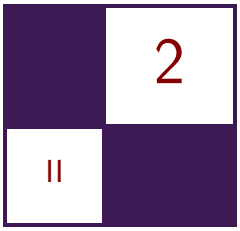

Fine Pruned Tiled Light Lists

Morten S. Mikkelsen

2.1 Overview

In this chapter we present a new tiled lighting variant with a primary focus on optimization for the AMD Graphics Core Next (GCN) architecture. Our approach was used for the game *Rise of the Tomb Raider*. In particular, we leverage asynchronous compute by interleaving light list generation with rendering of shadow maps. Light list generation is done per tile in two steps within the same compute kernel. An initial coarse pass that generates a light list in local storage using simple screen-space AABB bounding volume intersection testing regardless of light type. The second step is fine pruning, which performs further testing on the coarse list by testing each pixel in the tile if the point in 3D space is inside the true shape of the light source.

Furthermore, we present an efficient hybrid solution between tiled deferred and tiled forward.

2.2 Introduction

Traditionally, real-time deferred lighting is done using alpha blending to accumulate lighting contributions one light at a time. The main big advantage is the ability to assign and apply lights specifically to pixels representing points in 3D space inside the light volume. With a basic forward-lit shading model, on the other hand, light lists are built on the CPU per mesh instance based on bounding volume intersection tests between the mesh instance and the light volumes. This approach often results in a significant overhead in light count to process per pixel, particularly for large meshes because the light list is shared for all pixels occupied on screen by the mesh instance.

Recently, since the introduction of DirectX 11, compute-based tiled lighting has become a popular alternative to deferred lighting. Tiled lighting works by representing the frame buffer as an $n \times m$ grid of tiles where the tiles are of a fixed resolution. The GPU is used to generate a list of indices per tile containing

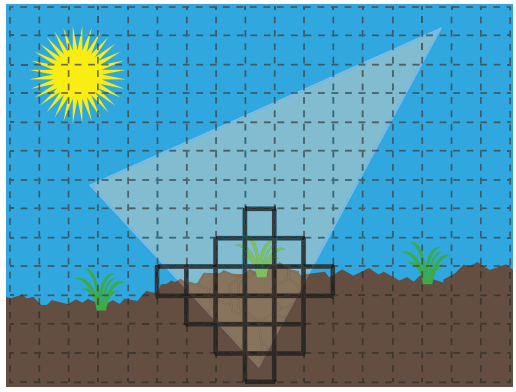

Figure 2.1. The screen separated into tiles. A spot light is shown intersecting with the foreground. This light is added to the light list of every tile containing a valid intersection, which is indicated here with a solid tile boundary.

references to the lights overlapping screen boundaries of a tile. A lighting pass pulls the light list from the tile containing the currently processed pixel (see Figure 2.1).

This is a high-level overview of how a basic tiled lighting scheme works in compute:

1. Per tile
   (a) For each tile find a minimum and a maximum depth in the depth buffer.
   (b) Each thread checks a disjoint subset of lights by bounding sphere against tile bounds.
   (c) Indices to lights intersecting the tile are stored in local data storage (LDS).
   (d) Final list is available to all threads for further processing.

Recently several proposed methods for tiled lighting have emerged such as AMD's Forward+ Tiled Lighting [Harada et al. 12], which is primarily aimed at moving away from traditional deferred lighting in order to leverage EQAA. This method partitions tiles evenly by depth into cells, thus making it a 3D grid of light lists $n \times m \times l$. Culling is performed by testing the bounding sphere of the light against the side planes of the tile frustum and the near and far plane of the cells. Another known variant is Insomniacs' Light Linked List [Bezrati 14], which proposes a solution where the footprint of the light lists is reduced/managed using linked lists on the GPU. Another variant is Clustered Deferred and Forward

Shading [Olsson et al. 12], which reduces tile occupancy further by clustering. This achieves a more ideal partitioning of tiles into cells.

Among main advantages of tiled lighting are the following:

1. Tiled deferred lighting is single pass since each pixel is lit simply by looping over the lights referenced by the list stored in the corresponding tile. This makes the approach more resilient to overlapping lights than traditional deferred lighting because the data in the G-buffer is only pulled once and because the resulting color is only written to the frame buffer once.

