pos.X \* vArray[vcnt].Pos.X + vArray[vcnt].Pos.Y \* vArray[vcnt].Pos.Y);

}

if(Change[1]) //内凹锥面映射 Z = sqrt(x \* x + y \* y)

{

vArray[vcnt].Pos.Z += sqrt(vArray[vcnt].Pos.X \* vArray[vcnt].Pos.X + vArray[vcnt].Pos.Y \* vArray[vcnt].Pos.Y) / 2;

}

if(Change[2]) //内凹柱面映射 Z = x \* x / 20 - 10

{

vArray[vcnt].Pos.Z -= 10 - vArray[vcnt].Pos.X \* vArray[vcnt].Pos.X / 20;

}

if(Change[3]) //内凹球面映射 Z = sqrt(35 \* 35 – x \* x – y \* y)

{

vArray[vcnt].Pos.Z += sqrt(35 \* 35 - vArray[vcnt].Pos.X \* vArray[vcnt].Pos.X - vArray[vcnt].Pos.Y \* vArray[vcnt].Pos.Y);

}

if(Change[4]) //正弦波浪映射 Z = 2 \* sin(x / 2)

{

vArray[vcnt].Pos.Z += 2 \* sin(vArray[vcnt].Pos.X / 2);

}

if(Change[5]) //马鞍面映射 Z = y \* y / 20 – x \* x / 20

{

vArray[vcnt].Pos.Z += - vArray[vcnt].Pos.X \* vArray[vcnt].Pos.X / 20 + vArray[vcnt].Pos.Y \* vArray[vcnt].Pos.Y / 20;

}

if(Change[6]) // 模型扭转映射

// z = - y / 2与z = y / 2沿x轴正负方向各拉开10单位后张成的曲面

{

double step = vArray[vcnt].Pos.Y \* 2 / 20;

vArray[vcnt].Pos.Z += vArray[vcnt].Pos.Y / 5 + (vArray[vcnt].Pos.X + 10 ) \* step;

}

if(Change[7]) //任意方向映射 将坐标系进行两次旋转（沿y轴与x轴），计算新的坐标，与之前的效果叠加使用

{

double s1 = Pos[0] / sqrt(Pos[0] \* Pos[0] + Pos[2] \* Pos[2]);

double c1 = Pos[2] / sqrt(Pos[0] \* Pos[0] + Pos[2] \* Pos[2]);

double s2 = Pos[1] / sqrt(Pos[0] \* Pos[0] + Pos[1] \* Pos[1] + Pos[2] \* Pos[2]);

double c2 = sqrt(Pos[0] \* Pos[0] + Pos[2] \* Pos[2]) / sqrt(Pos[0] \* Pos[0] + Pos[1] \* Pos[1] + Pos[2] \* Pos[2]);

double x0 = vArray[vcnt].Pos.X;

double y0 = vArray[vcnt].Pos.Y;

double z0 = vArray[vcnt].Pos.Z;

double x1 = c1 \* x0 + s1 \* z0;

double y1 = y0;

double z1 = -s1 \* x0 + c1 \* z0;

double x2 = x1;

double y2 = c2 \* y1 - s2 \* z1;

double z2 = s2 \* y1 + c2 \* z1;

vArray[vcnt].Pos.X = x2;

vArray[vcnt].Pos.Y = y2;

vArray[vcnt].Pos.Z = z2;

}

}

GLvoid readTxtFile(FILE\* file)

{

//功能：输入txt文本进行扭转功能的设置

//读入Change数组赋值

//读入旋转方向

}

BOOL ReadMesh ()

{

//功能：进行切换模型前每个文件的读取准备

//当前模型判断

//准备读取mtl文件

//准备读取txt文件

//准备读取obj文件

}

4.热度图效果

功能：以光源为热源，将模型渲染为类似热度图的颜色梯度效果

BOOL ChangeMode = false; // 是否要切换成热度图

// 如果ChangeMode改变，重新绘制

// 如果ChangeMode为true，则每次绘制都在确定明暗之后调用如下函数对模型颜色进行映射，得到热度图对应的RGB数据

void colorFunct ( float rangeX, float rangeY, float rangeZ, float tmp, float goal[3])

{

// 功能：给定一个原明暗程度，输出一个RGB数组，对应于热度图的对应颜色

// 参数：[rangeX, rangeY]是输入的明暗程度值得区间，rangeZ是RGB三个维度取值的最大值上限，tmp是输入的明暗值，goal是输出数组，代表求得的RGB值

float Red, Green, Blue; // 代表RGB表示法的各个值

// Blue

// Blue值在[RangeX, RangeY]前半区间（暗）值随tmp递增而连续平滑递减至0，在后半区间（亮）一直是0

// G

// Green值在[RangeX, RangeY]前半区间从0连续平滑递增至最大值，后半区间与前半区间对称，连续平滑递减至0

// R

// Red 值在前半区间一直是0， 后半区间连续平滑递增至最大值

goal[0] = Red;

goal[1] = Green;

goal[2] = Blue;

