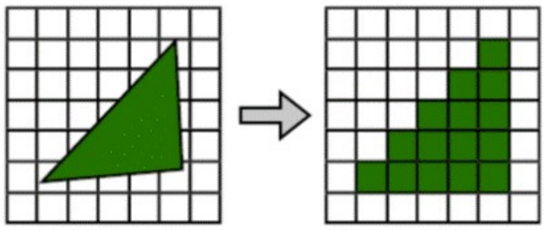
光栅化的深入理解

2013-08-19 12:33 2621人阅读 评论(0) 收藏 举报

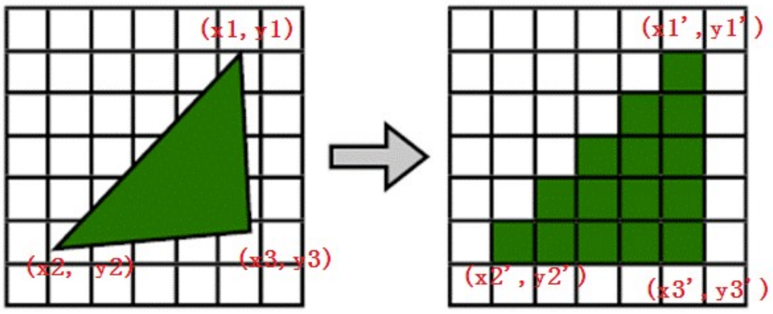
分类： 图形学（5） Direct3D（15）

一、先了解下 什么是光栅化及光栅化的简单过程?

光栅化是将几何数据经过一系列变换后最终转换为像素，从而呈现在显示设备上的过程，如下图：



光栅化的本质是坐标变换、几何离散化，如下图：



有关光栅化过程的详细内容有空再补充。

二、以下内容展示纹素到像素时的一些细节：

原文http://msdn.microsoft.com/en-us/bb219690(v=vs.85)或http://msdn.microsoft.com/en-us/ee417850%28VS.85%29.aspx

When rendering 2D output using pre-transformed vertices, care must be taken to ensure that each texel area correctly corresponds to a single pixel area, otherwise texture distortion can occur. By understanding the basics of the process that Direct3D follows when rasterizing and texturing triangles, you can ensure your Direct3D application correctly renders 2D output.

当使用已经执行过顶点变换的顶点作为2D输出平面的时候，我们必须确保**每个纹素正确的映射到每个像素区域**，否则纹理将产生扭曲，通过理解Direct3D在光栅化和纹理采样作遵循的基本过程，你可以确保你的Direct3D程序正确的输出一个2D图像。

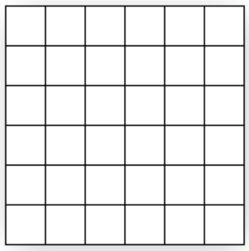


图1: 6 x 6 resolution display

Figure 1 shows a diagram wherein pixels are modeled as squares. In reality, however, pixels are dots, not squares. Each square in Figure 1 indicates the area lit by the pixel, but a pixel is always just a dot at the center of a square. This distinction, though seemingly small, is important. A better illustration of the same display is shown in Figure 2:
图片1展示了用一个方块来描述像素的。实际上，像素是点，不是方块，每个图片1中的方块表明了被一个像素点亮的区域，然而像素始终是方块中间的一个点，这个区别，看起来很小，但是很重要。图片二展示了一种更好的描述方式。