2. Unlike traditional deferred lighting, there exists a forward variant when using tiled lighting. The reason is that as we draw the polygonal meshes, we can pull the same light list in-process from the tile containing the pixel being shaded.

3. A less commonly known advantage to using tiled lighting is that the light list generation is an ideal candidate for asynchronous compute, which allows us to interleave this processing with unrelated graphics work earlier in the frame update.

Previous methods such as [Harada et al. 12] and [Olsson et al. 12] have excess lights in the per tile light lists because these are built based on a simple bounding spheres intersection test. Additional redundancy exists with all previous techniques because the cells contain significant amounts of unoccupied space. In AAA games many of our lights are not spheres, and in fact, we must support several light shapes such as cones, capsules, and boxes with different features. Building lists based on bounding volumes and partitioned tile bounds alone leaves much redundancy in the light lists compared to the final list we end up with when fine pruning.

We found that by writing a lean dedicated compute shader to perform fine pruning, we were able to achieve significant gains due to the more complex shader used for lighting not having to deal with the redundant lights in the list. Furthermore, the separation between light list building and actual lighting allowed us to run the list building in asynchronous compute during rendering of shadow maps, which in practice gives us fine pruned lists for free.

Furthermore, our approach is a hybrid between tiled deferred and tiled forward lighting. This allows us to light the majority of pixels by a deferred approach using a narrow G-buffer, which is more hardware efficient, and then deviate from this for cases where we need a material-specific lighting model by using tiled forward.

2.3 Our Method

Previous papers on tiled lighting such as [Harada et al. 12] and [Olsson et al. 12] are particularly aimed at processing high quantities of sphere lights. The num-

bers quoted are in the 1–2 thousand range. Furthermore, these papers describe algorithms that are designed to handle scenarios where the lights have a relatively optimal distribution in space. While our method is also capable of handling high numbers of lights, we found we generally have no more than 40–120 lights inside the camera frustum in our real world levels. In our case we found that we often have fewer very large lights that occupy the same space. Many of our lights are large spot lights that are narrow to achieve a good distribution of pixels in the shadow map. The bounding sphere is a bad representation in this case. Ultimately, without additional culling, our light lists would contain high numbers of lights, several of which do not affect any of the actual tile pixels.

In every frame we receive a set of lights that have been classified visible (inside the camera frustum) by the cell and portal system. For each tile on the screen, we generate a *fine pruned light list*. Each light is included only if at least one pixel in the tile represents a point in 3D space that is inside the light volume. Testing all pixels in a tile against all visible lights is prohibitively expensive. To solve this, we first build a *coarse light list* containing lights whose screen-space axis aligned bounding box (AABB) intersects the tile boundary. The tile boundary is trivially defined by its xy-region on the screen and minimum and maximum depths in the depth buffer within tile region. We determine, on the GPU, the screen-space AABB around each visible light.

The process is described below in pseudo-code.

1. Per camera
   (a) On the CPU find lights that intersect the camera frustum.
   (b) Sort this set of lights by shape.
   (c) On the GPU find the tight screen-space AABB per light source regardless of shape. This is done by finding the intersection volume between the camera and a convex hull for the light. We further constrain the AABB using a bounding sphere of the light.
2. Per 16×16 pixel tile
   (a) For each tile find a minimum and a maximum depth in the depth buffer.
   (b) Each compute thread tests the intersection of a disjoint subset of lights by an AABB against tile bounds.
   (c) Indices to lights intersecting the tile are stored in LDS. We refer to this as the *coarse list.*
   (d) In the same kernel loop over the coarse list of lights.
      i. Each thread tests four pixels of the tile depth buffer to see if these are inside the true shape of the light.

ii. The status of the test is stored in a bit field maintained by each thread where each bit represents the corresponding coarse light.

(e) Perform a bitwise OR of all bit fields into a single bit field and use it to generate a *fine pruned light list*.

