Implicit Clustered Deferred Shading

Kevin Örtegren[*]
Blekinge Institute of Technology

Figure 1: *CryTek Sponza scene with 1000 point lights.*

Abstract

In this report, a CPU-based light culling technique is presented. This method is designed for real-time applications that use large amounts of dynamic point lights. The technique is unaffected by depth discontinuities and is easily implemented into existing deferred rendering pipelines. More than 95% of lighting calculations are omitted, achieving an overall speed-up of up to 20 times running the CryTek Sponza [Meinl 2010] scene with 1000 point lights.

CR Categories: D.2.8 [Software]: Software Engineering—Metrics
I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism

Keywords: clustered shading, lighting, light culling

1 Introduction

Dynamic lights are a crucial part of making a virtual scene seem realistic and alive. Accurate lighting calculations are expensive and have been a major restriction in real-time applications. Many techniques have been developed to move the expensive per frame lighting calculations to an offline pass, where light maps are created and used for rendering static objects. These techniques can not be applied to dynamic environments and, with destructible and dynamic environments becoming more common in modern games, dynamic lighting solutions must be applied instead. In recent years, many new lighting pipelines have been explored and used in games to increase the number of dynamic light sources per scene.
This paper presents a CPU-based variation of Clustered Shading [Persson and Olsson 2013] , which is a technique that improves on the currently popular Tiled Shading [Olsson and Assarsson 2011; Swoboda 2009; Balestra and Engstad 2008; Andersson 2009] by utilizing higher dimensional tiles. The view-frustum is divided into 3D clusters instead of 2D tiles and addresses the depth discontinuity problem present in the Tiled Shading technique. The main goal of this paper is to show how Clustered Shading can be implemented in a traditional deferred rendering pipeline in a practical way to enable the use of many more lights per scene. Comparisons against the traditional deferred rendering technique are made to establish the level of lighting calculation reduction.

[*]e-mail: kevinortegren@gmail.com

2 Related work

Clustered Shading is a new technique which has not been covered to a large extent, nor has it been established in the game industry as there are no commercially available games using the technique as of now. It was first introduced by [Olsson et al. 2012a] in 2012 and their recent work has also focused on Clustered Shading [Olsson et al. 2012b]. Other, non-published works and presentations, are available from Intel who has released a demo application on Forward Clustered Shading [F. 2014] and from Avalanche Studios who has presented a practical solution for Clustered Shading in their game engine [Persson and Olsson 2013].

Tiled Deferred Shading is a widely used technique in game engines today and was developed as a solution to the growing number of dynamic lights in modern games. Attempts to solve the existing problems of Tiled Deferred Shading have been made through the years and have spawned a number of variations and extensions, most notably 2.5D culling [Harada et al. 2013] which effectively moves Tiled Shading towards 3D clusters.

3 Problem statement

The problem addressed in this paper is how to avoid doing unnecessary calculations with lights that are not affecting any geometry or not influencing a specific fragment.

3.1 Traditional Deferred Shading

The deferred rendering full screen lighting pass is the worst case solution in deferred shading as it calculates $numberOfFragments * numberOfLights$. This means that if the scene contains 1000 lights, each fragment will do lighting calculations based on all 1000 lights, even if none of the lights affect a specific fragment. This is a massive waste of computational resources and the premiss of this paper.

3.2 Tiled Deferred Shading

Tiled Deferred Shading divides the view-frustum into 2D tiles that start at the closest geometry and extend to the furthest geometry in each tile and assigns intersecting lights to the tiles. Each fragment will perform lighting calculations based on the lights in the corresponding tile. This reduces the number of light calculations per fragment significantly. The drawback of Tiled Deferred Shading is the scene dependency due to depth discontinuities. When a tile is based on geometry close the the viewer as well as geometry far away in the distance, the tiles become very long and includes a lot of empty space where lights are present. These lights, which do not affect any geometry, will be included in the lighting calculations of a fragment in that tile.

