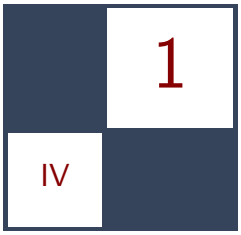

Real-Time Deep Shadow Maps

René Fürst, Oliver Mattausch, and Daniel Scherzer

In offline rendering the algorithm of choice for correctly shadowing transparent objects such as hair or smoke are *deep shadow maps* (DSMs). Algorithms trying to achieve the same effect in *real time* have hitherto always been limited to approximating the solution by depth-peeling techniques. Since the introduction of Direct3D 11, however, it has become feasible to implement the original algorithm using a single rendering pass from the light without introducing any approximations. In this chapter we discuss how to implement a DSM algorithm for rendering complex hair models that runs in real time on Direct3D 11 capable hardware, introducing a novel lookup scheme that exploits spatial coherence for efficient filtering of the deep shadow map.

1.1 Introduction

While real-time (soft) shadows are nowadays routinely used in games, correct shading and rendering of complex hair models, like the ones shown in Figure 1.1, remain nontrivial tasks that are hard to achieve with interactive or even real-time frame rates. The main problem is the complex visibility of hair with super-thin structures that easily creates reconstruction artifacts when using traditional shadow maps. Traditional shadow maps store the distances to the visible front as seen from the light source into each texel of a 2D texture. This means that only the nearest surfaces that block the light are captured. In a second step, a *binary* depth test is performed for each pixel that compares stored texel depth and pixel depth to determine if the pixel is either shadowed or not. While this works well for opaque objects, it has the disadvantage that transparent objects cannot be handled correctly, since every surface behind the visible front is assumed to be fully shadowed.

Shadowing of transparent objects is possible by computing the *percentage of light* that transmits through a material after taking the occlusion of all nearer surfaces along a ray into account. For each texel, a *deep shadow map* (DSM) [Lokovic and Veach 00] stores the transmitted amount of light as a function of

Figure 1.1. Hair models rendered with our real-time DSM algorithm.

depth (shown in Figure 1.2). In reality this amounts to a list of depth values with associated transmittance stored for each texel. When rendering the scene from the camera, the current depth value is searched and used as a lookup into this transmittance function, and the corresponding remaining light is then used

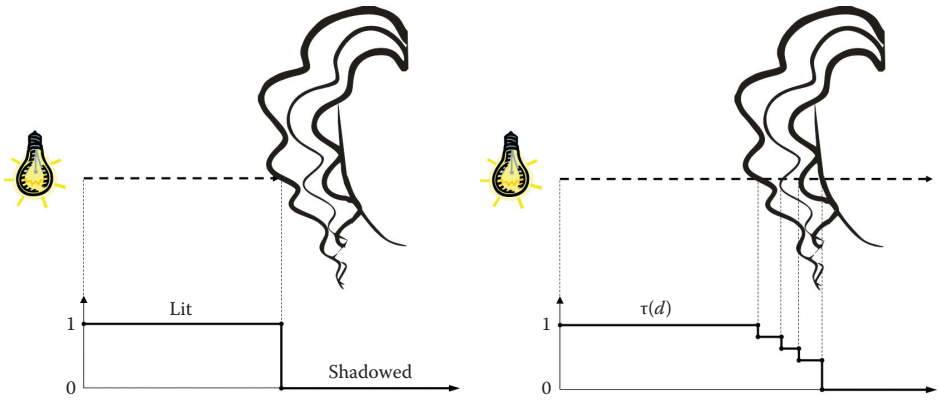

Figure 1.2. Visibility captured for one texel of a shadow map: For traditional shadow mapping (left) only the visible front is captured, while DSMs (right) also account for objects behind it. Here the transmittance function $\tau(d)$ stores the visibility along a ray from the light source.

to correctly shade each pixel. A major part of this article describes how to store and to sort incoming fragments with Direct3D 11 in order to reconstruct the transmittance function and have a suitable representation that allows a fast lookup of the transmittance.

