【翻译】第二十章节：凹凸表面光照（关于法线贴图）

2014-12-12 08:08:00 1921 人阅读 Unity3D cg 法线贴图

原文链接：http://en.wikibooks.org/wiki/Cg_Programming/Unity/Lighting_of_Bumpy_Surfaces

本教程介绍 normal mapping。

本篇是系列教程中第一篇超越二维表面（或表层）纹理的技术，在这个教程中，我们先从法线贴图开始，因为法线贴图是一个非常成熟的技术，它主要用来伪造小的凹凸光照——即使是在粗糙的多边形网格中，本篇教程基于《平滑的镜面高光》和《纹理球》两个章节。

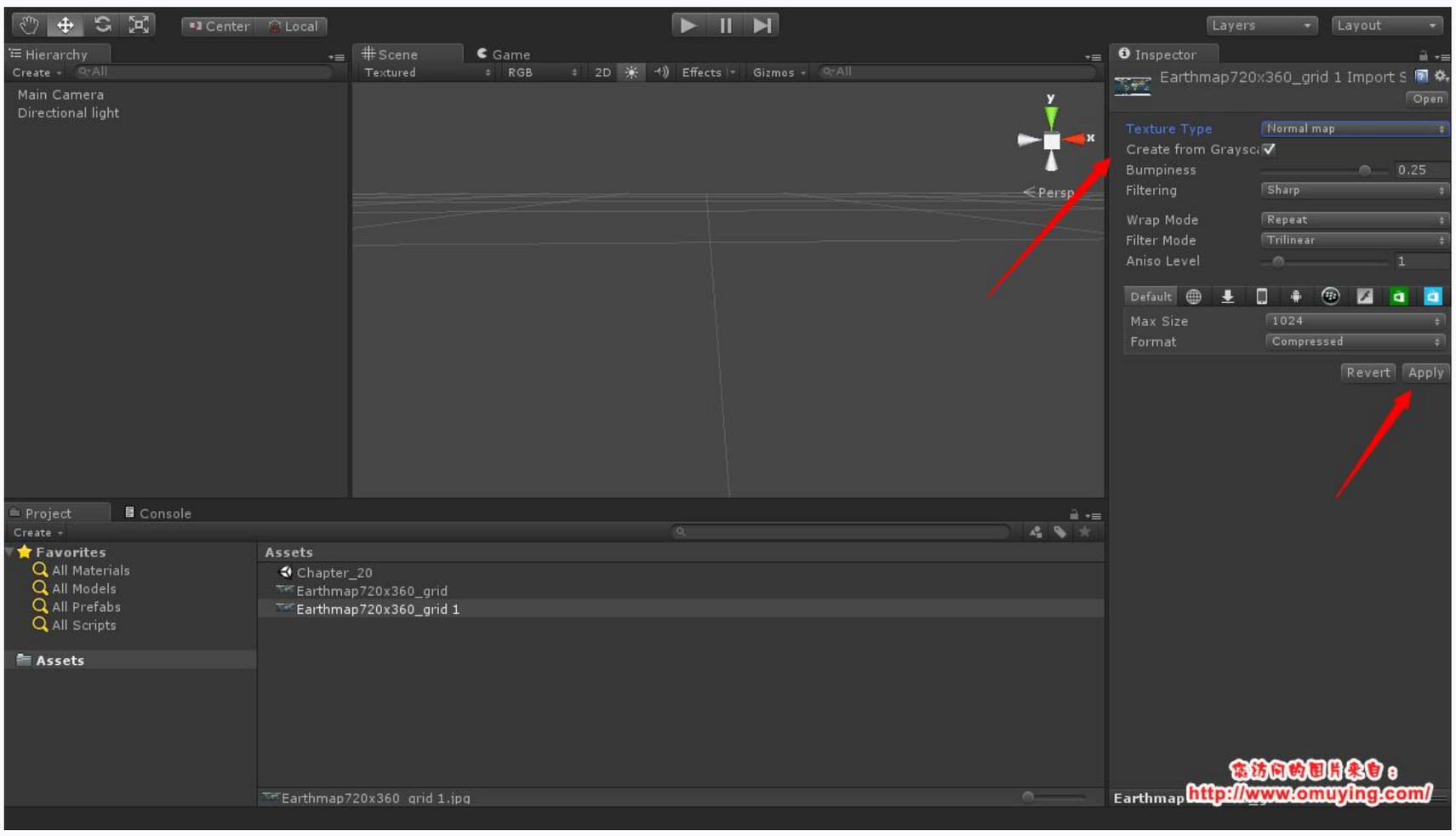
法线贴图

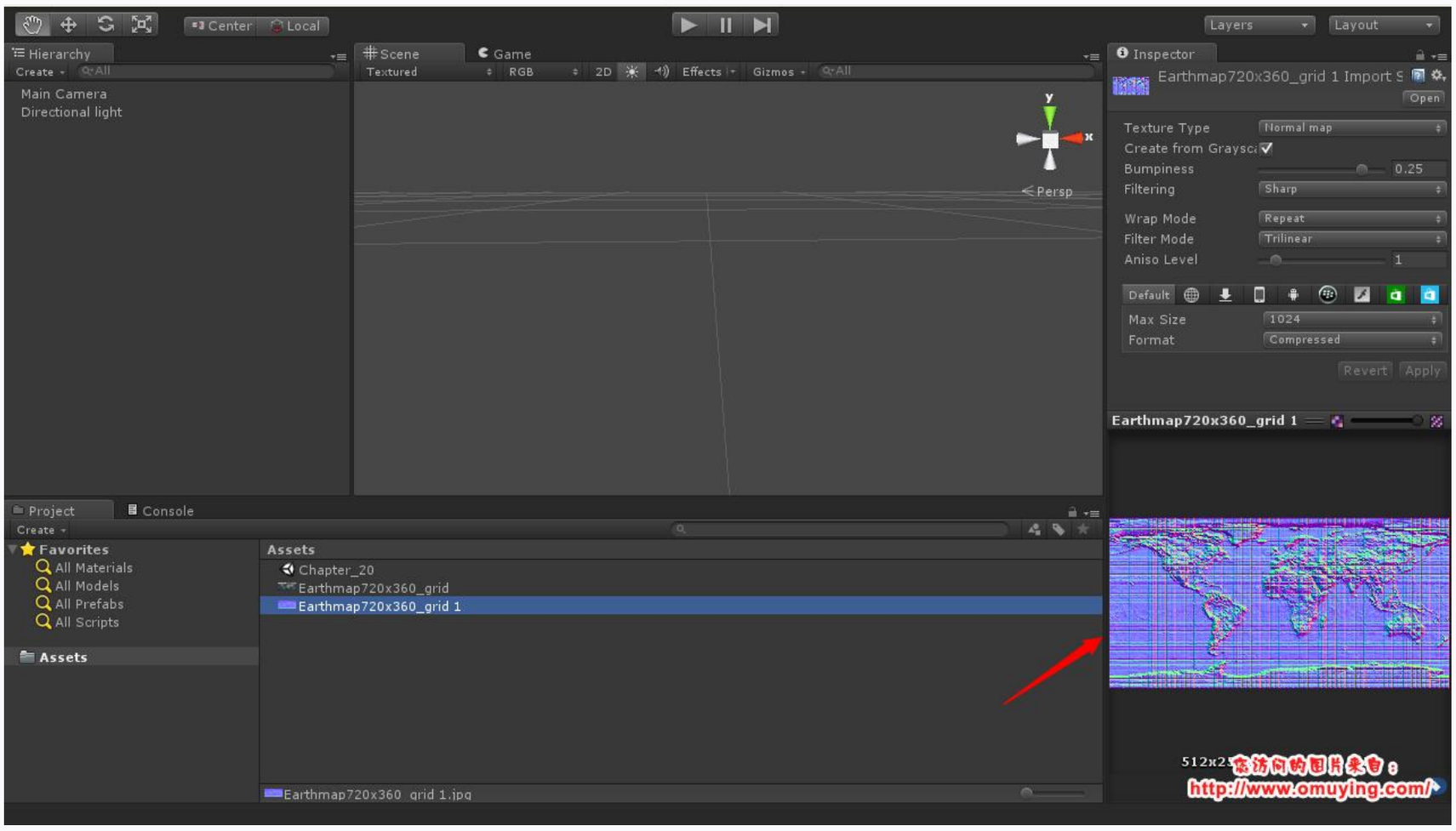
法线贴图试图根据一些虚拟的凸起来改变表面法线向量，然后在平滑的表面上表现凸起的效果（在三角形网格中进行法线插值），当用这些被修改的法线向量来计算光照时，观察者就可以看到虚拟的凸起——即使非常平坦的三角形已被渲染，但是这种错觉肯定会被看穿（尤其是在轮廓上），但是这在很多情况下它都令人相信。

