昵称： oayx
园龄： 10年9个月
粉丝： 794
关注： 0
+加关注

< 2020年9月 >

日	一	二	三	四	五	六
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	1	2	3
4	5	6	7	8	9	10

搜索

找找看

谷歌搜索

常用链接

我的随笔

我的评论

我的参与

最新评论

我的标签

我的标签

特效编辑器需求(1)

随笔分类

3DMax(8)

AI(3)

algorithm(8)

android(3)

AS(18)

boost(4)

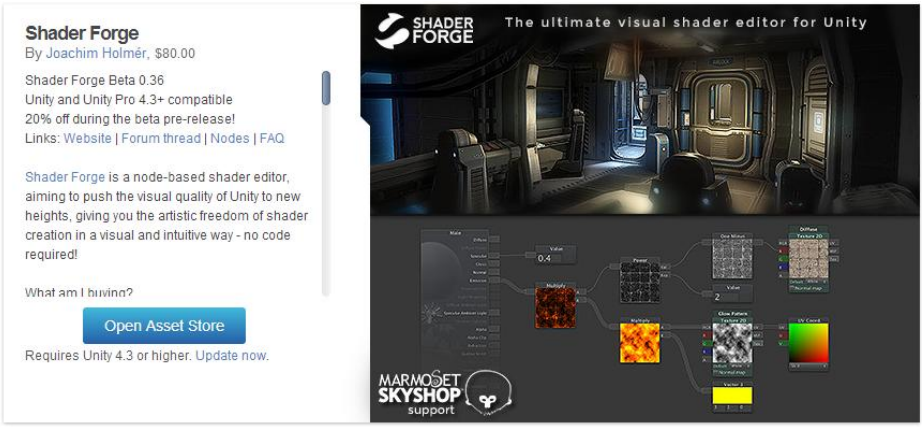
bullet(10)

c#(15)

Unity ShaderLab学习总结

Why Bothers?

为什么已经有ShaderForge这种可视化Shader编辑器、为什么Asset Store已经有那么多炫酷的Shader组件可下载，还是有必要学些Shader的编写?



2014-0718-1607-11-33.png

- 因为上面这些Shader工具/组件最终都是以Shader文件的形式而存在。
- 需要开发人员/技术美术有能力对Shader进行功能分析、效率评估、选择、优化、兼容、甚至是Debug。
- 对于特殊的需求，可能还是直接编写Shader比较实际、高效。

