

计算机图形学

Computer Graphics

OpenGL 中的纹理映射

福州大学数计学院 软件工程系 陈昱

目的

- 学习如何在 OpenGL 中使用纹理贴图功能，包括：
 - 纹理的定义
 - 映射方式的控制
 - 纹理坐标的指定
 - 纹理渲染的控制
 - 纹理对象的使用

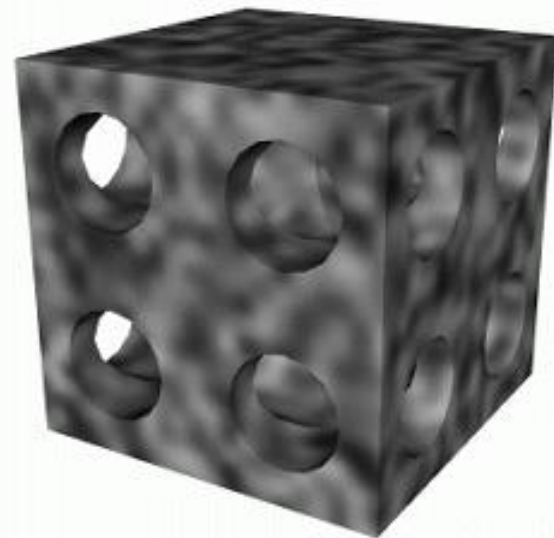
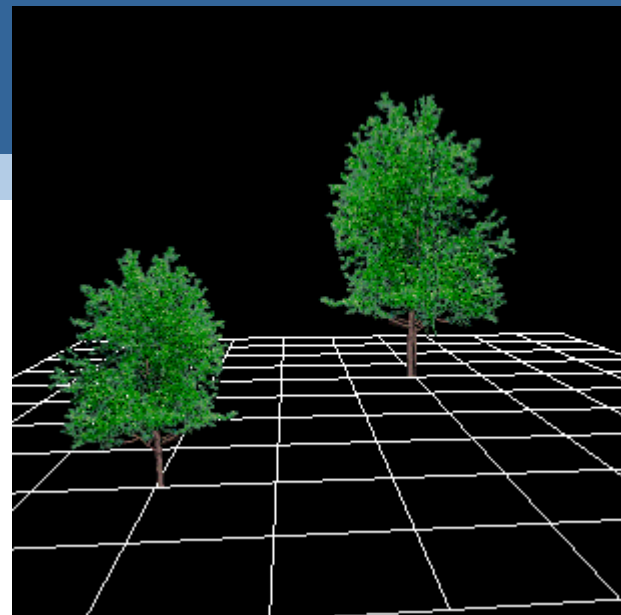
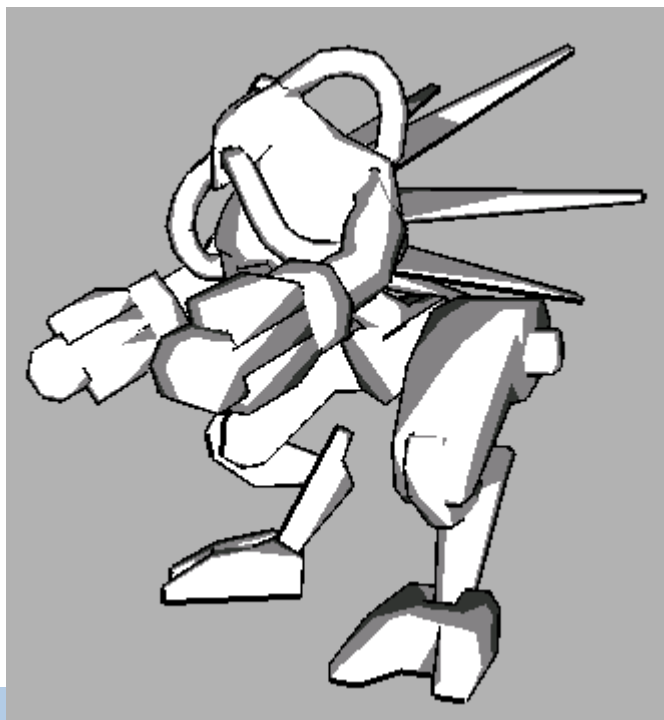
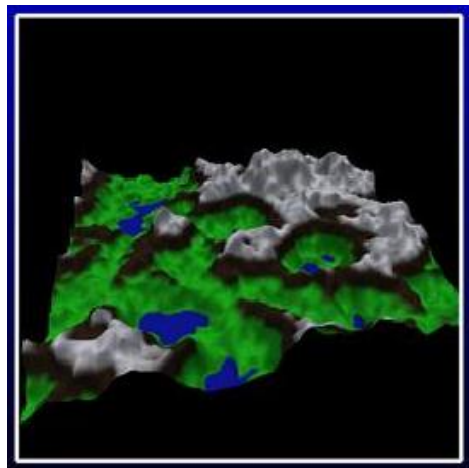
纹理贴图的步骤

1. 创建纹理对象，并为它指定一个纹理
 2. 确定纹理如何应用到每个像素上
 3. 启动纹理贴图功能
 4. 绘制场景，提供纹理坐标和几何坐标
- 注意：纹理坐标必须在 RGBA 模式下才可以使用，在颜色索引模式下使用纹理的结果是不能预测的

纹理定义

定义纹理

- `glTexImage1D()`
- `glTexImage2D()`
- `glTexImage3D()`



二维纹理的定义

- glTexImage2D (GLenum target, GLint level, GLint components, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels)
 - target : GL_TEXTURE_2D
 - level : LOD number (0: base image)
 - components : number of color components (1|2|3|4)

二维纹理的定义

– Format : 图像数据的格式 GL_RGB

– Type : 图像数据的类型
GL_UNSIGNED_BYTE

– Border: 0 or 1

– Width & height

• $2^m + 2$ (border bit)