更具体的说，表示虚拟凸起的法线向量首先被编码在一个纹理图像中（即法线贴图），然后片段着色器在纹理图像中查找这些向量，最后基于这些向量来计算光照，那么如何把法线向量编码到纹理图像中？这有不同的方法，但是片段着色器必须能够解码法线贴图。

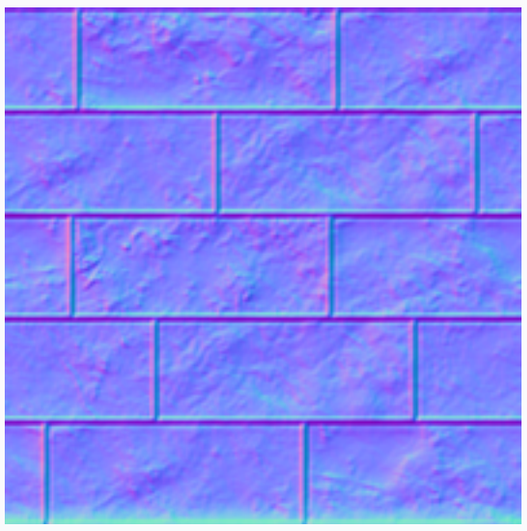
Unity 中的法线贴图

一个好的消息是，Unity 通过灰度图（gray-scale images）可以很容易生成法线贴图，首先用你最喜欢的画图工具生成一张灰度图（gray-scale image），并在表面指定一个灰色作为常规高度，然后比这个亮的灰色作为凸起，比这个暗的灰色作为凹陷，同时还应该确保不同灰色之间的过渡是平滑的，它看起来像一个模糊的图像。你可以通过 Assets > Import New Asset 导入图像，然后改变 Inspector 视图中的 Texture Type 为 Normal map，并选中 Create from Grayscale 属性，然后点击 Apply 按钮，这样在预览窗口中应该会显示一个带有红、绿边的蓝色图像，如下面两张图：





另外你也可以直接导入下面已编码过的法线贴图（但是不要忘记取消选中 Create from Grayscale 属性）：



不好的消息是片段着色器必须做一些计算来解码法线，首先纹理颜色被存储在一个双分组（two-component）纹理图像中，即只有 alpha 分量 A 和一个颜色分量是可用的，颜色分量在所有情况下访问红、绿、蓝分量都返回相同的值，这儿我们使用绿色分量 G，因为 Unity 也使用它，G 和 A 两个分量值的范围是 0 至 1，因此，他们对应坐标 n_x 和 n_y 的值的范围是 -1 至 1，映射公式为：

$$n_x = 2A - 1 \text{和} n_y = 2G - 1$$

根据这两个分量，法线向量（n=(n_x, n_y, n_z)）的第三个分量 n_z 的值也可以被计算，因为向量规范化长度公式为：

$$\sqrt{n_x^2 + n_y^2 + n_z^2} = 1 \quad \Rightarrow n_z = \pm\sqrt{1 - n_x^2 - n_y^2}$$

如果我们选择 z 轴作为垂直于表面的轴，那么只需要计算“+”解决方案就可以了，因为 z 轴法线向量不应该指向表面的内部，并且也不应该向内部渲染，所以片段着色中的代码应该像下面这样：

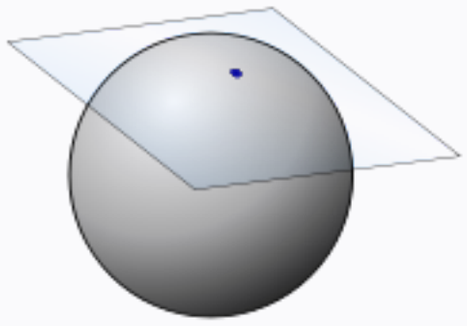
```
float4 encodedNormal = tex2D(_BumpMap, _BumpMap_ST.xy * input.tex.xy + _BumpMap_ST.zw);
float3 localCoords = float3(2.0 * encodedNormal.a - 1.0, 2.0 * encodedNormal.g - 1.0, 0.0);
localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
// approximation without sqrt:  localCoords.z =
// 1.0 - 0.5 * dot(localCoords, localCoords);
```

