Eurographics Symposium on Rendering 2017
P. Sander and M. Zwicker
(Guest Editors)

Volume 36 (2017), Number 4

Stochastic Light Culling for VPLs on GGX Microsurfaces (Supplemental Material)

Yusuke Tokuyoshi[1] and Takahiro Harada[2]

[1]Square Enix Co., Ltd.
[2]Advanced Micro Devices, Inc.

1. Maximum Fresnel Factor

Schlick Fresnel factor. The Schlick approximation of the Fresnel factor [Sch94] is defined as

$$F(\cos\theta_h) = F_0 + (1-F_0)(1-\cos\theta_h)^5, \tag{1}$$

where $\cos\theta_h = \boldsymbol{\omega}_i \cdot \boldsymbol{\omega}_h$, and $F_0 \in [0,1]$ is the Fresnel factor at the normal incidence. Since this $F(\cos\theta_h)$ is monotonically increasing for $\theta_h \in \left[0, \frac{\pi}{2}\right]$, the maximum Fresnel factor is yielded from the maximum of θ_h for $\boldsymbol{\omega}_i$ as follows:

$$F_{\max}(\boldsymbol{\omega}_i) = F\left(\cos\left(\theta_h^{\max}(\boldsymbol{\omega}_i)\right)\right), \tag{2}$$

where $\theta_h^{\max}(\boldsymbol{\omega}_i) = \frac{1}{2}\arccos(\boldsymbol{\omega}_i \cdot \mathbf{n}) + \frac{\pi}{4}$ is the maximum of θ_h, and $\cos\left(\theta_h^{\max}(\boldsymbol{\omega}_i)\right) = \sqrt{\frac{1+\sqrt{1-(\boldsymbol{\omega}_i\cdot\mathbf{n})^2}}{2}}$.

Unpolarized Fresnel factor for dielectrics. For dielectrics with unpolarized light, the Fresnel factor is the following equation [CT82]:

$$F(\cos\theta_h) = \frac{F_\perp(\cos\theta_h) + F_\parallel(\cos\theta_h)}{2}. \tag{3}$$

where $F_\perp(\cos\theta_h)$ and $F_\parallel(\cos\theta_h)$ are the perpendicular polarized reflection and parallel polarized reflection defined by

$$F_\perp(\cos\theta_h) = \left|\frac{\cos\theta_h - \sqrt{\eta^2 - \sin^2\theta_h}}{\cos\theta_h + \sqrt{\eta^2 - \sin^2\theta_h}}\right|^2, \tag{4}$$

$$F_\parallel(\cos\theta_h) = \left|\frac{\sqrt{\eta^2 - \sin^2\theta_h} - \eta^2\cos\theta_h}{\sqrt{\eta^2 - \sin^2\theta_h} + \eta^2\cos\theta_h}\right|^2, \tag{5}$$

where $\eta \in (0,\infty)$ is the relative refractive index. For $\theta_h \in \left[0, \frac{\pi}{2}\right]$, if $2-\sqrt{3} \leq \eta \leq 2+\sqrt{3}$, this Fresnel factor $F(\cos\theta_h)$ is monotonically increasing for θ_h. Otherwise, it is quasiconvex. Thus, the maximum Fresnel factor for $\boldsymbol{\omega}_i$ is obtained by

$$F_{\max}(\boldsymbol{\omega}_i) = \max\left(F(1), F\left(\cos\left(\theta_h^{\max}(\boldsymbol{\omega}_i)\right)\right)\right). \tag{6}$$

2. Experimental Results

Interleaved sampling. Fig. 1 shows rendered images with and without interleaved sampling and denoising. Although interleaved sampling and denoising slightly increases the error by blurring small details of illumination, the performance is improved significantly. Therefore, this paper employs interleaved sampling for real-time applications.

Equal-time comparison. Stochastic light culling has a trade-off between the performance and quality, which is controlled by the user-specified parameter δ. Therefore, our ellipsoid-based approach can render higher-quality images than the sphere-based approach when comparing at the same computation time. Fig. 2 shows the equal-time comparison between the bounding sphere and our bounding ellipsoid for the Museum scene. As shown in this experimental result, we are also able to improve the quality using our bounding ellipsoid.

3. HLSL Code

The HLSL code of our bounding ellipsoid generation and tiled culling is shown in Listings 1 and 2.

