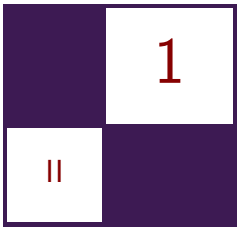


Clustered Shading: Assigning Lights Using Conservative Rasterization in DirectX 12

Kevin Örtegren and Emil Persson

1.1 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. 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 article presents a GPU-based variation of *practical clustered shading* [Persson and Olsson 13], which is a technique that improves on the currently popular *tiled shading* [Olsson and Assarsson 11, Swoboda 09, Balestra and Engstad 08, Andersson 09] by utilizing higher-dimensional tiles. The view frustum is divided into three-dimensional clusters instead of two-dimensional tiles and addresses the depth discontinuity problem present in the tiled shading technique. The main goal we aimed for was to explore the use of conservative rasterization to efficiently assign convex light shapes to clusters.

Clustered shading is a technique similar to tiled shading that performs a light culling step before the lighting stage when rendering a scene. The view frustum is divided into sub-frustums, which we call *clusters*, in three dimensions. The purpose of the light culling step is to insert all visible lights into the clusters that they intersect. When the light culling is done, the clusters contain information of which lights intersect them. It is then easy to fetch the light data from a cluster when shading a pixel by using the pixel's view-space position. The goal of the technique is to minimize the number of lighting calculations per pixel and to address some of the problems present in tiled shading. Tiled shading uses two-dimensional tiles and relies on a depth prepass to reduce the tiles in the z-dimension, whereas clustered shading has a fixed cluster structure in view space at all times.

(a) Conservative rasterization. (b) Normal rasterization.

Figure 1.1. Difference between rasterization modes. Red cells represent the pixel shader invocations for the triangle.

Previous work on clustered shading first surfaced in 2012 [Olsson et al. 12] and have since spawned a few presentations and demos on the subject: Intel demo on forward clustered shading [Fauconneau 14], GDC15 presentation from AMD on tiled and clustered shading [Thomas 15], and a practical solution to clustered shading from Avalanche [Persson and Olsson 13]. As of writing this, there is one released game using clustered shading, namely *Forza Horizon 2* [Leadbetter 14].

1.2 Conservative Rasterization

The use of the rasterizer has traditionally been to generate pixels from primitives for drawing to the screen, but with programmable shaders there is nothing stopping the user from using it in other ways. The normal rasterization mode will rasterize a pixel if the pixel center is covered by a primitive. *Conservative rasterization* is an alternative rasterization mode where if any part of a primitive overlaps a pixel, that pixel is considered covered and is then rasterized. The difference between these modes is illustrated in Figure 1.1.

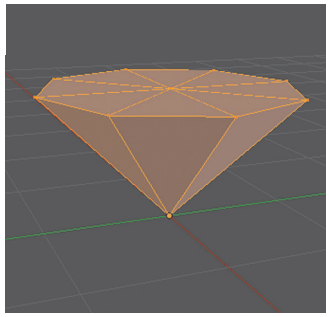
1.3 Implementation

This section will go through the different steps included in the light assignment algorithm as well as explain how the main data structure used for storing light and cluster data is created and managed, as it is an intricate part of the technique. An overview of the algorithm is listed below:

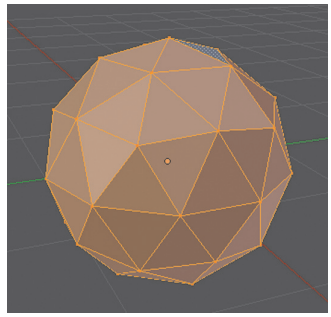
For each light type:

1. Shell pass: Find minimum and maximum depths in every tile for every light.
2. Fill pass: Use the minimum and maximum depths and fill indices into the light linked list.

The light assignment is complete when all light types have been processed and the light linked list can be used when shading geometry.



(a) A unit cone mesh with 10 vertices.



(b) A unit sphere mesh with 42 vertices.

Figure 1.2. Two example unit shapes created in Blender.

1.3.1 Light Shape Representation

Lights must have a shape representation to be able to be inserted into clusters. Approximating every light shape as an analytical sphere is the easiest and computationally cheapest approach, but it will be inaccurate for light shapes that are not sphere shaped. An analytic shape representation is suitable when performing general intersection calculations on the CPU or in, for example, a compute shader. Some shapes will, however, have a very complex analytical representation, which is why many techniques resort to using spheres.

The technique presented here uses the rasterizer and the traditional rendering shader pipeline, which is well suited to deal with high amounts of vertices. Shapes represented as vertex meshes are very simple and provide general representation models for all light shapes. The level of flexibility when working with vertex meshes is very high because the meshes can be created with variable detail.

Meshes are created as unit shapes, where vertices are constrained to -1 to 1 in the x-, y-, and z-directions. This is done to allow arbitrary scaling of the shape depending on the actual light size. Some light shapes may need to be altered at runtime to allow for more precise representations: for example, the unit cone will fit around a sphere-capped cone for a spot light, and thus the cap must be calculated in the vertex shader before light assignment. In the case of using low amounts of vertices for light shapes, the shapes could easily be created in code and also use very small vertex formats: for example, R8G8B8 is enough for the shapes in Figure 1.2.

1.3.2 Shell Pass

The *shell pass* is responsible for finding the clusters for a light shape that encompasses it in cluster space. The pass finds the near and far clusters for each tile for each light and stores them in an R8G8 render target for the following

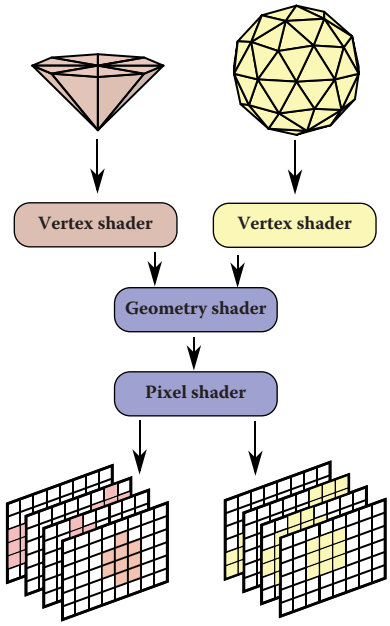


Figure 1.3. Illustration of the entire shell pass.

pass to fill the shell. The number of render targets for the shell pass correspond to the maximum number of visible lights for each light type. All render targets have the same size and format and are set up in a `Texture2DArray` for each light type. The sizes of the render targets are the same as the x- and y-dimensions of the cluster structure, otherwise known as the *tile dimension*. An overview illustration of the shell pass can be seen in Figure 1.3. The shell pass uses the traditional shader pipeline with conservative rasterization to ensure that the light meshes invoke all the pixels they touch. To activate conservative rasterization in DirectX 12, it is simply a matter of setting the `ConservativeRaster` flag to `D3D12_CONSERVATIVE_RASTERIZATION_MODE_ON` when creating a pipeline state object for the shader pipeline.

Vertex shader Each light type has its own custom vertex shader for translating, rotating, and scaling the light mesh to fit the actual light. This and the mesh are the only two things that have to be introduced when adding a new light type for the light assignment. The algorithm starts by issuing a `DrawIndexedInstanced` with the number of lights as the instance count. Also fed to the vertex shader is the actual light data containing position, color, and other light properties. The shader semantic `SV_InstanceID` is used in the vertex shader to extract the position, scale, and other properties to transform each vertex to the correct location in world space. Each vertex is sent to the geometry shader containing the view-

space position and its light ID, which is the same as the previously mentioned `SV_InstanceID`.