As with most shadow mapping algorithms, deep shadow mapping is also prone to aliasing artifacts. However, most of these artifacts can be overcome by adapting techniques that are also used for traditional shadow maps, e.g., percentage-closer filtering (PCF) [Reeves et al. 87]. We will show how DSMs can be filtered efficiently by exploiting spatial coherence among neighboring pixels. Furthermore, we extend this concept to allow filtering with *exponential shadow maps* (ESMs) [Annen et al. 08].

1.2 Transmittance Function

The major difference between traditional (binary) shadow maps and deep shadow maps is that for each *deep* shadow map test the transmittance function has to be evaluated for the current pixel's depth. Hence this function has to be created and stored first. Creation is made feasible for rasterization hardware by calculating transmittance out of surface opacities. The idea is to rasterize all geometry as seen from the point of view of the light, storing not only the first depth but also all depths (and associated opacities) in a list for each texel. After sorting these lists by depth, the transmittance at a certain depth d at a given texel location can be calculated out of the opacities α_i of all list entries with depth smaller than d (see Figure 1.2) by

$$\tau(d) = \prod_{i=0}^{n(d)} (1 - \alpha_i), \tag{1.1}$$

where $n(d)$ is the number of fragments before depth d. During shadow lookups, the depth of the current fragment can then be used to look up the correct shading value (i.e., percentage of light transmitted) for shadowing the pixels. In the next section we will show how to create and store such a function efficiently on Direct3D 11 hardware.

1.3 Algorithm

Our algorithm can be divided into the following steps:

- Creating list entries. The scene is rendered from the light source. The alpha value and depth value of *all* incoming fragments are stored in a two-dimensional structure of linked lists. Note that the sizes of the linked lists are only limited by video memory, which allows us to store transmittance functions of varying depth complexity.

- Processing the fragments. The fragments are sorted, and the transmittance functions are precomputed for the fragment depths from the individual alpha values (using Equation (1.1)) to allow a fast lookup into the transfer function. Finally the transmittance functions are simplified.

- Neighbor linking. For each fragment, the neighboring fragments at the same position in the linked lists are also linked, in order to quickly find those neighboring fragments that are nearest in terms of depth to the light source and achieve quicker filtering.

- Deferred shadowing. DSM lookups can become quite expensive for complex hair models (i.e., with big depth complexity and transmittance functions with many stored values). Hence, deferred shading is used to render the scene first and then compute the shading value *only once* for each pixel during deferred shading.

- Spatial filtering. At this point, we utilize neighbor links to provide fast lookups for large filter kernels. The DSM idea is combined with two well-known filtering methods for binary shadow maps.

The first three steps can be summarized as building up the DSM structure from the light; the last two steps apply the DSM for the final shading from the camera. Each of these steps will be discussed in more detail in the following sections.

1.3.1 Creating List Entries

One of the main issues for implementing DSMs on a modern GPU is that the amount of per-texel data is dependent on the depth complexity of the scene at the texel position and therefore can vary arbitrarily. In Direct3D 11, this problem can be solved by storing the depth and alpha values of *every* incoming fragment along a light ray in a per-texel linked list. In total each linked list element has the format shown in Listing 1.1. There, `next` represents the index of the next element of a linked list, and it contains -1 if the current element is the last element of the linked list. We also store additional links to the previous element links (making it a double-linked list) to fragments from neighboring pixels that come to use later on.

We create a two-level structure to be able to efficiently insert all fragments into these linked lists during a single rendering pass from the light. All fragments in the linked lists are stored in a structure that we denote as the *list element buffer*. For every pixel, we store the index of the first list element in each linked list in a separate buffer that we denote as the *head buffer*, since it points to the first element of a list. If a linked list corresponding to a shadow map texel is empty, the value -1 is stored in the head buffer. An example is shown in Figure 1.3.

```
struct LinkedListEntry
{
  float depth;
  float alpha;

  int next; // next element in linked list
  int prev; // previos element in linked list

  int right; // right neighbor link
  int upper; // upper neighbor link
};
```

Listing 1.1. Linked list entry structure.

In this example the elements 1 and 6 (2 and 5, respectively) form a single linked list, and the first elements 1 and 2 are stored in the head buffer.

