图形学Project 说明文档

Project 1 编程实现音乐节奏可视化

参考文献

本PJ参考CSDN上一位大哥的可视化项目，由本人魔改，主要使用了html5 audio api以及canvas工具。

参考文献地址：

https://blog.csdn.net/twoByte/article/details/62043425

https://blog.csdn.net/towrabbit/article/details/82991557

算法原理

如何将音乐节奏可视化？这里考虑用到音乐本身的频谱数据，我们将音乐理解成一种音波，那可以用傅里叶变换，将其转换到幅度域，我们记录其每一秒的频谱数据，随着音乐旋律改变，频谱数据也相应改变，那么，就可以借助频谱数据，将音乐的节奏动态地呈现出来。

因而，我们要做的事情主要有三点，一是**分析并存储音频数据**，二是**将数据转换为可见的图形**，三是**实时更新数据并将其反应在图形上。**

为满足上述条件，这里我们用到了html5 audio api，该api可以存储音频文件的各类数据，并提供工具分析音频的实时信息；而音频绘画我们使用canvas，将一定范围内的频谱转化成一个个微形的尖峰，辅以动感的线条。综上所述，我们就可以实现音乐节奏的可视化。

细节将在下方辅以源代码一同说明。

程序说明

文件结构

├─ audioVisualizer.html #主运行程序，构设文件组成

├─ audioiVisualizer.js #程序运行的主要逻辑

├─ background.jpg #demo的背景图片

├─ README.markdown #PJ1的运行注意事项

├─ 短裙-AOA.flac #可视化所用的音乐

代码细节

- **初始化**

```
//变量初设
var
AudioContext,context,source,analyser,p,penBg,dataArray,gradient,gradient
Right;
//获取屏幕长度与宽度，与屏幕适配
var screenHeight = document.body.clientHeight,
    screenWidth = document.body.clientWidth;
//获取canvas对象的长度与宽度
var width = canvas.width,
    height = canvas.height;
//手动设置需要读入的音乐文件
var name = "短裙-AOA.flac";
//初始化播放器
var audio = new Audio(name);
    audio.charset = 'utf-8';
```

- **设置播放互动按钮逻辑**

```
audio.oncanplaythrough = function() {
    loader.innerHTML = '<div>点我</div>';
    document.addEventListener('click',function(){
        //初始化
        init();
        //隐藏播放器
        loader.style.display = 'none'
        //显示歌名
        song_name.style.display = 'block'
        var sn = document.getElementById("song_name");
        sn.innerHTML = '<div id="song_name">'+ name.split('.')[0] +
'</div>';
        //播放曲目
        audio.play();
        //绘制动画
        draw();
        //为辐射层设置透明度
        penBg.globalAlpha = 0.2;
        document.removeEventListener('click',arguments.callee);
    })
};
```

- **各图形元素的初始化**

值得注意的是，我们需要将音频信息传递给图形绘画的工具，由此，需要创建一些Web Audio的元素，并将它们进行连接。它们分别是AudioContext，JS Node，Buffer Source以及Analyser。我们并不关注其中的实现细节，而着眼于这些工具带来的效果，结果上述操作，我么便可以借助分析器analyser得到音频的实时信息。

与analyser直接连接的是源缓冲，源缓冲再与destination连接，destination的实质就是音频输出，只有将其连接，才能真正听到声音。

```
function init(){
    AudioContext = AudioContext || webkitAudioContext;
    context = new AudioContext;
    //构建 source → analyser → destination的连接
    source = context.createMediaElementSource(audio);
    analyser = context.createAnalyser();
```

```
    source.connect(analyser);
    analyser.connect(context.destination);
    //主体与背景
    p = canvas.getContext("2d");
    penBg = bg.getContext("2d");
    //设置快速傅里叶变换的值，反应在音频频谱的密度
    analyser.fftSize = 4096;
    var length = analyser.fftSize;
    //创建数据
    dataArray = new Uint8Array(length);
    //设置可视化图形的渐变色
    gradient = p.createLinearGradient(0, 100, 480, 100);
    gradient.addColorStop("0", "#ff0055");
    gradient.addColorStop("1.0", "#ceaf11");
}
```

