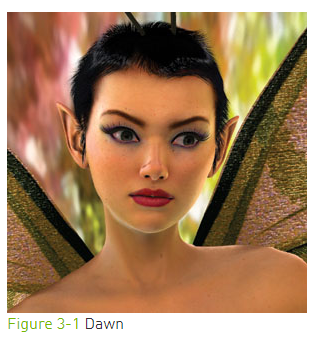
# 第三章 “黎明”Demo中的皮肤

## 介绍

“黎明”是由 NVIDIA 创建的Demo，用于介绍 GeForce FX 产品线并说明如何使用可编程着色来创建逼真的人物角色，如图 3-1 所示。 顶点着色器（在第 4 章“‘黎明’演示中的动画”中进行了描述）通过索引蒙皮和混合形状使高分辨率网格变形，它们为片段着色器中使用的照明模型提供设置。 在引入可编程图形硬件之前，皮肤和片段着色器产生的范围和细节是无法实现的。



本章讨论了在Demo中如何使用可编程图形硬件来计算光照和加速《黎明》角色上的皮肤着色器。

## 皮肤着色

由于各种原因，皮肤是一个难以在计算机图形中模拟的表面。 即使是电影中使用的高端制作图形也无法模拟出经得起仔细检查的逼真人物角色。 人类通过非语言线索获得大量信息：体重的变化、特定的行走方式、面部表情，甚至是某人皮肤的红晕。 研究人员表明，婴儿在出生时就能识别并注视面部表情。 因此，我们人类在检测角色移动和表情方式以及皮肤微妙颜色的异常时非常敏锐。 尽管很少有人能理解诸如次表面散射和边缘照明之类的术语，但几乎任何人都可以告诉你渲染何时出错。

除了着色问题，人们可能很快就会说，由于皮肤移动的方式，皮肤着色器看起来像塑料，因此解决角色设置问题很重要。 第 4 章描述了在《黎明》Demo中使用累积索引蒙皮驱动身体以及通过混合目标为 Dawn 的面部添加情感的技术。 这些技术为 Dawn 和她需要执行的动作提供了一个可以接受的近似值，但它们可能无法完全描述皮肤在真实骨骼上滑动、弯曲和下垂以用于其他目的的方式。 创建角色时，为观察者实际看到的部分当地进行骨骼绑定和着色计算非常重要。

## 场景中的光照

一旦皮肤以尽可能逼真的方式移动，您就可以专注于着色部分。 分解表面着色部分的最直接方法是检查场景的照明环境，然后查看皮肤对光线的反应。

### 高动态范围（HDR）环境

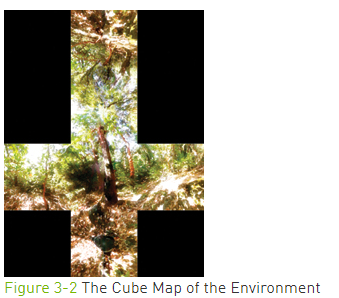
我们创建角色并将她放置在一个与传统实时图形技术典型的平面、简单的阴影不同的环境中。 较早的图形架构是围绕 Gouraud 和 Phong 着色的概念构建的，它们将所有入射光描述为从无限小的点、定向和聚光灯辐射。 材料表面的漫反射和镜面反射响应通常使用的纹理和方程无法描述合适范围的光响应。

前一种方法的一个不良结果是场景往往缺乏现实世界中看到的亮度范围。 地板上的一张白纸会像上面的白色光源一样明亮； 复杂环境中的角色通常看起来格格不入，因为它们是由离散点而不是周围环境照亮的。

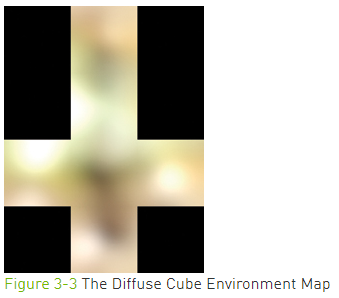
Paul Debevec 在高动态范围图像处理和处理领域进行了大量研究，以解决这个问题。 他的作品可以在各种 SIGGRAPH 出版物中找到，并通过他渲染的电影中的例子变得更加具体。 我们试图利用新的可编程实时图形架构将这些概念应用于实时角色。 特别是，我们希望Dawn被环境照亮，并允许亮点导致光线流失，软化她的轮廓边缘。