• w 和 h 可以不同

一些新的 OpenGL 扩展已不受这个限制

其他维数的纹理

- 除了定义二维纹理外，OpenGL 还支持一维纹理和三维纹理：
- glTexImage1D
 - GL_TEXTURE_1D
- glTexImage3D
 - GL_TEXTURE_3D

映射控制

映射方式的控制

- 之前的课程中提到过，光照和材质影响多边形上最终各个点的颜色
- 那么，纹理和材质能否同时使用呢？
- 可以。OpenGL允许你用 `glTexEnv` 命令来设置两者如何结合以决定最终的颜色

设定纹理环境

- `glTexEnv{if}{v}(GLenum target, GLenum pname, TYPEparam);`
- 设置纹理如何与着色相互作用:
 - Target : `GL_TEXTURE_ENV`
 - Pname: `GL_TEXTURE_ENV_MODE`
 - Param: modes
(`REPLACE|DECAL|MODULATE|BLEND`)

Texture Environment Modes

- GL_REPLACE
- GL_DECAL
- GL_MODULATE (default)
- GL_BLEND

新的环境模式：

- GL_ADD: $C_v = C_f + C_t$
- GL_COMBINE (ARB, see [here](#))

GL_REPLACE

- 表面颜色由纹理唯一决定



RGB Polygon: (1,0,0)

| Base Internal Format | Replace Texture Function |
|----------------------|--------------------------|
| GL_ALPHA | $C = C_f$ $A = A_t$ |
| GL_LUMINANCE | $C = L_t$ $A = A_f$ |
| GL_LUMINANCE_ALPHA | $C = L_t$ $A = A_t$ |
| GL_INTENSITY | $C = I_t$ $A = I_t$ |
| GL_RGB | $C = C_t$ $A = A_f$ |
| GL_RGBA | $C = C_t$ $A = A_t$ |

用纹理替代

GL_MODULATE

- 光照会影响到纹理的显示



| Base Internal Format | Modulate Texture Function |
|----------------------|--------------------------------|
| GL_ALPHA | $C = C_f$ $A = A_f A_t$ |
| GL_LUMINANCE | $C = C_f L_t$ $A = A_f$ |
| GL_LUMINANCE_ALPHA | $C = C_f L_t$ $A = A_f A_t$ |
| GL_INTENSITY | $C = C_f I_t$ $A = A_f I_t$ |
| GL_RGB | $C = C_f C_t$ $A = A_f$ |
| GL_RGBA | $C = C_f C_t$ $A = A_f A_t$ |

GL_BLEND



Use with:

`glTexEnvfv`

`(GL_TEXTURE_ENV,`

`GL_TEXTURE_ENV_COLOR,`

`colorPtr)`

Base Internal Format

`GL_ALPHA`

`GL_LUMINANCE`

`GL_LUMINANCE_ALPHA`

`GL_INTENSITY`

`GL_RGB`

`GL_RGBA`

Blend Texture Function

$$C = C_f$$
$$A = A_f A_t$$

$$C = C_f(1 - L_t) + C_c L_t$$
$$A = A_f$$

$$C = C_f(1 - L_t) + C_c L_t$$
$$A = A_f A_t$$

$$C = C_f(1 - I_t) + C_c I_t$$
$$A = A_f(1 - I_t) + A_c I_t$$

$$C = C_f(1 - C_t) + C_c C_t$$
$$A = A_f$$

$$C = C_f(1 - C_t) + C_c C_t$$
$$A = A_f A_t$$

C_c : texture environment color

GL_DECAL

decal : 贴花



RGB Polygon: (1,0,0)
Tree: (r,g,b,**0**)

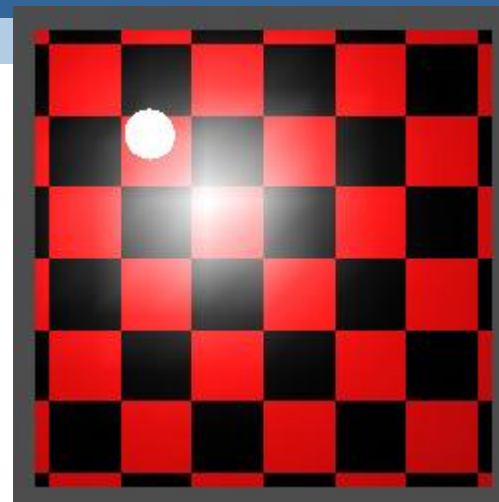
| Base Internal Format | Decal Texture Function |
|----------------------|---|
| GL_ALPHA | undefined |
| GL_LUMINANCE | undefined |
| GL_LUMINANCE_ALPHA | undefined |
| GL_INTENSITY | undefined |
| GL_RGB | $C = C_t$ $A = A_f$ |
| GL_RGBA | $C = C_f(1 - A_t) + C_t A_t$ $A = A_f$ |

Cp: replace

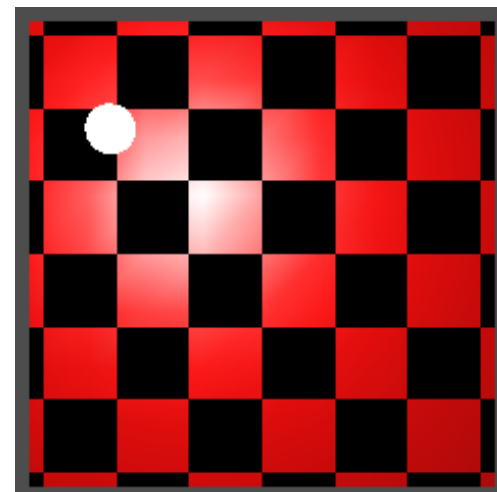
RGB:
DECAL=REPLACE

纹理 + 光照

- 使用 GL_MODULATE , 能显示出表面的颜色
- 在纹理映射之后应用镜面颜色:
 - glLightModeli (GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
 - 见之前的课件