The distinction between fine pruning and performing an early out during lighting is, in concept, subtle. However, the difference is significant for two reasons. First, the shader associated with lighting consumes more resources relative to a lean shader dedicated to culling, which, as we describe in the next section, has implications on performance. Second, by using asynchronous compute, we can absorb most of the cost of fine pruning, which includes the cost of looping through redundant lights.

2.4 Implementation Details

In the following we are targeting the AMD GCN architecture specifically, though the practices are generally good for any modern-day GPU. A modern GPU core hides latency by shuffling through jobs. We will refer to these cores as a CU (compute unit). All work is packaged into wavefronts. Whether it is compute, vertex shading, pixel shading, etc., each CU can harbor up to 40 wavefronts, and each wavefront represents 64 threads. These threads run in lock-step similar to how SSE4 is 4 wide running in lock-step. The pool of resources such as registers and local store LDS are shared on each CU, which implies that the more you consume these, the fewer jobs get to occupy each CU, which means the GPU's ability to hide latencies deteriorates dramatically.

As it turns out, the rendering of shadow maps and generation of fine pruned light lists are a great match. According to our timings, shadow map rendering generally takes 2–4 ms in our game. Furthermore, it is a process that generates very few wavefronts of work and relies primarily on the primitive scan converter and trafficking of data. The reason for this is that shadow map rendering is a depth-only pass, which means no actual pixel shading CU work is generated for opaque meshes. Generating fine pruned light lists, on the other hand, is primarily propagating ALU-heavy wavefronts. This allows us to absorb most of the time spent on generating the lists using asynchronous compute.

Let us describe the algorithm steps in detail. First, Step 1(a) is to gather all visible lights in the frame. We do this using a typical cell and portal system on the CPU.

In Step 1(b) we sort, on the CPU, the visible lights by their type of shape. This allows us to process the lights using a fixed sequence of loops where each loop is dedicated to a specific light type. This is particularly important in the context of tiled forward lighting since in this case the 64 pixels being processed in a wavefront do often not exist in the same tile. Because the 64 threads run in lock-step, a divergence in execution path is inefficient. Having sorted the lights

by type maximizes the likelihood that all threads are in alignment execution path-wise. In our case sphere/capsule is one type/execution path, cone/wedge is a type, and box is the final type.

Next, in Step 1(c) we find the screen-space AABB for each light in the visible set. As input each light source is represented by an oriented bounding box (OBB) with a nonuniform scale at the top four vertices, which allows us to represent narrow spot lights and wedges better. To determine the AABB for the light, we find the point set to the intersection volume between the camera frustum and the convex bounding volume. This is done by frustum clipping the quads of the scaled OBB and using the final point set of each resulting fan to update the AABB. Any of the eight points of the camera frustum that are inside the convex bounding volume must also be applied to the fitting of the AABB. Last, we determine the AABB around the bounding sphere of the light and then store the intersection between this and the already established AABB as the final result. It should be noted that though this is a lot of processing, it is done once per camera and *not per tile*. This work can be done on the CPU but we do it on the GPU as an asynchronous compute shader.

In Step 2 the work of generating the final per-tile light list is performed, and we describe the various components to it in the following. All the parts within Step 2 are performed on a per-tile level within one compute kernel. Since the tile size is 16×16 pixels, the dispatch of the kernel is executed using the following threadgroup counts: (width + 15)/16, (height + 15)/16, and 1. The kernel is declared as a single wavefront threadgroup: $64 \times 1 \times 1$.

First, in Step 2(a) we must establish the screen-space AABB associated with the tile being operated on by the threadgroup. Each of the 64 threads reads four individual depths of the tile and establish the minimum and maximum of the four samples. Next, the collective minimum and maximum of the tile is established using `InterlockedMin()` and `InterlockedMax()`, which are HLSL intrinsic functions.

In Steps 2(b)–(c) we perform the initial coarse pruning test. Each of the visible lights will have its screen-space AABB tested for intersection against the AABB of the tile regardless of the true light shape. Each thread handles a disjoint subset of lights and thus performs `numVisibleLights`/64 iterations. Furthermore, using a single wavefront threadgroup allows us to preserve the order of the lights passing the coarse test because the 64 threads run in lock-step. The resulting coarse list of indices to lights is stored in LDS.

