【翻译】第二十一章节：凹凸表面投影（关于视差贴图）

2014-12-15 08:42:00 1462 人阅读 Unity3D cg 视差贴图

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

这个教程介绍 parallax mapping，它是对《凹凸表面光照》章节中的扩展。

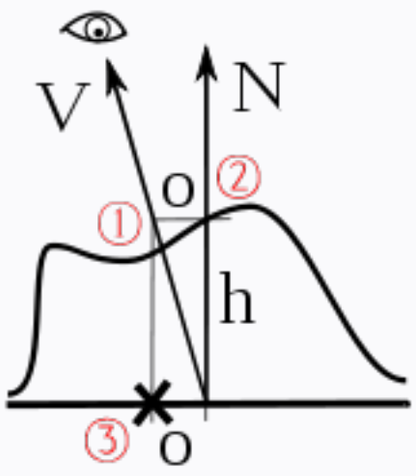
法线贴图的改进

在《凹凸表面光照》章节中，我们知道法线贴图技术只会改变平坦表面的光照来产生凹凸的错觉，如果直视表面（即表面法线的方向），这种办法就非常好，但是正常情况下如果我们从一个角度来观察表面，凸起的部分应该伸出表面，凹痕应该陷入表面，当然这些凹凸块可以通过建模来实现，但是这将会处理更多的顶点，这时我们可以使用单步视差贴图技术，它与法线贴图相似，它不需要添加额外三角形，但仍然可以通过几个像素来移动虚拟的凸起以使它们伸出表面，然而这种技术仅限于小高度的凹凸效果，并且还需要进行一些微调。

视差贴图原理

视差贴图在 2001 年由 Tomomichi Kaneko 等提出，它的基本思想是偏移表面纹理（特别是法线贴图）的纹理坐标，如果能够恰当的计算偏移的纹理坐标，它将能够移动纹理的一些部分（例如凸块）并在表面伸出。

如下图所示：



观察者的视图向量 V 和在片段着色器中被栅格化的表面点的法线向量 N，视差贴图有三个步骤：

> 1、在高度贴图的栅格化点中查找高度 h，它是图中波浪线顶部到底部的直线。
> 2、计算方向 V 的视图射线和平行于渲染表面并且高度为 h 的线的交点（①处），距离 o 是表面被栅格化点的法线向量 N 移动 h 方向之后（②处）和这个交点（①处）的距离，如果这两个点都在渲染表面投影， o 是表面上栅格化点和一个新点（③处）的距离，这个新点是个很好的近似点，如果表面被高度贴图替代（displaced）的话，那么对于方向 V 的视图射线这个点实际上也是可见的。
> 3、为了计算纹理坐标偏移值，我们需要转换偏移 o 到纹理坐标空间，这样是为了在纹理坐标中的查找所有的纹理偏移。

为了计算 o，我们需要高度贴图中栅格化点的高度 h，在这个例子中，它由纹理属性 _ParallaxMap 的 a 分量获得，_ParallaxMap 是一个灰度图用于表示高度，可以查看《凹凸表面光照》章节，我们还需要本地表面系统坐标（由法线向量（ z轴）、切线向量（ x轴）以及副法线向量（ y轴）构成）中的视图方向 V，最

最新文章

暂无图片 【原创】C# 基础之Lambda表达式 - 907 次阅读

暂无图片 【原创】C#基础之IEnumerable和IEnumerator - 792 次阅读

暂无图片 【原创】C#基础之事件 - 886 次阅读

暂无图片 【原创】C#基础之委托 - 912 次阅读

暂无图片 【原创】C#基础之委托的使用 - 856 次阅读

随机阅读

后我们可以像下面那样从本地表面坐标转换对象空间中：

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

其中 T、B、N 由对象坐标获得。

我们在对象空间（从世界空间转换到对象空间可以通过摄像机位置与栅格化点位置之间的差值来计算）中计算视图方向 V，然后用矩阵 $\mathrm{M}_{\mathrm{object}\rightarrow\mathrm{surface}}$ 转换到本地表面空间中：

$$\mathrm{M}_{\mathbf{object}\rightarrow\mathbf{surface}} = \mathrm{M}^{-1}_{\mathbf{surface}\rightarrow\mathbf{object}} = \mathrm{M}^{T}_{\mathbf{surface}\rightarrow\mathbf{object}}$$