- 图形的绘画

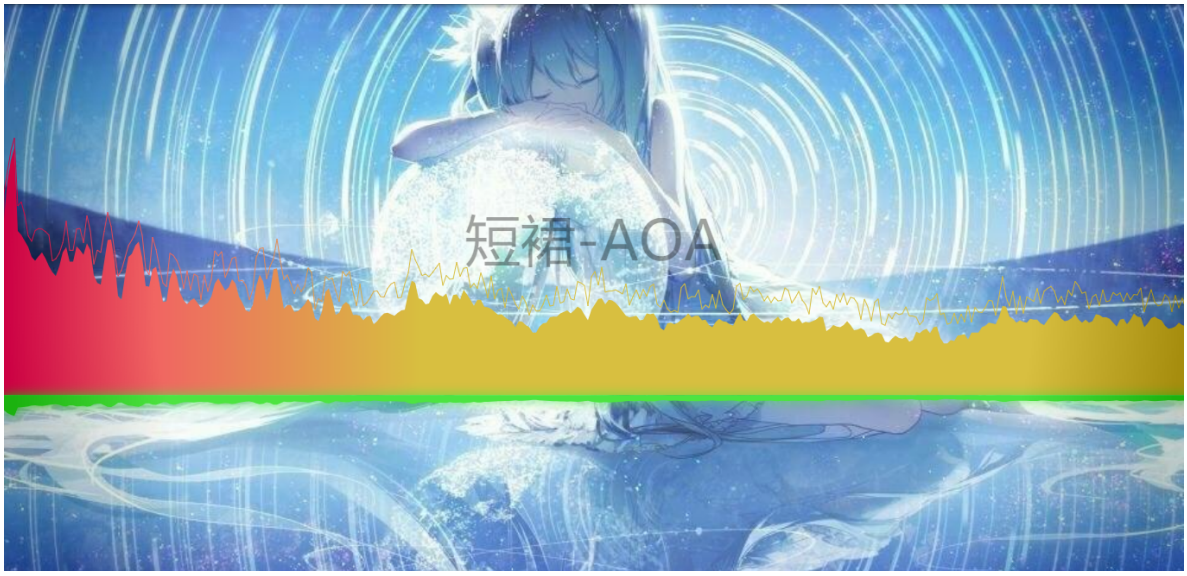
  我们的图形化界面主要由三块组成：填充部分、线条部分以及倒影部分，在背景我们还设置了一个放射性的动态圆圈背景，增加动感，该部分的代码就是这四块内容的绘画。

  在中间，还加入了频谱密度与中轴线位置的可变参数，使项目的呈现更加灵活。

```
function draw() {
    requestAnimationFrame(draw)
    analyser.getByteFrequencyData(dataArray);
    //清空背景
    p.clearRect(0, 0, width, height);
    //制作放射性背景，设定为一个圆圈
    var gradientBg = penBg.createRadialGradient(width / 2, height / 2,
height - Math.floor(Math.random() * 150 + 100), width / 2, height / 2,
width / 2);
    gradientBg.addColorStop(0, "white");
    gradientBg.addColorStop(1, '#000');

    //绘画可动的部分
    penBg.clearRect(0, 0, width, height);
    penBg.fillStyle = gradientBg;
    penBg.fillRect(0, 0, width, height);

    //设置频谱密集程度
     var m = 5;

     //设置中轴线位置
     var n = height / 3;

    //绘画可动的填充部分
    p.beginPath();
    p.moveTo(0, height - n);
    var x = 0;
    for (var i = 1; i < width; i++) {
        var lineHeight = dataArray[i] / 256 * height / 3;
        if (i < m) {
            p.lineTo(x, height - dataArray[i] / 256 * height / 2 - n)
        } else if (i > width - m) {
            p.lineTo(x - m, height - n)
        } else {
            p.lineTo(x, height - lineHeight - n)
```

```
        }
        x += m - 1;
    }
    p.fillStyle = gradient;
    p.fill();
    p.closePath();

    //绘制可动的线条部分
    p.beginPath();
    p.moveTo(0, height - n);
    var x = 0;
    for (var i = 1; i < width; i++) {
        var lineHeight = dataArray[i] / 256 * height / 3;
        if (i < m) {
            p.lineTo(x, height - dataArray[i] / 256 * height / 2 - n -
10 - Math.floor(Math.random() * 30))
        } else if (i > width - m) {
            p.lineTo(x - m, height - n - 20)
        } else {
            p.lineTo(x, height - lineHeight - n - 10 -
Math.floor(Math.random() * 30))
        }
        x += m - 1;
    }
    p.strokeStyle = gradient;
    p.stroke();
    p.closePath();

    //绘制下方的倒影
    p.beginPath();
    p.moveTo(0, height - n);
    var x = 0;
    for (var i = 1; i < width; i++) {
        var lineHeight = dataArray[i] / 256 * height / 40;
        if (i < m) {
            p.lineTo(x, dataArray[i] / 256 * height / 24 + height - n)
        } else p.lineTo(x, lineHeight + height - n)
        x += m - 1;
    }
    p.lineTo(x - m, height - n)
    p.fillStyle = '#21dd13';
    p.shadowBlur = 20;
    p.shadowColor = "#21dd13";
    p.fill();
    p.closePath();
    p.shadowBlur = 0;
}
```