It is worth noting that the screen-space AABB corresponds to a sheared subfrustum in the camera space, as shown in Figure 2.2. In [Harada et al. 12] and [Olsson et al. 12] tiled lighting is implemented such that the bounding sphere around each light is tested against the frustum planes associated with each tile. However, we can do the same test faster when we already know the screen-space AABB for each light. This also allows for a tighter fit than a bounding sphere around certain light types, such as a spot light, which allows us to spend less

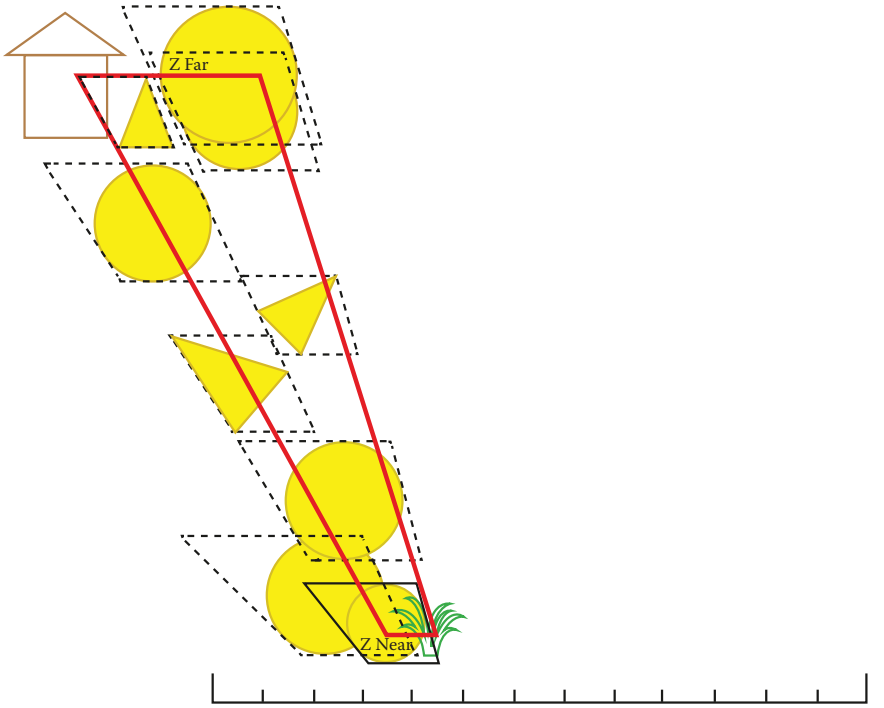

Figure 2.2. The sheared frustum associated with a tile from the frontmost pixel to the one farthest away. In this case there are six sphere lights and three spot lights. All nine lights pass the coarse intersection test but only one passes the fine pruned intersection test.

time on fine pruning. Potentially using AABB for the test also leads to a lower register count since we no longer need to keep six frustum planes for the tile in registers during iteration through the lights.

Finally, in Steps 2(d)–(e) we perform fine pruning. The fine pruned light list is a subset of the coarse light list. Each pixel in the tile is tested to see if the corresponding point in 3D space is inside the true shape of the light volume. Lights that contain one or more such points are put in the fine pruned light list. Each thread is responsible for testing 2×2 pixels of the 16×16 tile (1×1 for half-resolution lighting), and each thread maintains a record in the form of a 64-bit mask where each bit is enabled if the volume of the corresponding light contains at least one of the four points managed by the thread. Once we have processed all coarse lights in this manner, we finally determine the collective 64-bit mask by using the HLSL intrinsic `InterlockedOr()`. The resulting bit mask is used to remove redundancies from the coarse light list and write the final fine pruned list to memory. The effect of fine pruning is shown in Figure 2.2.

