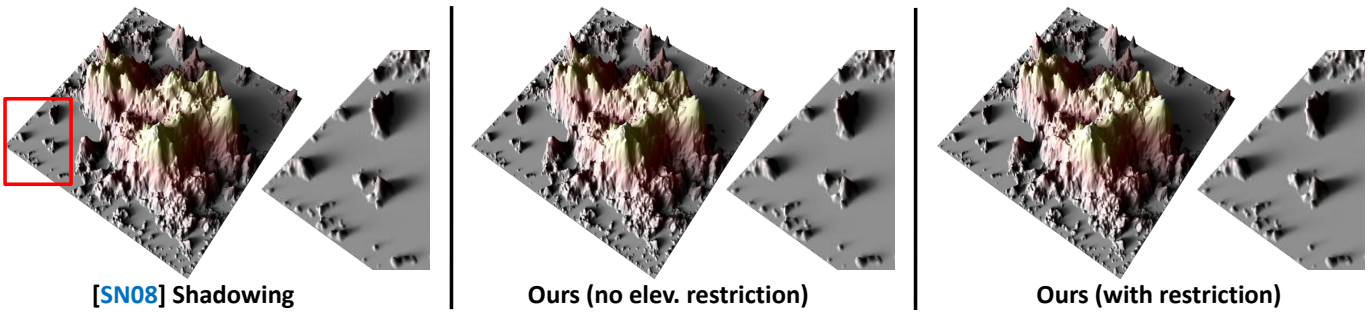
Eurographics Symposium on Rendering (2009)
Hendrik P. A. Lensch and Peter-Pike Sloan (Guest Editors)

Fast Global Illumination on Dynamic Height Fields: Supplemental Material

Derek Nowrouzezahrai
University of Toronto

John Snyder
Microsoft Research



Comparing [SN08] shadows (left) to our shadowing without (middle) and with (right) key light restriction in the elevation direction. All images are generated with a single key light restricted in the azimuthal direction as well.

1. Using NLPs for Visibility

In the case of direct-illumination, we represent visibility at each azimuthal direction in the NLP basis, whereas the approach in [SN08] simply stores the maximum blocking angle. In order to construct the SH visibility per azimuthal wedge, we use the NLP-to-SH blending matrices, whereas the approach in [SN08] queries pre-computed 2D tables of linearly blended SH projections of individual SH wedge functions. We interpolate pre-projected and blurred representations of the maximum blocking angle instead of using interpolated angles to query SH projections.

At a single azimuthal direction, an NLP coefficient vector captures all the variation in elevation of an SH coefficient vector of equal order. Using many azimuthally sampled NLP functions (with some form of interpolation between samples) is a much more powerful representation of spherical functions than SH, however our blending matrices combine interpolation with re-projection into SH, effectively smoothing out the sampled NLP representation. The benefit of going back to SH is the ability to perform double product integral reconstructions more efficiently.

Apart from having an analytic representation (as opposed to using pre-computed tables), another benefit of the azimuthally-sampled NLP representation of visibility is the ability to restrict key light source extents both in the azimuth *as well as* in elevation. In the figure above, we see that without restricting the size of the area light in the elevation (but with azimuthal restriction), the method in [SN08] generates slightly sharper shadows, however elevation restriction (which is possible only with our NLP representation) allows for slightly sharper shadows than those in [SN08].

2. Shading in Texture Space

We compute the visibility and IR vectors at (potentially down-sampled) height field grid samples and the shading is computed at the full-resolution height field grid samples. This can be thought of as shading in *receiver point* space, and not at pixels or vertices. Our performance is more or less insensitive to changes in rendering resolution.

In practice, we tessellate the height field geometry at the same resolution as the height field grid, and so we are effectively computing shading at vertices, however any alternative tessellation scheme may be employed. The tessellated geometry is texture mapped with the shading computed using a GPGPU kernel implemented in a fragment shader.

