Hashed Shading

Martinus Wilhelmus Tegelaers
KULeuven
mwtegelaers@gmail.com

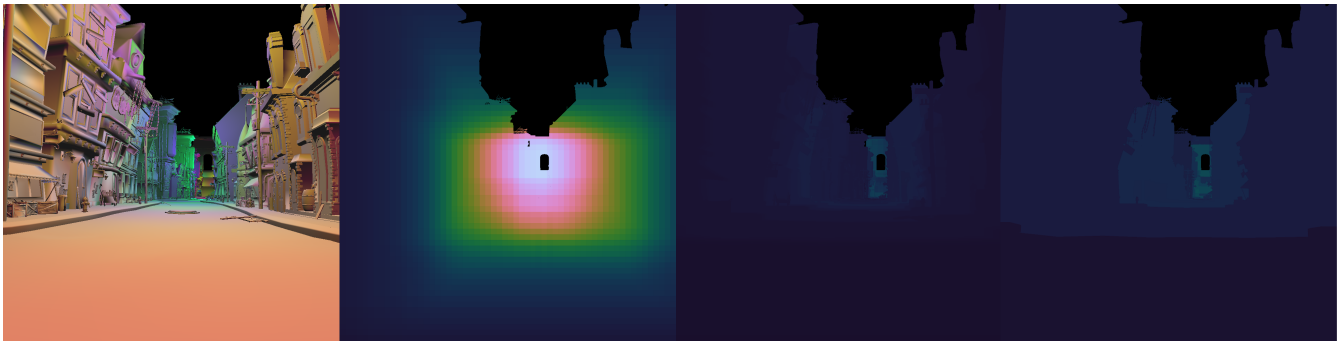


Figure 1: The number of light calculations required to shade the image left, for respectively Tiled, Clustered and Hashed Shading.

ABSTRACT

This paper introduces the Hashed Shading algorithm, a light assignment algorithm for forward and deferred shading. It uses a subdivision independent from the view frustum in order to reuse data structures between frames. The scene is subdivided into cubes of a specified size and represented using a linkless octree to store the data efficiently in memory and allow for a fast retrieval of relevant lights during shading. The performance of Hashed Shading is compared to both Tiled and Clustered Shading. Forward Hashed Shading reduces the number of light calculations and the execution time by a factor of two compared to Forward Tiled Shading. It achieves a similar reduction in the number of light calculations as Clustered Shading. Furthermore the Hashed Shading algorithm scales slightly better in regards to resolution than Tiled and Clustered Shading due to the camera-independent subdivision. Currently Hashed Shading does not support dynamic lights, and requires significant memory to store the linkless octree. Solutions for these issues are proposed but have not been implemented.

KEYWORDS

real-time graphics, light assignment, shading

1 INTRODUCTION

In order to push the limits of visual fidelity in modern games, the complexity of shaders and the number of lights keeps increasing. These complex calculations require significant processing time. To facilitate frames being rendered in real-time, the amount of unnecessary light calculations need to be kept to a minimum. There are several approaches to reduce this amount of work, while preserving the accuracy of the shading. One of such approaches is Light Assignment. This technique reduces the number of light calculations per pixel by excluding lights that do not influence it. This is possible due to the finite representation of lights generally used within real-time applications. Ideally per pixel only lights of which the corresponding light volume contains the pixel are evaluated. This reduces the number of light calculations to the bare minimum.

In recent games two light assignment algorithms are used commonly, Tiled Shading[OA11] and Clustered Shading[OBA12]. Both algorithms subdivide the view frustum. For each sub frustum the set of lights which overlap with the volume of the sub frustum are calculated and subsequently retrieved during shading to the reduce the number of lights which need to be evaluated.

This direct dependency on the view frustum requires the data structures of the two light assignment algorithms to be recalculated per frame. Due to the high frame rate at which images in real-time applications are generated, changes in the scene are generally small and local. Significant parts of the generated data structures could potentially be reused in order to reduce computation time and improve performance.

The goal of this paper is to explore whether reusing the generated data structures does indeed yield a performant algorithm. For this purpose the Hashed Shading algorithm is introduced. This algorithm makes use of a view frustum independent subdivision of the scene space, such that a change in camera position or orientation does not warrant a complete rebuild of the data structures. The subdivision itself is build upon the octree data structure, such that the scene space is represented efficiently in memory while still being precise.

In order to validate whether data structures persistent between frames can potentially achieve better performance than current light assignment data structures, the Hashed Shading algorithm needs to achieve at least a similar execution time speed up and light calculation reduction as Tiled and Clustered Shading.

2 RELATED WORK

2.1 Deferred Shading

A first step into reducing the number of light calculations is decoupling geometry and shading complexity. In the modern graphical pipeline all geometry in the view frustum generates samples to be shaded, regardless of whether these are visible or occluded. These samples are all shaded and only after executing the light calculations the visibility of pixels is determined in the per-pixel operations. In order to shade only the visible samples, the visibility has to be determined before shading. The concept to achieve this was introduced in hardware design in 1988[DWS+88]. A general purpose strategy to achieve this deferred shading was introduced in 1990, making use of so called geometry buffers or GBuffers[ST90]. The idea is to execute the render pipeline twice per frame. In the first pass all attributes necessary to perform the shading computation are rendered into screen size textures, the GBuffers. In the second pass one sample per pixel is generated and for these samples the values are retrieved from the GBuffer and used to shade the pixel. This way the visibility is determined for the attributes in the in the per-pixel operations of the first pass. Then only the attributes of the visible samples are retrieved in the second pass.

A first light assignment approach is to rasterise the light volumes during the second pass. The shading contributions per light are then only calculated for the samples generated by its light volume. This reduces the total number of light calculations, as not every light will be evaluated for every pixel. This optimisation is called the stencil-optimisation[AA03]. This approach has two downsides. It requires a significant number of memory accesses, as per generated sample the shading data has to be retrieved from the GBuffers. Also, the summation of shading contributions happens in a framebuffer. This reduces the precision as the framebuffer's precision generally does not have as much bits as the GPU registers.

2.2 Tiled Shading

Tiled Shading[OA11] was designed to alleviate the memory bandwidth bottleneck of Deferred Shading with the stencil-optimisation. The underlying idea is to calculate which lights effect which pixels before shading, instead of during. This is achieved by subdividing the screen space into tiles of $n \times n$ pixels. For each tile the light indices of the lights which projected light volumes overlap with the tile, are stored. The tiles can subsequently be used to retrieve the set of relevant lights during shading. This can be done by determining the tile in which a pixel falls, and retrieving the light indices associated with this tile. Tiles can be constructed by projecting the light volumes on the view port and determining with which tiles it overlaps. To each of the tiles with which the light overlaps, the light index of the light is added.

A further optimisation can be done by calculating the minimum and maximum z-values for each tile. These can then be used to reject any light which falls outside of these boundaries, thus reducing the number of lights associated with a tile.

This algorithm has been implemented in several game engines and games[BE08, Swo09, BE08], including Unreal[Kar13], frostbite[And09, Mag11] and unity[PD14].

2.3 Clustered Shading

Clustered Shading[OBA12] extends Tiled Shading by subdividing the volumes associated with the tiles further. The tile volumes are subdivided with regards to the camera's z-axis to obtain cube-like sub frustums. These sub frustums can be further partitioned based on attributes such as the normals. These higher dimension tiles are called clusters.

The clusters are constructed by first determining the set of cluster volumes which contain visible geometry. This is done by transforming the depth buffer into a cluster buffer by binning the z-values. This cluster buffer is sorted and compacted locally per tile to extract the set of unique clusters. At the same time a mapping is build, such that for each pixel the corresponding cluster can be looked up. To each of the unique clusters the indices of overlapping lights are added.

During the shading step for each pixel the set of relevant lights is determined by first looking up the corresponding cluster associated with a pixel and then retrieving the set of lights overlapping with this cluster. The shading contributions of each of these lights are evaluated similarly to Tiled shading.

This subdivision of the view frustum is more fine-grained than that of Tiled Shading, and thus reduces the number of light calculations per pixel to a greater extend. Clustered Shading is used in several modern games, including in the Avalance game engine, on which games as Just Cause 2 and 3[Per13], Doom 4[Tat16].

2.4 Perfect Spatial Hashing

In order to efficiently subdivide the scene space without using a large amount of memory, an octree data structure is used. A straightforward implementation using pointers would require a large number of control structures. Such control structures would degrade the performance of the data structure on the GPU[HA11]. A pointer based data structure would therefore be infeasible for a light assignment algorithm. In Hashed Shading a Linkless Octree[CJC+09] is used. This data structure builds upon perfect spatial hash functions[LH06].

A hash function h can be used to map sparse data D on a compact memory table H:

$$D(\mathbf{p}) = H[h(\mathbf{p})]$$

When the hash function h maps none of the keys $\mathbf{p}$ onto the same address it is collision-free and called perfect. Perfect hash functions are ideal for GPU applications. They allow sparse data to be represented efficiently in memory and each of the elements can be retrieved from memory without needing any control structures.