效果图



Project 2 编程画一个真实感静态景物——象棋棋子

参考文献

本Project选择使用OpenGL绘画，OpenGL是很经典的图形学绘画工具，网上有许多应用OpenGL绘画的实例，张江图书馆也有许多OpenGL的应用书籍。

此处提到的opengl是基于glut库实现的，故看的教程也大都是使用glut库的。

参考资料：

https://www.cnblogs.com/OctoptusLian/p/6872726.html

OpenGL超级宝典（中文版）

OpenGL编程指南（中文第8版）

算法原理

我们使用的OpenGL是基于C++实现的，因此其也有面向对象的特性，在OpenGL中，预存储有一些已经构建好的图形，我们在这里使用这些图形，构建我们想要的物品——棋子。

OpenGL还有许多复杂的部分，包括光照与纹理。

- 光照

光源有许多种类，包括环境光，点光源，平型光与聚光灯。由于我们要画的是一颗象棋棋子，这里环境光。环境光是一种无处不在的光，其光线来源于四面八方，也因此无论物体的法向量如何，都将表现为同样的明暗程度。

将光源细分，又可分为漫射光与平行光两个部分，其中，漫射光指光源中能够被漫反射的逛的颜色成分，平行光指光源中所有能被镜面反射的逛的颜色成分。

OpenGL可以为我们提供8个有效的光源，依次为GL_LIGHT0，GL_LIGHT1 .....我们可以为GL_LIGHT0指定环境光的成分。

我们调用glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight)函数设置环境光。第一个参数指定设置的光源，第二个参数指明设定环境光，第三个参数是一个数组，拥有四个元素，表示光源中的红、绿、蓝三种光先的成分以及透明度。

我们调用glLightfv(GL_LIGHT0, GL_DIFFUSE, DiffuseLight)函数设置漫射光。参数设置基本与环境光相同，DiffuseLight指漫射光成分，也同环境光类似。

我们调用glLightfv(GL_LIGHT0, GL_SPECULAR, SpecularLight)函数设置镜面光，其中的参数设置同上述两种光线一样，SpecularLight是漫射光的颜色成分。

很多时候，我们需要指定光源的位置来产生需要的效果。其方法仍然是调用glLightfv函数，不过需要换换参数：glLightfv(GL_LIGHT0, GL_POSITION, LightPosition)。LightPosition也是一个四维数组，四维数组的前3项依次为光源位置的X,Y,Z分量，第四个值很特殊，一般为1或-1。当LightPosition[4]=-1的时候，表示光源位于距离场景无限远的地方，无论前面设置的X,Y,Z是什么值。当LightPosition[4]=1时，光源的位置就是前三项所指定的位置。

- 纹理

OpenGL可以用材质贴图来实现纹理功能，需要使用glEnable(GL_TEXTURE_2D)开启2D纹理功能，使用glDisable(GL_TEXTURE_2D)关闭纹理。

我们需要使用函数glGenTextures（n,&textureID）来分配n个纹理编号，使用glTexParameteri来设置常用的4个纹理参数，这些参数包括了当前纹理图像大小/小于模型目标时扩展纹理的处理方式。