}

5.金属高光效果

功能：将模型渲染成类似于金属效果的类高光效果

BOOL GaoGuang = false; // 是否启用高光效果

// 如果GaoGuang的值改变，则重新绘制

// 在每次开始渲染之前，先进行如下判断

if(GaoGuang)

{

// TmpShade是明暗度值

if(TmpShade > 0.75f && TmpShade < 1.0f) // 介于某个区间之内

TmpShade = HighLight(TmpShade, 0.75);

else // 不是太亮就是太暗

TmpShade = 0; // 很高的阴影值

}

// 调用的主要函数

float HighLight(float t, float threshold)

{

// threshold是一个可以调节的参数

// 根据一定的公式进行映射，返回一个重新计算过的、介于原先的最亮与最暗之间的明暗度值，代替原先的TmpShader值参与之后的计算

// 因为这样的设定，所以这是一个可以和其他效果相互叠加的效果

}

6.带色调的明暗效果

功能：使物体的明与暗叠加上另外的两个相同或者不同的色调的渲染，可以是叠加任意的颜色，在此以明-黄，暗-青做例子进行说明（目前没有交互自由选择色调的功能，可以考虑优化界面并且添加此交互功能）

float DARKEN; // 用来使得颜色不要太亮的参数

float WEAKEN; // 用来使得明暗变化不要太明显的参数

BOOL ChangeMode2 = false; // 是否启用这个渲染模式

// 当ChangeMode2改变，重新渲染

// 当ChangeMode2值为true，在每次渲染的过程中先后执行以下过程：

// 1）色调基准

// 参数的含义和colorFunct是一致的，映射过程不一致

void colorFunct2 ( float rangeX, float rangeY, float rangeZ, float tmp, float goal[3])

{

float Red, Green, Blue;

float BW; // 用来调节明暗的参数

// 暗面色调的对应数据从最暗到最亮衰减至0，因变量连续平滑变化

// 亮面色调的对应数据从最亮到最暗衰减到0，因变量连续平滑变化

// 稍加调节，并且赋值：

goal[0] = Red / DARKEN + BW / WEAKEN;

goal[1] = Green / DARKEN + BW / WEAKEN;

goal[2] = Blue / DARKEN + BW / WEAKEN;

}

// 2）与原本的Toon-Shader颜色叠加

// 目前采用的是RGB数组存储的数据的线性加和

（五）未来展望

1. 关于最初的梦想

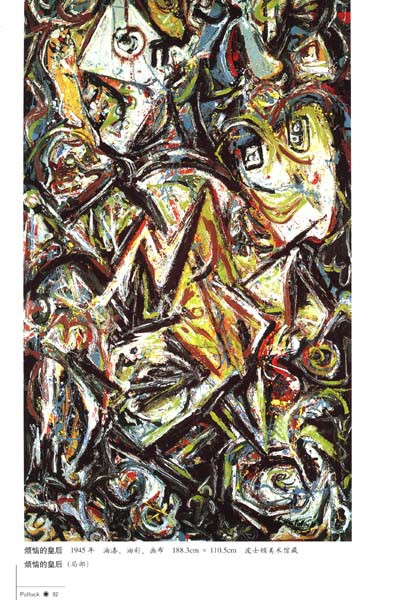
最初的梦想，是做出现代艺术风格的独特的渲染效果。

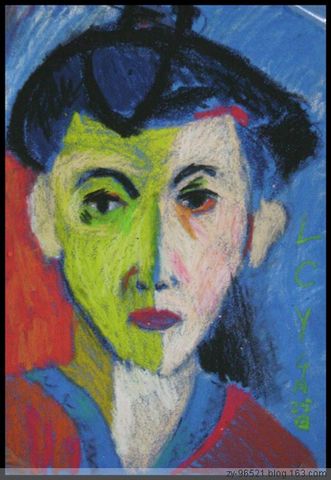
我们在当初的草案中提到了波普艺术、野兽派艺术，其实我们还产生过实现马赛克效果的想法。

最终，波普艺术得以初步实现（其实就是不同的变色调卡通渲染效果，其实要加上完善的人机交互、允许使用者灵活调整颜色才算是真正100%的完成）。



野兽派艺术因为没研究出高效、可靠的算法（主要问题是边线和色块，该如何计算“野兽化”之后的效果，我们研究了若干世界名画最终不得要领，老实说我们认为这方面的研究必须依靠于图形图像特征抓取、大数据分析和数学拟合）最终没能实现。