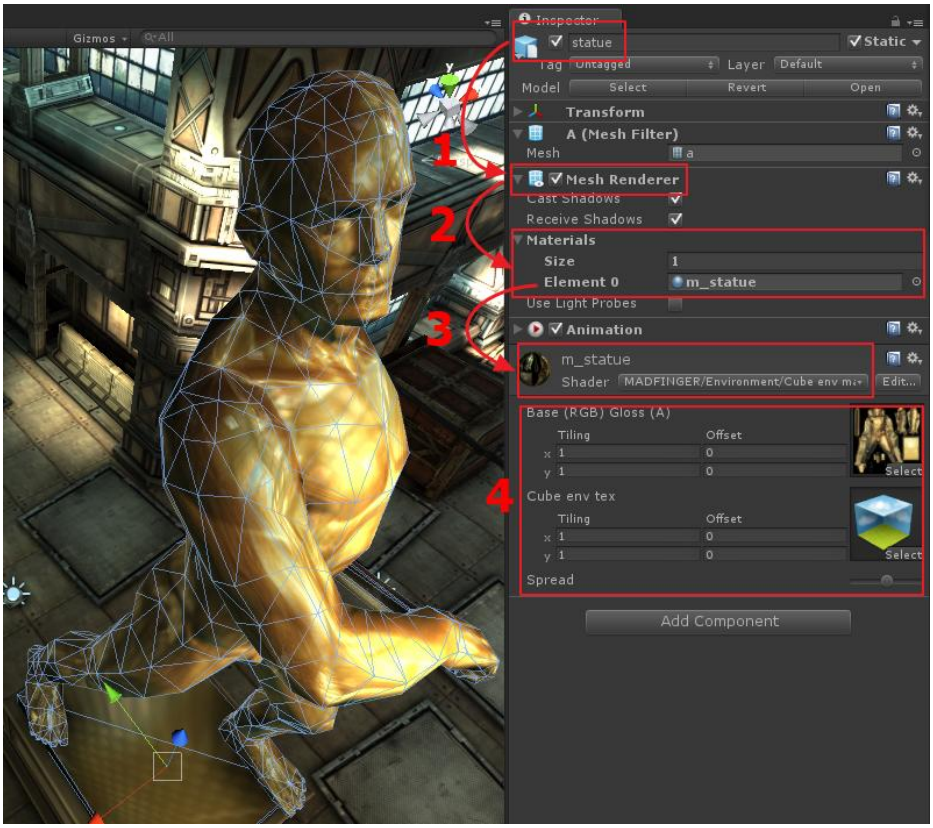
总之，Shader编写是重要的；但至于紧不紧急，视乎项目需求。

涉及范围

本文只讨论Unity ShaderLab相关的知识和使用方法。但，

- 既不讨论渲染相关的基础概念，基础概念可参考Rendering Pipeline Overview等文章。
- 也不讨论具体的渲染技巧
- 移动设备GPU和桌面设备GPU硬件架构上有较多不同点，详见下面的"移动设备GPU架构简述"一章。

使用Shader



2014-0720-1007-25-36.png

如上图，一句话总结：

1. GameObject里有MeshRenderer，
2. MeshRenderer里有Material列表，
3. 每个Material里有且只有一个Shader；
4. Material在编辑器暴露该Shader的可调属性。

所以关键是怎么编写Shader。

Shader基础

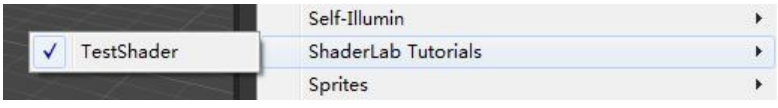
编辑器

使用MonoDevelop这反人类的IDE来编写Shader居然是让人满意的。有语法高亮，无语法提示。如果习惯VisualStudio，可以如下实现.Shader文件的语法高亮。

- 下载作者donaldwu自己添加的关键词文件usertype.dat。其包括了Unity ShaderLab的部分关键字，和HLSL的所有关键字。关键字以后持续添加中。
- 将下载的usertype.dat放到Microsoft Visual Studio xx.x\CommonX\IDE\文件夹下；
- 打开VS，工具>选项>文本编辑器>文件扩展名，扩展名里填"shader"，编辑器选VC++，点击添加；
- 重启VS，Done。

Shader

```
Shader "ShaderLab Tutorials/TestShader"
{
    // ...
}
```



2014-0720-1707-17-42.png

Shader的名字会直接决定shader在material里出现的路径

随笔档案

SubShader

```
Shader "ShaderLab Tutorials/TestShader" {
    SubShader
    {
        //...
    }
}
```

一个Shader有多个SubShader。一个SubShader可理解为一个Shader的一个*渲染方案*。即SubShader是为了针对不同的渲染情况而编写的。每个Shader至少1个SubShader、理论可以无限多个，但往往两三个就足够。
一个时刻只会选取一个SubShader进行渲染，具体SubShader的选取规则包括：

- 从上到下选取
- SubShader的标签、Pass的标签
  - 是否符合当前的"Unity渲染路径"
  - 是否符合当前的ReplacementTag
- SubShader是否和当前的GPU兼容
- 等

按此规则第一个被选取的SubShader将会用于渲染，未被选取的SubShader在这次渲染将被忽略。

SubShader的Tag

```
Shader "ShaderLab Tutorials/TestShader" {
    SubShader
    {
        Tags { "Queue"="Geometry+10" "RenderType"="Opaque" }
        //...
    }
}
```

