A MEMORY CONSERVING RENDERING METHOD FOR HAIR/FUR SYSTEMS IN COMPUTER GRAPHICS

By
ANTON ZUYKOV
Engineering degree, Control in
Automation and Information Systems
Penza State University
Penza, Russia
2006

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of MASTER OF SCIENCE
July, 2015

A MEMORY CONSERVING RENDERING METHOD

FOR HAIR/FUR SYSTEMS

IN COMPUTER GRAPHICS

Thesis Approved:

David Cline

Thesis Adviser

Douglas Heisterkamp

Committee Chair

Blayne Mayfield

Committee Member

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Prof. David Cline for his continuous support of my M.S. study and research, for his great patience, motivation, enthusiasm, and knowledge. I could not have imagined having a better advisor and mentor for my M.S. study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Douglas Heisterkamp and Prof. Blayne Mayfield for their insightful comments, and questions.

I also would like to thank my family: my parents Mikhail Zuykov and Galina Zuykova, my grandfather Yurii Sukhodolov and my grandmother Valentina Sukhodolova, my sister Elena Malashina and her family for their constant support and love.

I dedicate this thesis to the loving memory of my grandmother Valentina Sukhodolova, who always supported me in my decision to study abroad and who, sadly, passed away before I could finish my studies.

This last word of acknowledgment is for my dear wife Tanya Naidenova who has been with me through the course of my studies giving me enormous support, love and care.

Name: ANTON ZUYKOV

Date of Degree: JULY 2015

Title of Study: A MEMORY CONSERVING RENDERING METHOD FOR HAIR/FUR SYSTEMS IN COMPUTER GRAPHICS

Major Field: COMPUTER SCIENCE

Abstract:

In a very CPU/memory intensive field of photo-realistic computer graphics, various techniques are employed in the attempt to conserve resources. One group of such optimization methods is dedicated to optimizing a representation of hair systems, grass systems or any group of objects that can be looked at as a generalized hair system.

A classical method of computing hair systems is to represent each hair as a spline in memory and then compute intersections with each of them. This method gives good results, but usually consumes large amounts of memory. Another problem is - visually doubling the density of hair quadruples memory consumption.

Even when gigabytes of memory are available, a realistic hair scene, may overwhelm memory size, which may lead to an application crash or at least, to an I/O bottleneck and to increasing time of rendering.

Another method is to compute a hair system procedurally inside of a specified volume. This produces a small memory foot-print, but makes animation difficult because individual hairs within the volume are not controllable.

In this work we propose a hybrid approach, where a single hair particle represents a cylindrical volume, in which multiple hair fibers will be computed on-the-fly. This approach will produce a constant memory footprint for that cylindrical volume, regardless of how many individual hairs are computed and it will allow individual hairs to retain the behavior of that volume. As a result, a proposed approach provides a significant reduction in memory footprint while increasing the number of hairs being computed.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1. *Computer Graphics and photo-realistic images*

Computer graphics is concerned with generation of imagery by using computers. Initially, the imagery was produced purely for scientific data visualization and was used to display plots. As time went on, however, photo-realistic CGI became a research goal.

Prior to the 1980s, there were different research teams probing into the field and attempting to develop various techniques and algorithms so that they could compute images that would resemble real ones, without using photographs as source material.

A key event happened when ILM (Industrial Light and Magic), which was a division of Lucas Film, Ltd, started working on animation and rendering techniques, after the success of the original Star Wars films. They were the first to create a fully computer generated photo-realistic animated character, a knight composed of elements from a stained glass window ("Young Sherlock Holmes", 1985).

1.2. *Typical problems with computer-generated hair*

In order to create photo-realistic CGI, various algorithms need to be used to mimic the behavior and/or look of an object or a phenomenon that is being replicated. As a rule of thumb, the better an algorithm is at replicating the complexities of the real-world object, the higher space/time complexity that algorithm requires. This leads to resource constraints that stop artistic expressiveness

CGI research has focused on developing new algorithms that would help to solve those problems by utilizing resources in a smarter way or by offering a space/time trade-off.

Hair and hair-like object representations have become one of the areas of interest due to the need to represent hair, fur, grass and other hair-like objects for CGI. Unfortunately, due to their nature, those objects usually come in large numbers in the real world and that requires a lot of resources for dealing with them in the virtual reality. As a result, faster and less resource-hungry algorithms to represent such objects are still an active research area.

