MASTER'S THESIS | LUND UNIVERSITY 2015

Rendering Light and Shadows for Transparent Objects

Isak Lindbeck

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-07

Rendering Light and Shadows for Transparent Objects

(Using Linked Lists on the GPU)

Isak Lindbeck
adi09ili@student.lu.se

April 13, 2015

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Michael Doggett, michael.doggett@cs.lth.se
Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

This masters thesis presents a new approach to enable complex illumination and shadowing for transparent objects such as glass or smoke. This is achieved by combining two algorithms which are intended to run in a real-time environment commonly found in modern computer games. The first algorithm is called "Light Linked List", it solves the problem of illuminating transparent objects and was recently presented at the SIGGRAPH2014 conference. The second algorithm is used for shadow calculations and is a new take on the "Deep Shadow Maps" technique. Both algorithms use linked lists to a high degree. The lists are constructed on the GPU by taking advantage of recent hardware improvements and updates to the OpenGL API. By using these two algorithms together, we enable illumination and shadow casting of transparent object with a performance that allows for real-time frame rates.

Keywords: Light, Shadow, Deep Shadow Maps, Light Linked List, OpenGL

Acknowledgements

I would like to thank Ass. Prof. Michael Doggett for being the supervisor of this thesis and for providing excellent guidance and many interesting discussions. I would also like to thank my examiner Flavius Gruian for all the helpful feedback on this report.

Contents

Chapter 1
Introduction

Demands on modern real-time rendering is always increasing. One of the big challenges is calculating which parts of a scene is illuminated and the shadows that are created as a result. The commonly used methods today do not allow for object with transparency to be correctly lit. Another missing feature is having part of the light pass through transparent objects when they are illuminated. These technical limitations make todays algorithms produce unsatisfactory illumination and shadows for objects such as glass or thin fabric. Having these capabilities would also be advantageous when rendering volumetric objects such as clouds or smoke, because light propagation and self shadowing is an important factor in producing a credible image for these types of objects.

This thesis investigates the usage of an older algorithm proposed for offline rendering called "Deep Shadow Maps". This technique would provide us with the means to attain the light attenuation at any distance from a light source. This will also make it possible to color a light by the medium it is passing through. In addition, a newly proposed technique called "Light Linked Lists" is studied. This algorithm will make it possible to determine which lights are affecting any visible point in the scene.

With this chapter we will first introduce the reader to the background of light and shadow calculation. Then we will move on related work which either lays the foundation, or is otherwise relevant, to the work in this thesis.

1.1 Background

1.1.1 OpenGL

OpenGL is an API for interaction with the graphics card managed by the non-profit Khronos Group. It provides a graphical pipeline to render images on a computer. With version 4.3, "Shader Storage Buffer Object" was introduced [12]. This made it possible to allocate memory on the GPU which could be read and written from the graphic card. Having this capability is an important part for the algorithms investigated in this project since it allows for a more general purpose computing approach to the necessary calculations.

OpenGL also includes the specifications for the programming language GLSL. It has a C like syntax and is used to write shader programs that are executed on the GPU. Simplified, a shader program, or a shader for short, computes the samples in a render pass. Thus they are normally used to determine the color of the pixels on the screen. Different geometries or objects in a scene can have different shader assigned to them. This makes it possible to create specific visual properties for different materials.

1.1.2 Render Pipeline

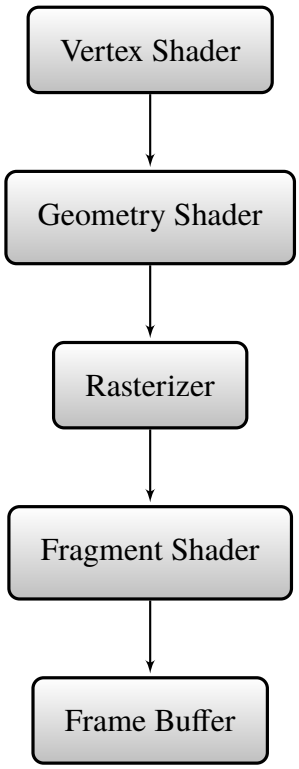


Figure 1.1: An example of the render pipeline.

An illustration of the render pipeline can be seen in figure 1.1. In each frame the pipeline takes the triangle representing the scene. The converts the triangles into pixels and finally determines a color for the pixels [1].

The vertex shader handles the processing of the individual vertices for all the triangles given to the GPU. It is used to transform the 3D position of the vertex into the 2D coordinate it should have on the screen. The vertex shader is programmable and can modify things such as position, texture coordinates and colors.

The geometry shader is an optional step in the pipeline. It can be used to create new, more complex geometries. One typical use is to remove the flatness of rough surfaces. This is done by dividing the triangle mesh into a finer grid, and then applying small offsets to the new vertices.

In the rasterizer phase, the triangles are converted into pixels. It determines which triangles covers each pixel. The result for each pixel and triangle intersection is called a fragment. So the input to the rasterizer is triangles and the output is fragments. This phase is not programmable.

The fragment shader handles the fragments produced by the rasterizer. It is mainly used to output a color corresponding to each fragment. The fragment shader can write output to several buffers and the data do not necessarily have to represent a color. This is the stage where the bulk of the algorithms for this thesis is implemented. Where we instead of creating a color output for the next phase in the pipeline, construct linked lists to save in the GPUs memory.

With the frame buffer all the fragments are put together to create the final color of the pixels. When fragments are located in the same pixel they can be blended together or discarded based on the distance to the camera. The result can either be sent to the monitor or be saved in memory. As the rasterization, this phase is not programmable.

1.1.3 Render Passes

An important part of todays light and shadow calculations is using extra render passes. A render pass samples the scene, from the point of view of a virtual camera, and produces an image. This image is usually the picture displayed to the end user on the screen. When preforming the light and shadow calculation the render pass is instead used to collect data that can be used later. Instead of displaying the image on the screen it is sent to be saved in the RAM. The pixels in the images can be used to store different types of data. For example, this can be the distance to the camera or the normal of the surface visible in that pixel.[3] This data can then be used in other calculations, such as using the normal to determine how light is reflected off a surface.

The render pass runs on the GPU and each sample of the scene is executed in a separate thread. This effectively takes advantage of the parallel computing power of the GPU and is the main reason for it being a popular technique in graphics. However this also puts some restrictions on what algorithms can effectively do in extra render passes. It needs to be possible to split the computation into a large amount of threads without introducing to much overhead. Therefore some general computing algorithms is unsuited to run on the GPU.

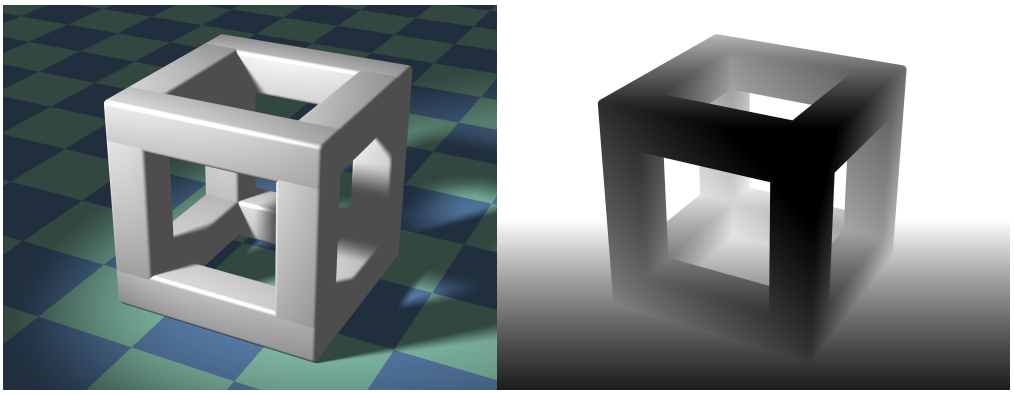
Both the algorithms investigated in this thesis use extra render passes to collect data in the scene. But instead of producing an image, the result is stored on the memory of the graphics card. Because of this the result can be stored in other forms than images, which we take advantage of.

1.1.4 Deferred Shading

Deferred shading and its cousin deferred lighting are very common techniques in modern computer games. They use extra render passes to collect information about the scene before performing the illumination of the final image. [10] Since the result of the render pass is represented as an image, the algorithms are unsuited to handle transparent objects. Each "pixel" can only store information about one visible surface. In the case of transparent objects, we have several visible surfaces and therefore the scene can not be accurately represented by the flat data structure of the image. This master thesis can be seen as a way to work around this by allowing such information to effectively be separated by depth from the camera.