Dawn是一个仙女，我们找到了一个合适的森林地带，拍了一些经过校准的照片。我们使用了一个iPIX套件，包括一个以相反角度锁定的三脚架、一台数码相机和一个183度的鱼眼镜头。在这种情况下，两张半球形的照片是使用相机的内置宏在几个快门速度下拍摄的。与在任何单独的快门速度下产生的单像素颜色相比，这组图像更好地提供了关于每个像素处光强度的信息。

我们使用iPIX软件将相应的半球缝合成一个全景图，然后使用Debevec的HDRShop软件创建一个高动态范围的全景图，该全景图在每个给定角度对环境的灯光颜色和强度进行编码。见图3-2。然后，HDRShop可以创建漫反射查找贴图，该贴图执行卷积，为具有给定表面粗糙度的曲面创建漫反射或镜面反射贴图（在过去，该曲面将使用Phong指数建模）。



漫反射贴图是一个立方体贴图，如图3-3所示，使用网格的曲面法线进行索引，并存储该方向入射光的余弦加权平均值。暂时忽略半球部分可能在给定点被遮挡的事实，该加权平均值为计算曲面上点的漫反射提供了适当的值。



镜面反射贴图也是一种立方体贴图，如图3-4所示，并使用反射向量进行索引（立方体贴图“常用”于反射）。该镜面反射贴图基于粗糙度因子进行模糊，以模拟曲面上任意给定点处不断变化的曲面法线。



### 遮挡项

使用环境贴图来描述环境或照明解决方案的一个问题是，映射的样本仅对空间中的单个点非常准确。这样做的第一个副作用是，如果仙女四处走动，背景似乎“无限”遥远。这对我们来说是一个合适的近似值，因为仙女很小，在这种情况下背景应该看起来很远。

第二个副作用是漫反射贴图和镜面反射贴图描述了来自拍摄环境的入射光，但它们没有描述由模拟元素（例如Dawn自身）引入的阴影。因此，有必要开发一种技术来整合这些信息。

许多实时阴影技术会产生硬阴影，我们想要一个特别柔和的外观。因此，我们生成了一个遮挡项，该项近似于每个顶点上方的半球被场景中其他对象遮挡的百分比。

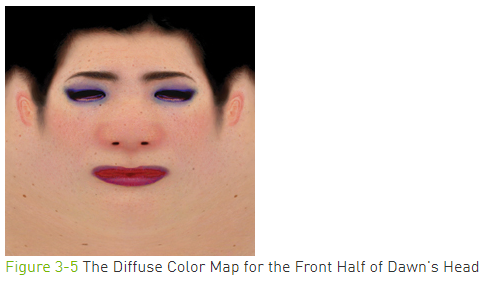
这是使用自定义软件工具完成的，该工具将光线随机投射到该顶点的可见半球并发现与其他几何体的碰撞。我们在“Dawn”Demo和“Ogre”Demo（内容由 Spellcraft Studio 提供）中使用了这种技术，尽管结果的使用方式有所不同。对于 Dawn，遮挡项对于她的身材的顶点保持不变；她的脸混合了来自各种变形目标的遮挡项。

我们在预发布的 GeForce FX 5800 上达到了所能做的极限（Dawn 被创建为该产品的发布Demo），我们不再害怕为每一帧上的每个顶点设置遮挡项的动画，就像我们为 Ogre 所做的那样——Spellcraft Studio 的《Yeah! The Movie》Demo中的角色。