A classical hair-system representation (and the most widely used today) would be a set of curve segments connected in a chain. This is a great way to create a controllable hair, and it doesn't take that much memory to store a single hair in computer memory.

Unfortunately, realistic scenes may contain millions of individual hairs, and the task of rendering those becomes a problem. If we assume that there is a square volume with the number of hairs at every side being N, then we would need N^2 hairs to populate the volume. And if we need to double the density of the hair, we would need four times more memory to do that. This leads to problems with the lack RAM, when dealing with large hair simulation/rendering scenarios.

1.3. *Open shading language (OSL)*

Open shading language is a language for describing materials, lights, displacement and procedurally generated textures (or, more generally, patterns) for realistic 3D rendering systems. OSL was developed by Sony Pictures Imageworks as a simple, yet powerful tool that allows for creating complex light-material interactions without changing the code of the underlying rendering package. This allows the renderer to simulate various physically-based interactions and, as a result, render more realistically looking images.

OSL works by obtaining different pieces of data directly from the renderer. These include

information about an intersection point between a ray that is being cast and a scene geometry, the type of that ray (camera, shadow, etc.), its length, type of geometry that is being intersected, texture coordinates, and other additional information that is specific for the geometry, such as the length of a hair, its radius, height at which the ray intersects the hair. After obtaining that information an OSL shader can compute the surface properties for the intersection point and pass that data back to the renderer to complete the 3D rendering. [1]

OSL has a C-style syntax and hence, can be translated into other shading languages without significant problems. OSL compiles into assembly-like byte code which brings it close enough to the hardware to give a good computational performance. [1]

One of the very powerful features of OSL is its so-called radiance closure – an explicit symbolic description of the way a surface or volume scatters or emits light, in units of radiance. This allows OSL to have closure primitives for BSDF, BSSRDF, emission, and volume scattering.

OSL is currently supported by several commercial rendering systems such the Arnold Rendering system (a stochastic ray tracing system from Sony Imageworks) as well as by the open-source 3D package Blender through its Cycles renderer.

CHAPTER II

LITERATURE REVIEW

2.1. *Related work*

One of the first major attempts at rendering hair by using 3d-texture (texel maps) in a volume (Figure 2) without the use of splines, line segments or other types of geometry, was done by Kajiya and Kay in 1989 [2]. The key idea was to represent the 3D material (fur) by a cubic reference volume, to be mapped onto a surface (like a thick skin). The copies of that volume (texels) are fitted on the bilinear patches of the surface, and are deformed in order to fit together(see Figure 1). Later Meyer and Neyret proposed a modified method, built of Kajiay's work, but with smaller volume patches which could be used multiple times. This resulted in an interactive algorithm that is fast enough for rendering images at 1-2 fps update rate.[3][4]

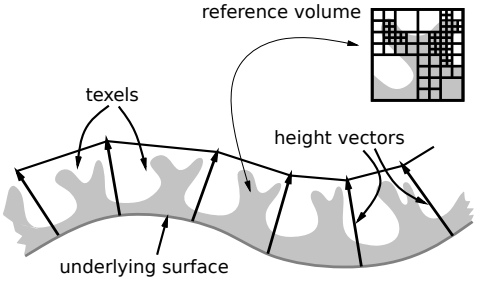


Figure 1. Cross-section of a volumetric texture representation

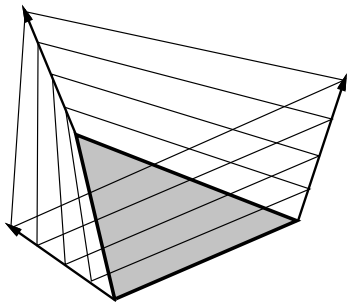


Figure 2. Texel is drawn by extruding textured triangles

In this method, 3d textures were mapped onto small patches of the deformed 3d-primitive volume that covers the whole surface of a complex 3D geometry. Mapping is done in accordance with the normal of the geometry surface under that primitive volume. In the essence, this is a variant of implicit representation of a hair system that doesn't keep individual fibers in memory.