GL_SEPARATE_SPECULAR_COLOR



GL_SINGLE_COLOR

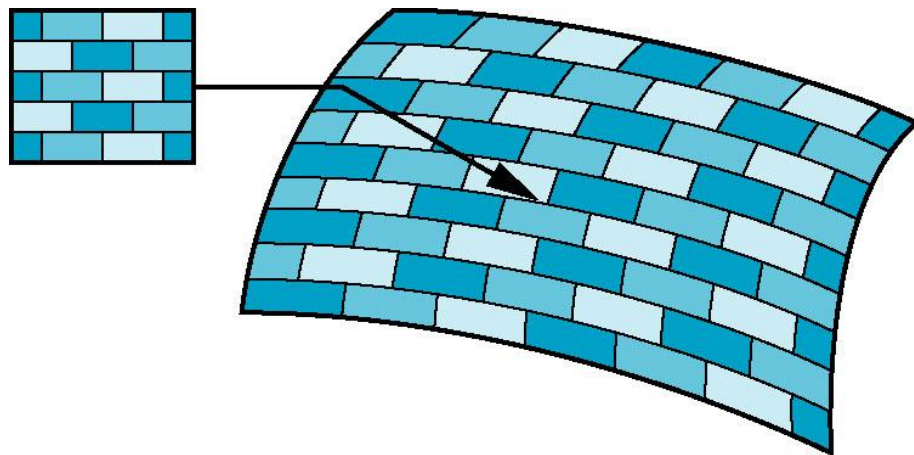
编译注意

- 注意，上面的代码中的 `GL_LIGHT_MODEL_COLOR_CONTROL` 和 `GL_SEPARATE_SPECULAR_COLOR` 常量是在 OpenGL 1.2 时加入的
- VC6~2008 所附带的 `gl.h` 只支持 OpenGL 1.1，因此无法编译通过
- 解决办法：
 - 使用 GLEW (The OpenGL Extension Wrangler Library) 提供的 `glew.h`，在包含 `gl.h` 之前先包含 `glew.h`

