第16. Crysis中植被的程序化动画和着色

在CryTek，Crysis（孤岛危机）其中一个主要的目标是为计算机游戏图形定义一个新的标准。在我们最新的游戏中，我们开发了一些新的技术，它们都包含在CryEngine2中。在Crysis这些技术中一个关键的特性是植被的渲染，它由几个部分组成，包括程序中断和物理交互、着色、程序动画，还有远距离画面生成，等等。

游戏中的植被通常主要是静止的，通过一些简单的弯曲来营造一种风的错觉。我们的游戏可以有上千个不同的植被，但是我们任然通过使得植被对全局和局部的风源做出交互来进一步推进植被数量的界限，并且我们不仅弯曲植被，还会对叶子进行弯曲，具体的，所有的程序化，高效的计算都在GPU上完成。

在这一张中，我们阐述了我们是怎样使用有效并且逼真的方法来处理着色和程序化植被动画。

16.1 程序化动画

在我们的方法中，我们把动画分成两部分：(1) 主弯曲，它会沿着风的方向驱动整个植被运动；（2）弯曲细节，它会驱动叶子运动。在世界坐标系统中，通过对每个影响实例的风力求和，计算每个实例的风的向量。风的区域可以是定向的风源，也可以是全向的风源。在我们的例子中，我们用一个非常类似于单个点受影响的光源的计算方法来计算这个和，并考虑方向和衰减。同样的，每个实例都有它自己的刚度，当实例停止受任何风源影响时风的强度会随着时间逐渐衰减。图16-1展示了一个例子。

图16-1 增加主弯曲强度的可视化

设计者可以把风源防止到一个特殊位置，把它们依附在一个实体上（比如，直升机），也可以依附在粒子特效系统上。通过这种方法，我们可以理论上在保持每个顶点的消耗不便的情况下提供大量的风源，尽管每个实例的CPU消耗有一些额外的线性增加。

我们通过使用风向量的 xy 分量来生成主弯曲，这会提供我们风向和风的强度，使用植被的网格作为方向上顶点变形的强度。注意，必须注意限制变形量；否则变形的结果看起来不可信。

对于弯曲叶子的细节，我们以相似的方式处理，但是在这个例子中只考虑风的强度。美术使用通用的3D模型软件对每个顶点画上一个RGB颜色，这个颜色提供给我们关于细节弯曲的额外信息。如果图16-2所示，红色通道用于叶子边缘的刚度，绿色通道用于每片叶子的相位变换，蓝色通道用于整片叶子的刚度。透明通道用于预先计算吸收的环境光。这个着色器的更多详细内容，请看16.2章节。

图16-2 使用顶点颜色

16.1.1 实现细节

我们主要的目标之一就是实现一个直观的系统，让美术或者策划（designer）用起来尽可能的简单。因此，策划的主要输入输入是风向和风速。如果在特殊的案例中有需求的话，美术仍然可以覆盖 叶子的风速、边缘和每片叶子的振幅的默认设置，生成一个更加赏心悦目的植被动画。

近似正弦波

传统方式的程序化顶点动画实现依赖于使用正弦波。我们可以利用三角形波以低消耗的方式近似的结果。具体地说，我们使用了总共四个顶点三角形波来弯曲细节：两个用于叶子的边缘，两个用于叶子的弯曲。在计算这些波之后，如程序16-1所演示的，我们使用三次差值来平滑这些波。图16-3说明这个过程。

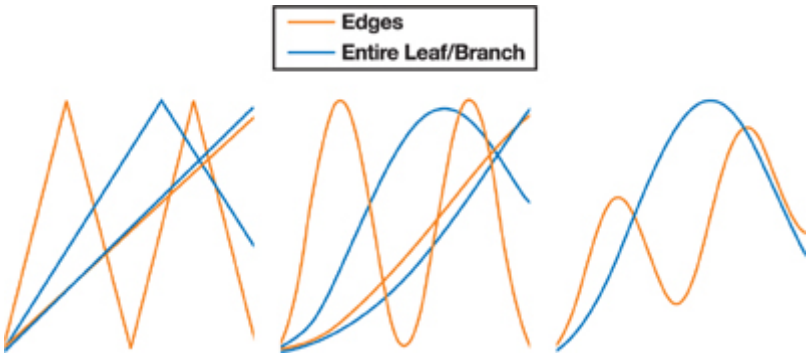

图16-3 合成波 （经过三次平滑，两个橙色波表示用于叶子边缘，两个蓝色波用于整个叶子的弯曲。第一张图是原始波，第二张图是进过平滑之后的波，第三张图是合成之后的波）