Both structures (head buffer and list element buffer) are stored as Direct3D 11 (`RW`)`StructuredBuffers` and filled by rendering the geometry once using interlocked operations and the buffer counter in the pixel shader. `InterlockedExchange` is used to exchange the head of the linked list to ensure that we do not face any problems regarding parallelization. The buffer counter is used to "allocate" linked list

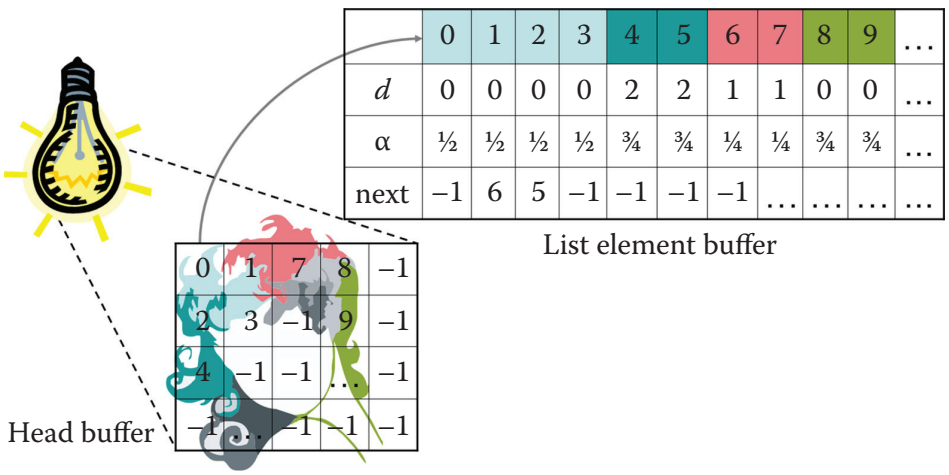

Figure 1.3. We implemented the DSMs as a two-level structure that consists of a per-pixel buffer storing the head of each linked list (head buffer) and the list element buffer for all incoming fragments. For each fragment, we store depth and alpha values, as well as indices of the next and the previous elements, and nearest-depth neighbor links (only the `next` index is shown here for brevity).

```
void ps_main(PS_IN input)
{
  // Allocate a new element in the list element buffer by
  // atomically incrementing the buffer counter.
  int counter = listElementBuffer.IncrementCounter();

  // Store the required information and apply a depth bias.
  listElementBuffer[counter].depth = input.posToShader.z + bias;
  listElementBuffer[counter].alpha = input.alpha;

  //pixel screen coordinate to buffer index
  int index = (int)(input.pos.y * Width + input.pos.x);

  int originalVal;
  // Atomically exchange the element in the head buffer.
  InterlockedExchange(headBuffer[index], counter, originalVal);

  // Create the link to the existing list.
  listElementBuffer[counter].next = originalVal;
}
```

Listing 1.2. Concurrent way of adding a new fragment to the list element buffer and updating the head buffer in a pixel shader.

elements in parallel. The pixel shader for filling both head buffer and list element buffer with a new incoming fragment is shown in Listing 1.2.

1.3.2 Processing the Fragments

Note that up to now the list entries neither are sorted nor contain the final transmittance. These processing tasks are the purpose of this step. The sorting of all fragments with respect to their depth is done in a separate compute shader (executed for every pixel). We load a single linked list per pixel into a local array and do a local sort, which does not require any shared memory. Since neither compute shaders nor OpenCL support recursion yet, a sorting algorithm like quick sort is not very well suited to the GPU architecture. Instead we use *insertion sort* due to its simplicity and because it is known to be very fast on small arrays (as is the case for models with reasonable depth complexity). This fact was confirmed in our experiments, where insertion sort yielded about two times the performance of a nonrecursive version of quick sort. Next we convert the alpha values in the linked list into a transmittance function according to Equation (1.1). This means that we pre-multiply transmittance for each of the depths d_i from the linked list and store the transmittance at each fragment instead of the alpha values. Note that this step is done to accelerate spatial coherent lookups for filtering later on. Here, having to traverse the list from the head to reconstruct the transmittance for each filter sample is exactly what we want to avoid. Furthermore, the `prev` links are stored in this step of the algorithm in order to create a double-linked list.

