Thesis no: MECS-2015-06

Clustered Shading

Assigning arbitrarily shaped convex light volumes using conservative rasterization

Kevin Örtegren

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Game and Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Kevin Örtegren
E-mail: kevinortegren@gmail.com

External advisor:
Emil Persson, Head of Research
Avalanche Studios

University advisor:
Ph.D. Hans Tap
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. In this thesis, a GPU-based light culling technique performed with conservative rasterization is presented. Accurate lighting calculations are expensive in real-time applications and the number of lights used in a typical virtual scene increases as real-time applications become more advanced. Performing light culling prior to shading a scene has in recent years become a vital part of any high-end rendering pipeline. Existing light culling techniques suffer from a variety of problems which clustered shading tries to address.
Objectives. The main goal of this thesis is to explore the use of the rasterizer to efficiently assign convex light shapes to clusters. Being able to accurately represent and assign light volumes to clusters is a key objective in this thesis.
Methods. This method is designed for real-time applications that use large amounts of dynamic and arbitrarily shaped convex lights. By using using conservative rasterization to assign convex light volumes to a 3D cluster structure, a more suitable light volume approximation can be used. This thesis implements a novel light culling technique in DirectX 12 by taking advantage of the hardware conservative rasterization provided by the latest consumer grade Nvidia GPUs. Experiments are conducted to prove the efficiency of the implementation and comparisons with AMD´s Forward+ tiled light culling are provided to relate the implementation to existing techniques.
Results. The results from analyzing the algorithm shows that most problems with existing light culling techniques are addressed and the light assignment is of high quality and allows for easy integration of new convex light types. Assigning the lights and shading the CryTek Sponza scene with 2000 point lights and 2000 spot lights takes 2.92ms on a GTX970.
Conclusions. The conclusion shows that the main goal of the thesis has been reached to the extent that all existing problems with current light culling techniques have been solved, at the cost of using more memory. The technique is novel and a lot of future work is outlined and would benefit the validity of the implementation if further researched.

Keywords: clustered shading, lighting, light culling

Acknowledgements

Thanks to Avalanche Studios for providing the necessary hardware and a creative work place for me to be able to do this research. Special thanks to my supervisor at Avalanche Studios, Emil Persson, who has followed my work and provided helpful input and ideas. This work would not have been possible without him. I would also like to thank Joel De Vahl for discussing theories and pointing me to useful resources. Additional thanks to Niclas Thisell and Alvar Jansson for taking interest in the performance critical parts of the algorithm.

Contents

List of Figures

Source code listings

List of Tables

Chapter 1

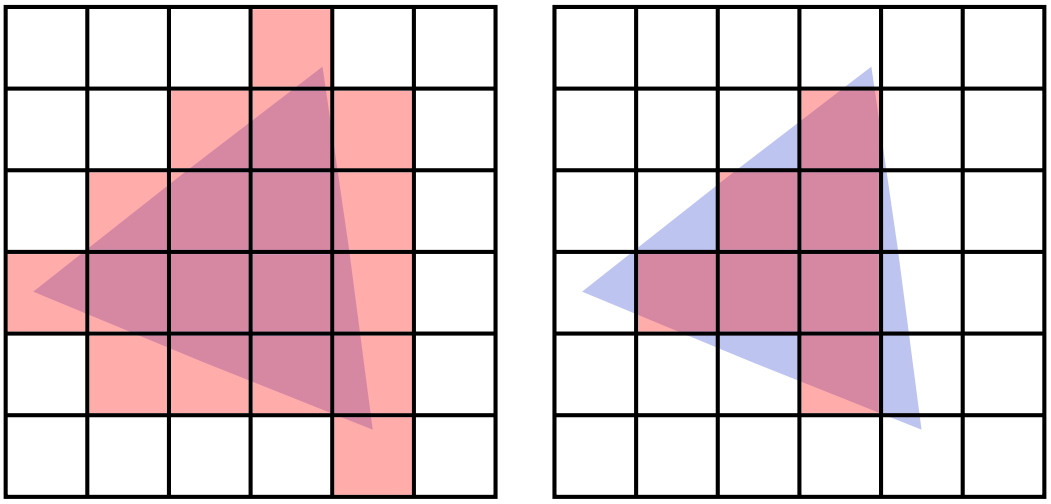
Introduction and background

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 thesis presents a GPU-based variation of Clustered Shading [PO13], which is a technique that improves on the currently popular Tiled Shading [OA11,Swo09,BE08,And09] 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 thesis is to explore the use of the rasterizer to efficiently assign convex light shapes to clusters.

1.1 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 users from using it in other ways. The normal rasterizer mode will rasterize a pixel if the pixel center is covered by a primitive. Conservative rasterization is an alternative rasterizer 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. As can be seen in the figure, only the conservative rasterization will capture the entire triangle which is important when having to find what pixels a triangle overlaps. This particular mode of conservative rasterization is called overestimated conservative rasterization where as the opposite, called underestimated conservative rasterization, only raterizes pixels that are fully covered. The reason for exploring the rasterizer in this thesis is because DirectX 12 and DirectX 11.3 have added hardware support for overestimated conservative rasterization. Worth noting is that [OA11], in section 6.1,

briefly mentioned the use of the rasterizer for light insertion as a possibility. With the popularity of the DirectX graphics API and the power of modern hardware there is an opportunity to find new and efficient ways to use the rasterizer.



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

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

Chapter 2

Related Work

Modern dynamic light culling techniques can be partitioned into two categories: tiled shading and clustered shading.

2.1 Clustered shading

Clustered Shading is a new technique which has not been covered to a large extent, nor has it been well established in the game industry as there is only one commercially available game using the technique as of now; Forza Horizon 2 [Lea14]. It was first introduced by [OBA12a] in 2012 and their recent work has also focused on Clustered Shading [OBA12b]. Other, non-published works and presentations, are available from Intel who has released a demo application and a presentation on Forward Clustered Shading [Fau14] and from Avalanche Studios who has presented a practical solution for Clustered Shading in their game engine [PO13]. AMD has a presentation on tiled shading and clustered shading [Tho15] where both are discussed. All previously presented work on clustered shading relies on testing spheres against clusters in some way.

2.2 Tiled shading

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 [HMY13] which effectively moves Tiled Shading towards 3D clusters.

2.3 Conservative rasterization

A common use case for conservative rasterization is to voxelize primitives [ZCEP07] for various techniques, e.g global illumination [CNS+11] or ray traced shadows [Sto15]. Some techniques resort to using software conservative rasteriza-

tion [HAMO05] as there has not been native hardware support for it previously. The technique presented in this thesis has some similarities to voxelizing techniques, but is not the main purpose.

Chapter 3

Aim and Objectives

This thesis aims to find a novel light culling technique that solves the existing problems with currently used techniques by using conservative rasterization. The research question defined for this thesis is:

RQ *Is it possible to create a solution for assigning arbitrarily shaped convex light volumes to a 3D cluster structure using conservative rasterization that solves existing problems with current light culling techniques?*

This chapter explains the problems that are present in current clustered shading and tiled shading techniques and what objectives are defined to answer the research question.

3.1 Problem statement

The point of doing light culling before shading a scene is to avoid doing unnecessary lighting calculations. The optimal solution would be to, for each pixel, only calculate the lights actually affecting the particular pixel. This is the solution that all light culling techniques are aiming for, but with current hardware and real-time limitations this is not a feasible solution. Thus, the light culling is performed on a coarser level. The solutions provided by tiled and clustered shading show good light culling results when applied on point lights. They do, however, suffer from inaccuracies when it comes to other light shapes, as spot lights for example.

A proposed solution for approximating a spot light is to simply use a sphere around it to be able to efficiently perform the light culling [Tho15]. Some techniques try to refine the sphere approximation to better fit another light type [PO13], but this requires a separate implementation for every light shape and they might not be perfect or efficient. With most virtual real-time 3D scenes having more than one light type, this proves to be a real problem.

3.2 Objectives

The objectives of this work are the following:

- Find an efficient solution for assigning lights to clusters using conservative rasterization.
- Evaluate the two depth distribution functions, linear and exponential, from previous clustered shading work [PO13].
- Perform experiments that produce results that answer and confirm the research question.

These objectives are aimed towards the main goal of the thesis; to explore the use of the rasterizer to efficiently assign convex light shapes to clusters.

Chapter 4

Method

This chapter describes the resulting implementation and how the previously stated problems were solved using the novel technique developed for this thesis. The last section presents details on how and why the experiments were set up and how the results were obtained.

4.1 Algorithm

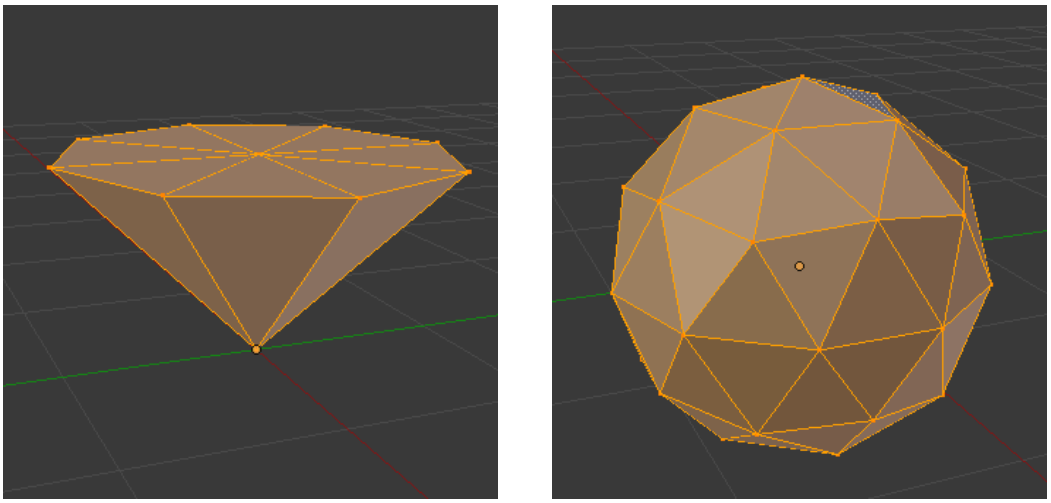
This section will go through the different steps included in the light assignment algorithm as well as explain how the main data structure used for storing light and cluster data is created and managed, as it is an intricate part of the technique. This section includes implementation details from the test application, which is created with C++, DirectX 12 and HLSL.

4.1.1 Light shape representation

Lights must have a shape representation to be able to be inserted in to clusters. Approximating every light shape as an analytical sphere is the easiest and computationally cheapest approach but 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 resorts 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 provides a general representation model for all light shapes. The level of flexibility when working with vertex meshes is very high as the meshes can be created with variable detail. Meshes are created as unit shapes, where vertices are constrained to -1 to 1 in 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 run time to allow 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. This also adds to the flexibility of the vertex mesh approximation as meshes are not necessarily bound to the shape they have at creation. In the case of using low amounts of vertices for light shapes, they could easily be created in code and also use very small vertex formats, for example R8G8B8 is enough for the shapes in figure 4.1.



(a) A unit cone mesh with 10 vertices. (b) A unit sphere mesh with 42 vertices.

Figure 4.1: Two example unit shapes created in Blender.

4.1.2 Shell pass

This 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 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 size 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 4.2.