Perfect spatial hashing[LH06] introduces a technique to construct memory efficient perfect hash functions for higher dimensions which are spatially coherent. This is done by using two simple imperfect hash functions h_0 and h_1, and one offset hash table Φ. The perfect hash function h can then be defined as

$$h(\mathbf{p}) = h_0(\mathbf{p}) + \Phi[h_1(\mathbf{p})]$$

where h_0 and h_1 are simple hash functions defined as

$$h_0 : \mathbf{p} \mapsto \mathbf{p} \bmod \dot{m}$$
$$h_1 : \mathbf{p} \mapsto \mathbf{p} \bmod \dot{r}$$

where $\dot{m}$ is the size of a single dimension of the hash table H and $\dot{r}$ is the size of a single dimension of the offset hash table Φ.

The underlying idea is that colliding keys in h_0 do not collide in h_1, thus by adding an offset defined in Φ to the calculated address of h_0 the collisions within h are dissolved.

This representation makes it possible to store sparse three dimensional data compactly in memory. The linkless octree data structure will be build upon these perfect spatial hash functions.

2.5 Linkless Octree

The linkless octree[CJC+09] is a GPU efficient octree implementation. Each layer of the linkless octree implementation is represented by a perfect spatial hash function, therefore removing the need for pointers between nodes.

An octree contains a number of layers, each containing nodes of a similar size. Because each node in a single layer has the same size, the position of a node can be represented by a three dimensional integer vector. Due to the structure of the octree, the nodes of each layer are generally sparsely distributed over the complete scene space. Each layer can thus be compactly represented by a perfect spatial hash function, where the keys are the integer positions of the nodes, and the data is a description of the node at that position. This description needs to encode whether an octree node is a branch or leaf node, and whether it contains data or not. Therefore each node can be described by two bits. Due to memory considerations eight sibling nodes in a layer $l + 1$ are represented directly by the branch node in layer l. This branch node has two 8-bit integers, which describe each of the child nodes of this branch node directly.

The data associated with the octree nodes needs to be stored as well. There are two options for doing so. The straightforward solution is to reserve the space necessary for each data element associated with a node directly in the hash table associated with a layer. This is a valid approach when the data is small or when the majority of the nodes contains data. However, when the data is large, or only a small subset of the nodes actually contains data, this solution would lead to unacceptable memory requirements. This can be remedied by constructing a second spatial hash function per layer which will hold the data elements. Each data element associated with a nonempty node in a layer is stored within this second spatial hash function. Data can then be retrieved from the octree by first descending into the octree, determining for each layer the type of octree node encountered. The leaf nodes specifies whether data is associated with a query. If it is, then this data can be retrieved from the second spatial hash function of the leaf node's layer.

3 HASHED SHADING ALGORITHM

The goal of Hashed Shading is to improve the performance compared to Tiled and Clustered Shading by reusing data structures between frames. The data structures can be reused because the subdivision is independent from the view frustum. Thus the data structures do not need to be rebuild when the camera position or orientation changes. The subdivision of the scene space is done with an octree representation such that it is both precise and memory efficient.

The following data structures are used within Hashed Shading:

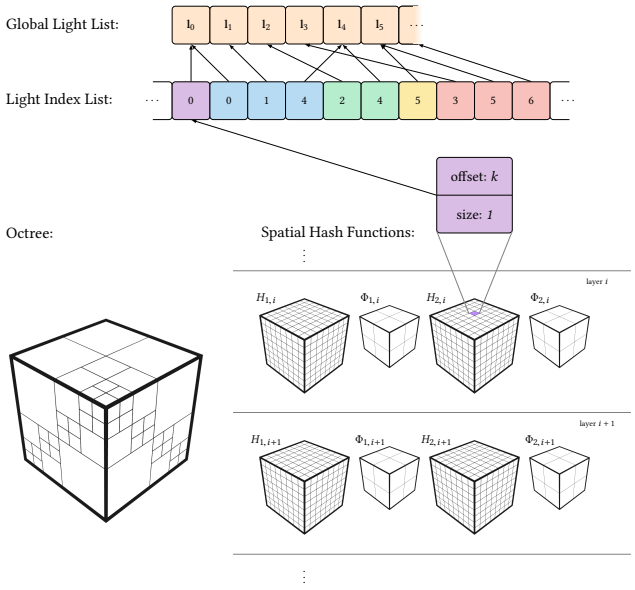


Figure 2: Overview of the data structures of Hashed Shading.

- Linkless Octree
- Light Index List
- Global Light List

These data structures are based on those of Tiled and Clustered Shading. The Global Light List contains all the lights within the scene. The Light Index List contains indices linking to the Global Light List. For each leaf node the set of indices specifying the relevant lights is added to this Light Index List. The set of relevant lights for each nonempty node within the linkless octree can thus be specified with an offset and length marking a subset within the Light Index List. This is illustrated in figure 2.

In order to construct these data structures the following steps need to be executed

- Define the subdivision of space.
- Calculate the influence of each light on this subdivision.
- Combine the influences into a single octree describing the whole scene.
- Construct the linkless octree and Light Index List based on the scene octree.

The first step requires the origin and size of the octree to be set. Once these values are specified it is possible to define the influence of each light. The combination of all these influences leads to an octree representation of the whole scene. In order to use this octree representation for light assignment, it needs to be transformed into a linkless octree and a Light Index List. Once all these steps have been completed, the constructed data structures can be used to speed up the shading step of the rendering pipeline. Each of these steps will be explained in more detail in the following sections.

3.1 Specification of the Octree

To construct the octree a minimum bounding box containing all samples generated at any point during the simulation needs to

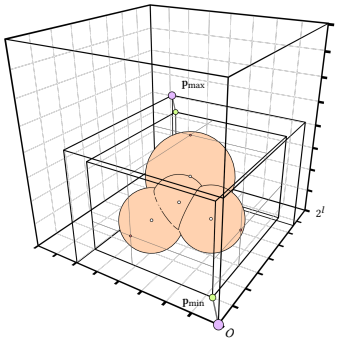


Figure 3: The subdivision of the scene.

be defined. The minimum and maximum point that define the bounding box can either be set by the developer or be constructed based on the light volumes.

In order to ensure no values are generated outside the bounding box due to rounding errors. a small offset is added to the points.

Once the bounding box has been defined, the size of the octree can be calculated. The developer is responsible for setting the size of the nodes in the deepest layer of the octree. Based on this node size, l_o, the size of the octree can be calculated. Assuming that d_i is the length of the longest face of the bounding box, the size of the octree l_o can be defined for the smallest possible k such that holds

$$l_o = l_n \cdot 2^k \geq d_i$$

where k is the number of layers within the octree. This is illustrated in figure 3. With the origin and size of the octree defined it is possible to assign the lights to this subdivision.

3.2 Influence of Lights

Within this paper only point lights are considered. Extending this algorithm to other light types would require a similar approach to be drafted to assign the light volume to the nodes with which it overlaps.

Any light has an influence on a node if the light volume overlaps with the node volume. In the case of point lights whether a light overlaps with a node can be easily calculated by comparing the distance between the light origin, $\mathbf{l}.\texttt{orig}$, and the point $\mathbf{p}$ within the node closest to the origin of the light, with the radius $\mathbf{l}.\texttt{radius}$ of the light. If this distance smaller than the radius, then the node overlaps with the light:

$$\|\mathbf{p} - \mathbf{l}.\texttt{orig}\| < \mathbf{l}.\texttt{radius}$$

Point $\mathbf{p}$ within the node and closest to the origin of the light can be easily calculated by clamping the dimensions of the light origin between the extreme values of the node:

$$\mathbf{v} = \begin{pmatrix} \mathbf{l}.\texttt{orig}_x|_{\mathbf{n}_x, \mathbf{n}_x + l_n} \\ \mathbf{l}.\texttt{orig}_y|_{\mathbf{n}_y, \mathbf{n}_y + l_n} \\ \mathbf{l}.\texttt{orig}_z|_{\mathbf{n}_z, \mathbf{n}_z + l_n} \end{pmatrix}$$

where $\mathbf{n}$ is the origin of the node.

