六、纹理：(1)纹理化管线[Pipeline]

**纹理化[Textureing]**，简单来说，就是将一张纹理附着到（或覆盖）网格上并利用纹理值改变网格属性，使其着色外观发生变化的过程。这个过程主要分别网格参数化、访问纹理两个过程，下面分别一一说明。

**网格参数化[Mesh Parameterization]**：网格参数化的目的是在3D空间曲面顶点与纹理空间之间建立映射关系。在RTR4第6.1.1节中，这被称为Projector函数。比如球面的极坐标参数化

是将纹理空间映射到曲面顶点。取其逆，得到

就是网格参数化或Projector函数。实际中网格是由三角化之后的很多平面三角形组成的，或者有很多分段平滑曲面组成，因此不像球面具有解析解，并且同一曲面的不同参数化会导致的密度不一致，进而导致访问纹理时产生不均匀的纹理值分布。所以网格参数化是一个很复杂的过程，有关更详细的资料请参考[1]。

**访问纹理**：通过网格参数化获得纹理坐标，接下来就是访问纹理。在RTR4第6.1.2节中，这被称为Corresponder函数。Corresponder函数有两个作用：一是将纹理坐标转换到纹理位置坐标。继续用上面的示例举例，通过球面到纹理空间的映射，得到的的值域为,而纹理位置坐标的取值范围是，所以转换过程为；二是根据“Wrapping模式”将超出有效范围的进行约束，现有的图形硬件都支持以下几种模式：

1. **wrap**(DirectX),**repeat**(OpenGL)或tile-图像在整个表面上重复.在算法上,纹理坐标的整数部分被删除.此功能对于使材料的图像重复覆盖表面很有用,通常是默认设置.
2. **mirror**-图像在整个表面上重复,但在其他每个重复上都被镜像(翻转).例如,图像看起来通常从0到1,然后在1和2之间反转,然后在2和3之间正常,然后反转,依此类推.这提供了沿纹理边缘的连续性.
3. **clamp**(DirectX)或**clamp to edge**(OpenGL)-超出[0,1]范围的值将被钳制到该范围.这导致图像纹理的边缘重复.此功能对于避免在纹理的边缘附近发生双线性插值时避免意外从纹理的相对边缘进行采样[885]很有用.
4. **border**(DirectX)或**clamp to border**(OpenGL)-[0,1]外部的纹理坐标使用单独定义的边框颜色进行渲染.例如,由于纹理的边缘将与边框颜色平滑融合,因此此功能可以很好地将贴图渲染到单色表面上.

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | OpenGL | DirectX | 算法实现 |
| Repeat |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

纹理：(2)图像纹理

在上一节中，简要介绍了纹理化管线的概念，即从曲面顶点到纹理坐标的变换流程。本篇将要总结RTR4对纹理值——像素的处理方式，主要包括像素之间的各种差值算法的实现，Magnification和Minification在直觉上的原理和应用，以及压缩纹理的概念和常用的纹理压缩格式。由于部分内容需要更进一步的研究和发掘，所有本篇内容会持续不断的更新和扩充。

OpenGL vs DirectX

尽管纹理坐标的取值范围在不同图形API中的定义是相同的，但是如果纹理坐标系不一致，就会导致计算稍有差别。OpenGL和DirectX就是这样的典型。在DirectX中,纹理的左上角为(0,0),右下角为(1,1).这与存储图像数据的图像类型匹配,第一行是图像中的第一行.在OpenGL中,纹理像素(0,0)位于左下角,右上角为(1,1)，即x轴是一致的，但y轴恰好是DirectX y轴的翻转。所有一张图像以相同的方式读入并加载到纹理对象，并使用相同的矩形顶点流渲染，则DirectX得到的图像是正的，OpenGL是反的。

这种差异还会产生两个额外的不同。一是在屏幕空间中，DirectX渲染出的图像是从上到下栅格化，而OpenGL是从下到上栅格化。因此直接保存图像，则DirectX得到的图像是正的，而OpenGL是反的。另一个是将投影坐标（曲面顶点经过世界、相机、投影矩阵变换后除以得到的坐标值）转换到纹理坐标的算法不一致,例如常见的屏幕空间转纹理坐标:

OpenGL

vec2 uv = proj.xy \* 0.5 + 0.5;

DirectX

float2 uv = proj.xy \* float2(0.5, -0.5) + 0.5;

其中proj是投影坐标。

依赖纹理读取[dependent texture read]

依赖纹理读取是与纹理读取性能有关的一个概念，有两个定义。

“第一个特别适用于移动设备.当通过texture2D或类似方法访问纹理时,只要像素着色器计算纹理坐标而不是使用从顶点着色器传入的未修改纹理坐标,就会发生依赖纹理读取.请注意，这意味着对传入的纹理坐标进行任何更改,甚至包括交换和值之类的简单操作.当着色器没有发生依赖纹理读取时,较早的不支持OpenGL ES 3.0的移动GPU可以更高效地运行,因为可以预取纹理像素数据.这个术语的另一个较旧的定义对于早期的台式机GPU特别重要.在这种情况下,当一个纹理的坐标依赖于某些先前纹理值的结果时,就会发生依赖纹理读取.例如,一种纹理可能会更改着色法线,进而改变用于访问立方体贴图的坐标.在早期的GPU上,此类功能受到限制甚至不存在.如今,**此类读取可能会影响性能,具体取决于批处理中要计算的像素数量以及其他因素**.”

笔者在早期从事移动端图形开发时，对依赖纹理读取深有体会，确实会严重降低性能。

像素的中心点

纹理是由一组离散的矩形点阵构成，为了方便插值，纹理坐标往往是连续的。因此为纹理定义具有浮点数的中心坐标就很有意义了。目前所有的主流硬件API都以像素的整数坐标加0.5偏移作为像素的重心坐标。比如(1,2)位置的像素以(1.5,2.5)作为像素的中心。OpenGL fragment shader中内置只读变量gl\_FragCoord也包含了0.5偏移。早期的DirectX 9以及之前的版本将中心点定义在整数坐标上，从DirectX 10之后，与OpenGL保持一致。

Magnification和Minification

给定一个256×256的纹理，假如要将其完整显示到512×512分辨率的屏幕上。显然一个纹理像素要对应四个屏幕像素，这种情况下我们把对纹理的采样称为Magnification。与之相反的是，如果将256×256纹理完整显示到128×128分辨率的屏幕上，此时一个屏幕像素对应四个纹理像素，我们把这种对纹理的采样称为Minification。这是一个不太严谨但很直观的解释。由于这两种情况经常出现在渲染中，处理不当会导致各种各样的artifact,比如画面随相机移动会闪烁，各种色阶等。

Magnification

Magnification的常见的处理有以下几种，

**邻近采样[nearest neighbor]**：这是最简单的方法，直接应用最近的像素值。缺点是如果放大倍数过大，则会导致严重的马赛克。

**双线性插值[bilinear interpolation]**：屏幕空间中每个像素的坐标值转换到纹理坐标是浮点值，因此（除了边界）总是可以找到四个邻近的像素进行插值。双线性插值就是处理四个邻近像素的插值算法。

假设是输入的一对纹理坐标,,和分别为纹理的宽和高。再假设像素值通过函数获取的,其中和是整数。首先要计算出插值权重和四个邻近的像素,水平方向插值,垂直方向插值.考虑到像素中心有0.5偏移，算法如下

其中表示对取整数坐标。有了上述值，接下来对像素值进行双线性插值

得到最终结果。