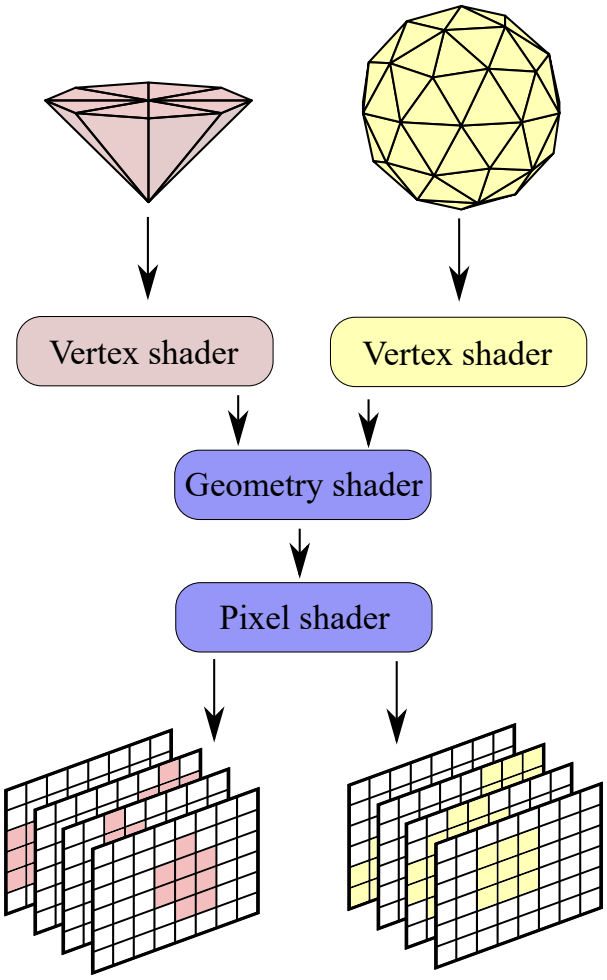


Figure 4.2: Illustration of the entire shell pass.

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 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.

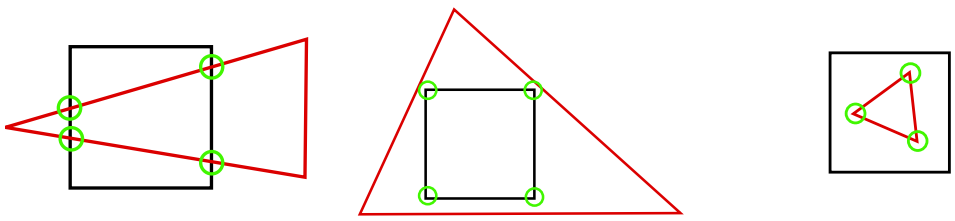
Geometry shader

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 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 as it, besides selecting the render target index, adds unnecessary overhead. Geometry shader source code is available in appendix B.

Pixel shader

The pixel shader performs most of the math 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 min and max depth 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 as 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 min and max depth for a triangle in a tile three cases are used, see figure 4.3. The three points that can be min or max depth in a tile are:



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

Figure 4.3: The three cases of finding min and max depth on a triangle in a tile.

Figure 4.3a. Where a vertex edge intersects the tile boundary planes.

Listing 4.1 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 it would return +/-INF. It is an optimization to not check for these cases as the IEEE 754-2008 floating point specification in HLSL [Mic] 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.

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

```
1  bool linesegment_vs_plane(float3 p0, float3 p1, float3 pn, out
       float lerp_val)
2  {
3      float3 u = p1 - p0;
4
5      float D = dot(pn, u);
6      float N = -dot(pn, p0);
7
8      lerp_val = N / D;
9      return !(lerp_val != saturate(lerp_val));
10 }
```

Listing 4.1: Vertex edge vs. tile boundary plane intersection.

Figure 4.3b. Where a triangle covers a tile corner.

To find the depth at a corner of a tile is simply a matter of performing four ray vs triangle intersections, one at each corner of the tile. The ray-triangle intersection function in listing 4.2 is derived from [MT05].

```
1  bool ray_vs_triangle(float3 ray_dir, float3 vert0, float3 vert1
       , float3 vert2, out float z_pos)
2  {
3      float3 e1 = vert1 - vert0;
4      float3 e2 = vert2 - vert0;
5      float3 q = cross(ray_dir, e2);
6      float a = dot(e1, q);
7
8      if(a > -0.000001f && a < 0.000001f)
9          return false;
10
11     float f = 1.0f / a;
```

```
12      float u = f * dot(-vert0, q);
13
14      if(u != saturate(u))
15          return false;
16
17      float3 r = cross(-vert0, e1);
18      float v = f * dot(ray_dir, r);
19
20      if(v < 0.0f || (u + v) > 1.0f)
21          return false;
22
23      z_pos = f * dot(e2, r) * ray_dir.z;
24
25      return true;
26  }
```

Listing 4.2: Ray vs triangle intersection.

Figure 4.3c. Where a vertex is completely inside a tile.

The signed distance from a point to a plane in 3D 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 the distance is calculated to. The variable d is defined as $d = -ax_0 - by_0 - cz_0$ where (x_0, y_0, z_0) is a point on the plane. As all planes go through the origin in view space, the variable d is eliminated and 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 which side of the plane the vertex is. See code listing 4.3 for HLSL code of these two functions.

```
1  bool is_in_xslice(float3 top_plane, float3 bottom_plane, float3
       vert_point)
2  {
3      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);
4  }
5
6  bool is_in_yslice(float3 left_plane, float3 right_plane, float3
       vert_point)
7  {
```

```
8     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 );
9 }
```

Listing 4.3: Vertex point vs. tile boundary planes intersection.

When all three cases have been evaluated the min and max depth for a tile has 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 min depth will be stored and in the back facing case the max 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 format R8G8_UNORM which allows for the cluster structure to have up to 256 clusters in Z-dimension. As many triangles can be in the same tile for a light shape it is important to find the min and max Z-cluster 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 which color channel the result is stored in. 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 4.4 illustrates the found min and max 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 4.5 where two point lights and a spot light have been processed, with the colored clusters representing the min and max Z-clusters for each tile and light. The complete source code for the pixel shader is available in appendix A.

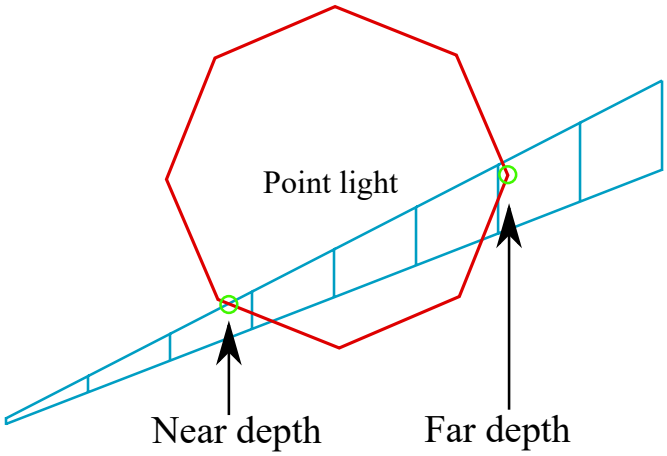


Figure 4.4: Top down view of one tile and what cases have found the min and max depth for a point light mesh.

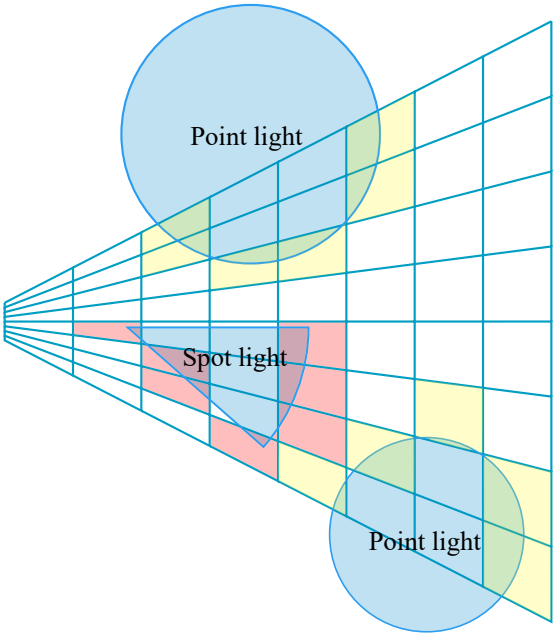


Figure 4.5: 2D top down view of shell pass.

Depth distribution

The depth distribution determines how the Z cluster planes are distributed along the Z-axis in view space. The depth distribution is represented as a function which 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

$$clusterZ = \log_2(d)\frac{1}{\log_2(f) - \log_2(n)}(c-1)+((1-\log_2(n))\frac{1}{\log_2(f) - \log_2(n)}(c-1))$$

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 z-dimension. Note that most of these are constants and are not recalculated. Figure 4.6 shows the two functions in a graph with example vaules.

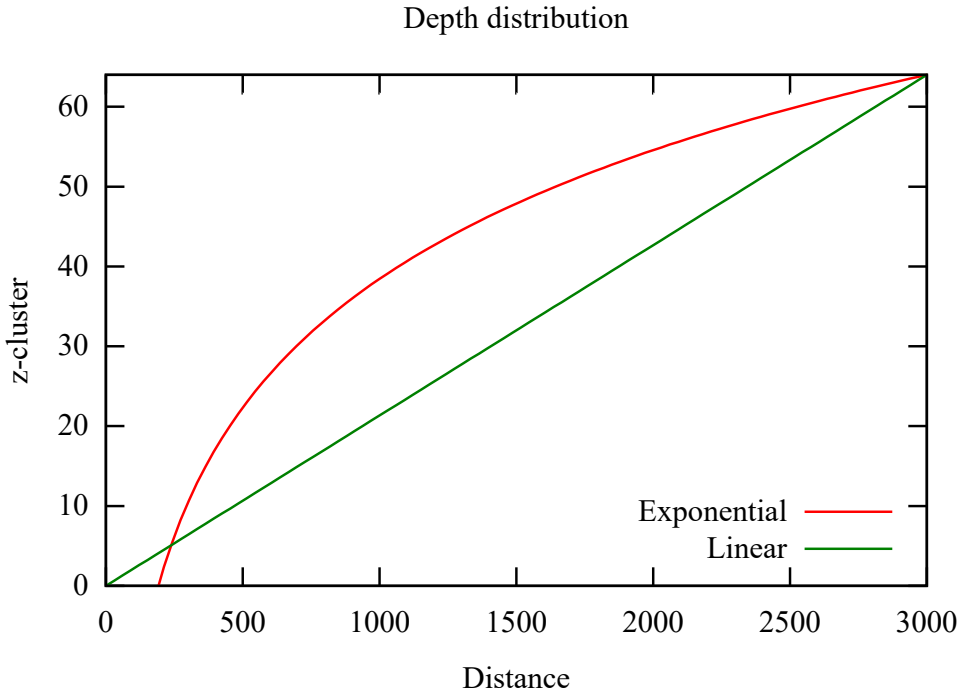


Figure 4.6: Graph of two distribution functions over an example depth of 3000 with 64 clusters in z-dimension. The second z-slice of the exponential function is set to start at 200.

4.1.3 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 [YHGT10].

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 is in most cases 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 bit or 32 bit depending on the maximum amount of lights needed in the game. Examples of the data in a node are shown in table 4.1. 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 trade-off has to be made between memory savings and maximum number of supported lights. Note that in table 4.1 the 32 bit node uses 2 bit for the light type and 10 bit for the lightID, which results in 4096 total lights. This can be switched around to whatever 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 4.7. The start offset buffer is a Direct3D ByteAddress-Buffer 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 which can be fetched and used for shading. The last part is the actual light data storage which 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 scattering memory access pattern is motivated by the ratio between number of clusters and number of tiles per light; to achieve a gathering memory access pattern, one thread per cluster would be used which would result in an amount of threads less than what the hardware has available, thus not utilizing the full computational power of the hardware. Take the cluster structure size of 24x12x128 with 4000 lights as an example, using one thread per cluster would result in 36864 threads in parallel where as using one thread per tile per light results in 1152000 threads. While using one thread per cluster would eliminate one of the atomic operations in the compute shader, it would require each thread to read the corresponding near and far clusters from each light render target, resulting in 147456000 texture loads; 128 times more texture loads.

The complete compute shader is outlined in listing 4.4.