To accelerate the lookup time, we have to simplify the transmittance function. In the case of volume data, a sophisticated algorithm for handling the inclination of the transmittance function has been proposed in the original DSM paper [Lokovic and Veach 00]. In the case of hair rendering, however, we deal with a simple, piecewise constant version of the transmittance function. Hence, it turned out that a simple but efficient optimization strategy is to merely cut off the transmittance function after its value does not change significantly any more, i.e., if $\prod_{i=0}^{k+1}(1-\alpha_i) - \epsilon < \prod_{i=0}^{k}(1-\alpha_i)$. In our experiments the frame time has been improved by approximately 40% using an ϵ of 0.001, which does not visibly compromise the quality of the shadows. Note that a more sophisticated GPU-friendly compression scheme has been proposed by Salvi et al. [Salvi et al. 10] that limits the transmittance functions to a fixed size and works for shadowing both hair and participating media. This method could alternatively be used instead of the simple truncation, and we believe that it would work well in combination with our neighbor-linking approach (possibly further accelerating the lookup time).

1.3.3 Neighbor Linking

In this step we store links with each entry (fragment) of a linked list to those entries (fragments) in the neighboring linked lists that are closest to it in terms of light-space depth. The reasoning behind this step is that for adjacent pixels in screen space, it's very likely that they have approximately the same depth in light space due to spatial coherence. During filtering, the links enable direct access to the depth-nearest neighbors of a fragment in the DSM. Once we found a fragment corresponding to a given light-space depth, the other corresponding fragments of nearby filter samples can be found very quickly. Note that we only create links for the great majority of pixels where the assumption of coherence of depth values holds.

In order to keep the memory overhead as low as possible, *only* links to the left and upper neighbor are stored. This suffices for computing all other filter samples when starting from the lower-left corner of a rectangular filtering window. For creating the links, in each thread (note that there is one thread for each shadow-map texel) we simultaneously traverse the linked list associated with the texel in question as well as the lists associated with the right texel neighbor and the upper texel neighbor. For each fragment, we traverse each neighboring list until we either find a fragment that is farther in depth or encounter the end of the list. Then, either the last traversed or the previously traversed neighboring fragment will be the one that is closest in depth to the current fragment. We link the current fragment to its depth-nearest neighbors by storing their indices, which we denote `right` and `upper` for each list element. The expense of storing three additional integers per list element (the two neighbor links and the `prev` link) increases the overall memory requirements by about 25%, but this pays off during filtering as will become clear in Section 1.3.5.

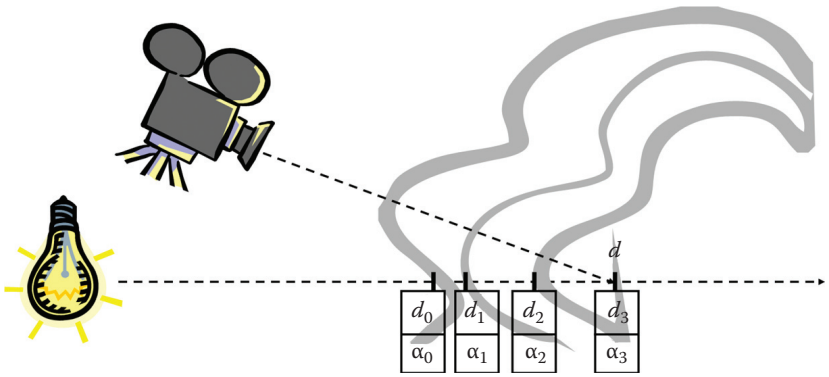

Figure 1.4. To calculate the transmittance for a DSM test evaluation, the opacities α_i of all the surfaces that are nearer than the current pixel's depth d have to be multiplied.

1.3.4 Deferred Shadowing

In the shadowing stage, we search for the depth in the transmittance function, which corresponds to the depth of the current pixel in the eye view (see Figure 1.4). This is done by front-to-back traversal of the linked list corresponding to the current position in the xy-plane. The head of the list is accessed by looking it up in the head buffer. A lookup of the corresponding element in the transmittance function gives the correct transmittance value, which is used to attenuate the shading.

