A 2.5D Culling for Forward+

Takahiro Harada*
Advanced Micro Devices, Inc.

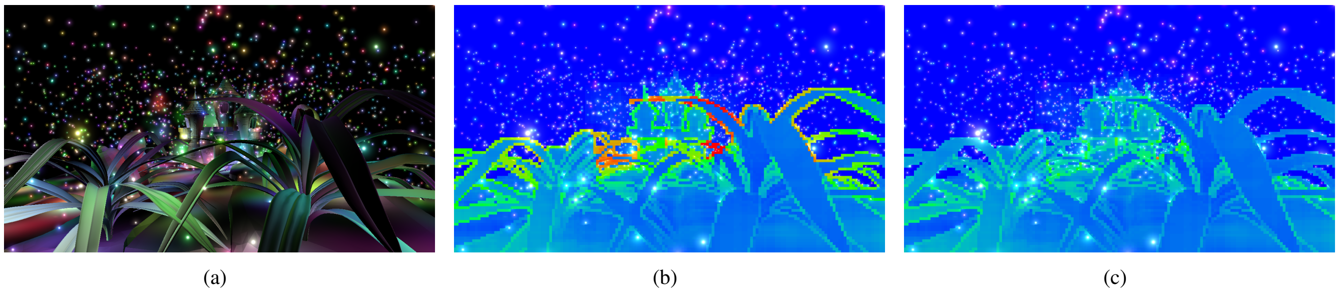


Figure 1: *(a) A scene with a large depth variance that the original Forward+ could not process efficiently. (b) Visualization of the number of lights per tile using frustum culling with maximum and minimum depth values in a tile. (c) Visualization of the number of lights per tile using the proposed 2.5D culling. Blue, green and red tiles have 0, 100, and 200 overlapping lights.*

1 Introduction

In recent years, deferred rendering, dating back to [Saito and Takahashi 1990], has gained in popularity for rendering in real time, especially in games. A deferred-rendering pipeline is good at dealing with many lights. However, it is not suited for handling arbitrary materials because it splits lighting and shading calculations. To overcome this drawback, Forward+ was developed [Harada et al. 2012]. Forward+ retains the ability to use a lot of lights, which is the primary advantage of a deferred-rendering pipeline, while it eliminates most of the restrictions, such as limited material variety and hardware anti-aliasing supports, at a lower computation cost. Forward+ has been already adapted in some games [Thomas 2012].

However, there are several parts of the Forward+ rendering pipeline that are inefficient when it comes to practical use. One of them is that the number of lights captured for a tile, which is a group of pixels, can be large if a tile has both foreground and background pixels, such as an edge of a model as shown in Fig. 1. This is because it is doing a 2D culling with minimum and maximum depth clip. Increasing of the number of lights captured for a tile directly affects shading performance.

To improve the culling accuracy (or reduce false positives), an obvious choice would be to use a 3D culling in which space is split into 3D cells. Although the use of 3D culling improves the culling accuracy, it adds complications to the culling and requires a larger memory footprint. The culling itself is an optimization, so we do not want to add overhead and want to keep it as simple as possible.

In this paper, we present a 2.5D culling for Forward+ that is simple, inexpensive, and effective. The proposed method culls lights that fall the void space between the foreground and background geometry. The proposed culling can be implemented by adding a few ALU instructions to the Forward+ rendering pipeline without adding additional data structure or memory consumption. Several tests are performed to evaluate the culling accuracy and computation overhead of the 2.5D culling.

*e-mail: takahiro.harada@amd.com

2 Related Works

The most relevant work is [Olsson et al. 2012] which also solves the same problem as ours. Their method is suited for a scene with a large number of lights, but the overhead of the algorithm makes it slower for a scene without more depth complexity or with a few thousands of lights, which is large enough for today's practical graphics applications such as games. Therefore, we need to select the scenes to which we apply the method. It also requires additional data structures, random memory access and more data transfer to the off-chip memory. These can be factors that degrade the run-time performance.

By contrast, our algorithm can be applied to any scene because the overhead is negligible, as we show in Sec. 6, and it does not require a scene to have massive amount of lights to benefit from the algorithm. Furthermore, our proposed method completes on the GPU without creating any additional data structures on the off-chip memory.

3 Forward+

In this section, we give a high-level description of Forward+ by comparing it to traditional forward- and deferred-rendering pipelines and then describe an implementation of Forward+.