A CU has both vector and scalar registers. For a vector register VGPR, every thread has an individual dword, which gives a total footprint of 256 bytes per register. A scalar register SGPR is a dword that is shared for all threads with a total of 4 bytes per register. As mentioned at the beginning of this section, a high consumption of resources by a shader has a negative impact on performance. A shader used for lighting often consumes a relatively high amount of vector registers due to the overall complexity in code. If we can ensure during lighting that every thread of a wavefront represents a pixel in the same tile and thus pulls the same light list, then the attributes of the light such as color, position, fall-off, etc. can be pulled into SGPRs instead of VGPRs. It is easy to organize the treads accordingly in a compute shader; however, as will be discussed in the next section, we are using a full-screen stencil tested pixel shader for the final deferred lighting pass. This means that we are no longer in direct control of how the wavefronts are packaged. For a full-screen primitive we can ensure that the pixel shader wavefronts are fully packaged as 8×8 pixels by calling `SetScanConverterModeControl(false, false)` on the pixel shader used for deferred tiled lighting at initialization time. In addition to this, we must also run the pass after high stencil testing but before the low stencil test to maintain the execution in blocks of 8×8. Finally, we must inform the shader compiler to pull the attributes as scalar as opposed to vector. This is done by using the HLSL intrinsic `__XB_MakeUniform()` wherever we pull data from the tile.

For us this resulted in a drop from 76 to 52 in VGPRs and up to about a 1.3-ms reduction in execution time. In comparison, our kernel for generating fine pruned light lists consumes only 28 VGPRs, which as expected is much less.

The API calls mentioned above are for Xbox One only, though we suspect the equivalent API calls exist for Playstation 4 as well. No equivalent exists in the generic DirectX 11 API, though, so in this case there are two options: Either settle for vector registers on this platform, which preserves the stencil optimize, or alternatively implement the deferred lighting as a compute shader. In the latter case we would read the stencil as a texture look-up in the compute shader and perform the stencil test manually to avoid lighting the pixel twice.

2.5 Engine Integration

In order to achieve greater flexibility in shaders, it has become common to use a wide G-buffer to allow storage of more parameters. However, this consumes larger amounts of memory and puts a significant strain on the bus. It was an early decision that our game must run at 1920×1080; to achieve this, we decided to use a prepass deferred implementation, which is described in [Engel 09], with a narrow G-buffer. Our G-buffer contains a depth buffer and normal and specular power in signed R8G8B8A8 format. The sign bit of the specular power is used to indicate whether Fresnel is to be enabled or disabled on the specular reflection. The lean footprint allows us to leverage fast ESRAM on Xbox One.

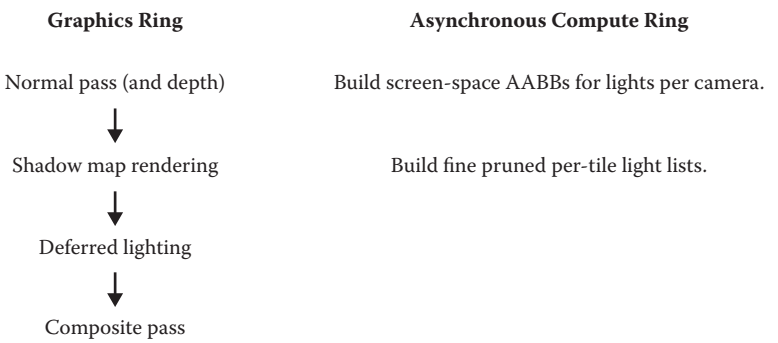

Figure 2.3. Our primary stages running on the main command buffer (left) and our asynchronous compute work (right). Generation of AABBs is interleaved with the normal-depth pass, and generation of fine pruned per-tile light lists is interleaved with rendering of the shadow maps.

The stages of our prepass rendering pipeline are shown on the left side of Figure 2.3. The geometry is rendered to the screen twice. The first time is the normal-depth pass that creates the G-buffer as depth, with world-space normal and specular power. The second time the geometry is rendered to the screen is the composite pass, which is the last stage. This stage does the shading and folds in the lighting. Rendering of shadow maps comes after the normal-depth pass, while generation of per-tile light lists is scheduled to run at the same time as an asynchronous compute job. The next stage is deferred lighting, which runs as a full-screen pass. Each pixel is lit by accumulating contributions from lights stored in the list associated with the tile to which the pixel belongs. We write the final diffuse and specular results to separate render targets, which allows us to modulate these by different textures during the final composite pass.