Another approach was taken by Koh and Huang (2001) [5], Liang and Huang (2003)[6], Noble and Tang (2004)[7], Yuksel, Schaefer and Keyser (2009) [8] who wrote papers on various flavors of hair meshes and their implementations. The idea behind these was that an artist working with a high-level representation of the volume occupied by hair, controls the volume and hair shape without changing individual hair strand positions and the proposed methods take care of generating actual hair inside of that volume during rendering. A standard logic behind this set of algorithms can be viewed as: NURBS → hair strands → deformed/styled hair strands [8].

Also, there exists a more conventional, explicit method of rendering hair systems that depends upon representing a strand of hair or fur as a set of chained line/spline segments which may or may not include explicit translation from that form into polygons, before computing lighting, shadows and other things such as occlusion, self shadowing, that are important for realistic computer generated imagery[9]. There are various works done on that topic, including dynamic simulations and use of various techniques that are covered very extensively in a comprehensive review by Yuksel et al (2007).

Finally there is another paper combining an idea of instancing and displacement mapping into a single algorithm, to provide relief maps [10] .

2.2. *Existing techniques*

Current and practical techniques include:

- use of spline(s) or line(s) connected into a chain for representing a single fiber. This technique is the most common one used in photorealistic rendering. It requires storing coordinates of each of line/spline segment in memory (for splines, it will be additional control points).

- use of spline(s) or line(s) connected into a chain for representing a single fiber with addition of children that are not controllable but yet generated as an interpolation of parent fibers. It is essentially the same as the first one, because children get generated before rendering begins and they require memory space to be stored in. This is how Blender3D hair system works.

- a completely procedural generation of fibers without any intermediate splines/line storing involved. This approach lacks control over individual strands or fibers. It consumes less memory but is not controllable at a fine level.

- use of standard mesh with a texture of hair + transparency channel. A technique that is currently not used very much. It was used when computers didn't have any other methods, described above and were lacking enough memory to do that. It doesn't cost that much memory-wise, compared to storing individual strands. There is minimal control over general motion of a group of hair that the set of polygons represents, but by todays standards is considered to be obsolete.

- NURBS based patches with a texture of hair. A method is based on projecting hair textures onto the animated NURBS surfaces, so that it becomes easier to animate. It resembles mesh with hair method, except NURBS surfaces provide smother surfaces.

CHAPTER III

A PROPOSED METHOD

3.1. *Proposed method*

Our proposed method offers a hybrid approach which should help to conserve memory, while still providing a sufficient control over hair system so that it could be used in today's photo-realistic rendering software packages and in dynamic simulations. Pseudo-code of the proposed algorithm can be found in Appendix A, page 34.

Each individual hair cylinder of the conventional hair system (Figure 3) in our method represents a cylindrical volume that will be used to generate fibers, that will be placed inside of the that volume(Figure 4).

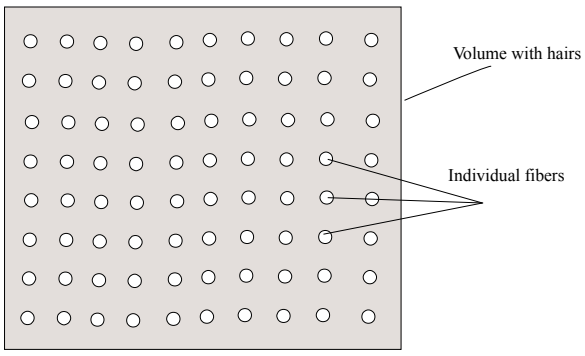


Figure 3. Conventional hair system

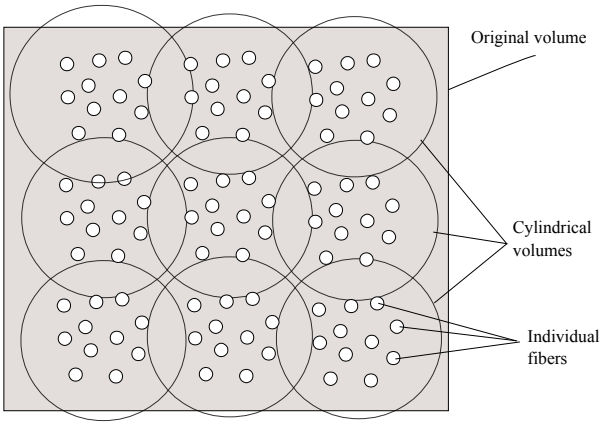


Figure 4. Proposed hair system

The hair cylinder position and its orientation are stored in memory, but internally generated fibers

are not, and instead they are computed on the fly in pseudo-random positions, and only when that cylindrical volume is intersected by a ray (Figure 5). This allows us to have the same memory footprint for a single hair cylinder, regardless of how many fibers are generated inside of it. Thus memory consumption will decrease, compared to the standard method (Figure 4 vs Figure 6).