## 皮肤如何对光做出反应

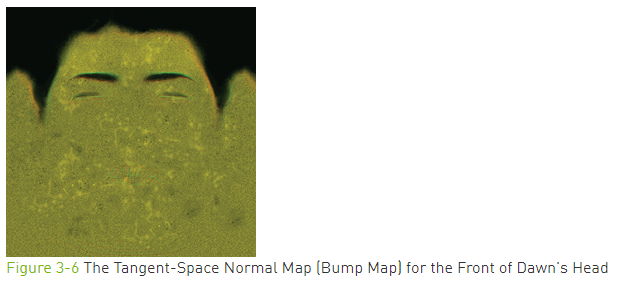
皮肤与我们在计算机图形中建模的大多数表面不同，因为它由半透明的表皮、真皮和皮下组织层组成。 次表面散射（Subsurface scattering）是我们对光进入表面、通过介质散射并在另一点离开的现象的术语。 当您在太阳前举起手时，这种效果通常可以看作是穿过皮肤的橙色光。 这种在皮肤下的散射对于皮肤在各个角度的外观都很重要，并有助于赋予皮肤柔软、高度独特的特性。 不幸的是，这一现实违背了图形 API 和架构中的一个常见假设：即物体上某一点的光不会影响其他点的反射。

过去，一些研究小组曾尝试使用多层纹理贴图模拟皮肤的复杂性。一般来说，这种方法已被证明是难以管理的，而且纹理艺术家在对最终混合颜色进行预视觉化时也很难使用。相反，我们使用了单色贴图，通过着色添加颜色变化，如图3-5所示。



此外，皮肤有极其细微的变化，影响其反射特性。这些都会对皮肤的外观产生微妙的影响，尤其是当光线与相机位置（即边缘和边缘照明）直接相反时。真正的皮肤有细小的特征，如绒毛和毛孔，可以捕捉光线。这些细节太小，我们无法明确建模，但我们希望曲面仍能给我们一个合适的整体外观。添加凹凸贴图在特写时提供了一些额外的细节，特别是对于小皱纹，但我们希望外观柔和，而不是有光泽的斑点塑料，并且我们希望效果在屏幕上可见，而不管大小（凹凸贴图通常仅在近距离观看时可见）。

我们认识到可以根据曲面法线和照明或观察向量将它们建模为简单的公式，从而对这两个着色属性进行了近似。特别是，沿着轮廓边缘，我们对Dawn后的照明进行了采样，通过将“穿过”Dawn的光线与她的基本肤色混合在一起的视图向量进行索引，以创建亚表面和边缘效果照明的错觉，尤其是对于背景图中非常明亮的区域。见图3-6。



## 应用

在顶点和片段着色器之间分割照明过程。这是一种一次性照明解决方案：创建蒙皮曲面不需要额外的渲染过程或Alpha blending。

### 顶点着色器

顶点着色器的主要功能是将坐标转换为投影空间，并执行那些在片段着色器中计算起来非常昂贵的数学运算。如第1.3.2节所述，《Dawn》演示中的顶点着色器首先应用变形目标（如果有的话），然后使用98个骨骼的骨架为超过180000个顶点的网格蒙皮。详见第4章。

对于每个顶点，我们从CPU应用程序接收例子3-1所示的数据。

在顶点着色器中计算并在片段着色器中作为插值传递的因子包括世界空间视角方向向量（worldEyeDirection），描述从观察者的眼睛到任何给定顶点的方向；3x3切线转世界空间矩阵（tangentToWorld）；各种术语统称为血液转换（bloodTransmission）。例子3-2显示了输出顶点的数据结构。

**例3-1，从CPU应用程序接收的逐顶点数据：**

// Here is the PER-VERTEX data -- we use 16 vectors,

// the maximum permitted by our graphics API

struct a2vConnector

{

    float4 coord;                // 3D location

    float4 normal;

    float4 tangent;

    float3 coordMorph0;          // 3D offset to target 0

    float4 normalMorph0;         // matching offset

    float3 coordMorph1;          // 3D offset to target 1

    float4 normalMorph1;         // matching offset

    float3 coordMorph2;          // 3D offset to target 2

    float4 normalMorph2;         // matching offset

    float3 coordMorph3;          // 3D offset to target 3

    float4 normalMorph3;         // matching offset

    float3 coordMorph4;          // 3D offset to target 4

    float4 normalMorph4;         // matching offset

    float4 boneWeight0\_3;        // skull and neck bone

    float4 boneIndex0\_3;         // indices and weights

    float4 skinColor\_frontSpec;  // UV indices

};