马赛克效果因为不知道该怎样设定每一个“格”的位置和形状，以及该如何确定“格”的大小，该怎样绘制格的间隙，该怎样定义格的间隙宽度，该怎样挪动“格子”网络……等等细节没有想通，兼之这种奇怪的风格似乎很没有市场（想不出它能用来干嘛），也就没有继续下去。

以上各个方案，还有其他一些更多的可能的方案，都不是不可能实现的，只是需要时间，需要人力物力而已。也许在不远的将来我们会想办法实现它们。

1. 关于高光

我们组的一大遗憾就是没有真正实现高光。一方面，事实上我们在网上找到了许多计算高光的算法，但是最后发现和我们的渲染流程很难结合到一起。另一方面，我们所尽的努力在我们实现的金属风格渲染上体现了出来。

高光不是一个特别容易实现的事情。首先就是关于各个材质在高光方面的特性不同，很难做到自动配比。这方面，在程序细节的实现上最好是以“面元组”为基本单位决定高光的性质，而且，必须设置用起来比较便捷的人机交互，使得用户可以轻易调节高光的柔和度、亮度、面积等等特性。

如果对所有物体的所有面都打上高光，其实效果不会太好，看起来像是被油泼过，或者被塑料纸蒙起来了的世界。

我们希望有机会的话实现高光，可是首先我们没有发现效果符合期望的高光算法。也许需要我们自己发明相关公式，也许需要我们补习光学。我们猜测物理世界里高光的产生大约是和全反射之类的现象有关系的，如果想要得到比较漂亮的高光效果，我们觉得也许研究物理公式是必须的。

仅仅研究光学恐怕仍然不够，因为我们是卡通主题的非真实感渲染，如果采取完全和物理世界的高光相同的处理，很可能效果不太理想。如果想要做出非常契合我们的风格的高光效果，公式还得自己琢磨。

以下是一些MMD中截出来的，高光起到了画龙点睛的作用的效果图。我们希望能加上这种效果的高光。自然、柔和，令整体画面更美观。







1. 其他的新奇点子

比如说，组员前几天在人人上发现的low-poly。根据我们查到的资料推测，low-poly（低多边形）是一种效果，一种风格，一般通过对许多2D、3D的图形图像处理软件的综合使用（涉及到了人类的智能）生成一个颜色诡异怪诞（也可以是正常颜色）的、物体外观形状多面体化的（看起来就像一个没雕刻完的雕塑）效果图。

我们认为，在3D环境下，我们只需要采取某种算法将若干个面元合并成一个大的面绘制出来即可，效果也许会不错。

只是我们在考虑用什么算法决定合并哪些面元的时候遇到了困难——我们不知道哪些面元合并之后，总体效果不会太“走形”，而且看起来还算均匀。尤其是当一个物体时对称的，它的“low-poly”合并效果也应该是关于原先的对称轴对称的，这一点，我们的算法该怎样保证呢？难道暴力搜索来取定所有对称轴吗？那时间成本会高到PC机承受不了的程度的。目前没有证据显示可以全自动完成一个高效美观的low-poly过程。

1. 数学方面

如果我们小组的项目还想要更进一步，目前这种假象效果，猜测公式，然后使用自编的低次（三次以下）简单方程进行映射、通过手工调参来优化效果的做法显然是太落伍了。不仅低效，而且有很多效果，比如“野兽派”渲染效果，因为这样的原始的工作方式，难以尝试出结果。

我们的团队至少需要一个强而有力的数学建模、数学拟合、数学分析工具，最好是能够找到一个强大的数院外援。最好这个外援本身有一点艺术直觉，而且也对我们的项目感兴趣。不过我们觉得，找这样一个完全对胃口的理想外援可能有点难度。也许我们需要自己学习数学来弥补没有外援的不足。

也许最后我们需要分析大数据，对大量的世界名画提取公共特征计算映射函数的参数，那时也许MapReduce或者Hadoop会被我们用起来（前提是我们会用，其实目前我们都还不会用）。

1. 人机交互

不可否认，我们的项目，最后的结果在人机交互上差了点意思。如果能够提升这个模块，肯定能大幅度提升我们项目的质量。至少是观赏质量，和使用时感觉到的质量。

我们有两个方向可以优化界面的用户友好度：

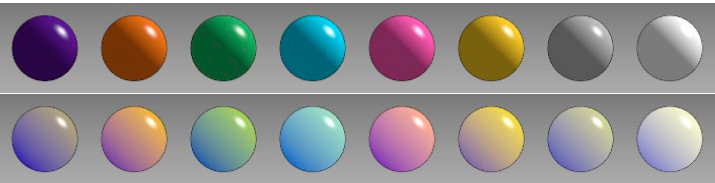
* 1. 加入右键下拉菜单
  2. 加入菜单栏
  3. 直接感知鼠标点击的图元所在区域，以这种直观的方式交互

