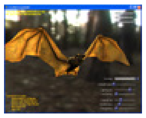
LocalDeformablePRT Sample

See Also

Collapse All

This sample demonstrates a simple usage of locally-deformable precomputed radiance transfer (LDPRT). This implementation does not require an offline simulator for calculating PRT coefficients; instead, the coefficients are calculated from a thickness texture. This allows an artist to create and tweak subsurface scattering PRT data in an intuitive way.



Path

Source	*SDK root*\Samples\C++\Direct3D\LocalDeformablePRT
Executable	*SDK root*\Samples\C++\Direct3D\Bin\x86 or x64\LocalDeformablePRT.exe

Sample Overview

This sample shows what is likely the easiest non-trivial implementation of a PRT technique (it doesn't even use the offline simulator). The animated bat in the scene is textured with a thickness map. The alpha channel of the albedo texture stores this scalar value - the higher the value the more backlight is allowed to transmit through the wings. Each color channel is also influenced by a shader constant that allows selective tinting of transmitted light. Because this is a true PRT effect, the entire lighting environment can be used for illumination.

For simplicity, the per-pixel transfer function is globally set to the standard diffuse, clamped, cosine kernel (for N dot L lighting above the tangent plane and a scaled N dot L lobe mirrored across the tangent plane for subsurface contribution). Far more local control over light transport is possible with the addition of coefficient data per vertex or per texel; however, the simplicity and accessibility of the demonstrated technique make it worth considering even without further improvements.

Technical Overview

PRT enables complex lighting transport effects like soft self-shadowing, colored interreflections, and subsurface scattering (that is, translucency). LDPRT is no exception, and is built upon the same concepts as standard PRT, so it's helpful to have a basic understanding of the math behind PRT before investigating this sample (such as used in PRT Demo Sample, **Precomputed Radiance Transfer (Direct3D 9)**, and **PRT Equations (Direct3D 9)**).

LDPRT differs from standard PRT by employing a single coefficient per spherical harmonic (SH) order; this ensures that the transfer function has radial symmetry around the normal. This radial symmetry allows for very quick rotations and makes LDPRT suitable for use on animated meshes. The decrease in coefficients naturally results in constraints on possible transfer functions, but this is largely mitigated by selecting proper normals for the coefficients (see the reference page **ID3DXPRTEngine::ComputeLDPRTCoeffs** for remarks about shading normals). D3DX can also generate LDPRT coefficients and shading normals from a given set of PRT coefficients, resulting in a surprisingly good LDPRT approximation to the PRT original. See the PRTDemo sample for a comparison of PRT and LDPRT techniques on a variety of scenes.

The LDPRT technique employed within the LocalDeformablePRT sample is actually easier to implement than a standard PRT solution because it doesn't necessitate running a simulator to generate transfer coefficients. The values that control the transmitted light are taken directly from a texture map and are therefore particularly easy to tweak in an intuitive way.

Implementation

This specific implementation of LDPRT requires very minimal additional art requirements; the unused alpha channel in the albedo texture was painted to contain the transmission factor of subsurface light. This can be thought of as the thickness of the textured object, where higher values indicate a thinner material (and therefore more transmitted light). To display the full usefulness of LDPRT, vertex blending (also known as "skinning") data was added to the mesh within the 3D modeling application. This model also uses a normal map to store normal offset values; the perturbed normals add apparent texture to the surface and are used as the shading normals for LDPRT calculations (demonstrating the compatibility of LDPRT with other surface lighting techniques).

Application Initialization

On startup, the sample:

- Loads the skybox cubemap and computes the spherical harmonic projection of the lighting environment.
- Generates four cubemaps to store the first 16 SH basis functions (each color channel stores a unique basis function). Note that this sample only uses SH bases through order 2, but the sample supports through order 3 for simple extension.
- Loads the mesh hierarchy and prepares the matrix palette. (For more information on vertex blending see SkinnedMesh Sample.)
- Loads the effect file and required textures.