SubShader内部可以有标签（Tags）的定义。Tag指定了这个SubShader的渲染顺序（时机），以及其他的一些设置。

- "RenderType"标签。Unity可以运行时替换符合特定RenderType的所有Shader。Camera.RenderWithShader或Camera.SetReplacementShader配合使用。Unity内置的RenderType包括：
  - "Opaque"：绝大部分不透明的物体都使用这个；
  - "Transparent"：绝大部分透明的物体、包括粒子特效都使用这个；
  - "Background"：天空盒都使用这个；
  - "Overlay"：GUI、镜头光晕都使用这个；
  - 用户也可以定义任意自己的RenderType这个标签所取的值。
  - 应注意，Camera.RenderWithShader或Camera.SetReplacementShader不要求标签只能是RenderType，RenderType只是Unity内部用于Replace的一个标签而已，你也可以自定义自己全新的标签用于Replace。
    比如，你为自己的ShaderA.SubShaderA1（会被Unity选取到的SubShader，常为Shader文件中的第一个SubShader）增加Tag为"Distort"="On"，然后将"Distort"作为参数replacementTag传给函数。此时，作为replacementShader实参的ShaderB.SubShaderB1中若有也有一模一样的"Distort"="On"，则此SubShaderB1将代替SubShaderA1用于本次渲染。
  - 具体可参考Rendering with Replaced Shaders
- "Queue"标签。定义渲染顺序。预制的值为
  - "Background"。值为1000。比如用于天空盒。
  - "Geometry"。值为2000。大部分物体在这个队列。不透明的物体也在这里。这个队列内部的物体的渲染顺序会有进一步的优化（应该是从近到远，early-z test可以剔除不需经过FS处理的片元）。其他队列的物体都是按空间位置的从远到近进行渲染。
  - "AlphaTest"。值为2450。已进行AlphaTest的物体在这个队列。
  - "Transparent"。值为3000。透明物体。

    - "Overlay"。值为4000。比如镜头光晕。
    - 用户可以定义任意值，比如"Queue"="Geometry+10"
- "ForceNoShadowCasting"，值为"true"时，表示不接受阴影。
- "IgnoreProjector"，值为"true"时，表示不接受Projector组件的投影。

另，关于渲染队列和Batch的*非官方经验总结*是，一帧的渲染队列的生成，依次决定于每个渲染物体的：

- Shader的RenderType tag，
- Renderer.SortingLayerID，
- Renderer.SortingOrder，
- Material.renderQueue（默认值为Shader里的"Queue"），
- Transform.z(ViewSpace)（默认为按z值从前到后，但当Queue是"Transparent"的时候，按z值从后到前）。

这个渲染队列决定了之后（可能有dirty flag的机制？）渲染器再依次遍历这个渲染队列，"同一种"材质的渲染物体合到一个Batch里。

Pass

```
Shader "ShaderLab Tutorials/TestShader" {
    SubShader {
        Pass
        {
            //...
        }
    }
}
```

一个SubShader（渲染方案）是由一个个Pass块来执行的。每个Pass都会消耗对应的一个DrawCall。在满足渲染效果的情况下尽可能地减少Pass的数量。

Pass的Tag

```
Shader "ShaderLab Tutorials/TestShader" {
    SubShader {
        Pass
        {
            Tags{ "LightMode"="ForwardBase" }
            //...
        }
    }
}
```

和SubShader有自己专属的Tag类似，Pass也有Pass专属的Tag。
其中最重要Tag是 "LightMode"，指定Pass和Unity的哪一种渲染路径（"Rendering Path"）搭配使用。除最重要的ForwardBase、ForwardAdd外，这里需额外提醒的Tag取值可包括：

- Always，永远都渲染，但不处理光照
- ShadowCaster，用于渲染产生阴影的物体
- ShadowCollector，用于收集物体阴影到屏幕坐标Buff里。

其他渲染路径相关的Tag详见下面章节"Unity渲染路径种类"。
具体所有Tag取值，可参考ShaderLab syntax: Pass Tags。

FallBack

```
Shader "ShaderLab Tutorials/TestShader"{
    SubShader { Pass {} }

    FallBack "Diffuse" // "Diffuse"即Unity预制的固有Shader
    // FallBack Off //将关闭FallBack
}
```

当本Shader的所有SubShader都不支持当前显卡，就会使用FallBack语句指定的另一个Shader。
FallBack最好指定Unity自己预制的Shader实现，因其一般能够在当前所有显卡运行。