纹理坐标

什么是纹理映射

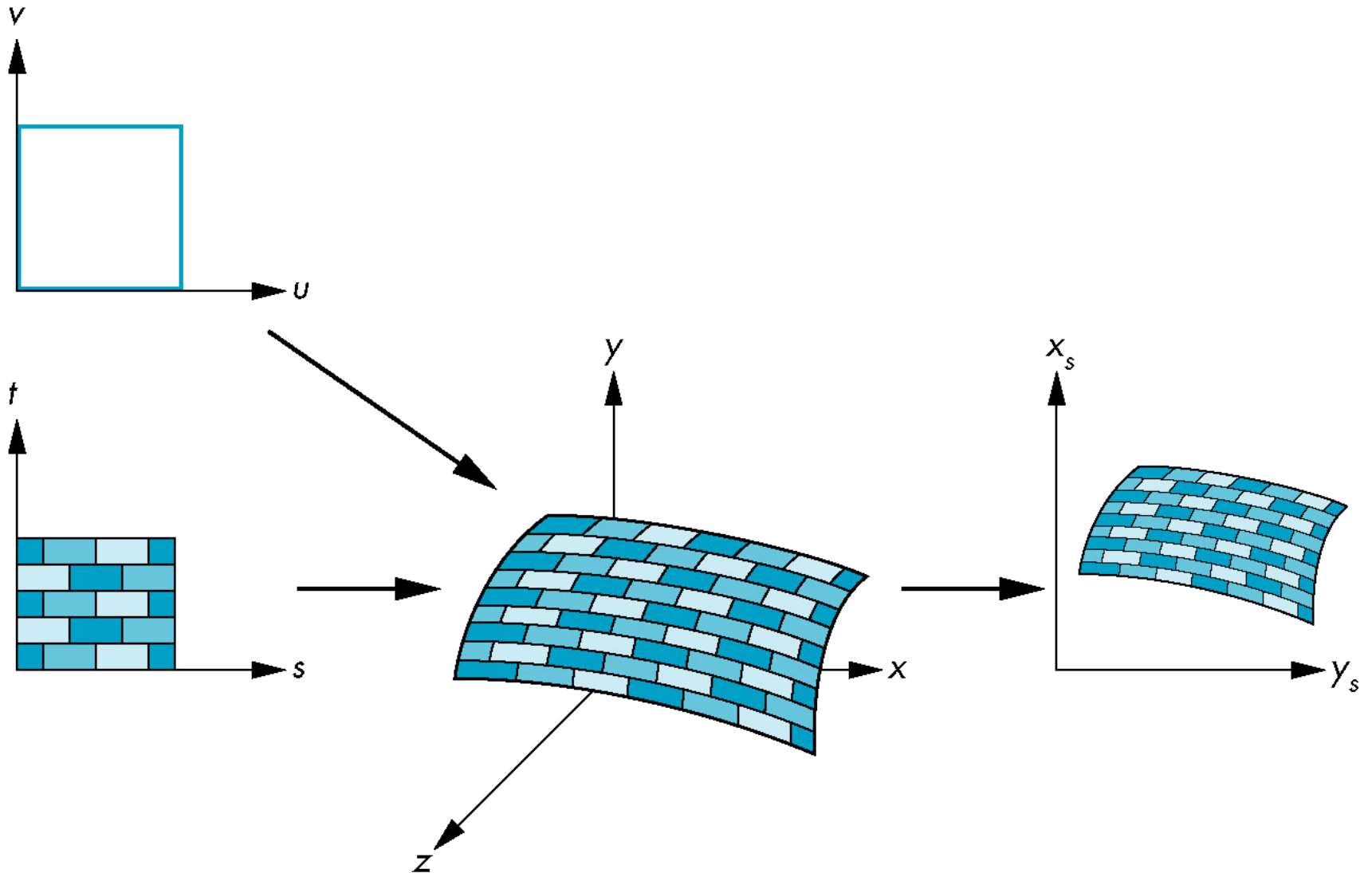
- 给定一个模型和一个 2D 纹理图像，映射图像到模型上的过程



什么是纹理映射

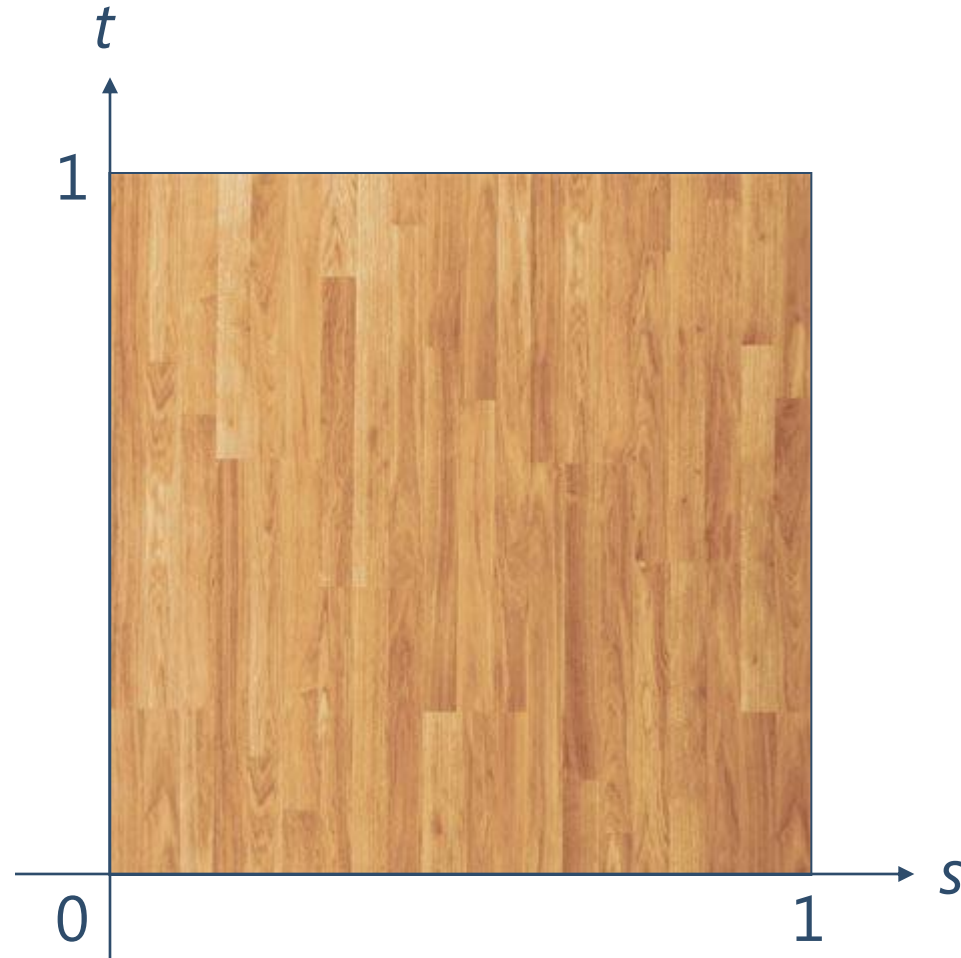
- 用一个函数映射模型上的一个点到图像上的坐标，这个函数称为表面映射函数
- 当对模型上的一个点着色的时候，我们从 2D texture 上查找合适的像素，用它来决定最终的着色颜色

什么是纹理映射



Texture Space (纹理空间)

- 我们用一个坐标系来指定一张纹理图上像素的地址，此坐标系统称为纹理空间
- 纹理图上的像素又称为 texel (texture element)



纹理坐标

- 纹理坐标:
 - 在多边形定义中将纹理的位置(在纹理空间上)和顶点指定在一起
- `glTexCoord2i (s, t);` `glVertex2i (x, y);`
- 顺序不能够对调!
 - [TexCoord 也是 OpenGL 中的状态量, 指定后不修改就一直有效]