上面的公式是可以的，因为 T、B、N 相互正交并且都是规范化的，但是这实际上有些复杂，为了在其他的转换中使用他们的长度，我们一般不会规范化这些向量，因此为了把 V 从对象空间转换到本地表面空间中，我们可以与转置矩阵 $\left(\mathrm{M}_{\mathrm{surface}\rightarrow\mathrm{object}}\right)^{T}$ 相乘，这样做其实非常好，因为在 Cg 中，我们通过把 T、B、N 作为转置矩阵的行向量可以很容易的构建转置矩阵。

一旦我们有了本地表面坐标系统（法线向量 N 的 z 轴方向组成）中的 V，我们可以通过相似三角形计算 o_x 偏移（ x 方向）和 o_y 偏移（ y 方向）：

$\frac{o_x}{h} = \frac{V_x}{V_z}$ 和 $\frac{o_y}{h} = \frac{V_y}{V_z}$

因此

$o_x = h\frac{V_x}{V_z}$ 和 $o_y = h\frac{V_y}{V_z}$

需要注意是的，我们没有必要规范化 V，因为我们只使用它的组成部分。

最后，我们必须把 o_x 和 o_y 转换到纹理空间中，如果 Unity 不帮助我们的话，这将是非常困难的：切线属性 tangent 实际上是适当缩放的，它的第四个分量 tangent.w 用于副法线向量的缩放，这样使得视图方向 V 的转换率 V_x 和 V_y 适当的拥有纹理坐标空间 o_x 和 o_y，因此无需进一步计算。

实现

实现视差贴图的着色器代码大部分与《凹凸表面光照》章节中的相同，特别指出的是 tangent 属性的第四个分量被使用，因为它与副法线向量缩放相同，这样是为了把本地表面空间的偏移映射到纹理空间：

```
float3 binormal = cross(input.normal, input.tangent.xyz) * input.tangent.w;
```

对于本地表面坐标系统中的视图方向 V（考虑轴缩放映射到纹理空间）我们必须添加一个输出参数，这个参数为 viewDirInScaledSurfaceCoords，它通过已转换到对象坐标（viewDirInObjectCoords）中的视图方向与矩阵 $\mathrm{M}^{T}_{\mathrm{surface}\rightarrow\mathrm{object}}$（localSurface2ScaledObjectT）计算：

```
float3 viewDirInObjectCoords = mul(modelMatrixInverse,
float4(_WorldSpaceCameraPos, 1.0)).xyz - input.vertex.xyz;
float3x3 localSurface2ScaledObjectT = float3x3(input.tangent.xyz, binormal,
input.normal);
// vectors are orthogonal
output.viewDirInScaledSurfaceCoords = mul(localSurface2ScaledObjectT,
viewDirInObjectCoords);
// we multiply with the transpose to multiply with
// the "inverse" (apart from the scaling)
```

顶点着色器的其他部分与法线贴图中的相同，这可以查看《凹凸表面光照》章节，所不同的是用顶点着色器中代替片段着色器计算世界坐标中的视图方向，这样做的目的为了减小 GPU 的消耗。

在片段着色器中，我们首先查询高度贴图中的栅格化点的高度，这个高度由纹理 _ParallaxMap 的 a 分量指定，这个值的范围在 0 至 1 之间，我们可以通过着色器属性 _Parallax 把范围转换到 -_Parallax/2 至 +_Parallax，这样做是可以给用户提供一引起强度的控制：

```
float height = _Parallax * (-0.5 + tex2D(_ParallaxMap, _ParallaxMap_ST.xy *
input.tex.xy + _ParallaxMap_ST.zw).x);
```