Properties

```
Shader "ShaderLab Tutorials/TestShader"
{
    Properties {
    _Range ("My Range", Range (0.02,0.15)) = 0.07 // sliders
    _Color ("My Color", Color) = (.34, .85, .92, 1) // color
    _2D ("My Texture 2D", 2D) = "" {} // textures
    _Rect("My Rectangle", Rect) = "name" { }
    _Cube ("My Cubemap", Cube) = "name" { }
    _Float ("My Float", Float) = 1
    _Vector ("My Vector", Vector) = (1,2,3,4)

    // Display as a toggle.
    [Toggle] _Invert ("Invert color?", Float) = 0
    // Blend mode values
    [Enum(UnityEngine.Rendering.BlendMode)] _Blend ("Blend mode", Float) = 1
    //setup corresponding shader keywords.
    [KeywordEnum(Off, On)] _UseSpecular ("Use Specular",  Float) = 0
    }

    // Shader
    SubShader{
        Pass{
          //...
          uniform float4 _Color;
          //...
          float4 frag() : COLOR{ return fixed4(_Color); }
          //...
             #pragma multi_compile __ _USESPECULAR_ON
          }
    }

    //fixed pipeline
    SubShader    {
        Pass{
            Color[_Color]
        }
    }
}
```

- Shader在Unity编辑器暴露给美术的参数，通过Properties来实现。
- 所有可能的参数如上所示。主要也就Float、Vector和Texture这3类。
- 除了通过编辑器编辑Properties，脚本也可以通过Material的接口（比如SetFloat、SetTexture编辑）
- 之后在Shader程序通过[name]（固定管线）或直接name（可编程Shader）访问这些属性。
- 在每一个Property前面也能类似C#那样添加Attribute，以达到额外UI面板功能。详见MaterialPropertyDrawer.html。

Shader中的数据类型

有3种基本数值类型：float、half和fixed。
这3种基本数值类型可以再组成vector和matrix，比如half3是由3个half组成、float4x4是由16个float组成。

- float：32位高精度浮点数。
- half：16位中精度浮点数。范围是[-6万，+6万]，能精确到十进制的小数点后3.3位。
- fixed：11位低精度浮点数。范围是[-2, 2]，精度是1/256。

数据类型影响性能

- 精度够用就好。
  - 颜色和单位向量，使用fixed

- 其他情况，尽量使用`half`（即范围在[-6万，+6万]内、精确到小数点后3.3位）；否则才使用`float`。

ShaderLab中的Matrix

当提到"Row-Major"、"Column-Major"，根据不同的场合，它们可能指不同的意思：

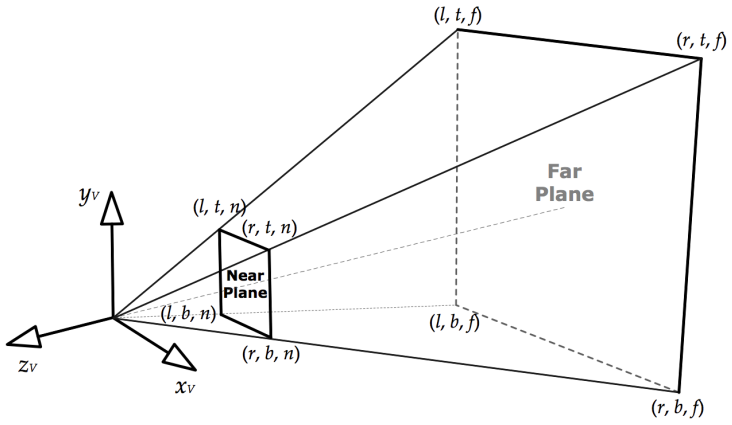
- 数学上的，主要是指矢量V是Row Vector、还是Column Vector。引用自[Game Engine Architecture 2nd Edition, 183]。留意到V和M的乘法，当是Row Vector的时候，数学上写作VM，Matrix在右边，Matrix的最下面一行表示Translate；当是Column Vector的时候，数学上写作MtVt，Matrix在左边并且需要转置，Matrix最右面一列表示Translate。
- 访问接口上的：Row-Major即MyMatrix[Row][Column]、Column-Major即MyMatrix[Column][Row]。HLSL/CG的访问接口都是Row-Major，比如MyMatrix[3]返回的是第3行；GLSL的访问接口是Column-Major，比如MyMatrix[3]返回的是第3列。
- 寄存器存储上的：每个元素是按行存储在寄存器中、还是按列存储在寄存器中。需要关注它的一般情况举例是，float2x3的MyMatrix，到底是占用2个寄存器（Row-Major）、还是3个寄存器（Column-Major）。在HLSL里，可以通过#pragmapack_matrix设定row_major或者column_major。