对于使用 OpenGL ES 设备的解码比较简单，因为 Unity 在这种情况下不会使用双分组纹理（two-component），因此对于手机平台的解码应该像这样：

```
float4 encodedNormal = tex2D(_BumpMap, _BumpMap_ST.xy * input.tex.xy + _BumpMap_ST.zw);
float3 localCoords = 2.0 * encodedNormal.rgb - float3(1.0, 1.0, 1.0);
```

不过本章节的其他部分（以及《Projection of Bumpy Surfaces》章节）将只讨论桌面平台。

Unity 使用一个本地表面坐标系统为法线贴图中每个表面的点指定法线向量，如下面的图片：



在这个本地坐标系统中， z 轴可以根据法线向量 N 在世界空间中的插值数据获得 ， $x-y$ 平面作为表面点的切线平面，具体而言， x 轴由 Unity 顶点输入参数中的切线参数 T 指定（可以查看《在着色器中调试》章节），有了 x 和 z， y 可以在顶点着色器中通过叉积计算出来，即：B = N × T（B 是指向量的传统名称“binormal”）。

注意，法线向量 N 通过转置逆模型矩阵从对象空间转换到世界空间（因为它正交于表面，详情可以查看《Applying Matrix Transformations》章节），当表面的点被模型矩阵转换时切线向量 T 会被指定一个方向，副法线向量 B 是第三类向量（third class），因为它的转换是不同的，因此，最好的选择是首先把 N 和 T 转换到世界空间，然后在世界空间中使用 N 和 T 的叉积计算 B。

这些计算在顶点着色器中进行，比如：

```
struct vertexInput
{
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
};
struct vertexOutput
{
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    // position of the vertex (and fragment) in world space
    float4 tex : TEXCOORD1;
    float3 tangentWorld : TEXCOORD2;
    float3 normalWorld : TEXCOORD3;
    float3 binormalWorld : TEXCOORD4;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // unity_Scale.w is unnecessary

    output.tangentWorld = normalize(mul(modelMatrix, float4(input.tangent.xyz, 0.0)).xyz);
    output.normalWorld = normalize(mul(float4(input.normal, 0.0), modelMatrixInverse).xyz);
    output.binormalWorld = normalize(cross(output.normalWorld, output.tangentWorld)* input.tangent.w);
    // tangent.w is specific to Unity

    output.posWorld = mul(modelMatrix, input.vertex);
    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}
```

在 binormalWorld 计算中使用了 Unity 指定的 input.tangent.w 因子，由于是 Unity 提供的切线向量和法线贴图原因，我们必须乘以这个值。

在世界空间中有了规范化的 T、B 和 N 方向，我们根据一个矩阵很容易把法线贴图中的任何法线向量 n 从本地表面坐标系统映射到世界空间中，因为一个矩阵的列轴正好是一个向量，因此，在 3 * 3 矩阵映射 n 到世界空间中是：

$$\mathrm{M}_{\mathrm{surface}\rightarrow\mathrm{world}} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

在 Cg 中，实际上更加容易构建转置矩阵，因为矩阵是按行构建：

$$\mathrm{M}^{T}_{\mathrm{surface}\to\mathrm{world}} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix}$$

可以在片段着色器中像这样构建：

```
float3x3 local2WorldTranspose = float3x3(input.tangentWorld,
input.binormalWorld, input.normalWorld);
```

我们想用 local2WorldTranspose 来转换 n，因此，我们需要在矩阵中左乘 n，例如：

```
float3 normalDirection = normalize(mul(localCoords, local2WorldTranspose));
```