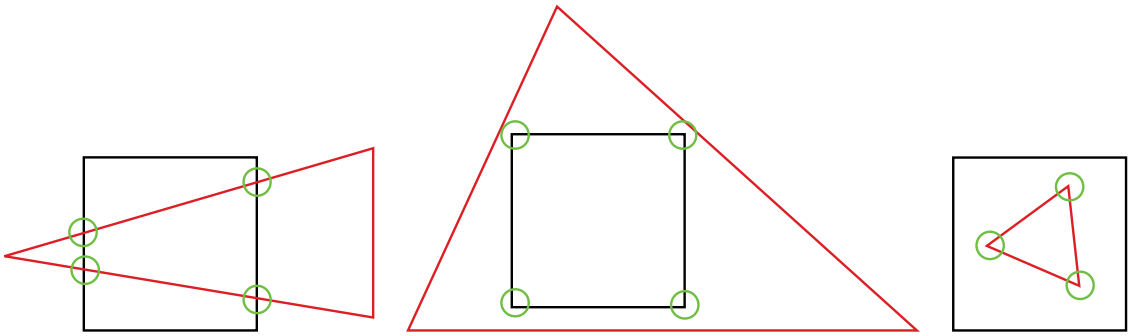
Geometry shader The vertices will simply pass through the geometry shader where packed view positions for each vertex in the triangle primitive are appended to every vertex. The vertex view positions are flagged with `nointerpolation` as they have to remain correctly in the view space through the rasterizer. The most important task of the geometry shader is to select the correct render target as output for the pixel shader. This is done by writing a render target index to the `SV_RenderTargetArrayIndex` semantic in each vertex. `SV_RenderTargetArrayIndex` is only available through the geometry shader; this is a restriction of the current shading model and makes the use of the geometry shader a requirement. The geometry shader is unfortunately not an optimal path to take in the shader pipeline because it, besides selecting the render target index, adds unnecessary overhead.

Pixel shader The pixel shader performs most of the mathematics and does so for every triangle in every tile. Each pixel shader invocation corresponds to a tile, and in that tile the nearest or farthest cluster must be calculated and written for every light. When a pixel shader is run for a tile, it means that part of a triangle from a light shape mesh is inside that tile, and from that triangle part the minimum and maximum depths must be found. Depth can be directly translated into a Z-cluster using a depth distribution function, which is discussed in more detail in the next section.

All calculations are performed in view space because vertices outside a tile must be correctly represented; if calculations were performed in screen space, the vertices behind the near plane would be incorrectly transformed and become unusable. Tile boundaries are represented as four side planes that go through the camera origin $(0, 0, 0)$. Each pixel shader invocation handles one triangle at a time. To find the minimum and maximum depths for a triangle in a tile, three cases are used; see Figure 1.4. The three points that can be the minimum or maximum depths in a tile are as follows:

(a) Where a vertex edge intersects the tile boundary planes: Listing 1.1 shows the intersection function for finding the intersection distance from a vertex to a tile boundary plane. The distance is along the edge from vertex `p0`. Note that both `N` and `D` can be 0, in which case `N / D` would return `NaN` or, in the case of only `D` being 0, would return `+/-INF`. It is an optimization to not check for these cases, as the IEEE 754-2008 floating point specification in HLSL [Microsoft] states that

1. the comparison `NE`, when either or both operands is `NaN`, returns `TRUE`;
2. comparisons of any non-`NaN` value against `+/-INF` return the correct result.



(a) Edges. (b) Corners. (c) Vertices.

Figure 1.4. The three cases of finding minimum and maximum depths on a triangle in a tile.

```
bool linesegment_vs_plane(float3 p0, float3 p1, float3 pn,out
                          float lerp_val)
{
    float3 u = p1 - p0;

    float D = dot(pn, u);
    float N = -dot(pn, p0);

    lerp_val = N / D;
    return !(lerp_val != saturate(lerp_val));
}
```

Listing 1.1. Vertex edge versus tile boundary plane intersection.

The second rule applies to the intrinsic function saturate. These two rules make sure that the function always returns the correct boolean.

(b) Where a triangle covers a tile corner: Finding the depth at a corner of a tile is simply a matter of performing four ray-versus-triangle intersections, one at each corner of the tile. The ray–triangle intersection function in Listing 1.2 is derived from [Möller and Trumbore 05].

(c) Where a vertex is completely inside a tile: The signed distance from a point to a plane in three dimensions is calculated by

$$D = \frac{ax_1 + by_1 + cz_1 + d}{\sqrt{a^2 + b^2 + c^2}},$$

where (a, b, c) is the normal vector of the plane and (x_1, y_1, z_1) is the point to which the distance is calculated. The variable d is defined as $d = -ax_0 -$

```
bool ray_vs_triangle(float3 ray_dir, float3 vert0, float3 vert1,
                     float3 vert2, out float z_pos)
{
  float3 e1 = vert1 - vert0;
  float3 e2 = vert2 - vert0;
  float3 q = cross(ray_dir, e2);
  float a = dot(e1, q);

  if(a > -0.000001f && a < 0.000001f)
    return false;

  float f = 1.0f / a;
  float u = f * dot(-vert0, q);

  if(u != saturate(u))
    return false;

  float3 r = cross(-vert0, e1);
  float v = f * dot(ray_dir, r);

  if(v < 0.0f || (u + v) > 1.0f)
    return false;

  z_pos = f * dot(e2, r) * ray_dir.z;

  return true;
}
```

Listing 1.2. Ray versus triangle intersection.

$by_0 - cz_0$, where (x_0, y_0, z_0) is a point on the plane. As all planes go through the origin in the view space, the variable d is eliminated; because the plane normals are length 1, the denominator is also eliminated. This leaves the function as $D = ax_1 + by_1 + cz_1$. Further simplification can be done by splitting the function into two separate functions: one for testing the side planes and one for testing the top and bottom planes. These functions are $D = ax_1 + cz_1$ and $D = by_1 + cz_1$, respectively, as the y-component of the plane normal is zero in the first case and the x-component is zero in the second case. By knowing the direction of the plane normals, the sign of the distance tells on which side of the plane the vertex is. See Listing 1.3 for HLSL code of these two functions.

When all three cases have been evaluated, the minimum and maximum depths for a tile have been determined and the result can be stored. The result is stored in a render target with the same size as the x- and y-dimensions of the cluster structure. When a triangle is run through a pixel shader, it can be either front facing or back facing. In the case of a triangle being front facing, the minimum depth will be stored, and in the back facing case, the maximum depth will be stored.

To save video memory, the depth values are first converted into Z-cluster space, which is what is used in the following pass. The render target uses the

```
bool is_in_xslice(float3 top_plane, float3 bottom_plane,
                  float3 vert_point)
{
  return (top_plane.y * vert_point.y + top_plane.z * vert_point.z
            >= 0.0f && bottom_plane.y * vert_point.y +
            bottom_plane.z * vert_point.z >= 0.0f);
}

bool is_in_yslice(float3 left_plane, float3 right_plane,
                  float3 vert_point)
{
  return (left_plane.x * vert_point.x + left_plane.z * vert_point↩
      .z
            >= 0.0f && right_plane.x * vert_point.x +
            right_plane.z * vert_point.z >= 0.0f );
}
```

Listing 1.3. Vertex point versus tile boundary planes intersection.

format `R8G8_UNORM`, which allows for the cluster structure to have up to 256 clusters in the z-dimension. As many triangles can be in the same tile for a light shape, it is important to find the minimum and maximum Z-clusters for all the triangles. This is done by writing the result to the render target using using a `MIN` rasterizer blend mode, which ensures that the smallest result is stored. To be able to use the same shader and the same blend mode for both front-facing and back-facing triangles, the HLSL system value `SV_IsFrontFace` is used to select in which color channel the result is stored. In the case of back-facing triangles, the result must be inverted to correctly blend using the MIN blend mode; the result is then inverted again in the next pass to retrieve the correct value. Figure 1.5 illustrates the found minimum and maximum depth points in a tile for a point light shape. A top-down illustration of the final result of the shell pass can be seen in Figure 1.6, where two point lights and a spot light have been processed, with the colored clusters representing the minimum and maximum Z-clusters for each tile and light.