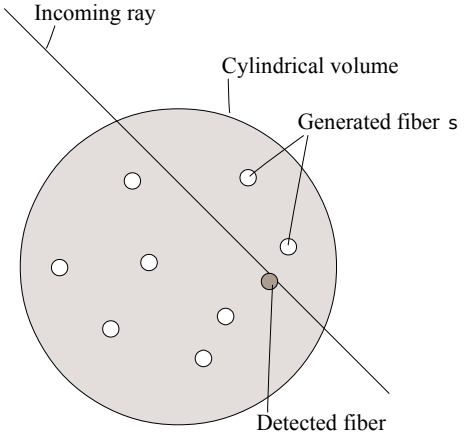


Figure 5. Fibers placed at random locations within hair cylinder

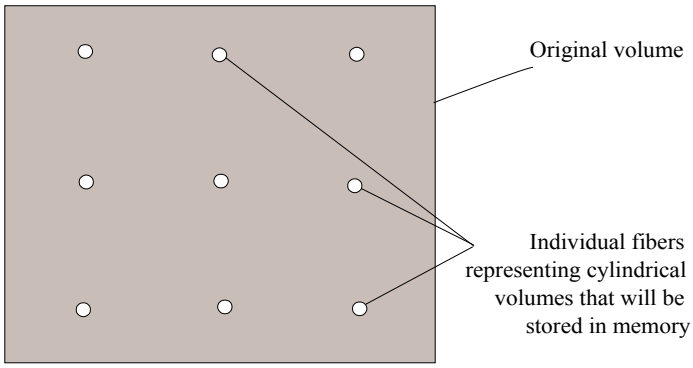


Figure 6. Fibers that will be stored in memory according to our method

This approach assumes that the IO bottleneck from accessing memory (for hair coordinates) is much larger, than what it will be for generating those fiber positions on the fly.

Such a hybrid approach still allows the use of conventional hair dynamics calculation methods/tools that are currently available in various 3d software packages (without the need of rewriting tools or changing a working pipeline), while significantly decreasing a memory footprint. In fact, given that the number of hair cylinders is kept the same, and only the number of fibers that are generated, is changed, the memory footprint for a given 3D scene should stay constant, regardless of the number of fibers being produced by a single cylinder. Despite the drawback of the proposed approach that doesn't allow for dynamic interaction between fibers inside of a single cylindrical volume, this is a very similar way of how today's hair systems compute hair dynamics. A set of fibers representing primary fibers and their movements are computed, while the secondary fibers (located in between primary fibers) assume the role of “slave” fibers, simply following the behavior of primary fibers. This reduces computational cost by ignoring collisions between secondary fibers, but unlike

our method, secondary fibers are still stored in memory, albeit their dynamics are not computed, but rather inherited from the primary fibers.

Unlike a conventional hair system, where a spline or a line represents an individual fiber, in our system, each individual fiber, provided by a 3D-package should be viewed as a cylinder with a finite height and radius, so that its radius is comparable to its height. During rendering process, if a camera ray intersects that cylinder (original single fiber), then a shader is invoked.

The shader obtains point P of intersection between the incoming ray and the cylinder, and the shader also obtains incoming ray direction I, and the normal of the surface of the cylinder N at the point of intersection (Figure 7). Based on that information, we compute center of the cylinder C (Expression 3.1).