References

[CT82] COOK R. L., TORRANCE K. E.: A reflectance model for computer graphics. *ACM Trans. Graph. 1*, 1 (1982), 7–24. 1

[Sch94] SCHLICK C.: An inexpensive BRDF model for physically-based rendering. *Comput. Graph. Forum 13*, 3 (1994), 233–246. 1

[Ste15] STEWART J.: Compute-based tiled culling. In *GPU Pro 6: Advanced Rendering Techniques*. A K Peters/CRC Press, 2015, pp. 435–458. 4

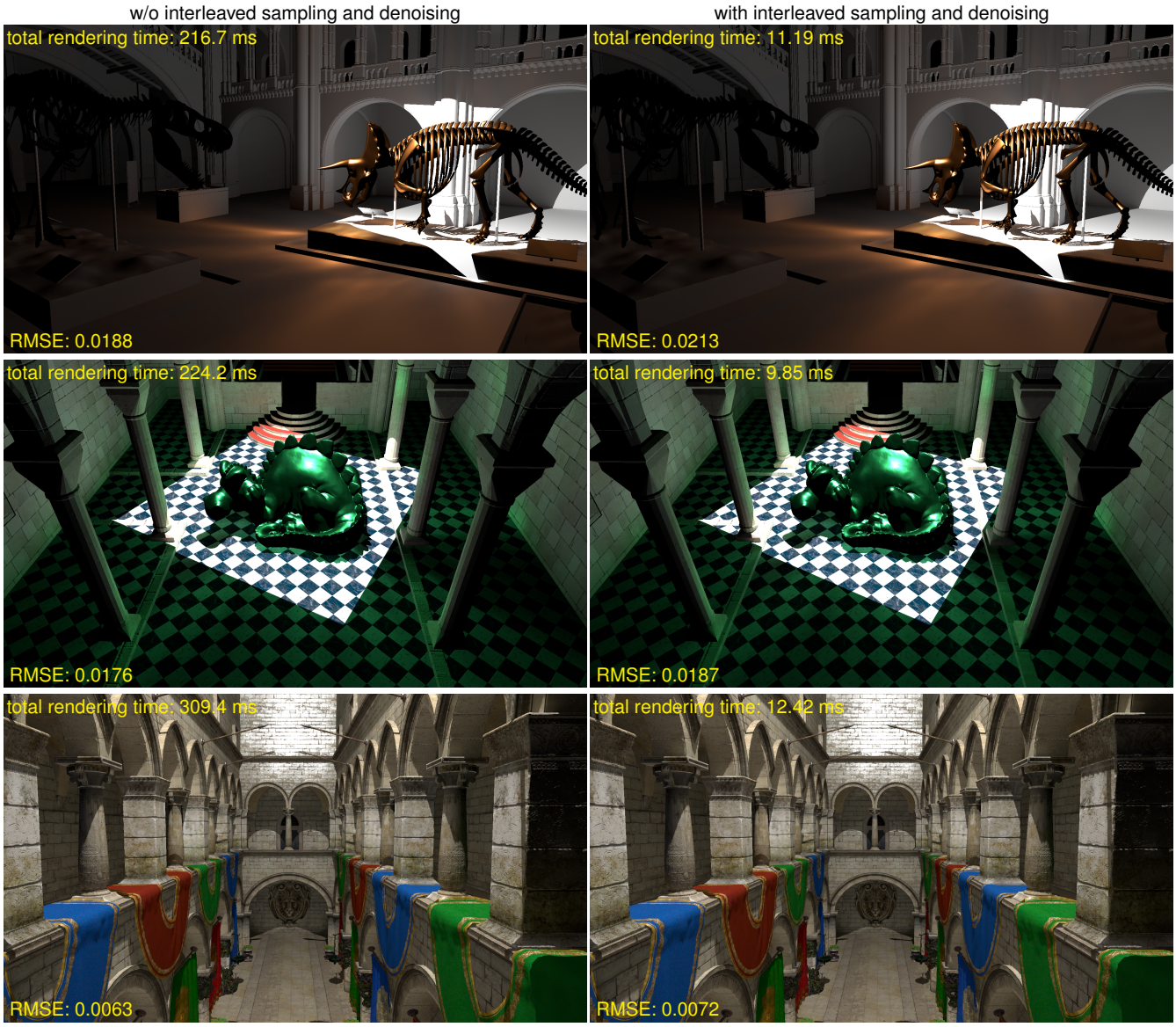
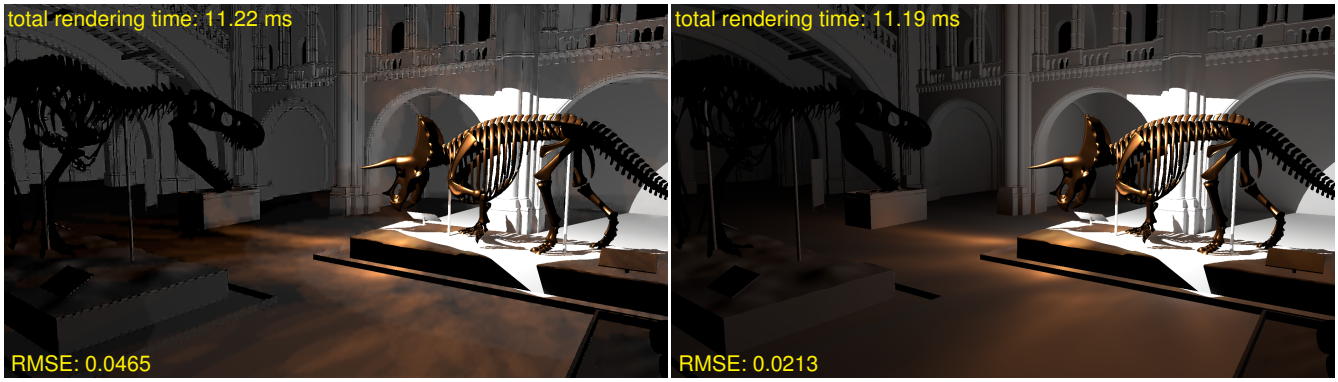