**例3-2，输出顶点的数据结构：**

// Here is the data passed from the vertex shader

// to the fragment shader

struct v2fConnector

{

    float4 HPOS              : POSITION;

    float4 SkinUVST          : TEXCOORD0;

    float3 WorldEyeDir       : TEXCOORD2;

    float4 SkinSilhouetteVec : TEXCOORD3;

    float3 WorldTanMatrixX   : TEXCOORD5;

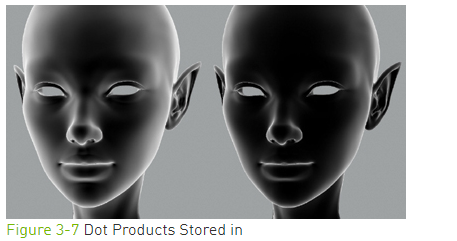
    float3 WorldTanMatrixY   : TEXCOORD6;

    float3 WorldTanMatrixZ   : TEXCOORD7;

};

因为我们是凹凸贴图，我们的片段着色器必须找到世界空间的凹凸法线，所以我们必须提供一种凹凸法线从切线空间（由纹理贴图提供）转换到世界空间的方法。 常用方法是让顶点着色器传递世界空间法线、副法线和切线，然后将这三个向量作为一个 3x3 矩阵将向量从世界空间旋转到切线空间进行计算。 在这种情况下，片段着色器将不得不使用世界空间向量来查看光照解决方案，因此我们映射该矩阵的转置（转置是旋转矩阵的逆矩阵），从而在顶点中产生9 个 MOV指令着色器加载 WorldTanMatrixX、WorldTanMatrixY 和 WorldTanMatrixZ 项。

最后，顶点着色的bloodTransmission或“皮肤轮廓”项是一个float4向量，由遮挡项组成； 表达式（N·V）的不同变化（即表面法线和视角向量的点积）； 以及法线相对于立方体贴图照明的坐标系的旋转。 请参见图 3-7。



OUT.SkinSilhouetteVec = float4(objectNormal.w,

                        oneMinusVdotN \* oneMinusVdotN,

                        oneMinusVdotN,

                        vecMul(G\_DappleXf, worldNormal.xyz).z);

这些步骤提供了一些参数化像素与轮廓的接近程度的方法，它们为我们提供了一个玩具盒，在开发片段着色器时可以使用这些值。数学上精明的读者可能会注意到，（N·V）项可能无法通过Gouraud（线性）着色在大三角形上正确插值。幸运的是，在这种特殊情况下，Dawn被精细地镶嵌在一起，相机需要非常靠近她的脸，才能发现任何错误。这是一个使用着色器的好例子，该着色器非常特定于特定模型，可以在一系列可预测的屏幕大小中看到。（N·V）本可以在片段着色器中完成，但这会使着色器的成本显著提高（因此速度也会降低）。

请注意，《Dawn》Demo中的所有顶点着色器都是按程序生成的。我们在Maya中指定片段着色器，我们有一个顶点着色器生成器，可以查看角色设置（骨骼、变形目标等）和片段着色器请求的输入；然后使用a\*搜索从规则文件生成最佳顶点着色器。[7]

清单3-3是一个示例注释顶点着色器，用于Dawn的面部区域（包括矩阵蒙皮和形状混合，以及片段着色器中用于颜色计算的值）。

// Helper function:

// vecMul(matrix, float3) multiplies like a vector

// instead of like a point (no translate)

float3 vecMul(const float4x4 matrix, const float3 vec)

{

    return(float3(dot(vec, matrix.\_11\_12\_13),

            dot(vec, matrix.\_21\_22\_23),

            dot(vec, matrix.\_31\_32\_33)));

}

// The Vertex Shader for Dawn's Face

v2fConnector faceVertexShader(a2vConnector IN,

                        const uniform float MorphWeight0,

                        const uniform float MorphWeight1,

                        const uniform float MorphWeight2,

                        const uniform float MorphWeight3,

                        const uniform float MorphWeight4,

                        const uniform float4x4 BoneXf[8],

                        const uniform float4   GlobalCamPos,

                        const uniform float4x4 ViewXf,

                        const uniform float4x4 G\_DappleXf,

                        const uniform float4x4 ProjXf)

{

    v2fConnector OUT;

    // The following large block is entirely

    // concerned with shape skinning.