1.1.5 Shadow Mapping

Shadow mapping was introduced as early as 1978 [13]. It is a technique where a render pass is performed from a light sources point of view. The result of the render pass is stored as a texture where each pixel contains a depth value, effectively giving the distance form the light to the parts of the scene being illuminated. This texture is called a shadow map and can later be used to do a shadow lookup which determine whether a point in the scene is in the shadow of another object. There are many advantages with using shadow mapping. First of all it is fast as it effectively uses the GPU with a render pass. Another big advantage is that the result of the shadow lookup can be considered a picture, this enables the application of different image processing algorithms. For example, it is popular to use blur in order to create a softer edge on the resulting shadow [4].



(a) **(b)**

Figure 1.2: **(a)** A cubic structure. **(b)** Depth map corresponding to the cubic structure. Points further away appears lighter. Source: Wikipedia page for depth map.

Figure 1.2 is a example of a depth map. The distance between the scene and the camera is stored as a gray scale value where points further away has a lighter color.

There are some inherent problems with shadow mapping. Many of the problems can be attributed to the resolution of the shadow map. A resolution that is to low can result in visual artifacts. While a higher resolution produces a better image, the computational cost is often to high. One common problem is artifacts caused by a surface shadowing itself. This can be worked around by applying a small offset when checking if a surface is in shadow or not. With this offset the surface is calculated as being closer to the light source and is therefore less likely to be in shadow. However the size of the offset is scene dependent which means it has to be adjusted on a case-by-case basis.

1.1.6 Game Engine

For game logic and scene handling a Java based game engine called JMonkeyEngine[5] was chosen. This choice was made primarily because of previous experience with the engine which speeds up the implementation of function not directly related to the two algorithms investigated in this project. The other reason was that it was an easy way to get portability for the purpose of testing on different platforms.

A game engine sets up a platform for a lot of underlying features required for an interactive 3D environment. One important part is the scene graph. It handles the location, rotation and scale of all the objects placed in the scene. The JMonkeyEngine also acts as a layer in between the OpenGL API and the game code. However, it does not support the features from OpenGL 4.3 that was required for this project. In order to solve this we had, too some extent, work around this layer in order to take advantage of the lower level APIs.

Overall we consider the choice of game engine to be a good one as it provided a solid platform to build on and made portability between different operating systems easy.

1.2 Related Work

There are many algorithms that do illumination and shadow calculations but there are few that aim to solve this for transparent objects. In this chapter we give a brief description of previous work that are relevant to this thesis.

1.2.1 Clustered Deferred and Forward Shading

Algorithm similar to the Light Linked List in that it preprocesses the lights and supplies information about what parts of the scene, in relation to the screen, is illuminated [8]. It is not aimed at handling transparent objects but is focused on performance and handling many light sources. The algorithm considers a 3 dimensional grid inside the camera frustum where the tiles are called clusters. Each cluster is paired with a list of relevant lights. The strength of the algorithm is that the structure is divided by depth from the camera along with the screen position. This makes the performance less view dependent and makes it possible to process an impressive number of lights.

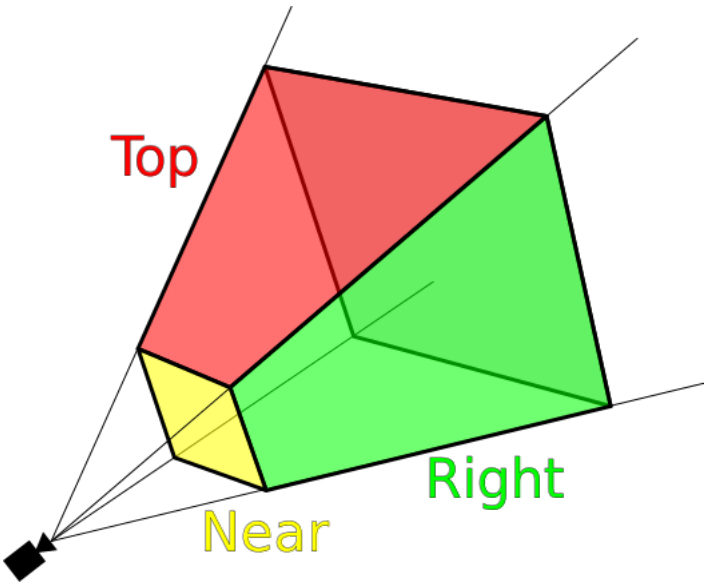


Figure 1.3: Example of a camera frustum.
Source: Wikipedia page for viewing frustum.

Figure 1.3 shows an example of a camera frustum. Everything inside the box is considered visible. Geometries outside of the frustum is ignored when a render is performed and will therefore not show in the result. This also gives the camera a limited view range and objects far away will not be visible.

1.2.2 Real-Time Concurrent Linked List Construction on the GPU

This is a paper that presents an algorithm which achieves order independent transparency [14]. While this technique does not target light and shadow calculations and is outside the scope of this project, it does provide a thorough explanation of implementing linked lists on the GPU. Linked list is used extensively in constructing both the light volumes and deep shadow maps ands is an integral part of this thesis.

The linked list data structure on the GPU consists of three parts. The first part is a buffer containing the elements of all lists which we call the element buffer. Second part is the index buffer which contains pointers to the first element in each list. The pointers position within the buffer corresponds to a set of pixels on the screen. The third and final part is a counter for the total amount of stored elements over all the lists. When a list is empty the pointer in the index buffer has the value negative one.

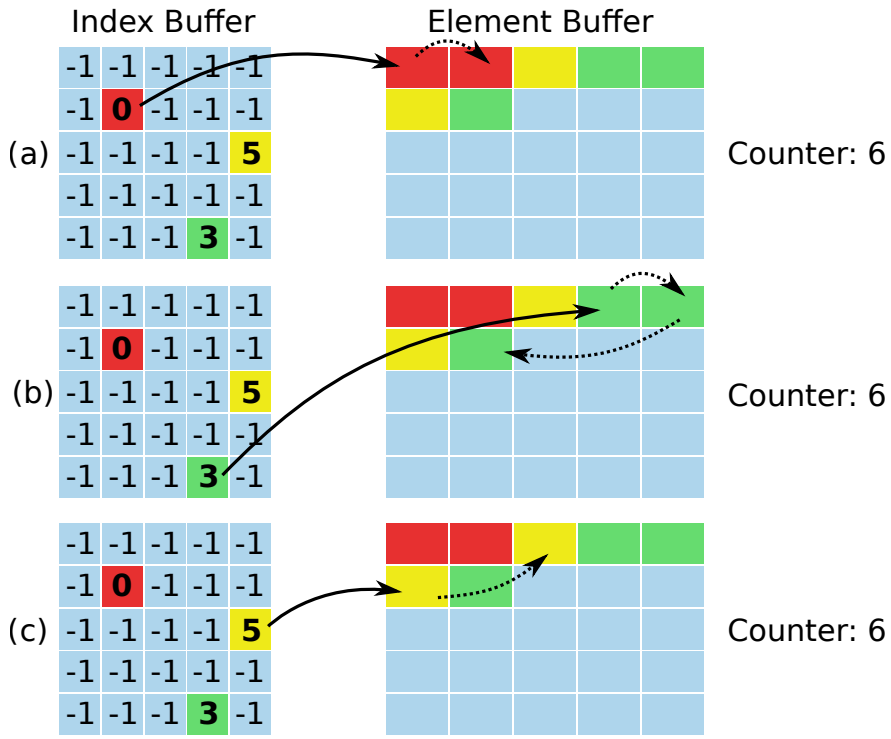


Figure 1.4: Example of three linked lists and how they are stored in the buffers.

In figure 1.4 we have an example of the linked list structure for the GPU. The first list is marked red and is trailed in example (a). The pointer to the first element in the first list is stored in the index buffer at coordinate (1,4) and point to position zero in the element buffer. Example (b) trails a second list marked in green. This example demonstrates how elements in a list do not have to be stored consecutively in the element buffer. The third and final list is marked yellow and shows that the order of the elements in a list is independent of the order they are stored in the element buffer. For this example the element counter would have the value six.