偏移 X大 和 Y大 被计算出来，但是我们也需要限制每个偏移值的范围在用户指定的 -_MaxTexCoordOffset

至 +_MaxTexCoordOffset 之间，这样做是为了确保偏移的值在合理的范围内（在高度图中，如果这些凸起或多或少的按照恒定的高度平滑的进行过渡，_MaxTexCoordOffset 应小于这些过渡区的厚度，否则采样点在不同的凸起之间具有不同的高度，这意味着交叉的近似值是比较糟糕的），代码如下：

```
float2 texCoordOffsets = clamp(height * input.viewDirInScaledSurfaceCoords.xy
/ input.viewDirInScaledSurfaceCoords.z, -_MaxTexCoordOffset,
+_MaxTexCoordOffset);
```

在下面的代码中，我们必须在纹理坐标的所有纹理查找中应用偏移，因此我们必须使用 (input.tex.xy + texCoordOffsets) 代替 float2(input.tex)，代码如下：

```
float4 encodedNormal = tex2D(_BumpMap, _BumpMap_ST.xy * (input.tex.xy +
texCoordOffsets) + _BumpMap_ST.zw);
```

片段着色器的其他部分与《凹凸表面光照》章节中的类似。

完成着色器代码

正如前面所讨论的那样，大部分的着色器代码与《凹凸表面光照》章节中的相同，如果你希望在移动设备上使用下面这个着色器，那么你一定要改变法线贴图的解码方式。

关于视差贴图的部分实际上只有几行代码，着色器中属性名称大部分是根据 fallback 着色器来选择的，用户界面的标签更具描述性，着色器的代码如下：

```
Shader "Cg parallax mapping"
{
   Properties
   {
      _BumpMap ("Normal Map", 2D) = "bump" {}
      _ParallaxMap ("Heightmap (in A)", 2D) = "black" {}
      _Parallax ("Max Height", Float) = 0.01
      _MaxTexCoordOffset ("Max Texture Coordinate Offset", Float) =
         0.01
      _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 sampler2D _ParallaxMap;
         uniform float4 _ParallaxMap_ST;
         uniform float _Parallax;
         uniform float _MaxTexCoordOffset;
         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;
            float3 viewDirWorld : TEXCOORD5;
```