This calculation does not need to be executed for each node within the whole space. Any node that overlaps with a light volume

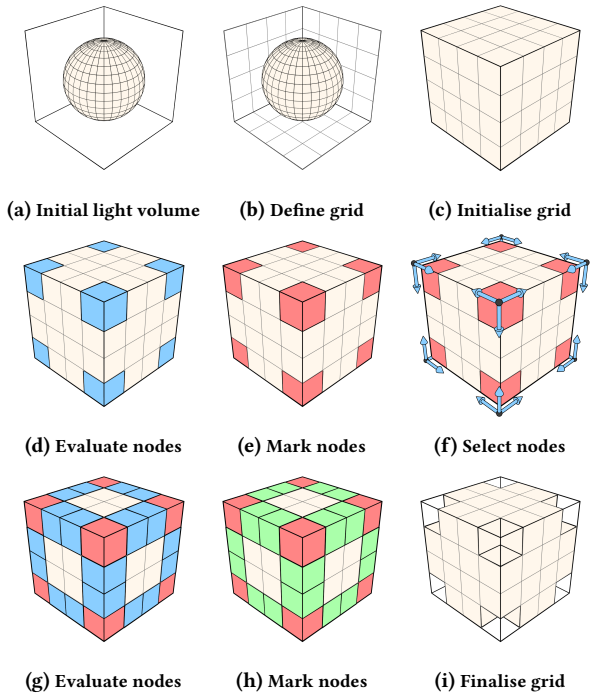
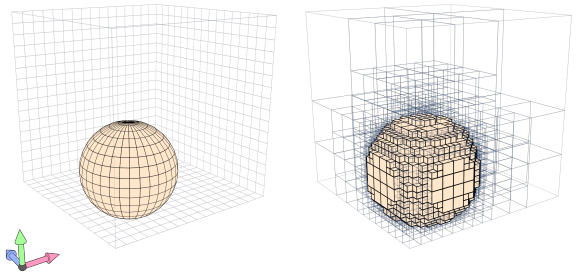


Figure 4: Algorithm to determine the overlap of a single light with a grid of minimal nodes.

falls within the minimal grid of nodes containing the whole light volume. Furthermore, since each light volume is homogeneous, it is only necessary to establish the boundaries of overlapping and non-overlapping nodes in order to know of all nodes whether they overlap. This can be exploited by using a flood fill algorithm to reduce the amount of comparisons. First, all nodes are assumed to be either overlapping or non-overlapping, then the nodes which are not are marked as such. Since a sphere occupies slightly over half of its bounding box volume and partially overlapping nodes are considered to be overlapping, all nodes within the minimal grid, of nodes are initialised to be overlapping. Then starting from the corners of the grid non-overlapping nodes are marked with a breadth first flood fill algorithm. This process is illustrated in figure 4.

Lastly, the way lights are represented in memory needs to be considered. In order to efficiently calculate the change of dynamic lights the influence of lights in the previous frame can be compared to the influence of lights in the current frame. In this case it is beneficial to save the individual lights such that lights that actually change between frames can be evaluated individually. The grid constructed in the previous step can be saved directly. In this case each node requires a single bit in order to represent it. For small lights or large node sizes this is a feasible approach. In case the lights are large or a small node size is used, this approach could lead to a significant memory requirement. In these cases it is more beneficial to represent the lights as an octree, to reduce the memory footprint. An example of such a representation is given in figure 5.



(a) Light volume (b) Octree subdivision

Figure 5: Octree representation of a single light volume.

The octree representing a light can be efficiently constructed in a top-down fashion using the constructed grid. First the octree node containing the whole grid is found. This node will serve as the root node of the light octree. Then for each octree node the type is evaluated. Initially there are three possible situations:

- The octree node volume does not overlap with the grid.
- The octree node volume overlaps partly with the grid.
- The octree node volume falls within the grid.

In the first case the octree node is an empty leaf node, as no filled nodes lie outside of the grid.

In the second case there are two possibilities. Either the node closest to the light origin is empty, or it is nonempty. The octree node is respectively an empty leaf node or a branch node. In the last case there are three possible situations. If the node closest to the light origin is empty, then the octree node is empty as well. If this is not the case, then either the furthest node from the light origin within the octree node is empty or nonempty. The octree node is then respectively a branch node or a nonempty leaf node. In the case that an octree node is a branch node, then the types of the children of this node are evaluated as well.

3.3 Construction of the Scene Octree

The scene octree combines the influences of the individual lights. Each leaf node contains a set of references to the lights which overlap with the leaf node's volume. This octree can be constructed either top-down or bottom-up, depending on the chosen representation of the single lights.

In case the lights are represented by grid nodes, these nodes need to be combined such that a set of nodes is obtained where each node contains the references to its overlapping lights. Subsequently the shallower layers can be constructed. If eight sibling nodes share the same set of indices they can be combined into a single leaf node in the proceeding layer. If they are dissimilar a branch node is added to the proceding layer.

When the lights are represented by octrees, they can be added top-down to a scene octree initialised with an empty leaf node. First the octree node corresponding with the root node of the light is retrieved, constructing additional nodes where necessary. Then each of the nodes of the light octree is added to the scene octree.

3.4 Construction of the Linkless Octree and Light Index List

The construction of the linkless octree follows the original algorithm. The scene octree is represented by two spatial hash functions per layer, as only nonempty leaf nodes have data associated with them. The first spatial hash function describes the octree structure as described in the original algorithm. The second spatial hash function contains a data element consisting of two integers per nonempty leaf node. These integers specify the subset of light indices within the Light Index List that describe the lights which overlap with the leaf node.

In order to construct this linkless octree, the nodes of the specified starting depth are gathered. For each of these nodes the octree node encoding is calculated. If a node is a nonempty leaf node, the current length of the Light Index List and the size of the set of lights associated with the leaf node are added to the second spatial hash function, as the offset and the length values respectively. The light indices of the set of lights are then added to the Light Index list.

Once all layers have been constructed, the linkless octree and the Light Index List can be loaded into GPU memory.

3.5 Light Assignment during Shading

The final step of the algorithm is to determine the set of relevant lights for the samples generated during rendering. This step is similar to that of Tiled and Clustered Shading. First the leaf node containing the sample has to be calculated by descending into the octree until a leaf node is encountered. If this leaf node is nonempty then the offset and length can be retrieved from the corresponding data hash table. Else a length of zero is returned. Once the offset and length are retrieved the set of light indices from the Light Index List can be obtained. The sample is then shaded by summing the shading contributions of the corresponding lights. This step is equal to that of Tiled and Clustered Shading.

4 IMPLEMENTATION

In order to evaluate the Hashed Shading algorithm a new program, `nTiled`, was developed. This program was build with `C++` and `openGL` and compiled with Visual Studio 2015. Besides the Hashed Shading algorithm, also a naive implementation which evaluates each light per sample, and the Tiled and Clustered Shading algorithms were implemented. All programs execute the same shading code, a simple white Lambertian material.

4.1 Deferred Shading

Deferred Shading is implemented by executing two render passes. In the first pass the normal, a white albedo colour, and the depth are written to a GBuffer containing the corresponding textures. In the second pass a full screen quad is rendered, and for each generated sample the shading information is retrieved from the GBuffer and the light contributions calculated.

4.2 Tiled Shading

In the forward and deferred Tiled shaders, the tiles which are influenced by a light are determined by computing the bounding box of the projected light volume in screen space[MM12]. Then to each

of the tiles overlapping with this boundary box, the light index is added. This is all done on the CPU. The implementation does not take into account the minimum and maximum z-values within tiles. Thus the subdivision is not constrained with regards to these values.

4.3 Clustered Shading

Clustered Shading was implemented without the bounding volume hierarchy to assign lights to the clusters specified in the original algorithm. The lights instead were assigned in a brute force fashion by adjusting the Tiled Shading algorithm, and evaluating whether it overlapped with any clusters which contained geometry. The sort and compact step were implemented with `openGL` compute shaders. The assignment of lights is implemented on the CPU which required the use of `glGetTexImage`[1] This leads to a significant slowdown of the algorithm which skews the execution time results.

4.4 Hashed Shading

Hashed Shading is completely implemented on the CPU. The individual lights are represented by by octrees. The size of the offset hash table Φ is selected by using the $\frac{n}{6}$ approximation, and increased geometrically until a correct perfect spatial hash function is constructed. The offset and length values are represented by 32 bit unsigned integers.

5 EVALUATION

The performance of the Hashed Shading algorithm is evaluated with regards to both the construction of the data structures and the execution during shading. The construction of the data structures is evaluated with regards to both the time required to construct the data structures as the memory they take up. The total amount of pixels required by the linkless octree and the size of the Light Index List are used as indicators for this memory usage.

The performance of the shading is evaluated by timing the execution time per frame and by calculating the total number of light calculations required to shade a frame in the deferred pipeline. These values are compared to those of the Tiled and Clustered implementation. Due to the performance issues of the Clustered Shading implementation the execution time of Clustered Shading is not taken into account.

All timings are performed with the `QueryPerformanceCounter` provided on the Windows platform.

The different tests were performed for three scenes:

- Spaceship Indoor: an indoor spaceship scene based on CG Lighting Challenge #18[2].
- Piper's Alley: a street scene based on CG Lighting Challenge #42[3].
- Ziggurat City: A shot from the open movie Sintel[4].

These scenes were chosen to represent the different potential game environments.