When entering an element into a list the index of the first element in the list have to be determined. This is done by doing a lookup in the index buffer where the screens x- and y- coordinates are used to select the position in the index buffer. This gives a pointer to the first element in the list. Then an atomic operation is performed on the counter that reads the current value and increments by one. The returned value is the position in which the entered element will be written to in the element buffer. After that the correct position in the list is determined and address pointers are adjusted so the new element is part of the list.

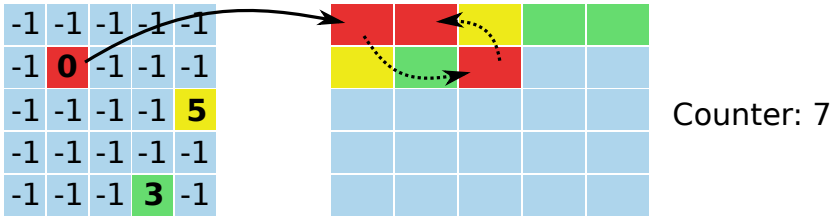


Figure 1.5: Example of an element insertion.

In figure 1.5 a new element have been inserted into the first list from the examples in figure 1.4. The new element is stored at position six in the element buffer. This position was determined from the counter variable. Our new element is located between the two previous elements in the linked list and the pointer are adjusted accordingly. The counter is incremented to seven to account for the new element.

1.2.3 Real-Time Lighting via Light Linked List

The "Light Linked List" algorithm was first presented by Abdul Bezrati at the SIGGRAPH-2014 conference and no technical paper have been released that go into detail of an implementation. This technique is the foundation of the light calculations used in this thesis. The presentation shows that it is possible to illuminate transparent objects using buffers containing linked lists on the GPU [2]. Each linked list represents an area of the screen and the elements in the list contains information that identifies a light source and two depth values representing distances to the camera. Together the two depth values creates a volume of light and if a point in the scene is located between these depth values it is illuminated by the identified light source.

Since these "light volumes" are constructed based on the pixels on the screen, the algorithm is bounded to screen space. This means that any parts of the scene outside of the camera frustum, that are affected by a light, will not be taken into account. Effectively this avoids unnecessary calculation that would not yield any visible result on the screen.

The presentation also show it is not necessary to have a linked list for each screen pixel, instead the buffers can be constructed with a virtual resolution a fraction the size of the images rendered to the screen. This saves a lot of memory usage and calculation time and the performance results shows that there is no problem running the algorithm in real-time.

1.2.4 Deep Shadow Maps

Introduces a technique for rendering with deep shadow maps [6] and is the foundation for the deep shadow map algorithm implementer in this master thesis. The algorithm is made for offline rendering and is therefore not directly applicable to the goals of this project. The paper presents the idea that instead of storing a single depth value in each pixel of a shadow map, it is possible to store a visibility function that approximates how much light is passed through to any possible depth from the light source. This allows light to propagate through materials and enables features such as self-shadowing for clouds, smoke and high detailed geometries like hair or fur. The visibility function can be described as points along the z-axis of the camera. Each point represents the light attenuation at the corresponding depth.

The visibility function is calculated by taking several samples per pixel in the shadow map. Each sample measures how much the light drops off at different depths from the camera. An average of all the samples in a pixel is determined and the result is called a transmittance function. The transmittance function for surfaces and volumetric objects are calculated in separate steps and then joined together. The resulting function is compressed to save memory and the visibility function of the pixel is complete.

1.2.5 Deep Shadow Maps from Volumetric Data on the GPU

This paper investigates deep shadow maps with a real-time render technique[11]. It has many similarities to the objective of this thesis but has a more narrow scope which focuses on volumetric objects. The paper discusses how to sample a volumetric object that is represented as a 3D texture and how to use the acquired data to construct a deep shadow map. The deep shadow map is represented as a normal texture where the RGB color channels corresponds to the depths from the light source at which the light is decreased by 10%, 50% and 90%. When preforming a shadow lookup the lights opacity is determined by linearly interpolating between two of the three values depending on the distance to the light source. So if a point is in the middle of the 50% and 90% depths, the drop in light intensity will be calculated to 70%.

1.2.6 Ray tracing

Ray tracing is an alternative method of rendering for computer graphics. Rays are sent from the camera and bounce around the scene in order to determine how much light reaches the camera [9]. This technique is a more accurate simulation of the real world and it does solve the problem of handling transparent objects. So from that point of view it is an alternative to the algorithms investigated in this thesis. However, ray tracing is in general not used in real time graphics. The reason for this is that it is to calculation heavy. There is much research being done to make ray tracing more effective and with future hardware it could be a viable option even in real time applications.

Chapter 2
Algorithm

In this chapter we will go through the details of the "Light Linked List" and "Deep Shadow Mapping" algorithms implemented in this thesis. But first we will give a short explanation of the general system.

The system supports three different types of light, directional, spot- and cone-light. Volumetric objects are represented as 3D textures contained within a cube. Two different types of volumetric objects are implemented. The first is fully homogeneous and mainly used for testing purposes. The other is a procedurally generated smoke column.

2.1 Light Linked List

In the implementation of the "Light Linked List" algorithm each light source is represented by a geometrical shape called the light geometry. For example, this light geometry can have the shape of a cone or a sphere. The light emanating from a point will be contained within this geometry.

We construct linked lists as explained in 1.2.2, where each list covers a set of pixels on the screen. An element in the lists represent a light affecting that part of the screen. Each element contains two depth values called the front and back depth. These values are determined by the front- and back-face of the light geometry. It is therefore important the lights geometry is of the nature that for any possible line passing through, it will always intersect two faces and two faces only.

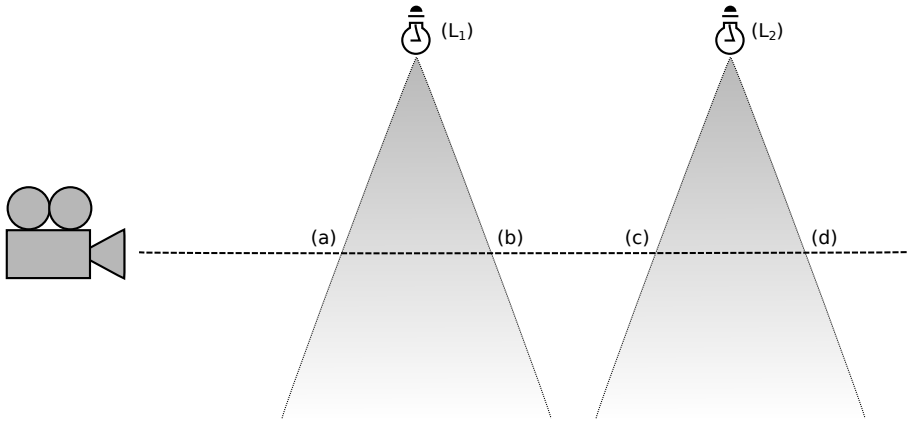


Figure 2.1: Example scene with two lights. **(a, c)** Front-faces of the light geometries. **(b, d)** Back-faces of the light geometries.

When determining whether a point in the scene is illuminated by a light, we can compare the distance between the point and the camera to the distances for the front- and back-face of the light. Figure 2.1 shows an example scene with two lights. Note that the illustration is simplified in order to be represented in two dimensions.

The required amount of memory for this algorithm is scene and view dependent which means it can be hard to predict. The worst case scenario is if every light geometry in the scene is covering the whole screen. However this is unlikely to happen in a practical scenario so allocating less memory than the worst case scenario is a good idea. In this implementation the lists covers 16 pixels on the screen and each element is 16 byte. In the case of 100 light in the scene and a screen resolution of 1920x1080, the worst case scenario would have a required memory space of approximately 207MB.

In each frame before the construction starts, the element counter is reset to zero and all the values in the index buffer are set to negative one which effectively makes all the linked lists empty.

2.1.1 Constructing the Light Linked List

When the linked lists are constructed, the lights are processed one by one. Each light requires two render passes where the only geometry being rendered is the shape representing the light. Because we only render one geometry in each render pass along with the practical resolution being a fraction of the actual screen resolution, these two render passes can be performed at a low computational cost.

The first render pass calculates the depth to the face closest to the camera. This is the front depth. The information is saved for the second render pass where the depth to the face furthest away from the camera, the back depth, is determined. If the front depth is larger then the back depth the camera is within the light geometry and the front depth is set to zero. Continuing with the second render pass the two depth values are entered, together with a reference to the currently processed light source, into a light element. This element is inserted into the light linked list as specified in 1.2.2. When the element is inserted the position is chosen so that the list is sorted based on the front depth in ascending order, i.e. the closest light will be first in the list. After all the lights have been processed in this manner the light linked list is complete and can be used in any following render passes.