Data	Size	Max value
Light type	8 bit	256
LightID	24 bit	16777216
Link	32 bit	4294967296

(a) 64 bit node layout.

Data	Size	Max value
Light type	2 bit	4
LightID	10 bit	1024
Link	20 bit	1048576

(b) 32 bit node layout.

Table 4.1: 64 and 32 bit node layout examples.

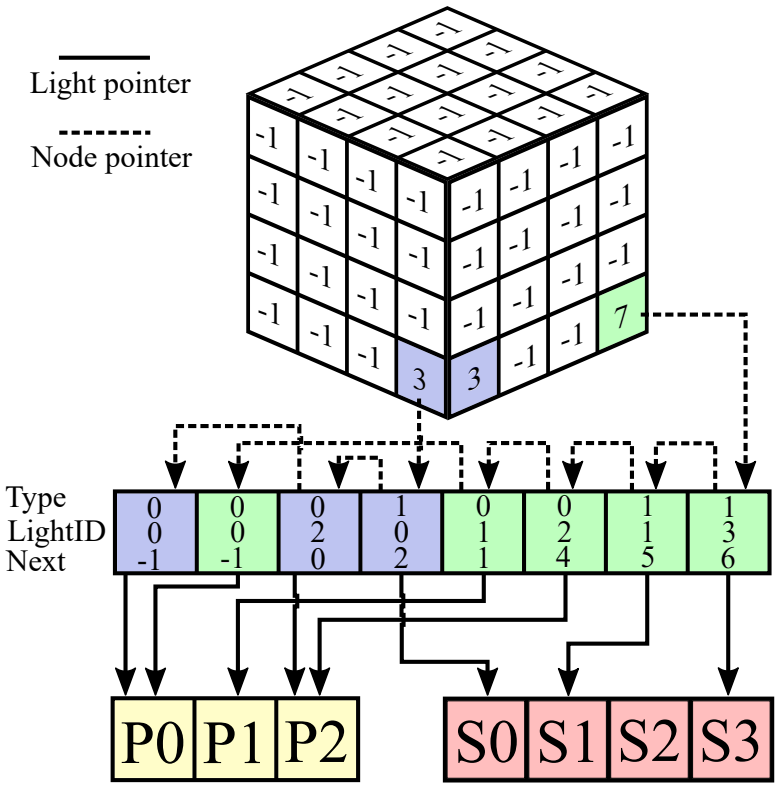


Figure 4.7: Illustration of the light linked list. The "Next" field is the index to the next node in the linked list, if a node is pointed to and has -1 as next node it means that it is the tail node and no more nodes are linked to that sequence. The 3D structure contains a pointer from each cluster 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.

```
1 //This array has NUM_LIGHTS slices and contains the near and far
     cluster z for each tile
2 Texture2DArray<float2> conservativeRTs : register(t0);
```

```
3
4  //Linked list of light IDs.
5  RWByteAddressBuffer StartOffsetBuffer : register(u0);
6  RWStructuredBuffer<LinkedLightID> LinkedLightList : register(u1);
7
8  [numthreads(TILESX, TILESY, 1)]
9  void main( uint3 thread_ID : SV_DispatchThreadID ){
10     //Load near and far values(x is near and y is far)
11     float2 near_and_far = conservativeRTs.Load(int4(thread_ID, 0));
12
13     if(near_and_far.x == 1.0f && near_and_far.y == 1.0f)
14         return;
15
16     //Unpack to clusterZ space([0,1] to [0,255]). Also handle cases
       where no near or far cluster was written.
17     uint near = (near_and_far.x == 1.0f) ? 0 : uint(near_and_far.x *
        255.0f + 0.5f);
18     uint far  = (near_and_far.y == 1.0f) ? (CLUSTERSZ - 1) : uint(((
       CLUSTERSZ - 1.0f) / 255.0f - near_and_far.y) * 255.0f + 0.5f);
19
20     //Loop through near to far and fill the light linked list
21     uint offset_index_base = 4 * (thread_ID.x + CLUSTERSX *
       thread_ID.y);
22     uint offset_index_step = 4 * CLUSTERSX * CLUSTERSY;
23     uint type = light_type;
24     for(uint i = near; i <= far; ++i){
25         uint index_count = LinkedLightList.IncrementCounter();
26         uint start_offset_address = offset_index_base +
       offset_index_step * i;
27
28         uint prev_offset;
29         StartOffsetBuffer.InterlockedExchange(start_offset_address,
       index_count, prev_offset);
30
31         LinkedLightID linked_node;
32         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.
33         linked_node.link = prev_offset;
34
35         LinkedLightList[index_count] = linked_node;
36     }
37 }
```

Listing 4.4: The complete compute shader for the fill pass.

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 4.8.

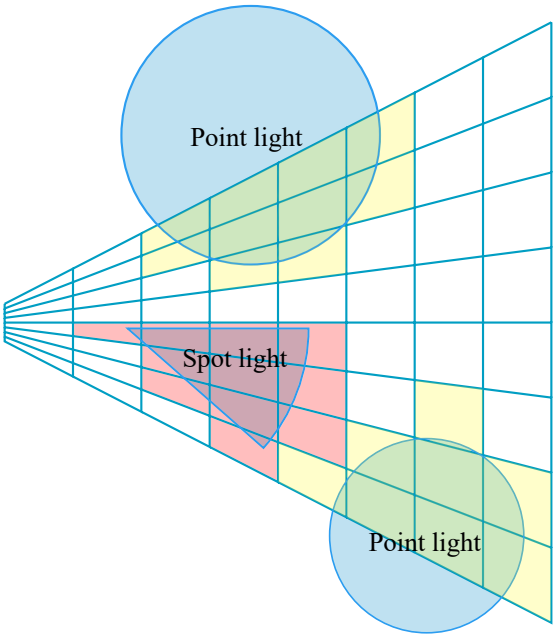


Figure 4.8: 2D top down view of fill pass.

4.2 Shading

The shading is done in the pixel shader by calculating which cluster the pixel lies in 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 code listing 4.5.

```
1  uint light_index = start_offset_buffer[clusterPos.x + CLUSTERSX *
       clusterPos.y + CLUSTERSX * CLUSTERSY * zcluster];
2
3      float3 outColor = float3(0,0,0);
4
5      LinkedLightID linked_light;
6
7      if(light_index != 0xFFFFFFFF)
8      {
9          linked_light = light_linked_list[light_index];
10
11         //Spot light
12         while((linked_light.lightID >> 24) == 1)
13         {
14             uint lightID = (linked_light.lightID & 0xFFFFFF);
15
16             outColor += SpotLightCalc(pos, norm, diff, spotLights[
       lightID]);
```

```
17
18              light_index = linked_light.link;
19
20              if(light_index == 0xFFFFFFFF)
21                  break;
22
23              linked_light = light_linked_list[light_index];
24          }
25
26          //Point light
27          while((linked_light.lightID >> 24) == 0)
28          {
29              uint lightID = (linked_light.lightID & 0xFFFFFF);
30
31              outColor += PointLightCalc(pos, norm, diff, pointLights[
    lightID]);
32
33              light_index = linked_light.link;
34
35              if(light_index == 0xFFFFFFFF)
36                  break;
37
38              linked_light = light_linked_list[light_index];
39          }
40      }
41
42      return float4(outColor, 1.0f);
```

Listing 4.5: Pixel shader code for going through the light linked list for shading a pixel.

Finding out which cluster the pixel should pull the lights from is done by translating the screen space x and y position of the pixel into cluster x and y space. 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 getting the z-position of the interpolated geometry in the case of forward shading. The sampled depth is then translated into z cluster space by applying the same depth distribution function used in the shell pass. Figure 4.9 shows what clusters are used for shading in an example scene using the assigned lights from figure 4.8.

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 the linked sequence. See figure 4.7 where the node pointer arrows show how the linked list will be traversed.

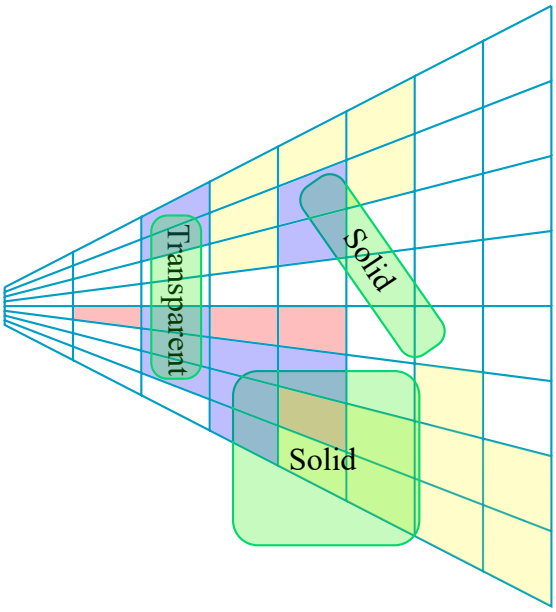


Figure 4.9: 2D top down view of sampled clusters in a scene with objects. Note that transparent objects are shaded the same way as opaque objects. Colored clusters contain lights and the blue clusters are used for shading the geometry.

4.3 Experiment

The test scenario consists of the CryTek Sponza Atrium model [Mei10] with point lights and spot lights of variable sizes placed in the scene using a deterministically seeded random number generator. The scene is set up with a fixed camera position and camera direction which represents an average view of the scene. The view contains geometry close to the camera, far away from the camera and large depth discontinuities. The test scene view is displayed in figure 4.10.

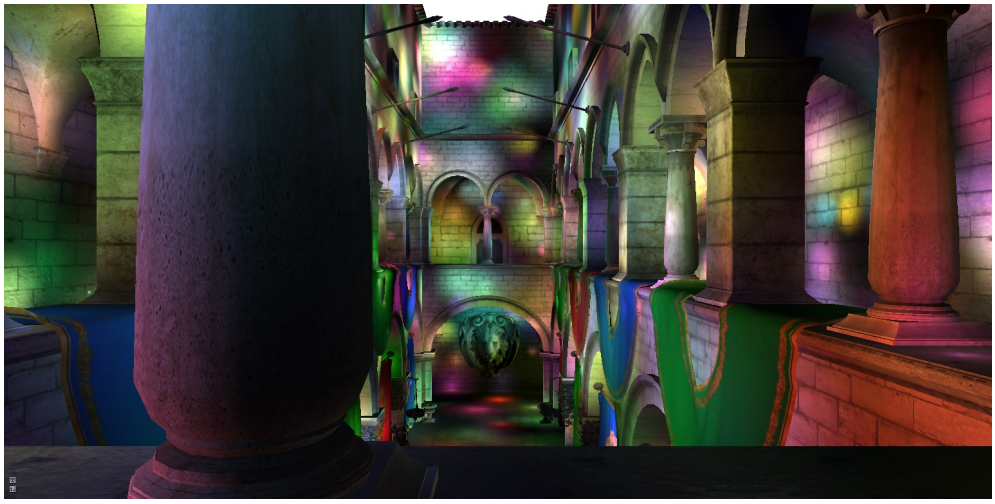


Figure 4.10: Screen capture of the test scene with 2000 point lights and 2000 spot lights.

The test application is run at a resolution of 1536x768 and tested at different cluster structure sizes. The number of lights also differ in the experiment set up. All test cases are measured in terms of performance, memory usage, clustering statistics and a set of visualizations. The clustering statistics include the total number of clusters generated as well as the number of clusters generated for a few selected light shapes. The visualizations are shown and explained in figures 4.11 and 4.12.