二次曲面 (Quadrics) 的纹理坐标

- OpenGL 中绝大多数二次曲面对象拥有默认的纹理坐标设定
- 打开默认值:
 - `gluQuadricTexture(qobj, GL_TRUE)`

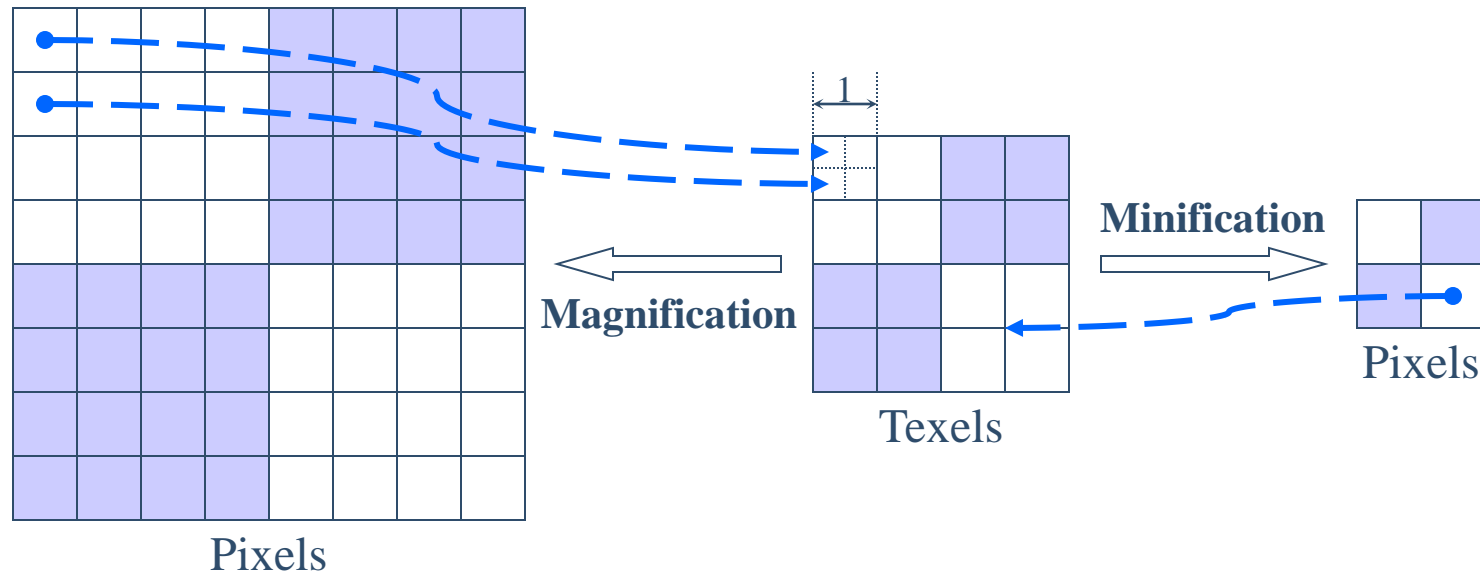


纹理坐标的自动生成

- 在某些场合（环境映射等）下，为获得特殊效果需要自动产生纹理坐标，并不要求为用函数`glTexCoord*()`为每个物体顶点赋予纹理坐标值
- OpenGL 提供了自动产生纹理坐标的函数：
- `void glTexGen{if}[v](GLenum coord, GLenum pname, TYPE param);`

纹理控制

Magnification & Minification



- 放大称为 Magnification
- 缩小称为 Minification
- 结果： pixel 和 texel 之间不是一一对应

Magnification Filter Options

- `glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER,
GL_NEAREST);`
- Magnification Filter Options
 - `GL_NEAREST` 最邻近滤波
 - `GL_LINEAR` 双线性滤波

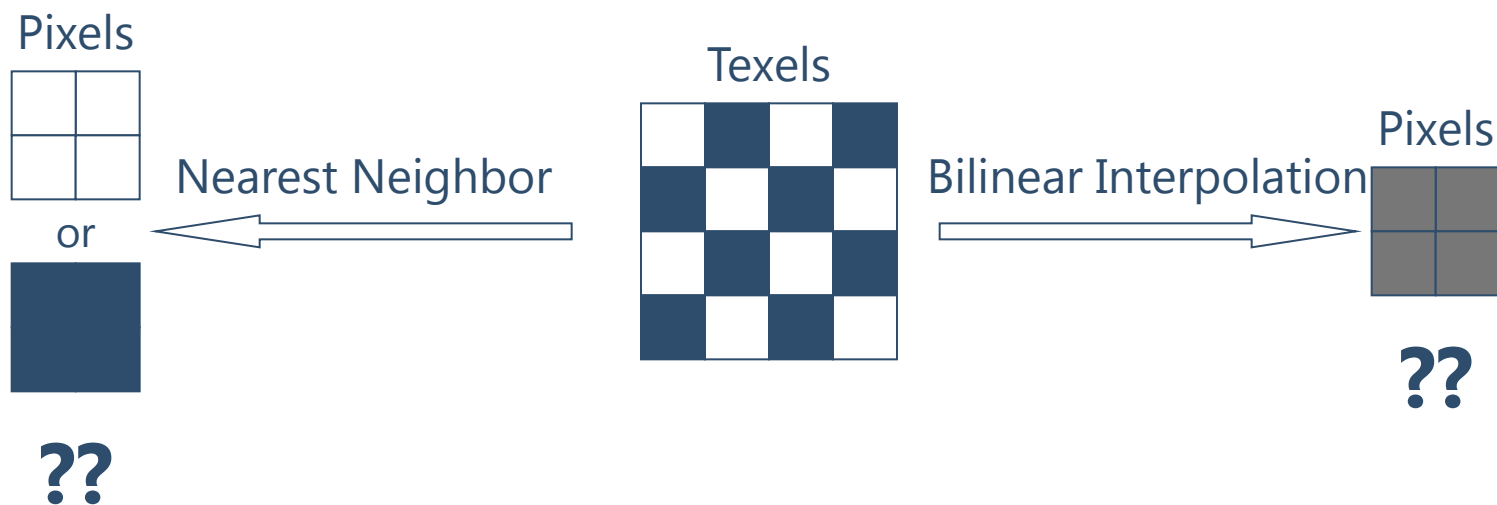
Magnification Filter Options

- GL_NEAREST
 - 选择最邻近的 texel
- GL_LINEAR
 - 双线性插值 texel
 - 消耗更多时间但更精确



Minification 的问题

- Minification 的时候也可以使用 nearest 或 bilinear, 但会导致许多走样的问题



Problem with Minification



without mipmap



with mipmap

Minification Filter Options

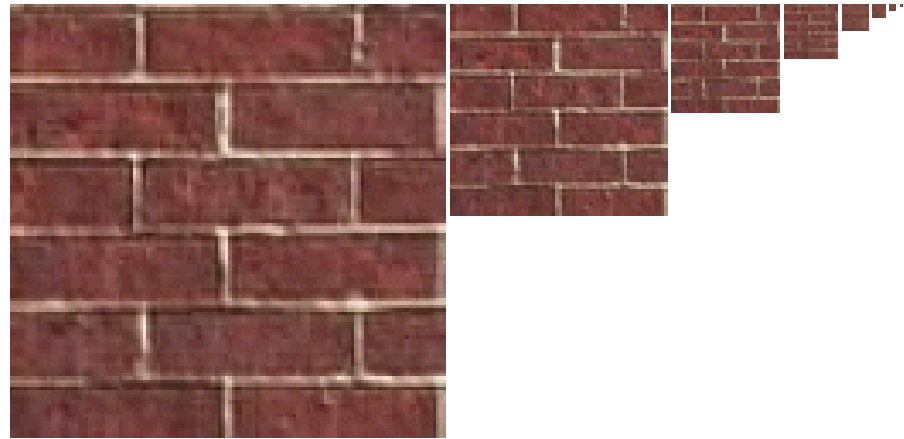
- `glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);`
- 可选选项有：
 - `GL_NEAREST`
 - `GL_LINEAR`
 - `GL_NEAREST_MIPMAP_NEAREST`
 - `GL_LINEAR_MIPMAP_NEAREST`
 - `GL_NEAREST_MIPMAP_LINEAR`
 - `GL_LINEAR_MIPMAP_LINEAR`