2.1.2 Using the Light Linked list

To determine which lights are illuminating a specific point in the scene, a light lookup is performed. We use the fragments x- and y-coordinates of the point to obtain a pointer to the first element in the list via the index buffer. The algorithm loops through the list and compares the depth value of the fragment to the front and back depth. If the depth from the camera to the point is between the two values, it is affected by the light corresponding to the current element. When the algorithm reaches the end of the list or if the fragment depth is smaller than the front depth, all the lights that could affect the fragment have been tested and the light lookup is done. The color of the fragment can be adjusted according to which light are affecting it.

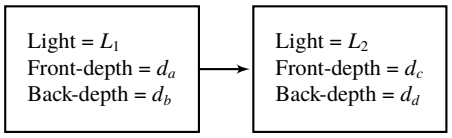


Figure 2.2: An example of linked list in one pixel for the camera in figure 2.1. d_x = distance to face x.

In the example of figure 2.1, a pixel would contain a linked list with two elements. An example of such a list is illustrated in figure 2.2. If we want to find out if a point is illuminated by light one, we would first find the correct list and loop through it until we find element corresponding to L_1. Then we compare the distance to the point with d_a and d_b. If the point is between those values we know it is illuminated by the light.

2.2 Deep shadow maps

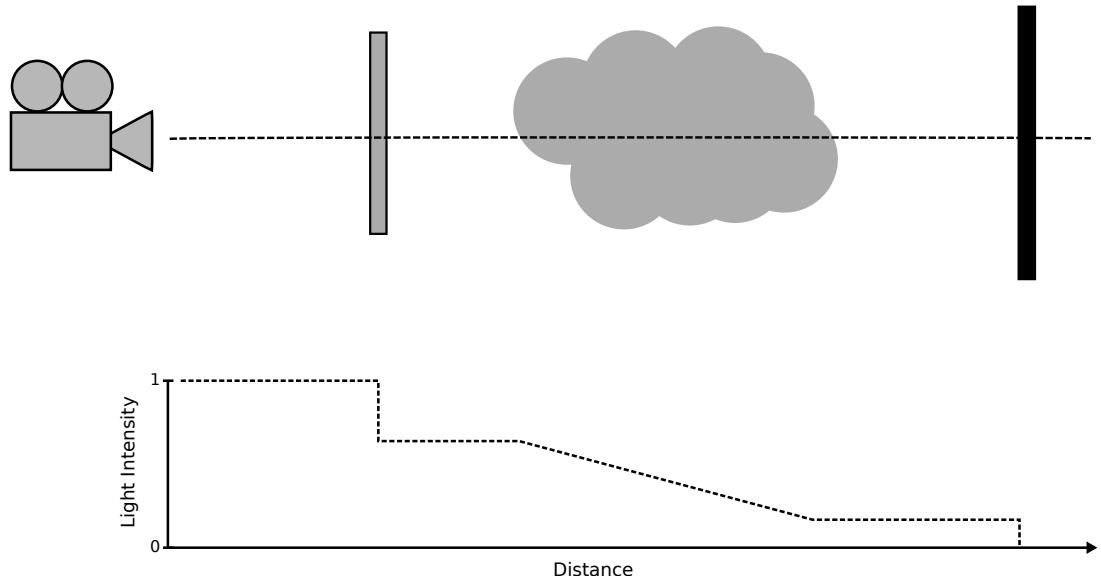


Figure 2.3: Example of light attenuation through different objects. The first object in front of the camera represents a translucent surface. The second a volumetric object. The third an opaque surface.

A deep shadow map is used to represent a drop in light intensity at any possible distance from a light source. Figure 2.3 show an example of this, where the light passes through a transparent surface, a volumetric object and finally an opaque object object. The graph shows how the light intensity drops for the different objects. This is calculated for each "pixel" in our shadow map which stores a linked list. Each element in the linked list corresponds to a drop in light intensity. The example in figure 2.3 would result in a linked list with three elements.

The implementation of deep shadow mapping in this thesis is conceptually based on the algorithm introduced in 1.2.4, however they are technically very different. In order to achieve real-time rendering performance, we opted to not use supersampling when constructing the deep shadow map. The supersampling was mainly aimed at creating good shadows for high detail geometries like hair or fur, however such geometries are rare in real-time environments because of their high computational cost. Another thing that we cut in our implementation was the compression step. Once again we rely on the fact that a real-time environment is less complex and will have fewer points along the camera depth where the light is attenuated. We take advantage of the fact that we can specify different shader programs on a per object basis. This means we don't have to calculate surfaces and volumetric objects in different steps.

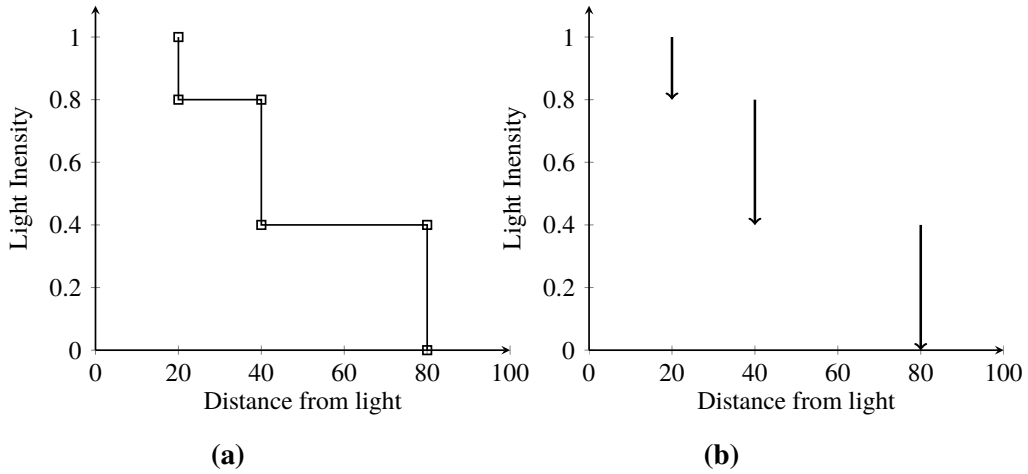


Figure 2.4: **(a)** Visibility function from the original deep shadow map algorithm. A drop in light intensity is represented with two nodes. **(b)** Visibility function implemented in this thesis. A drop in light intensity is represented with only one node.

Our implementation also use a more memory efficient way to represent the light attenuation along the camera depth. Instead of having each node in the visibility function of 1.2.4 represent a fixed light attenuation, we have each node represent a drop in light intensity. This way the amount of points required is effectively halved. Figure 2.4 shows a visual example of this. As each node is represented as an element in a linked list, the number of nodes has a direct correlation with the required amount of memory.

As with the "Light Linked List" algorithm, predicting the required amount of memory can be a hard task. Even more so with deep shadow mapping as the length of the linked lists will also depend on the complexity of objects in the scene. Naturally the more surfaces light passes through the longer the linked lists will be.

The deep shadow map algorithm has many things in common with a regular shadow map. This brings along the inherent problems with the shadow map. Problems that are resolution dependent could even be considered more severe for the deep shadow map since the computational cost for each pixel is higher. However this means that many of the image processing techniques used along with shadow mapping also applies to the deep shadow map algorithm.

In our implementation, the deep shadow map algorithm is performed with one render pass for each light, where all the shadow casting geometries in the scene are rendered. Each rendered object can have its own shader program for construction of the deep shadow map. This makes a flexible system where objects with specific materials can have different effects on the resulting shadow.

2.2.1 Constructing the Deep Shadow Map

First the screen x- and y-coordinates are used to determine the position in the index buffer. When we have access to a list, the current depth of the shadow casting surface is used to find the correct position in the list. Each list is sorted in ascending order by depth. If the new element is located behind another element with zero light pass through, we do not have to insert it. With the position determined, our new element is created and entered into the list. The drop in light intensity contained in the new element is determined by the shader program assigned to the object casting the shadow.

Each element consists of a depth value, a range value, three separate values representing the light pass through for each of the RGB channels and finally there are 8 bits that can be used as flags. The range value is used when it is a volumetric object casting a shadow which lacks a single depth position where the light is decreased. This makes it possible to represent light attenuation trough the volumetric object with only one element in a linked list. Using only one element is quite a coarse approximation. For some highly detailed volumetric objects it might be desirable to have a more accurate approximation. This should be quite easy be achieve by using more elements to represent the light attenuation inside the volumetric object. However, this was not implemented because of time constraints and that it would be hard to test accurately.

