Real-Time Approximate Sorting for Self Shadowing and Transparency in Hair Rendering

Erik Sintorn[*]
Chalmers University Of Technology

Ulf Assarsson[†]
Chalmers University Of Technology

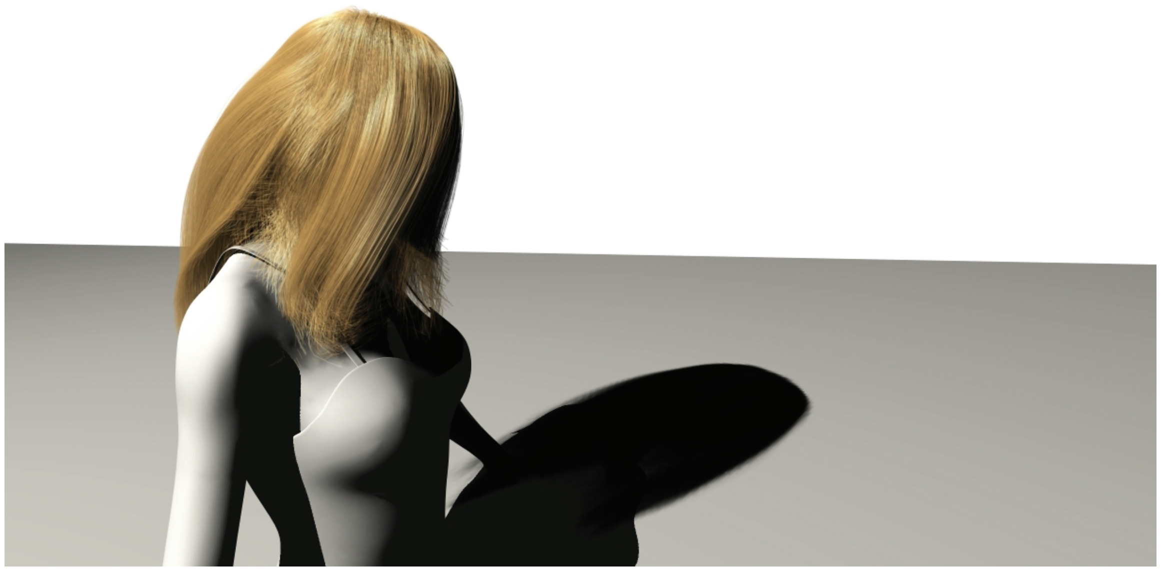


Figure 1: About half a million line segments rendered with 256 Opacity Map slices and approximate alpha sorting rendered in 7.5 fps.

Abstract

When rendering materials represented by high frequency geometry such as hair, smoke or clouds, standard shadow mapping or shadow volume algorithms fail to produce good self shadowing results due to aliasing. Moreover, in all of the aforementioned examples, properly approximating self shadowing is crucial to getting realistic results. To cope with this problem, opacity shadow maps have been used. I.e., an opacity function is rendered into a set of slices parallel to the light-plane. The original Opacity Shadow Map technique [Kim and Neumann 2001] requires the geometry to be rendered once for each slice, making it impossible to render complex geometry into a large set of slices in real time. In this paper we present a method for sorting n line primitives into s number of sub-sets, where the primitives of one set occupy a single slice, in $O(nlog(s))$, making it possible to render hair into opacity maps in linear time. It is also shown how the same method can be used to roughly sort the geometry in back-to-front order for alpha blending, to allow for transparency. Finally, we present a way of rendering self shadowed geometry using a single 2D opacity map, thereby reducing the memory usage significantly.

Keywords: hair rendering, opacity maps, deep shadow maps

[*]e-mail: erik.sintorn@chalmers.se
[†]e-mail:uffe@ce.chalmers.se

1 Introduction

Rendering correctly shadowed hair, smoke, fur, foliage and clouds is a hard and computationally expensive problem and to do so in real time has until recently been near-impossible. Hair, which we will concentrate on throughout this paper, is often rendered as hundreds of thousands of lines, representing very thin segments of hair strands each of which may or may not occlude any of the other segments. To further complicate the problem, blonde or light hair strands are highly transparent making correct shadowing impossible with shadow maps [Williams 1978] or shadow volumes [Crow 1977].