我们通过gluPerspective和gluLookAt设置“视角”和“观察点”的相关参数。

具体的使用将在下方辅以代码说明。

程序说明

文件结构

├─ Chess #项目主体文件夹

├─ Debug #debug用可执行文件

├─ ChaHu.sln #项目文件

├─ iron.bmp #材质贴图

代码细节

- 绘制棋子

```
void Chess() {

    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, texture[0]);

    glPushMatrix();
    //圆球
    glTranslatef(0.0, 0.8, 0.0);
    glutSolidSphere(0.3, 40, 10);

    //小环
    glTranslatef(0.0, -0.25, 0.0);
    glRotatef(90.0, 1.0, 0.0, 0.0);
    glutSolidTorus(0.05, 0.2, 20.0, 30.0);

    //圆锥体
    glRotatef(180.0, 1.0, 0.0, 0.0);
    glTranslatef(0.0, 0.0, -1.0);
    glutSolidCone(0.4, 1.4, 20, 6);
```

```
    //大环
    glutSolidTorus(0.07, 0.4, 20.0, 20.0);
    glPopMatrix();

    glDisable(GL_TEXTURE_2D);
}
```

- 加载纹理

```
//读纹理图片
unsigned char *LoadBitmapFile(char *filename, BITMAPINFOHEADER
*bitmapInfoHeader)
{
    FILE *filePtr;
    BITMAPFILEHEADER bitmapFileHeader;
    unsigned char    *bitmapImage;
    int    imageIdx = 0;
    unsigned char    tempRGB;
    filePtr = fopen(filename, "rb");
    if (filePtr == NULL) {
        printf("file not open\n");
        return NULL;
    }
    fread(&bitmapFileHeader, sizeof(BITMAPFILEHEADER), 1, filePtr);
    if (bitmapFileHeader.bfType != BITMAP_ID) {
        fprintf(stderr, "Error in LoadBitmapFile: the file is not a
bitmap file\n");
        return NULL;
    }
    fread(bitmapInfoHeader, sizeof(BITMAPINFOHEADER), 1, filePtr);
    fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);
    bitmapImage = new unsigned char[bitmapInfoHeader->biSizeImage];
    if (!bitmapImage) {
        fprintf(stderr, "Error in LoadBitmapFile: memory error\n");
        return NULL;
    }

    fread(bitmapImage, 1, bitmapInfoHeader->biSizeImage, filePtr);
    if (bitmapImage == NULL) {
        fprintf(stderr, "Error in LoadBitmapFile: memory error\n");
        return NULL;
    }
    for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage;
imageIdx += 3) {
        tempRGB = bitmapImage[imageIdx];
        bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
        bitmapImage[imageIdx + 2] = tempRGB;
    }
    fclose(filePtr);
    return bitmapImage;
}

//加载纹理的函数
void texload(int i, char *filename)
{
    BITMAPINFOHEADER bitmapInfoHeader;
    unsigned char*   bitmapData;
```

```
    bitmapData = LoadBitmapFile(filename, &bitmapInfoHeader);
    glBindTexture(GL_TEXTURE_2D, texture[i]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

    glTexImage2D(GL_TEXTURE_2D,
        0,          //mipmap层次(通常为，表示最上层)
        GL_RGB,     //该纹理有红、绿、蓝数据
        bitmapInfoHeader.biWidth, //纹理宽带，必须是n，若有边框+2
        bitmapInfoHeader.biHeight, //纹理高度，必须是n，若有边框+2
        0, //边框(0=无边框, 1=有边框)
        GL_RGB,     //bitmap数据的格式
        GL_UNSIGNED_BYTE, //每个颜色数据的类型
        bitmapData);    //bitmap数据指针
}

//初始化纹理
void textureInit() {
    glGenTextures(TEXTURE_INT, texture);
    const char* csc = "E://FDU//COLLEGE//2019-
2020.1//TuXing//PJ2//ChaHu//iron.bmp";
    char* cc;
    int length = strlen(csc);
    cc = new char[length + 1];
    strcpy(cc, csc);
    texload(0, cc);
    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1); //设置像素存储模式控制所读取的图像
数据的行对齐方式.
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);//
放大过滤，线性过滤
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);//
缩小过滤，线性过滤
}
```