上述情况，互不相干。
然后，ShaderLab中，数学上是Column Vector、访问接口上是Row-Major、存储上是（尚未查明）。

ShaderLab中各个Space的坐标系

一般情况下，从Vertex Buff输入顶点到Vertex Shader，

- 该顶点为左手坐标系Model Space中的顶点`vInModel`，
  其用w=1的Homogenous Cooridniates（故等效于Cartesian Coordinates）表达`vInModel = float4(xm, ym, zm, 1);`
- `vInWrold = mul(_Object2World , vInModel)`后，得出左手坐标系World Space中的`vInWorld`，其为w=1的Homogenous Cooridniates（故等效于Cartesian Coordinates）`vInWorld = float4(xw, yw, zw, 1);`
- `vInView = mul(UNITY_MATRIX_V , vInWrold)`后，得出*右手坐标系*View Space中的`vInView`，其为w=1的Homogenous Cooridniates（故等效于Cartesian Coordinates）`vInWorld = float4(xv, yv, zv, 1);`
- `vInClip = mul(UNITY_MATRIX_P , vInView)`后，得出左手坐标系Clip Space中的`vInClip`，其为*w往往不等于1*的Homogenous Cooridniates（故*往往不等效于*Cartesian Coordinates）`vInClip = float4(xc, yc, zc, wc);`
  设r、l、t、b、n、f的长度绝对值如下图：



注意View Space中摄像机前方的z值为负数、-z为正数。则GL/DX/Metal的Clip Space坐标为：

- GL:
  - `xc=(2nx+rz+lz)/(r-l);`
  - `yc=(2ny+tz+bz)/(t-b);`
  - `zc=(-fz-nz-2nf)/(f-n);`
  - `wc=-z;`

4. 关于技术美术的一些个人理解(5)

5. 跨平台的游戏客户端Socket封装(4)

推荐排行榜

1. C++内存管理(45)

2. 完成端口(Completion Port)详解(20)

3. Sobel边缘检测算法(10)

4. IOCP模型与EPOLL模型的比较(7)

5. 死锁产生的原因及四个必要条件(5)

    - DX/Metal:
        - xc=(2nx+rz+lz)/(r-l);
        - yc=(2ny+tz+bz)/(t-b);
        - zc=(-fz-nf)/(f-n);
        - wc=-z;

- vInNDC = vInClip / vInClip.w后，得出左手坐标系Normalized Device Coordinates中的vInNDC，其为w=1的Homogenous Cooridniates（故等效于Cartesian Coordinates）vInNDC = float4(xn, yn, zn, 1)。
  xn和yn的取值范围为[-1,1]。
    - GL: zn=zc/wc=(fz+nz+2nf)/((f-n)z);
    - DX/Metal: zn=zc/wc=(fz+nf)/((f-n)z);
    - 在Unity中，zn的取值范围可以这样决定:
        - 如果UNITY_REVERSED_Z已定义，zn的取值范围是[UNITY_NEAR_CLIP_VALUE, 0]，即[1, 0]
        - 如果UNITY_REVERSED_Z未定义，zn的取值范围是[UNITY_NEAR_CLIP_VALUE, 1]
            - 如果SHADER_API_D3D9/SHADER_API_D3D11_9X定义了，即[0,1]
            - 否则，即OpenGL情况，即[-1,1]

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
    // 1 、2、3是等价的，和4是不等价的
    // 因为是M在左、V在右，所以是Column Vector
    // 因为是HLSL/CG语言，所以是访问方式是Row-Major
    o.rootInView = mul(UNITY_MATRIX_MV, float4(0, 0, 0, 1)); // 1
    o.rootInView = float4(UNITY_MATRIX_MV[0].w, UNITY_MATRIX_MV[1].w, UNITY_MATRIX_MV[2].w,
    o.rootInView = UNITY_MATRIX_MV._m03_m13_m23_m33;  // 3
    //o.rootInView = UNITY_MATRIX_MV[3]; // 4

    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    // 因为是ViewSpace是右手坐标系，所以当root在view前面的时候，z是负数，所以需要-z才能正确
    fixed4 col = fixed4(i.rootInView.x, i.rootInView.y, -i.rootInView.z, 1);
    return col;
}

struct appdata
{
    float4 vertex : POSITION;
};
struct v2f
{
    float4 rootInView : TEXCOORD0;
    float4 vertex : SV_POSITION;
};
```

Shader形态

Shader形态之1：固定管线

固定管线是为了兼容老式显卡。都是顶点光照。之后固定管线可能是被Unity抛弃的功能，所以最好不学它、当它不存在。特征是里面出现了形如下面Material块、没有CGPROGRAM和ENDCG块。