    // First, do shape blending between the five

    // blend shapes ("morph targets")

    float4 objectCoord = IN.coord;

    objectCoord.xyz += (MorphWeight0 \* IN.coordMorph0);

    objectCoord.xyz += (MorphWeight1 \* IN.coordMorph1);

    objectCoord.xyz += (MorphWeight2 \* IN.coordMorph2);

    objectCoord.xyz += (MorphWeight3 \* IN.coordMorph3);

    objectCoord.xyz += (MorphWeight4 \* IN.coordMorph4);

    // Now transform the entire head by the neck bone

    float4 worldCoord = IN.boneWeight0\_3.x \*

            mul(BoneXf[IN.boneIndex0\_3.x], objectCoord);

    worldCoord += (IN.boneWeight0\_3.y \*

            mul(BoneXf[IN.boneIndex0\_3.y], objectCoord));

    worldCoord += (IN.boneWeight0\_3.z \*

            mul(BoneXf[IN.boneIndex0\_3.z], objectCoord));

    worldCoord += (IN.boneWeight0\_3.w \*

            mul(BoneXf[IN.boneIndex0\_3.w], objectCoord));

    // Repeat the previous skinning ops

    // on the surface normal

    float4 objectNormal = IN.normal;

    objectNormal += (MorphWeight0 \* IN.normalMorph0);

    objectNormal += (MorphWeight1 \* IN.normalMorph1);

    objectNormal += (MorphWeight2 \* IN.normalMorph2);

    objectNormal += (MorphWeight3 \* IN.normalMorph3);

    objectNormal += (MorphWeight4 \* IN.normalMorph4);

    objectNormal.xyz = normalize(objectNormal.xyz);

    float3 worldNormal = IN.boneWeight0\_3.x \* vecMul(BoneXf[IN.boneIndex0\_3.x],objectNormal.xyz));

    worldNormal += (IN.boneWeight0\_3.y \* vecMul(BoneXf[IN.boneIndex0\_3.y],objectNormal.xyz));

    worldNormal += (IN.boneWeight0\_3.z \* vecMul(BoneXf[IN.boneIndex0\_3.z],objectNormal.xyz));

    worldNormal += (IN.boneWeight0\_3.w \* vecMul(BoneXf[IN.boneIndex0\_3.w],objectNormal.xyz));

    worldNormal = normalize(worldNormal);

    // Repeat the previous skinning ops

    // on the orthonormalized surface tangent vector

    float4 objectTangent = IN.tangent;

    objectTangent.xyz = normalize(objectTangent.xyz - dot(objectTangent.xyz, objectNormal.xyz) \*objectNormal.xyz);

    float4 worldTangent;

    worldTangent.xyz = IN.boneWeight0\_3.x \* vecMul(BoneXf[IN.boneIndex0\_3.x], objectTangent.xyz);

    worldTangent.xyz += (IN.boneWeight0\_3.y \* vecMul(BoneXf[IN.boneIndex0\_3.y], objectTangent.xyz));

    worldTangent.xyz += (IN.boneWeight0\_3.z \* vecMul(BoneXf[IN.boneIndex0\_3.z], objectTangent.xyz));

    worldTangent.xyz += (IN.boneWeight0\_3.w \* vecMul(BoneXf[IN.boneIndex0\_3.w], objectTangent.xyz));

    worldTangent.xyz = normalize(worldTangent.xyz);

    worldTangent.w = objectTangent.w;

    // Now our deformations are done.