Minification Filter Options

- 后缀是 MIPMAP_NEAREST 选择与被贴图的像素最接近的一个 mipmap
 - GL_NEAREST_MIPMAP_NEAREST
 - GL_LINEAR_MIPMAP_NEAREST
- 后缀是 MIPMAP_LINEAR 选择与被贴图的像素最接近的两个 mipmap 插值
 - GL_NEAREST_MIPMAP_LINEAR
 - GL_LINEAR_MIPMAP_LINEAR

Mipmap Generation

- 使用下面的函数可以自动生成 mipmap
 - gluBuild2DMipmaps
- 也可以手工为每一个 level 指定一个单独的 mipmap 图像
- 存储量的增加：



$$1 + \frac{1}{4} + \frac{1}{4^2} + \dots = \frac{1}{1 - \frac{1}{4}} = \frac{4}{3}$$

注意：

Mipmap 的选择是每像素独立的

如果使用尚未准备好的 mipmap，那么纹理映射的功能将会被禁掉

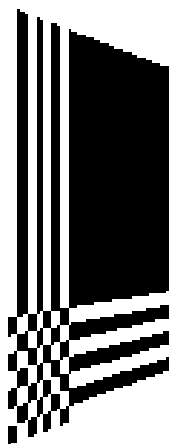
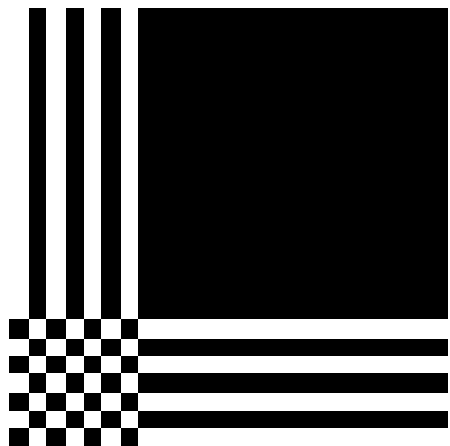
纹理重复

- 当纹理坐标超过 $[0,1]$ 访问的时候如何处理
- `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, param);`
 - 设置 S 方向上的重复方式
- `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, param);`
 - 设置 T 方向上的重复方式
- 参数: `GL_CLAMP` | `GL_REPEAT`

重复的参数

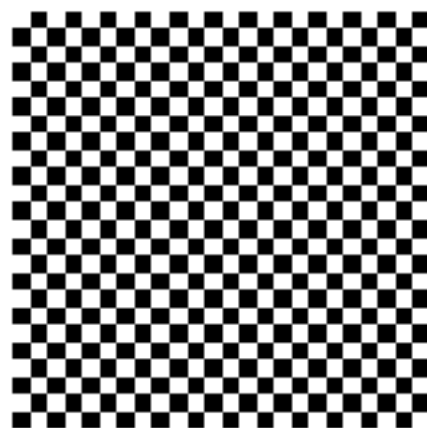
- GL_CLAMP

- 使用纹理坐标为 1 的纹理单元的颜色



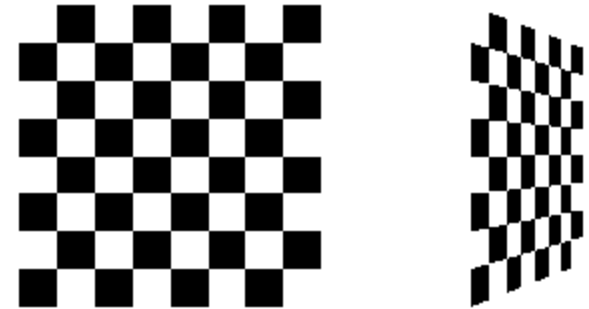
- GL_REPEAT

- 在 $[0,1]$ 上重复纹理



例程： checker.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
```



```
/* Create checkerboard texture */
#define checkImageWidth 64
#define checkImageHeight 64
static GLubyte
    checkImage[checkImageHeight][checkImageWidth][4];

static GLuint texName;
```

生成棋盘纹理

```
void makeCheckImage(void)
{
    int i, j, c;

    for (i = 0; i < checkImageHeight; i++) {
        for (j = 0; j < checkImageWidth; j++) {
            c = (((i&0x8)==0)^(j&0x8)==0)*255;
            checkImage[i][j][0] = (GLubyte) c;
            checkImage[i][j][1] = (GLubyte) c;
            checkImage[i][j][2] = (GLubyte) c;
            checkImage[i][j][3] = (GLubyte) 255;
        }
    }
}
```

初始化

```
void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);

    makeCheckImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    // 命名纹理图像以及创建纹理对象
    glGenTextures(1, &texName);
    glBindTexture(GL_TEXTURE_2D, texName);
```


初始化

// 设置重复方式

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

// 设置滤波方式

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  
                GL_NEAREST);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  
                GL_NEAREST);
```

// 定义二维纹理

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,  
            checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,  
            checkImage);
```

```
}
```

启用纹理，指定纹理坐标

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_TEXTURE_2D); // 启用纹理
    // 用纹理颜色替换物体表面颜色
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
              GL_REPLACE);

    glBindTexture(GL_TEXTURE_2D, texName);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
}
```

启用纹理，指定纹理坐标

```
glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);  
glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);  
glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -  
1.41421);  
glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -  
1.41421);  
glEnd();  
glFlush();  
glDisable(GL_TEXTURE_2D);  
}
```

纹理对象

纹理对象

- 纹理对象用于存储纹理数据以便随时使用它们，并使得纹理可以重复使用
- 通过纹理对象你可以同时控制许多纹理并且使用之前加载过的纹理资源
- 使用纹理对象通常是最快的应用纹理的方式，可以获得性能上的巨大提升
 - 通常将纹理对象与物体进行绑定（重用）比使用`glTexImage*D()`加载一个纹理图像更快