3.1 Comparison of Real-time Rendering Pipelines

The rendering equation has to be solved to obtain a physically accurate rendered image. The rendering equation can be split into direct illumination and indirect illumination terms. The direct illumination term, which is the primary focus of this paper, can be rewritten to a summation when all the lights in the scene can be expressed explicitly:

$$L = \int_{\Omega} L_e f(x, w_{camera}, w_{light}) V(w_{light}) d\omega$$

$$\approx \sum_{i}^{n} \{L_e f(x, w_{camera}, w_{light}) V(w_{light})\} \quad (1)$$

where L_e, f, V are light emission, BRDF of surface point x in w_{camera} and w_{light} directions and visibility of the light source. However, solving this equation in real time is too computatilnally expensive because there are usually a large number of lights in the scene; thus, rendering pipelines approximate the equation so that it can be solved under the budget of a real-time application.

Traditional forward-rendering pipelines only pick m lights for each object. Thus, the rendering equation for traditional forward rendering is

$$L_{Forward} = \sum_{i=0}^{m} \{L_e f(x, w_{camera}, w_{light}) V'(w_{light})\} \quad (2)$$

where m is the number of lights evaluated for each object. Another approximation we use is for the visibility term. It is desirable to evaluate the visibility term for all the light sources, but this is expensive. Instead, a few lights are selected and occlusion is calculated for those by using a method such as shadow mapping.

This approximation is not too bad, but m is usually too small compared to n $(m < n)$. We need to use more lights to archieve more realistic results. Deferred-rendering pipelines were developed to overcome the drawback of forward-rendering pipelines and are used widely in today's games. Deferred lighting, which is a method of deferred-rendering pipeline, decouples light accumulation and shading, approximating the rendering equation as

$$L_{Deferred} = \sum_{i=0}^{\tilde{n}} \{L_e V'(w_{light})\} f(x, w_{camera}) \quad (3)$$

Deferred lighting can use more lights $\tilde{n}$ which is more than m $(m < \tilde{n} < n)$, but it drops w_{light} from BRDF function f. Thus, it can be said that this approach trades accuracy of materials or BRDFs for the ability to use more lights. [1]

Forward+ is a hybrid rendering pipeline of forward- and deferred-rendering pipelines. Forward+ transforms the rendering equation to

$$L_{Forward+} = \sum_{i=0}^{\tilde{n}} \{L_e f(x, w_{camera}, w_{light}) V'(w_{light})\} \quad (4)$$

An advantage of Forward+ compared to deferred lighting comes from the fact that the BRDF is not factorized out; instead, it is estimated for each light, which removes the restriction of the use of arbitrary materials while retaining the ability to use many lights. This is important because it improves the quality and realism of the final image.

3.2 Forward+ Overview

Forward+ is an extension of a forward-rendering pipeline. To process a large number of lights, it adds a light-culling stage to a traditional forward rendering pipeline. The rendering pipeline of Forward+ consists of three stages:

1. Depth prepass
2. Light culling
3. Final shading

[1]This depends on implementation. Deferred shading does not have this limitation but requires to write many variables for each pixel.

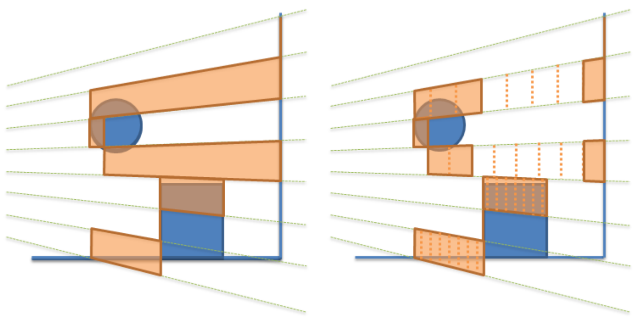


Figure 2: *A 2D illustration of frustum culling plus 2.5D culling. (a) Frustum culling creates a long frustum (orange) for a tile with foreground and background. If a light touches a frustum, it is captured. (b) 2.5D culling splits depth into cells, reducing the detection volume; thus, it does not capture lights falling between foreground and background. In this illustration, a frustum is split into eight cells.*

During light culling, light geometries are tested against a frustum of each tile that is clipped by the maximum and minimum depth values of a tile. This light culling works well if there is little variance in the depth in a tile. However, when a tile has a large depth variance, it can create a long frustum for a tile. This results in capturing a lot of unnecessary lights for a tile as we can see at the edge of geometries in Fig. 1(b). We can capture lights that have no influence on any of the pixels in a tile when a light falls in the void space in the frustum. As the number of lights reported for a tile increases, the computation cost of final shading increases. This is critical especially when the shading computation for a light is computationally expensive. This can happen often because one motivation for employing Forward+ is its ability to use sophisticated BRDFs for shading.