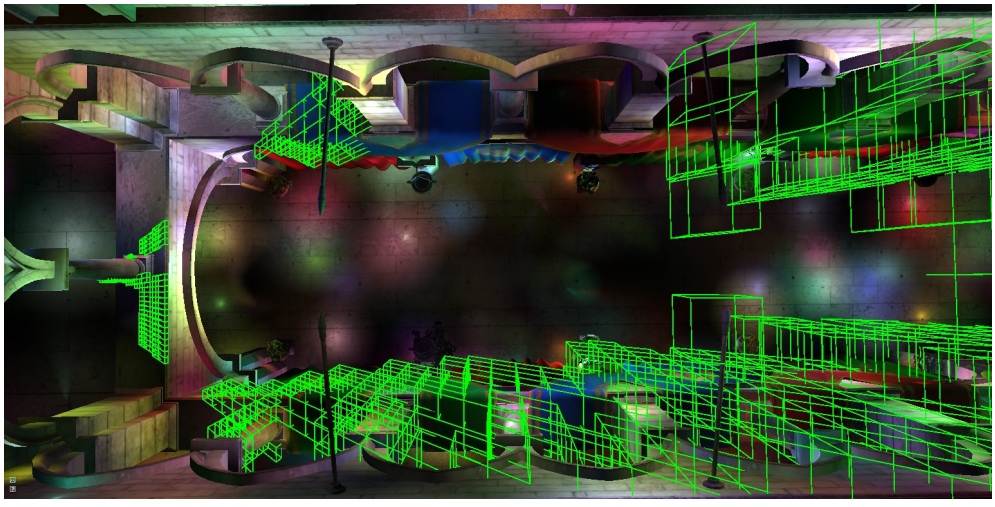


Figure 4.11: A 3D cluster grid representation of the clusters used when shading from the fixed test camera position.

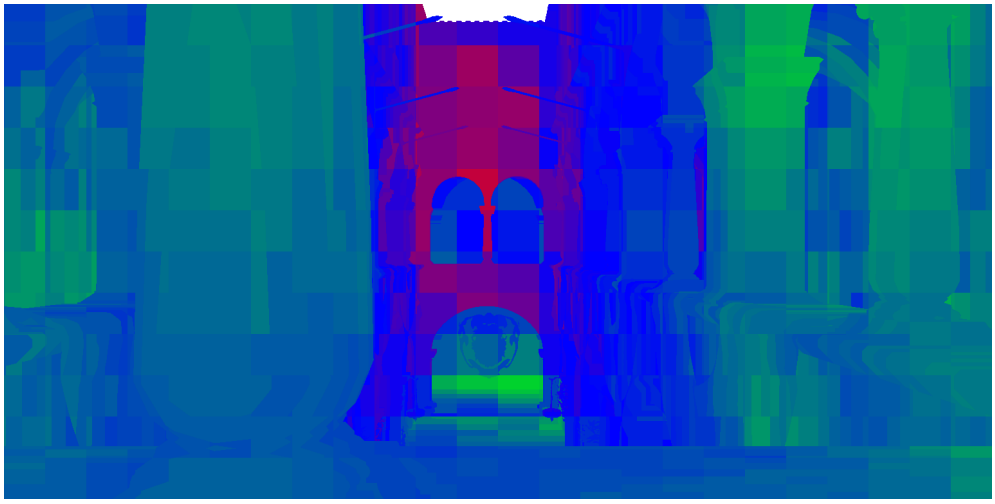


Figure 4.12: A density map of the number of sampled lights in every pixel where the colors correspond to a number of lights.

To evaluate the quality of the light assignment, a direct comparison to AMD´s Forward+ Tiled Shading demo [HMY13] is done by showing a color map using weather radar colors from both implementations. To make the light assignment comparable, both implementations use the same light sizes and positions as the AMD demo and the cluster structure has been chosen due to the performance similarities; light assignment GPU time and memory usage. the AMD demo uses only the min/max tile depth optimization, which can be seen as the base tiled light culling implementation. The full demo and source code is available on AMD´s website.

The hardware and software used for the test is listed below:

- GPU: Nvidia GTX 970 4GB, beta driver 352.63
- CPU: Intel i7-4790K 4.0GHz
- RAM: 16GB 1600MHz DDR3
- OS: Windows 10 Technical Preview Build 10074
- API: DirectX 12, latest early access preview build as of 2015-05-11

The experiment runs 100 frames on each test case, where a test case consists of the unique combination of the cluster structure size, number of lights and the depth distribution. All GPU timings displayed are the average of each experiment. Statistical significance has not been taken into account, as all GPU timings have lower than 1% timing variation.

Listed below is a summary of the variables of the experiment:

- Number of cluster tiles(24x12 and 48x24)
- Number of cluster slices(32, 64 and 128)
- Number of lights of different light types(400, 1000, 2000, 4000)
- Depth distribution(linear and exponential)

The experiment variables are chosen to cover the important aspects of evaluating a light culling technique and to be able to answer and relate to the research question and goal of the thesis. The specific variables cover the low-end, mid-end and high-end cases for the clustered light culling evaluation.

Chapter 5

Results and analysis

This chapter will show results from the performed experiments and presents an analysis of performance, memory, number of assigned clusters and depth distribution 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" means Linear and Exponential, respectively. Performance is measured in milliseconds and all measurements are done on the GPU.

5.1 Performance

Apart from the performance inconsistencies between depth distribution functions, which is analysed in detail in section 5.4, the performance results are consistent. A few observations can be made by examining figures 5.1, 5.2, 5.3 and 5.4: the shell pass increases in time only when the x and y dimensions increase, the fill pass increases in time when any of the three dimensions of the cluster structure increases and the total time increases close to linearly with regards to the number of lights. The times for the two shell passes increase at different rates when the x and y dimensions increase; the point light shell pass has an increase of 4% when going from 24x12 to 48x24 tiles and the spot light shell pass increases by 28% going from 24x12 to 48x24. 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 24x12 tiles to 48x24 will yield up to four times the number of pixel shader invocations for any number of triangles, which in turn means that the slight increase of 4% for the point light shell pass 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 up to 6.5 times slower between 24x12x32 and 48x24x128 at 4000 lights, see figure 5.4. 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 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 light also directly affects the thread coherency of the compute shader as lights that cover many clusters in Z dimension will write more data as each thread writes data from the near to far cluster 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 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 sized 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. Performance data in table format is available in appendix C.

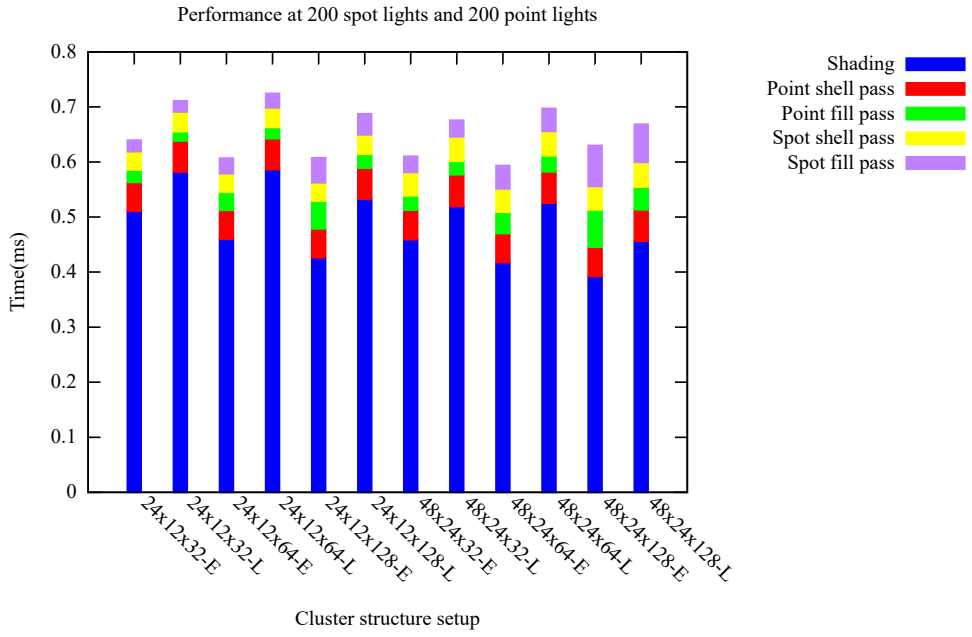


Figure 5.1: Total GPU timings in milliseconds split up into the different passes of the algorithm at 400 lights. Lower is better.

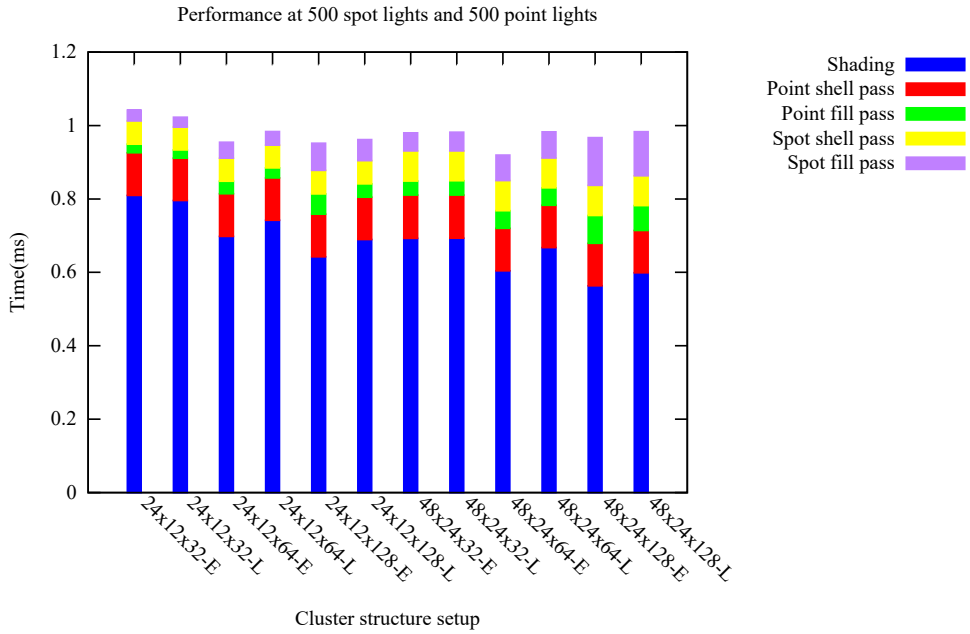


Figure 5.2: Total GPU timings in milliseconds split up into the different passes of the algorithm at 1000 lights. Lower is better.

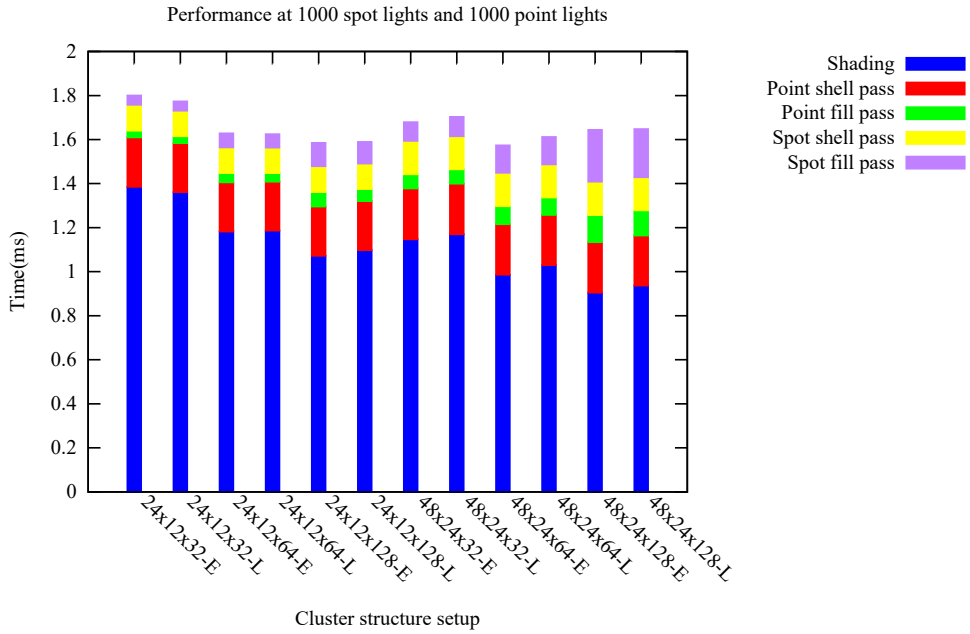


Figure 5.3: Total GPU timings in milliseconds split up into the different passes of the algorithm at 2000 lights. Lower is better.