16-1. 用来生成波的函数

```
float4 SmoothCurve( float4 x )
{
  return x * x * (3.0 - 2.0 * x);
}

float4 TriangleWave( float4 x )
{
    return abs(frac( x + 0.5 ) * 2.0 - 1.0);
}

float4 SmoothTriangleWave( float4 x )
{
    return SmoothCurve(TriangleWave( x ));
}
```

弯曲详情

叶子的弯曲，正如我们提到的，通过变形叶子的边缘实现，使用顶点色的红色通道来控制边缘的刚度。这种变形是沿着世界空间的顶点法线xy方向处理的。

终于，我们来到每片叶子的弯曲，每片叶子的弯曲可以简单的通过沿着z轴从上往下变形得到，叶子的刚度则使用蓝色通道形成。

顶点颜色的绿色通道包含了每片叶子的相位变化，我们使用蓝色通道提供属于每片叶子的自己相位，以便每片叶子的运动都不相同。程序16-2展示了我们细节弯曲方法的代码。

叶子的形状和顶点数量可以根据植被的类型而改变。比如，我们为棕榈树的整个叶子建模，这样给与我们高度可控的方式去对它制作很好的动画。然而，对于比较大的树，我们只能把叶子建模成若干个面；我们接受不可能完全控制这种叶子的情况，因为有些叶子被放置在作为纹理的相对低面数（low-polygon-count）的平面上（Plane）。不过，我们使用同样方法在所有不同的案列中得到了比较好的结果。

主弯曲

我们通过沿着风向来替换顶点的xy位置完成了植被的主弯曲，使用标准化的高度来缩放替换大顶端位置。仅仅替换顶点是不够的，因为我们需要约束顶点的运动，来最小化变形中导致的失真。我们通过计算顶点到网格中心点的距离并使用这个距离重新调节替换的标准化坐标来到达这点。

这个处理结果形成了一个球形的有限运动，它足够用于我们的植被案例中，因为它是一个单一的植被对象。弯曲变形的量也需要小心的调整：太多的弯曲会破坏这种错觉（通过改变顶点形成叶子弯曲的这种错觉）。程序16-3展示了我们的实现。图16-4展示了一些使用了我们主弯曲技术的案列。

图16-4 不同类型的植被弯曲

程序16-2. 我们详细弯曲方法的实现

```
//Phases (object, vertex, branch)
float fObjPhase = dot(worldPos.xyz, 1);
fBranchPhase += fObjPhase;
float fVtxPhase = dot(xPos.xyz, fDetailPhase + fBranchPhase);

//x is used for edges; y is used for branches
float2 xWavesIn = fTime + float2(fVtxPhase, fBranchPhase);

//1.975, 0.793, 0.375, 0.193 are good frequencies
float4 vWaves = (frac(vWavesIn.xxyy * float4(1.975, 0.793, 0.375, 0.193)) * 2.0 - 1.0) * fSpeed * fDetailFreq;
vWaves = SmoothTriangleWave(vWaves);
float2 vWavesSum = vWaves.xz + vWaves.yw;

//Edge(xy) an branch bending(z)
vPos.xyz += vWavesSum.xxy * float3(fEdgeAtten * fDetailAmp * vNormal.xy, fBranchAtten * fBranchAmp);
```

程序 16-3. 主弯曲的实现

```
//Bend factor - Wind variation is done on the GPU.
float fBF = vPos.z * fBendScale;

//Smooth bending factor an increase its nearby height limit
fBF += 1.0;
fBF * = fBF;
fBF = fBF * fBF - fBF;

//Displace position
float3 vNewPos = vPos;
vNewPos.xy += vWind.xy * fBF;

//Rescale
vPos.xyz + normalize(vNewPos.xyz) * fLength;
```