使用纹理对象的步骤

- 生成纹理名称
- 初次把纹理对象绑定（创建）到纹理数据上，包括图像数组和纹理属性
- 绑定和重新绑定纹理对象，使它们的数据当前可以用于渲染模型

纹理对象 API

- glGenTextures
 - 命名纹理对象
 - 分配 n 个纹理对象 (整数作为标识符)
- glBindTexture
 - 通过编号创建和使用纹理对象
- glDeleteTextures
 - 删除纹理对象，释放分配的纹理编号
 - glGenTextures 的逆过程

纹理对象

- 当用 `glGenTextures` 生成的纹理名称用 `glBindTexture()` 初次绑定时，OpenGL 就会创建一个新的纹理对象，并把纹理图像和纹理属性设置为默认值
- 接下去 `glTexImage*()`，`glTexParameter*()` 等函数的后续调用将把数据存储在纹理对象里
- 被保存在纹理对象中的纹理属性有：
 - Minification and magnification filters, wrapping modes, border color and texture priority

纹理对象

- 当纹理对象再次被绑定，它的数据成为当前的纹理状态，用户可以编辑绑定的纹理对象的内容
- 任何用来修改上述属性的命令都可以用来修改当前的绑定纹理
- 而纹理环境，纹理生成（texgen）等属性不会被保存在纹理对象中

例程：texbind.c

```
void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);

    makeCheckImages();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

```
    glGenTextures(2, texName);
```

```
    glBindTexture(GL_TEXTURE_2D, texName[0]);
```

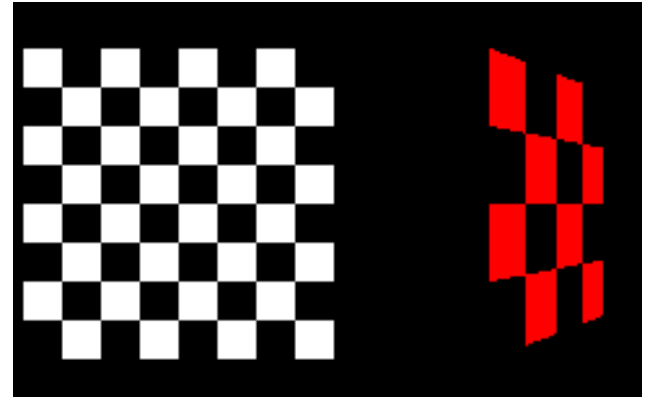
```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

```
    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, GL_RGBA, checkImageWidth,
                checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                checkImage);
```



初始化

glBindTexture(GL_TEXTURE_2D, texName[1]);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
 checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
 otherImage);

glEnable(GL_TEXTURE_2D);

}

显示函数

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D, texName[0]);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
    glEnd();

    glBindTexture(GL_TEXTURE_2D, texName[1]);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
    glEnd();
    glFlush();
}
```

使用 glBind 比每次都
用 glTexImage 创建纹
理速度更快

纹理变换

- 纹理坐标在纹理映射发生前会乘上一个 4×4 的矩阵
- 纹理动画：使用上述功能，我们可以生成物体表面的纹理动画，旋转，移动，拉伸.....
- 所有的矩阵操作都可以应用：
Push/Pop/Mult/Load...


```
glMatrixMode(GL_TEXTURE); /* 进入纹理矩阵模式 */  
glRotated(...);  
/* ...其他矩阵操作... */  
glMatrixMode(GL_MODELVIEW);  
/* 返回模型视图矩阵模式 */
```

PNG 纹理图像加载 (glpng)

- <http://www.fifi.org/doc/libglpng-dev/glpng.html>
 - 功能
 - 创建 mipmap
 - 加载并绑定 2d texture
 - 加载 png 图像数据 (image loader)
 - 可以操作透明 png (cut-out texture)
- Glpng.zip

glpng 重要 API

- 重要 API



纹理的 id 会由
png loader 返回
给你

- id = pngBind(filename, mipmap, trans, info, wrapst, minfilter, magfilter)
- pngLoad(filename, mipmap, trans, info)
- pngSetStencil(red, green, blue)
- pngSetStandardOrientation(1)

加载并绑定

- `id = pngBind(filename, mipmap, trans, info, wrapst, minfilter, magfilter)`
 - filename : 文件名
 - mipmap : 层数 , PNG_BUILD_MIPMAPS
 - trans : 透明度设置
 - info : 指向一个包含 png 文件信息的 `pngInfo` 结构的指针
 - wrapst, minfilter, magfilter : 重复方式 , 滤波方式设置

glpng 例子

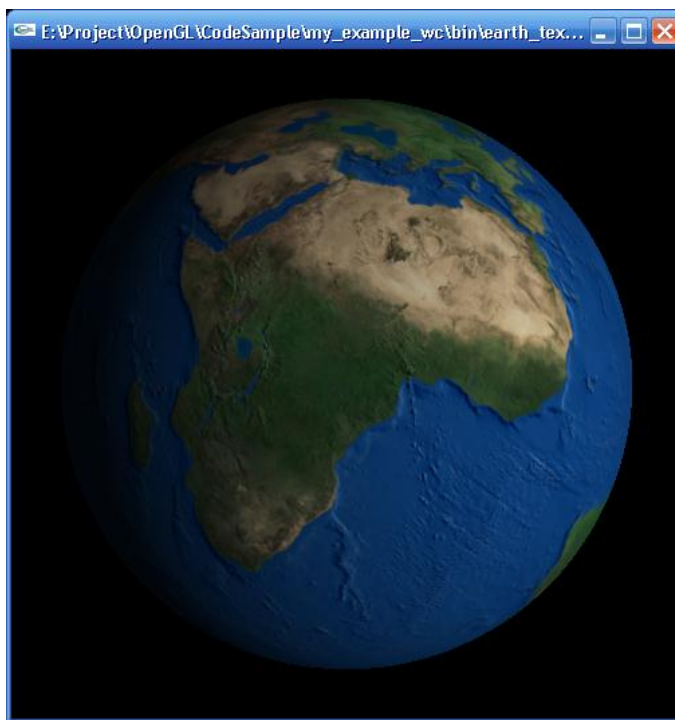
```
pngInfo info;
GLuint id = pngBind("Texture.png", PNG_NOMIPMAP,
    PNG_SOLID, &info, GL_CLAMP, GL_NEAREST,
    GL_NEAREST);