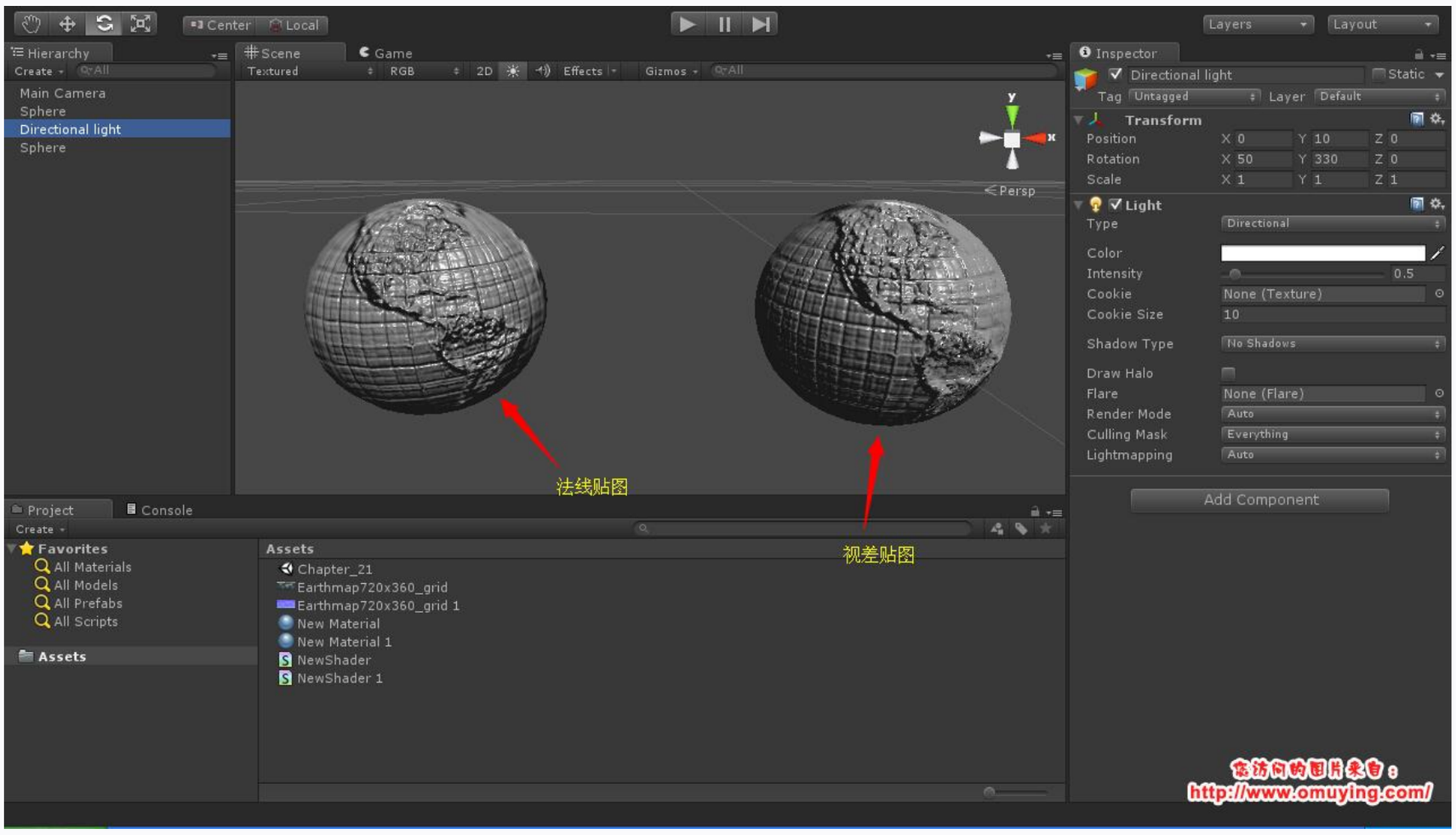
```
058             float3 viewDirInScaledSurfaceCoords : TEXCOORD6;
059          };
060
061          vertexOutput vert(vertexInput input)
062          {
063             vertexOutput output;
064
065             float4x4 modelMatrix = _Object2World;
066             float4x4 modelMatrixInverse = _World2Object;
067             // unity_Scale.w is unnecessary
068
069             output.tangentWorld = normalize(mul(modelMatrix,
    float4(input.tangent.xyz, 0.0)).xyz);
070             output.normalWorld = normalize(mul(float4(input.normal, 0.0),
    modelMatrixInverse).xyz);
071             output.binormalWorld = normalize(cross(output.normalWorld,
    output.tangentWorld)* input.tangent.w);
072             // tangent.w is specific to Unity
073
074             float3 binormal = cross(input.normal, input.tangent.xyz) *
    input.tangent.w;
075             // appropriately scaled tangent and binormal
076             // to map distances from object space to texture space
077
078             float3 viewDirInObjectCoords = mul(modelMatrixInverse,
    float4(_WorldSpaceCameraPos, 1.0)).xyz - input.vertex.xyz;
079             float3x3 localSurface2ScaledObjectT = float3x3(input.tangent.xyz,
    binormal, input.normal);
080             // vectors are orthogonal
081             output.viewDirInScaledSurfaceCoords =
    mul(localSurface2ScaledObjectT, viewDirInObjectCoords);
082             // we multiply with the transpose to multiply with
083             // the "inverse" (apart from the scaling)
084
085             output.posWorld = mul(modelMatrix, input.vertex);
086             output.viewDirWorld = normalize(_WorldSpaceCameraPos -
    output.posWorld.xyz);
087             output.tex = input.texcoord;
088             output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
089             return output;
090          }
091
092          float4 frag(vertexOutput input) : COLOR
093          {
094             // parallax mapping: compute height and
095             // find offset in texture coordinates
096             // for the intersection of the view ray
097             // with the surface at this height
098
099             float height = _Parallax * (-0.5 + tex2D(_ParallaxMap,
    _ParallaxMap_ST.xy * input.tex.xy + _ParallaxMap_ST.zw).x);
100
101             float2 texCoordOffsets = clamp(height *
    input.viewDirInScaledSurfaceCoords.xy / input.viewDirInScaledSurfaceCoords.z,
    -_MaxTexCoordOffset, +_MaxTexCoordOffset);
102
103             // normal mapping: lookup and decode normal from bump map
104
105             // in principle we have to normalize tangentWorld,
106             // binormalWorld, and normalWorld again; however, the
107             // potential problems are small since we use this
108             // matrix only to compute "normalDirection",
109             // which we normalize anyways
110
111             float4 encodedNormal = tex2D(_BumpMap, _BumpMap_ST.xy *
    (input.tex.xy + texCoordOffsets) +
    _BumpMap_ST.zw);
112             float3 localCoords = float3(2.0 * encodedNormal.a - 1.0, 2.0 *
    encodedNormal.g - 1.0, 0.0);
113             localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