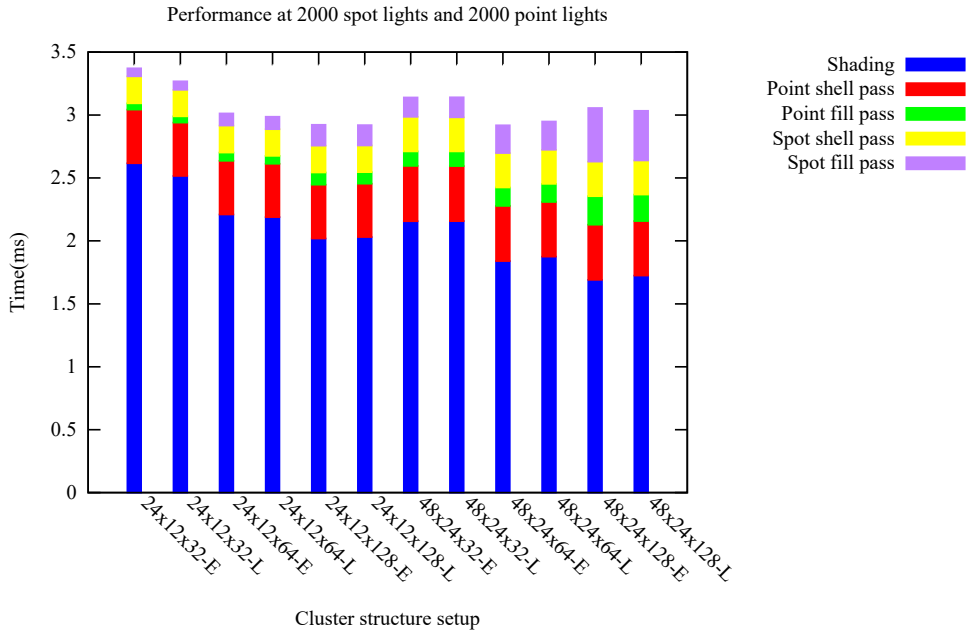


Figure 5.4: Total GPU timings in milliseconds split up into the different passes of the algorithm at 4000 lights. Lower is better.

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 5.5 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 as 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 blocks in the final shaded image. As can be seen in figure 5.5, the actual linked list is a large majority of the memory usage at 4000 lights. Using a 32 bit node would only use half the memory of the linked list, but as previously shown in table 4.1 there would only fit 1048576 linked nodes 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 4000 lights with 24x12 tiles this adds up to 2304000 bytes.

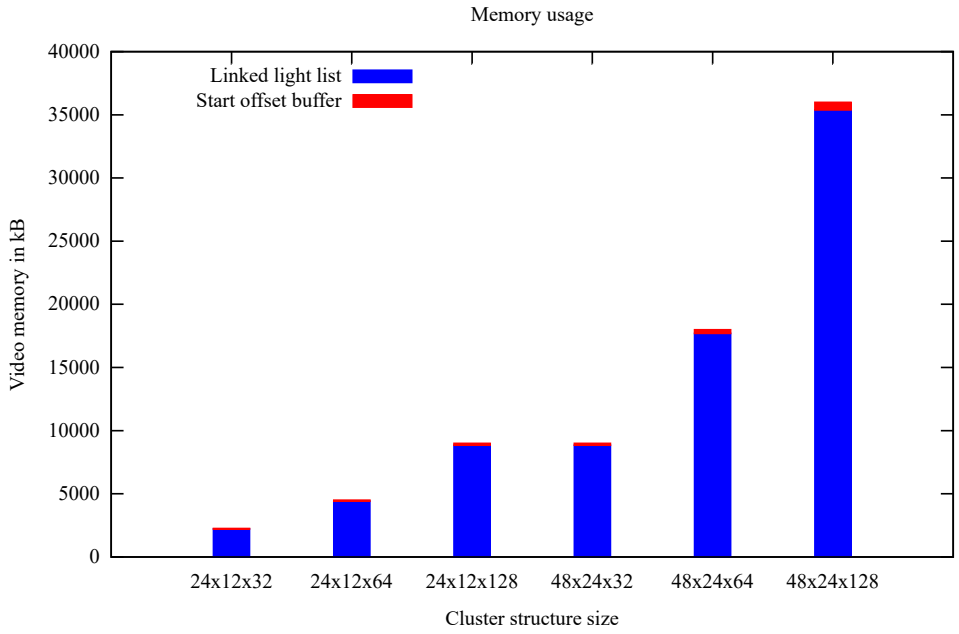


Figure 5.5: Video memory used by the cluster structure and light linked list at 4000 lights. Lower is better.

5.3 Light assignment

Counting the number of assigned nodes in the light linked list will give a rough estimate of the needed safe size of the light linked list. It also gives the number of writes to memory the fill pass has done. Figure 5.6 displays the number of light assignments for all test cases. Note that the linear depth distribution does fewer light assignments, which can be used to lower the size of the light linked list if memory is an issue.

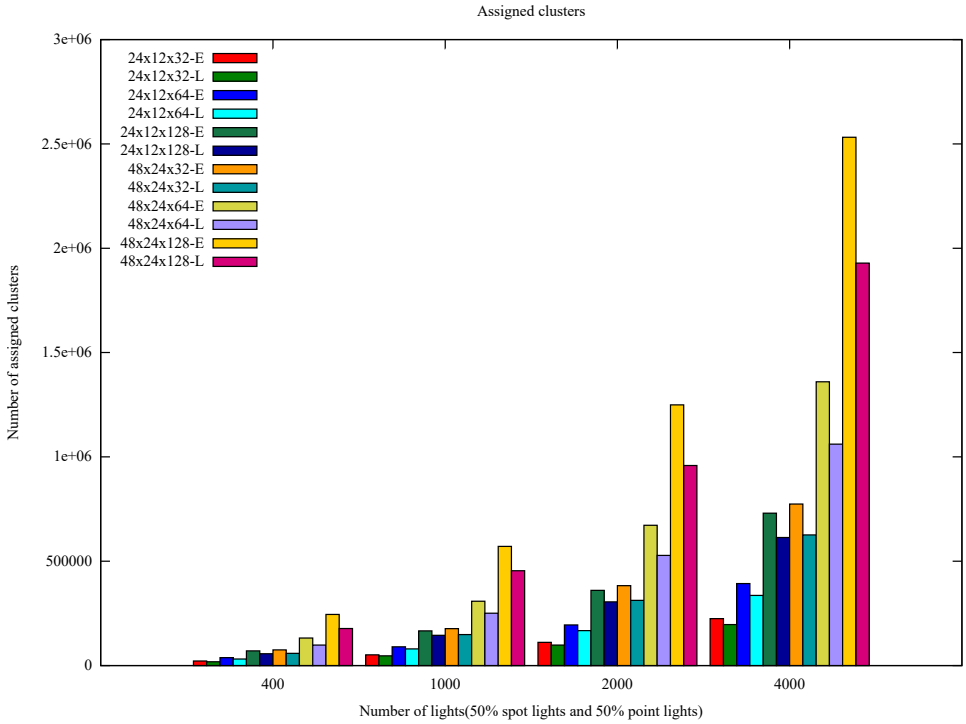
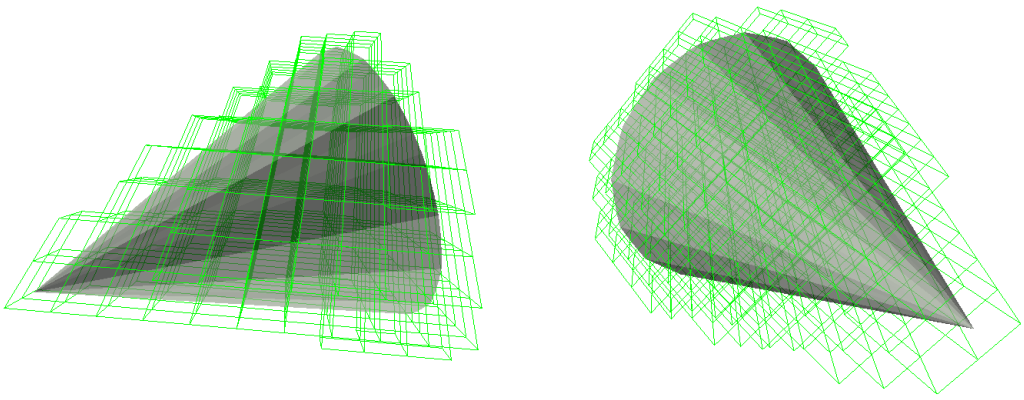


Figure 5.6: Number of cluster light assignments.

Figure 5.7 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. Smaller clusters will give less empty space for an assigned shape and give better shading times.



(a) (b)

Figure 5.7: Clusters that were assigned that spot light are visualized and viewed from two different angles. Perfect clustering with exponential depth distribution captured from a medium distance at a clustered structure size of 24x12x64.

The results from comparing AMD´s Forward+ tiled light culling with the 24x12x128-E cluster structure are demonstrated in Figures 5.9, 5.10 and 5.10. The colors correspond to the number of lighting calculations, where lower is better. AMD´s tiled light culling implementations uses 96x48 tiles, using 6488064 bytes video memory and performing the light assignment in 0.6 ms as average. The 24x12x128-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 0.62 ms as average.

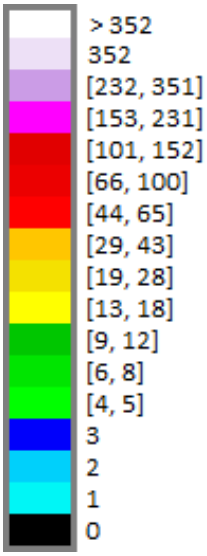
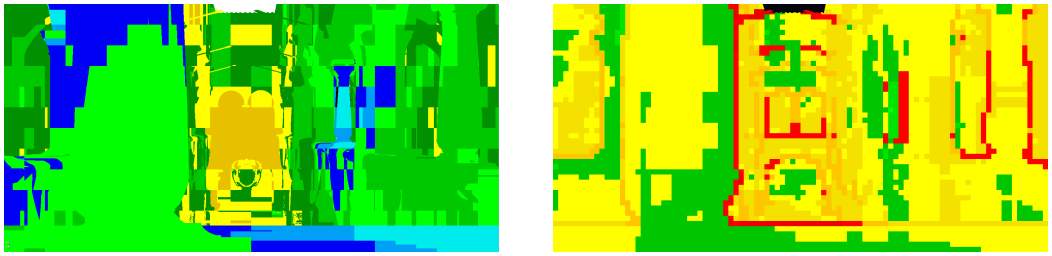


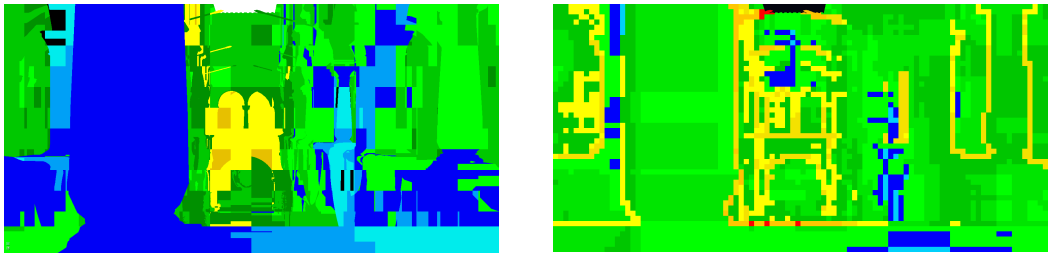
Figure 5.8: Weather radar colors corresponding to a number of lighting calculations.