In previous work, Opacity Maps have been used to solve the problem. The Opacity Map is a 3D texture where each slice contains a sampling of the opacity at a certain distance from the light source. Nguyen et al. show how to render and use 16 opacity map slices in real-time but to achieve good quality renderings, many more slices are needed. Using too low a number of slices not only produces banding artifacts, but also does not correctly capture the self-

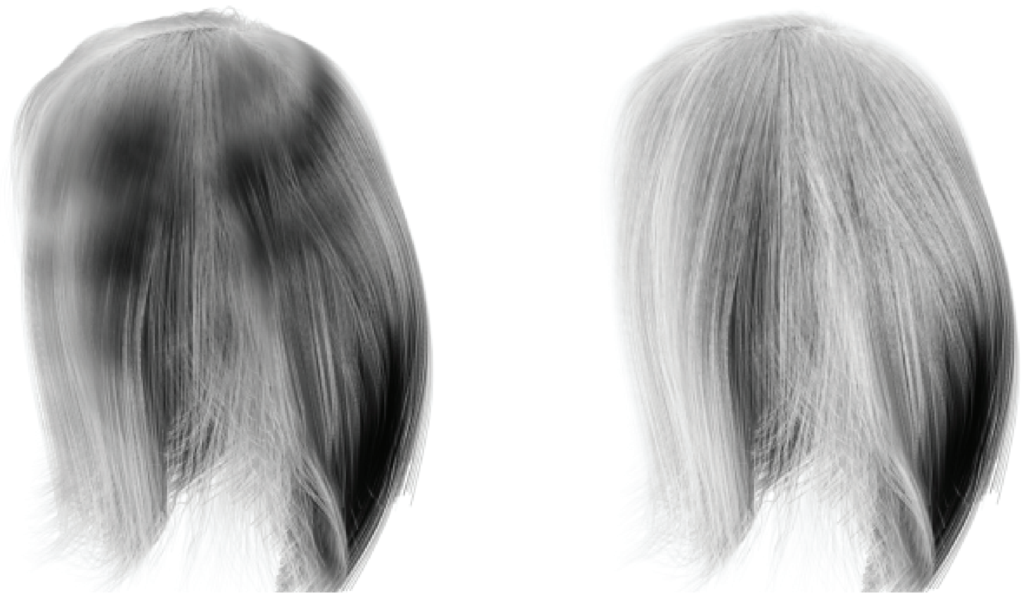

(a) With 16 Opacity Map slices (b) With 256 Opacity Map slices

Figure 2: The same model rendered under the same conditions with 16 and 256 Opacity Map slices. Here, for each fragment we render only the visibility calculated from the opacity map.

shadowing detail required for rendering for example hair (See Figure 2).

In this paper, we show how to sort the hair-geometry into sublists that contain only primitives belonging to one slice and how this can be used to solve both the problems of opacity map rendering and transparency. A novel GPU-algorithm based on quicksort is presented that can be used to divide the original list of lines into depth-ordered sublists from the lights point of view and render these, with additive blending, front to back, copying the result into a new slice of a 3D texture for every list, to produce the Opacity Map. We then use the same sorting algorithm to roughly sort the geometry in depth order from the camera's point of view and render the geometry back-to-front to the screen, achieving high quality transparency for the alpha blended hair strands. Finally, we show how the alpha blending can be sacrificed to allow for rendering the hair with self shadowing without using more than a single 2D texture for storing the opacity map.