The influence of both the number of lights, as the resolution on the execution time and number of lights of the different light assignment algorithms is evaluated. For this six different resolutions ranging from 320×320 to 2560×2560 are evaluated, and number of lights ranging from less than a hundred till more than a thousand lights per scene are evaluated.

All of these simulations were executed on a Dell XPS laptop with the following hardware:

Operating system	Windows 10 64-bit
CPU	Inter Core i7 6700 HQ @ 2.60 Ghz
Memory	16 GB
GPU	NVIDIA GeForce GTX 960M
GPU drivers	372.70

6 RESULTS AND DISCUSSION

In order for Hashed Shading to be viable its performance needs to be at least on par with Tiled and Clustered Shading. This requires a low memory footprint and a similar reduction in execution time and number of light calculations.

6.1 Construction Time and Memory Usage

The results of the construction time as function of the node size relative to the radius of the lights for different number of lights can be found in figure 8. The construction time for individual functions for the largest sets of lights can be found in figure 9. A cubic relationship between the node size and the construction time can be clearly observed. This leads to a significant increase of construction time when more than 8^3 nodes are required to contain a light volume. This increase in construction time is primarily due to the construction of the spatial hash functions representing the layers of the linkless octree.

The same cubic relationship can be seen between the memory usage and the node size. Both the combined number of pixels used in the spatial hash functions of the linkless octree, fig 10, and the size of the Light Index List, fig 11, show cubic growth with regards to a smaller node size. Figure 12 and 13 show that the majority of the pixels used in the linkless octree are used within the deepest layers of the linkless octree. This indicates that a significant number of leaf nodes exist in the deepest layers of the linkless octree, when the node size decreases. This is further supported by the increase in size of the Light Index List. Each leaf node adds its set of light indices to the Light Index List. Thus the total size of the Light Index List is an indicator for the total number of leaf nodes. A smaller node size allows for a more fine-grained description of scene space. When lights partially overlap the leaf nodes can not be combined as their light index sets differ. A smaller node size will increase this effect, and more small octree nodes will be spawned as a result.

The node size effects the size in all three dimensions of the node. Thus a reduction of the node size potentially leads to a cubic increase in number of nodes. This cubic increase of nodes is the source of the increase in number of nodes which drives up the memory usage and construction time.

The influence of the starting depth on the construction time and the combined size of the textures is shown in figure 14 and 15 respectively. The construction time and memory usage are not effected by this change. However, the memory overhead is reduced

[1] `glGetTexImage`: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glGetTexImage.xhtml
[2] scene url:http://www.3drender.com/challenges/
[3] scene url:http://forums.cgsociety.org/archive/index.php?t-1309021.html
[4] open movie url: durian.blender.org

as the total number of required textures is reduced. Increasing the starting depth removes the shallowest layers of the linkless octree. These have a relatively small influence on the total memory usage and the construction time as the spatial hash functions are small compared to the deeper layers of the linkless octree. Thus a greater starting depth can be used to reduce the overhead of the textures and the total number of textures used, without significantly increasing the amount of nodes required. Furthermore removing the shallowest layers of the octree will reduce the number of memory accesses during the shading step, thus improving the performance during shading.

6.2 Execution Time and Light Calculations

Figure 16 and 17 show the execution time per frame for the forward and deferred pipelines during a single simulation of both Tiled and Hashed Shading. Figure 18 shows the number of light calculations per frame during a single simulation of the deferred pipeline for Tiled, Clustered and Hashed Shading.

In the deferred pipeline a similar performance with regards to the execution time can be observed for both Tiled and Hashed Shading, even though the Hashed Shading implementation requires only half of the light calculations. This is a result of the simplicity of the used fragment shader. The bottleneck in this case is not the shading pass but the geometry pass. The change in number of calculations does not influence the geometry pass, thus performance is comparable. When the complexity of the shader, and therefore the amount of work during the shading pass, is increased the difference in the number of light calculations should become noticeable. The Forward Hashed Shading implementation does perform about twice as well as the Forward Tiled Shading implementation, which is in line with the reduction of the number of light calculations achieved with Hashed Shading. The forward pipeline has significant overdraw, thus per pixel multiple fragments are evaluated. This amplifies the influence of the reduction of the number of light calculations.

For small node size the reduction of light calculations achieved by Hashed Shading approaches that of Clustered Shading. When the node size would be reduced further, it should equate and potentially surpass Clustered Shading.

In figure 19 and 20 the average execution time per frame for the Forward and Deferred pipelines as function of the number of lights is plotted. All of the algorithms scale linearly with the number of lights. The results are comparable to those observed for the single simulation. For Deferred Shading the performance increase of Hashed Shading is similar to that of Tiled Shading. In the Forward pipeline, again an improvement of a factor two is noticeable. The number of light calculations, figure 21, shows that the reduction in calculation time is a direct result of the reduction of number of light calculations. The smallest node size shows a comparable curve to that of Clustered Shading.

The behaviour with regards to the resolution is shown in figure 22, 23 and 24. Both the execution time and the number of light calculations are linearly dependent on the number of fragments. Hashed Shading performs slightly better with increasing resolutions, though the effect is almost negligible. The memory usage of both Tiled and Clustered Shading is directly linked to the size of view port. Hashed Shading has no such dependency, thus it will not be significantly affected by changing resolutions. With the trend of growing resolutions, this might prove to be an important attribute of Hashed Shading.

7 CONCLUSION

In this paper the Hashed Shading algorithm has been presented and evaluated. Hashed Shading is a light assignment algorithm which subdivides the scene space independently of the view frustum. This subdivision is then stored in a linkless octree which can be queried during shading. Because the subdivision is independent of the view frustum, changes in the camera position and orientation do not require a complete recalculation of the data structures. Thus the data structures can be reused between frames.

The Hashed Shading algorithm performs better than the Tiled Shading algorithm and approaches the reduction of light calculations achieved by Clustered Shading. Furthermore Hashed Shading performs slightly better with increasing resolutions compared to both Tiled and Clustered Shading, because its subdivision does not depend directly on the direction.

However there are still two major problems with the current Hashed Shading implementation. The memory footprint increases cubicly with a decreasing node size. This leads to unacceptable memory requirements for small node sizes. Secondly, dynamic lights are not currently not supported within the Hashed Shading implementation. Adding this support would increase computation time during shading.

Overall, it can be concluded that camera-independent light assignment is a viable strategy. The decoupling of resolution could play a role with the trend of increasing screen resolutions. Future work would need to solve the memory requirements and allow for efficient support of dynamic lights in order for Hashed Shading and similar algorithms to be competitive with Clustered Shading.

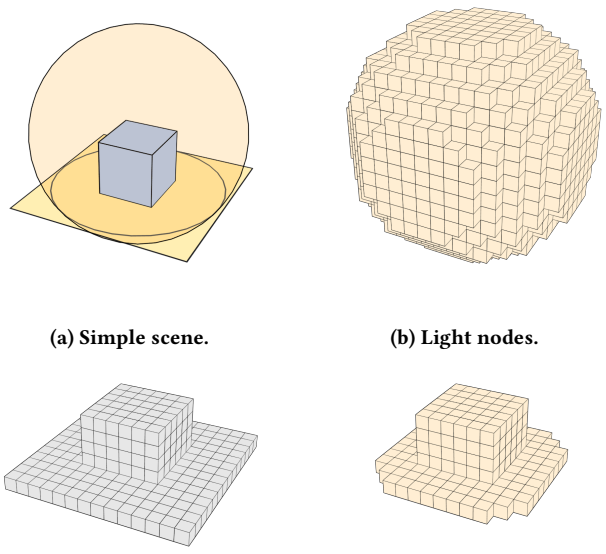
8 FUTURE WORK

The two main issues with the current implementation of Hashed Shading are the large memory footprint and the lack of support for dynamic lights. These two issues have to be addressed before Hashed Shading can become a viable alternative to Clustered Shading. Proposals to solve those two issues are introduced in the next sections.

Besides these two issues, there are several other directions for future work. Currently the Hashed Shading implementation only supports point lights. In order to support other light types, strategies to convert their light volumes into octrees need to be drafted. The current implementation also does not support shadows currently. There are several algorithms which leverage the clustered shading clusters to reduce the amount of work required to support shadows within Clustered Shading[OSK+14, KSDA16]. Similar approaches could be used to support shadows in Hashed Shading by leveraging the octree data structure.

8.1 Memory Usage Reduction

8.1.1 Geometry. An important insight Clustered Shading uses to reduce memory usage is that only clusters which contain geometry will potentially be queried. Thus only clusters that contain potential samples need to be constructed and kept in memory. This



(a) Simple scene. (b) Light nodes.

(c) Geometry nodes. (d) Light geometry nodes.