```
Shader "ShaderLab Tutorials/TestShader"
{
    Properties {
    _Color ("My Color", Color) = (.34, .85, .92, 1) // color
    }
```

```
    // Fixed Pipeline
    SubShader
    {
        Pass
        {
            Material{
            Diffuse [_Color]
            Ambient [_Color]
            }

            Lighting On
        }
    }
}
```

Shader形态之2：可编程Shader

```
Shader "ShaderLab Tutorials/TestShader"
{
    Properties {}

    SubShader
    {
        Pass
        {
          // ... the usual pass state setup ...

          CGPROGRAM
          // compilation directives for this snippet, e.g.:
          #pragma vertex vert
          #pragma fragment frag

          // the Cg/HLSL code itself
          float4 vert(float4 v:POSITION) : SV_POSITION{
              return mul(UNITY_MATRIX_MVP, v);
          }
          float4 frag() : COLOR{
              return fixed4(1.0, 0.0, 0.0, 1.0);
          }
          ENDCG
          // ... the rest of pass setup ...
          }
    }
}
```

- 功能最强大、最自由的形态。
- 特征是在Pass里出现CGPROGRAM和ENDCG块
- 编译指令#pragma。详见官网Cg snippets。其中重要的包括：

编译指令	示例/含义
#pragma vertex name #pragma fragment name	替换name，来指定Vertex Shader函数、Fragment Shader函数。
#pragma target name	替换name（为2.0、3.0等）。设置编译目标shader model的版本。
#pragma only_renderers name name ... #pragma exclude_renderers name name...	#pragma only_renderers gles gles3, #pragma exclude_renderers d3d9 d3d11 opengl, 只为指定渲染平台（render platform）编译

- 引用库。通过形如#include "UnityCG.cginc"引入指定的库。常用的就是UnityCG.cginc了。其他库详见官网Built-in shader include files。
- ShaderLab内置值。Unity给Shader程序提供了便捷的、常用的值，比如下面例子中的UNITY_MATRIX_MVP就代表了这个时刻的MVP矩阵。详见官网ShaderLab built-in values。

- Shader输入输出参数语义（Semantics）。在管线流程中每个阶段之间（比如Vertex Shader阶段和FragmentShader阶段之间）的输入输出参数，通过语义字符串，来指定参数的含义。常用的语义包括：COLOR、SV_Position、TEXCOORD[n]。完整的参数语义可见HLSL Semantic（由于是HLSL的连接，所以可能不完全在Unity里可以使用）。
- 特别地，因为Vertex Shader的的输入往往是管线的最开始，Unity为此内置了常用的数据结构：

数据结构	含义
appdata_base	vertex shader input with position, normal, one texture coordinate.
appdata_tan	vertex shader input with position, normal, tangent, one texture coordinate.
appdata_full	vertex shader input with position, normal, tangent, vertex color and two texture coordinates.
appdata_img	vertex shader input with position and one texture coordinate.

Shader形态之3：SurfaceShader

```
Shader "ShaderLab Tutorials/TestShader"
{
    Properties {   }

    // Surface Shader
    SubShader {
      Tags { "RenderType" = "Opaque" }
      CGPROGRAM
      #pragma surface surf Lambert
      struct Input {
          float4 color : COLOR;
      };
      void surf (Input IN, inout SurfaceOutput o) {
          o.Albedo = 1;
      }
      ENDCG
    }
    FallBack "Diffuse"
}
```

- SurfaceShader可以认为是一个光照Shader的*语法糖*、一个光照VS/FS的*生成器*。减少了开发者写重复代码的需要。
- 在手游，由于对性能要求比较高，所以*不建议使用SurfaceShader*。因为SurfaceShader是一个比较"通用"的功能，而通用往往导致性能不高。
- 特征是在SubShader里出现CGPROGRAM和ENDCG块。（而不是出现在Pass里。因为SurfaceShader自己会编译成多个Pass。）
- 编译指令是：
  `#pragma surface surfaceFunction lightModel [optionalparams]`
  - surfaceFunction：surfaceShader函数，形如`void surf (Input IN, inout SurfaceOutput o)`
  - lightModel：使用的光照模式。包括Lambert（漫反射）和BlinnPhong（镜面反射）。
    - 也可以自己定义光照函数。比如编译指令为`#pragma surface surf MyCalc`
      - 在Shader里定义`half4 LightingMyCalc (SurfaceOutput s, 参数略)`函数进行处理(函数名在签名加上了"Lighting")。
      - 详见Custom Lighting models in Surface Shaders
- 你定义输入数据结构（比如上面的Input）、编写自己的Surface函数处理输入、最终输出修改过后的SurfaceOutput。SurfaceOutput的定义为