    // Create a binormal vector as the cross product

    // of the normal and tangent vectors

    float3 worldBinormal = worldTangent.w \* normalize(cross(worldNormal, worldTangent.xyz));

    // Reorder these values for output as a 3 x 3 matrix

    // for bump mapping in the fragment shader

    OUT.WorldTanMatrixX = float3(worldTangent.x,worldBinormal.x, worldNormal.x);

    OUT.WorldTanMatrixY = float3(worldTangent.y,worldBinormal.y, worldNormal.y);

    OUT.WorldTanMatrixZ = float3(worldTangent.z,worldBinormal.z, worldNormal.z);

    // The vectors are complete. Now use them

    // to calculate some lighting values

    float4 worldEyePos = GlobalCamPos;

    OUT.WorldEyeDir = normalize(worldCoord.xyz - worldEyePos.xyz);

    float4 eyespaceEyePos = {0.0f, 0.0f, 0.0f, 1.0f};

    float4 eyespaceCoord = mul(ViewXf, worldCoord);

    float3 eyespaceEyeVec = normalize(eyespaceEyePos.xyz - eyespaceCoord.xyz);

    float3 eyespaceNormal = vecMul(ViewXf, worldNormal);

    float VdotN = abs(dot(eyespaceEyeVec, eyespaceNormal));

    float oneMinusVdotN = 1.0 - VdotN;

    OUT.SkinUVST = IN.skinColor\_frontSpec;

    OUT.SkinSilhouetteVec = float4(objectNormal.w, oneMinusVdotN \* oneMinusVdotN, oneMinusVdotN,

            vecMul(G\_DappleXf, worldNormal.xyz).z);

    float4 hpos = mul(ProjXf, eyespaceCoord);

    OUT.HPOS = hpos;

    return OUT;

}

### 片段着色器

给定顶点着色器的输出（以及Dawn身体上的所有地方，顶点着色器输出一致的数据结构），我们可以生成实际的纹理颜色。

例子3-4显示了人脸使用的完整片段着色器。

**例3-4，Dawn脸上的片段着色器：**

float4 faceFragmentShader(v2fConnector IN,

            uniform sampler2D SkinColorFrontSpecMap,

            uniform sampler2D SkinNormSideSpecMap,

            // xyz normal map

            uniform sampler2D SpecularColorShiftMap,

            // and spec map in "w"

            uniform samplerCUBE DiffuseCubeMap,

            uniform samplerCUBE SpecularCubeMap,

            uniform samplerCUBE HilightCubeMap) : COLOR

{

    half4 normSideSpec=tex2D(SkinNormSideSpecMap,IN.SkinUVST.xy);

    half3 worldNormal;

    worldNormal.x = dot(normSideSpec.xyz, IN.WorldTanMatrixX);

    worldNormal.y = dot(normSideSpec.xyz, IN.WorldTanMatrixY);

    worldNormal.z = dot(normSideSpec.xyz, IN.WorldTanMatrixZ);

    fixed nDotV = dot(IN.WorldEyeDir, worldNormal);

    half4 skinColor = tex2D(SkinColorFrontSpecMap, IN.SkinUVST.xy);

    fixed3 diffuse = skinColor \* texCUBE(DiffuseCubeMap, worldNormal);

    diffuse = diffuse \* IN.SkinSilhouetteVec.x;

    fixed4 sideSpec = normSideSpec.w \* texCUBE(SpecularCubeMap,worldNormal);

    fixed3 result = diffuse \* IN.SkinSilhouetteVec.y + sideSpec;

    fixed3 hilite = 0.7 \* IN.SkinSilhouetteVec.x \*IN.SkinSilhouetteVec.y \*texCUBE(HilightCubeMap, IN.WorldEyeDir);

    fixed reflVect = IN.WorldEyeDir \* nDotV - (worldNormal \* 2.0x);

    fixed4 reflColor = IN.SkinSilhouetteVec.w \*texCUBE(SpecularCubeMap, reflVect);

    result += (reflColor.xyz \* 0.02);

    fixed hiLightAttenuator = tex2D(SpecularColorShiftMap,IN.SkinUVST.xy).x;

    result += (hilite \* hiLightAttenuator);

    fixed haze = reflColor.w \* hiLightAttenuator;

    return float4(result.xyz, haze);