第三种交互方式明显实现难度最大，debug难度最大，对程序框架的改动也是最大的，但是如果能够实现，效果将会最好。因为这种交互方式更加符合人类的直觉，使用最为自然。

前两个方式其实难度都不是很大，我们找到的一份代码，OBJ LOADER当中就使用了右键下拉菜单的交互方式。我们也做了结果来讲并不成功的插入菜单的尝试，因为网上很难找到可靠的详细介绍如何在OpenGL的窗口建立右键菜单的教程，我们暂时搁置这个想法。

因为对WinAPI不是很熟悉，而且不知道从何学起，再加上“这是锦上添花的细枝末节”这一判断，全组成员都把有限的时间投入到了不同的绘制效果开发当中去了。

目前是使用各种快捷键来交互，如果能够添加这样的一排按钮，来决定Tone-Based Shading的不同色调组合，显然效果会更好。



1. 关于野兽派的折中方案

本来原定计划是将一些世界名画（主要是野兽派的）作为贴图贴到模型表面，以此作为野兽派计划的一个折中方案。

此方案经本小组讨论认为相当具有可行性，事实上已经加以实施，只是尚未完成，所以并没有提交半成品版本。其完工只是时间的问题。

这个方案在期末之后才加以实施，本来时间尚有余裕，只是负责这一模块的刘子渊同学身在外地，网络状况很差，文件很难传过去，之后还集中地发生了一系列诸如手机停机、手机被小偷偷走、发生车祸的小概率事件……

当在医院终于缝完针、修养过来的刘子渊同学向全组报告他的情况的时候，很明显已经来不及了。所以本组决定，这个方案以后有机会再完成。

（六）参阅资料

我们的资料大多来源于网络。那时候google还没有2014年6月这么瘫痪，学术文献的搜索还是一个比较愉悦的过程。

在那个时候我们找到了以下这些主要资料：

1. 文献类
   1. OpenGL Shading Language Course/Chapter 1 – Introduction to GLSL——By Jacobo Rodriguez Villar （[jacobo.Rodriguez@typhoonlabs.com](mailto:jacobo.Rodriguez@typhoonlabs.com)）
   2. NPR - Implementaci´on en 3D utilizando OpenGL

Mart´ın Massera y Marcela Nievas

Facultad de Ciencias Exactas y Naturales — Universidad de Buenos Aires — ARGENTINA.

mmassera@dc.uba.ar (54)(011) 4783-1479 y mn6s@dc.uba.ar (54)(011) 4716-6016

* 1. A Non-Photorealistic Lighting Model For Automatic Technical Illustration

Amy Gooch Bruce Gooch Peter Shirley Elaine Cohen

Department of Computer Science

University of Utah

<http://www.cs.utah.edu/>

* 1. Real-Time Nonphotorealistic Rendering

Lee Markosian Michael A. Kowalski Samuel J. Trychin Lubomir D. Bourdev

Daniel Goldstein John F. Hughes

Brown University site of the

NSF Science and Technology Center for

Computer Graphics and Scientific Visualization

Providence, RI 02912

* 1. 非真实感绘制技术研究

苏延辉,韦欢,费广正,石民勇

(中国传媒大学动画学院,北京100024)

* 1. 本科毕业论文（设计）题目：非真实感渲染技术研究

计算机科学系2004 级杨志（20040532）指导教师：金汉钧

二○○ 八年四月

* 1. 日式动画风格非真实感三维实时渲染算法的研究

乐大山 龙晓苑 汪国平

北京大学机器感知与智能教育部重点实验室

北京大学多媒体与人机交互实验室

北京100871

对于其中一部分资料我们参考了其中提到的算法，对于另一部分资料我们采纳了他们的创意。在此对所有资料的著作者们表示由衷的感谢。

1. 代码类

虽然大部分代码并未被借鉴，甚至因为与本小组的final project相差很远，本小组因为某些最后证明没有借鉴价值的代码的存在而将很多时间花在了对项目的完成其实没有帮助的工作上，我们还是要对这些代码的编写者们表示由衷的感谢。

这些代码（或者包含可运行的opengl代码的教程）基本上都能在CSDN下载频道得到，本小组即是如此获得的。

* 1. OpenGL教程\_Nehe（中文版，比较详细地指导初学者使用openl完成各项复杂功能的教程，内含可运行的代码以及非常详细的逐段解说）
  2. OBJ LOADER（很普通的用于从.obj和.mtl里面读取数据生成图像的代码，不包含读取贴图文件的功能，而且几乎不含注释，来源不详，可以很轻易从CSDN上下载到）
  3. Cg（从CSDN上下载得到，主旨为教导初学者使用CG库的一系列简单代码，package的名字就叫cg，里面包含使用CG库需要的.lib, .dll, .cpp, .h等一系列文件，还有一个文件夹OpenGL，里面包含了许多可运行的使用CG的Opnegl代码）