```
struct SurfaceOutput {
  half3 Albedo; // 纹理颜色值（r, g, b)
  half3 Normal; // 法向量(x, y, z)
```

```
    half3 Emission; // 自发光颜色值(r, g, b)
    half Specular; // 镜面反射度
    half Gloss; // 光泽度
    half Alpha; // 不透明度
};
```

Shader形态之4：Compiled Shader

点击a.shader文件的“Compile and show code”，可以看到该文件的“编译”过后的ShaderLab shader文件，文件名形如Compiled-a.shader。
其依然是ShaderLab文件，其包含最终提交给GPU的shader代码字符串。
先就其结构进行简述如下，会发现和上述的编译前ShaderLab结构很相似。

```
// Compiled shader for iPhone, iPod Touch and iPad, uncompressed size: 36.5KB
// Skipping shader variants that would not be included into build of current scene.
Shader "ShaderLab Tutorials/TestShader"
{
    Properties {...}
    SubShader {
        // Stats for Vertex shader:
        //        gles : 14 avg math (11..19), 1 avg texture (1..2)
        //       metal : 14 avg math (11..17)
        // Stats for Fragment shader:
        //       metal : 14 avg math (11..19), 1 avg texture (1..2)
        Pass {
            Program "vp" // vertex program
            {
                SubProgram "gles" {
                    // Stats: 11 math, 1 textures
                    Keywords{...} // keywords for shader variants ("uber shader")

                    //shader codes in string
                    "
                    #ifdef VERTEX
                    vertex shader codes
                    #endif

                    // Note, on gles, fragment shader stays here inside Program "vp"
                    #ifdef FRAGMENT
                    fragment shader codes
                    #endif
                    "
                }

                SubProgram "metal"  {
                    some setup
                    Keywords{...}

                    //vertex shader codes in string
                    "..."
                }
            }

            Program "fp" // fragment program
            {
                SubProgram "gles" {
                    Keywords{...}
                    "// shader disassembly not supported on gles" //(because gles fragment
                }

                SubProgram "metal" {
                    common setup
                    Keywords{...}

                    //fragment shader codes in string
                    "..."
                }
```