$$C = P - normalize(\vec{N}) \cdot r_{cyl} \qquad (3.1)$$

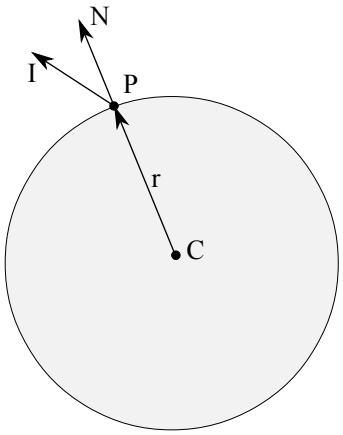


Figure 7. Finding center of the cylinder

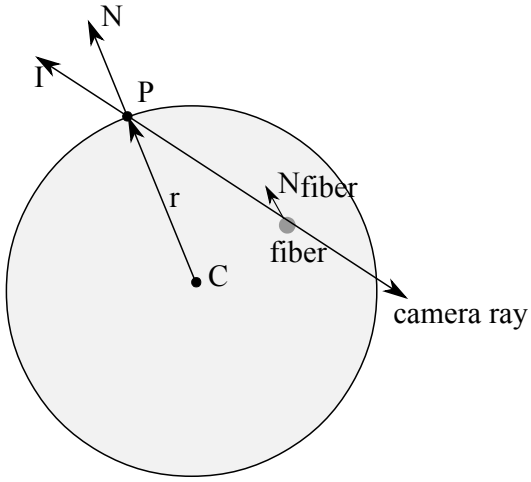


Figure 8. Ray-fiber intersection

Using that, we can generate a fiber, located inside of that cylinder and check if the incoming camera ray intersects with that fiber. We do this once for each fiber in the cylindrical volume. If the ray hits a fiber, the algorithm computes that fiber's normal at the point where the ray intersects it (Figure 8). Then we assign a material for the surface point and return that information to the renderer. After obtaining all that data from the shader, the rendering system finishes computing the lighting for

the point. Thus, our method is an insertion into the existing pipeline, and not its complete overhaul.

3.2. *Optimization for the method*

The approach described above is better than the conventional one in terms of memory footprint, but there is still room for improvement. The problem is that if we compute all those fibers inside of the cylinder, the ray doesn't not intersect with most of them. As a result such computation becomes very inefficient (Figure 9).

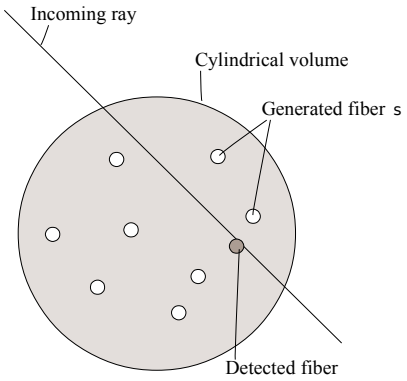


Figure 9. Multiple fibers generated by brute force approach

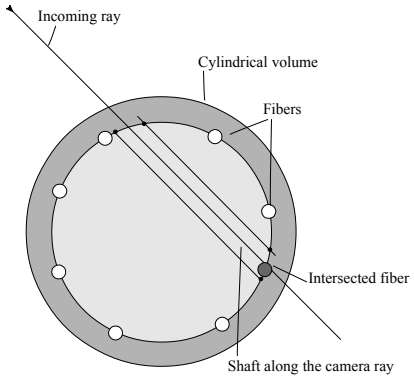


Figure 10. Modified fiber generation routine for a single circular path

Instead of computing the whole set of fibers inside of the cylindrical volume, we can place fibers at a certain radius from the center of the main cylinder. This would create an internal coaxial cylinder and, by calculating where camera ray intersects with an internal cylinder, we can figure out the exact segment (shaft) of that cylinder and two arcs (at most) along which we need the fibers to be generated and checked for possible intersections with the ray (Figure 10).