Figure 6: Reduction of the number of nodes by excluding empty space.

reduces the total number of clusters required by Clustered Shading. This insight can also be leveraged by Hashed Shading. There is no need to keep nodes which contain no geometry in memory as they will never be queried. If a second octree with the same origin and size of the screen octree is constructed, where the leaf nodes specify whether the corresponding volumes contains geometry, then this octree can be used to filter out nodes which will not be queried. Only if a node contains both geometry and light information, it needs to be stored in the linkless octree. Thus such a geometry octree could be used to greatly reduce the number of nodes in the linkless octree as is shown in figure 6.

A second advantage of such an approach would be that the number of nodes would not necessarily be cubicly dependent on the node size. A smaller node size would also increase the precision with which geometry is represented, thus it would reduce the space which contains geometry and therefore reduce the increase in number of nodes which need to be represented within the linkless octree.

Lastly the conditions required for a set of sibling nodes to be combined into a single leaf node in the preceding layer could be weakened. Since nodes which do not contain any geometry data will never be queried, an arbitrary value could be assigned to them, without influencing the performance of Hashed Shading. The current condition requires all the sibling nodes to have the same set of lights, before they can be combined into a single leaf node. With this optimisation this can be weakened to requiring to all sibling nodes which contain geometry to contain an equal set of lights. This would further reduce the number of nodes needed to represent the scene space.

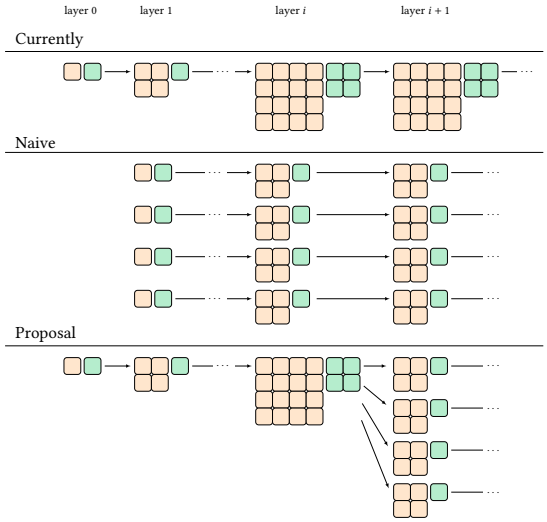


Figure 7: Subdivision strategy deeper layers of the linkless octree.

8.1.2 Subdivision of Space. A common approach in many graphical applications is to only load data into memory which is visible, i.e. only that which falls within the view frustum. In the current implementation the whole scene is loaded into memory, without any regard for the view frustum, which leads to an unnecessary large memory usage. If the view frustum is significantly smaller than the whole scene, it can be advantageous to subdivide the light assignment octree in chunks, and only load the visible chunks into memory. If the chunk size is set to be similar to the view frustum, at least eight chunks are required to cover the view frustum at any point. This could be extended to 27 chunks if camera rotations are common. In this case no new chunks will have to be loaded in when the camera orientation changes, thus reducing the texture bandwidth. When these chunks require less memory than the complete representation, the memory usage is reduced.

A potential problem of this approach is the large number of textures required to represent the octree. Each layer of each chunk would require two separate textures, four if the layer also contains leaf nodes. If 27 chunks are used, and each chunk uses several layers, this would quickly lead to an infeasible amount of textures. However looking at the results of the memory usage, it can be shown that the majority of the memory is used in the deeper layers, which contain a large number of nodes, and that the shallow layers are relatively small. An alternative would be to only subdivide the space into chunks at a certain depth. In the current implementation each layer represents the full scene space, however there is no obstacle into subdividing this space into smaller chunks. If we would subdivide layer $l+1$ into chunks, then the branch nodes in layer l would only require a single value to identify the chunk in which the child node lays. This would make it possible to reduce the memory usage in the deeper layers, while keeping the number of changing textures to a minimum. The possible options are shown in figure 7.

Finally, the chunk approach would make it trivial to support level of detail. Objects close to the camera will generally take up

more screen space than objects further away from the camera, due to perspective. Thus if the same reduction of light calculations is created for objects far away from the camera as for objects close by the camera, the amount of memory used to reduce the number of lights for samples far away will be relatively greater. By reducing the amount of memory used for space further away from the camera, and increasing it for space close to the camera, a greater reduction of light calculations overall can be achieved. In Clustered Shading this is done by scaling the cluster volume with regards to the camera's z-axis.

The linkless octree could support this level of detail by defining multiple spatial hash functions per layer. When a chunk is loaded close to the camera, the regular spatial hash functions would be loaded. For chunks further from the camera, only layers till a certain value could be loaded in, where the last loaded layer contains only leaf nodes. Thus per layer one regular spatial hash function and one spatial hash function only containing leaf nodes would have to be defined.

8.2 Dynamic Light Support

The second issue of the current implementation of Hashed Shading is the lack of support for dynamic lights. The downside of not rebuilding data structures per frame is that changes in lights are not implicitly reflected in the data structures any more. Supporting dynamic lights would thus always increase the amount of work per frame. Games and other real-time applications generally have a high frame rate, between 30 and 60 frames per second, therefore changes in lights are generally small and local. This could be leveraged to efficiently update the Hashed Shading data structures. The bottleneck operation in the construction of the Hashed Shading data structures is the construction of the spatial hash functions of the layers of the octree. In order to efficiently support dynamic lights, these operations thus need to be kept to a minimum. A spatial hash function needs to be rebuild if a new node is added that collides with an existing value:

$$H\left[h_0\left(\mathbf{k}\right) + \Phi\left[h_1\left(\mathbf{k}\right)\right]\right] \neq \varnothing$$

where $\mathbf{k}$ is the position of a new node. In all other cases the existing textures can be updated without a complete recalculation of the spatial hash function of a layer.

There are two situations which require the addition of nodes in layers. If a light index is added to a previously empty node, a new data element needs to be added to the data hash function of a layer. When a leaf node gets subdivided into eight child nodes, a new node is added to the first spatial hash function of the next layer describing these eight child nodes, and for each of the child nodes that contain data, the data nodes are added to the data spatial hash function of the next layer. The subdivision of nodes therefore is the most likely source of recalculations.

It might prove efficient to slightly extend the size of the hash tables to increase the amount of empty space within them. While this would increase the memory foot print, it could greatly reduce the number of times a spatial hash function needs to be recalculated. The effects of such a change would be interesting to explore.

In the current implementation the whole space of each layer is globally linked, therefore a local change of a single light might require the complete space to be recalculated. Currently the implementation can not take advantage of the local nature of light changes. If the support of dynamic lights is combined with the chunk optimisation, only the chunks affected by a light change need to be updated, reducing the space affected by that change. Furthermore, because the construction time of a spatial hash functions is directly linked to its size, recalculating the spatial hash function of a chunk would be smaller than recalculating the spatial hash function of the complete space, further reducing the amount of time needed to update. Lastly, only chunks that are currently in memory need to be updated immediately, as changes to these chunks are directly visible. Changes to chunks that are not currently loaded on the GPU will not be visible until the chunk is loaded in GPU memory. Changes to invisible chunks can be queued, and only executed once the chunk has to be loaded into memory. This would reduce the bandwidth and potentially the execution time.

REFERENCES

[AA03] Jukka Arva and Tima Aila. Optimized shadow mapping using the stencil buffer. *Journal of Graphics Tools*, 8(3):23–32, 2003.

[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: Drakes fortune. In *Game Developer Conference*, 2008.

[CJC+09] Myung Geol Choi, Eunjung Ju, Jung-Woo Chang, Jehee Lee, and Young J Kim. Linkless octree using multi-level perfect hashing. In *Computer Graphics Forum*, volume 28, pages 1773–1780. Wiley Online Library, 2009.

[DWS+88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 21–30. ACM, 1988.

[HA11] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[Kar13] Brian Karis. Real shading in unreal engine 4. *Proc. ACM SIGGRAPH Courses*, page 22, 2013.

[KSDA16] Viktor Kämpe, Erik Sintorn, Dan Dolonius, and Ulf Assarsson. Fast, memory-efficient construction of voxelized shadows. *IEEE transactions on visualization and computer graphics*, 22(10):2239–2248, 2016.

[LH06] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 579–588. ACM, 2006.

[Mag11] Kenny Magnusson. Lighting you up in battlefield 3. In *Proc. 25th Annual Game Developers Conference*, 2011.

[MM12] Michael Mara and Morgan McGuire. 2d polyhedral bounds of a clipped, perspective-projected 3d sphere. *JCGT. in submission*, 5, 2012.

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

[OBA12] 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.

[OSK+14] Ola Olsson, Erik Sintorn, Viktor Kämpe, Markus Billeter, and Ulf Assarsson. Efficient virtual shadow maps for many lights. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, pages 87–96, New York, NY, USA, 2014. ACM.

[PD14] Aras Pranckevicius and Rendering Dude. Physically based shading in unity. In *Game Developer's Conference*, 2014.

[Per13] Emil Persson. Practical clustered shading. *SIGGRAPH Course: Advances in Real-Time Rendering in Games*, 2013.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 197–206. ACM, 1990.