3. Shader Source Code

We include source code listings for reference purposes. The GPGPU fragment shader computes visibility and IR computation as well as direct and indirect shading. Furthermore, it only implements Diffuse BRDF shading, assumes a single environmental light source and does not use sub-sampling.

References

[SN08] SNYDER J., NOWROUZEZAHRAI D.: Fast soft self-shadowing on dynamic height fields. *Computer Graphics Forum: Proc. Eurographics Symposium on Rendering* (2008).

```
// The B-spline coefficients used for 1D B-spline interpolation
const float4 W_mid_and_knot[2] = { 1.f / 6.f, 2.f / 3.f, 1.f / 6.f, 0,
                                   1.f / 48.f, 23.f / 48.f, 23.f / 48.f, 1.f /48.f };

//------------------------------------ GPGPU PIXEL SHADER ------------------------------------
// Takes in the Volume Texture with the Height and Shading Pyramids,
// and renders the direct and indirect illumination to a bound RenderTexture.
//--------------------------------------------------------------------------------------------
float4 Shade(VertexShaderOutput In, uniform int iNumAzimuthalSamples,
             uniform int iLevelOffset, uniform int iShadingLevelOffset,
             uniform float fMultiResUVStepSizes[NUM_PYRAMID_LEVELS],
             uniform float fMultiResUVStepSizeReciprocals[NUM_PYRAMID_LEVELS],
             uniform float2 fCosSinCurrentPhi0s[NUM_ENV_WEDGES]):SV_Target {

    // Store the final IR RGB vectors, visibility vector, as well some temporary IR vectors.
    float4 IRSHVectorR[4] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    float4 IRSHVectorG[4] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    float4 IRSHVectorB[4] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    float4 Visibility[4] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    float4 tempIR[4] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    float4 tempIRRotated[4] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    float3 receiverNormal = g_HFNormals.Sample( g_samLinearClamp,
                                                float2( In.tex.x, 1.0f - In.tex.y ) ).xyz;

    // Pre-fetch swath-independent data for the heigth-difference calculation...
    float f_x0[NUM_PYRAMID_LEVELS];
    for( int i = 0; i < NUM_PYRAMID_LEVELS; i++ )
        f_x0[i] = LookupHFPyramidLowToHigh( In.tex.x, In.tex.y, i, NUM_PYRAMID_LEVELS );

    //-----------------------------------------------------------------------
    // Iterate over all azimuthal swaths:
    // 1. Generate the visibility/IR vector for each swatch
    // 2. Interpolate, rotate and sum them up
    //-----------------------------------------------------------------------

    // Discrete blocking angles (with padding: 1 at [0]
    // and 2 at [NUM_PYRAMID_LEVELS] for B-Spline support)
    float w_plus[NUM_PYRAMID_LEVELS + 3] =(float[NUM_PYRAMID_LEVELS + 3])0;

    float D[NUM_PYRAMID_LEVELS] = (float[NUM_PYRAMID_LEVELS])0;
    float2 f_delta[NUM_PYRAMID_LEVELS] = (float2[NUM_PYRAMID_LEVELS])0;

    // Legendre IR and visibility coefficient vectors
    float4 IR_LPr[NUM_ENV_WEDGES] = (float4[NUM_ENV_WEDGES])0;
    float4 IR_LPg[NUM_ENV_WEDGES] = (float4[NUM_ENV_WEDGES])0;
    float4 IR_LPb[NUM_ENV_WEDGES] = (float4[NUM_ENV_WEDGES])0;