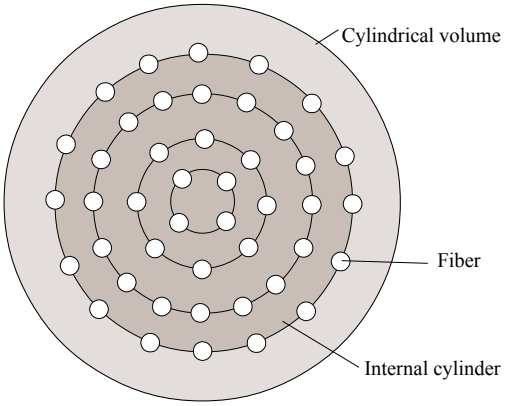


Figure 11. Multiple circular paths with fibers

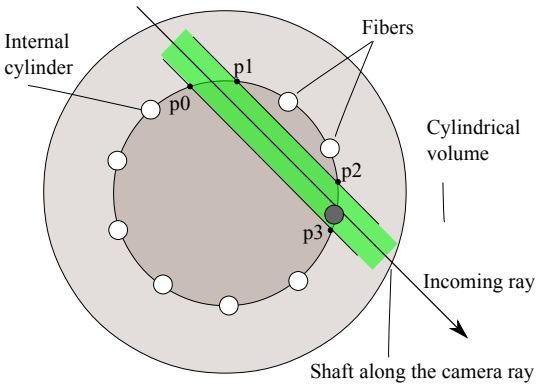


Figure 12. Ray shaft

It is obvious that a single internal cylinder is not enough to create enough fibers to fill the whole cylindrical volume. This can be solved by adding several coaxial internal cylinders, each with a different radius (Figure 11).

3.3. *Fiber placement*

When the ray intersects with a cylindrical volume, we create a shaft parallel to the ray (Figure 12). The shaft is an area that is created by offsetting the ray by some value toward the center C of the cylindrical volume as well as away from it. The resulting area is bound by the offset rays is the shaft. It is used to calculate intersection of the ray with internal cylinders and computing points p_0, p_1, p_2, p_3 . When the internal cylinder intersects with the shaft, there are several possible cases which can create a different number of intersection points. Cases are shown in Figure 13 below: Note that Case3 is not supported by our algorithm - it treats it as the case with no intersections at all.

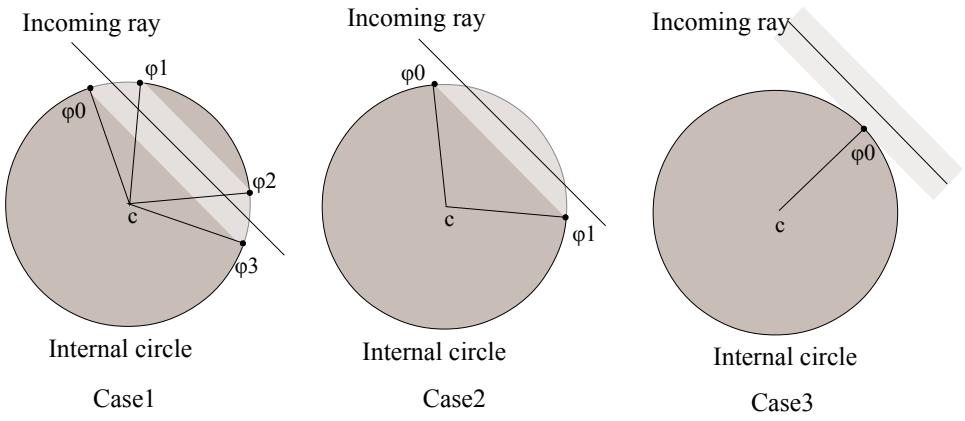


Figure 13. Possible shaft-internal cylinder intersection cases

It is worth mentioning that, $\varphi_0, \varphi_1, \varphi_2, \varphi_3$ in Figure 13 are not points in space, and instead each of those values is a single *float* value (in degrees) from [0..360). They are obtained through computing a vector $\vec{pv}$ (Expression 3.2) from points p_0, p_1, p_2, p_3 and then computing the angle (Expression 3.3) which that vector has in relation to the coordinate system. This allows us to compute angular bounds for each of the sectors and subsequently helps to consistently generate fibers on its internal circular path.