2 Previous Work

An early solution to the problem of self shadowing in hair, fur and smoke was Lokovic and Veach's Deep Shadow Map algorithm [Lokovic and Veach 2000] where they in contrast to the standard Shadow Map algorithm, for each pixel in the shadow image plane, stored a visibility function instead of a simple step function. The visibility at a certain depth is approximated by a piecewise linear function and the light attenuation for any fragment can quickly be found by searching in the stored node values. The creation of the Deep Shadow Maps is not designed for real-time applications, however. In a recent paper by Hadwiger et al. [Hadwiger et al. 2006] the GPU is used to generate the Deep Shadow Maps for opaque objects. They achieve high quality results, but performance is too low for a real-time application. Also, their method does not support semitransparent geometry such as hair on current hardware. Opacity Shadow Maps [Kim and Neumann 2001] is a similar technique but where the opacity is instead sampled at regularly spaced depths and stored in a 3D texture. To generate this texture, the geometry is rendered once for each slice, each time moving the far clip-plane away from the light source. At the time the paper was written, it was impossible to render even a small number of slices in real time. In the Nalu demo [Nguyen and Donelly 2005] Nguyen et al. use 16 slices of opacity maps in real time by encoding each slice in one color channel of one of four MRTs. Thus, they can render the opacity of each slice in a single pass, but to obtain an opacity for a certain fragment, they need to sum the opacity values of all contributing slices in the fragment shader, which would quickly become a bottleneck as more slices were used. To reduce the number of slices required for good quality images, Yuksel et al. [Yuksel and Keyser 2007] suggest using a depth map to find a per-pixel starting depth for the slice distribution, but it's not clear how this will help in situations where, for example, some geometry is close to the light but the majority is far away.

Alpha blending is a common technique in real-time computer graphics to simulate the effect of semi-transparent, non-refractive materials, and for rendering sub-pixel sized geometry. Since hair-strands are rendered as very thin cylinders, each strand should only contribute slightly to the pixels being drawn, or the hair will look too thick. This can be alleviated somewhat by using a good multisampling scheme, but to our knowledge the only technique that correctly captures the thinness of hair is to draw the strands back to front with a very low alpha value. Hair, especially blonde hair, is also highly transparent, which in itself requires alpha blending to be done correctly no matter how good multisampling is available. Sorting the geometry by depth has for a long time been considered impossible for complex scenes in real time, and other solutions to the alpha blending problem have been used. One popular solution is the *depth peeling* algorithm, described in [Everitt 2001]

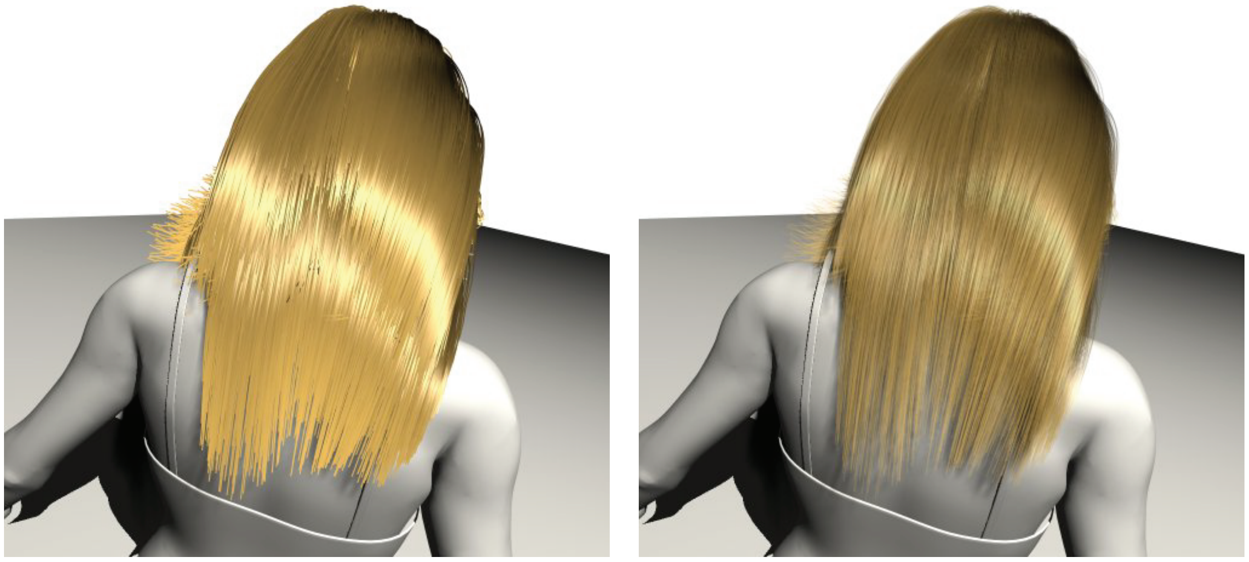

(a) Hair rendered without alpha blending. (b) Hair rendered with alpha blending ($\alpha = 0.2$).