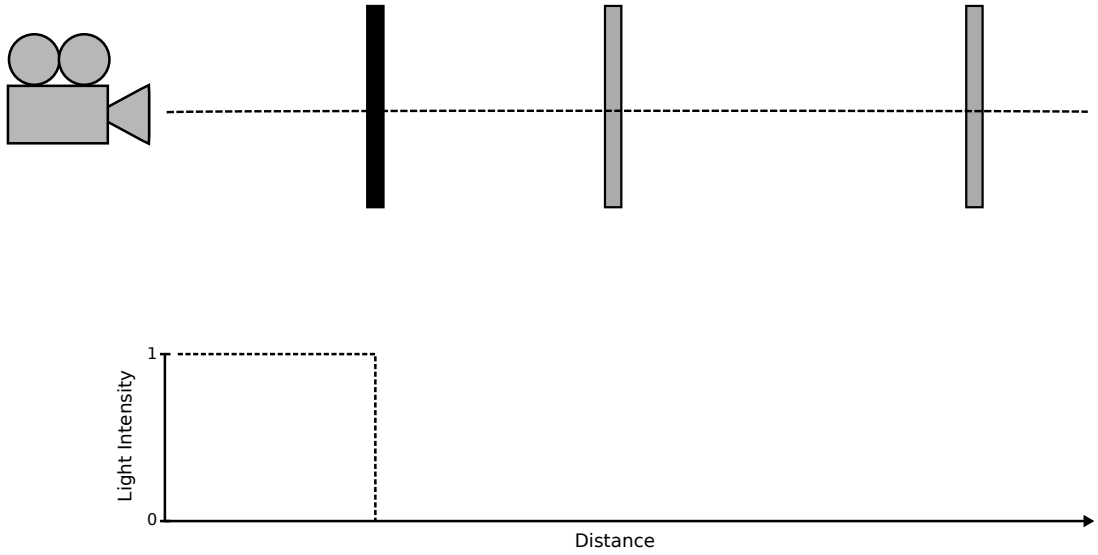


Figure 2.5: Example of light attenuation where the closest object to the camera is opaque.

When constructing the deep shadow map, there is no guarantee of the order in which surfaces will be processed. In the case where we have an opaque surface in front of other surfaces, we would ideally avoid handling the blocked surfaces. Figure 2.5 is an example of this scenario. If the first surface we handle is the one closest to the camera, there is no problem since we can abort an insertion once we know an element is behind another element with zero light pass through.

However, if the first processed surface is behind the closest one, we will enter it into the linked list unnecessarily. When we process the closest surface, the elements behind it in

the linked list will be removed from the list but still be in memory. In figure 2.5 the worst case would be that we process all three surfaces. This would mean we perform three times as many memory operations and take up three times as much space as we actually need. This is of course unfortunate from a performance point of view, it also is a big reason for why predicting the required amount of memory is hard.

2.2.2 Concurrency

There are some scenarios where we run into concurrency issues that has to be addressed. If an object is shaped so it can cast a shadow on itself, we might try to enter several elements into the same list at the same time. This can corrupt some pointers, which means we lose elements from the list. To avoid write collisions in the linked list we implemented a spin lock system. The pointer stored in the index buffer is also used as a lock value. When a thread wants to access a list, it needs to read the pointer from the index buffer. Instead of simply reading the value, the thread replaces it with a value that indicates the list it is locked. If another thread wants to access the list, it has to wait for the first thread to finish. The replacement of the value in the index buffer is done with an atomic operation to avoid write collisions on the pointer.

By doing this we can lock each list separately. If no collision occurs we have not introduced much overhead since we replaced a read with an atomic swap operation. In a practical scenario, write collisions should be fairly rare and therefore the performance cost of dealing with them should not be to severe.

2.2.3 Using the Deep Shadow Map

The deep shadow map is constructed and a shadow lookup can be performed. For each light affecting the current fragment, the position in the shadow map is determined and the list is traversed until the fragments depth value has been reached. The light pass through value for each traversed element is multiplied together to find the final light attenuation. If the range value is not set to zero the light pass through is linearly interpolated based on the range to determine the actual light attenuation.

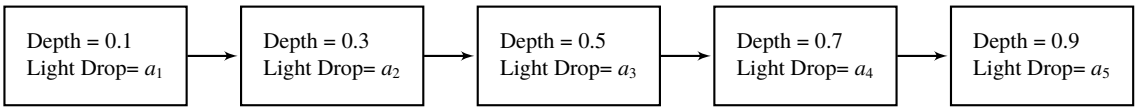


Figure 2.6: An example of linked list in one pixel for the shadow camera.

Figure 2.6 illustrates a linked list stored in a pixel of the deep shadow map. Say we want to find the light intensity of a point located at a distance of 0.6 from the camera. The first thing we do is find the correct list. Lets say this list is represented by figure 2.6. Then we would loop through the list to the third element. As we loop through our list, the light intensity at the point would then be calculated as $L \times a_1 \times a_2 \times a_3$, where L is the light emanating from our light source.

2.2.4 Flags

The flags in the shadow elements are mainly a by-product of 32-bit blocks, there where simply some bits left over. How they are used is up to the shader programs used in the scene. We have implemented two use cases for them. The first is that an object can use the flags to identify a shadow cast from itself, which makes it possible to turn off self shadowing for objects where this is desirable. This is done by setting the flags to a unique identifier associated to the shadow casting object when constructing the element. When performing a shadow lookup of a surface on the same object, it can check the flag to see if a shadow is cast by itself. We can then chose to ignore the drop in light intensity of the current element.

When light passes through several surfaces of the same model it could be desirable to only cast a shadow once. In a similar manner as the first use case, an object can identify a shadow cast by itself. When inserting a new element, we can check the neighbors to see if their shadow is cast by the current object. If so, we can abort the insertion and thus avoid casting more than one shadow from the object.

Chapter 3
Result

This section will start with the specifications of the test setup. Then we will move on to the visual results where we will present some test cases to demonstrate the output of our implementation. After that we continue with performance tests for the two algorithms.

All the test are done in the crytek sponza scene, which is a popular test model for light algorithms. The curtains in the scene have been made transparent to show how light interacts with transparent objects. The resolution for all rendered images is 1024x768 and the test are performed on a linux system using a nvidia 660 gtx graphics card with driver version 346.47 [7].

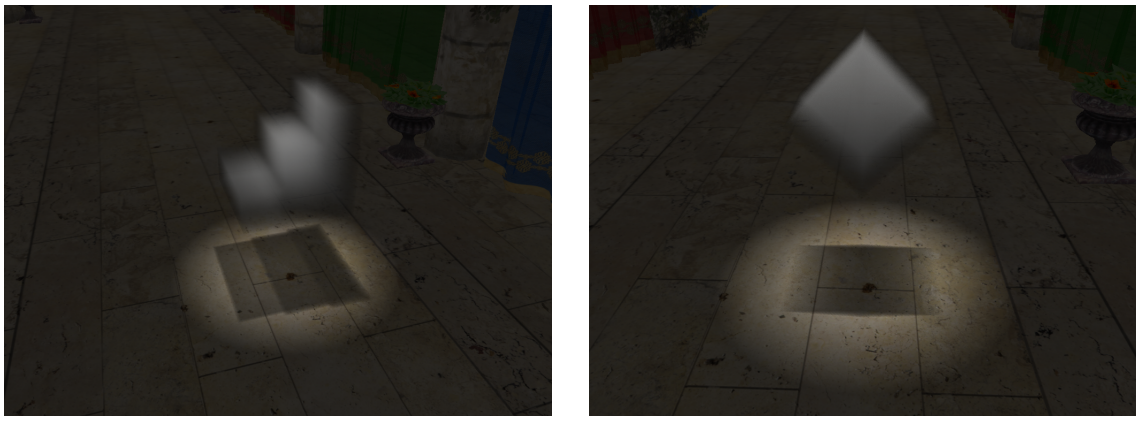
We have the constraint that the algorithms should be run at real-time frame rates. We consider any frames per second (fps) over 30 to be real-time. It should be noted that what is considered real-time is dependent on the application and personal preferences. Figure 3.1 to 3.6 all show the running fps for the corresponding scene in the description. In each frame we precalculate and store data for all the lights and shadows and perform light and shadow lookups for the visible surfaces. There is also some time required for handling the scene, such as updating the lights and moving the camera.