1.3.3 Depth Distribution

The *depth distribution* determines how the Z-cluster planes are distributed along the z-axis in the view space. The depth distribution is represented as a function that takes a linear depth value as input and outputs the corresponding Z-cluster. Two functions have been evaluated in this implementation; one linear and one exponential. The linear distribution simply divides the z-axis into equally spaced slices while the exponential function is

$$Z = \log_2(d)\frac{1}{\log_2(f) - \log_2(n)}(c-1) + \left((1 - \log_2(n))\frac{1}{\log_2(f) - \log_2(n)}(c-1)\right),$$

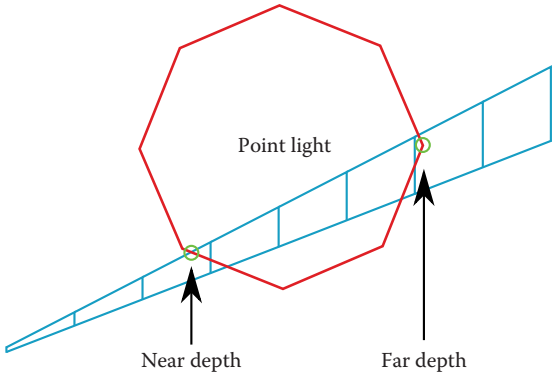


Figure 1.5. Top-down view of one tile and the found minimum and maximum depths for a point light mesh.

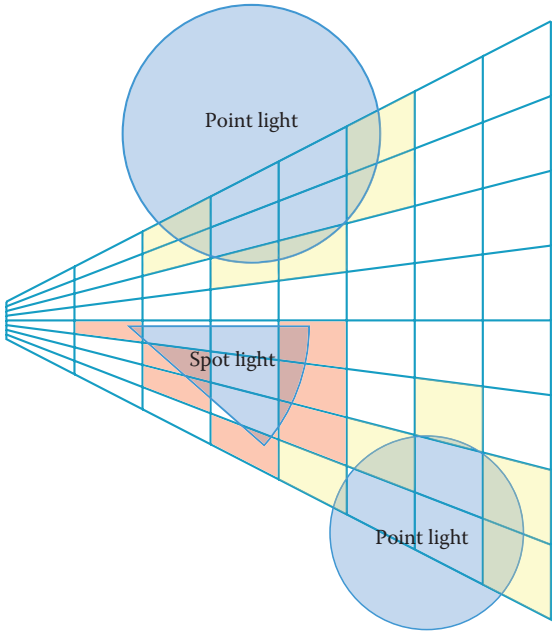


Figure 1.6. Two-dimensional top-down view of a shell pass.

where d is the view-space distance along the z-axis, f is the distance to the last z-plane, n is the distance to the second z-plane, and c is the number of clusters in the z-dimension. Note that most of these are constants and are not recalculated. Figure 1.7 shows the two functions in a graph with example values.

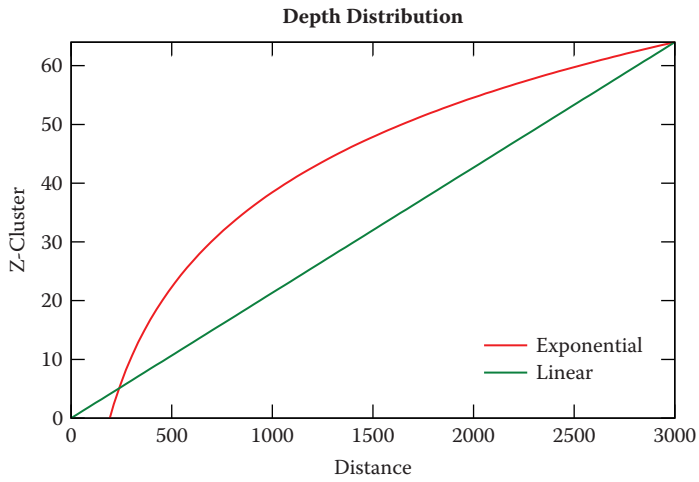


Figure 1.7. Graph of two distribution functions over an example depth of 3000 with 64 clusters in the z-dimension. The second z-slice of the exponential function is set to start at 200.

1.3.4 Fill Pass

The *fill pass* is a compute-shader-only pass with one purpose: to write the assigned lights into the light linked list, which is a linked list on the GPU derived from [Yang et al. 10].

Light linked list A light linked list is a GPU-friendly data structure for storing and managing many index pointers to larger data. In the case of this algorithm, a fixed number of unique lights are active each frame, and hundreds of clusters can contain the same instance of a light. It would be wasteful to store the actual light data (position, color, etc.) in every cluster; instead, an index to the light data is stored. Light data can differ between light types and implementation, but in most cases they are larger than 64 bit, which is the size of the light linked list node. More specifically, the light linked list node contains three pieces of data: the pointer to the next node in the list, the pointer to the actual light data, and the light type. These can fit into either 64 bits or 32 bits, depending on the maximum amount of lights needed in the game. Examples of the data in a node are shown in Table 1.1. The 64-bit node has support for more lights than modern hardware can manage in real time, but the 32-bit node is at the limit of what could be viable in a modern game engine. A tradeoff has to be made between memory savings and the maximum number of supported lights. Note that in Table 1.1 the 32-bit node uses 2 bits for the light type and 10 bits for the light ID, which results in 4096 total lights. This can be switched around to whatever

(a)

Data	Size	Max. Value
Light type	8 bits	256
LightID	24 bits	16777216
Link	32 bits	4294967296

(b)

Data	Size	Max. Value
Light type	2 bits	4
LightID	10 bits	1024
Link	20 bits	1048576

Table 1.1. Examples of (a) 64-bit and (b) 32-bit node layouts.

fits the implementation best; for example, if only point lights and spot lights are used, the light type would only need 1 bit.

The data structures used to build the light linked list consists of three parts and can be seen in Figure 1.8. The start offset buffer is a Direct3D `ByteAddressBuffer` with cells corresponding to each cluster. The elements are `uint32` and act as pointers into the linked node light list. Each cell in the start offset buffer points to the head node for a cluster. Simply following the head node in the linked list will go through all nodes for a given cluster. The light linked list is a large one-dimensional structured buffer containing the previously mentioned nodes. Each used node points to actual light data that can be fetched and used for shading.