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

[Tat16] Natalya Tatarchuk. Advances in real-time rendering, part ii. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH '16, New York, NY, USA, 2016. ACM.

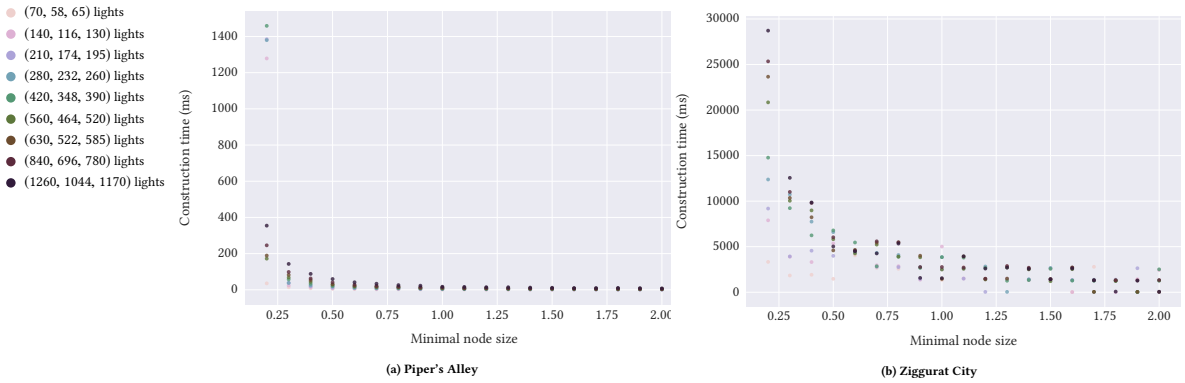


Figure 8: Construction time of the Hashed Shading data structures.

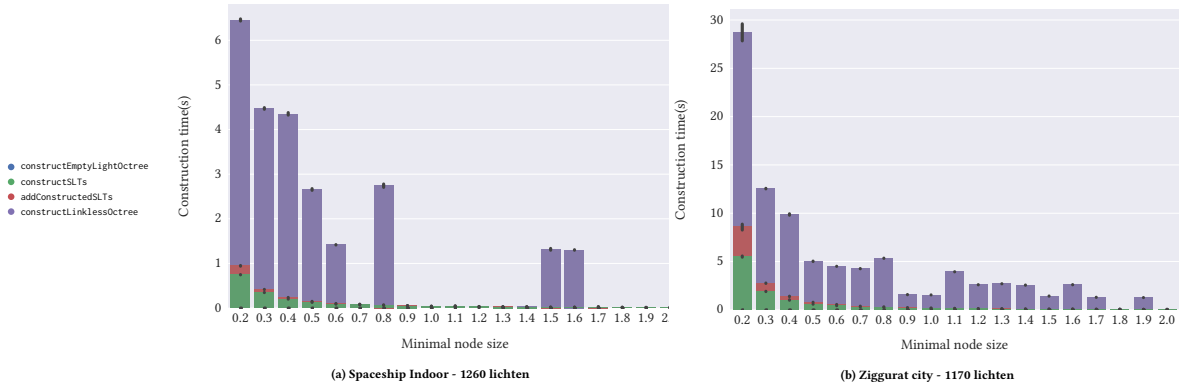


Figure 9: Construction time of the individual steps of Hashed Shading.

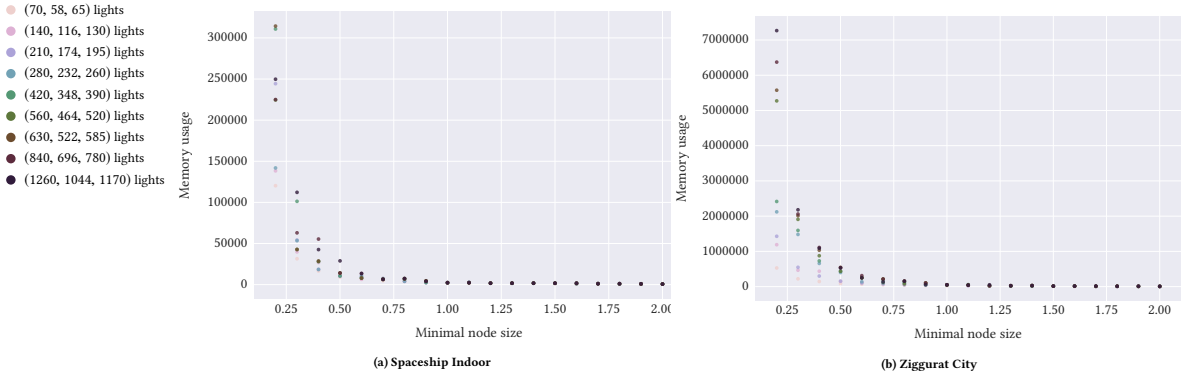


Figure 10: The combined number of pixels of the Linkless Octree.

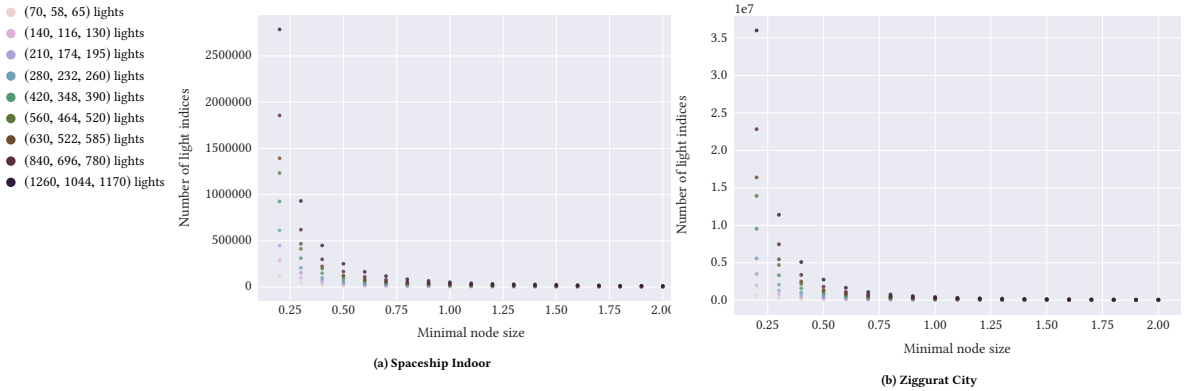


Figure 11: Size of the Light Index List.

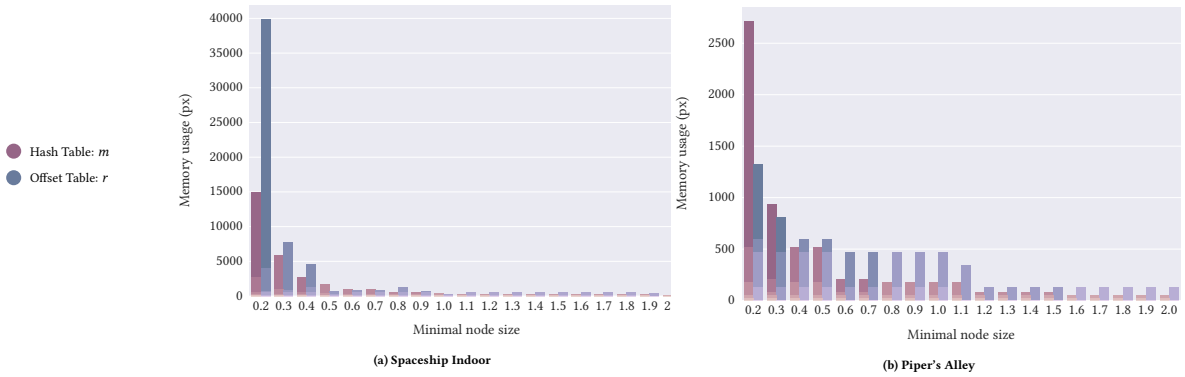


Figure 12: Number of pixels used by the octree description spatial hash functions per layer of the linkless octree.

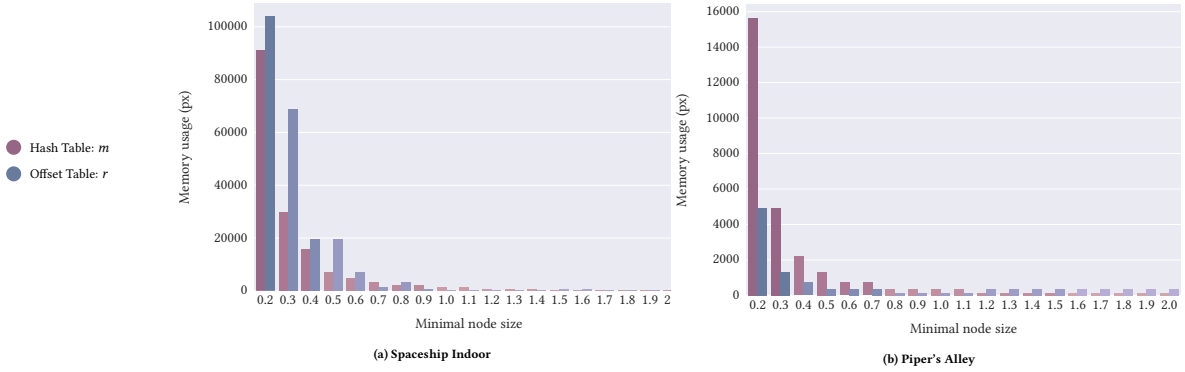


Figure 13: Number of pixels used of the data description spatial hash functions per layer of the linkless octree.