Frame Rendering

At the beginning of every frame, the SH lighting environment is calculated from a weighted combination of the light probe and the adjustable spotlight:

```
void UpdateLightingEnvironment()
{
    // Gather lighting options from the HUD
    g_vLightDirection = g_LightControl.GetLightDirection();
    g_fLightIntensity = g_SampleUI.GetSlider( IDC_LIGHT_SLIDER )->GetValue() / 100.0f;
    g_fEnvIntensity = g_SampleUI.GetSlider( IDC_ENV_SLIDER )->GetValue() / 1000.0f;

    // Create the spotlight
    D3DXSHEvalConeLight( D3DXSH_MAXORDER, &g_vLightDirection, D3DX_PI/8.0f,
                         g_fLightIntensity, g_fLightIntensity, g_fLightIntensity,
                         m_fRLC, m_fGLC, m_fBLC);

    float fSkybox[3][D3DXSH_MAXORDER*D3DXSH_MAXORDER];

    // Scale the light probe environment contribution based on input options
    D3DXSHScale( fSkybox[0], D3DXSH_MAXORDER, g_fSkyBoxLightSH[0], g_fEnvIntensity );
    D3DXSHScale( fSkybox[1], D3DXSH_MAXORDER, g_fSkyBoxLightSH[1], g_fEnvIntensity );
    D3DXSHScale( fSkybox[2], D3DXSH_MAXORDER, g_fSkyBoxLightSH[2], g_fEnvIntensity );

    // Combine the environment and the spotlight
    D3DXSHAdd( m_fRLC, D3DXSH_MAXORDER, m_fRLC, fSkybox[0] );
    D3DXSHAdd( m_fGLC, D3DXSH_MAXORDER, m_fGLC, fSkybox[1] );
    D3DXSHAdd( m_fBLC, D3DXSH_MAXORDER, m_fBLC, fSkybox[2] );
}
```

The remaining LDPRT work happens inside the pixel shader:

```
float4 PS( VS_OUTPUT In, uniform bool bNdotL ) : COLOR0
{
    // Albedo
    float4 Color = tex2D( AlbedoSampler, In.TexCoord );

    // Normal map
    float3 Normal = tex2D( NormalMapSampler, In.TexCoord );
    Normal *= 2;
    Normal -= 1;

    // Move the normal from tangent space to world space
    float3x3 mTangentFrame = { In.Binormal, In.Tangent, In.Normal };
    Normal = mul( Normal, mTangentFrame );

    if( bNdotL )
    {
         // Basic N dot L lighting
         Color *= g_fLightIntensity * saturate( dot( Normal, g_vLightDirection ) );
    }
    else
    {
         // 1 channel LDPRT, 3 channel lighting loading Ylm coeffs from cubemaps
         Color *= LDPRTCoeffLighting( In.TexCoord, Normal );
    }

    return Color;
}
```