Figure 3: Hair rendered with and without alpha blending.

and [Mammen 1989]. The scene is then rendered several times, each time using the depth buffer of the last pass as the qualifier for the depth test. In this way, each depth layer in the image is peeled off until an occlusion query responds that no fragments are drawn. This will require as many passes through the geometry as there are primitives occupying the same fragment in the scene, making it unsuitable for rendering hair, where the geometry is complex and hundreds of hair strands are likely to contribute to the same pixel. Recently, an acceleration of the depth-peeling technique was presented [Liu et al. 2006], where the fragment shader is used to apply several peelings in a single pass. Even with only around 20 transparent layers and a few thousand polygons, however, the technique does not run in real-time on a GeForce 6800.

Another factor of great importance when rendering hair is obviously the shading model used. Two different models have been widely used, the first being Kayija and Kay's phenomenological model [Kajiya and Kay 1989] which provides a simple formula for simulating the diffuse component and first specular reflection of a hair strand, when modeled as an infinitely thin cylinder. A more complex, and more physically correct, model was presented by Marschner et al. [Marschner et al. 2003]. Here, two additional phenomena are taken care of, namely the light that due to the transparency of the strand reflects on the inside of the strand, creating a colored highlight at some offset from the primary highlight, and the light transmitted through the hair creating a strong forward scattering component. For the sake of simplicity, the Kayija Kay model has been used through this work.

3 Algorithm

Our solution to speed up both opacity map rendering and transparency sorting is to quickly sort the input geometry, on the GPU, based on the primitives' distances from some plane. In both cases, the geometry need not be completely sorted, but simply partitioned into sufficiently thin slices. We first introduce a GPU based quicksort implementation that does such a partitioning, in $O(n \log s)$, where n is the number of primitives and s is the number of slices, making it trivial to then render the geometry either to the individual opacity maps, or, in back to front order to the screen in linear time. In the last section of this chapter we suggest a way of rendering the roughly sorted geometry to the screen in front-to-back order from the light's point of view while keeping only the currently needed opacity map in memory, making it possible to render self-shadowing hair without using more memory than that required by a standard shadow map.

3.1 Quicksorting lines on the GPU

The original Quicksort algorithm [Hoare 1961] is fairly straightforward. A a pivot element is chosen from the list. The list is then reordered so that all elements smaller than the pivot element come before that element and all elements that are larger come after it. This process is then recursively repeated on the two lists on either side of the pivot element. We can use a similar approach to sort points into regularly spaced subsets in order of their distance from some plane:

1. Find the bounding box of the points and the $near$ and far distance of that box from the lightsource. The $middle$ distance is $near + (far - near)/2$
2. Now partition the points into two subsets so that one subset contains all points closer than $middle$ and the other subset contains the rest.
3. Recursively partition the two subsets using $near = near$ and $far = middle$ for one of them and $near = middle$ and $far = far$ for the other.
4. Terminate when the correct number of subsets are generated.

Note that while completely sorting a list with quicksort has a bad theoretical worst case complexity ($O(n^2)$), using quicksort to partition n elements into s sublists like this will always have the complexity $O(n \log(s))$. Furthermore, other sorting algorithms, such as radix-sort or merge-sort will not give useful results after a partial sort. Bucket-sort could be used. While it has been shown how to implement GPU-based bucket-sort [Sintorn and Assarsson 2007], that implementation requires atomic operations, currently only available on NVIDIA's mid-end hardware Geforce 8500 and 8600.

Implementing the algorithm above on the GPU is also quite simple, given the *Geometry Shader* and *Transform Feedback* extensions to the OpenGL programming API. To partition a list of points into two sublists as above, we need only run the input points through two passes in the geometry shader; One where we discard the points closer than the $middle$ distance and one where we discard the other points. The output stream of both passes is caught by transform feedback and the number of elements in each sublist can be found with a query.