这样在世界空间中有一个新的法线向量，这样我们就可以像《平滑的镜面高光》章节中的那样来计算光照。

完成着色器代码

下面这个着色器只是简单的集成了上面讨论的代码，然后在两个 pass 中使用像素光照：

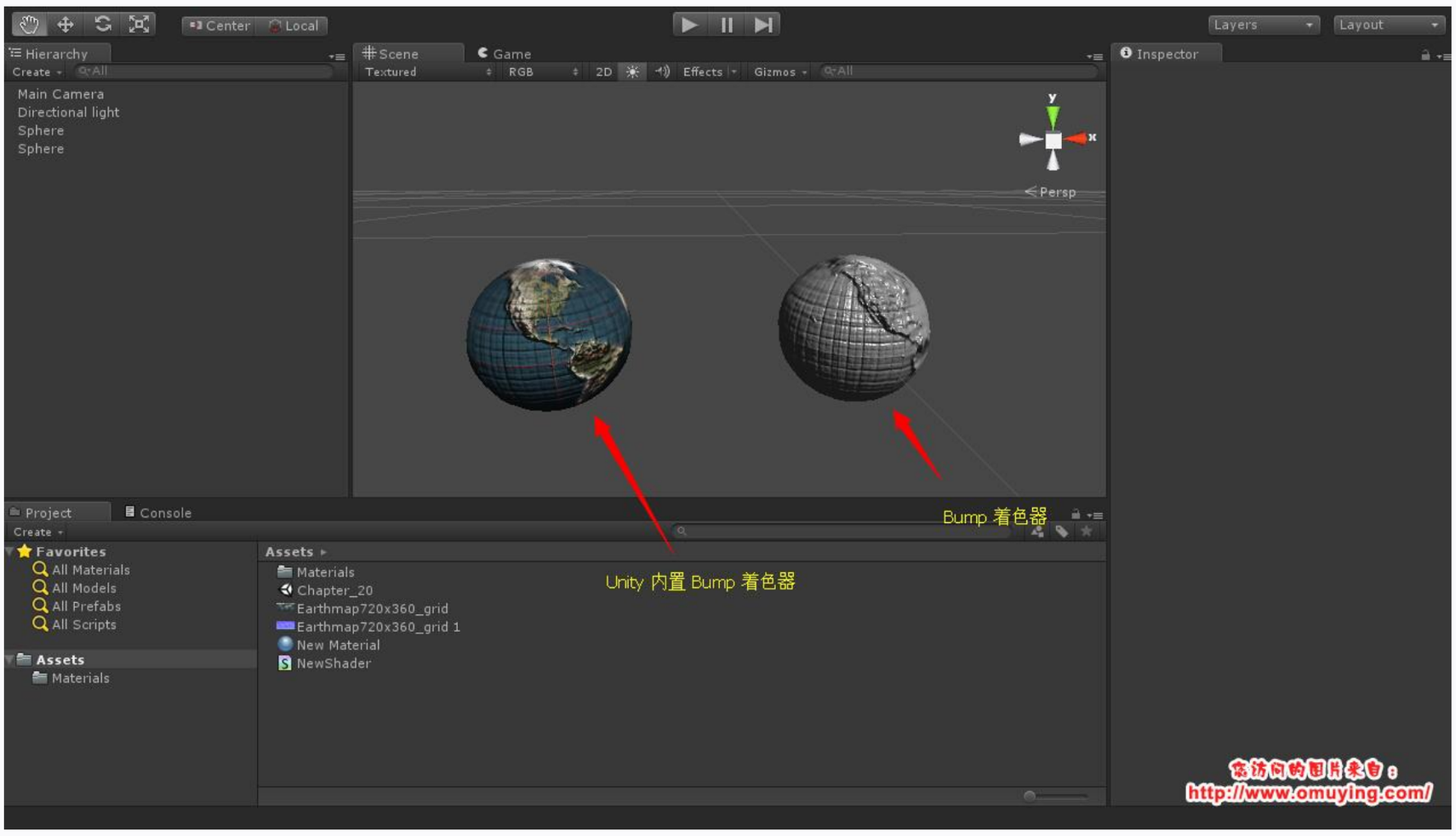
```
Shader "Cg normal mapping"
{
   Properties
   {
      _BumpMap ("Normal Map", 2D) = "bump" {}
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader
   {
      Pass
      {
         Tags { "LightMode" = "ForwardBase" }
         // pass for ambient light and first light source

         CGPROGRAM

         #pragma vertex vert
         #pragma fragment frag

         #include "UnityCG.cginc"
         uniform float4 _LightColor0;
         // color of light source (from "Lighting.cginc")

         // User-specified properties
         uniform sampler2D _BumpMap;
         uniform float4 _BumpMap_ST;
         uniform float4 _Color;
         uniform float4 _SpecColor;
         uniform float _Shininess;

         struct vertexInput
         {
            float4 vertex : POSITION;
            float4 texcoord : TEXCOORD0;
            float3 normal : NORMAL;
            float4 tangent : TANGENT;
         };
         struct vertexOutput
         {
            float4 pos : SV_POSITION;
            float4 posWorld : TEXCOORD0;
            // position of the vertex (and fragment) in world space
            float4 tex : TEXCOORD1;
            float3 tangentWorld : TEXCOORD2;
            float3 normalWorld : TEXCOORD3;
            float3 binormalWorld : TEXCOORD4;
         };

         vertexOutput vert(vertexInput input)
         {
            vertexOutput output;

            float4x4 modelMatrix = _Object2World;
            float4x4 modelMatrixInverse = _World2Object;
            // unity_Scale.w is unnecessary

            output.tangentWorld = normalize(mul(modelMatrix,
float4(input.tangent.xyz, 0.0)).xyz);
            output.normalWorld = normalize(mul(float4(input.normal, 0.0),
modelMatrixInverse).xyz);
            output.binormalWorld = normalize(cross(output.normalWorld,
output.tangentWorld)* input.tangent.w);
            // tangent.w is specific to Unity
```

```
063
064             output.posWorld = mul(modelMatrix, input.vertex);
065             output.tex = input.texcoord;
066             output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
067             return output;
068          }
069
070          float4 frag(vertexOutput input) : COLOR
071          {
072             // in principle we have to normalize tangentWorld,
073             // binormalWorld, and normalWorld again; however, the
074             // potential problems are small since we use this
075             // matrix only to compute "normalDirection",
076             // which we normalize anyways
077
078             float4 encodedNormal = tex2D(_BumpMap, _BumpMap_ST.xy *
input.tex.xy + _BumpMap_ST.zw);
079             float3 localCoords = float3(2.0 * encodedNormal.a - 1.0, 2.0 *
encodedNormal.g - 1.0, 0.0);
080             localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
081             // approximation without sqrt:  localCoords.z =
082             // 1.0 - 0.5 * dot(localCoords, localCoords);
083
084             float3x3 local2WorldTranspose = float3x3(input.tangentWorld,
input.binormalWorld, input.normalWorld);
085             float3 normalDirection = normalize(mul(localCoords,
local2WorldTranspose));
086
087             float3 viewDirection = normalize(_WorldSpaceCameraPos -
input.posWorld.xyz);
088             float3 lightDirection;