    float4 Visibility_LP[NUM_ENV_WEDGES] = (float4[NUM_ENV_WEDGES])0;
    float fMaxBlockingAngle;

    for( int iSwathStart = 0; iSwathStart < iNumAzimuthalSamples; iSwathStart++ ) {
        float fBaseBlockingAngle = 0;
        float NdotAzimuthalDirection = dot(receiverNormal.xy, fCosSinCurrentPhi0s[iSwathStart]);
        if( NdotAzimuthalDirection > 0 ) fBaseBlockingAngle = -asin( NdotAzimuthalDirection );

        fMaxBlockingAngle = fBaseBlockingAngle;
        float3 prevColor = { 0, 0, 0 };
        float3 color = { 0, 0, 0 };
        float4 LegendreCoeffs = {0,0,0,0};

        // Travel in +ve \tau direction
        for( int i = NUM_PYRAMID_LEVELS - 1; i >= 0; i-- ) {
```

```
        int iLevel = min( i - 1 + iLevelOffset, NUM_PYRAMID_LEVELS - 1 );
        f_delta[i] = fMultiResUVStepSizes[i] * fCosSinCurrentPhi0s[iSwathStart];
        float f_x1 = LookupHFPyramidLowToHigh( In.tex.x + f_delta[i].x, In.tex.y +
                                               f_delta[i].y, iLevel, NUM_PYRAMID_LEVELS );
        D[i] = ( f_x1 - f_x0[iLevel] ) * fMultiResUVStepSizeReciprocals[i];
        w_plus[i+1] = atan( D[i] );
    }

    w_plus[0] = 0;
    w_plus[NUM_PYRAMID_LEVELS + 1] = w_plus[NUM_PYRAMID_LEVELS];
    w_plus[NUM_PYRAMID_LEVELS + 2] = w_plus[NUM_PYRAMID_LEVELS];

    // Reconstruct blocking angles with B-spline interpolation in positive \tau direction
    for( int m = NUM_PYRAMID_LEVELS - 1; m >= 0; m-- ) {
        for( int j = 0; j < 1; j++ ) { // Sample b-spline at knots and mid-points
            float fBlockingAngle = dot( W_mid_and_knot[j],
                                        float4( w_plus[m], w_plus[m + 1],
                                                w_plus[m + 2], w_plus[m + 3] ) );
            if( fBlockingAngle > fMaxBlockingAngle ) {
                int iShadingLevel = min( m - 1 + iShadingLevelOffset,
                                         NUM_PYRAMID_LEVELS - 1 );
                color = g_ShadingMipPyramid.SampleLevel( HFPyramidSampler,
                                                         float3( (In.tex + f_delta[m]).xy,
                                                                 NUM_PYRAMID_LEVELS - 1 -
                                                                 iShadingLevel ), 0 ).xyz;

                // NLP accumulation: accumulation lags behind by one value
                // (which implicitly handles the base case).
                // Last value added at the end separately. (Equation 12)
                NLPOcclusionCoefficients( fMaxBlockingAngle, LegendreCoeffs );
                IR_LPr[ iSwathStart ] += LegendreCoeffs * ( prevColor.r - color.r);
                IR_LPg[ iSwathStart ] += LegendreCoeffs * ( prevColor.g - color.g);
                IR_LPb[ iSwathStart ] += LegendreCoeffs * ( prevColor.b - color.b);
                prevColor = color;
                fMaxBlockingAngle = fBlockingAngle;
            }
        }
    }

    // NLP accumulation: last value added at the end here (Equation 12)
    NLPOcclusionCoefficients( fMaxBlockingAngle, LegendreCoeffs );
    IR_LPr[ iSwathStart ] += LegendreCoeffs * ( prevColor.r );
    IR_LPg[ iSwathStart ] += LegendreCoeffs * ( prevColor.g );
    IR_LPb[ iSwathStart ] += LegendreCoeffs * ( prevColor.b );

    // Now that we have the maximum blocking angle, we can