3.1 Visual Results

(a) Runs at ~ 105 fps **(b)** ~ 205 fps

Figure 3.1: Light shining through fabric.

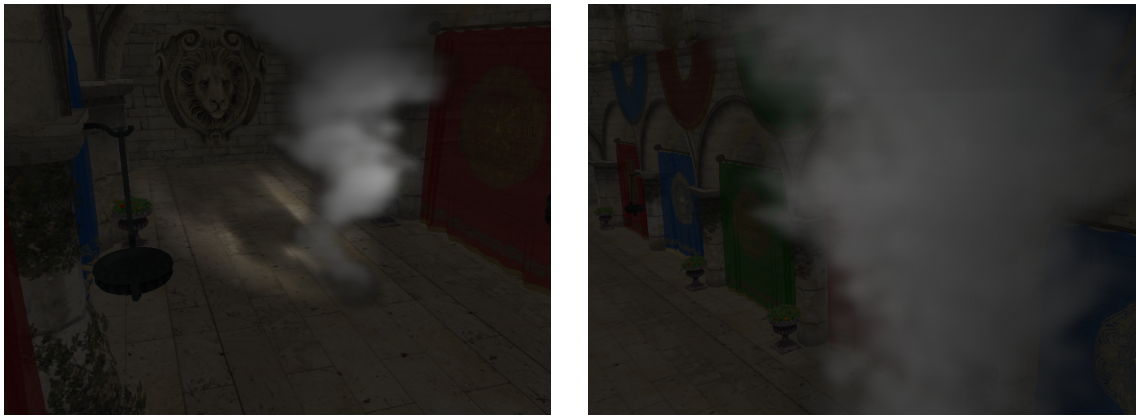
Figure 3.1.a shows a point light behind a curtain. Each of the three illuminated curtains throws a shadow colored after the fabric. This demonstrates how the light is colored by the object it passes through. Figure 3.1.b shows a similar scenario but uses a spot-light and the camera is on the same side of the curtain as the light.



(a) ~ 89 fps **(b)** ~ 150 fps

Figure 3.2: Test cases for volumetric objects.

Figure 3.2 displays three test cases for volumetric shadowing where all the geometries are being illuminated directly from above. Image 3.2.b gives a clear illustration of how the shadow becomes darker when the light propagates through the thicker parts of the volume. The geometry is shaped like a three step staircase and the shadow have three matching areas of shadow intensity. Image 3.2.a illustrates the same property but the geometry is a cube rotated so light passes through areas with different thickness. This makes the projected shadow appear like a gradient instead of discreet steps like in image 3.2.b.



(a) ~ 51 fps **(b)** ~ 22 fps

Figure 3.3: **(a)** Smoke casting shadow. **(b)** Light propagating through smoke.

Figure 3.3 shows light interacting with smoke which can be considered a more complex volumetric object than displayed in figure 3.2. 3.3.a shows a picture of a shadow cast by a smoke column. In image 3.3.b the camera is located slightly behind the smoke column, it illustrates how the light attenuates as it passes through the volume.

Figure 3.4: Shadow from a transparent teapot. ~ 110 fps

Figure 3.4 and 3.5 illustrates some examples of what effect can be applied to the shadow when it passes through a surface. In 3.4 the drop in light intensity is calculated based on the angle between the light direction and the normal vector. This gives a larger drop in light intensity for the edges of the teapot which results in a more dynamic shadow.

Figure 3.5: Shadow colored by texture. ~ 476 fps

Figure 3.5 shows the light passing through a textured quad. This shadow is colored after the texture which makes it possible to project images on other surfaces.

Figure 3.6: Colored shadow interacting with smoke. ~ 49 fps

In figure 3.6 we see light passing through a surface and then a volumetric object. The colored light from the surface can clearly be seen in the smoke column. And the combined shadow of both objects can be seen on the ornamental lion head in the background.

Figure 3.7: Smoke intersecting with other geometry.

Figure 3.7 to 3.9 shows instances where the algorithm produces artifacts and unwanted results. In figure 3.7 we see what happens when a volumetric object intersects with another geometry. Volumetric objects are contained within a bounding box and are rendered with front face culling. This means that the back-face of the bounding box needs to be visible to the camera. Any object inside the bounding box of the volume will block the back face, which stops the volume from rendering. The result is that the smoke will not rendered in those pixel where other objects are inside the bounding box.

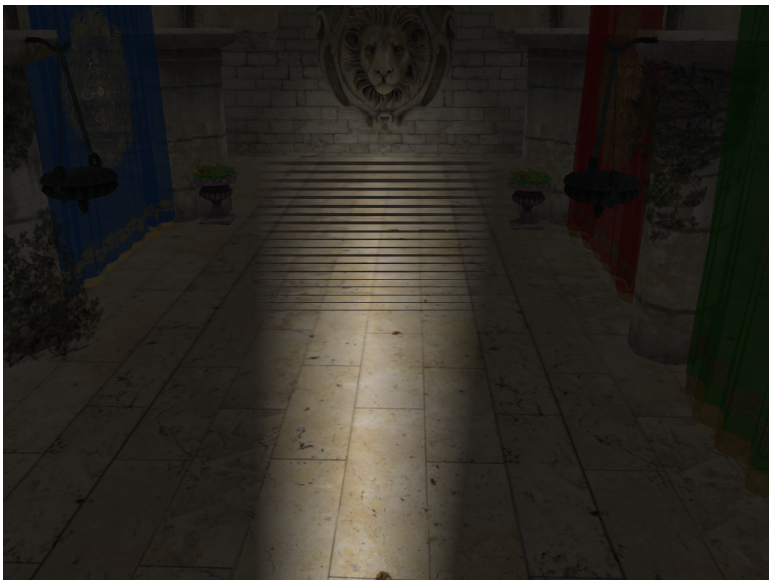


Figure 3.8: Point light in smoke.

Figure 3.8 shows an artifact that is inherited from the shadow map algorithm. Getting a bad result from the shadow lookup may happen depending on shadow map resolution and angle of the incoming light. The artifacts appears as shaded lines on the floor.

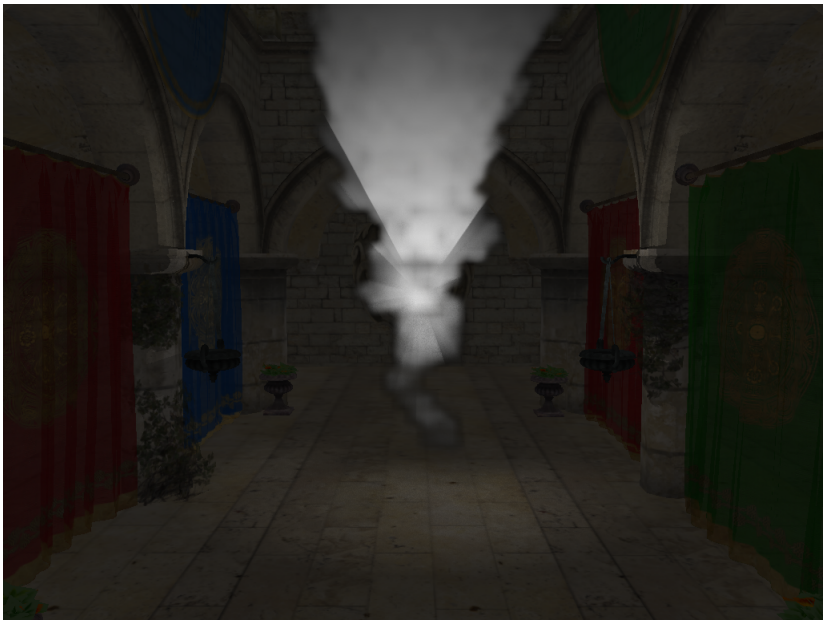


Figure 3.9: Point light in smoke.

In figure 3.9 a point light is located within a volumetric object. The smoke above the light is not self shadowing. Once again, front face culling is the reason. In this case the light do not reach the edge of the volumetric object. This means the back-face is outside the frustum of the shadow camera and no render call for shadow casting is made.