Figure 1: *Rendered images with and without interleaved sampling and denoising. Interleaved sampling and denoising significantly improves the performance with slight error increase.*



Bounding sphere ($\delta = 0.125$) Our bounding ellipsoid ($\delta = 0.001$)

Figure 2: *Equal-time comparison between the bounding sphere and our bounding ellipsoid.*

Listing 1: *Our bounding ellipsoid generation.*

```
struct ELLIPSOID
{
  float3 transformedCenter;
  float3x3 transformation;
};

float3x3 CalculateStretchingMatrix( const float3 eu, const float3 ev, const float3 ew, const float u, const float v, const float w )
{
  return mul( transpose( float3x3( eu * u, ev * v, ew * w ) ), float3x3( eu, ev, ew ) );
}

void GenerateBoundingEllipsoid( const uint lightIndex, const float r, const float3 position,
                                const float3 direction, const float3 normal, const float roughness )
{
  const float squaredRoughness = roughness * roughness;
  const float polarRadius = r * ( 1.0 + squaredRoughness ) / ( 2.0 * roughness );
  const float3 reflectionDirection = reflect( direction, normal );
  const float3 tangent = normalize( cross( direction, normal ) );
  const float3 ellipsoidCenter = position + reflectionDirection * ( r * ( 1.0 - squaredRoughness ) / ( 2.0 * roughness ) );
  const float3 ellipsoidCenterViewSpace = mul( g_cameraViewMatrix, float4( ellipsoidCenter, 1.0 ) ).xyz;
  const float3 ellipsoidU = mul( ( float3x3 )( g_cameraViewMatrix ), reflectionDirection );
  const float3 ellipsoidW = mul( ( float3x3 )( g_cameraViewMatrix ), tangent );
  const float3 ellipsoidV = cross( ellipsoidU, ellipsoidW );
  ELLIPSOID ellipsoid;
  if ( r != 0.0 ) {
    const float3x3 stretch = CalculateStretchingMatrix( ellipsoidU, ellipsoidV, ellipsoidW, 1.0 / polarRadius, 1.0 / r, 1.0 / r );
    const float3 x = mul( stretch, float3( 1.0, 0.0, 0.0 ) );
    const float3 y = mul( stretch, float3( 0.0, 1.0, 0.0 ) );
    const float3 normalizedZ = normalize( cross( x, y ) );
    float3 normalizedX, normalizedY;
    if ( dot( x, x ) > dot( y, y ) ) {
      normalizedX = normalize( x );
      normalizedY = cross( normalizedZ, normalizedX );
    }
    else {
      normalizedY = normalize( y );
      normalizedX = cross( normalizedY, normalizedZ );
    }
    const float3x3 rotation = { normalizedX, normalizedY, normalizedZ };
    const float3x3 transformation = mul( rotation, stretch );
    ellipsoid.transformedCenter = mul( transformation, ellipsoidCenterViewSpace );
    ellipsoid.transformation = transformation;
  }
  else {
    ellipsoid.transformedCenter = FLT_MAX;
    ellipsoid.transformation = 0.0;
  }
  lightBoundingEllipsoidBuffer[ lightIndex ] = ellipsoid;
}
```