16.2 植被着色

在Crysis中我们有上千种不同的植被对象，经常覆盖整个屏幕。所以我们需要记住 质量/效率 比。

由于这个原因，我们将逐像素着色和顶点着色结合起来，在草地的案例中，它完全使用了这种方式。仅有的不同是由于草地的高填充率的需求我们所有的着色都是逐顶点的。

树干着色只用的标准的Lambert和Phong着色模型。

叶子通过不同的方式来完成。我们把它渲染成双面的，并且叶子使用了透明度测试（alpha test）而草地使用了透明度混合（alpha blending）。在我们的项目最开始时我们尝试了不同的方法，比例用两个Pass做透明度测试或者透明度混合，但是对于我们项目，质量/效率比不能承受这样做。

现实中，叶子可以有不同的厚度（对于我们的叶子，这是我们经常做的假设）并且让不同数量的光通过不同区域。英雌，我们需要一个近似的次表面的散射。

我们使用美术制作的次表面纹理贴图来近似这个效果，这个展示在图16-5中，它是使用常规的图片编辑软件制作而成的。这个贴图可以有叶子的内部深度细节，比如脉络和分支，并且它可以是相对比分辨率的（比如，128x128），这取决于细节程度的要求。16.2.4章节中的程序16-4提供了详细实现。

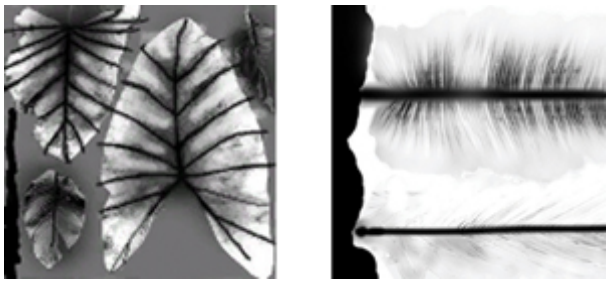

图16-5 次表面贴图纹理的例子

因为植被的性能和质量是至关重要的，基于此，我们决定使用低消耗和对美术友好的次表面散射近似的计算每个顶点，简单的使用 **-N · L**乘以可见光的位置，**E · L**（视角方向和光向量做点积）。两者都乘以次表面贴图的厚度变量。图16-6展示了一些例子。

图16-6 直观的近似的次表面散射

值得提出来的是，在CryENGINE2中的所有着色器是在世界坐标中处理的，这是我们独有的解决方案，有助于在基于属性实例的案例在硬件极限情况下运行，并且在一些特殊的案例中最小化着色的不连续性。

16.2.1 环境光

传统的恒定的环境颜色显得非常单调，并且变得过时了。在开发的初期，我们有两个不同的植被着色器。一个是我们本章所介绍的这个实现方案；另外一个更加复杂，在间接光照中使用了球面谐波，他是主要用于大树木。为了减少着色器排列数量并且由于球面谐波的复杂性，我们遗憾的放弃了后面那种方法，转而使用统一的并且低消耗的户外环境光的解决方案。

环境关在我们的引擎中现在有三个变量：户外，室内和低配硬件的简单解决方案。我们处理户外和室内环境光的方法相当复杂，这需要超过两章篇幅来解释；因此，这不在本章内容范围之内。

全部的三个变量尝试给一个不发光的表面提供一个有趣的表现，针对本章节，我们将假设低配硬件的解决方案是通过使用半球光照照射到一个平坦的阴影区域来实现的（读过LearingOpenGL的应该知道，其实就是一个半球型光打在一个黑暗中的木地板上的效果）。

最后我们也会使用预计算环境光遮挡数据存储在顶点的透明通道中，它是通过美术或者计算机使用标准的3D模型绘制而成。

16.2.2 边缘平滑

使用透明度测试的一个大的问题是硬边。在我们开发植被的主要着色器的那时，任何硬件中都不支持alpha-to-coverage（现在支持了）（alpha-to-coverage是一种以AlphaTest为基础的技术，对边缘进行拼花的一个技术）。所以我们提出了一个特殊的解决方案，通过后期处理（在像素着色器之后）来平滑边缘。