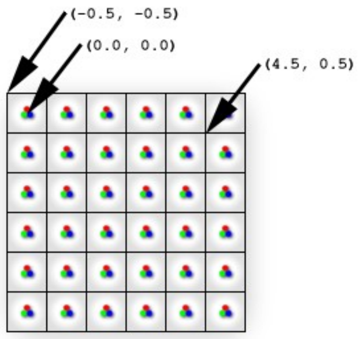


图 2: Display is composed of pixels

This diagram correctly shows each physical pixel as a point in the center of each cell. The screen space coordinate (0, 0) is located directly at the top-left pixel, and therefore at the center of the top-left cell. The top-left corner of the display is therefore at (-0.5, -0.5) because it is 0.5 cells to the left and 0.5 cells up from the top-left pixel. Direct3D will render a quad with corners at (0, 0) and (4, 4) as illustrated in Figure 3.
这张图正确地通过一个点来描述每个单元中央的物理像素。屏幕空间的坐标原点（0，0）是位于左上角的像素，因此就在最左上角的方块的中央。最左上角方块的最左上角因此是（-0.5，-0.5），因为它距最左上角的像素是（-0.5，-0.5）个单位。Direct3D将会在（0，0）到（4，4）的范围内渲染一个矩形，如图3所示

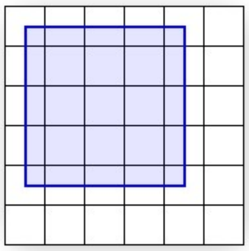


图3

Figure 3 shows where the mathematical quad is in relation to the display, but does not show what the quad will look like once Direct3D rasterizes it and sends it to the display. In fact, it is impossible for a raster display to fill the quad exactly as shown because the edges of the quad do not coincide with the boundaries between pixel cells. In other words, because each pixel can only display a single color, each pixel cell is filled with only a single color; if the display were to render the quad exactly as shown, the pixel cells along the quad's edge would need to show two distinct colors: blue where covered by the quad and white where only the background is visible.
Instead, the graphics hardware is tasked with determining which pixels should be filled to approximate the quad. This process is called rasterization, and is detailed inRasterization Rules. For this particular case, the rasterized quad is shown in Figure 4:

文章存档

2016年04月 (7)
2016年03月 (2)
2016年02月 (1)
2015年12月 (3)
2014年12月 (1)

阅读排行

VS项目属性的一些配置项 (11820)
mysql-5.6.14-winx64免安 (7853)
2010年国内外最新最全量 (6783)
印度人教小孩学的数学(乘 (3216)
WPF之DataGrid应用--几 (3198)
MFC界面--利用CHtmlVie (3164)
经济萧条时代让我们温暖 (2853)
光栅化的深入理解 (2621)
C语义char*显示中文---A: (2602)
手把手教你如何破解软件 (2196)

评论排行

VS项目属性的一些配置项 (2)
mysql-5.6.14-winx64免安 (1)
ogre1.8自己写的建立地形 (1)
c# Using语句三用途 (1)
机器学习入门的书单（数 (1)
五险一金-社保基数-住房: (1)
经典SQL语句大全(总结) (1)
理解顶点着色器和像素着 (0)
面向接口编程详解（三） (0)
hao-单例模式（Singletor (0)

推荐文章

*搭建docker私有仓库
*Android杂谈之RadioGroup+ViewPager制作的底部导航栏
* Android几种常见的多渠道(批量)打包方式介绍
*Oculus Rift, HTC Vive, SONY PSVR的全面对比
*Android View的事件分发机制探索
*Redis源码解析：15Resis主从复制之从节点流程

最新评论

mysql-5.6.14-winx64免安装配置
Future小来: 出现错误1067
机器学习入门的书单（数据挖掘、
大圣喜欢吃香蕉: 学习了。想用机器学习作用于机器人行动策略，作为小白学习下去！~很佩服，博主已经博览群书了呢。
VS项目属性的一些配置项的总结
huo_yun: 很详细，正好解决了我的疑惑
c# Using语句三用途
CoolYJCK: 最近使用using比较多，学习了！~using定义别名，这个平时用的少，谢谢楼主分享！~
ogre1.8自己写的建立地形源码（

请叫我汪海: 请问那个地形文件能不能给我一份，694175346@qq.com谢谢！
VS项目属性的一些配置项的总结
yuzhihaojia: 好文章
五险一金-社保基数-住房公积金基
zhongfenglin: om
经典SQL语句大全(总结)
system32: good ooadfico