114             // approximation without sqrt:  localCoords.z =
115             // 1.0 - 0.5 * dot(localCoords, localCoords);
116
117             float3x3 local2WorldTranspose = float3x3(input.tangentWorld,
    input.binormalWorld, input.normalWorld);
118             float3 normalDirection = normalize(mul(localCoords,
    local2WorldTranspose));
119
120             float3 lightDirection;
121             float attenuation;
122
123             if (0.0 == _WorldSpaceLightPos0.w) // directional light?
124             {
125                attenuation = 1.0; // no attenuation
126                lightDirection = normalize(_WorldSpaceLightPos0.xyz);
127             }
128             else // point or spot light
129             {
130                float3 vertexToLightSource = _WorldSpaceLightPos0.xyz -
    input.posWorld.xyz;
131                float distance = length(vertexToLightSource);
132                attenuation = 1.0 / distance; // linear attenuation
133                lightDirection = normalize(vertexToLightSource);
```

```
            }

            float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb *
_Color.rgb;

            float3 diffuseReflection = attenuation * _LightColor0.rgb *
_Color.rgb * max(0.0, dot(normalDirection, lightDirection));

            float3 specularReflection;
            if (dot(normalDirection, lightDirection) < 0.0) // light source on
the wrong side?
            {
               specularReflection = float3(0.0, 0.0, 0.0);
               // no specular reflection
            }
            else // light source on the right side
            {
               specularReflection = attenuation * _LightColor0.rgb *
_SpecColor.rgb * pow(max(0.0, dot(reflect(-lightDirection, normalDirection),
input.viewDirWorld)), _Shininess);
            }
            return float4(ambientLighting + diffuseReflection +
specularReflection, 1.0);
         }
         ENDCG
      }

      Pass
      {
         Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
         Blend One One // additive blending

         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 sampler2D _ParallaxMap;
         uniform float4 _ParallaxMap_ST;
         uniform float _Parallax;
         uniform float _MaxTexCoordOffset;
         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;
            float3 viewDirWorld : TEXCOORD5;
            float3 viewDirInScaledSurfaceCoords : TEXCOORD6;
         };

         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


            float3 binormal = cross(input.normal, input.tangent.xyz) *
input.tangent.w;
            // appropriately scaled tangent and binormal
```

```
            // to map distances from object space to texture space

            float3 viewDirInObjectCoords = mul(modelMatrixInverse,
float4(_WorldSpaceCameraPos, 1.0)).xyz - input.vertex.xyz;
            float3x3 localSurface2ScaledObjectT = float3x3(input.tangent.xyz,
binormal, input.normal);
            // vectors are orthogonal
            output.viewDirInScaledSurfaceCoords =
mul(localSurface2ScaledObjectT, viewDirInObjectCoords);
            // we multiply with the transpose to multiply with
            // the "inverse" (apart from the scaling)

            output.posWorld = mul(modelMatrix, input.vertex);