An important point for stabilizing the frame rate and to achieve real-time frame rates is to use a deferred shading pipeline for shadowing. This means that we first render the geometry to store depth and diffuse shading values in render targets before using a single shadow lookup per pixel in the deferred shadowing pass. Consider that with forward rendering, the depth complexity of a hair model potentially requires multiple costly shadow lookups per pixel (and even more when using PCF).

1.3.5 Spatial filtering

Spatial antialiasing is as important for DSM as it is for binary shadow maps in order to achieve high-quality images (see Figure 1.6). Contrary to standard filtering methods for binary shadow maps (e.g., using PCF), we now deal with a list of depth values per pixel. In our case, these depth values are not samples but represent the full transmittance function as it is, which means that we do not need to deal with reconstruction or filtering in z-direction. Therefore we use a 2D filter kernel as in the classical PCF formulation and compare light-space depths of adjacent pixels in screen space.

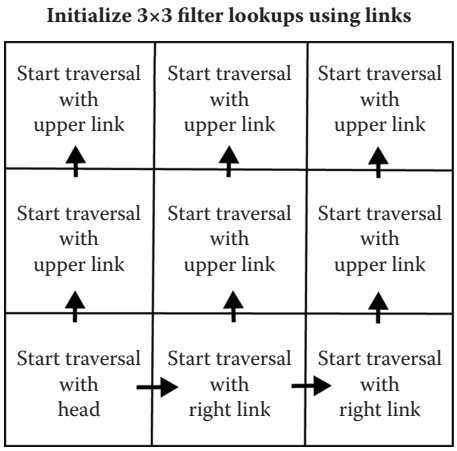

Figure 1.5. We initialize the traversal of the transmittance function using the corresponding link to the closest-depth fragment of the neighboring pixel (going first in y- and then in x-direction).

Since lookup time is slow for DSM, it is essential for DSM filtering to avoid a naive implementation where the overall lookup time grows linearly with the filter size. Searching the whole list of a neighboring pixel for the correct fragment is prohibitively expensive. A binary search would reduce the complexity but is not very suitable for implementation in a compute shader and on linked lists.

Instead we exploit spatial coherence and assume that the neighboring fragment closest in depth has a similar index in its linked list. At this stage we utilize the linking structure from Section 1.3.3, which links list elements closest in depth in order to get a good initial guess for the lookup of a neighboring fragment. This way, the transmittance function has to be traversed from the beginning (the head element) *only once* per pixel, regardless of the used filter size. Once a fragment corresponding to the current depth is found, it is possible to quickly access the neighboring fragments for computing the remaining filter samples.

In our implementation, first the fragment corresponding to the lower-left corner of a filtering window is computed (see Figure 1.5). Next the positions of the fragments linked by `right` and `upper` are used as initial guesses for finding the correct fragment position of a neighboring sample in the DSM (as depicted in Figure 1.4). Then this sample's linked list is traversed by following either `prev` or `next` to determine the correct depth (fragment position in the list). If spatial coherence holds, each traversal only requires a few iterations.

For a filter kernel size of 7×7 and realistic transparency settings, the links speed up the frame times by up to 50% for a frame-buffer resolution of $1{,}280 \times 720$ and up to 100% for a resolution of $1{,}920 \times 1{,}080$ when compared to a brute-force

traversal for each filter kernel sample. Note that this technique scales well with the complexity of the hair model (the more complex, the larger the speedup), because the lookup times become roughly constant as long as there is sufficient coherence. Also observe that this technique even scales well with larger filter kernels, since spatial coherence is only required between neighboring pixels. In case of small transmittance functions (e.g., due to high opacity), the gain from the links are minor, which nevertheless only results in a barely noticeable constant overhead due to the links (of about 2%).

Apart from PCF, we also adapted another antialiasing algorithm, exponential shadow mapping, for use with DSM.

Exponential shadow mapping. The standard binary shadow map test causes antialiasing artifacts since it is effectively a step function that jumps between 0 and 1. Hence *exponential shadow mapping* (ESM) [Annen et al. 08] approximates the shadow test with an exponential function (yielding continuous results between [0..1]). This continuous value is subsequently used to attenuate the shading of a pixel.