(a) Clustered shading using 24x12x128-E cluster structure.

(b) Tiled shading using 96x48 tiled structure.

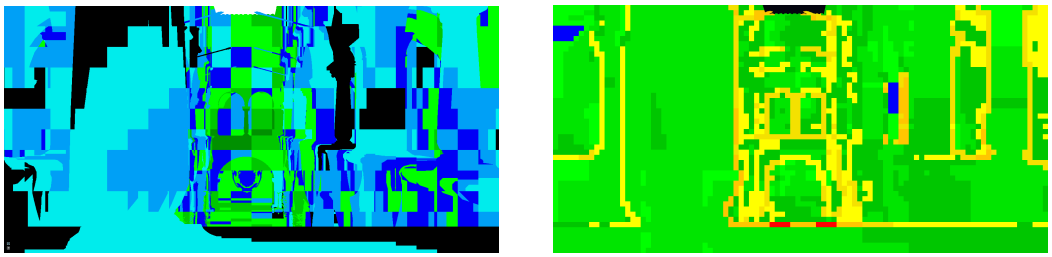
Figure 5.9: Comparison between AMD´s Forward+ tiled light culling demo using 2048 point lights and 2048 spot lights. Legend can be viewed in Figure 5.8.



(a) Clustered shading using 24x12x128-E cluster structure.

(b) Tiled shading using 96x48 tiled structure.

Figure 5.10: Comparison between AMD´s Forward+ tiled light culling demo using 2048 point lights and no spot lights. Legend can be viewed in Figure 5.8.



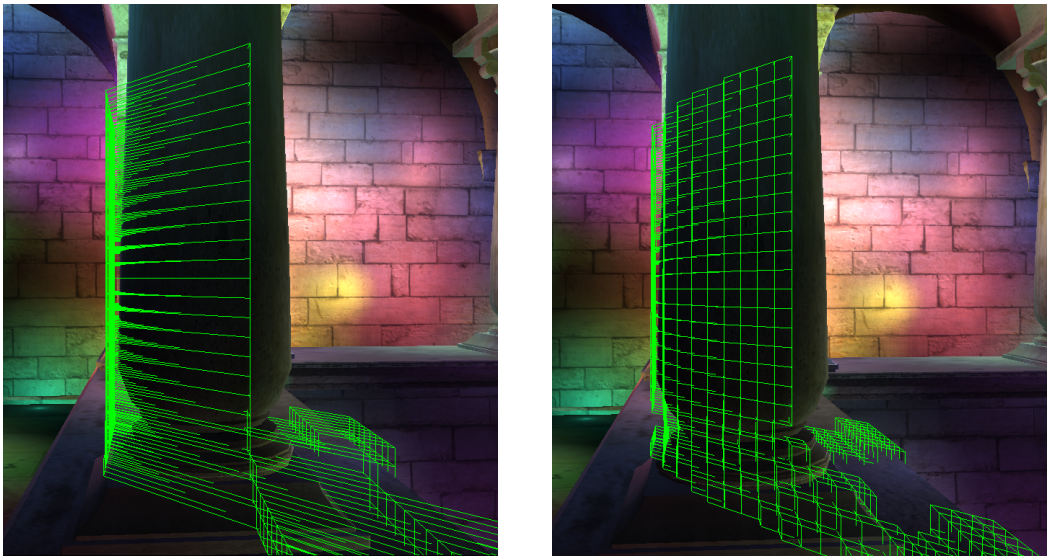
(a) Clustered shading using 24x12x128-E cluster structure.

(b) Tiled shading using 96x48 tiled structure.

Figure 5.11: Comparison between AMD´s Forward+ tiled light culling demo using no point lights and 2048 spot lights. Legend can be viewed in Figure 5.8.

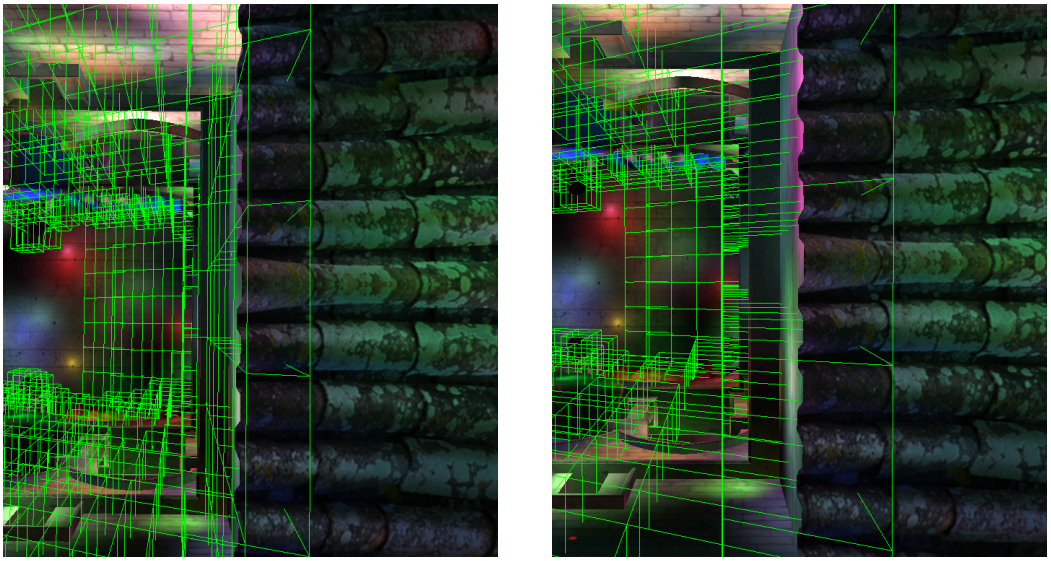
Figure 5.9 clearly shows that tiled light culling suffers from depth discontinuities and that at comparable performance the clustered light assignment performs better light assignment over all as well as having no depth discontinuities. The same is true when looking at the light types individually in Figures 5.10 and 5.11, but the spot light comparison also shows a significant reduction in lighting 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.

5.4 Depth distribution



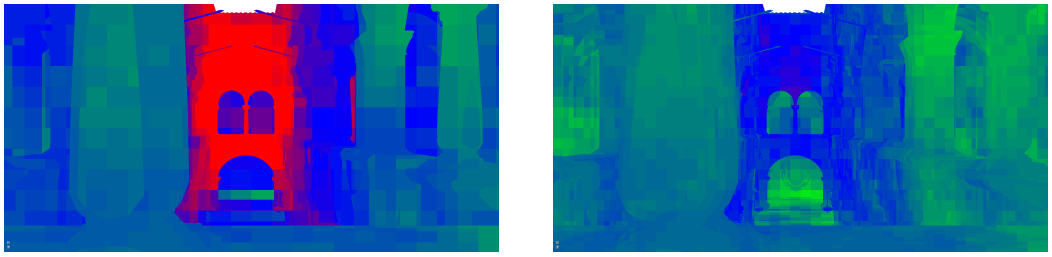
(a) Linear depth distribution. (b) Exponential depth distribution.

Figure 5.12: Two screen captures that show clusters close to the camera. Side view. Cluster structure size is 48x24x128.



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

Figure 5.13: Two screen captures that show clusters far from the camera. Top down view. Cluster structure size is 48x24x128.



(a) 24x12x32 cluster structure. (b) 48x24x128 cluster structure.

Figure 5.14: Two screen captures that show the number of lights used for shading each pixel. 4000 lights are used in this scene. Green is 1 light, blue is 35 lights and red is 70 or more lights. Values in between are interpolated colors.

Figure 5.14 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 5.12 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 creates a very good best case, but as can be seen in figure 5.13 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 X and Y dimensions and capturing many lights.

Figure 5.1 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 5.4 where both the 24x12x32 and 24x12x64 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 24x12x128-E and 48x24x32-E, where both contain the same amount of

clusters, in figure 5.2 it is evident that the more cube-like clusters in 24x12x128-E results in better performance. The performance increase gained when going from 24x12x128-L to 24x12x128-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 48x24x32-E as the clusters will still have a dominant slice depth compared to the x and y dimensions.

Chapter 6

Conclusion

This thesis has presented a novel technique for assigning arbitrarily shaped convex light types to clusters using conservative rasterization with good results and performance. By analyzing the results, the research question has been answered positively and the main goal of the thesis has been reached to the extent that all existing problems with current light culling techniques have been solved, at the cost of using more memory.

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 pre-pass. Looking at the result section it can be said 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 draw back of doing fine clustering; the memory usage. The total memory usage for the 48x24x128 cluster structure at 4000 lights adds up to 45.2MB while the 24x12x64 cluster structure uses 6.8MB. The larger cluster structure achieves 23.5% better shading performance at a cost of using 6.65 times more memory.

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 24x12x128 structure with 32 bit nodes to the 48x24x32 structure with 64 bit nodes results in 6.87MB and 18.2MB, respectively. In this implementation the 24x12x128 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. 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. This technique has been accepted and will be published in the book GPU Pro 7, due to come out in 2016.

Chapter 7

Future work

As the field of clustered shading is rather new and with the proposed technique in this thesis there are many unexplored areas.

7.1 Concave light volumes

Only convex light volumes are supported with the current implementation. This is not really a limiting factor for the technique, as all common light types are convex. There are however new visual experiences to be explored using concave light types, and also possible shading optimizations when using partially occluded point lights or nonuniform attenuation.

7.2 Cascaded clusters

As proposed in [OBP14], the use of a cascaded cluster structure can even out the cluster size differences between near clusters and far away clusters. This can possibly allow for a more even light assignment with clusters of similar size through out the view frustum. The restriction to using cascades in this implementation is that the number of tiles must be the same for every light in a shell pass. Solutions that could work would be to run the shell pass multiple times for different cascades. Alternatively the shell pass could be run at the highest cascade resolution and then added to the correct lower resolution cascade in the fill pass. The latter solution could be a suitable solution as the results show that the shell pass scales well with the number of tiles. The memory requirement for using the highest level cascade for all render targets would increase, but at the same time the light linked list would become smaller.

7.3 No geometry shader

The geometry shader is commonly known as a slow shader stage due to the data reads and writes to memory. It is used in this implementation only because the choice of an active render target must be done through it. There are extensions

in OpenGL from both Nvidia and AMD [Bro, Sel] that act as workarounds for the geometry shader requirement when needing to choose a render target. A feature like this will most likely make its way into the new DX12 API and allow render targets to be chosen from the vertex shader without any geometry shader emulation. This could prove to be a major optimization for this implementation.

7.4 Level of detail

As stated in chapter 5 there is a significant cost to processing vertices. With clusters being larger far away from the camera it would be a natural fit to use discrete steps of varied light shape resolution and using the correct one at a certain distance from the camera. It would definitely benefit the technique in terms of light assignment performance. There is however a downside to this proposed improvement; it introduces the need to sort the light list into multiple lists depending on the distance from the camera. What has to be tested in this scenario is how much the shading is affected and how quickly the lights can be sorted into the correct LOD level.

References

[And09] Johan Andersson. Parallel graphics in frostbite-current & future. *SIGGRAPH Course: Beyond Programmable Shading*, 2009.

[BE08] Christophe Balestra and Pål-Kristian Engstad. The technology of uncharted: Drake's fortune. In *Game Developer Conference*, 2008.

[Bro] Pat Brown. Nv_geometry_shader_passthrough. `https://developer.nvidia.com/sites/default/files/akamai/opengl/specs/GL_NV_geometry_shader_passthrough.txt`. [Online; accessed 18-May-2015].