For the purpose of rendering hair, we will want to sort line segments in this manner, instead of simple points. This complicates matters somewhat since a line may be part of both sublists. If the geometry shader finds that the line segment it is working on is split by the plane defined by $middle$, it will output the line in both sublists, but will clip it on that plane (see Figure 4). This means that the sum of the number of elements in the two sublists can be larger than the number of elements in the original list. To be certain that our algorithm will not need a huge amount of memory, or become very slow, we must thus impose some restriction on the maximum length of a line segment. If, for example, we say that a line segment can not be longer than the minimum width of a slice, we can be sure that the fully partitioned list will not be larger than $2n$. In practice, such a restriction is not unreasonable since hair, especially curly or wavy hair, requires fairly high tessellation to be realistic.

If the lines are sorted breadth first, we will only need to allocate two buffers for storing the lines during the sorting. The two first sublists from buffer A are generated into buffer B. Then, the four sublists of the second pass from buffer B are generated into buffer A and so on.

To account for cases where a low memory footprint and performance is more important than correctness, we have also implemented a version where we instead assign the line segment to the slice it most occupies, and do not clip it at all. The result is a more "rough" partitioning but the sorting has the nice property of always using the same amount of memory, and generating the same amount of hair segments. The performance of this version is slightly better, and the results are almost indistinguishable from the correct version (see Figure 6).

3.2 Rendering Opacity Maps

Since we now have a method for sorting the line segments into sublists representing regularly spaced slices of the frustum, the opacity maps can easily be rendered in time proportional to $O(n)$. The sublists are simply rendered in order into a framebuffer with additive blending and between rendering each sublist we copy the current result into a 3D texture slice. Each slice of the 3d texture will then contain the opacity, o, at that point in space, that is the sum of all the previous slices plus the opacity of the hair segments in that slice. The visibility, v, of the fragment is calculated as $v = e^{-k/o}$, where k is some constant. The fragment color, calculated by the shading model, is then multiplied by this visibility value (see Figure 5).

To create a convincing image of a head with hair, we must make sure that the semi-transparent geometry shadows the opaque geometry and vice versa. The shadows that the hair casts on the body and scalp are important visual cues, and if ignored can make the hair look "glued on" to the head. The opacity map can be used in

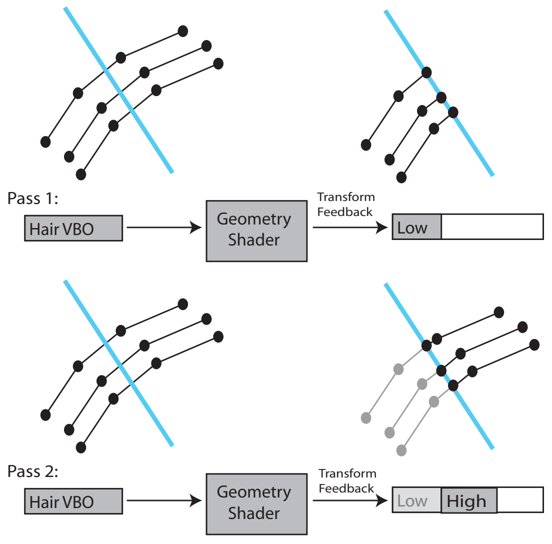


Figure 4: One step of the Quicksort algorithm is done with two passes through the Geometry Shader.

combination with a normal shadow map for opaque objects, so that whether we are rendering hair or opaque object, we first look in the shadow map to see if the point is shadowed by an opaque object (such as the head) and otherwise we use the opacity from the opacity map.

