nogodoss的专栏

个人资料

nogodoss

访问： 367016次
积分： 4745
等级： BLOG 5
排名： 第2562名

原创： 98篇 转载： 109篇
译文： 1篇 评论： 56条

文章搜索

文章分类

iphone (154)
iphone (1)
发布iphone程序 (2)
distribution (1)
App store (0)
App store distribution (1)
连续播放图片 (1)
UIScrollView捏合放大缩小 (1)
ios版本检测 (1)
图片截屏 (1)
图片合并 (1)
动画效果 (1)
object－c 压缩 (1)
解压缩 (1)
UIImage处理 (1)
监控软件 (1)
Linux (3)
VBA (1)
Xamarin (3)

文章存档

2015年05月 (4)
2015年04月 (4)
2015年03月 (2)
2015年02月 (1)
2015年01月 (8)

深入了解OpenGL——纹理基础

分类： iphone 2014-05-29 09:35 864人阅读 评论(0) 收藏 举报

转至：http://www.cocoachina.com/bbs/simple/?t38052.html

很感谢这位大牛。 未经过[illegible]他的文章，有问题请留言。

在前几讲，我们介绍了Ope[illegible]绘制方法、顶点线性变换、光照以及其它着色技巧，现在我们将讲述OpenGL中一个非常大的话题——纹理（texture）。

有些教材喜欢在一开始就把纹理贴图带一下，其实对此个人以为没有必要。把前面对于基于顶点绘制的技巧掌握好以后，再理解纹理反而会更容易些。

什么是纹理？这个名词似乎有些抽象，我们略懂一点，但又说不清楚。
在Wiki上，我们可以查到相关解释：
1、一只来自荷兰的金属乐队——这个显然不是我们所要的
2、时空结构中的虚拟拓扑瑕疵（天文学）——这个显然也不是我们所要的
3、共享某个方向度数的材料个体的微晶（结晶）——有些搭边，而且挺学术的，呵呵
4、一块岩石的外型和特征（地质学）——有些搭边
5、由一段音乐反复部分的交互创建的整体音乐（音乐）——在音乐上也是很不错的比喻，呵呵
6、路表面的特征，比起路的粗糙更短小的波形——有些搭边的
7、设计元素与其在艺术中的应用——蛮抽象的，呵呵
8、纹理映射——在计算机图形中应用于一个表面的位图图像——这正是我们想要的，呵呵
9、盐纹理——一颗盐的颗粒大小的相对比例——可以参考，呵呵
10、基于所使用的颜料的画布的感觉以及应用方法——这个值得参考，呵呵

OpenGL并没有对纹理（Texture）一次做很详细的解释。但是我们可以参考Wikipedia中的第8条——纹理映射。其实，作为偶个人来讲，一个纹理其实就是一幅图像。我们可以把这幅图像的整体或部分贴到我们先前用顶点勾画出的物体上去——比如对一个立方体、圆等贴上纹理图。我们也可以对纹理图像的整体或某个部分重复使用，贴到我们的目标物体上。

更精确地来讲，把纹理视作为图像是狭隘的。因为纹理可以是一维、二维或三维的。

我们如何获得纹理呢？我们大部分获得纹理的途径是通过专业的绘图软件来进行绘制，最后把图片保存为png、jpg等格式。当然，还用更直接的方法就是用数码相机拍摄（这个时候iPhone4的摄影功能就能派上用场了，呵呵）。除此之外，我们还可以通过编程来创建一幅图像作为纹理（在我们的第一个例子中就是这么做的）。

我们下面将通过一个简单的例子来阐明如何将一个2维纹理贴到一个立方体上。

完整的工程可以通过以下附件获得。

下面代码描述了对OpenGL状态的初始化。这部分初始化包含了对纹理参数的设置。

复制代码

```
1.
- (void)prepareOpenGL
{
    initCheckImage();

    glEnable(GL_TEXTURE_2D);
    glEnable(GL_CULL_FACE);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_MULTISAMPLE);

    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    glFrontFace(GL_CCW);
    glCullFace(GL_BACK);

    glShadeModel(GL_SMOOTH);

    glClearColor(0.4, 0.4, 0.4, 1.0);

    GLuint texName;
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glGenTextures(1, &texName);
    glBindTexture(GL_TEXTURE_2D, texName);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);
    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_REPLACE);
    glTexCoordPointer(2, GL_FLOAT, 0, texCoords);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 64, 64, 0, GL_RGBA,
GL_FLOAT, checkImage);

    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glColorPointer(4, GL_FLOAT, 0, colors);

    glViewport(0, 0, 320, 320);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-2.0, 2.0, -2.0, 2.0, 1.0, 5.0);
```

```
    glMatrixMode(GL_MODELVIEW);

    [[NSTimer scheduledTimerWithTimeInterval:0.05f target:self
selector:@selector(timerFireMethod:) userInfo:nil repeats:YES] retain];
}
```