图片3展示了数学上应该显示的矩形。但是并不是Direct3D光栅化之后的样子。实际上，像图3这样光栅化是根本不可能的，因为每个像素点亮区域只能是一种颜色，不可能一半有颜色一半没有颜色。如果可以像上面这样显示，那么矩形边缘的像素区域必须显示两种不同的颜色：蓝色的部分表示在矩形内，白色的部分表示在矩形外。因此，图形硬件将会执行判断哪个像素应该被点亮以接近真正的矩形的任务。这个过程被称之为光栅化，详细信息请查阅Rasterization Rules.。对于我们这个特殊的例子，光栅化后的结果如图4所示

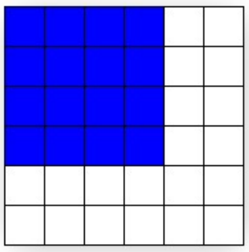


图4

Note that the quad passed to Direct3D (Figure 3) has corners at (0, 0) and (4, 4), but the rasterized output (Figure 4) has corners at (-0.5,-0.5) and (3.5,3.5). Compare Figures 3 and 4 for rendering differences. You can see that what the display actually renders is the correct size, but has been shifted by -0.5 cells in the x and y directions. However, except for multi-sampling techniques, this is the best possible approximation to the quad. (See theAntialias Sample for thorough coverage of multi-sampling.) Be aware that if the rasterizer filled every cell the quad crossed, the resulting area would be of dimension 5 x 5 instead of the desired 4 x 4.
If you assume that screen coordinates originate at the top-left corner of the display grid instead of the top-left pixel, the quad appears exactly as expected. However, the difference becomes clear when the quad is given a texture. Figure 5 shows the 4 x 4 texture you'll map directly onto the quad.
注意我们传给Direct3D（图三）的两个角的坐标为（0，0）和（4，4）（相对于物理像素坐标）。但是光栅化后的输出结果（图4）的两个角的坐标为（-0.5，-0.5）和（3.5，3.5）。比较图3和图4，的不同之处。你可以看到图4的结果才是正确的矩形大小。但是在x,y方向上移动了-0.5个像素矩形单位。然而，抛开multi-sampling技术，这是接近真实大小矩形的最好的光栅化方法。注意如果光栅化过程中填充所有被覆盖的物理像素的像素区域，那么矩形区域将会是5x5，而不是4x4.
如果你结社屏幕坐标系的原点在最左上角像素区域的最左上角，而不是最左上角的物理像素，这个方块显示出来和我们想要的一样。然而当我们给定一个纹理的时候，区别就显得异常突出了，图5 展示了一个用于映射到我们的矩形的4x4的纹理。

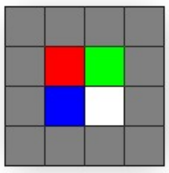


图5

Because the texture is 4 x 4 texels and the quad is 4 x 4 pixels, you might expect the textured quad to appear exactly like the texture regardless of the location on the screen where the quad is drawn. However, this is not the case; even slight changes in position influence how the texture is displayed. Figure 6 illustrates how a quad between (0, 0) and (4, 4) is displayed after being rasterized and textured.
　　因为纹理有4x4个纹素，并且矩形是4x4个像素，你可能想让纹理映射后的矩形就像纹理图一样。然而，事实上并非如此，一个位置点的轻微变化也会影响贴上纹理后的样子，图6阐释了一个（0，0）（4，4）的矩形被光栅化和纹理映射后的样子。