4 2.5D Culling

One obvious way to improve the accuracy of culling is to cull the lights using a 3D grid. However, this increases the computation and also the size of data to be exported. It is easy to develop a sophisticated culling but it should not be overkill. The proposed 2.5D culling constructs an adaptive 3D culling without adding much computation or stressing memory bandwidth.

The idea is illustrated in Fig. 2. The approach first constructs a frustum for a tile in the same way as screen-space culling does. Then the extent of the frustum is split into cells in the depth direction, and for each pixel in a tile we flag a cell to which the pixel belongs. We call the data we construct for a tile a frustum and an array of occupancy flags a depth mask.

To check overlap of light geometry on the tile, the light geometry first is checked against the frustum. If the light overlaps, a depth mask is created for the light. This is done by calculating the extent of the light geometry in the depth direction of the frustum and flagging the cells to that extent. By comparing the depth mask for a light to the depth mask for the tile, we can tell whether they overlap in the depth direction or not. Overlap of the light is reported only if there is at least one cell flagged by both depth masks.

If a tile has foreground and background geometry, the 2.5D culling can detect it and cull lights that fall between these two surfaces,

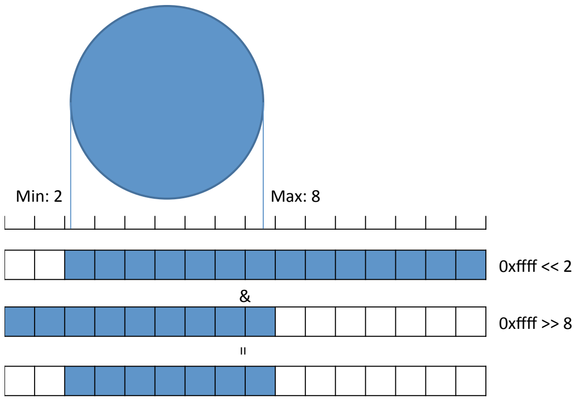


Figure 3: *Depth mask computation for a sphere.*

thus reducing the number of lights to be processed during the final shading.

4.1 Implementation

The proposed 2.5D culling is an extension to screen-space light culling. We first describe the implementation detail of screen-space light culling, which is not included in [Harada et al. 2012], and then extend it to 2.5D culling.

Screen-space light culling is performed by a compute shader executing a thread group for a tile. Each thread of a thread group first reads the depth of a pixel and calculates its extent in the depth direction. Then light geometry is read from a buffer and tested against the clipped frustum. Overlapping lights are accumulated in local data share (LDS); a per-pixel linked list could be used, but we did not employ it because it requires a few atomic operations to insert a light and pointer traversal on read. After all the lights are tested, the accumulated light index is flushed to the global memory at the end of the kernel. The kernel uses a global atomic operation per tile to allocate memory for the light list of the tile and fill the allocated memory location with the light indices. Pseudo-code for 2.5D culling is shown in Algorithm 1.

2.5D culling splits a frustum into 32 cells in the depth direction, as shown in Fig. 2(b). Of course, we can use much higher resolution, but it is a trade-off between culling accuracy (reducing more false positives) and memory footprint. We choose 32 because a depth mask can be stored in a 32-bit value and an overlap check can be performed by a logical operation. A depth mask is allocated in LDS for a thread group to make it is available from all the threads in a group. This 32-bit LDS memory allocation is the only overhead to the memory compared to the original implementation by [Harada et al. 2012].

The first modification to the light-culling kernel is constructing the depth mask of the surface. This is performed after calculating the frustum extent in the depth direction. The pitch of a cell is calculated from the depth extent. Once the pitch and the minimum depth value are obtained, any depth value can be converted to cell index. To create the depth mask for a tile, we iterate through all the pixels in the tile and calculate a cell index for each pixel. Then a flag for the occupied cell is created by a bit-shift, which is used to mark the depth mask in LDS using an atomic-or operation (Ln. 6 in Algorithm 1).