在CrgyENGINE 2中我们使用了延迟渲染（deferred rendering）的方式，首先渲染一个z-pass并将其深度值写入到一个浮点型纹理中。

这个技术对制作那些普遍需要深度信息的效果变得可行。边缘平滑就是其中一个效果；它使用深度纹理进行边缘检测，使用旋转三角形对依赖的纹理进行采样，并使用双线性过滤（bilinear filtering）。由于透明几何体不会写入深度值到目标缓冲区，边缘平滑只能对不透明几何体生效。图16-7展示了边缘平滑带来的好处。

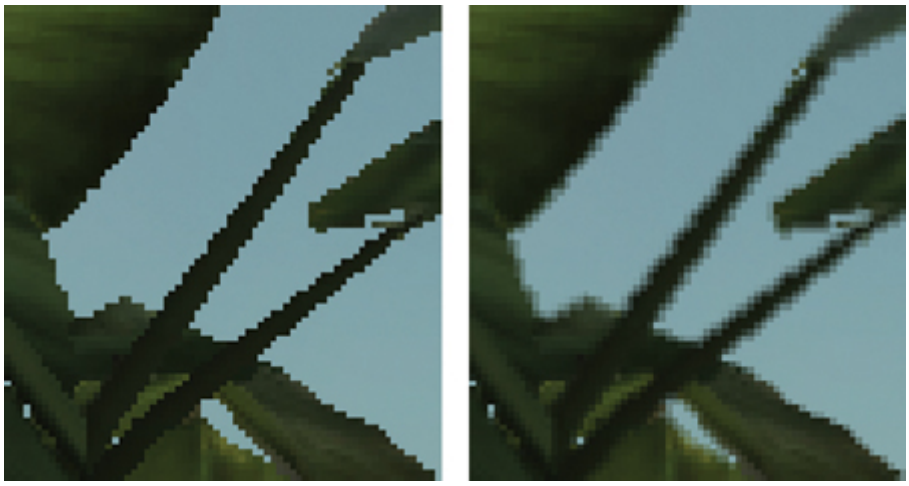

图17-7 边缘平滑的好处

16.2.3 整合

在结合高质量的阴影和指数色调映射器，各种后期处理方法，比如Bloom和太阳光束，以及一些其他的技术我们得到了最终的结果，得益于结合所有这些技术，我们最终得到了我们期望的画质，如图16-8所展示的。

图16-8 最终的结果

16.2.4 实现细节

在展示着色器实现之前，我们想指出一点，为CryENGINE 2开发的统一着色器解决方法不仅简化了着色器，而且使得它们可读性更强易于维护。还为它增加了强制命名约束和最小化代码冗余的好处。

目的是尽可能简单的共享内核但是减少用户与着色器内核的交互。结果是，所有的灯光控制，环境光的计算，一些其他重要的计算（如视差遮挡映射（Parallax Occlusion Mapping）和贴花）不在暴露给着色器的编写者。

用户可以访问四种自定义功能，它们允许用户通过自定义的数据来初始化，来计算每个灯光，环境光和最终的着色。其他的一些操作都像是一个黑盒，通过统一的数据结构它为四个功能的每个功能提供给使用者重要的数据。这个数据结构包含了重要的可共享的数据，如视角向量，法线向量，漫反射贴图颜色，高度图颜色，透明值等等。

程序16-4和16-5展示了最终的着色器实现。程序16-5展示的函数是用户自定义功能，这些函数里我们处理每个光源的着色计算，应用灯光的诸如灯光漫反射和高光颜色的属性。在漫反射函数里我们处理了近似半球面光照。在程序的结尾，如漫反射贴图，麻烦颜色和高光颜色的材质属性最终被合成应用。