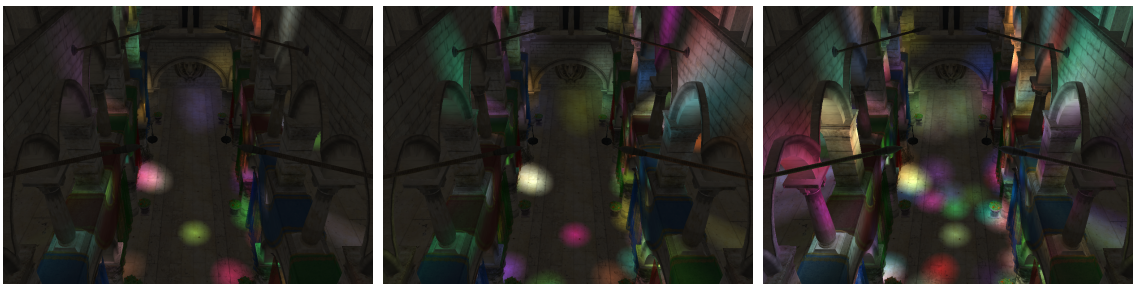
3.2 Performance

The performance is in general good. This is of course highly dependent on the scene and view complexity but the frame rate is for most tested use-cases well above what is required for real-time graphics. There is however one case where performance is a problem. Due to the sampling done when rendering a volumetric object, several shadow lookups have to be performed for each fragment. Depending on how much of the screen is covered by the volume and the sample rate, the calculation cost can easily become very large. So in general, close-ups of illuminated volumetric objects has a much higher performance cost than the rest of the scene.

When testing the performance of the two algorithms implemented in this thesis, there are basically an infinite amount of possible variations of different scenes and setups that could be considered. We have chosen three test cases that we think represents practical and common scenarios. The first test case is for the light linked list algorithm. The second is for the deep shadow map algorithm and the third test case is for both algorithms together. These test cases should give a good idea of the general performance of our implementation.

In each test, the average execution time is measured over 10 000 frames.

3.2.1 Light Linked List



(a) (b) (c)

Figure 3.10: **(a)** Test scene for 50 lights. **(b)** Test scene for 100 lights. **(c)** Test scene for 200 lights.

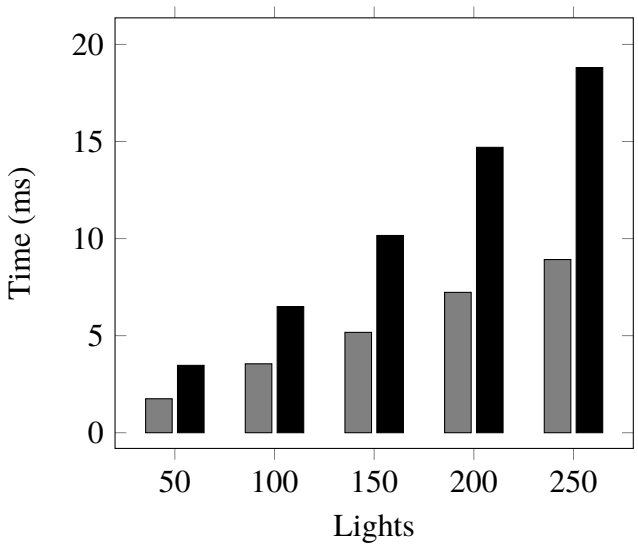


Figure 3.11: The gray bar displays the average construction time for the light linked list. The black bar shows the average total frame time.

This is a test case devised to measure the performance of the "Light Linked List" implementation. Because of that the shadow calculations have been turned off. We test five scenes with 50 to 250 cone-lights. The average time for constructing the light linked lists each frame along with the average total frame time is measured. Figure 3.10 shows the test scene for the different number of lights. The result is displayed in figure 3.11.

It is worth noting that running the scenes with the time measurements off gives a performance boost of about 50%. The reason is that we need to wait for the GPU to finish a calculation before staring another. If this is not done we can not separate one calculation from the others, making any measurement meaningless.

3.2.2 Deep Shadow map

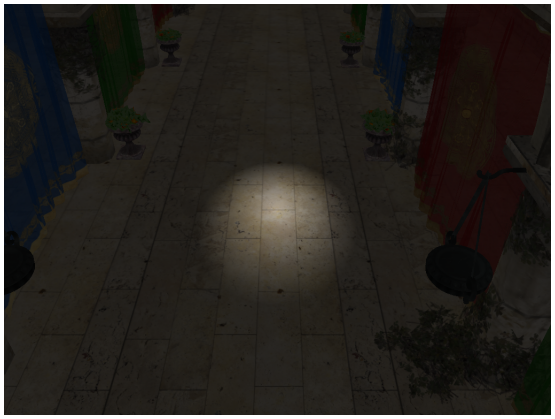


Figure 3.12: Test scene for deep shadow map performance.

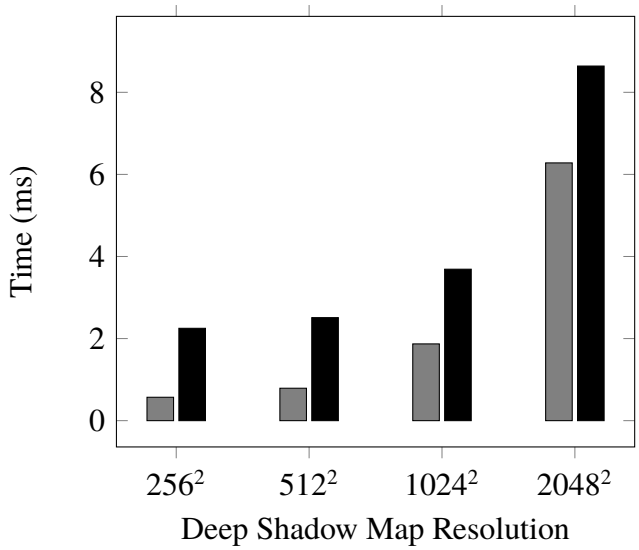


Figure 3.13: The gray bar represents average construction time for the deep shadow map. The black bar shows the average total frame time.

Figure 3.12 shows the setup for this test case. A single spot light intersects a flat surface. The test case is performed with four different resolution on the shadow map. The execution times for each resolution can be seen in figure 3.13. Running the test case without measuring time has a significant increase in performance for the lower resolution, 256^2 runs about 50% faster while 2048^2 gets a boost of about 8%.

3.2.3 Light and Shadow

(a) **(b)** **(c)**

Figure 3.14: **(a)** Test scene for 5 lights. **(b)** Test scene for 15 lights. **(c)** Test scene for 25 lights.

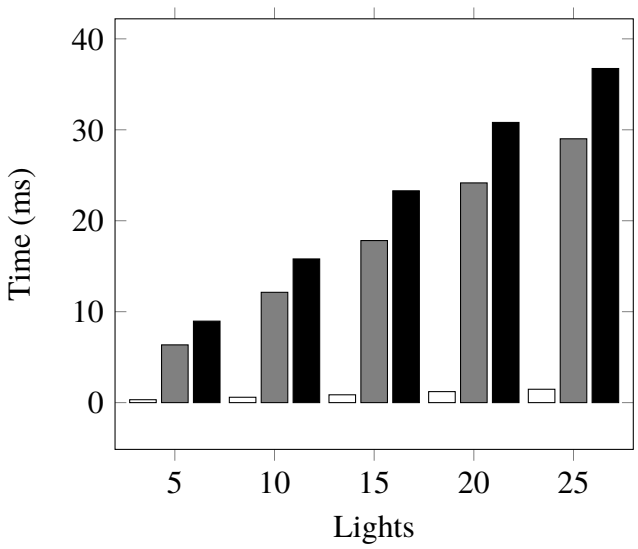


Figure 3.15: The white bar displays the average construction time for the light linked list. The gray bar represents the deep shadow map construction time. The black bar shows the average total frame time.

In this test we measure the execution times for both the algorithms combined. Each light has a shadow map resolution of 512^2. The images in figure 3.14 shows examples of the test scenes. The test-case has five different scenarios with 5, 10, 15, 20 and 25 lights. Figure 3.15 displays the result of the test. When turning off the time measurements, the performance boost is about 29%.

Chapter 4
Discussion

Overall the algorithms can be run with real-time frame rates which we consider a success. The exception to this is when taking many samples of volumetric objects, as such the problem does not lie in the construction of the deep shadow map but in the sheer number of required lookups. This could be mitigated by simply optimizing the method of sampling a volume. Another way to increase performance would be to be stricter with the memory management. Reducing the amount of data stored in each list element the required memory space and bandwidth usage could be minimized. To achieve this, reducing the float precision to store two floats in on 32-bit block was considered during the development phase. However this was not implemented as a high precision is also desirable.