第三行，调用自己定义的函数initCheckImage()用于创建一个国际象棋棋盘样式的纹理，并且存放在全局变量checkImage中。

第21行，我们定义了一个texName，作为一个纹理名。这个纹理名将作为一个纹理的一个ID。在第23行中，我们通过调用OpenGL接口glGenTextures来生成一个纹理对象，那么texName就可以被看作为这个纹理对象的句柄。

下面简单地介绍一下这个接口：

```
[objc]
01. void glGenTextures(    GLsizei      n,
02.     GLuint *      textures);
```

这个接口用于生成纹理名。它含有两个参数：

n：指定要生成多少个纹理名。在本例中，我们只有一个纹理，因此传给n的就是1。

textures：用于存放纹理名。由于纹理名可以有多个，因此这里可以是GLuint的数组。

glGenTextures将生成的纹理名存放到textures中。

这一步也是用于初始化纹理的第一步。

第二步，我们看第22行，对glPixelStorei的调用。

glPixelStorei用于设置像素存储模式。下面将简单介绍一下这个接口。

```
[objc]
01. void glPixelStorei(    GLenum pname,
02.     GLint param);
```

它含有两个参数：

pname：指定所要被设置参数的符号名。这里，参数的符号名有两种，一种是GL_PACK_ALIGNMENT，它影响将像素数据写回到主存的打包形式，对glReadPixels的调用产生影响；还有一种是GL_UNPACK_ALIGNMENT，它影响从主存读到的像素数据的解包形式，对glTexImage2D以及glTexSubImage2D产生影响。

param：指定相应的pname设置为什么值。这个数值一般是1、2、4或8，用于指定存储器中每个像素行有多少个字节对齐。对齐的字节数越高，系统就越能优化。

在实际代码中，我们看到的是glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

实际上，我们可以这里可以用4（默认值）。因为checkImage能够保证是4字节对齐的。当然，我们可以通过对checkImage的修改使其保证是8字节对齐：

```
[objc]
01. static GLfloat __attribute__((align(8))) checkImage[64 * 64 * 4];
```

这样，我们就能放心大胆地使用glPixelStorei(GL_UNPACK_ALIGNMENT, 8);了。

第三步，则是24行，调用glBindTexture(GL_TEXTURE_2D, texName);来绑定纹理。

glBindTexture将一个命名纹理绑定到纹理目标。

```
[objc]
01. void glBindTexture(    GLenum      target,
02.     GLuint      texture);
```

它含有两个参数：

target：指定纹理要绑定到哪个目标。这个参数必须是GL_TEXTURE_1D、GL_TEXTURE_2D、GL_TEXTURE_3D或

GL_TEXTURE_CUBE_MAP。对于OpenGL ES来说，只能是GL_TEXTURE_2D。
texture：这个就是我们在第一步时通过调用glGenTextures所获得的纹理名。

第四步，设置纹理参数。我们参见3楼代码的第26到29行。这里对OpenGL接口glTexParameteri进行调用。
由于OpenGL2.1上的参数种类太多，因此这里将仅对OpenGL ES1.1所兼容的参数进行讲解。
首先介绍接口：

```objc
void glTexParameteri(    GLenum target,
     GLenum pname,
     GLint param);
```

target：指定目标纹理，一维、二维、三维等。
pname：指定一单值纹理参数的符号名。在OpenGL ES1.1中可以是：GL_TEXTURE_MIN_FILTER、GL_TEXTURE_MAG_FILTER、GL_TEXTURE_WRAP_S、GL_TEXTURE_WRAP_T或GL_GENERATE_MIPMAP。
param：指定pname的值。