- 设置灯光与投影

```
void init(void)
{
    GLfloat AmbientLight[] = { 0.0, 0.0, 0.0, 1.0 };  //环境光
    GLfloat Light_Model_Ambient[] = { 0.5, 0.5, 0.5, 1.0 }; //环境光参数

    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };  //镜面反射参数
    GLfloat mat_shininess[] = { 50.0 };               //高光指数

    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 }; //设定灯光的位置

    GLfloat white_light[] = { 1.0, 1.0, 1.0, 1.0 };   //灯位置(1,1,1), 最
后1-开关
    GLfloat blue_light[] = { 0.0, 0.0, 1.0, 1.0 };
    GLfloat red_light[] = { 1.0, 0.0, 1.0, 1.0 };
    GLfloat green_light[] = { 0.0, 1.0, 0.0, 1.0 };

    glClearColor(0.08, 0.0, 0.0, 0.0);  //背景色
    glShadeModel(GL_SMOOTH);           //多变性填充模式
```

```
    //材质属性
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);


    //灯光设置
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_AMBIENT, AmbientLight);   //环境光
    glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);   //散射光属性
    glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);  //镜面反射光
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, Light_Model_Ambient);  //环境
光参数

    glEnable(GL_LIGHTING);   //开关:使用光
    glEnable(GL_LIGHT0);     //打开0#灯
    glEnable(GL_DEPTH_TEST); //打开深度测试
}

void reshape(int w, int h)
{
    glViewport(0.5, 0, (GLsizei)w, (GLsizei)h);

    //设置投影参数
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    //正交投影
    if (w <= h)
        glOrtho(-1.5, 1.5, -1.5*(GLfloat)h / (GLfloat)w, 1.5*(GLfloat)h
/ (GLfloat)w, -10.0, 10.0);
    else
        glOrtho(-1.5*(GLfloat)w / (GLfloat)h, 1.5*(GLfloat)w /
(GLfloat)h, -1.5, 1.5, -10.0, 10.0);

    //设置模型参数--几何体参数
    glMatrixMode(GL_MODELVIEW);

    glLoadIdentity();
}
```

