Point Cloud Culling
for Imperfect Shadow Maps

Yusuke Tokuyoshi
Square Enix Co., Ltd.
Virtual point lights are many light sources used for approximating indirect illumination. For interactive or real-time applications, a lot of VPL-based methods have been developed. We’ll also present a new technique entitled “Virtual Spherical Gaussian Lights for Real-Time Glossy Indirect Illumination” in the Technical Brief session. For the detail of this method, please come to the technical brief session.

In this course, we’d like to focus on shadow mapping for each VPL. This shadow mapping can be a bottleneck for VPL-based real-time global illumination.
Imperfect shadow maps are often used for VPLs. This is a single-pass method to render an arbitrary number of shadow maps which are tiled in a single render target. This shadow mapping is done by point-based rendering.
To generate imperfect shadow maps, the scene geometry is first approximated by a point cloud. Instead of polygons, we render this point cloud. Similar to particle rendering, the points are rendered using splatting. This is done by generating a small quad or triangle splat centered on each input point, and then this splat is projected onto a single shadow map. This splatting can be done with a single pass. The computation time depends on the total number of points.
In order to generate imperfect shadow maps for many lights, we have to render many points. Generally, 8 to 16 K points should be used for each VPL. For 1 K VPLs, this means a total of 8 to 16 M points. Splatting these many points can be a bottleneck for global illumination rendering.
Therefore, culling invisible points before the rendering is very effective. In this figure, only green points are visible to a VPL. Gray points are invisible.

All points are generated on surfaces, and each point can have a surface normal. Thus, similar to conventional shadow mapping, we can use front- or back-face culling for each point. In addition, points under the surface of a VPL are also invisible. These points should be culled before the splatting. This culling can be implemented using a compute shader using an AppendStructuredBuffer.
This is the compute shader.
As you can see from the code, it is quite simple.
Please note the red lines.
If a point is visible, it is appended into the pointIdBuffer which is an
AppendStructuredBuffer.
By using this compute shader, points are culled in parallel on the GPU.
After the culling, only the visible points stored in the `pointIdBuffer` are rendered. This is done by `DrawInstancedIndirect` for triangle splats.

In order to obtain the vertex count for the input argument of this draw call, `CopyStructureCount` and a compute shader are used. `CopyStructureCount` is used for getting the hidden counter value of the `pointIdBuffer` which is an `AppendStructuredBuffer`. After that, the total vertex count of all the splats are calculated using a single thread compute shader. For triangle splats, this is calculated by multiplying the counter value by three.

Finally, `DrawInstancedIndirect` is called using the input arguments stored in the device memory.
So, let me show you an experimental result of the point cloud culling.
In this experiment, 1 K virtual spherical Gaussian lights and adaptive imperfect shadow maps are employed.
For each shadow map, we use 8 K points.

The computation time of adaptive points generation for this scene is about 4 milliseconds.
The point splatting time without culling is about 17 milliseconds.
On the other hand, the point splatting time with culling is 7.5 milliseconds.
The overhead of the culling is only 0.67 milliseconds.
So, the splatting pass can be twice as fast by using point cloud culling.
To conclude, culling invisible points for imperfect shadow maps is very effective. This culling can be performed on the GPU by using an AppendStructuredBuffer. Then, we render only the visible points for imperfect shadow maps using CopyStructureCount and DrawInstancedIndirect.
References