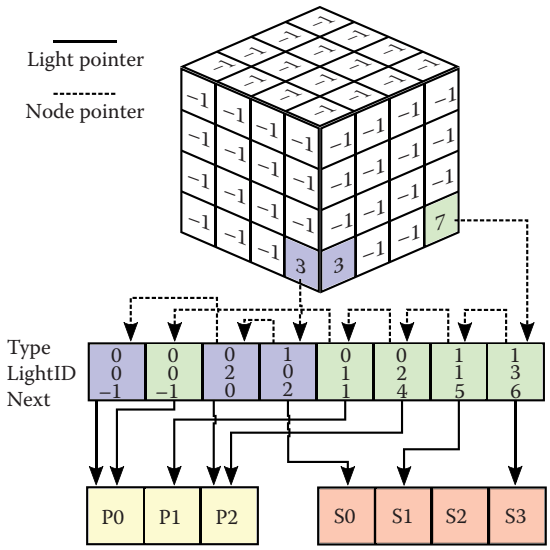


Figure 1.8. Illustration of the light linked list. The "Next" field is the index to the next node in the linked list; if a node is pointed to and has −1 as its next node, it means that it is the tail node and no more nodes are linked to that sequence. The three-dimensional structure contains a pointer from each cluster to the head node for that cluster. If a cluster is empty, there will be −1 in the corresponding cell. The types can be chosen per implementation, and in this case 0 stands for point lights and 1 stands for spot lights. For example, the cluster that points to node 7 touches lights P0, P1, P2, S1, and S3.

```
//This array has NUM_LIGHTS slices and contains the near and far
//Z-clusters for each tile.
Texture2DArray<float2> conservativeRTs : register(t0);

//Linked list of light IDs.
RWByteAddressBuffer StartOffsetBuffer : register(u0);
RWStructuredBuffer<LinkedLightID> LinkedLightList : register(u1);

[numthreads(TILESX, TILESY, 1)]
void main( uint3 thread_ID : SV_DispatchThreadID ){
  //Load near and far values (x is near and y is far).
  float2 near_and_far = conservativeRTs.Load(int4(thread_ID, 0));

  if(near_and_far.x == 1.0f && near_and_far.y == 1.0f)
    return;

  //Unpack to Z-cluster space([0,1] to [0,255]). Also handle
    //cases where no near or far clusters were written.
  uint near = (near_and_far.x == 1.0f) ? 0 :
        uint(near_and_far.x * 255.0f + 0.5f);
  uint far  = (near_and_far.y == 1.0f) ? (CLUSTERSZ - 1) :
        uint(((CLUSTERSZ - 1.0f) / 255.0f - near_and_far.y)
             * 255.0f + 0.5f);

  //Loop through near to far and fill the light linked list.
  uint offset_index_base = 4 * (thread_ID.x + CLUSTERSX *
                                thread_ID.y);
  uint offset_index_step = 4 * CLUSTERSX * CLUSTERSY;
  uint type = light_type;
  for(uint i = near; i <= far; ++i){
    uint index_count = LinkedLightList.IncrementCounter();
    uint start_offset_address = offset_index_base
                                + offset_index_step * i;

    uint prev_offset;
    StartOffsetBuffer.InterlockedExchange(start_offset_address,
                                          index_count, prev_offset);

    LinkedLightID linked_node;
    linked_node.lightID = (type << 24) | (thread_ID.z & 0xFFFFFF);
            //Light type is encoded in the last 8bit of the
            //node.lightID and lightID in the first 24bits.
    linked_node.link = prev_offset;

    LinkedLightList[index_count] = linked_node;
  }
}
```

Listing 1.4. The complete compute shader for the fill pass.

The last part is the actual light data storage that can be set up in multiple ways as long as it can be indexed using a `uint32`. In this implementation, the light data is stored in structured buffers. The complete compute shader is outlined in Listing 1.4.

When the fill pass is complete, the linked light list contains all information necessary to shade any geometry in the scene. An example of a completely assigned cluster structure is illustrated in Figure 1.9.

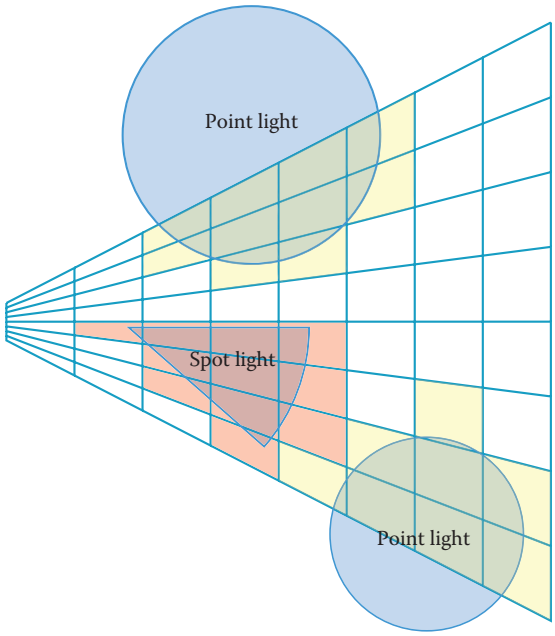


Figure 1.9. Two-dimensional top-down view of a fill pass.

1.4 Shading

The shading is done in the pixel shader by calculating in which cluster the pixel lies and getting the lights from that cluster. As the light types are stored sequentially in the light linked list, it is easy to loop through all lights without having to perform expensive branching. The pixel shader code is listed in Listing 1.5.

Finding out from which cluster the pixel should pull the lights is done by translating the screen-space x- and y-positions of the pixel into the cluster's x- and y-spaces. If the tile pixel size is a power of two, this can be done by a bit shift operation rather than using division. Finding the z-position of the cluster requires a depth value for the pixel, which could be sampled from a depth buffer in the case of deferred shading or could be the z-position of the interpolated geometry in the case of forward shading. The sampled depth is then translated into the Z-cluster space by applying the same depth distribution function used in the shell pass. Figure 1.10 shows what clusters are used for shading in an example scene using the assigned lights from Figure 1.9.

Each light type has its own `while` loop, and the `while` loops are in the reversed order from how the light types were assigned due to the the light linked list having its head pointing at the end of the linked sequence. For example, if point lights are assigned before spot lights, the spot lights will be before the point lights in

```
uint light_index = start_offset_buffer[clusterPos.x + CLUSTERSX *↩
     clusterPos.y + CLUSTERSX * CLUSTERSY * zcluster];

  float3 outColor = float3(0,0,0);

  LinkedLightID linked_light;

  if(light_index != 0xFFFFFFFF)
  {
    linked_light = light_linked_list[light_index];

    //Spot light
    while((linked_light.lightID >> 24) == 1)
    {
      uint lightID = (linked_light.lightID & 0xFFFFFF);

      outColor += SpotLightCalc(pos, norm, diff, spotLights[↩
          lightID]);

      light_index = linked_light.link;

      if(light_index == 0xFFFFFFFF)
        break;

      linked_light = light_linked_list[light_index];
    }

    //Point light
    while((linked_light.lightID >> 24) == 0)
    {
      uint lightID = (linked_light.lightID & 0xFFFFFF);

      outColor += PointLightCalc(pos, norm, diff, pointLights[↩
          lightID]);

      light_index = linked_light.link;

      if(light_index == 0xFFFFFFFF)
        break;

      linked_light = light_linked_list[light_index];
    }
  }

   return float4(outColor, 1.0f);
```

Listing 1.5. Pixel shader code for going through the light linked list for shading a pixel.

the linked sequence. See Figure 1.8, where the node pointer arrows show how the linked list will be traversed.