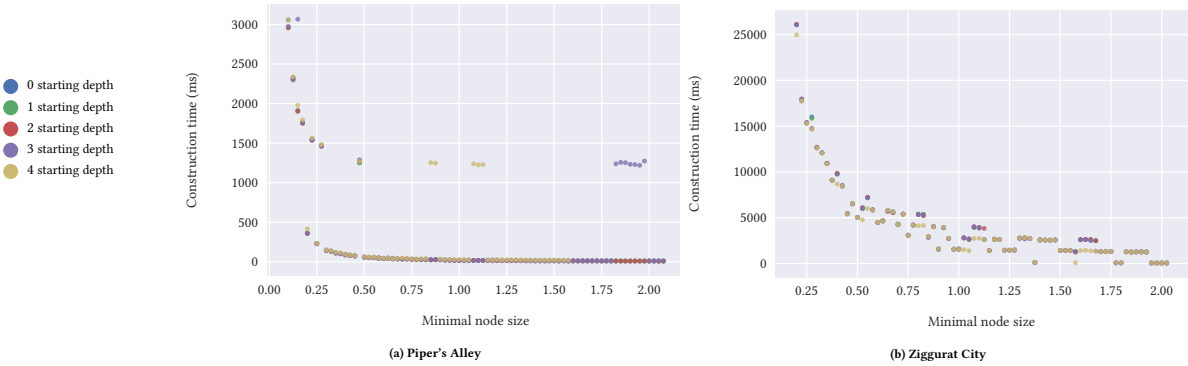


Figure 14: Construction time as function of the starting depth.

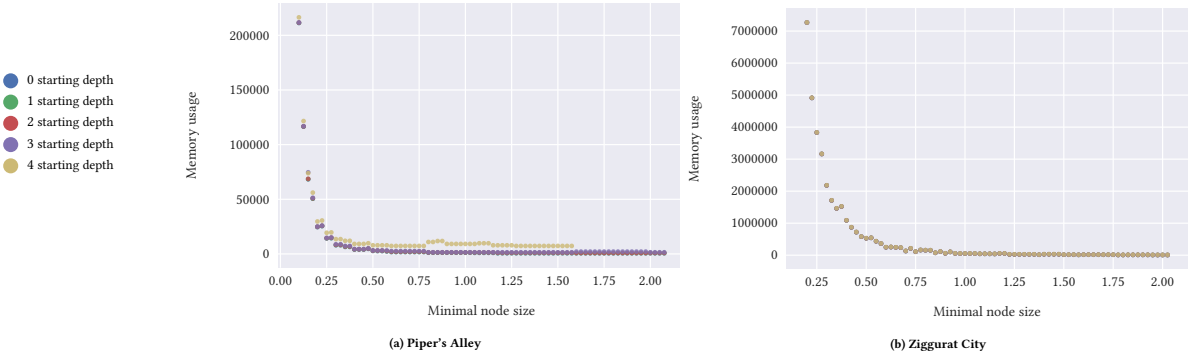


Figure 15: Number of pixels of the linkless octree as function of the starting depth.

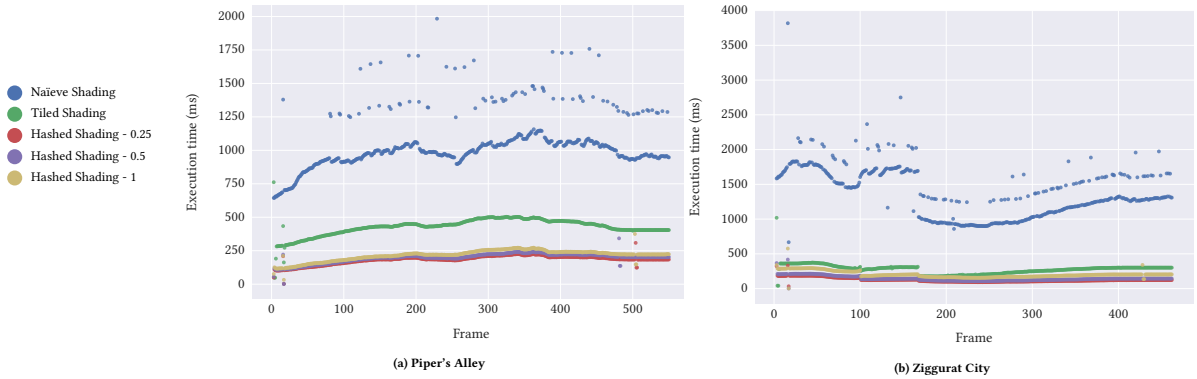


Figure 16: The execution time per frame of Forward Shading.

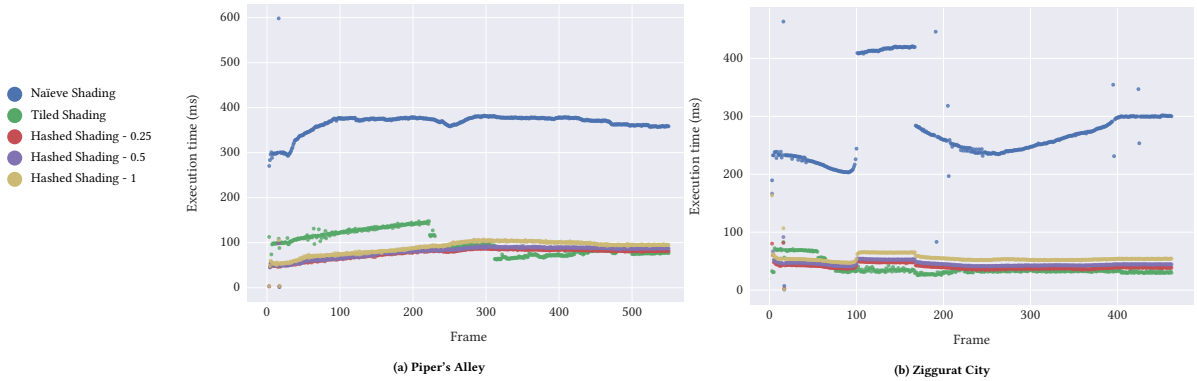


Figure 17: The execution time per frame of Deferred Shading.

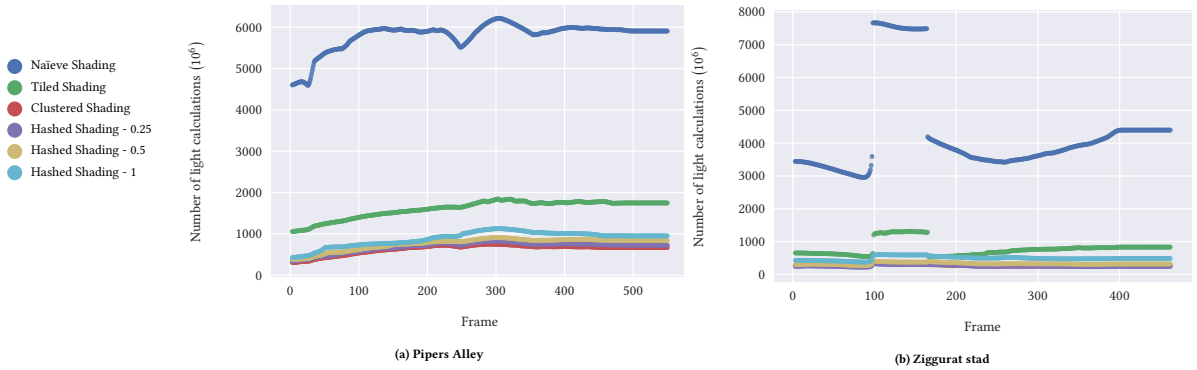


Figure 18: The number of light calculations per frame of Deferred Shading.