The shader is first responsible for determining the shading normal for the LDPRT calculations. In this case, the normal stored in the standard normal map is used as the shading normal; because the lighting calculations in this sample are conducted in world space, a tangent frame is constructed within the vertex shader and passed to the pixel shader for transforming the perturbed normal into world space. As shown, the LDPRT illumination is calculated within the LDPRTCoeffLighting helper function:

```
float4 LDPRTCoeffLighting(float2 vTexCoord, float3 vSHCoeffDir)
{
    float4 vExitRadiance[3] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    float4 vTransferVector[4], vYlm[4];
    Ylm(vSHCoeffDir, vYlm);

    // A 4 channel texture can hold a cubics worth of LDPRT coefficients.
    float4 vLDPRTCoeffs = { 1,  2.0/3.0, 0.25, 0};
    BuildSHTransferVector(vLDPRTCoeffs, vYlm, vTransferVector);

    // Calculate sub-surface contribution

    // Negating the odd-order coefficients will mirror the lobe across the tangent plane
    vLDPRTCoeffs.y *= -1;
    float4 vTransferVectorBehind[4];
    BuildSHTransferVector(vLDPRTCoeffs, vYlm, vTransferVectorBehind);

    // The alpha channel of the albedo texture is being used to store how 'thin'
    // the material is, where higher values allow more light to transmit.
    // Although each color channel could have a separate thickness texture, we're
    // simply scaling the amount of transmitted light by a channel scalar.
    float4 vAlbedo = tex2D( AlbedoSampler, vTexCoord );
    for( int i=0; i < 4; i++ )
    {
        vTransferVectorBehind[i] *= vAlbedo.a;
    }

    // Red
    vExitRadiance[0] += g_vLightCoeffsR[0] * (vTransferVector[0] + g_vColorTransmit.r * vTransferVectorBehind[0]);
    vExitRadiance[0] += g_vLightCoeffsR[1] * (vTransferVector[1] + g_vColorTransmit.r * vTransferVectorBehind[1]);
    vExitRadiance[0] += g_vLightCoeffsR[2] * (vTransferVector[2] + g_vColorTransmit.r * vTransferVectorBehind[2]);
    vExitRadiance[0] += g_vLightCoeffsR[3] * (vTransferVector[3] + g_vColorTransmit.r * vTransferVectorBehind[3]);

    // Green
    vExitRadiance[1] += g_vLightCoeffsG[0] * (vTransferVector[0] + g_vColorTransmit.g * vTransferVectorBehind[0]);
    vExitRadiance[1] += g_vLightCoeffsG[1] * (vTransferVector[1] + g_vColorTransmit.g * vTransferVectorBehind[1]);
    vExitRadiance[1] += g_vLightCoeffsG[2] * (vTransferVector[2] + g_vColorTransmit.g * vTransferVectorBehind[2]);
    vExitRadiance[1] += g_vLightCoeffsG[3] * (vTransferVector[3] + g_vColorTransmit.g * vTransferVectorBehind[3]);

    // Blue
    vExitRadiance[2] += g_vLightCoeffsB[0] * (vTransferVector[0] + g_vColorTransmit.b * vTransferVectorBehind[0]);
    vExitRadiance[2] += g_vLightCoeffsB[1] * (vTransferVector[1] + g_vColorTransmit.b * vTransferVectorBehind[1]);
    vExitRadiance[2] += g_vLightCoeffsB[2] * (vTransferVector[2] + g_vColorTransmit.b * vTransferVectorBehind[2]);
    vExitRadiance[2] += g_vLightCoeffsB[3] * (vTransferVector[3] + g_vColorTransmit.b * vTransferVectorBehind[3]);

    return float4( dot(vExitRadiance[0], 1),
                   dot(vExitRadiance[1], 1),
                   dot(vExitRadiance[2], 1),
                   1 );
}
```

This function first generates the required rotated SH basis functions by calling the Ylm helper function, which relies on the SH cubemap textures generated on initialization. The shader then builds the LDPRT coefficients; this sample's shader uses the coefficients { 1, 2.0/3.0, 0.25, 0 } that describe the second-order, clamped cosine kernel. More complex light transport would use coefficients stored in the vertex or a texture map. Next, the BuildSHTransferVector function multiplies the LDPRT coefficients against the basis functions to create the transfer vector vTransferVector; this first vector gathers the direct lighting contribution. By negating the odd-order coefficients, the resulting second transfer vector vTransferVectorBehind is a mirror of the first across the tangent plane, and therefore gathers the subsurface lighting contribution. The subsurface vector is then scaled against the thickness factor at that pixel. Finally, the subsurface transmission is scaled per color channel and both transfer vectors are convolved against the lighting environment. The resulting radiance calculation is output as the pixel color.

See Also

Precomputed Radiance Transfer (Direct3D 9)
PRT Equations (Direct3D 9)