}

首先，我们得到凹凸贴图表面法线。 存储在 SkinNormSideSpecMap 中的纹理在其 RGB 组件中包含切线空间法线，而高光贴图（表示高光强度的灰度值）在 alpha 通道中（为了代码清晰，我们将在此处将RGB称为 xyz）。 通过使用 WorldTanMatrix 旋转切线空间 xyz 值，我们获得它们在世界坐标中的坐标——这正是我们执行世界空间光照算法所需的。

然后我们将新计算的表面法线与视图方向进行比较。 我们稍后会使用这个 nDotV 值。

half4 normSideSpec tex2D(SkinNormSideSpecMap,IN.SkinUVST.xy);

half3 worldNormal;

worldNormal.x = dot(normSideSpec.xyz, IN.WorldTanMatrixX);

worldNormal.y = dot(normSideSpec.xyz, IN.WorldTanMatrixY);

worldNormal.z = dot(normSideSpec.xyz, IN.WorldTanMatrixZ);

fixed nDotV = dot(IN.WorldEyeDir, worldNormal);

漫反射颜色是皮肤纹理贴图，乘以预卷积的漫反射光照立方体贴图。 我们通过在 SkinSilhouetteVec 中传递的半球遮挡项对此进行了一些调整。

half4 skinColor = tex2D(SkinColorFrontSpecMap, IN.SkinUVST.xy);

fixed3 diffuse = skinColor \* texCUBE(DiffuseCubeMap, worldNormal);

diffuse = diffuse \* IN.SkinSilhouetteVec.x;

边缘高光颜色来自我们的高光立方体贴图，由我们使用法线贴图（即在 SkinNormSideSpecMap 的 alpha 通道中）获得的高光强度贴图调制。 我们开始建立一个累积的结果。

fixed4 sideSpec = normSideSpec.w \* texCUBE(SpecularCubeMap,worldNormal);

fixed3 result = diffuse \* IN.SkinSilhouetteVec.y + sideSpec;

接下来，我们通过索引 WorldEyeDir 来检索 Dawn 背后的环境颜色，并获得传统的反射立方体贴图颜色。 将这些以及一些艺术“软糖因素”添加到我们的结果中。

fixed3 hilite = 0.7 \* IN.SkinSilhouetteVec.x \*IN.SkinSilhouetteVec.y \*texCUBE(HilightCubeMap, IN.WorldEyeDir);

fixed reflVect = IN.WorldEyeDir \* nDotV - (worldNormal \* 2.0x);

fixed4 reflColor = IN.SkinSilhouetteVec.w \*texCUBE(SpecularCubeMap, reflVect);

result += (reflColor.xyz \* 0.02);

fixed hiLightAttenuator = tex2D(SpecularColorShiftMap,IN.SkinUVST.xy).x;

result += (hilite \* hiLightAttenuator);

最后，我们在最终输出的 alpha 通道中添加了一些额外的剪影信息，这样在 alpha 混合时，沿着 Dawn 的剪影边缘的“绽放”看起来更自然。

fixed haze = reflColor.w \* hiLightAttenuator;

return float4(result.xyz, haze);

## 结论

尽管 Dawn 皮肤着色器中的许多实现细节对于许多游戏引擎来说可能过于严格，但大多数概念都可以使用其他方式来实现。高动态范围、次表面散射、边缘照明等的基本原理也可以从合成光源或其他场景信息中计算出来。

在许多方面，很难在Dawn被环境照亮的情况下工作。更复杂和更逼真的照明解决方案通常以牺牲艺术控制为代价。在本例中，我们希望她的鸡皮疙瘩更加明显，但环境过于分散，以至于我们不得不不切实际地夸大她的表面疙瘩以进行补偿。

如果我们第二次实现 Dawn，我们可能会使用更加混合的光照方法，在这种方法中，我们会查看漫反射和镜面反射贴图（给定平滑法线），然后使用主要的“光照方向”来计算凹凸贴图的贡献。这将使我们能够更直接地控制凹凸贴图的外观，并消除在顶点着色器中执行昂贵的矩阵转置的需要。