下面先介绍GL_TEXTURE_MAG_FILTER和GL_TEXTURE_MIN_FILTER这两个参数。

GL_TEXTURE_MAG_FILTER和GL_TEXTURE_MIN_FILTER这两个参数指定纹理在映射到物体表面上时的缩放效果。
GL_TEXTURE_MIN_FILTER是缩小情况；GL_TEXTURE_MAG_FILTER是放大情况。
这两个参数所对应的值有两种，一种是GL_NEAREST。这个值指定了放大或缩小所采取的算法是最近邻法，即采样的像素是纹理映射进行缩放时，距离目标坐标（或称为中心像素的坐标）最近的像素。举个简单的例子。比如说对纹理做两倍的缩小，那么缩小后的纹理，第一个像素是原来的第一个像素，而第二像素则是原来第三个像素.....对于两倍放大的情况下，放大后的第一个像素和第二个像素都是采原来第一个像素.....
最近邻算法的优势是速度快，所需的计算资源很小。而缺点是缩放的效果会显得很尖锐，在物体旋转时，往往会伴随有较严重的锯齿。我们在本章第一个示例演示程序中可以观察到，立方体表面的纹理锯齿现象很严重。
第二个值是GL_LINEAR。这个值指定了纹理进行缩放时采用双线性插值（Bilinear Interpolate）算法。这个算法并不是直接将源纹理采到的像素搬到缩放后的纹理上。而是选中了目标像素（或称为中心像素）后，对其上下左右4个相邻的像素按照它们各自对目标像素所占的比重进行插值。这种插值算法的特点是运算量较大（将会用到三次乘法（或乘加）计算），但缩放效果会显得非常平滑。

我们可以将3楼代码中第28、29行改成如下设置，然后查看结果：

```objc
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

我们将看到立方体的表面纹理将变得非常平滑，效果比原来强很多。

我们在接着讲述纹理参数之前，先再引入一些关于纹理的基本概念。
下面我们将引入“纹理元素”（texel）这个概念。我们直到一个图像的基本元素就是“像素”。一个像素有两种属性，一种是像素的位置（坐标），还有一种就是像素的颜色。对于像素，我们用(x, y)来指明一个坐标，并且一般用(r, g, b, a)来指明其颜色。
而纹理元素与像素类似。但由于纹理可以是1到3维的，因此texel的坐标可以由s（一维）或(s, t)（二维）或(s, t, r, q)（三维）来表示。对于三维纹理的texel而言，(s, t, r, q)对应于3D物体的坐标(x, y, z, w)。而二维纹理只需要(s, t)即可。

下面讨论参数GL_TEXTURE_WRAP_S与GL_TEXTURE_WRAP_T。这两个参数分别设置纹理s方向（水平方向）和t方向（垂直方向）的包裹方式。
什么是包裹方式呢？当我们在进行纹理映射时，如果某个方向上的纹理坐标超出了纹理范围（某个方向上的坐标值小于0或大于1），那么我们必须指定一种方式来选取纹理像素。
由于目前OpenGL ES1.1仅支持GL_CLAMP_TO_EDGE和GL_REPEAT，因此我们着重介绍这两种包裹方式。

为了能更透彻地看清这两种模式的工作方式，我这边对上述代码进行了修改，请用下述函数来替换原有的：

复制代码

```
static void initCheckImage(void)
{
    GLfloat initColor = 1.0f;
    for(int row = 0; row < 64; row++)
    {
        if((row & 7) == 0)
            initColor = 1.0f - initColor;

        GLfloat color1 = 1.0f - initColor;
        CGFloat color2 = initColor;

        for(int col = 0; col < 64; col++)
        {
            if((col & 7) == 0)
            {
                color1 = 1.0f - color1;
                color2 = 1.0f - color2;
            }

            checkImage[row * 64 * 4 + col * 4 + 0] = color1;
            checkImage[row * 64 * 4 + col * 4 + 1] = 0.0f;
            checkImage[row * 64 * 4 + col * 4 + 2] = color2;
            checkImage[row * 64 * 4 + col * 4 + 3] = 1.0f;
        }
    }
}

static GLfloat texCoords[] = {

    // left
    0.0f, 0.0f,
    1.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,

    // front
    0.8f, 0.8f,
    1.8f, 0.8f,
    0.8f, 1.8f,
    1.8f, 1.8f,

    // right
    0.0f, 0.0f,
    1.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,

    // back
    0.0f, 0.0f,
    1.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,
```

```
    // top
    0.0f, 0.0f,
    1.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,

    // bottom
    0.0f, 0.0f,
    1.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f
};

- (void)prepareOpenGL
{
    initCheckImage();

    glEnable(GL_TEXTURE_2D);
    glEnable(GL_CULL_FACE);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_MULTISAMPLE);

    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    glFrontFace(GL_CCW);
    glCullFace(GL_BACK);

    glShadeModel(GL_SMOOTH);

    glClearColor(0.4, 0.4, 0.4, 1.0);

    GLuint texName;
    glPixelStorei(GL_UNPACK_ALIGNMENT, 8);
    glGenTextures(1, &texName);
    glBindTexture(GL_TEXTURE_2D, texName);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);

    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_REPLACE);
    glTexCoordPointer(2, GL_FLOAT, 0, texCoords);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 64, 64, 0, GL_RGBA,
GL_FLOAT, checkImage);
```

```
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glColorPointer(4, GL_FLOAT, 0, colors);

    glViewport(0, 0, 320, 320);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1.0, 1.0, -1.0, 1.0, 1.0, 5.0);

    glMatrixMode(GL_MODELVIEW);