    // compute the Legendre visibility vector (Equation 11)
    Visibility_LP[ iSwathStart ] -= LegendreCoeffs;
    NLPOcclusionCoefficients( M_PI / 2.f, LegendreCoeffs );
    Visibility_LP[ iSwathStart ] += LegendreCoeffs;
}

// Take all the LP vectors here and generate the SH vector:
for( int w = 0; w < iNumAzimuthalSamples - 1; w++ ) {
    // Linear LP->SH Blending (using equation from appendix)
    LPtoSH_LinearBlending( IR_LPr[w], IR_LPr[w+1], tempIR );
    SH_rotatez_4( fCosSinCurrentPhi0s[w], tempIR, tempIRRotated );
    IRSHVectorR[0] += tempIRRotated[0]; IRSHVectorR[1] += tempIRRotated[1];
    IRSHVectorR[2] += tempIRRotated[2]; IRSHVectorR[3] += tempIRRotated[3];

    LPtoSH_LinearBlending( IR_LPg[w], IR_LPg[w+1], tempIR );
```

```
        SH_rotatez_4( fCosSinCurrentPhi0s[w], tempIR, tempIRRotated );
        IRSHVectorG[0] += tempIRRotated[0]; IRSHVectorG[1] += tempIRRotated[1];
        IRSHVectorG[2] += tempIRRotated[2]; IRSHVectorG[3] += tempIRRotated[3];

        LPtoSH_LinearBlending( IR_LPb[w], IR_LPb[w+1], tempIR );
        SH_rotatez_4( fCosSinCurrentPhi0s[w], tempIR, tempIRRotated );
        IRSHVectorB[0] += tempIRRotated[0]; IRSHVectorB[1] += tempIRRotated[1];
        IRSHVectorB[2] += tempIRRotated[2]; IRSHVectorB[3] += tempIRRotated[3];

        // Note: re-using the temporary variables to compute the SH Visibility
        LPtoSH_LinearBlending( Visibility_LP[w], Visibility_LP[w+1], tempIR );
        SH_rotatez_4( fCosSinCurrentPhi0s[w], tempIR, tempIRRotated );
        Visibility[0] += tempIRRotated[0]; Visibility[1] += tempIRRotated[1];
        Visibility[2] += tempIRRotated[2]; Visibility[3] += tempIRRotated[3];
    }

    // Direct Illumination Shading: (Visibility * BRDF) dot SH_Lighting
    float4 Color;
    const float4 zhemi_canon = { 1.f / M_PI * 0.886227f, 1.f / M_PI * 1.02333f,
                                 1.f / M_PI * 0.495416f, 1.f / M_PI * 0.f };
    float4 rotated_hemi[4], transfer[4];
    SH_zh_rot_4( receiverNormal, zhemi_canon, rotated_hemi );
    SH_product_4( rotated_hemi, Visibility, transfer );

    Color.r = dot( transfer[0], RLight[0] ) + dot( transfer[1], RLight[1] ) +
              dot( transfer[2], RLight[2] ) + dot( transfer[3], RLight[3] );
    Color.g = dot( transfer[0], GLight[0] ) + dot( transfer[1], GLight[1] ) +
              dot( transfer[2], GLight[2] ) + dot( transfer[3], GLight[3] );
    Color.b = dot( transfer[0], BLight[0] ) + dot( transfer[1], BLight[1] ) +
              dot( transfer[2], BLight[2] ) + dot( transfer[3], BLight[3] );

    // Indirect Illumination Shading: rotate canonical BRDF ZH vector and dot with IR vectors.
    Color.r += dot( rotated_hemi[0], IRSHVectorR[0] ) + dot( rotated_hemi[1], IRSHVectorR[1] ) +
               dot( rotated_hemi[2], IRSHVectorR[2] ) + dot( rotated_hemi[3], IRSHVectorR[3] );
    Color.g += dot( rotated_hemi[0], IRSHVectorG[0] ) + dot( rotated_hemi[1], IRSHVectorG[1] ) +
               dot( rotated_hemi[2], IRSHVectorG[2] ) + dot( rotated_hemi[3], IRSHVectorG[3] );
    Color.b += dot( rotated_hemi[0], IRSHVectorB[0] ) + dot( rotated_hemi[1], IRSHVectorB[1] ) +
               dot( rotated_hemi[2], IRSHVectorB[2] ) + dot( rotated_hemi[3], IRSHVectorB[3] );

    Color.a = 1.0f;
    return Color;
}
```