4 Methods

4.1 Algorithm

The basic steps of the algorithm are based on the SIGGRAPH presentation [Persson and Olsson 2013]. The algorithm is listed below with details of each step in the following sections.

1. Render scene to G-Buffers.
2. Build cluster planes.
3. Assign lights to clusters.
4. Fill 3D cluster texture and light index list.
5. Shade in pixel/fragment shader full screen pass.

As with any deferred rendering technique, the scene must be rendered to the G-Buffers. The only part used from the G-Buffers in this technique is the depth buffer which means that forward rendering with a depth prepass could benefit from this technique as well. The second step is where the planes that make up the view-frustum 3D cluster structure are calculated. The third step finds out which lights affects which clusters. In the fourth step the index and offset of every cluster are stored as well as a corresponding light index list.

4.1.1 Build cluster planes

The planes make up the subdivision of the view-frustum and decides the spacing in depth. The depth spacing is the spatial distribution of z-planes and can be modified after specific needs and applications. For this paper, a uniform world space depth spacing is used, see Figure 2 on page 2 for a top down view of a uniformly spaced frustum. An important thing to note here is that building the planes in world space requires them to be rebuilt every frame, building the planes in view space does not. Building the planes in view space will however introduce the need to transform all lights to view space as well, which can be more expensive than rebuilding planes at a certain number of lights. The outer planes are same as the sides of the viewing frustum; near, far, right, left, top and bottom planes.

4.1.2 Assign lights to clusters

Light assignment is the most important part of this algorithm as it scales linearly with the number of lights. This is the part where every cluster must find all the lights that influence it. All point lights are represented as a sphere with a 3D position and a radius. The solution presented here iterates over every plane to step-by-step reduce the frustum until a point light is contained in the smallest possible frustum with six cluster planes. For every iteration a sphere-

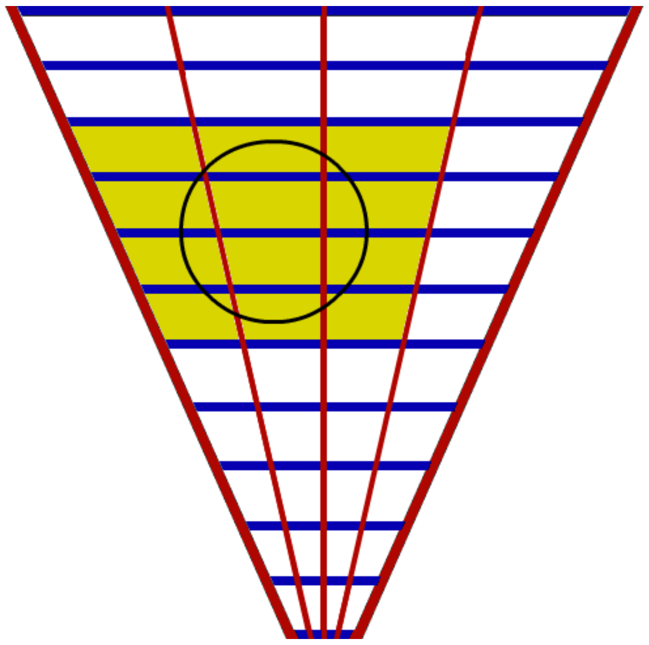

Figure 2: *Top down 2D view of an assigned light. The black circle represents a light sphere.*