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

[Fau14] Mark Fauconneau. Forward Clustered Shading. `https://software.intel.com/sites/default/files/managed/27/5e/Fast%20Foward%20Clustered%20Shading%20(siggraph%202014).pdf`, 2014. [Online; accessed 20-May-2015].

[HAMO05] Jon Hasselgren, Tomas Akenine-Möller, and Lennart Ohlsson. Conservative rasterization. *GPU Gems*, 2:677–690, 2005.

[HMY13] Takahiro Harada, Jay McKee, and Jason C Yang. Forward+: A step toward film-style shading in real time. *GPU Pro 4: Advanced Rendering Techniques*, 4:115, 2013.

[Lea14] Richard Leadbetter. The making of forza horizon 2. `http://www.eurogamer.net/articles/digitalfoundry-2014-the-making-of-forza-horizon-2`, 2014. [Online; accessed 18-May-2015].

[Mei10] Frank Meinl. The Atrium Sponza Palace, Dubrovnik. `http://www.crytek.com/cryengine/cryengine3/downloads`, 2010. [Online; accessed 20-May-2015].

[Mic] Microsoft. Floating-point rules. `https://msdn.microsoft.com/en-us/library/windows/desktop/jj218760(v=vs.85).aspx`. [Online; accessed 18-May-2015].

[MT05] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[OA11] Ola Olsson and Ulf Assarsson. Tiled shading. *Journal of Graphics, GPU, and Game Tools*, 15(4):235–251, 2011.

[OBA12a] 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*, pages 87–96. Eurographics Association, 2012.

[OBA12b] Ola Olsson, Markus Billeter, and Ulf Assarsson. Tiled and clustered forward shading. In *SIGGRAPH '12: ACM SIGGRAPH 2012 Talks*, New York, NY, USA, 2012. ACM.

[OBP14] Ola Olsson, Markus Billeter, and Emil Persson. Efficient real-time shading with many lights. In *SIGGRAPH Asia 2014 Courses*, SA '14, pages 11:1–11:310, New York, NY, USA, 2014. ACM.

[PO13] Emil Persson and Ola Olsson. Practical clustered deferred and forward shading. SIGGRAPH Course: Advances in Real-Time Rendering in Games, 2013.

[Sel] Graham Sellers. Amd_vertex_shader_layer. `https://www.opengl.org/registry/specs/AMD/vertex_shader_layer.txt`. [Online; accessed 18-May-2015].

[Sto15] Jon Story. Advanced visual effects with directx 11 & 12: Sparse fluid simulation and hybrid ray-traced shadows for directx 11 & 12. Game Developer Conference, 2015. [Online; accessed 18-May-2015].

[Swo09] Matt Swoboda. Deferred lighting and post processing on playstation 3. In *Game Developer Conference*, 2009.

[Tho15] Gareth Thomas. Advanced visual effects with directx 11 & 12: Advancements in tile-based compute rendering. Game Developer Conference, 2015. [Online; accessed 18-May-2015].

[YHGT10] Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. In *Computer Graphics Forum*, volume 29, pages 1297–1304. Wiley Online Library, 2010.

[ZCEP07] Long Zhang, Wei Chen, David S Ebert, and Qunsheng Peng. Conservative voxelization. *The Visual Computer*, 23(9-11):783–792, 2007.

Appendix A

Complete shell pass pixel shader code

```
1  #include "ShaderCommon.h"
2  #include "LightHelper.h"
3
4  struct PS_INPUT
5  {
6      float4 pos : SV_Position;
7      nointerpolation uint4 vertPos : VERTPOS;
8      nointerpolation uint vertPosv2z : VERTPOSZ;
9      nointerpolation uint rtArrayIndex : SV_RenderTargetArrayIndex;
10 };
11
12 bool linesegment_vs_plane(float3 p0, float3 p1, float3 pn, out float
       lerp_val)
13 {
14     float3 u = p1 - p0;
15
16     float D = dot(pn, u);
17     float N = -dot(pn, p0);
18
19     //IEEE 754-2008 in HLSL tricks
20     lerp_val = N / D; //Division by zero returns -/+ INF (except for
       0/0 which returns NaN)
21     return !(lerp_val != saturate(lerp_val)); //saturate NaN returns
       0 and saturate -/+INF returns correct value //The comparison NE,
       when either or both operands is NaN returns TRUE.
22 }
23
24 bool is_in_xslice(float3 top_plane, float3 bottom_plane, float3
      vert_point)
25 {
26     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);
27 }
28
29 bool is_in_yslice(float3 left_plane, float3 right_plane, float3
      vert_point)
30 {
```

```
31      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 );
32  }
33
34  bool ray_vs_triangle(float3 ray_dir, float3 vert0, float3 vert1,
        float3 vert2, out float z_pos)
35  {
36      float3 e1 = vert1 - vert0;
37      float3 e2 = vert2 - vert0;
38      float3 q = cross(ray_dir, e2);
39      float a = dot(e1, q);
40
41      if(a > -0.000001f && a < 0.000001f)
42          return false;
43
44      float f = 1.0f / a;
45      float u = f * dot(-vert0, q);
46
47      if(u != saturate(u))
48          return false;
49
50      float3 r = cross(-vert0, e1);
51      float v = f * dot(ray_dir, r);
52
53      if(v < 0.0f || (u + v) > 1.0f)
54          return false;
55
56      z_pos = f * dot(e2, r) * ray_dir.z;
57
58      return true;
59  }
60
61  float2 min_max(float2 depth_min_max, in float depth)
62  {
63      depth_min_max.x = min(depth_min_max.x, depth);
64      depth_min_max.y = max(depth_min_max.y, depth);
65
66      return depth_min_max;
67  }
68
69  float2 main(PS_INPUT input, bool is_front_face : SV_IsFrontFace) :
        SV_TARGET
70  {
71      float3 vert0 = f16tof32(uint3((input.vertPos.x >> 16), (input.
        vertPos.x & 0xFFFF), (input.vertPos.y >> 16)));
72      float3 vert1 = f16tof32(uint3((input.vertPos.y & 0xFFFF), (input
        .vertPos.z >> 16), (input.vertPos.z & 0xFFFF)));
73      float3 vert2 = f16tof32(uint3((input.vertPos.w >> 16), (input.
        vertPos.w & 0xFFFF), (input.vertPosv2z & 0xFFFF)));
74
```

```
75      float2 depthMinMax = float2(FARZ, 0.0f);
76
77      const float ey = tan(PI / 4.0f * 0.5f);
78      const float ex = ey * WINDOW_WIDTH / WINDOW_HEIGHT;
79
80      const float3 left_plane     =  normalize(float3(1.0f,  0.0f,
        (1.0f - 2.0f * floor(input.pos.x) / CLUSTERSX) * ex));
81      const float3 right_plane    = -normalize(float3(1.0f,  0.0f,
        (1.0f - 2.0f * floor(input.pos.x + 1.0f) / CLUSTERSX) * ex));
82      const float3 top_plane      =  normalize(float3(0.0f, -1.0f,
        (1.0f - 2.0f * floor(input.pos.y) / CLUSTERSY) * ey));
83      const float3 bottom_plane   = -normalize(float3(0.0f, -1.0f,
        (1.0f - 2.0f * floor(input.pos.y + 1.0f) / CLUSTERSY) * ey));
84
85
86      //Case where the min/max depth is one of the corners of the tile
87      //cross product of plane normals return the ray direction
        through the corner. All planes and rays go through (0,0,0)
88      float z_pos;
89      depthMinMax = ray_vs_triangle(cross(top_plane, left_plane),
        vert0, vert1, vert2, z_pos) ? min_max(depthMinMax, z_pos) :
        depthMinMax;
90      depthMinMax = ray_vs_triangle(cross(top_plane, right_plane),
        vert0, vert1, vert2, z_pos) ? min_max(depthMinMax, z_pos) :
        depthMinMax;
91      depthMinMax = ray_vs_triangle(cross(right_plane, bottom_plane),
        vert0, vert1, vert2, z_pos) ? min_max(depthMinMax, z_pos) :
        depthMinMax;
92      depthMinMax = ray_vs_triangle(cross(bottom_plane, left_plane),
        vert0, vert1, vert2, z_pos) ? min_max(depthMinMax, z_pos) :
        depthMinMax;
93
94      //Case where a vertex is the min/max depth of the tile
95      //Check if vertex is inside all four planes
96      depthMinMax = (is_in_xslice(top_plane, bottom_plane, vert0) &&
        is_in_yslice(left_plane, right_plane, vert0)) ? min_max(
        depthMinMax, vert0.z) : depthMinMax;
97      depthMinMax = (is_in_xslice(top_plane, bottom_plane, vert1) &&
        is_in_yslice(left_plane, right_plane, vert1)) ? min_max(
        depthMinMax, vert1.z) : depthMinMax;
98      depthMinMax = (is_in_xslice(top_plane, bottom_plane, vert2) &&
        is_in_yslice(left_plane, right_plane, vert2)) ? min_max(
        depthMinMax, vert2.z) : depthMinMax;
99
100     //Case where a vertex edge intersects a tile side is the min/max
        depth
101     float lerp_val;
102
103     //Left side
104     depthMinMax = is_in_xslice(top_plane, bottom_plane,
        linesegment_vs_plane(vert0, vert1, left_plane, lerp_val) ? lerp(
```

```
    vert0, vert1, lerp_val) : float3(0, 0, -1)) ? min_max(depthMinMax
    , lerp(vert0.z, vert1.z, lerp_val)) : depthMinMax;
105  depthMinMax = is_in_xslice(top_plane, bottom_plane,
    linesegment_vs_plane(vert1, vert2, left_plane, lerp_val) ? lerp(
    vert1, vert2, lerp_val) : float3(0, 0, -1)) ? min_max(depthMinMax
    , lerp(vert1.z, vert2.z, lerp_val)) : depthMinMax;
106  depthMinMax = is_in_xslice(top_plane, bottom_plane,
    linesegment_vs_plane(vert2, vert0, left_plane, lerp_val) ? lerp(
    vert2, vert0, lerp_val) : float3(0, 0, -1)) ? min_max(depthMinMax
    , lerp(vert2.z, vert0.z, lerp_val)) : depthMinMax;
107
108  //Right side
109  depthMinMax = is_in_xslice(top_plane, bottom_plane,
    linesegment_vs_plane(vert0, vert1, right_plane, lerp_val) ? lerp(
    vert0, vert1, lerp_val) : float3(0, 0, -1)) ? min_max(depthMinMax
    , lerp(vert0.z, vert1.z, lerp_val)) : depthMinMax;
110  depthMinMax = is_in_xslice(top_plane, bottom_plane,
    linesegment_vs_plane(vert1, vert2, right_plane, lerp_val) ? lerp(
    vert1, vert2, lerp_val) : float3(0, 0, -1)) ? min_max(depthMinMax
    , lerp(vert1.z, vert2.z, lerp_val)) : depthMinMax;
111  depthMinMax = is_in_xslice(top_plane, bottom_plane,
    linesegment_vs_plane(vert2, vert0, right_plane, lerp_val) ? lerp(
    vert2, vert0, lerp_val) : float3(0, 0, -1)) ? min_max(depthMinMax
    , lerp(vert2.z, vert0.z, lerp_val)) : depthMinMax;
112
113  //Bottom side
114  depthMinMax = is_in_yslice(left_plane, right_plane,
    linesegment_vs_plane(vert0, vert1, bottom_plane, lerp_val) ? lerp
    (vert0, vert1, lerp_val) : float3(0, 0, -1)) ? min_max(
    depthMinMax, lerp(vert0.z, vert1.z, lerp_val)) : depthMinMax;
115  depthMinMax = is_in_yslice(left_plane, right_plane,
    linesegment_vs_plane(vert1, vert2, bottom_plane, lerp_val) ? lerp
    (vert1, vert2, lerp_val) : float3(0, 0, -1)) ? min_max(
    depthMinMax, lerp(vert1.z, vert2.z, lerp_val)) : depthMinMax;
116  depthMinMax = is_in_yslice(left_plane, right_plane,
    linesegment_vs_plane(vert2, vert0, bottom_plane, lerp_val) ? lerp
    (vert2, vert0, lerp_val) : float3(0, 0, -1)) ? min_max(
    depthMinMax, lerp(vert2.z, vert0.z, lerp_val)) : depthMinMax;
117
118  //Top side
119  depthMinMax = is_in_yslice(left_plane, right_plane,
    linesegment_vs_plane(vert0, vert1, top_plane, lerp_val) ? lerp(
    vert0, vert1, lerp_val) : float3(0, 0, -1)) ? min_max(depthMinMax
    , lerp(vert0.z, vert1.z, lerp_val)) : depthMinMax;
120  depthMinMax = is_in_yslice(left_plane, right_plane,
    linesegment_vs_plane(vert1, vert2, top_plane, lerp_val) ? lerp(
    vert1, vert2, lerp_val) : float3(0, 0, -1)) ? min_max(depthMinMax
    , lerp(vert1.z, vert2.z, lerp_val)) : depthMinMax;
121  depthMinMax = is_in_yslice(left_plane, right_plane,
    linesegment_vs_plane(vert2, vert0, top_plane, lerp_val) ? lerp(
    vert2, vert0, lerp_val) : float3(0, 0, -1)) ? min_max(depthMinMax
```

```
    , lerp(vert2.z, vert0.z, lerp_val)) : depthMinMax;
122
123 #ifdef LINEAR_DEPTH_DIST
124     return is_front_face ? float2(1.0f, (CLUSTERSZ - 1.0f) / 255.0f
    - floor(max(CLUSTERSZ * (depthMinMax.y / FARZ), 0.0f)) / 255.0f)
125     : float2(floor(max(CLUSTERSZ * (depthMinMax.x / FARZ), 0.0f)) /
    255.0f, 1.0f);
126 #else
127     // Look up the light list for the cluster
128     const float min_depth = log2(NEAR_CLUST);
129     const float max_depth = log2(FARZ);
130
131     const float scale = 1.0f / (max_depth - min_depth) * (CLUSTERSZ
    - 1.0f);
132     const float bias = 1.0f - min_depth * scale;
133
134     //Wrong winding order on triangles, should be the other way
    around.
135     return is_front_face ? float2(1.0f, (CLUSTERSZ - 1.0f) / 255.0f
    - floor(max(log2(depthMinMax.y) * scale + bias, 0.0f)) / 255.0f)
136     : float2(floor(max(log2(depthMinMax.x) * scale + bias, 0.0f)) /
    255.0f, 1.0f);
137 #endif
138 }
```