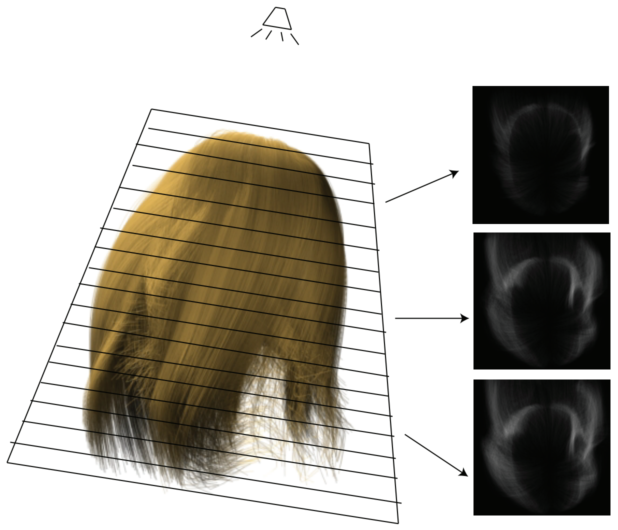
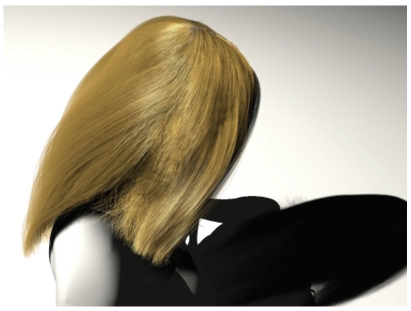
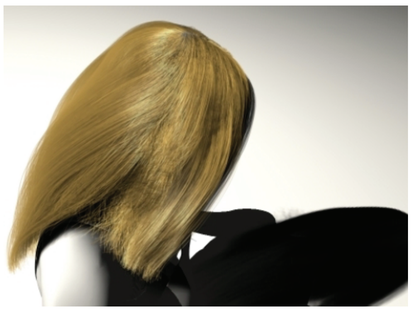


Figure 5: The opacity maps store, for each pixel, how much of the light has been blocked.

When rendering hair, the tangent of the hair is needed at each fragment so that the angles of the incoming light can be calculated for the hair shading model, and this tangent may be stored with each vertex. In our implementation, to minimize the amount of data shuffling required by the sorting algorithm, we simply calculate the required angle for each vertex in a first pass, and encode it in the w-component of each vertex. One might however also want to store other data about the hair segments such as color or thickness and in that case it would make more sense to quicksort key/value pairs, where the key is a pointer to the hair segment and the value is its current distance from the plane. This would be easy by just storing the key in the w-component instead of the angle.


(a) The hair is rendered with the complete model.


(b) The hair is rendered with the "rough" partitioning scheme described in section 3.1

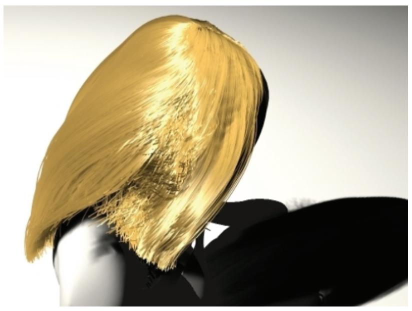

(c) The hair is rendered with the two-dimensional approach described in section 3.4

Figure 6: Three approaches of rendering

3.3 Transparency sorting

Hair strands, especially in blonde hair, are semi transparent which presents another challenge when rendering hair. Also, since hair strands are actually much thinner than the pixel-size, unless the camera is very close to the hair, drawing opaque lines for each strand will make the strands look much too thick. The common solution to both problems is to render the hair segments with alpha blending. With alpha blending, the fragment f with some transparency α which is drawn to screen will be blended with what is already in the framebuffer, b, as:

$$b_{new} = \alpha * f + (1 - \alpha) * b.$$

For this to work, it is important that the transparent geometry is rendered in order, back-to-front from the camera's viewpoint. For complex geometry, this has for a long time presented a problem and the solutions so far include sorting the geometry per frame on the CPU, which strongly limits the amount of geometry that can be rendered in real time, or using techniques such as depth-peeling [Mammen 1989], which can be run entirely on the GPU but is not fast enough to handle many depth slices in real time.

The quicksorting routine presented above, however, allows us to quickly sort the primitives into 256 slices in back to front order for every frame, offering a solution to the transparency-sort problem when rendering lines or points (see Figure 2). Note that we have not yet tested alpha sorting of triangles, although sorting with the "rough" partitioning scheme explained above could trivially be used for any geometric primitives as long as their size is small compared to the depth of a slice.