plane intersection is performed to see if the current plane has entered the light sphere, this gives the plane on one side of the light. After finding a plane, the iterations continue to find the second plane, on the other side of the light sphere. This is performed three times, one for each axis, and results in six planes that contain the light. The clusters within the planes are thus affected by that light, see algorithm Figure 2 on page 2 for a 2D overview of an assigned light. This results in a worst case of $numberOfXPlanes + numberOfYPlanes + numberOfZPlanes$ sphere-plane intersections per light. Algorithm 1 shows the pseudo code for the x-plane reduction, the same applies for the y and z-planes. Each plane is represented by a point on the plane, which in this case is the virtual camera position for all the planes as they all go through there, and a normalized plane normal. In the algorithm 1 listing xpn represents the x-plane normal, lr is the light radius, lp is the light position and cp is the camera position. Variables lxp and rxp are the result for the xplanes. They are indices to the the left x-plane and the right x-plane for the currently processed light. Once the algorithm has been run for x, y and z-planes the final result is six indices, as mentioned earlier. These six indices are used to add the light to the 3D clusters.

4.1.3 Fill 3D cluster texture and light index list

As there are no real dynamic buffers on the GPU some extra data structures and buffers have to be added to conserve memory. The main problem is that there is no way of knowing how many lights will be assigned to each cluster and it would be wasteful to allocate a large amount of light slots for each cluster on the GPU. This is where the light index list provides a solution. The light index list acts as a pool for all the clusters combined and the clusters simply points into the light index list and provides an offset. In Figure 3 on page 3 a), the first cluster (top left) points to element zero in the index light list and there are two lights in that cluster, indicated by the second element in that cluster. Using this approach a calculated safe limit can be used when allocating memory for all clusters on the GPU. Each element in the light index list points to a point light. There is only one copy of every point light on the GPU, but multiple

Algorithm 1 Light assignment

```
1:  for each light do
2:      if light is inside outer frustum then
3:          for each $Xplane$ do
4:              $dist = xpn \cdot (lp + (-xpn * lr) - cp)$
5:              if $dist \leq 0$ then
6:                  $lxp = Xplane - 1$
7:                  for each remaining $Xplane$ do
8:                      $dist = xpn \cdot (lp + (xpn * lr) - cp)$
9:                      $rxp = Xplane - 1$
10:                     if $dist \leq 0$ then
11:                         break
12:                     end if
13:                 end for
14:                 break
15:             end if
16:         end for
17:     end if
18: end for
```

light indices can point to the same light. Point lights contain much more data than a single index and this provides a memory efficient solution.

To keep all the data tightly packed on the GPU some sacrifices must be made on the CPU side. When performing the light assignment on the CPU, all the clusters must store their own light indices temporarily to avoid having to sort anything or dynamically allocate more memory when filling the light index buffer. Most of the time there is much more system RAM than video RAM in a system and this trade-off ensures the high performance requirement when assigning lights.

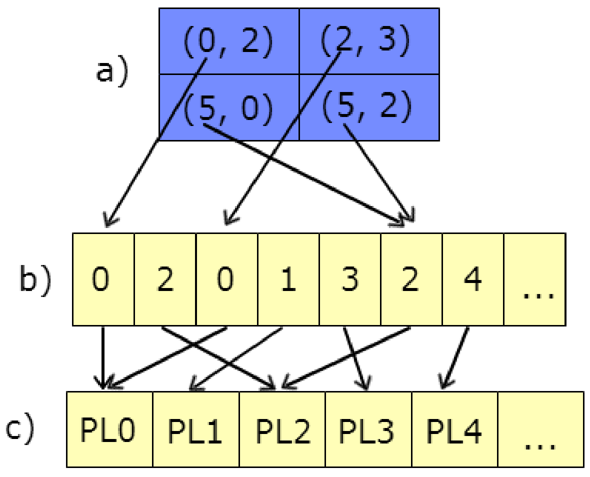

Figure 3: *a) Represents a 3D texture with two components per cell. b) The light index list. c) The buffer containing point light data.*

4.1.4 Shade in pixel/fragment shader full screen pass

To light a fragment in the shader program it first needs to find which cluster it has to fetch light from. This is done by sampling the depth buffer from the previous G-Buffer pass. The depth tells the fragment which z-index the cluster has and then the screen position of the fragment tells it what x and y-index the cluster has. To extract the x and y coordinates in an efficient manner a power of two sized x and y cluster width and height is used. In that case a simple bit shift operation is needed to find the x and y coordinate in the 3D texture. Once the 3D coordinate for the cluster has been calculated, it is only a matter of lighting the fragment with the point lights in that cluster.