1.5 Results and Analysis

This section will show results from the performed experiments and presents an analysis of performance, memory, number of assigned clusters, and depth distri-

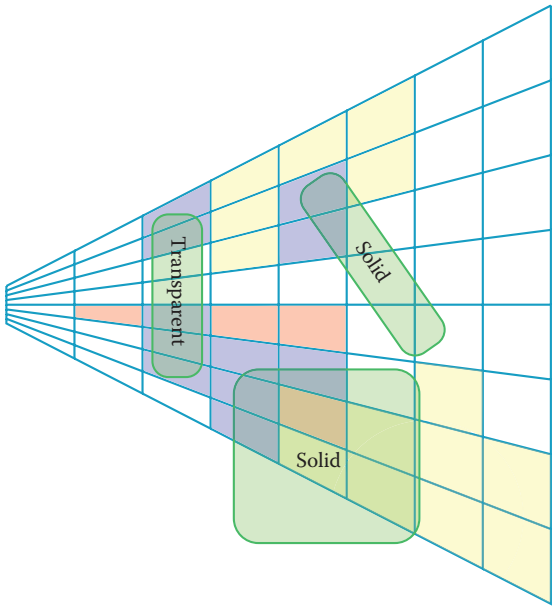


Figure 1.10. Two-dimensional top-down view of sampled clusters in a scene with objects. Note that transparent objects are shaded the same way as opaque objects. Colored clusters contain lights, and the blue clusters are used for shading the geometry.

bution in separate sections. The charts compare many different cluster structure setups, and in some of them the key legend describes the cluster structure dimensions and the depth distribution function used. The suffixes "-L" and "-E" mean linear and exponential, respectively. Performance is measured in milliseconds, and all measurements are done on the GPU.

The test scene is the CryTek Sponza Atrium with up to 4096 lights, and the test scenario is set up exactly as AMD's Forward+ demo [Harada et al. 13], which is also used as a comparison in the light assignment results. A screenshot of the test scene can be seen in Figure 1.11. All tests are performed on an NVIDIA GTX970 graphics card running DirectX 12 on Windows 10 build 10130. The resolution is 1536×768.

1.5.1 Performance

Apart from the performance inconsistencies between depth distribution functions, which are analysed in detail in Section 1.5.4, the performance results are consistent. A few observations can be made by examining Figures 1.12, 1.13 and 1.14: The shell pass remains constant in time when the x- and y-dimensions change, the fill pass increases in time when any of the three dimensions of the cluster

Figure 1.11. CryTek Sponza Atrium test scene.

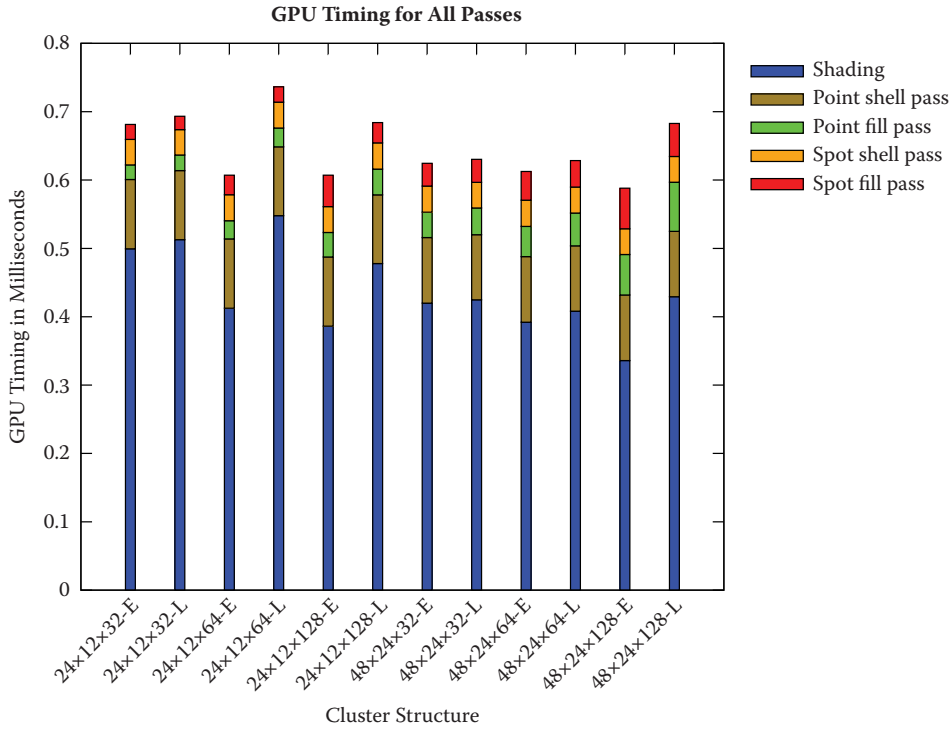


Figure 1.12. Total GPU timings in milliseconds split up into the different passes of the algorithm at 1024 lights. Lower is better.

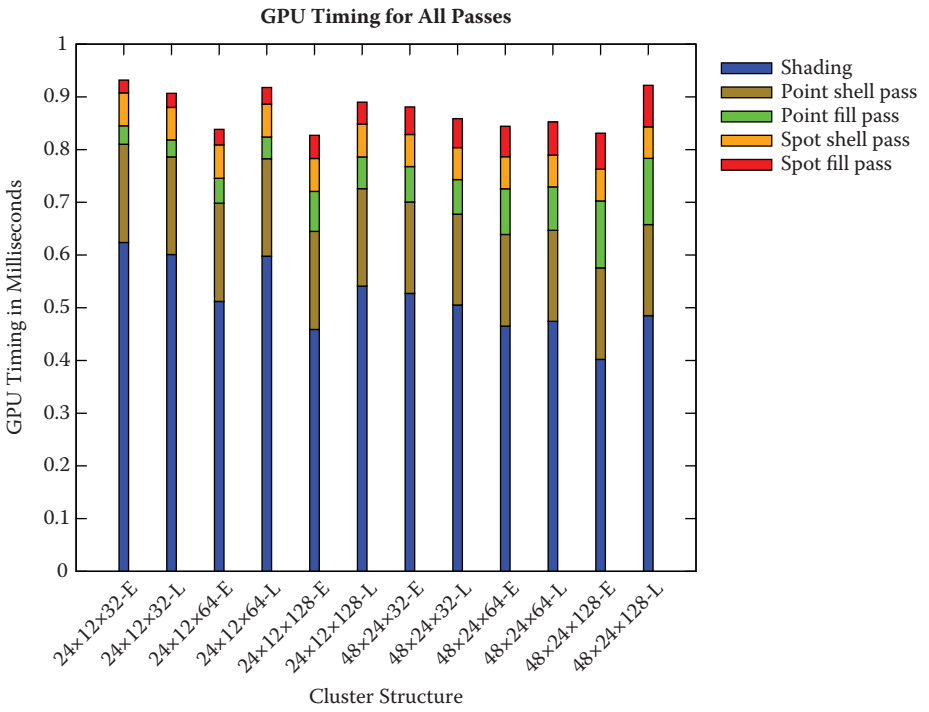


Figure 1.13. Total GPU timings in milliseconds split up into the different passes of the algorithm at 2048 lights. Lower is better.

structure increases, and the total time increases close to linearly with regards to the number of lights.

The times for the two shell passes remain constant when going from 24×12 to 48×24 tiles, but there is a significant difference between them in time. The light shape mesh vertex count used for the respective shell passes are 42 and 10, which indicates that the pixel shader is not the bottleneck. This observation is further strengthened by the fact that going from 24×12 tiles to 48×24 will yield up to four times the number of pixel shader invocations for any number of triangles, which in turn means that the constant time for the shell passes is caused by the triangle processing and data transfer being the bottleneck. Packing data for transfer between shader stages has given the best performance increases when optimizing the shaders.