After that, overlap of a frustum to all the lights are tested in parallel. If we find a light overlapping to the frustum, a depth mask is created for the light (Ln. 10 in Algorithm 1). The minimum and maximum depth values of the geometry are calculated and converted to cell indices. Once the cell indices for the maximum and minimum depth values are calculated, two bit-shift operations and a bit-and operation are necessary to create the depth mask for the light (Fig. 3). If a light and any pixel in the tile occupy the same cell, both have flagged the cell so taking logical and operation between these two masks is enough to check the overlap (Ln. 11 in Algorithm 1).

Algorithm 1 Pseudo-code for the 2.5D culling. The difference from frustum culling is highlighted in red.

```
1:  frustum[0-4] ← Compute 4 planes at the boundary of a tile
2:  z ← Fetch depth value of the pixel
3:  ldsMinZ ← atomMin(z)
4:  ldsMaxZ ← atomMax(z)
5:  frustum[5,6] ← Compute 2 planes using ldsMinZ, ldsMaxZ
6:  depthMaskT ← atomOr( 1 ≪ getCellIndex(z) )
7:  for all the lights do
8:      iLight ← lights[i]
9:      if overlaps( iLight, frustum ) then
10:         depthMaskL ← Compute mask using light extent
11:         overlapping ← depthMaskT ∧ depthMaskL
12:         if overlapping then
13:             appendLight( i )
14:         end if
15:     end if
16: end for
17: flushLightIndices()
```

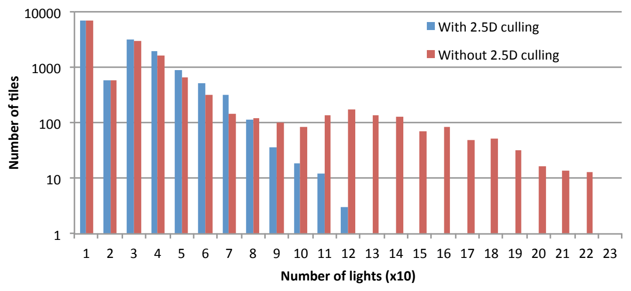
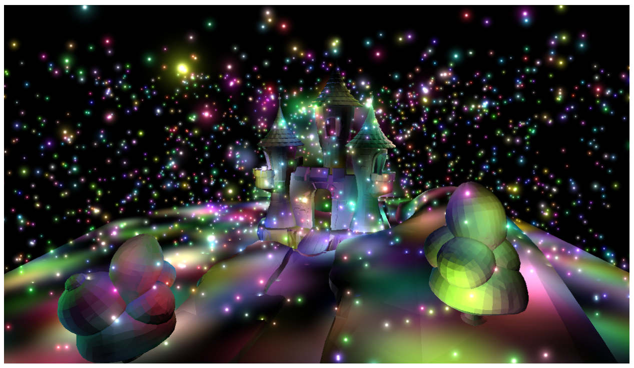


Figure 4: *The count of tiles in terms of the number of lights for the scene shown in Fig. 1(a).*

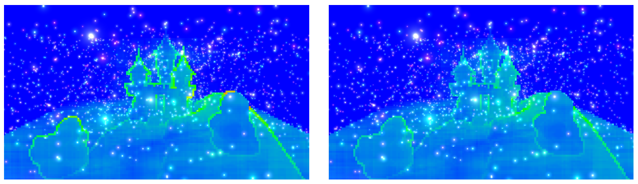
5 Results and Discussion

The proposed method is implemented using Direct Compute in DirectX 11. We set tile size to 8 × 8. We took several scenes and counted the number of lights per tile with the original Forward+ and Forward+ with our proposed 2.5D culling. The first benchmark is performed against the scene in Fig. 1(a), which has a large variance in the depth. Figs. 1(b) and 1(c) visualize the number of lights overlapping each tile using Forward+ with frustum culling and the proposed 2.5D culling. Fig. 1(b) makes clear that tiles which has an edge of an object capture a large number of lights. The number of overlapping lights is reduced dramatically when the 2.5D culling is used (Fig. 1(c)).