4.2 Experiment

The testing and data gathering for this paper using the described implementation was performed on a laptop running a Core i7-3630QM CPU at 2.4GHz, 8GB DDR3 RAM and a Nvidia GTX660m on a Windows 8.1 64bit operating system. The implementation uses DirectX 11 from the Windows 8 SDK. The scene is the CryTek Sponza model. The G-Buffer contains a normal texture, a position texture, a diffuse texture and a depth texture. The resolution of the back buffer is 1536x768 to keep all the clusters even in size, even though resolutions that are not divisible by a power of two are no problem at all. The lighting model used is the per fragment Phong shading model but without the specular calculation.

5 Result

The implementation provides an efficient light culling and shading technique with good results. As can be seen in Figure 4 on page 3 there are less than 50 lights affecting each fragment when having 1000 lights in the scene. Using traditional deferred shading without light culling results in 1000 lights per fragment.

The VRAM column shows the theoretical overhead memory needed on the GPU to perform Clustered Shading. This does not include the point light data.

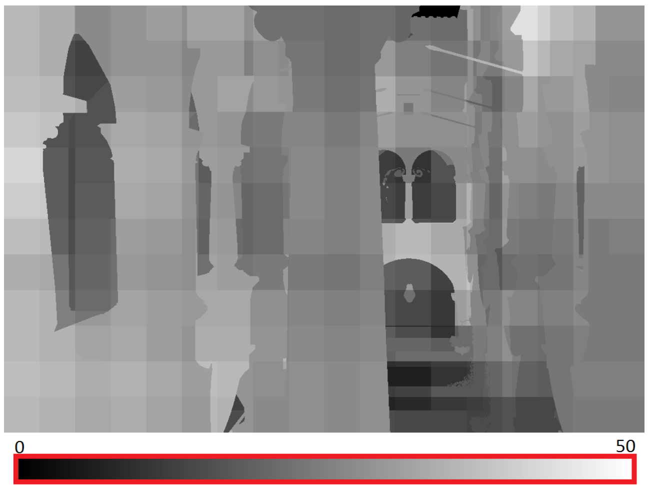

Figure 4: *The color grading describes the number of light calculations per fragment. Black is 0 lights and white is 50 lights.*

# clusters	Shading	Light assignment	Total	VRAM
24x12x16	8.12ms	1.04ms	9.16ms	137kB
24x12x32	6.08ms	2.2ms	8.28ms	273kB
24x12x64	5.97ms	3.13ms	9.1ms	546kB
No clustering	156ms	-	156ms	0kB

Table 1: *Results from 1000 lights*

# clusters	Shading	Light assignment	Total	VRAM
24x12x16	0.53ms	0.1ms	0.63ms	27kB
24x12x32	0.5ms	0.14ms	0.64ms	54kB
24x12x64	0.5ms	0.28ms	0.78ms	109kB
No clustering	1.6ms	-	1.6ms	0kB

Table 2: *Results from 10 lights*

6 Discussion

The results in tables 2 and 1 show a dramatic decrease in shading time in both 10 lights and 1000 lights. The traditional deferred shading scales linearly with the number of lights and is 18.8 times slower than the best cluster setup. At 10 lights it is only 2.5 times slower than the best clustering setup. The scenarios from the result section only differ in the number of z-planes, causing the cluster structure to produce more and thinner clusters. This is why there is a diminishing result at higher number of z planes. There are always lights affecting a fragment and no matter how small the cluster is those lights will still be used when shading. This is a drawback of the static cluster structure size of this technique. In [Olsson et al. 2012a] they present alternative clustering techniques that are more flexible and adaptive which results in better shading times, but that gained shading time is lost in light assignment and other surrounding steps. The balance between light assignment and shading time is very hard to pin point as they both depend on the number and position of lights in the viewing frustum.