3.4 Two dimensional Opacity Maps

The Opacity Map algorithm relies on the light-transfer function being sampled into a 3D texture, and since both the resolution of each slice and the number of slices need to be quite large for good quality renderings, this texture will occupy quite a lot of GPU memory. With 32-bit opacity samples, using 256 slices of 512x512 texels, the memory usage will be about 64MB, which is a significant amount of the total memory available on many consumer level graphics cards. In addition, often only 4-float component 3D textures are supported, requiring four times as much storage or using some custom storage and filtering. We will here suggest a different approach to rendering geometry with high resolution opacity maps, without using much more memory than a standard shadow-map.

The idea is simply to draw the geometry to screen in the same order as it is drawn into the opacity map, i.e. front-to-back from the light-source. That is, we sort the geometry, then draw the first slice to the screen and opacity map simultaneously. We then draw the second slice to screen, now using the opacity map that was created by the first slice, and so on through the slices. The opacity map will be built while drawing to the screen and will at any point in time contain the opacity values needed to draw the next slice to screen. This way, we can do correct self shadowing without using more than a single double-buffered 2D opacity map. If we want to have filtering between slices, which we get for free with 3D textures, we will need to store the last result in one more opacity map and do the filtering in the fragment shader. We have not yet implemented this.

When using this approach, however, we will not be able to do transparency sorting. The reason is that the geometry is now rendered as sorted from the lightsource, instead of from the camera, and at no time, the entire contents of the opacity map is known. With a good multisampling scheme, the results can still be quite appealing though (see Figure 6c). Some degree of alpha sorting can be performed by rendering the final image into multiple render targets, passing the fragment to the target representing its slice, and then blending these rendertargets back to front.

When using a two-dimensional opacity map together with a standard shadow map for opaque objects, the shadow-map is created first. Then, the opacity map is updated only if the fragment is not in opaque shadow (checked by using the standard shadow map). When rendering to screen, the values in the opacity map are unimportant if the fragment is in opaque-shadow, and when rendering a fragment of an opaque object that is unshadowed by any opaque objects, it is important that the fragment is only shadowed by the transparent geometry with lower depth values.

4 Results

In Figure 7, we present some performance and picture quality results that we have obtained when testing our implementation. All tests have been performed on a 2.66 Ghz Intel Dual Core system with a GeForce 8800 GTX graphics card. For the tests, the same model has been used in three versions with different amounts of hair strands. The number of segments per strand were 30 in all cases, but the number of strands were, in *Small*: 5000, in *Medium*: 10000 and in *Large*: 20000, resulting in 150k, 300k and 600k line segments respectively (see Figure 8). For each model, timings have been made when rendering hair with the full model, with the "rough" partition-

Model	T_{SO}	T_{SA}	T_{RO}	T_{RS}	Frametime
Small	19.5	19.5	22	22	87
Medium	30	34	22	34	120
Large	56	56	22	60	195

(a) Full Opacity Maps model

Model	T_{SO}	T_{SA}	T_{RO}	T_{RS}	Frametime
Small	17	17	22	20	80
Medium	24	24	22	27	100
Large	40	40	22	44	147

(b) Rough Opacity Maps model

Model	T_{SO}	$T_{RO} + T_{RS}$	Frametime
Small	19	72	104
Medium	30	71	114
Large	57	69	140

(c) 2D Opacity Map model

Figure 7: The time taken, in milliseconds, for various parts of the algorithm, for three different approaches to rendering and three models of different complexity. T_{SO} is the time taken to sort geometry for the opacity map rendering, T_{SA} is the time taken to sort for alpha blending, T_{RO} is the time taken to render the hair into the opacity maps and T_{RS} is the time taken to render the model to screen, with alpha blending.