    [[NSTimer scheduledTimerWithTimeInterval:0.05f target:self
selector:@selector(timerFireMethod:) userInfo:nil repeats:NO] retain];
}
```

首先谈谈GL_REPEAT方式。这种方式下，我们可以想像把整个纹理进行从左到右，从下到上进行复制，以至于扩大后的纹理总能在指定的坐标范围内。然后将这个巨大的纹理进行缩放贴到目标物体的表面上。对于上述例子，我们可以想像这个棋盘纹理被复制了四份，以至于能容下(0.8, 0.8)到(1.8, 1.8)的坐标范围。为了证明这点，我们把front面所对应的纹理坐标再次改成如下形式，看看效果：

```
// front
0.0f, 0.0f,
2.0f, 0.0f,
0.0f, 2.0f,
2.0f, 2.0f,
```

我们将发现，这个表面的最终颜色将是16 * 16的棋盘，而不是8 * 8了，呵呵。

最后谈谈GL_CLAMP_TO_EDGE方式。这个方式比GL_REPEAT要简单些。它仅仅对超出范围的纹理坐标做单纯的截断。如果某个方向上的纹理坐标小于0，那么取0；如果大于1，则取1。所以我们可以把模式改成GL_CLAMP_TO_EDGE来看看效果，很直白，呵呵。

第五步：设置纹理环境并指定纹理坐标。

我们用以下接口来设置纹理环境：

```
[objc]

01. void glTexEnvi(    GLenum target,
02.     GLenum pname,
03.     GLint param);
```

target：指定一个纹理环境。对于OpenGL，可以是GL_TEXTURE_ENV、GL_TEXTURE_FILTER_CONTROL或GL_POINT_SPRITE；对于OpenGL ES，可以是GL_TEXTURE_ENV或GL_POINT_SPRITE_OES。
pname：指定一个单值纹理环境参数的符号名。我们这里将使用GL_TEXTURE_ENV_MODE。
param：指定pname的值。我们这里将使用GL_REPLACE。这个值表明用纹理元素（texel）替换掉原来的像素。
对于这个函数的调用，我们可以参见3楼代码的第31行。

我们使用以下接口来指定纹理顶点：

```objc
void glTexCoordPointer(    GLint size,
    GLenum type,
    GLsizei stride,
    const GLvoid * pointer);
```

size：每个纹理坐标的元素个数。我们这里用的是二维纹理，因此纹理坐标只需要s和t两个值即可。所以这里，这个参数应该传2。
type：指定纹理坐标的数据类型。我们这里用的是单精度浮点，因此传GL_FLOAT。
stride：在相继的两个纹理坐标之间的字节偏移。这里没有使用额外的信息，因此每个顶点坐标之间都是紧挨的，偏移为0。
pointer：指向纹理坐标数组首地址。
关于这个接口的调用请参见3楼代码中的第32行。

下面还要再描述一下纹理坐标。
我们在定义纹理坐标时，每个坐标与顶点相对应。纹理坐标(0, 0)表示纹理数组中的第一个texel（纹理元素）的坐标位置；(N, 0)表示第N行第一列的texel的坐标位置；(0, N)表示第一行第N列texel的坐标位置；(N, N)表示最后一个texel的坐标位置。
比如我们定义：

复制代码

```
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

static GLfloat vertices[] = {
    -1.0f, 1.0f, 0.0f, 1.0f,
    -1.0f, -1.0f, 0.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f,
    1.0f, -1.0f, 0.0f, 1.0f
};

static GLfloat texCoords[] = {

    0.0f, 1.0f,
    0.0f, 0.0f,
    1.0f, 1.0f,
    1.0f, 0.0f
};
```

那就表明纹理的第一个元素坐落在第二个顶点处，第一行最后一个元素坐落在第四个顶点处，第一列最后一行的元素坐落在第一个顶点处，最后一行最后一列的元素坐落在第三个顶点出。

而如果是：
复制代码

```
static GLfloat texCoords[] = {

    0.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 0.0f,
    1.0f, 1.0f
};
```

那么纹理的首个元素坐落在第一个顶点，第一行最后一列元素坐落在第三个顶点处，第一列最后一行的元素坐落在第二个顶点处，最后一行最后一列的元素坐落在第四个顶点出。
所以这个图与上面这个图正好成上下颠倒。即，沿着垂直于y轴的中线做180度的旋转。

上一篇 GCDAsyncUdpSocket 组播监听端口接收数据
下一篇 Apple's OpenGL——Vertex Shader基础

猜你在找

准备好了么？跳吧！

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap