本章紧跟Ken Perlin在GPU Gems中的一章，“实施改进的Perlin噪声”（Perlin 2004）。 肯（Ken）的章节讨论了如何使用3D纹理对程序噪声进行快速逼近，而在这里，我们以Microsoft Direct 3D Effects（FX）和CgFX语法描述了与参考CPU实现完全匹配的改进噪声算法的有效GPU实现。

26.1 平滑随机

噪声是生成纹理中添加自然外观的重要组成部分。在现实世界中，没有什么是完全统一的，并且噪波提供了一种可控制的方式来将这种随机性添加到着色器中。

噪声函数具有几个重要特征：

1. 它为每个输入位置产生一个可重复的伪随机值。
2. 它具有已知范围（通常为[-1，1]）。
3. 它具有频带受限的空间频率（即平滑）。
4. 它没有显示明显的重复模式。
5. 它的空间频率在平移下不变。

Perlin改进的噪声算法可以满足所有这些要求（Perlin 2002）。现在我们继续说明如何在GPU上实现它。

26.2 存储VS计算

程序噪声通常是在当今的着色器中使用预先计算的3D纹理实现的。与这种方法相比，直接在像素着色器中实现噪声有几个优点：

它需要较少的纹理内存。

周期很大（也就是说，模式不会经常重复）。

结果与现有的CPU实现完全匹配。

它允许产生4维噪点，这对于动画3D效果很有用。（当前的硬件不支持4D纹理。）

与硬件纹理过滤相比，所使用的插值质量更高，从而产生了看起来更平滑的噪声。

图26-1a显示了使用3D纹理渲染具有噪声的对象的结果。注意由于线性插值被用于纹理贴图过滤的伪影。相反，图26-1b中的图像显示了使用本章中描述的过程方法的结果，实际上更好。

这种方法的明显缺点是计算量更大。这主要是因为我们必须在着色器中执行插值，而没有利用GPU中存在的纹理过滤硬件。使用最新的编译器，优化的3D噪声实现可编译为大约50个Pixel Shader 2.0指令；幸运的是，这对于当前的高端GPU而言并不构成挑战。

26.3 实现细节

Perlin噪声算法包括两个主要阶段。

第一阶段为3D空间中的每个整数位置生成一个可重复的伪随机值。这可以通过多种方式实现，但是Perlin的算法使用哈希函数。哈希函数基于排列表，该排列表以随机顺序包含从0到255的整数。（此表可以在实现之间标准化，以便它们产生相同的结果。）首先，根据位置的x坐标对表进行索引。然后，将y坐标添加到表中此位置的值，然后使用结果再次在表中查找。在对z坐标重复此过程之后，结果是每个可能的位置都是伪随机整数。

在算法的第二阶段，该伪随机整数用于索引3D梯度向量表。在“改进”算法中，仅使用了八个不同的梯度。标量值是通过获取梯度和噪声空间内的小数位置之间的点积来计算的。通过在空间中相邻的八个点中的每个点的噪声值之间进行插值来获得最终值。

Perlin改进的噪声算法的CPU实现将排列和梯度表存储在数组中。像素着色器当前不支持索引到恒定内存，因此我们将这些表存储在纹理中，并使用纹理查找来访问它们。纹理寻址设置为环绕（或重复）模式，因此我们不必担心扩展表，以避免像在CPU实现中那样在数组末尾进行索引。清单26-1显示了如何在FX文件中初始化这些纹理。

虚拟机是Microsoft Direct 3D效果和CgFX运行时最有用的功能之一，可用于从函数中以过程方式生成纹理。 我们使用此功能从参考实现中提供的相同数据构建置换和渐变纹理。 如果您使用的是另一种着色语言，例如OpenGL着色语言，那么在您的应用程序中包含此代码应该很简单。

**Example 26-1. Source Code for Initializing the Permutation Texture for Noise**

**// permutation table**

**static int** permutation[] = { 151, **/\* 254 values elided . . . \*/**, 180 }; **// Generate permutation and gradient textures using CPU runtime**

**texture** permTexture <**string** texturetype = "2D";**string** format = "l8"; **string** function = "GeneratePermTexture"; **int** width = 256, height = 1;>;

**float4** GeneratePermTexture(**float** p : **POSITION**) : **COLOR** {

**return** permutation[p \* 256] / 255.0;

}

**sampler** permSampler = sampler\_state {

texture = <permTexture>; AddressU = Wrap; AddressV = Clamp; MAGFILTER = POINT; MINFILTER = POINT; MIPFILTER = NONE; };

该参考实现使用位处理代码直接从哈希值生成梯度向量。由于当前的像素着色器硬件不包括整数运算，因此该方法不可行，因此我们预先计算了包含16个梯度矢量的小的一维纹理。清单26-2显示了生成该纹理的代码。

清单26-3给出了使用这两个纹理来计算噪声值的最终代码。它包括Perlin的新插值函数，它是5级的Hermite多项式，并产生C2连续噪声函数。或者，您也可以使用原始插值函数，该函数的计算成本较低，但会导致不连续的二阶导数。

**Example 26-2. Source Code to Compute Gradient Texture for Noise**

**// gradients for 3D noise**

**static float3** g[] = {1,1,0, -1,1,0, 1,-1,0, -1,-1,0, 1,0,1, -1,0,1, 1,0,-1, -1,0,-1, 0,1,1, 0,-1,1, 0,1,-1, 0,-1,-1, 1,1,0, 0,-1,1, -1,1,0, 0,-1,-1, };

**texture** gradTexture<**string** texturetype = "2D"; **string** format = "q8w8v8u8"; **string** function = "GenerateGradTexture"; **int** width = 16, height = 1; >; **float3** GenerateGradTexture(**float** p : **POSITION**) : **COLOR** {

**return** g[p \* 16];

}

**sampler** gradSampler = sampler\_state { texture = <gradTexture>; AddressU = Wrap; AddressV = Clamp; MAGFILTER = POINT; MINFILTER = POINT; MIPFILTER = NONE; };

**Example 26-3. Source Code for Computing 3D Perlin Noise Function**

**float3** fade(**float3** t) { **return** t \* t \* t \* (t \* (t \* 6 - 15) + 10); **// new curve** **// return t \* t \* (3 - 2 \* t); // old curve** } **float** perm(**float** x) { **return tex1D**(permSampler, x / 256.0) \* 256; } **float** grad(**float** x, **float3** p) { **return dot**(**tex1D**(gradSampler, x), p); } **// 3D version** **float** inoise(**float3** p) { **float3** P = **fmod**(**floor**(p), 256.0); p -= **floor**(p); **float3** f = fade(p); **// HASH COORDINATES FOR 6 OF THE 8 CUBE CORNERS** **float** A = perm(P.x) + P.y; **float** AA = perm(A) + P.z; **float** AB = perm(A + 1) + P.z; **float** B = perm(P.x + 1) + P.y; **float** BA = perm(B) + P.z; **float** BB = perm(B + 1) + P.z; **// AND ADD BLENDED RESULTS FROM 8 CORNERS OF CUBE** **return lerp**( **lerp**(**lerp**(grad(perm(AA), p), grad(perm(BA), p + **float3**(-1, 0, 0)), f.x), **lerp**(grad(perm(AB), p + **float3**(0, -1, 0)), grad(perm(BB), p + **float3**(-1, -1, 0)), f.x), f.y), **lerp**(**lerp**(grad(perm(AA + 1), p + **float3**(0, 0, -1)), grad(perm(BA + 1), p + **float3**(-1, 0, -1)), f.x), **lerp**(grad(perm(AB + 1), p + **float3**(0, -1, -1)), grad(perm(BB + 1), p + **float3**(-1, -1, -1)), f.x), f.y), f.z); }

经过修改，也可以为顶点着色器编译相同的代码。顶点着色器支持将变量索引添加到常量内存中，因此没有必要使用纹理查找。

26.3.1 优化

清单26-3中的代码是参考CPU实现的直接端口。有几种方法可以优化此代码，以更好地利用图形硬件。（优化的代码包括在随附的CD上，以及实现4D噪声的方法。）

参考实现使用对置换表的六个递归查找来生成初始的四个哈希值。在GPU上实现此功能的明显方法是将6个纹理查找到1D纹理中，但相反，我们可以预先计算一个256x256像素RGBA 2D纹理，该纹理在每个texel中包含四个值，并使用单个2D查找。

我们还可以通过扩展渐变表并置换渐变来将最终查询删除到置换表中。因此，我们不使用16像素的渐变纹理，而是创建一个256像素的纹理，并根据排列表重新排列了渐变。这样就消除了八次1D纹理查找。

未优化的实现可编译为81个Pixel Shader 2.0指令，包括22个纹理查找。经过优化后，它是53条指令，其中只有9条是纹理查找。由于许多代码是标量的，因此也有可能更多地利用向量运算。

26.4 总结

我们已经描述了像素着色器的过程噪声的实现。程序噪声是丰富视觉效果的重要构建块，可用于凹凸贴图和其他效果，如图26-2所示。尽管运行在当今GPU上的大多数实时应用程序的开发人员都不想将50像素着色器指令专用于单个噪声查找，但该技术对于高质量，离线渲染应用程序非常有用，在这些应用程序中，匹配现有CPU噪声实现非常重要。随着GPU的计算能力增加，并且由于持续的硬件趋势，存储器访问变得相对昂贵，诸如此类的过程技术将变得越来越有吸引力。