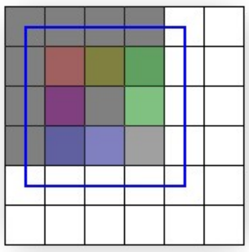


图6

The quad drawn in Figure 6 shows the textured output (with a linear filtering mode and a clamp addressing mode) with the superimposed rasterized outline. The rest of this article explains exactly why the output looks the way it does instead of looking like the texture, but for those who want the solution, here it is: The edges of the input quad need to lie upon the boundary lines between pixel cells. By simply shifting the x and y quad coordinates by -0.5 units, texel cells will perfectly cover pixel cells and the quad can be perfectly recreated on the screen. (Figure 8 illustrates the quad at the corrected coordinates.)
图6中展示了贴上纹理后的矩形(使用线性插值模式和CLAMP寻址模式)，文中剩下的部分将会解释为什么他看上去是这样而不像我们的纹理图。先提供一个解决这个问题的方法:输入的矩形的边界线需要位于两个像素区域之间。通过简单的将x和y值移动-0.5个像素区域单位，纹素将会完美地覆盖到矩形区域并且在屏幕上重现（图8阐释了这个完美覆盖的正确的坐标）（译者：这里你创建的窗口的坐标必须为整数，因此位于像素区域的中央，你的客户区屏幕最左像素区域的边界线在没有进行移位-0.5之前也必位于某个像素区域的中央）
The details of why the rasterized output only bears slight resemblance to the input texture are directly related to the way Direct3D addresses and samples textures. What follows assumes you have a good understanding oftexture coordinate space And bilinear texture filtering.
关于为什么光栅化和纹理映射出来的图像只有一点像我们的原始纹理图的原因和Direct3D纹理选址模式和过滤模式有关。详情见texture coordinate space Andbilinear texture filtering.
Getting back to our investigation of the strange pixel output, it makes sense to trace the output color back to the pixel shader: The pixel shader is called for each pixel selected to be part of the rasterized shape. The solid blue quad depicted in Figure 3 could have a particularly simple shader:
回到我们调查为什么会输出奇怪像素的过程中，为了追踪输出的颜色，我们看看像素着色器：像素作色器在光栅后的图形中的每个像素都会被调用一次。图3中蓝色的线框围绕的矩形区域都会使用一个简单的作色器：

```
float4 SolidBluePS() : COLOR

{

        return float4( 0, 0, 1, 1 );

}
```

For the textured quad, the pixel shader has to be changed slightly:

```
texture MyTexture;


sampler MySampler =

sampler_state

{

        Texture = <MyTexture>;

        MinFilter = Linear;

        MagFilter = Linear;

        AddressU = Clamp;

        AddressV = Clamp;

};


float4 TextureLookupPS( float2 vTexCoord : TEXCOORD0 ) : COLOR

{

        return tex2D( MySampler, vTexCoord );

}
```

That code assumes the 4 x 4 texture of Figure 5 is stored in MyTexture. As shown, the MySampler texture sampler is set to perform bilinear filtering on MyTexture. The pixel shader gets called once for each rasterized pixel, and each time the returned color is the sampled texture color at vTexCoord. Each time the pixel shader is

called, the vTexCoord argument is set to the texture coordinates at that pixel. That means the shader is asking the texture sampler for the filtered texture color at the exact location of the pixel, as detailed in Figure 7:
代码假设图5中的4x4的纹理存储在MyTexture中。MySampler被设置成双线性过滤。光栅化每个像素的时候调用一次这个Shader.每次返回的颜色值都是对sampled texture使用vTexCoord取样的结果，vTexCoord是物理像素值处的纹理坐标。这意味着在每个像素的位置都会查询纹理以得到这点的颜色值。详情如图7所示：