The fill pass suffers from bad scaling with being up to 6.5 times slower between $24 \times 12 \times 32$ and $48 \times 24 \times 128$ at 4096 lights; see Figure 1.14. As opposed to the pixel shader in the shell pass, which uses mostly ALU instructions, the fill pass writes a lot of data to the light linked list and becomes bandwidth intensive at a

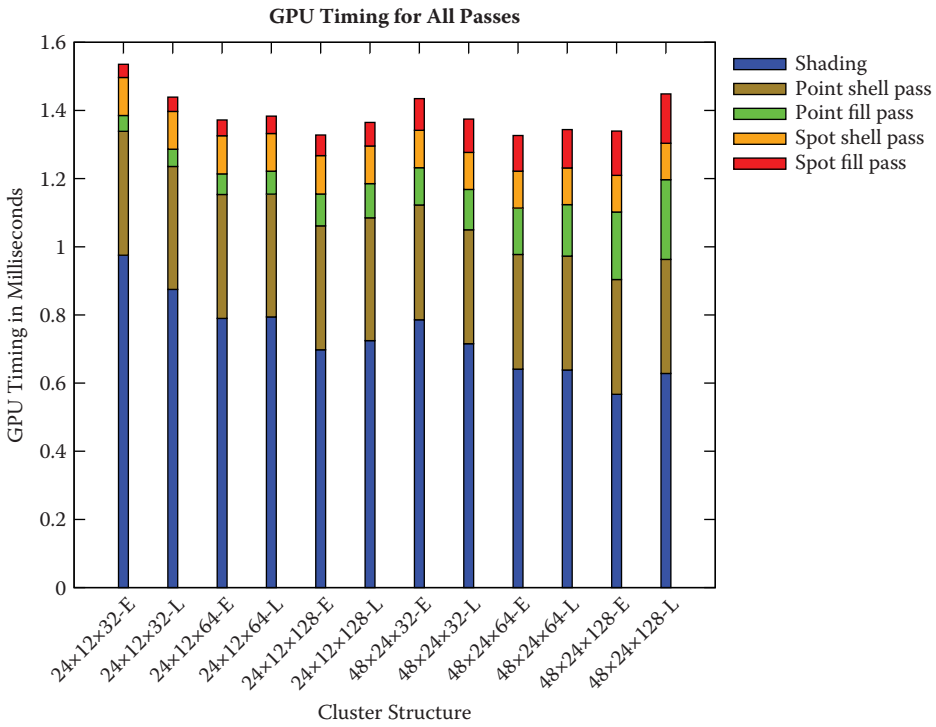


Figure 1.14. Total GPU timings in milliseconds split up into the different passes of the algorithm at 4096 lights. Lower is better.

large number of lights and clusters. The compute shader in the fill pass has low thread coherency and occupancy due to the shape of the cluster structure: lights close to the camera fill up most of their render targets while lights far away from the camera only fill a minimal part the their render targets. The compute shader will invoke threads for all texels, where empty texels cause an early exit for a thread. When using exponential depth, the lights close to the camera will be assigned to a large majority of the clusters. The shape and size of the lights also directly affects the thread coherency of the compute shader as lights that cover many clusters in the z-dimension will write more data as each thread writes data from the near to far clusters in each tile. This is also why the largest relative increases in time occur when adding more slices to the cluster structure. On top of those general observations, all the data writing is done by using atomic functions, which limits the level of parallel efficiency of the compute shader. The spot light fill pass goes from being one of the cheapest passes at a low cluster count to one of the most expensive passes at a high cluster count. The reason for having the fill

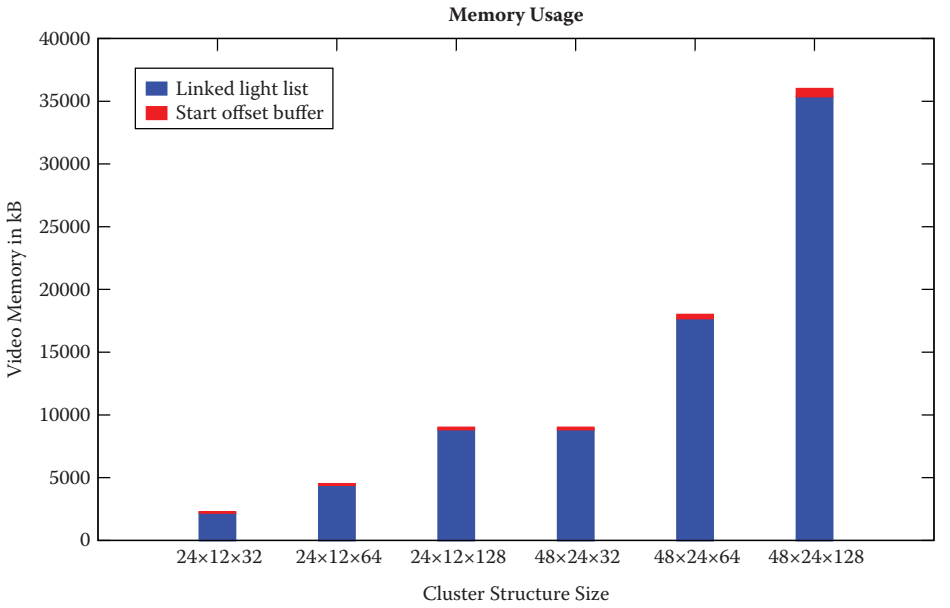
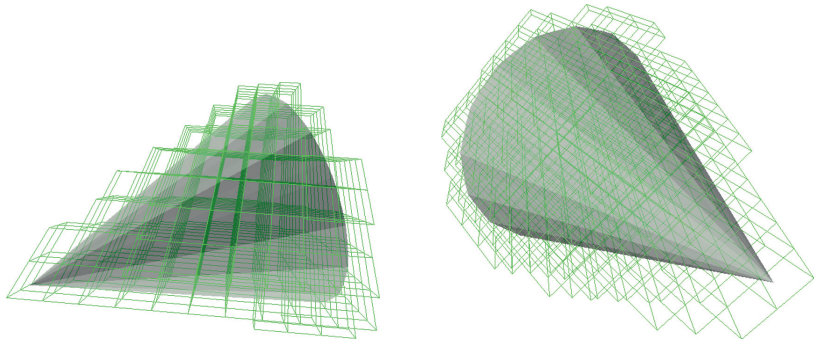


Figure 1.15. Video memory used by the cluster structure and light linked list at 4096 lights. Lower is better.

pass is because of the choice of data structure, the light linked list. The fill pass is decoupled from the shell pass and can be replaced by something else if another data structure is desired, this adds to the flexibility of the algorithm and could be a possible optimization. Another performance optimization possibility is to use fixed-size arrays for each cluster, but this will severely limit the number of lights as it would significantly increase the needed memory to store light pointers.

1.5.2 Memory

The memory model of this implementation is simple; it consists of the light linked list and the render targets for the lights. Figure 1.15 shows the memory used by the linked light list for the tested cluster structure sizes with 64-bit list nodes. The start offset buffer is always `numberOfClusters * 4` bytes large, and the light linked list is initialized to a safe size because it works like a pool of light pointers. In this case, the light linked list is `numberOfClusters * 8 * 30` bytes large; 30 is an arbitrarily chosen multiplier that provides a safe list size for this particular scenario. If the list size is not large enough, there will be lights missing at shading time. The missing lights will be noticeable: a light pointer could be missing from one cluster and correctly assigned to a neighbouring cluster, creating a hard edge at the tile border. Visually, missing light assignments will show up as darker



(a) (b)

Figure 1.16. Clusters that were assigned to a spot light are visualized and viewed from two different angles. Perfect clustering with exponential depth distribution was captured from a medium distance at a clustered structure size of $24 \times 12 \times 64$.

blocks in the final shaded image. As can be seen in Figure 1.15, the actual linked list is a large majority of the memory usage at 4096 lights. Using a 32-bit node would only use half the memory of the linked list, but as previously shown in Table 1.1, only 1048576 linked nodes would fit at a 20-bit link size, which would limit the maximum cluster structure size depending on the concentration of lights in a scene.