One thing to note is that the light assignment time does not increase linearly. This is because of the light assignment algorithm and how it iterates over planes when reducing the frustum. For the light assignment to increase linearly all the lights would have to be all the way back at the far plane, as the iteration starts from the near plane and works its way back. A way to increase the efficiency of the light assignment would be to use a binary frustum reduction method. Instead of iterating from one side to another the intersection could be performed like binary search, jumping to the middle plane and finding out which way to jump next by using the existing distance. Binary frustum reduction coupled with threading of the three different plane phases of the light assignment could yield a significant speed up. The most significant improvement to the shading time would be to achieve a tighter cluster fit when assigning lights. Around 20-30% of clusters are incorrectly assigned due to the nature of fitting a sphere in a collection of frustums [Persson and Olsson 2013].

As can be seen in Figure 4 on page 3 there are no depth discontinuities present, even with the camera placed partially behind a pillar. The small pillar in the distance is shaded using a different cluster from the background, where as a Tiled Shading implementation would use all the lights between the pillar and the background.

7 Conclusion

Large amounts of dynamic lights are becoming the norm in modern games. Many techniques have emerged and are used to effectively increase the number of lights per scene. The technique presented in this paper is fast, light weight and easy to implement. As a bonus it runs on the CPU which allows it to run on older hardware that do not have access to the latest GPUs. Compared to Tiled Shading, this approach has the advantage of not being dependent on the scene depth discontinuities and provides stable performance.

8 Future work

As discussed there are some ways of increasing the efficiency of the light assignment and increase the number of clusters to achieve better shading times. With an implicit clustering structure the bottleneck will soon become the data transfer between CPU and GPU if the clusters are small enough and the number of lights increases. This leads the future of this technique towards the GPU. It is a highly parallelizable algorithm and the only real problem is how to store and manage the memory for fast access when shading. A GPU version of this implicit clustering structure is something that will be looked into.

Other light types and shapes must be supported for the technique to become widely accepted. Especially now with the current surge of area lights in game engines.

References

ANDERSSON, J. 2009. Parallel graphics in frostbite-current & future. *SIGGRAPH Course: Beyond Programmable Shading*.

BALESTRA, C., AND ENGSTAD, P.-K. 2008. The technology of uncharted: Drake's fortune. In *Game Developer Conference*.

F., M., 2014. Forward Clustered Shading. `https://software.intel.com/en-us/articles/forward-clustered-shading`. [Online; accessed 09-January-2015].

HARADA, T., MCKEE, J., AND YANG, J. C. 2013. Forward+: A step toward film-style shading in real time. *GPU Pro 4: Advanced Rendering Techniques 4*, 115.

MEINL, F., 2010. The Atrium Sponza Palace, Dubrovnik. `http://www.crytek.com/cryengine/cryengine3/downloads`. [Online; accessed 09-January-2015].

OLSSON, O., AND ASSARSSON, U. 2011. Tiled shading. *Journal of Graphics, GPU, and Game Tools 15*, 4, 235–251.

OLSSON, O., BILLETER, M., AND ASSARSSON, U. 2012. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, Eurographics Association, 87–96.

OLSSON, O., BILLETER, M., AND ASSARSSON, U. 2012. Tiled and clustered forward shading. In *SIGGRAPH '12: ACM SIGGRAPH 2012 Talks*, ACM, New York, NY, USA.

PERSSON, E., AND OLSSON, O., 2013. Practical clustered deferred and forward shading. SIGGRAPH Course: Advances in Real-Time Rendering in Games.

SWOBODA, M. 2009. Deferred lighting and post processing on playstation 3. In *Game Developer Conference*.