程序16-4. 叶子的正反面着色函数

```
half3 LeafShadingBack( half3 vEye,
                       half3 vLight,
                       half3 vNormal,
                       half3 cDiffBackK,
                       half fBackViewDep )
{
  half fEdotL = saturate(dot(vEye.xyz, -vLight.xyz));
  half fPowEdotL = fEdotL * fEdotL;
  fPowEdotL *= fPowEdotL;
  // Back diffuse shading, wrapped slightly
   half fLdotNBack = saturate(dot(-vNormal.xyz, vLight.xyz)*0.6+0.4);
  // Allow artists to tweak view dependency.
   half3 vBackShading = lerp(fPowEdotL, fLdotNBack, fBackViewDep);
  // Apply material back diffuse color.
   return vBackShading * cDiffBackK.xyz;
}

void LeafShadingFront(half3 vEye,
                      half3 vLight,
                      half3 vNormal,
                      half3 cDifK,
                      half4 cSpecK,
                      out half3 outDif,
                      out half3 outSpec)
{
  half fLdotN = saturate(dot(vNormal.xyz, vLight.xyz));
  outDif = fLdotN * cDifK.xyz;
  outSpec = Phong(vEye, vLight, cSpecK.w) * cSpecK.xyz;
}
```

程序 16-5. 用于植被的像素着色器代码

```
void frag_custom_per_light( inout fragPass pPass,
                            inout fragLightPass pLight ) {
  half3 cDiffuse = 0, cSpecular = 0;
  LeafShadingFront( pPass.vReflVec, pLight.vLight, pPass.vNormal.xyz,
                    pLight.cDiffuse.xyz, pLight.cSpecular,
                    cDiffuse, cSpecular );
  // Shadows * light falloff * light projected texture
   half3 cK = pLight.fOcclShadow * pLight.fFallOff * pLight.cFilter;
  // Accumulate results.
  pPass.cDiffuseAcc += cDiffuse * cK;
  pPass.cSpecularAcc += cSpecular * cK;
  pPass.pCustom.fOcclShadowAcc += pLight.fOcclShadow;
}
void frag_custom_ambient(inout fragPass pPass, inout half3 cAmbient)
{
```

```
  // Hemisphere lighting approximation
  cAmbient.xyz = lerp(cAmbient*0.5f, cAmbient,
                      saturate(pPass.vNormal.z*0.5f+0.5f));
  pPass.cAmbientAcc.xyz = cAmbient;
}
void frag_custom_end(inout fragPass pPass, inout half3 cFinal) {
  if( pPass.nlightCount && pPass.pCustom.bLeaves ) {
    // Normalize shadow accumulation.
   half fOccFactor = pPass.pCustom.fOcclShadowAcc/pPass.nlightCount;
    // Apply subsurface map.
    pPass.pCustom.cShadingBack.xyz *= pPass.pCustom.cBackDiffuseMap;
    // Apply shadows and light projected texture.
    pPass.pCustom.cShadingBack.xyz *= fOccFactor *
                                      pPass.pCustom.cFilterColor;
}
// Apply diffuse texture and material diffuse color to
   // ambient/diffuse/sss terms.
cFinal.xyz = (pPass.cAmbientAcc.xyz + pPass.cDiffuseAcc.xyz +
              pPass.pCustom.cShadingBack.xyz) *
              pPass.cDiffuseMap.xyz * MatDifColor.xyz;
// Apply gloss map and material specular color, add to result.
cFinal.xyz += pPass.cSpecularAcc.xyz * pPass.cGlossMap.xyz *
              MatSpecColor.xyz;
// Apply prebaked ambient occlusion term.
cFinal.xyz *= pPass.pCustom.fAmbientOcclusion;
}
```

16.3 总结

在这一章，我们介绍了在Crysis中植被的着色和程序化动画是怎样实现的。以CryENGINE 2如此之快的速度发展，可能在Crysis完成之前这些技术已经加入到引擎当中了。

这里介绍的程序化动画技术用通用的方式去实现的，所以对那些不是植被对象应用风力也是有可能的，比如服饰和头发；这些对象与植被仅有的不同是没有主弯曲。

我们可以用极其有效的方式用直升机，手榴弹，甚至是武器射击来影响植被，服饰和头发。

如同我们推动植被渲染的改进一样，随着硬件快速的发展越来越快，植被渲染也一直存在改进的空间。举个例子，着色器可以改进，叶子的高光在大多数情况下是各自不同的，我们也可以用更多的计算次每个像素的近似表面散射。最终，使用更多的波来弯曲，可以扩大动画变化的范围。