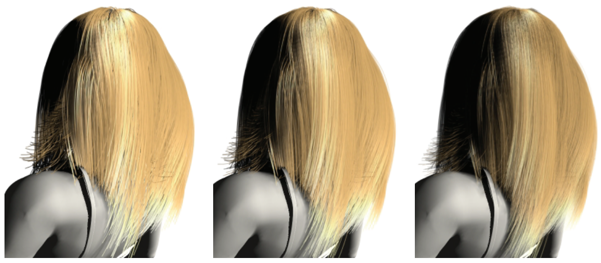


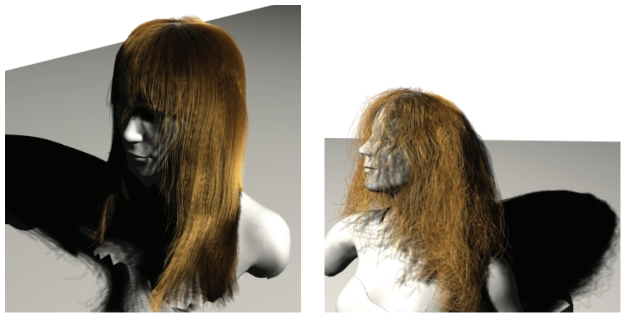
Figure 8: The small, the medium and the large model in left-to-right order, with 150k, 300k and 600k line segments respectively.

ing scheme presented in section 3.1 and with the two-dimensional opacity map scheme explained in section 3.4.

5 Conclusion and Future Work

This paper has presented a method using the GPU for rough sorting of primitives for the rendering of opacity maps, used for self shadowing, and also for the transparency blending. We demonstrate the algorithm on hair rendering for real-time applications. The method allows us to use more than an order of magnitude more slices for the opacity maps, which is significant for increasing the quality. It also relieves the CPU from having to do any transparency sorting. In addition, we also present an opacity maps based method for self shadowing requiring only the same amount of memory as standard shadow maps, with the trade-off of not being able to use transparency sorting but using far less memory. We demonstrate how GPU-based quicksort can be implemented and used for the sorting. The result is self shadowed and transparency sorted hair with a higher quality than previously has been possible outside the offline rendering community.

In the future, we would like to test the algorithm not only on hair, but also for smoke, fur, foliage and clouds.



(a) (b)

Figure 9: Two examples of other hair styles, rendered in 12 fps for (a) and 10 fps for (b), using the rough partitioning described in section 3.1.

References

CROW, F. C. 1977. Shadow algorithms for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH 77)*, 242–248.

EVERITT, C. 2001. Interactive order-independent transparency. *Technical report, NVIDIA Corporation, May 2001. Available at http://www.nvidia.com/.*.

HADWIGER, M., KRATZ, A., SIGG, C., AND BÜHLER, K. 2006. Gpu-accelerated deep shadow maps for direct volume rendering. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, ACM Press, New York, NY, USA, 49–52.

HOARE, C. A. R. 1961. Quicksort: Algorithm 64. *Communications of the ACM*, 321–322.

KAJIYA, J. T., AND KAY, T. L. 1989. Rendering fur with three-dimensional textures. *Proceedings of SIGGRAPH*, 271–280.

KIM, T. Y., AND NEUMANN, U. 2001. Opacity shadow maps. *Proc. of Eurographics Syyymposium on Rendering*, 177–182.

LIU, B., WEI, L., AND XU, Y. 2006. Multi-layer depth peeling via fragment sort. *Microsoft Technical Report, MSR-TR-2006-81*.

LOKOVIC, T., AND VEACH, E. 2000. Deep shadow maps. *Proceedings of ACM SIGGRAPH*, 385–392.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics*, 43–45.

MARSCHNER, S. R., JENSEN, H. W., CAMMARANO, M., WORLEY, S., AND HANRAHAN, P. 2003. Light scattering from human hair fibres. *Proceedings of ACM SIGGRAPH 2003*, 780–791.

NGUYEN, H., AND DONELLY, W. 2005. Hair animation and rendering in the nalu demo. *GPU Gems 2*, 361–380.

SINTORN, E., AND ASSARSSON, U. 2007. Fast parallel gpu-sorting using a hybrid algorithm. *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)* (October).

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, 270–274.

YUKSEL, C., AND KEYSER, J. 2007. Deep opacity maps. *Technical Report, Department of Computer Science, Texas A&M University*.