The render target memory usage is not dependent on the cluster structure slice depth; it is dependent on the number of lights and the number of tiles. Each light needs `numberOfTiles * 2` bytes, and at 4096 lights with 24×12 tiles, this adds up to 2,359,296 bytes.

If memory is an issue, there is the alternative to use a 32-bit node in the light linked list and choosing an appropriate cluster structure size. Comparing the $24 \times 12 \times 128$ structure with 32-bit nodes to the $48 \times 24 \times 32$ structure with 64-bit nodes results in 6.87 MB and 18.2 MB, respectively. In this implementation, the $24 \times 12 \times 128$ structure even achieves better shading and light assignment times. This goes to show that knowing the use case of the application and choosing the right setup for this technique is important.

1.5.3 Light Assignment

Figure 1.16 shows a perfectly clustered spot light and how it fits in the cluster structure. Perfect clustering refers to the fact that a light shape is never assigned to clusters it does not intersect. Even with perfect clustering the shading pass will perform some unnecessary shading calculations due to parts of the clusters not being covered by the shape, as can be seen in the Figure 1.16. Smaller clusters will give less empty space for an assigned shape and give better shading times.

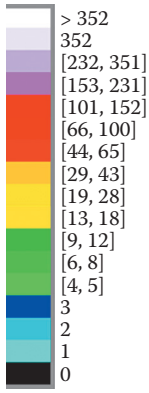
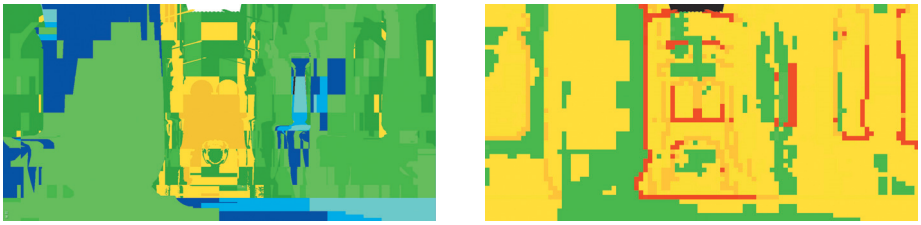


Figure 1.17. Weather radar colors corresponding to the number of lighting calculations.



(a) Clustered shading using 24 × 12 × 128-E cluster structure.

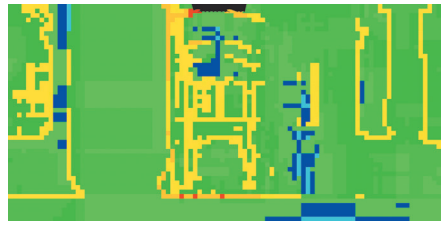
(b) Tiled shading using 96 × 48 tiled structure.

Figure 1.18. Comparison between AMD's Forward+ tiled light culling demo using 2048 point lights and 2048 spot lights. Legend can be viewed in Figure 1.17.

The results from comparing AMD's Forward+ tiled light culling with the 24 × 12×128-E cluster structure (following the legend in Figure 1.17) are demonstrated in Figures 1.18, 1.19, and 1.20. The colors correspond to the number of lighting calculations, where lower is better. AMD's tiled light culling implementations uses 96 × 48 tiles, using 6488064 bytes video memory and performing the light assignment in 0.6 ms on average. The 24 × 12 × 128-E cluster structure uses a total of 8349696 bytes video memory including the 4096 render targets, as this comparison uses 2048 point lights and 2048 spot lights with the same light setup as AMD's demo. The clustered light assignment case takes 0.63 ms on average.

Figure 1.18 clearly shows that tiled light culling suffers from depth discontinuities and that at comparable performance the clustered light assignment performs better light assignment over all as well as having no depth discontinuities. The same is true when looking at the light types individually in Figures 1.19 and 1.20, but the spot light comparison also shows a significant reduction in lighting

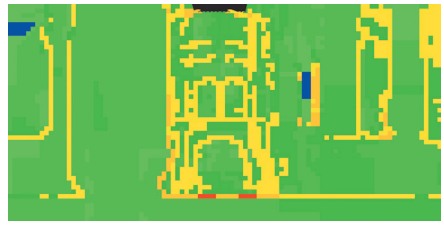
(a) Clustered shading using 24 × 12 × 128-E cluster structure.



(b) Tiled shading using 96 × 48 tiled structure.

Figure 1.19. Comparison between AMD's Forward+ tiled light culling demo using 2048 point lights and no spot lights. Legend can be viewed in Figure 1.17.

(a) Clustered shading using 24 × 12 × 128-E cluster structure.



(b) Tiled shading using 96 × 48 tiled structure.

Figure 1.20. Comparison between AMD's Forward+ tiled light culling demo using no point lights and 2048 spot lights. Legend can be viewed in Figure 1.17.

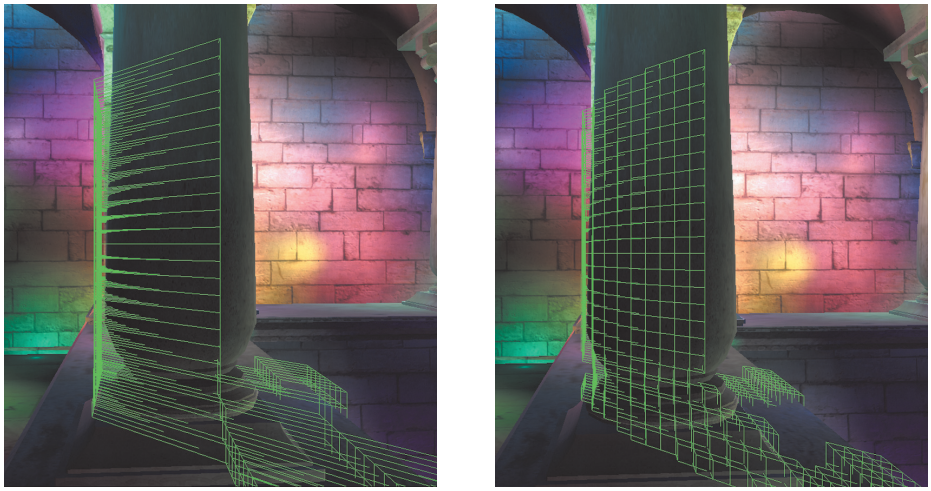
calculations when using clustered light assignment. This proves both that approximating light types as spheres is detrimental to shading performance when using non-spherical light types and that using conservative rasterization with light meshes is efficient.

1.5.4 Depth Distribution

Figure 1.21 displays the negative side of having a perspective cluster structure with exponential depth distribution. Clusters far away will always be larger than the ones up close, and they will accumulate more lights, causing a large worst-case shading time for pixels in the red zone. Using a cluster structure with a large amount of clusters will mitigate the worst case, but the same ratio between worst and best case is still present. Using a linear depth distribution will reduce the worst case but at the same time increase the best case times. Figure 1.22 shows how linear depth distribution covers more empty space where the exponential depth distribution is very fine grained and follows the structure of the pillar. The small clusters are what create a very good best case, but as can

(a) $24 \times 12 \times 32$ cluster structure. (b) $48 \times 24 \times 128$ cluster structure.