- 设置参数

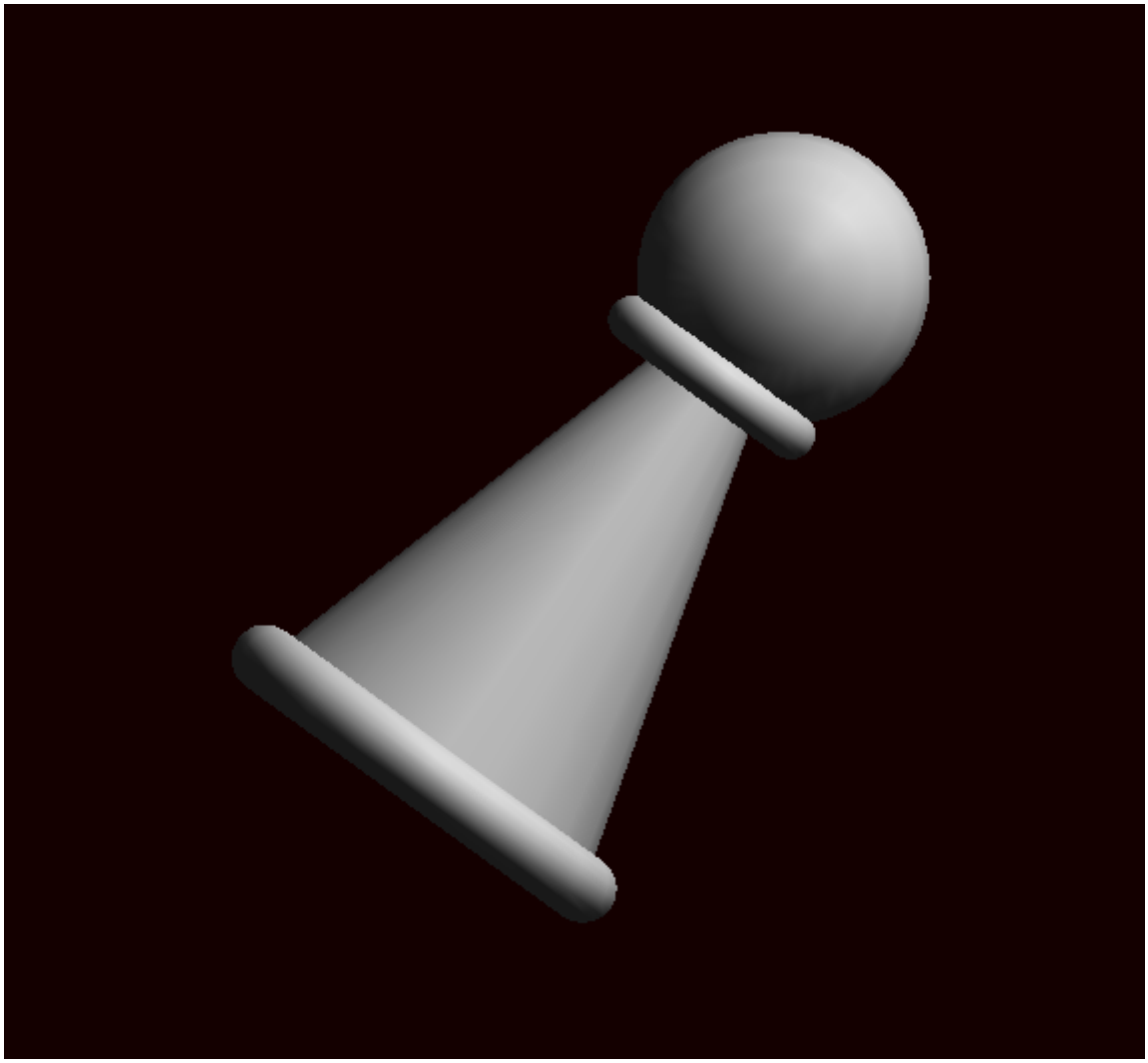
```
void init(void)
{
    GLfloat AmbientLight[] = { 0.0, 0.0, 0.0, 1.0 };  //环境光
    GLfloat Light_Model_Ambient[] = { 0.5, 0.5, 0.5, 1.0 }; //环境光参数

    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };  //镜面反射参数
    GLfloat mat_shininess[] = { 50.0 };               //高光指数

    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 }; //设定灯光的位置

    GLfloat white_light[] = { 1.0, 1.0, 1.0, 1.0 };   //灯位置(1,1,1), 最
后1-开关
    GLfloat blue_light[] = { 0.0, 0.0, 1.0, 1.0 };
    GLfloat red_light[] = { 1.0, 0.0, 1.0, 1.0 };
    GLfloat green_light[] = { 0.0, 1.0, 0.0, 1.0 };
```

```
    glClearColor(0.08, 0.0, 0.0, 0.0);  //背景色
    glShadeModel(GL_SMOOTH);            //多变性填充模式

    //材质属性
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);


    //灯光设置
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_AMBIENT, AmbientLight);   //环境光
    glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);   //散射光属性
    glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);  //镜面反射光
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, Light_Model_Ambient);  //环境
光参数

    glEnable(GL_LIGHTING);   //开关:使用光
    glEnable(GL_LIGHT0);     //打开0#灯
    glEnable(GL_DEPTH_TEST); //打开深度测试
}
```

- 主函数

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(700, 700);
    glutInitWindowPosition(50, 50);
    glutCreateWindow("一枚oyyt的象棋棋子");
    textureInit();
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(display);
    glutMainLoop();
    return 0;
}
```

效果图



Project 3 创作一个Flash动画

参考文献

这里参看的官方提供的说明文档和网上的一众教程，诸多繁杂，这里便不列出了。

算法原理

该PJ采用的是现有的工具，这里便掠过了。

程序说明

需要创作的是一个flash动画，老师提到，要画一个有趣的动画，我想到平常常看的美妆视频，便有了制作一个涂口红的小动画的想法，并将其实现。

详情请看实际的动画，这里截取一张效果图：