Point Cloud Culling for Imperfect Shadow Maps

Yusuke Tokuyoshi*
Square Enix Co., Ltd.

1 Introduction

Imperfect shadow maps (ISMs) [Ritschel et al. 2008; Ritschel et al. 2011] are well established for approximating visibilities of many lights (e.g., virtual point lights (VPLs)). However, this shadow mapping is often the main bottleneck for real-time applications. For acceleration of ISMs, a simple culling technique using a compute shader is very effective.

1.1 Imperfect Shadow Maps

ISM is a single-pass technique to render an arbitrary number of shadow maps. This is done by approximating the scene geometry using point cloud (Fig. 1). To make ISMs, these points are rendered instead of polygons by using splatting. For many lights, a point is projected onto only a single shadow map. Therefore, the computation time of this technique is approximately proportional to the total number of the points, and thus independent from the number of lights and primitives. To render these shadow maps in a single pass, low-resolution shadow maps (e.g., 64^2) are tiled in a large single render target as shown in Fig. 2.

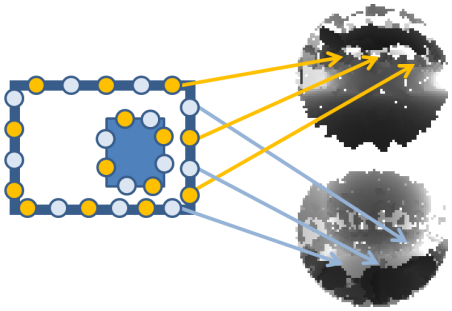

Figure 1: *Point cloud approximation of a scene geometry for ISMs. A point is splatted onto a single shadow map.*

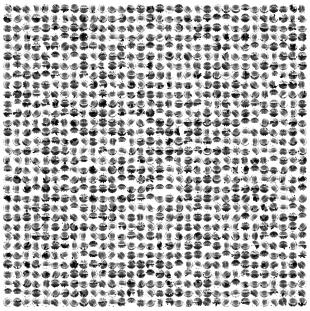

Figure 2: *Render target of ISMs (64^2 resolution $\times$ 1024).*

2 Point Cloud Culling

2.1 Invisible Points

For point splatting, front- or back-face culling can be used, similar to general shadow mapping. Therefore, approximately half of the points can be invisible to lights (Fig. 3). In addition, points under the surface of a VPL are also invisible. These redundant points should be culled before rendering ISMs.

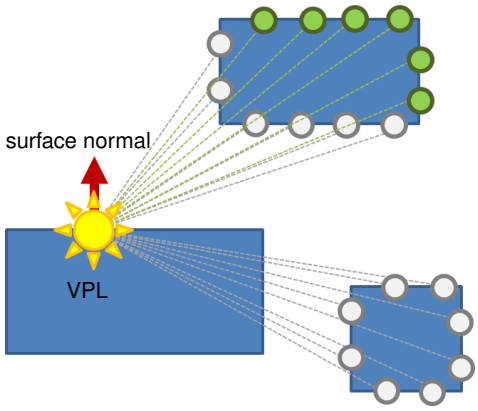

Figure 3: *Visible points (green) and invisible points (gray) for a VPL. Front-face culling is used for ISMs*

*e-mail:tokuyosh@square-enix.com

2.2 Culling via Compute Shader

Program 1 is the compute shader of culling for VPLs. The system value *id* is the point index. This compute pass simply outputs only visible points using *AppendStructuredBuffer*. In order to splat these outputted visible points onto shadow maps, *ID3D11DeviceContext::DrawInstancedIndirect* is used. This API needs a buffer in device memory for input arguments (i.e., the number of visible points). To update this buffer, the hidden counter value of *pointIdBuffer* is copied using *ID3D11DeviceContext::CopyStructureCount*. For splatting using a geometry shader, the counter value is copied to the buffer for arguments directly. However, for vertex shader splatting [Wu 2012], the input argument must be the product of the number of points (i.e., counter value) and number of vertices of the splat. Therefore, a single thread compute pass shown in Program 2 is additionally required for this case.

Program 1: *Culling invisible points for ISMs.*

```
AppendStructuredBuffer< uint > pointIdBuffer : register( u0 );

[ numthreads( WORKGROUP_SIZE, 1, 1 ) ]
void CullIsmPointsCS( const uint id : SV_DispatchThreadID )
{
  const float3 pos = GetPointPosition( id );
  const float3 normal = GetPointNormal( id );
  const uint vplIndex = GetVplIndex( id );
  const float3 vplPos = GetVplPosition( vplIndex );
  const float3 vplNormal = GetVplNormal( vplIndex );

  // front-face culling
  if( dot( vplNormal, normal ) > 0.0 && dot( vplNormal, pos - vplPos ) > 0.0 ) {
    pointIdBuffer.Append( id );
  }
}
```

Program 2: *Update kernel of the input argument of DrawInstancedIndirect for vertex shader splatting.*

```
RWBuffer< uint > bufferForArgs : register( u0 );
Buffer< uint > pointCount : register( t0 );

[ numthreads( 1, 1, 1 ) ]
void CalculateInputArgsCS( )
{
  bufferForArgs[ 0 ] = pointCount[ 0 ] * SPLAT_VERTEX_NUM;
}
```

3 Results and Conclusion

Here, we employ adaptive ISMs (AISMs) [Ritschel et al. 2011] and virtual spherical Gaussian lights (VSGLs, an improved version of VPLs) [Tokuyoshi 2014] as an experimental example. Front-face culling and vertex shader splatting is used for AISMs. Table 1 is the computation time of AISMs for the experimental scene shown in Fig. 4. By culling, the splatting pass, which is the main bottleneck, is more than twice as fast. The overhead of this culling is about 9% of the splatting pass.

Although the complexity of Program 1 is the same as splatting, the overhead of this culling is small in practice. For other applications, splatting and particle rendering can be also accelerated by culling invisible points in a similar fashion.

References

RITSCHEL, T., GROSCH, T., KIM, M. H., SEIDEL, H.-P., DACHSBACHER, C., AND KAUTZ, J. 2008. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph. 27*, 5, 129:1–129:8.

RITSCHEL, T., EISEMANN, E., HA, I., KIM, J. D., AND SEIDEL, H.-P. 2011. Making imperfect shadow maps view-adaptive: High-quality global illumination in large dynamic scenes. *Comput. Graph. Forum 30*, 8, 2258–2269.

TOKUYOSHI, Y. 2014. Virtual spherical gaussian lights for real-time glossy indirect illumination. In *SIGGRAPH ASIA '14 Tech. Briefs*.

WU, O. 2012. Enhancing graphics in unreal engine 3 titles using new code submissions. In *GDC'12*.

Figure 4: *Experimental scene (264k triangles, polygon models are courtesy of R. W. Sumner, J. Popovic, and F. Meinl). The total rendering time is 29 ms (1024 VSGLs, frame buffer resolution: 1920×1088, ISM resolution: 64^2, 8192 points for each ISM, GPU: Radeon™ HD 6990).*

Table 1: *Computation times of AISMs (ms).*

	w/o culling	with culling
Points Generation	4.21	4.21
Culling	N/A	0.67
Splatting	16.73	7.50