089             float attenuation;
090
091             if (0.0 == _WorldSpaceLightPos0.w) // directional light?
092             {
093                attenuation = 1.0; // no attenuation
094                lightDirection = normalize(_WorldSpaceLightPos0.xyz);
095             }
096             else // point or spot light
097             {
098                float3 vertexToLightSource = _WorldSpaceLightPos0.xyz -
input.posWorld.xyz;
099                float distance = length(vertexToLightSource);
100                attenuation = 1.0 / distance; // linear attenuation
101                lightDirection = normalize(vertexToLightSource);
102             }
103
104             float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb *
 Color.rgb;
105
106             float3 diffuseReflection = attenuation * _LightColor0.rgb *
_Color.rgb * max(0.0, dot(normalDirection, lightDirection));
107
108             float3 specularReflection;
109             if (dot(normalDirection, lightDirection) < 0.0) // light source on
the wrong side?
110             {
111                specularReflection = float3(0.0, 0.0, 0.0);
112                // no specular reflection
113             }
114             else // light source on the right side
115             {
116                specularReflection = attenuation * _LightColor0.rgb *
_SpecColor.rgb * pow(max(0.0, dot(reflect(-lightDirection, normalDirection),
viewDirection)), _Shininess);
117             }
118             return float4(ambientLighting + diffuseReflection +
specularReflection, 1.0);
119          }
120          ENDCG
121       }
122
123       Pass
124       {
125          Tags { "LightMode" = "ForwardAdd" }
126          // pass for additional light sources
127          Blend One One // additive blending
128
129          CGPROGRAM
130
131          #pragma vertex vert
132          #pragma fragment frag
133
134          #include "UnityCG.cginc"
135          uniform float4 _LightColor0;
136          // color of light source (from "Lighting.cginc")
137
138          // User-specified properties
139          uniform sampler2D _BumpMap;
140          uniform float4 _BumpMap_ST;
141          uniform float4 _Color;
142          uniform float4 _SpecColor;
143          uniform float _Shininess;
```

```
144
145         struct vertexInput
146         {
147            float4 vertex : POSITION;
148            float4 texcoord : TEXCOORD0;
149            float3 normal : NORMAL;
150            float4 tangent : TANGENT;
151         };
152         struct vertexOutput
153         {
154            float4 pos : SV_POSITION;
155            float4 posWorld : TEXCOORD0;
156            // position of the vertex (and fragment) in world space
157            float4 tex : TEXCOORD1;
158            float3 tangentWorld : TEXCOORD2;
159            float3 normalWorld : TEXCOORD3;
160            float3 binormalWorld : TEXCOORD4;
161         };
162
163         vertexOutput vert(vertexInput input)
164         {
165            vertexOutput output;
166
167            float4x4 modelMatrix = _Object2World;
168            float4x4 modelMatrixInverse = _World2Object;
169            // unity_Scale.w is unnecessary
170
171            output.tangentWorld = normalize(mul(modelMatrix,
   float4(input.tangent.xyz, 0.0)).xyz);
172            output.normalWorld = normalize(mul(float4(input.normal, 0.0),
   modelMatrixInverse).xyz);
173            output.binormalWorld = normalize(cross(output.normalWorld,