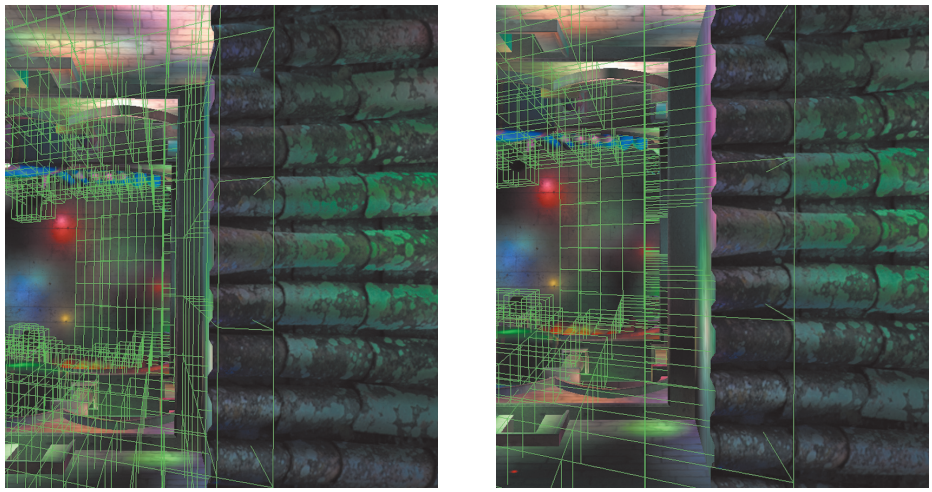
Figure 1.21. Two screen captures that show the number of lights used for shading each pixel. In this scene, 4096 lights are used: Green is 1 light, blue is 19 lights, and red is 38 or more lights. Values in between are interpolated colors.



(a) Linear depth distribution. (b) Exponential depth distribution.

Figure 1.22. Two screen captures that show clusters close to the camera. Side view. Cluster structure size is $48 \times 24 \times 128$.

be seen in Figure 1.23, the exponential depth distribution causes large clusters far from the camera as opposed to the linear distribution. Note that the depth distribution only affects the slice depth of the clusters, and even when increasing the number of cluster slices, making them thinner, the x- and y-size will remain the same. Increasing the number of slices will give better light assignment but will experience diminishing returns at a certain point due to the clusters still being large in the x- and y-dimensions and capturing many lights.



(a) Linear depth distribution.

(b) Exponential depth distribution.

Figure 1.23. Two screen captures that show clusters far from the camera. Top-down view. Cluster structure size is $48 \times 24 \times 128$.

Figure 1.12 shows that the exponential depth distribution, compared to linear depth distribution, results in better shading times in all cases. This is, however, not the case when looking at Figure 1.14, where both the $24 \times 12 \times 32$ and $24 \times 12 \times 64$ cluster structures have better shading times when using a linear depth distribution. This is caused by the fact that those cluster structures contain large clusters far away from the camera. This does not become an issue in a scene with few lights as the worst case large clusters only make up a minority of the shading cost. When a large amount of lights are used in the scene, the worst-case large clusters will be a majority of the shading cost. As can be seen in the cases where the clusters are smaller, the exponential depth distribution gives a better shading time.

There is a correlation between cluster shape and light assignment results where a cube-like cluster shape provides a good base shape. Looking at clusters structures $24 \times 12 \times 128$-E and $48 \times 24 \times 32$-E in Figure 1.14, where both contain the same amount of clusters, it is evident that the more cube-like clusters in $24 \times 12 \times 128$-E results in better performance. The performance increase gained when going from $24 \times 12 \times 128$-L to $24 \times 12 \times 128$-E is attributed to the exponential distribution creating cube-like clusters as opposed to the linear distribution, but 48x24x32-L does not benefit from going to $48 \times 24 \times 32$-E as the clusters will still have a dominant slice depth compared to the x- and y-dimensions.

1.6 Conclusion

This chapter has presented a novel technique for assigning arbitrarily shaped convex light types to clusters using conservative rasterization with good results and performance. The technique is not limited to clusters as many steps can be shared with a tiled shading implementation, nor is the technique limited to deferred shading. Using the technique to shade transparent object works without having to modify anything, and there is no requirement for a depth prepass.

Looking at the results in Section 1.5, it can be concluded that doing a finer clustering will be worthwhile as the shading pass becomes faster. Using costly shading models with many lights will increase the shading time significantly, while the light assignment will stay constant and be a minor part of the entire cost. With that said, there is a drawback of doing fine clustering: the memory usage. The total memory usage for the $48 \times 24 \times 128$ cluster structure at 4096 lights adds up to 45.4 MB, while the $24 \times 12 \times 64$ cluster structure uses 6.9 MB. The larger cluster structure achieves 28.3% better shading performance at a cost of using 6.6 times more memory.

As for finding the right cluster setup, the results have proven that cluster shape and size matters and that large and unevenly shaped clusters will be detrimental to the shading performance compared to cube-like clusters. Using an exponential depth distribution can help create cube-like clusters and gain some performance compared to linear depth distribution. However, if there are too few slices, the exponential structure will suffer from very large, far away clusters and provide worse light assignment.

Bibliography

[Andersson 09] Johan Andersson. "Parallel Graphics in Frostbite—Current and Future." Beyond Programmable Shading, SIGGRAPH Course, New Orleans, LA, August 3–7, 2009.

[Balestra and Engstad 08] Christophe Balestra and Pål-Kristian Engstad. "The Technology of Uncharted: Drake's Fortune." Game Developers Conference, San Francisco, CA, February 18–22, 2008.

[Fauconneau 14] Mark Fauconneau. "Forward Clustered Shading." https://software.intel.com/sites/default/files/managed/27/5e/Fast%20Foward%20Clustered%20Shading%20(siggraph%202014).pdf, 2014. Accessed May 20, 2015.

[Harada et al. 13] Takahiro Harada, Jay McKee, and Jason C. Yang. "Forward+: A Step Toward Film-Style Shading in Real Time." In *GPU Pro 4: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 115–135. Boca Raton: A K Peters/CRC Press, 2013.

[Leadbetter 14] Richard Leadbetter. "The Making of Forza Horizon 2." http://www.eurogamer.net/articles/digitalfoundry-2014-the-making-of-forza-horizon-2, 2014. Accessed May 18, 2015.

[Microsoft] Microsoft. "Floating-Point Rules." https://msdn.microsoft.com/en-us/library/windows/desktop/jj218760(v=vs.85).aspx. Accessed May 18, 2015.

[Möller and Trumbore 05] Tomas Möller and Ben Trumbore. "Fast, Minimum Storage Ray/Triangle Intersection." In *ACM SIGGRAPH 2005 Courses,* article no. 7. New York: ACM, 2005.

[Olsson and Assarsson 11] Ola Olsson and Ulf Assarsson. "Tiled Shading." *Journal of Graphics, GPU, and Game Tools* 15:4 (2011), 235–251.

[Olsson et al. 12] Ola Olsson, Markus Billeter, and Ulf Assarsson. "Clustered Deferred and Forward Shading." In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, pp. 87–96. Aire-la-Ville, Switzerland: Eurographics Association, 2012.

[Persson and Olsson 13] Emil Persson and Ola Olsson. "Practical Clustered Deferred and Forward Shading." Advances in Real-Time Rendering in Games, SIGGRAPH Course, Anaheim, CA, July 23, 2013. Available online (http://s2013.siggraph.org/attendees/courses/session/advances-real-time-rendering-games-part-i).

[Swoboda 09] Matt Swoboda. "Deferred Lighting and Post Processing on Playstation 3." Game Developers Conference, San Francisco, CA, March 23–27, 2009.

[Thomas 15] Gareth Thomas. "Advanced Visual Effects with DirectX 11 and 12: Advancements in Tile-Based Compute Rendering." Game Developers Conference, San Francisco, CA, March 2–6, 2015. Available online (http://www.gdcvault.com/play/1021764/Advanced-Visual-Effects-With-DirectX).

[Yang et al. 10] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. "Real-Time Concurrent Linked List Construction on the GPU." *Computer Graphics Forum* 29:4 (2010), 1297–1304.