Listing 2: *Our tiled culling using bounding ellipsoids. This implementation is based on Modified HalfZ [Ste15].*

```
uint GetSubRegionIndex( const uint2 id )
{
  const uint2 subregionID = id / g_subregionThreadCountXY;
  return subregionID.y * INTERLEAVING_WIDTH + subregionID.x;
}

uint2 GetPixelPosition( const uint2 id )
{
  const uint2 subregionID = id / g_subregionThreadCountXY;
  const uint2 localID = id % g_subregionThreadCountXY;
  return localID * INTERLEAVING_WIDTH + subregionID;
}

float2 GetProjectedPosition( const float2 texcoord, const float4x4 projectionMatrixInv )
{
  const float2 vp = texcoord * 2.0 - 1.0;
  const float3 pos = mul( projectionMatrixInv, float4( vp.x, -vp.y, 0.0, 1.0 ) ).xyz;
  return pos.xy / pos.z;
}

void CalculateFrustumAABB( out float3 aabbCenter, out float3 aabbHalfSize, const float2 rectangleMin, const float2 rectangleMax,
                           const float rectangleZ, const float zmin, const float zmax )
{
  const float2 aabbMinXY = min( rectangleMin * zmin, rectangleMin * zmax );
  const float2 aabbMaxXY = max( rectangleMax * zmin, rectangleMax * zmax );
  const float3 aabbMin = float3( aabbMinXY, rectangleZ * zmin );
  const float3 aabbMax = float3( aabbMaxXY, rectangleZ * zmax );
  aabbCenter = ( aabbMin + aabbMax ) * 0.5;
  aabbHalfSize = ( aabbMax - aabbMin ) * 0.5;
}

float ComputeSquaredDistanceToBox( const float3 position, const float3 boxCenter, const float3 boxHalfSize )
{
  const float3 p = max( abs( boxCenter - position ) - boxHalfSize, 0.0 );
  return dot( p, p );
}

void LightCulling( const uint2 groupID, const uint groupThreadIndex,
                   const float zmin1, const float zmax1, const float zmin2, const float zmax2 )
{
  const uint lightIndexBase = GetSubRegionIndex( groupID * TILE_WIDTH ) * LIGHT_COUNT_PER_SUBREGION;
  const uint2 tileMinPixelPosition = GetPixelPosition( groupID * TILE_WIDTH );
  const float2 tileMin = g_screenResolutionInv * tileMinPixelPosition;
  const float2 tileMax = g_screenResolutionInv * ( tileMinPixelPosition + TILE_WIDTH * INTERLEAVING_WIDTH );
  const float2 baseVertex0 = GetProjectedPosition( tileMin, g_cameraProjectionMatrixInv );
  const float2 baseVertex1 = GetProjectedPosition( tileMax, g_cameraProjectionMatrixInv );
  for ( uint i = groupThreadIndex; i < LIGHT_COUNT_PER_SUBREGION; i += TILE_WIDTH * TILE_WIDTH ) {
    const uint lightIndex = lightIndexBase + i;
    const ELLIPSOID ellipsoid = lightBoundingEllipsoidBuffer[ lightIndex ];
    const float2 vertex0 = mul( ( float2x3 )( ellipsoid.transformation ), float3( baseVertex0, 1.0 ) );
    const float2 vertex1 = mul( ( float2x3 )( ellipsoid.transformation ), float3( baseVertex1.x, baseVertex0.y, 1.0 ) );
    const float2 vertex2 = mul( ( float2x3 )( ellipsoid.transformation ), float3( baseVertex1, 1.0 ) );
    const float2 vertex3 = vertex2 - vertex1 + vertex0;
    const float2 rectangleMin = min( min( min( vertex0, vertex1 ), vertex2 ), vertex3 );
    const float2 rectangleMax = max( max( max( vertex0, vertex1 ), vertex2 ), vertex3 );
    const float rectangleZ = ellipsoid.transformation._m22;
    float3 aabbCenter1, aabbHalfSize1, aabbCenter2, aabbHalfSize2;
    CalculateFrustumAABB( aabbCenter1, aabbHalfSize1, rectangleMin, rectangleMax, rectangleZ, zmin1, zmax1 );
    CalculateFrustumAABB( aabbCenter2, aabbHalfSize2, rectangleMin, rectangleMax, rectangleZ, zmin2, zmax2 );
    const float squaredDistanceToAABB1 = ComputeSquaredDistanceToBox( ellipsoid.transformedCenter, aabbCenter1, aabbHalfSize1 );
    const float squaredDistanceToAABB2 = ComputeSquaredDistanceToBox( ellipsoid.transformedCenter, aabbCenter2, aabbHalfSize2 );
    if ( squaredDistanceToAABB1 < 1.0 ) {
      uint address;
      InterlockedAdd( sharedLightCount1, 1, address );
      sharedLightList[ address ] = lightIndex;
    }
    if ( squaredDistanceToAABB2 < 1.0 ) {
      uint address;
      InterlockedAdd( sharedLightCount2, 1, address );
      sharedLightList[ address ] = lightIndex;
    }
  }
}
```