Listing A.1: Complete shell pass pixel shader code.

Appendix B

Complete shell pass geometry shader code

```
1  cbuffer CameraBuffer : register(b0)
2  {
3      float4x4 viewProjMat;
4      float4x4 viewMat;
5      float4x4 projMat;
6  }
7
8  struct GS_OUTPUT
9  {
10     float4 pos : SV_Position;
11     nointerpolation uint4 vertPos : VERTPOS;
12     nointerpolation uint vertPosv2z : VERTPOSZ;
13     nointerpolation uint rtArrayIndex : SV_RenderTargetArrayIndex;
14 };
15
16 struct GS_INPUT
17 {
18     float3 viewPos : VIEW_POS;
19     uint lightID : LIGHTID;
20 };
21
22 [maxvertexcount(3)]
23 void main(triangle GS_INPUT input[3], inout TriangleStream<GS_OUTPUT
       > stream)
24 {
25     GS_OUTPUT out_struct;
26
27     //Set render target array index to the lightID(instanceID from
       vertex shader)
28     out_struct.rtArrayIndex = input[0].lightID;
29
30     //Send packed viewspace pos
31     uint3 pack_value0 = f32tof16(float3(input[0].viewPos.xy, -input
       [0].viewPos.z));
32     uint3 pack_value1 = f32tof16(float3(input[1].viewPos.xy, -input
       [1].viewPos.z));
33     uint3 pack_value2 = f32tof16(float3(input[2].viewPos.xy, -input
       [2].viewPos.z));
34
```

```
35      out_struct.vertPos.x = (pack_value0.x << 16) | (pack_value0.y);
36      out_struct.vertPos.y = (pack_value0.z << 16) | (pack_value1.x);
37      out_struct.vertPos.z = (pack_value1.y << 16) | (pack_value1.z);
38      out_struct.vertPos.w = (pack_value2.x << 16) | (pack_value2.y);
39      out_struct.vertPosv2z = (pack_value2.z);
40
41      out_struct.pos = mul(projMat, float4(input[0].viewPos, 1.0f));
42      stream.Append(out_struct);
43      out_struct.pos = mul(projMat, float4(input[1].viewPos, 1.0f));
44      stream.Append(out_struct);
45      out_struct.pos = mul(projMat, float4(input[2].viewPos, 1.0f));
46      stream.Append(out_struct);
47  }
```

Listing B.1: Complete shell pass geometry shader code.

Appendix C

Raw result data

Clusts	24x12x32-E	24x12x32-L	24x12x64-E	24x12x64-L	24x12x128-E	24x12x128-L	48x24x32-E	48x24x32-L	48x24x64-E	48x24x64-L	48x24x128-E	48x24x128-L
400	21809	18413	38137	31144	70647	56576	75317	58500	131777	98409	244977	177697
1000	51735	46804	89976	79613	166084	145125	176991	148540	308398	250849	570954	454694
2000	111352	98203	194582	167294	360630	305231	383065	312243	672104	527930	1249466	958518
4000	224822	196729	393160	335930	730033	613473	774062	625930	1359897	1061304	2531817	1929030

Point shell time	24x12x32-E	24x12x32-L	24x12x64-E	24x12x64-L	24x12x128-E	24x12x128-L	48x24x32-E	48x24x32-L	48x24x64-E	48x24x64-L	48x24x128-E	48x24x128-L
400	0.0523	0.0563	0.0523	0.0564	0.0523	0.0566	0.0537	0.0577	0.0528	0.0570	0.0528	0.0568
1000	0.1160	0.1153	0.1160	0.1151	0.1159	0.1152	0.1179	0.1172	0.1159	0.1155	0.1160	0.1153
2000	0.2240	0.2223	0.2238	0.2222	0.2238	0.2223	0.2314	0.2296	0.2297	0.2279	0.2295	0.2277
4000	0.4270	0.4236	0.4268	0.4237	0.4269	0.4236	0.4397	0.4362	0.4379	0.4346	0.4378	0.4341

Point fill time	24x12x32-E	24x12x32-L	24x12x64-E	24x12x64-L	24x12x128-E	24x12x128-L	48x24x32-E	48x24x32-L	48x24x64-E	48x24x64-L	48x24x128-E	48x24x128-L
400	0.0224	0.0172	0.0327	0.0203	0.0505	0.0251	0.0260	0.0246	0.0387	0.0289	0.0676	0.0411
1000	0.0235	0.0224	0.0340	0.0273	0.0554	0.0362	0.0379	0.0392	0.0477	0.0475	0.0756	0.0678
2000	0.0309	0.0310	0.0417	0.0388	0.0658	0.0544	0.0643	0.0649	0.0816	0.0794	0.1224	0.1144
4000	0.0494	0.0493	0.0656	0.0632	0.0969	0.0911	0.1148	0.1165	0.1458	0.1439	0.2266	0.2099

Spot shell time	24x12x32-E	24x12x32-L	24x12x64-E	24x12x64-L	24x12x128-E	24x12x128-L	48x24x32-E	48x24x32-L	48x24x64-E	48x24x64-L	48x24x128-E	48x24x128-L
400	0.0337	0.0355	0.0336	0.0354	0.0334	0.0353	0.0424	0.0443	0.0425	0.0445	0.0429	0.0450
1000	0.0632	0.0622	0.0630	0.0619	0.0642	0.0631	0.0821	0.0813	0.0821	0.0813	0.0820	0.0813
2000	0.1172	0.1162	0.1168	0.1155	0.1172	0.1161	0.1517	0.1501	0.1519	0.1502	0.1516	0.1500
4000	0.2141	0.2119	0.2139	0.2118	0.2141	0.2119	0.2743	0.2713	0.2745	0.2715	0.2742	0.2712

Spot fill time	24x12x32-E	24x12x32-L	24x12x64-E	24x12x64-L	24x12x128-E	24x12x128-L	48x24x32-E	48x24x32-L	48x24x64-E	48x24x64-L	48x24x128-E	48x24x128-L
400	0.0207	0.0203	0.0286	0.0265	0.0453	0.0383	0.0293	0.0303	0.0423	0.0417	0.0743	0.0694
1000	0.0297	0.0272	0.0436	0.0372	0.0740	0.0577	0.0493	0.0511	0.0696	0.0713	0.1300	0.1201
2000	0.0426	0.0424	0.0640	0.0612	0.1061	0.0986	0.0848	0.0885	0.1252	0.1251	0.2354	0.2195
4000	0.0653	0.0674	0.0975	0.0989	0.1657	0.1629	0.1546	0.1591	0.2211	0.2249	0.4266	0.3948

LA	24x12x32-E	24x12x32-L	24x12x64-E	24x12x64-L	24x12x128-E	24x12x128-L	48x24x32-E	48x24x32-L	48x24x64-E	48x24x64-L	48x24x128-E	48x24x128-L
400	0.1291	0.1293	0.1472	0.1386	0.1815	0.1553	0.1514	0.1569	0.1762	0.1721	0.2376	0.2122
1000	0.2323	0.2271	0.2566	0.2415	0.3095	0.2722	0.2872	0.2888	0.3154	0.3156	0.4036	0.3845
2000	0.4147	0.4119	0.4463	0.4377	0.5129	0.4915	0.5321	0.5331	0.5883	0.5827	0.7390	0.7117
4000	0.7558	0.7523	0.8039	0.7975	0.9036	0.8893	0.9834	0.9831	1.0792	1.0749	1.3652	1.3100

Shading	24x12x32-E	24x12x32-L	24x12x64-E	24x12x64-L	24x12x128-E	24x12x128-L	48x24x32-E	48x24x32-L	48x24x64-E	48x24x64-L	48x24x128-E	48x24x128-L
400	0.5105	0.5819	0.4600	0.5861	0.4261	0.5323	0.4589	0.5191	0.4172	0.5253	0.3925	0.4565
1000	0.8099	0.7959	0.6982	0.7421	0.6426	0.6897	0.6928	0.6932	0.6043	0.6674	0.5634	0.5989
2000	1.3857	1.3613	1.1820	1.1866	1.0722	1.0984	1.1468	1.1703	0.9858	1.0297	0.9049	0.9365
4000	2.6166	2.5162	2.2099	2.1890	2.0194	2.0310	2.1567	2.1579	1.8402	1.8747	1.6907	1.7239

Total	24x12x32-E	24x12x32-L	24x12x64-E	24x12x64-L	24x12x128-E	24x12x128-L	48x24x32-E	48x24x32-L	48x24x64-E	48x24x64-L	48x24x128-E	48x24x128-L
400	0.6396	0.7112	0.6073	0.7247	0.6076	0.6876	0.6103	0.6761	0.5934	0.6974	0.6301	0.6687
1000	1.0422	1.0230	0.9549	0.9837	0.9521	0.9618	0.9800	0.9820	0.9197	0.9830	0.9670	0.9835
2000	1.8004	1.7732	1.6283	1.6243	1.5850	1.5899	1.6789	1.7034	1.5741	1.6124	1.6439	1.6481
4000	3.3724	3.2685	3.0138	2.9865	2.9230	2.9203	3.1401	3.1410	2.9194	2.9496	3.0558	3.0339

Table C.1: Timings in milliseconds. Y axis are number of lights and X axis is the different cluster structures.