To achieve custom lighting on certain materials such as eyes, skin, and cloth, we use tiled forward lighting. In this case the lighting is done in-process during the composite pass by pulling and processing the light list for the tile similar to how we do this during deferred lighting. This presents a problem since we will pay the cost of lighting the pixel both deferred and forward. To solve this problem, we mark every pixel in the stencil buffer that is lit as tiled forward. During deferred lighting, we skip such pixels by using stencil testing.

In regards to the format of the per-tile light list, it can be stored in a number of ways. The obvious option is one buffer for the whole screen where each tile consumes some fixed number of 8-bit or 16-bit entries for the light indices. Using 8 bits will only allow for 256 lights on screen, and 16 bits give more range than we need. To achieve a more compact footprint, we chose to store the list as blocks of `R10G10B10A2_UINT`, where the 10-bit components each store an index to a light and the 2-bit component tells us how many of the three indices are active. We store

eight such blocks per tile, which results in a final limit of 24 lights per tile after fine pruning. As previously mentioned, we allow up to 64 lights in the coarse list while in LDS. The total footprint for eight such blocks is 32 bytes per tile and thus 1 bit per pixel on screen. Note that 10-bit indices indicate a limit of 1024 lights intersecting the camera frustum per frame.

In our implementation we use separate light lists for direct lights and probe lights, each with a limit of 24 per tile. It is important to note that the light list generation is executed *once only*. This is possible since up to 64 fine pruned lights may exist temporarily on the compute side during the execution of Step 2(e). Subsequently, in this step we separate these in LDS according to their designated light list.

As mentioned in the introduction, it is common for a tiled lighting implementation to partition the tile farther along depth into cells, as is done in [Olsson et al. 12]. This grows the footprint further because each cell stores a separate list of lights. A different problem with this strategy during deferred lighting is that each thread may pull the list of lights from a different cell than other threads in the wavefront. This forces us to pull the attributes of the lights into vector registers instead of scalar registers, which as mentioned in the previous section reduces our ability to populate more wavefronts per CU, which reduces our ability to hide latency on the GPU. Ultimately, we found that the act of fine pruning our lists of lights removes most redundancies in practice, which negates the need for partitioning into cells. This is also indicated in Figure 2.2 and evident from the heat map in the next section.

One limitation when using our method is that the generated lists only work for opaque surfaces that write to a depth buffer. In our case the majority of transparencies are particle effects with many stacked layers occupying the same local space. We concluded that we could not afford to light these per pixel as it would be too costly, and we decided to light these using vertex lighting.

For regular mesh-based transparencies, we decided to use traditional forward lighting where light lists are built on the CPU for each mesh based on a bounding volume intersection. Since our transparent surfaces are sorted on a per-material basis, these are not large meshes and thus do not benefit as much from tiled light lists. Additionally, we support *light groups*, which allow artists to manually remove specific lights from the light lists of traditionally forward-lit objects. This feature allows them to prune the list to the most essential set of lights that intersect the transparent surface.

2.6 Results

In this section we show an interior scene running at 1920×1080 with and without fine pruning. Figure 2.4 shows the results of coarse culling. The coarse list generation takes 0.5 ms and runs asynchronously. Figure 2.5 shows the results

Figure 2.4. Number of lights per tile after coarse culling.

after fine pruning, which costs 1.7 ms in list generation. The cost is however well hidden because of asynchronous compute. The heat map in Figure 2.6 indicates the occupancy of lights per tile in Figures 2.4 and 2.5. We can see that the light counts without fine pruning are significantly higher in almost every tile. As expected, we see a significant drop in execution time of deferred lighting, dropping from 5.4 ms to 1.4 ms with fine pruning enabled.

2.7 Conclusion

We have demonstrated a new tiled lighting variant that performs light list generation per tile in two steps within the same compute kernel. The initial coarse pass generates a light list in local storage based on simple screen-space AABB bounding volume intersection testing regardless of light type. The second step is fine pruning, which performs further testing on the coarse list by testing each pixel in the tile if the corresponding point in 3D space is inside the true shape of the light source. Lights that contain one or more such points are put in the fine pruned list, which is written to memory. We have found that in practice this process reduces the light count per tile significantly.