            output.viewDirWorld = normalize(_WorldSpaceCameraPos -
output.posWorld.xyz);
            output.tex = input.texcoord;
            output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
            return output;
         }

         float4 frag(vertexOutput input) : COLOR
         {
            // parallax mapping: compute height and
            // find offset in texture coordinates
            // for the intersection of the view ray
            // with the surface at this height

            float height = _Parallax * (-0.5 + tex2D(_ParallaxMap,
_ParallaxMap_ST.xy * input.tex.xy + _ParallaxMap_ST.zw).x);

            float2 texCoordOffsets = clamp(height *
input.viewDirInScaledSurfaceCoords.xy / input.viewDirInScaledSurfaceCoords.z,
-_MaxTexCoordOffset, +_MaxTexCoordOffset);

            // normal mapping: lookup and decode normal from bump map

            // in principle we have to normalize tangentWorld,
            // binormalWorld, and normalWorld again; however, the
            // potential problems are small since we use this
            // matrix only to compute "normalDirection",
            // which we normalize anyways

            float4 encodedNormal = tex2D(_BumpMap, _BumpMap_ST.xy *
(input.tex.xy + texCoordOffsets) +
_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);

            float3x3 local2WorldTranspose = float3x3(input.tangentWorld,
input.binormalWorld, input.normalWorld);
            float3 normalDirection = normalize(mul(localCoords,
local2WorldTranspose));

            float3 lightDirection;
            float attenuation;

            if (0.0 == _WorldSpaceLightPos0.w) // directional light?
            {
               attenuation = 1.0; // no attenuation
               lightDirection = normalize(_WorldSpaceLightPos0.xyz);
            }
            else // point or spot light
            {
               float3 vertexToLightSource = _WorldSpaceLightPos0.xyz -
input.posWorld.xyz;
               float distance = length(vertexToLightSource);
               attenuation = 1.0 / distance; // linear attenuation
               lightDirection = normalize(vertexToLightSource);
            }

            float3 diffuseReflection = attenuation * _LightColor0.rgb *
_Color.rgb * max(0.0, dot(normalDirection, lightDirection));

            float3 specularReflection;
            if (dot(normalDirection, lightDirection) < 0.0) // light source on
the wrong side?
            {
               specularReflection = float3(0.0, 0.0, 0.0);
               // no specular reflection
            }
            else // light source on the right side
            {
               specularReflection = attenuation * _LightColor0.rgb *
_SpecColor.rgb * pow(max(0.0, dot(reflect(-lightDirection, normalDirection),
input.viewDirWorld)), _Shininess);
            }
            return float4(diffuseReflection + specularReflection, 1.0);
         }
         ENDCG
      }
```

```
293         }
294     }
```

我们在场景中添加两个球体，一个应用法线贴图，一个应用视差贴图，观察效果如图：



恭喜你！在这个章节中你应该了解：

> 1、如果通过改进法线贴图实现视差贴图。
>
> 2、视差贴图的如何用数学描述。
>
> 3、视差贴图的实现。

资源下载地址：点击下载，共下载 30 次。

前一篇：Unity3D 模仿《魔兽世界》的第三人称角色控制器

后一篇：第二十二章节：Cookies（关于投影纹理贴图塑造光的形状）

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