```
            }
        }
    }

    ...
}
```

Unity渲染路径（Rendering Path）种类

概述

开发者可以在Unity工程的PlayerSettings设置对渲染路径进行3选1：

- *Deferred Lighting*，延迟光照路径。3者中最高质量地还原光照阴影。光照性能只与最终像素数目有关，光源数量再多都不会影响性能。
- *Forward Rendering*，顺序渲染路径。能发挥出Shader全部特性的渲染路径，当然也就支持像素级光照。最常用、功能最自由，性能与光源数目*受光照物体数目有关，具体性能视乎其具体使用到的Shader的复杂度。
- *Vertex Lit*，顶点光照路径。顶点级光照。性能最高、兼容性最强、支持特性最少、品质最差。

渲染路径的内部阶段和Pass的LightMode标签

每个渲染路径的内部会再分为几个阶段。
然后，Shader里的每个Pass，都可以指定为不同的LightMode。而LightMode实际就是说：“我希望这个Pass在这个XXX渲染路径的这个YYY子阶段被执行”。

Deferred Ligting

渲染路径内部子阶段	对应的Light Mode	描述
Base Pass	`"PrepassBase"`	渲染物体信息。即把法向量、高光度到一张ARGB32的*物体信息纹理*上，把深度信息保存在Z-Buff上。
Lighting Pass	无对应可编程Pass	根据Base Pass得出的物体信息，在屏幕坐标系下，使用BlinnPhong光照模式，把光照信息渲染到ARGB32的*光照信息纹理*上（RGB表示diffuse颜色值、A表示高光度）
Final Pass	`"PrepassFinal"`	根据*光照信息纹理*，物体再渲染一次，将光照信息、纹理信息和自发光信息最终混合。LightMap也在这个Pass进行。

Forward Rendering

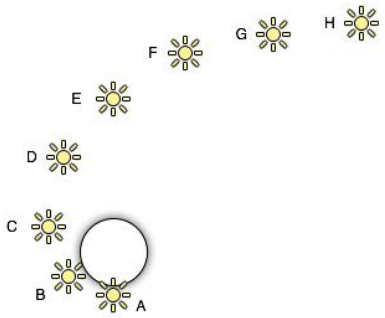
渲染路径内部子阶段	对应的Light Mode	描述
Base Pass	`"ForwardBase"`	渲染：最亮一个的方向光光源（像素级）和对应的阴影、所有顶点级光源、LightMap、所有LightProbe的SH光源（Sphere Harmonic，球谐函数，效率超高的低频光）、环境光、自发光。
Additional Passes	`"ForwardAdd"`	其他需要像素级渲染的的光源

注意到的是，在Forward Rendering中，光源可能是像素级光源、顶点级光源或SH光源。其判断标准是：

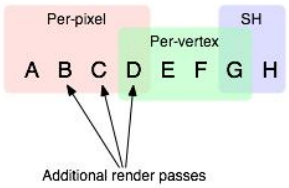
- 配制成“*Not Important*”的光源都是顶点级光源和SH光源
- *最亮的方向光*永远都是像素级光源
- 配置成“*Important*”的都是像素级光源
- 上面2种情况加起来的像素级光源数目小于“Quality Settings”里面的“*Pixel Light Count*”的话，会把第1种情况的光源补为额外的像素级光源。

另外，配置成“*Auto*”的光源有更复杂的判断标注，截图如下：

For example, if there is some object that's affected by a number of lights (a circle in a picture below, affected by lights A to H):



Let's assume lights A to H have the same color and intensity and all of them have Auto rendering mode, so they would be sorted in exactly this order for this object. The brightest lights will be rendered in per-pixel lit mode (A to D), then up to 4 lights in per-vertex lit mode (D to G), and finally the rest of lights in SH (G to H):



Note that light groups overlap; for example last per-pixel light blends into per-vertex lit mode so there are less "light popping" as objects and lights move around.

2014-0720-1607-31-40.png

具体可参考Forward Rendering Path Details。

Vertex Lit

渲染路径内部子阶段	对应的LightMode	描述
Vertex	"Vertex"	渲染无LightMap物体
VertexLMRGBM	"VertexLMRGBM"	渲染有RGBM编码的LightMap物体
VertexLM	"VertexLM"	渲染有双LDR编码的LightMap物体

不同LightMode的Pass的被选择

一个工程的渲染路径是唯一的，但一个工程里的Shader是允许配有不同LightMode的Pass的。
在Unity，策略是"从工程配置的渲染路径模式开始，按Deferred、Forward、VertxLit的顺序，搜索最匹配的LightMode的一个Pass"。
比如，在配置成Deferred路径时，优先选有Deferred相关LightMode的Pass；找不到才会选Forward相关的Pass；还找不到，才会选VertexLit相关的Pass。
再比如，在配置成Forward路径时，优先选Forward相关的Pass；找不到才会选VertexLit相关的Pass。

- d3d11 - Direct3D 11/12
- glcore - OpenGL 3.x/4.x
- gles - OpenGL ES 2.0
- gles3 - OpenGL ES 3.x
- metal - iOS/Mac Metal
- vulkan - Vulkan
- d3d11_9x - Direct3D 11 9.x feature level, as commonly used on WSA platforms
- xboxone - Xbox One
- ps4 - PlayStation 4
- psp2 - PlayStation Vita

- n3ds - Nintendo 3DS
- wiu - Nintendo Wii U

分类: unity3D

oayx
关注 - 0
粉丝 - 794
+加关注

1 0

« 上一篇: shader内置变量
» 下一篇: unity 质量设置 Quality Settings

posted @ 2017-06-08 20:46 oayx 阅读(2587) 评论(0) 编辑 收藏