The error introduced by measuring the execution times does provide some uncertainty to the results but they give a good indication of how much the work load increases and how large part of the frame time is spent with the measured algorithm.

The construction time for the light linked lists seems to be increasing roughly linearly with the number of lights being processed. Although the execution time of scene management and light lookup are not separated, the result indicates that the cost for light lookup also increases linearly. When examining the algorithm, one would assume that the execution time would increase linearly with the number of screen pixels each light covers. The test case is of a practical scenario where all the lights are placed randomly in the scene. This causes different lights to cover different amounts of pixels, which introduces some variance to the execution times. Having a more synthetic test case where each light is identically placed in the scene could yield a result closer to a strict linear increase in calculation time.

When looking at the execution time for the deep shadow maps, the result is a bit harder to interpret. For each power of two increase in the resolution, the amount of work needed to construct a deep shadow map is expected to increase by a factor of four. This is not the case and the increase is less then expected. This can be explained with the other factors having too much of an impact on the execution time, especially for the lower resolutions. When looking at the two largest resolutions, the increase is closer to what is expected but does not reach it. It might be that rendering at even higher resolutions would yield more insight into the time complexity for the algorithm but at that point the memory space requirements became unfeasible for the test system.

The third performance test case is probably closest to what one would see in a modern computer game. It shows that a large majority of the time in each frame is spent on constructing the deep shadow maps. Once again he execution time looks to increase linearly. This would be expected since the two algorithms seams to behave similarly when measured separately.

Visually the algorithms mostly works as expected. Light interacting with volumetric objects and the ability to customize shadows are very nice features. This creates a more believable and visually appealing image. The problem with surfaces blocking volumetric objects when they intersect is most likely the worst of the artifacts. There is probably many ways to solve this problem. One solution would be to implement the rendering technique for order independent transparency as presented by the paper in 1.2.2.

The problems inherited from the general shadow map technique have no known solution. The impact of these problems is limited and there is a lot of experience in dealing with them, providing workarounds that can be applied.

When light is attenuated in volumetric object, the decrease in light intensity is represented as a linear function across the whole object, this approximation might be bad for complex objects with a high variation in density. To increase the accuracy of the light attenuation, the algorithm could be expanded to create several elements in the deep shadow map. Depending on the number of extra elements chosen this could be a low cost upgrade to the algorithm that produces a better visual result.

Another way to increase the accuracy could be to generate the volumetric objects on the GPU using noise functions. This way the light attenuation inside a volumetric object could be calculated when doing the shadow lookup. This could yield an even more accurate visual result. Also it might have additional benefits with regards to the artifacts in figure 3.7 and 3.9, since this could make the shadow casting algorithm for volumetric objects independent of geometrical faces.

There are many implementation details that could affect the result and therefore a lot of room for further optimization. One obvious optimization would be to look into whether the required amount of data for each list element could be lowered.

One of the biggest problems with the algorithms is the difficulty in estimating how much memory they will use. The problem is to some extent inevitable as the basic idea of the thesis was to work around "flat" data structure of textures and instead store data with regards to a dynamic depth. As a direct result of this dynamic structure, the memory requirements of the algorithm will be unbounded.

The memory estimation problem is in general worse for the deep shadow maps then for the light linked lits. Since the deep shadow maps are naturally larger and in the worst case scenario, we will store elements for surfaces that are not visible to the shadow camera. It could be possible to implement a system where the memory space of such element are returned so that they can be reused. However, this would most likely come at a large computational cost. Another and better solution would be to avoid the problem. This might be done by introducing an extra render pass for shadow calculations. In this pass a regular shadow map to all the opaque object could be computed and then used to cull redundant elements in the deep shadow map.

Together the two algorithms make a solid foundation for light and shadow calculations. Compared to the common methods today they will naturally be more costly in performance and memory. However they do provide more usable data that makes it possible to handle transparent object in a better way. Since that data provides general information about the scene, having access to it could be a great benefit for other algorithms. One example is volumetric light scattering which could benefit from the precomputed light volumes.

Chapter 5
Conclusion

The goal of this master thesis was to investigate the usage of linked lists on the GPU for illumination and shadow casting of transparent objects in a real-time environment. This was done by using the "Light Linked List" and "Deep Shadow Mapping" algorithms.

The visual results are pleasing except for a few scenarios where the rendered images contain artifacts. The visual problems specific to this implementation could, with a high probability, be corrected with further development. The performance results is also good as the algorithms can in general be run at real-time speed. There is no getting around that deep shadow maps is more costly than regular shadow maps as they simply contain more data. But if shadow casting for transparent objects is a desired feature, deep shadow mapping is a viable technique.

Overall we consider the project a success. Using these algorithms provides a solid foundation for rendering light and shadows in general and enables features that have previously been restricted in real-time graphics. The system is very flexible allowing us to casting complex shadows and enables high customization with regards to the materials being illuminated.

Bibliography

[1] Tomas Akenine-Moller, Tomas Moller, and Eric Haines. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2002.

[2] Abdul Bezrati. Real-time lighting via light linked list. Presentation at SIGGRAPH2014, 2014.

[3] 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 *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 21–30, New York, NY, USA, 1988. ACM.

[4] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[5] JMonkeyEngine. Home page for jmonkeyengine. `http://jmonkeyengine.org/`. Accessed: 2015-04-10.

[6] Tom Lokovic and Eric Veach. Deep shadow maps. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 385–392, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[7] Nvidia. Graphics card specification. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-660/specifications`. Accessed: 2015-04-12.

[8] 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*, EGGH-HPG'12, pages 87–96, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[9] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[10] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 197–206, New York, NY, USA, 1990. ACM.

[11] Adam J. Shook. Deep shadow maps from volumetric data on the gpu. University of Maryland Baltimore County, 2011.

[12] OpenGL specification. Shader storage buffer object. `https://www.opengl.org/registry/specs/ARB/shader_storage_buffer_object.txt`. Accessed: 2015-04-10.

[13] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pages 270–274, New York, NY, USA, 1978. ACM.

[14] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. In *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR'10, pages 1297–1304, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-03-20

EXAMENSARBETE Rendering Light and Shadows for Transparent Objects
STUDENT Isak Lindbäck
HANDLEDARE Michael Doggett (LTH)
EXAMINATOR Flavius Gruian (LTH)

Lights and Shadows in 3D Graphics

POPULÄRVETENSKAPLIG SAMMANFATTNING **Isak Lindbäck**

A big challenge in computer graphics is calculating light and shadows. Common methods today do not produce these correctly for transparent objects. The objective of this master thesis is to solve that.

To do so we combine a newly presented light algorithm together with a new approach to an old shadow algorithm. By doing this we make it possible to correctly illuminate and create complex shadows for objects such as glass, thin fabric, smoke or fog. This enables us to create a more visually appealing picture when dealing with transparent objects. It is especially important when handling volumetric objects such as smoke as it can now cast a shadow on itself. This means that when the light shines trough the smoke it will gradually decrease in intensity the deeper it gets into the smoke.

Our method also makes it possible to color the light as it passes trough a medium. For example shining a light through a blue colored piece of glass will create a blue light. This also works for complex patterns such as those found in stained glass windows where the differently colored parts of an object will produce a different colored light. Thus it is possible to project an image in a similar fashion to how a slide projector works.

Performance is of utmost importance as the algorithms is targeted at interactive real time media such as computer games. To make this possible we utilize the powerful processing power provided by modern graphics cards.

This first algorithm we use is called ”Light Linked List” and is used to determine which parts of the visible scene are illuminated. This technique differs from the ones common in modern 3D graphics by separating data by distance from the camera. This makes it possible to differentiate between light affecting an object close to the camera from another object further away. The second algorithm, called ”Deep Shadow Mapping”, is similar to the first. The main difference is that the separation in depth is performed in relation to a light source. This enables us to have light pass through objects at different depths from the source.

In conclusion, these algorithms create a flexible system for light and shadow calculations. They make it easier to use transparent objects in computer games as they will behave more natural when interacting with light. This is important for creating a credible visual experience.

Curtains being lit from behind, casting colored shadows.

Light passing through stained glass and a column of smoke.

A transparent teapot casting a shadow which gets darker towards the edges.