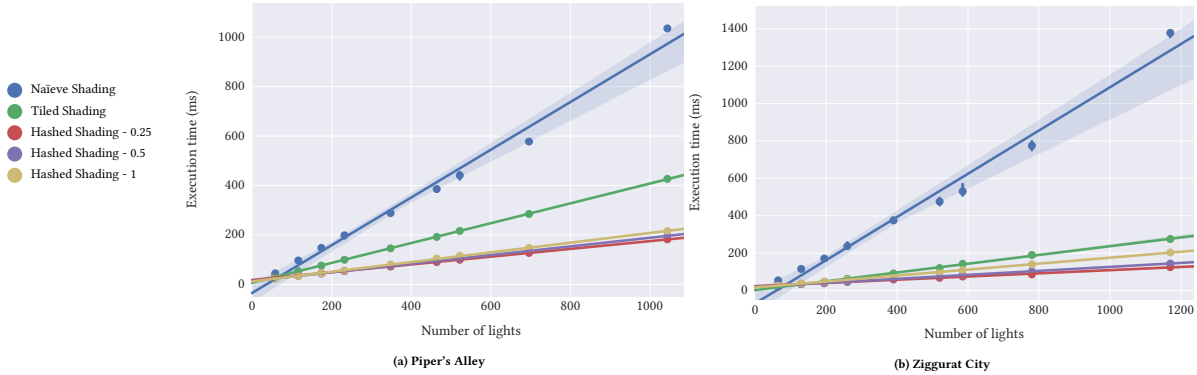


Figure 19: The execution time per frame as function of the number of lights for Forward Shading.

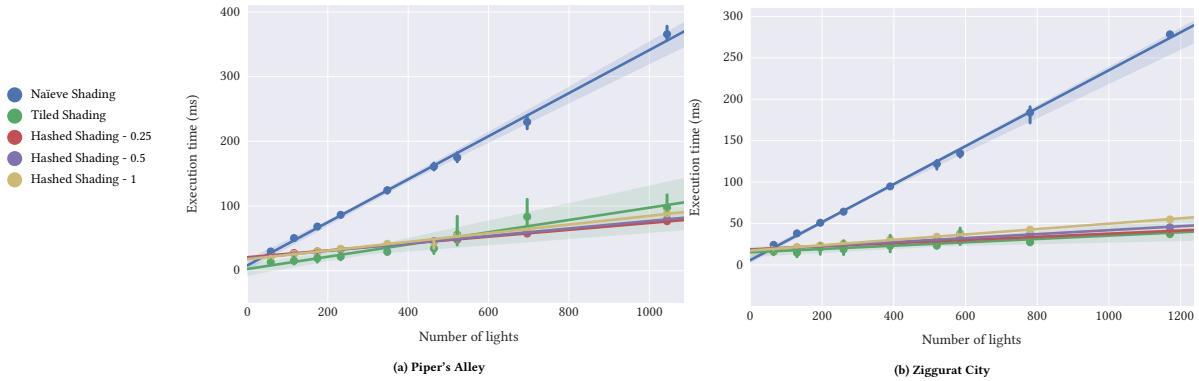


Figure 20: The execution time per frame as function of the number of lights for Deferred Shading.

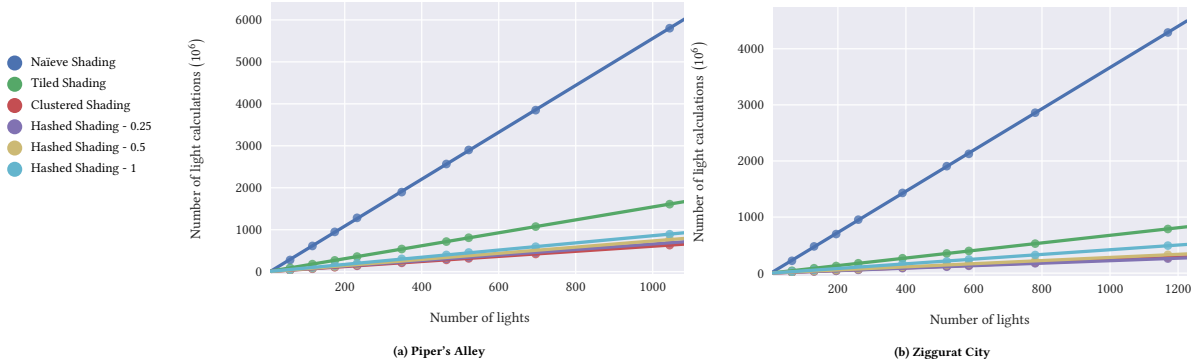


Figure 21: The number of light calculations as function of the number of lights.

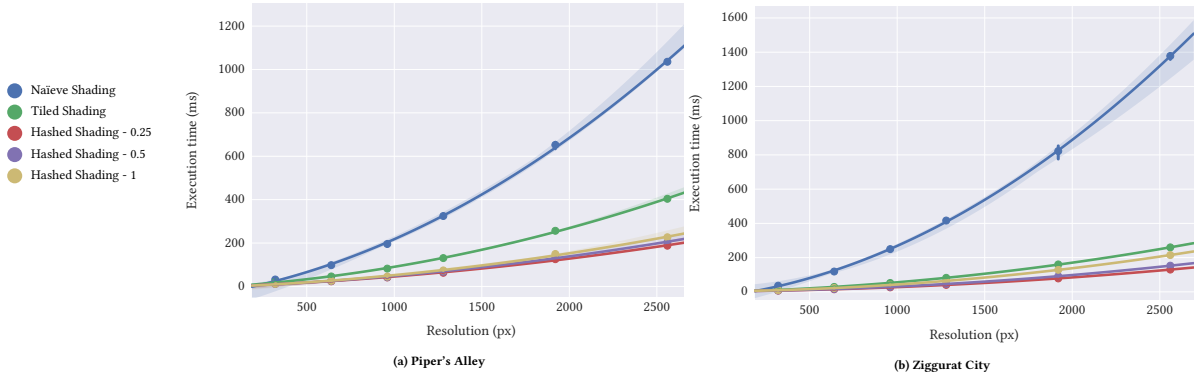


Figure 22: The execution time per frame as function of the resolution for Forward Shading.

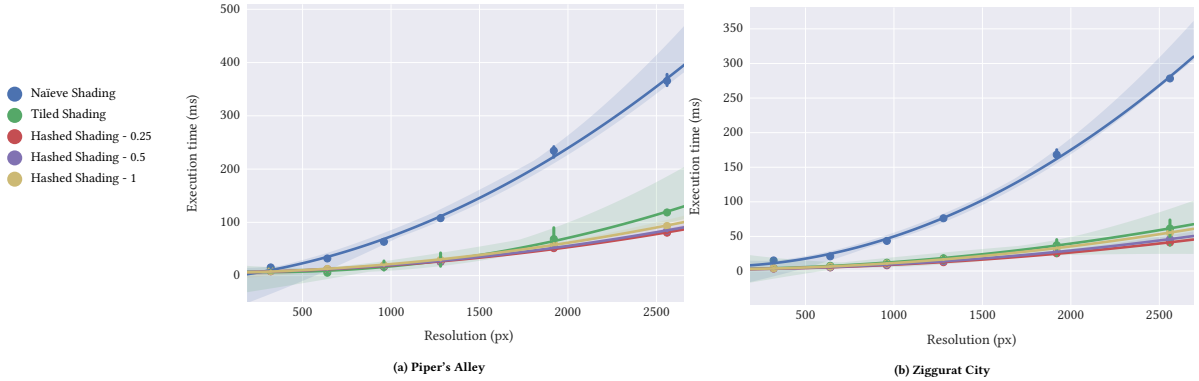


Figure 23: The execution time per frame as function of the resolution for Deferred Shading.

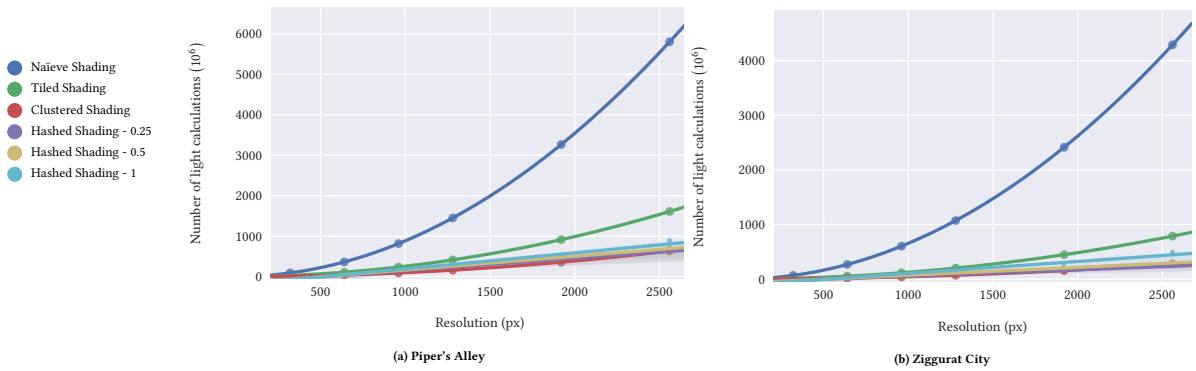


Figure 24: The number of light calculations as function of the resolution.