The DSM algorithm can be combined with ESM in a straightforward fashion, and we denote this combination as *exponential deep shadow mapping* (EDSM). The resulting transmittance is weighted with the continuous shadow test value. As can be seen in Figure 1.6, EDSM performs much better than PCF in terms of visual quality. Note that while the original ESM algorithm supports prefiltering, this feature cannot be used in combination with DSM since the lookup depth along the transmittance function is not known beforehand.

1.4 Results

We computed all our results on an Intel Core i7-2700K Processor (using one core) and using a Geforce GTX 680. All images were rendered in resolution $1{,}280 \times 720$ and using a deferred rendering pipeline with four 32-bit render targets. The hair model used in our experiments has 10,000 individual strands of hairs and about 87,000 vertices.

SM res	SM	DSM	DSM3	DSM5	EDSM	EDSM3	EDSM5
256	222.8	220.9	192.4	160.3	207.2	166.2	121.6
512	121.2	120.2	111.9	98.4	116.0	99.5	79.9
768	70.7	70.2	66.4	61.4	69.2	60.8	51.0

Table 1.1. This table compares typical FPS values for binary shadow mapping by using only the first fragment of a DSM for shading (SM), our method without filtering (DSM) and with PCF (DSM3 and DSM5), and our EDSM algorithm using different filter kernel footprints. The number after the algorithm's name is the filter kernel size (e.g., a 3×3 kernel for EDSM3).

Figure 1.6. DSMs without filtering (left) exhibit resampling artifacts. Smoother shadows can be achieved with PCF (center), while ESM provides even higher quality (right).

In Table 1.1 we compare the timings for a special version of binary shadow mapping (simulated by using only the first fragment of a DSM for shading), DSMs, and DSMs using a 3×3 and a 5×5 PCF kernel size, respectively, and show the comparison for several shadow map resolutions. The comparison to this version of binary shadow mapping demonstrates the overhead of the DSM lookups (using both optimizations, i.e., neighbor links and truncation of the transmittance function). While the overhead is more pronounced for small shadow maps, it becomes small in relation to the DSM creation for increasing shadow map size.

In Figure 1.6 we compare the quality of DSMs without filtering (left), with PCF (center), and using ESM (right). As can be seen, PCF performs solid antialiasing, while EMS improves the rendering quality even more. Furthermore, while all DSM methods require a depth bias to avoid Z-fighting artifacts, ESM needs significantly less bias for artifact-free rendering than unfiltered and PCF rendering.

1.5 Conclusions

We presented an optimized implementation of deep shadow maps for complex hair models that achieves real-time frame rates by employing new features of current graphics hardware. Note that our implementation requires only Direct3D 11 shader features and compute shaders, which makes our algorithm attractive in environments where GPUs from different vendors are used. In our experiments it turned out that the best DSM quality can be achieved by combining it with ESM. While interactive applications are the main target for this algorithm, we also see applications in the movie industry, where such a real-time DSM implementation could save valuable production time and provide immediate feedback to the artists.

1.6 Acknowledgments

We want to thank Cem Yuksel for the permission to use his hair models and Murat Afsharand for his head model. All models are available at Cem Yuksel's website, www.cemyuksel.com/research/hairmodels.

Bibliography

[Annen et al. 08] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. "Exponential Shadow Maps." In *Proceedings of Graphics Interface 2008*, pp. 155–161. Toronto, Canada: Canadian Information Processing Society, 2008.

[Lokovic and Veach 00] Tom Lokovic and Eric Veach. "Deep Shadow Maps." In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 385–392. New York: ACM Press/Addison-Wesley Publishing Co., 2000.

[Reeves et al. 87] William T. Reeves, David H. Salesin, and Robert L. Cook. "Rendering Antialiased Shadows with Depth Maps." *Computer Graphics (SIGGRAPH '87 Proceedings)* 21:4 (1987), 283–291.

[Salvi et al. 10] Marco Salvi, Kiril Vidimče, Andrew Lauritzen, and Aaron Lefohn. "Adaptive Volumetric Shadow Maps." In *Proceedings of the 21st Eurographics Conference on Rendering*, pp. 1289–1296. Aire-la-Ville, Switzerland: Eurographics Association, 2010.