   output.tangentWorld) * input.tangent.w);
174            // tangent.w is specific to Unity
175
176            output.posWorld = mul(modelMatrix, input.vertex);
177            output.tex = input.texcoord;
178            output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
179            return output;
180         }
181
182         float4 frag(vertexOutput input) : COLOR
183         {
184            // in principle we have to normalize tangentWorld,
185            // binormalWorld, and normalWorld again; however, the
186            // potential problems are small since we use this
187            // matrix only to compute "normalDirection",
188            // which we normalize anyways
189
190            float4 encodedNormal = tex2D(_BumpMap, _BumpMap_ST.xy *
   input.tex.xy + _BumpMap_ST.zw);
191            float3 localCoords = float3(2.0 * encodedNormal.a - 1.0, 2.0 *
   encodedNormal.g - 1.0, 0.0);
192            localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
193            // approximation without sqrt:  localCoords.z =
194            // 1.0 - 0.5 * dot(localCoords, localCoords);
195
196            float3x3 local2WorldTranspose = float3x3(input.tangentWorld,
   input.binormalWorld, input.normalWorld);
197            float3 normalDirection = normalize(mul(localCoords,
   local2WorldTranspose));
198
199            float3 viewDirection = normalize(_WorldSpaceCameraPos -
   input.posWorld.xyz);
200            float3 lightDirection;
201            float attenuation;
202
203            if (0.0 == _WorldSpaceLightPos0.w) // directional light?
204            {
205               attenuation = 1.0; // no attenuation
206               lightDirection = normalize(_WorldSpaceLightPos0.xyz);
207            }
208            else // point or spot light
209            {
210               float3 vertexToLightSource = _WorldSpaceLightPos0.xyz -
   input.posWorld.xyz;
211               float distance = length(vertexToLightSource);
212               attenuation = 1.0 / distance; // linear attenuation
213               lightDirection = normalize(vertexToLightSource);
214            }
215
216            float3 diffuseReflection = attenuation * _LightColor0.rgb *
   _Color.rgb * max(0.0, dot(normalDirection, lightDirection));
217
218            float3 specularReflection;
219            if (dot(normalDirection, lightDirection) < 0.0) // light source on
   the wrong side?
220            {
221               specularReflection = float3(0.0, 0.0, 0.0);
222               // no specular reflection
223            }
224            else // light source on the right side
225            {
```

```
                specularReflection = attenuation * _LightColor0.rgb *
_SpecColor.rgb * pow(max(0.0, dot(reflect(-lightDirection, normalDirection),
viewDirection)), _Shininess);
            }
            return float4(diffuseReflection + specularReflection, 1.0);
        }
        ENDCG
    }
  }
}
```

注意我们在着色器中使用 uniform _BumpMap_ST 是为了能够使用 tiling 和 offset 属性（详情可以查看《纹理球》章节），因为这两个选项上在凹凸贴图中很有用，我们在场景中添加两个球体，一个应用 Unity 内置的 Bump 着色器，一个应用上面的着色器，效果如图：



恭喜你，在本章节中你应该了解：

1、人们如何通过光照来观察事物形状。

2、什么是法线贴图。

3、Unity 如何编码法线贴图。

4、片段着色器如何解码 Unity 的法线贴图并计算光照。

资源下载地址：点击下载，共下载 28 次。

前一篇：第十九章节：纹理层（关于多重纹理）

后一篇：Unity3D 利用 ScriptableObject 把 Xml 打包成 assetbundle

说点什么吧...

发布

最终幻想正在使用多说

0条评论

最新 最早 最热

还没有评论，沙发等你来抢

社交帐号登录: 微信 微博 QQ 人人 更多»

说点什么吧...

发布

最终幻想正在使用多说