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。

邻近采样和双线性插值均被GPU支持。

**三次曲面插值**:上述讨论的双线性插值仅仅保证曲面在像素点处连续,而高阶滤波模式通常会保证曲面的导数在点处也连续，产生更高的图像质量.常用的三次插值是B样条插值。通常的做法是在采样点附近选择4x4个样本来构建B样条曲面，但是这样做需要付出16个采样的昂贵代价。[2]提出了一种利用GPU本身支持的线性插值从而将采样率降低为4。下面介绍该算法。

假设要在纹理坐标轴x处(y轴类似)构建三次过滤，就需要评估四个附近纹理的加权卷积

其中是B样条系数，对于三次B样条，它们分别为

表示第点处的像素值.

公式(6.2)需要四个采样点，而每个采样点在y轴方向上也需要四个。因此总共需要16个采样点。现在需要将公式(6.2)中四个采样点降低为2两个。

由于GPU本身支持线性插值，根据线性插值的定义,

其中表示插值坐标的整数部分,表示位于区间的小数部分.看成是texture2D函数。基于此公式,可以将两个相邻样本的加权加法重写为加权线性插值:

根据公式(6.3)，公式(6.2)可重写为两个加权线性插值:

其中

公式(6.4)-(6.8)是最终的结果。和相当于texture2D函数,其余的和可以在shader中实现.

另一种更为简单的三次过滤算法不需要四次采样，只需重新映射纹理坐标，再利用GPU双线性采样即可。纹理坐标重映射算法为:

函数就是shader中内置的函数smoothstep,具有一阶平滑导数。而具有二阶平滑导数.

Minification

当一个屏幕像素对应多个纹理像素时，就需要从这些像素中计算正确的颜色值。在上一小节Magnification中所提到的所有算法都可以应用到Minification的情况中，但是这些算法都存在一定的缺陷。

**邻近采样[nearest neighbor]**：由于从覆盖的多个像素中仅选取最近的，那么在相机移动或物体移动时，必然会出现像素值剧烈抖动，即artifacts，这种artifacts被称为时间混叠[temporal aliasing].

**双线性插值[bilinear interpolation]**：该过滤器仅比最邻近的方法略胜一筹.它混合了四个纹理像素,而不是仅使用一个,但是当一个像素受到四个以上纹理像素的影响时,滤镜很快就会失效并产生artifact.

Mipmapping

如上所述，当一个屏幕像素覆盖的纹理像素超过四个时，必须提高采样率才能保证颜色正确。因此需要对纹理进行预过滤，将不同频率的颜色值存储在不同的纹理中，所以Mipmapping应运而生。

Mipmapping是一组纹理链，使用级别表示不同纹理的索引。级别0表示最原始的纹理，级别1表示原始纹理过滤后的纹理，其长宽均为原始纹理的一半，因此分辨率是原始纹理的1/4，过滤时需要将原理纹理的四个矩形相邻像素做box过滤，将结果存储在级别1对应的像素中。级别2、3、4纹理的属性和过滤方式与之类似。

除了Box过滤器之外，还有其它的一些过滤算法，这里稍后总结：

一个长宽均为W的纹理能包含的Mipmap数量N为:

由于纹理中存储的数据类型各不相同，比如线性颜色、非线性颜色、法线、粗糙度等各种着色属性。普通的box过滤器或其它一些过滤器仅支持线性数据，如果应用到非线性数据会导致渲染结果不正确。这里总结出一些如何正确过滤非线性数据的方法（不定期更新）：

**sRGB颜色**：作为最常用的非线性颜色，大多数图形API都支持sRGB格式的纹理，因此使用API可以正确生成mipmap链。如果需要手动生成sRGB纹理的mipmap链，首先需要将sRGB转换到线性RGB，然后做box过滤，最后将过滤结果再转换成sRGB存储。

法线：

访问Mipmapping

目前所有硬件API都默认支持mipmapping，并且大多数支持自动生成mipmapping。访问Mipmapping时，只需要在调用纹理函数时传入纹理坐标即可，硬件API会根据纹理坐标相对于屏幕空间坐标的导数自动计算mipmap级别。一般情况下，得到的mipmap级别是浮点数而不是整数，在这种情况下一般根据mipmap级别混合邻近的两个mipmap，因此这被称为三线性插值。

mipmapping的好处在于,访问并插值了预组合的像素集,而不是试图对影响像素的所有像素进行求和.无论缩小多少,此过程都会花费固定的时间.但是,mipmapping有几个缺陷.一个主要的是模糊.想象一下,一个像素单元在u方向上覆盖大量纹理像素,而在v方向上仅覆盖少数像素.这种情况通常发生在观看者近乎边缘地沿着带纹理的表面看时.实际上,可能需要沿着纹理的一个轴进行最小化,而沿着另一个轴进行放大.访问mipmap的效果是检索了纹理上的正方形区域.无法检索矩形区域.

求和面积表(SAT)

避免过度模糊的另一种方法是求和面积表(Summed-Area Table,SAT).要使用此方法,首先要创建一个数组,该数组的大小与纹理大小相同,但包含更多的颜色存储精度位(例如,红色,绿色和蓝色分别为16位或更多).在此阵列中的每个位置,必须计算并存储所有相应纹理的纹理像素的总和,该矩形由该位置和纹理像素(0,0)(原点)形成.在纹理化过程中,像素单元在纹理上的投影由矩形限制.然后访问求和面积表以确定该矩形的平均颜色,该颜色作为像素的纹理颜色传回.使用图6.17中所示的矩形的纹理坐标计算平均值.这可使用公式6.10中给出的公式完成:

这里,和是矩形的纹素坐标,而是该纹素的求和面积值.该方程的工作原理是:从右上角到原点,取整个区域的总和,然后通过减去相邻角的贡献减去面积A和B.区域C已被减去两次,因此将其添加到左下角.注意是区域C的右上角,即是边界框的左下角.

求和面积表是所谓的各向异性过滤算法的一个示例[**691**].这样的算法在非正方形区域上检索纹理像素值.但是,SAT能够在主要水平和垂直方向上最有效地做到这一点.还要注意,求和区域表对大小为16×16或更小的纹理的占用内存至少是其两倍，而较大的纹理则需要更高的精度.