if (id != 0) {
    puts("Loaded Texture.png with resounding success");
    printf("Size=%i,%i Depth=%i Alpha=%i\n", info.Width,
        info.Height, info.Depth, info.Alpha);
}
else {
    puts("Can't load Texture.png");
    exit(1);
}
```

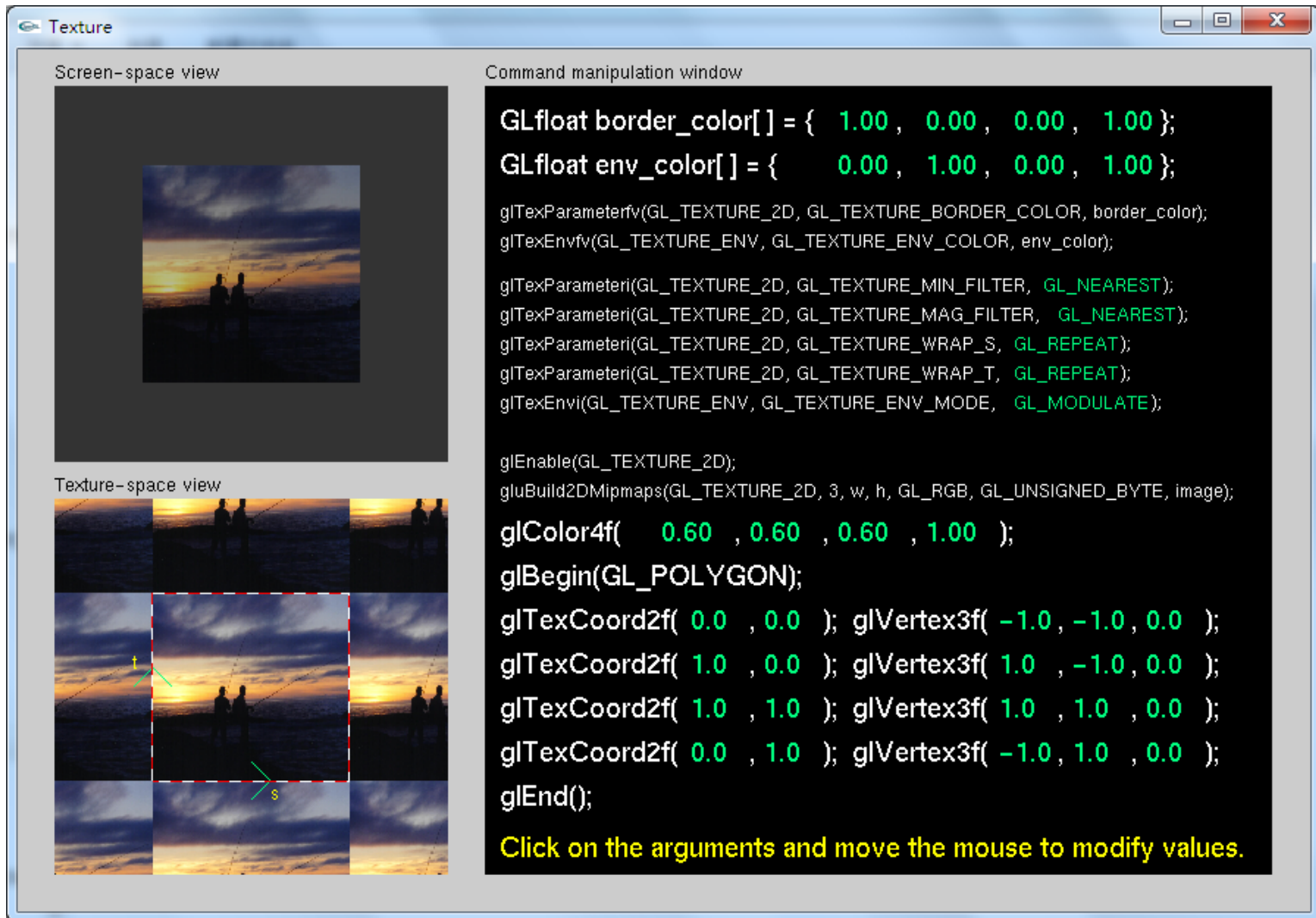
使用加载的纹理：glBindTexture(GL_TEXTURE_2D, id);

作业 6

- 为球体添加光照和纹理，完成一个地球模型动画



Nate Robin 纹理教学程序



Nate Robin 纹理教学程序

1、运行 texture.exe

2、改变纹理贴图环境属性glTexEnvi，并观察几个值的效果。如果使用GL_MODULATE，注意 glColor4f() 所指定的颜色的效果。如果选择GL_BLEND，可以观察一下如果改变 env_color数组所指定的颜色会发生什么情况。

Nate Robin 纹理教学程序

- 3、在4个不同的顶点上试验glTexCoord2f()的参数，并观察如何使用整个纹理图像的一部分进行贴图。（如果把纹理坐标设置为小于0或大于1会发生什么情况？）
- 4、观察环绕参数GL_REPEAT和GL_CLAMP的效果，需要把顶点（glTexCoord2f()的参数）的纹理坐标设置为小于0或大于1，以观察重复或截取的效果

Nate Robin 纹理教学程序

5、改变 `glTexParameterf` 的参数，同时改变 `glTexCoord` 的值，产生纹理缩小和放大的情况，观察各种纹理滤波方式的效果