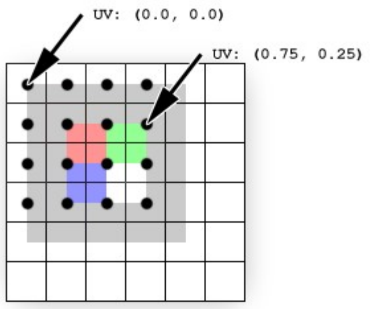


图7

The texture (shown superimposed) is sampled directly at pixel locations (shown as black dots). Texture coordinates are not affected by rasterization (they remain in the projected screen-space of the original quad). The black dots show where the rasterization pixels are. The texture coordinates at each pixel are easily determined by interpolating the coordinates stored at each vertex: The pixel at (0,0) coincides with the vertex at (0, 0); therefore, the texture coordinates at that pixel are simply the texture coordinates stored at that vertex, UV (0.0, 0.0). For the pixel at (3, 1), the interpolated coordinates are UV (0.75, 0.25) because that pixel is located at three-fourths of the texture's width and one-fourth of its height. These interpolated coordinates are what get passed to the pixel shader.
纹理（重叠上的区域）是在物理像素的位置采样的（黑点）。纹理坐标不会受光栅化的影响（它们被保留在投影到屏幕空间的原始坐标中）黑点是光栅化的物理像素点的位置。每个像素点的纹理坐标值可以通过简单的线性插值得到:顶点（0，0）就是物理像素（0,0）UV是(0.0,0.0)。像素（3，1）纹理坐标是UV(0.75,0.25)因为像素值是在3/4 纹理宽度和1/4纹理高度的位置上。这些插过值的纹理坐标被传递给了像素着色器。
The texels do not line up with the pixels in this example; each pixel (and therefore each sampling point) is positioned at the corner of four texels. Because the filtering mode is set to Linear, the sampler will average the colors of the four texels sharing that corner. This explains why the pixel expected to be red is actually three-fourths gray plus one-fourth red, the pixel expected to be green is one-half gray plus one-fourth red plus one-fourth green, and so on.
每个纹素并不和每个像素重叠，每个像素都在4个纹素的中间。因为过滤模式是双线性。过滤器将会取像素周围4个颜色的平均值。这解释了为什么我们想要的红色实际上确是3/4的灰色加上1/4的红色。应该是绿色的像素点是1/2的灰色加上1/4的红色加上1/4的绿色等等。
To fix this problem, all you need to do is correctly map the quad to the pixels to which it will be rasterized, and thereby correctly map the texels to pixels. Figure 8 shows the results of drawing the same quad between (-0.5, -0.5) and (3.5, 3.5), which is the quad intended from the outset.
为了修正这个问题，你需要做的就是正确的将矩形映射到像素，然后正确地映射纹素到像素。图8显示了将(-0.5, -0.5) and (3.5, 3.5)的矩形进行纹理映射后的结果。

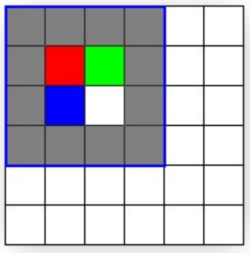


图8

Summary
In summary, pixels and texels are actually points, not solid blocks. Screen space originates at the top-left pixel, but texture coordinates originate at the top-left corner of the texture's grid. Most importantly, remember to subtract 0.5 units from the x and y components of your vertex positions when working in transformed screen space in order to correctly align texels with pixels.

总结：

总的来说，像素和纹素实际上是点，不是实体的块。屏幕空间原点是左上角的物理像素，但是纹理坐标原点是纹素矩形的最左上角。最重要的是，记住当你要将纹理中的纹素正确的映射到屏幕空间中的像素时，你需要减去0.5个单位

参考自：http://www.cnblogs.com/ttthink/articles/1577987.html

顶 0　　踩 0

上一篇　3D世界空间单位的理解

下一篇　ogre1.8自己写的建立地形源码（注释，可直接使用）

我的同类文章

图形学（5）　　Direct3D（15）

• 关于背面消影和深度测试区别	2014-05-19	阅读 346	• 好-纹理和材质区别总结	2014-05-19	阅读 380
• 矩阵的本质-运动的描述【…	2013-09-27	阅读 493	• 一个游戏程序员的学习资料…	2013-07-16	阅读 997
• 四元数入门（简单容易理解）	2013-07-10	阅读 1028			

猜你在找

HTML 5移动开发从入门到精通
webgl基础篇-坚如磐石
韦东山嵌入式Linux第一期视频
精讲精练_参悟Android核心技术
数据结构和算法
AGG 光栅化Scanline Rasterizer
中点Bresenham算法光栅化画圆八分法
GPU大百科全书 第二章 凝固生命的光栅化
中点Bresenham算法光栅化画椭圆四分法
OpenGL中的原语组装和光栅化

50分钻戒价格　外汇ea　ea智能交易　上海婚纱照　网站设计　嘉定房价　上海嘉定房价

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