We also counted the number of lights overlapping each tile for quantitative comparison of these two culling methods. The comparison is shown in Fig. 4. Without the proposed method, there are a lot of tiles with more than 200 lights overlapping. However, by using the 2.5D culling, a tile has 120 overlapping lights at most. The benefit we can get from final shading depends on the imple-



(a)



(b) (c)

Figure 5: *(a) A scene without a large depth variance. (b) Visualization of the number of lights per tile using frustum culling with maximum and minimum depth values in a tile. (c) Visualization of the number of lights per tile using the proposed 2.5D culling. Blue, green and red tiles have 0, 100, and 200 overlapping lights.*

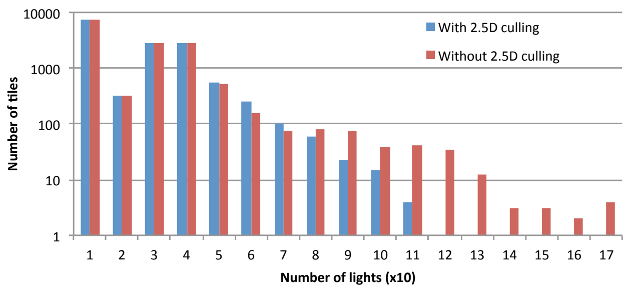


Figure 6: *The count of tiles in terms of the number of lights for the scene shown in Fig. 5(a).*

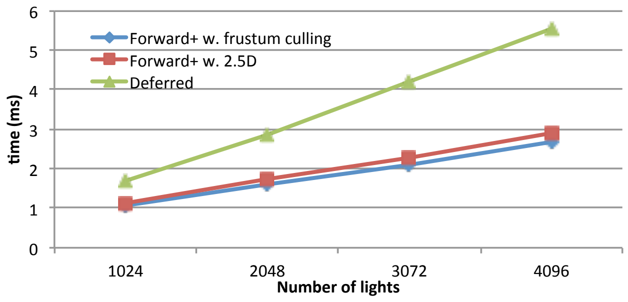


Figure 7: *Comparison of computation time for the light-culling stage of Forward+ using frustum culling only and frustum culling and the 2.5D culling by changing the number of lights in the scene shown in Fig. 1(a). Computation time for the light accumulation in compute-based deferred lighting is also shown.*

mentation of the shader, but culling eliminates a lot of unnecessary memory reads and computation for the final shader. Detailed performance investigation of an entire pipeline is a future work.

We also performed a test on the scene shown in Fig. 5(a), which does not have as much depth variance as the scene in Fig. 1(a). Because the depth difference is not large in these scenes, the number of lights overlapping a tile with an edge of an object is less than in the previous scene. However, we can still see that the temperature is low when 2.5D culling is used. A quantitative comparison is shown in Fig. 6. Although the improvement is not as large as in was in Fig. 1, the proposed method reduces the number of overlapping lights on tiles.

Fig. 7 compares the computation time for the light-culling stage for the scene of Fig. 1(a) as measured on an AMD Radeon^{TM}HD 7970 GPU. This comparison indicates the overhead of additional computation in 2.5D culling is less than 10% of the time without culling; when there are 1,024 lights, the overhead is about 5%. Therefore, the 2.5D culling is effective regardless of the number of lights in the scene. This graph also contains the light accumulation time of the compute-based deferred lighting; the light-culling stage with the 2.5D culling much faster than its counterpart in deferred lighting [Andersson 2011].

6 Conclusion

We presented a 2.5D culling that also culls light in the depth direction. The proposed 2.5D culling was integrated in the Forward+ rendering pipeline and several experiments were performed to evaluate the culling accuracy. These experiments showed that the proposed method can reduce the number of false positives significantly, especially for a tile with a lot of overlapped lights, jus by using tile-frustum culling. We also measured the computation overhead of 2.5D culling and confirmed that the overhead is less than 10% compared to the tile-frustum culling.

The proposed 2.5D culling can be applied to for Forward+ and is also applicable to any computation using tile culling such as tiled deferred-rendering pipeline. Future works include performance evaluation of shading with the proposed method in a fully authored scene and extension of the proposed method for rendering transparent objects.

References

ANDERSSON, J. 2011. DirectX 11 Rendering in Battlefield 3. *Game Developers Conference*.

HARADA, T., MCKEE, J., AND YANG, J. C. 2012. Forward+: Bringing Deferred Lighting to the Next Level. In *Proc. of Eurographics 2012 - Short Papers*, 5–8.

OLSSON, O., BILLETER, M., AND ASSARSSON, U. 2012. Clustered deferred and forward shading. In *HPG '12: Proceedings of the Conference on High Performance Graphics 2012*.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4 (Sept.), 197–206.

THOMAS, G. 2012. Compute in the future of gaming. In *AMD Fusion Developers Summit*.