On the AMD GCN architecture a depth-only pass of opaque meshes generates very little work for the GPU cores. We take advantage of this fact by using asynchronous compute to hide most of the combined cost of the coarse and the fine pruning steps by interleaving this work with the rendering of shadow maps, which gives no redundancy light lists for free.

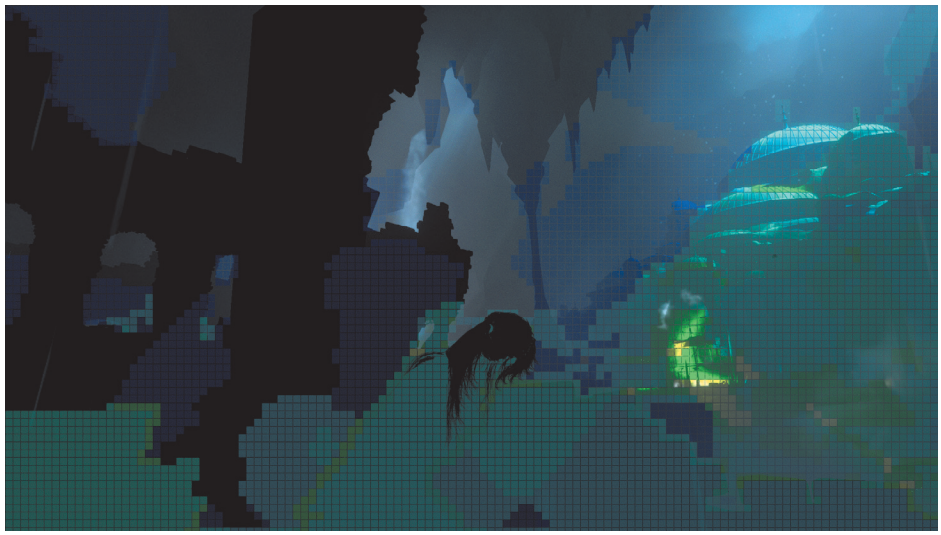

Figure 2.5. Number of lights per tile after fine pruning.

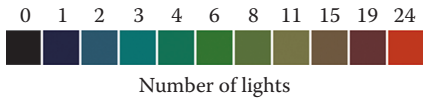

Figure 2.6. Color heatmap with number of lights per tile.

While supporting multiple light types, the final footprint for the light list is 1 bit per pixel with a maximum number of 24 fine pruned lights per tile.

Finally, an efficient hybrid between tiled deferred and tiled forward is presented where tiled deferred lighting is done as a stencil tested full-screen pass to avoid lighting twice for pixels that are lit by tiled forward materials. To further accelerate tiled forward, we keep the light list sorted by type in a fixed order. This allows us to maximize the chance that all pixels in a wavefront are processing lights in the same light loop.

2.8 Acknowledgments

Thank you to editor Michal Valient for his support in the development of this chapter and for his reviews. Additionally, thank you to Manchor Ko and Kasper H. Nielsen for proofreading the chapter. I would also like to thank Paul Houx at Nixxes for his excellent work and collaboration on integrating this method into the foundation engine at Crystal Dynamics. And finally, thanks go to Scott Krotz for his support and help getting asynchronous compute to work properly.

Bibliography

[Bezrati 14] Abdul Bezrati. "Real-Time Lighting via Light Linked List." Paper presented at SIGGRAPH, Vancouver, Canada, August 12–14, 2014.

[Engel 09] Wolfgang Engel. "The Light Pre-Pass Renderer: Renderer Design for Efficient Support of Multiple Lights." SIGGRAPH Course: Advances in Real-Time Rendering in 3D Graphics and Games, New Orleans, LA, August 3, 2009.

[Harada et al. 12] Takahiro Harada, Jay McKee, and Jason C. Yang. "Forward+: Bringing Deferred Lighting to the Next Level." Eurographics Short Paper, Cagliari, Italy, May 13–18, 2012.

[Olsson et al. 12] Ola Olsson, Markus Billeter, and Ulf Assarsson. "Clustered Deferred and Forward Shading." Paper presented at High Performance Graphics, Paris, France, June 25–27, 2012.