$$\vec{pv} = normalize(p_i - c) \tag{3.2}$$

$$\varphi = atan2(pv.x, pv.y) \tag{3.3}$$

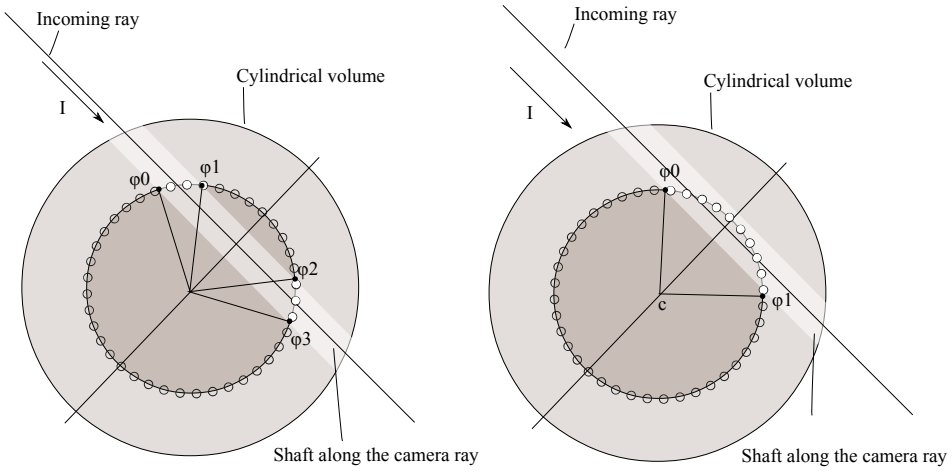


Figure 14. Fibers are generated on the fly (white)

Figure 14 shows an example of how fibers are generated on the fly. Light color represents fibers that are being generated, while dark fibers represent only potential positions, that do not fall within the ray shaft (gray area) and are not generated.

3.4. *Order of traversal*

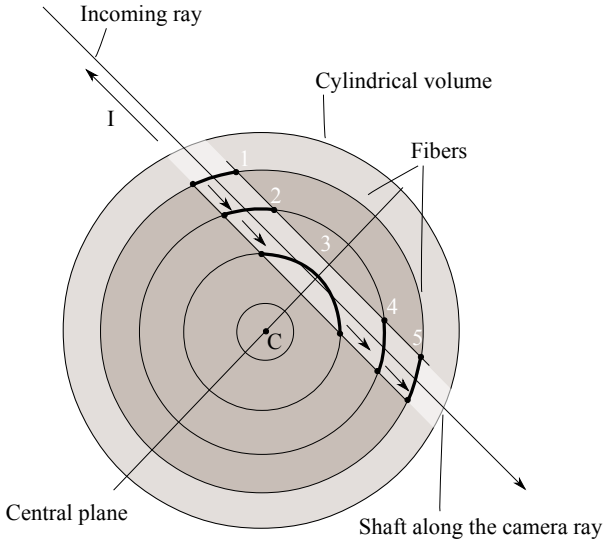


Figure 15. Correct order of traversal

For the images of fibers to look correct, we must choose the intersection p_i closer to the ray origin. We can do this by traversing the internal cylinder in the proper order and returning the first intersection point found with a fiber.

The order of checking should follow this rule: it starts at the intersection between a ray and an internal cylinder, that is closest to the camera then it proceeds through all arcs of that circular path, toward the center of the cylinder, up until the central plane that is perpendicular to the ray and passes through C, then it goes from the central plane toward the other side of the main cylinder shown on the Figure 15.

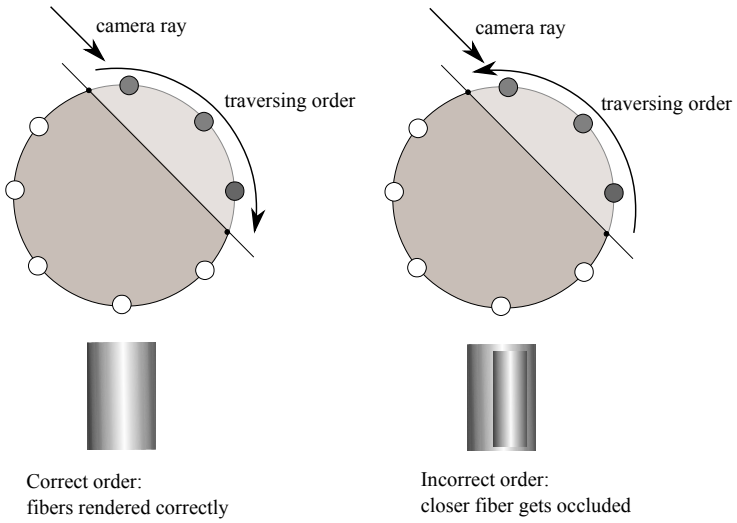


Figure 16. Incorrect vs correct order for Case2 intersection

Another problem is that in certain cases we have to have a special order for traversing fibers even inside of a certain arc/sector . In Case 2, (Figure 13) when a shaft intersects partially with an internal cylinder, only a single arc/sector bounded by two angle values is created. In this case, due to orientation of the sector to the camera, traversal order does matter; otherwise, it might create problems if distant fibers occlude fibers that are closer to the camera (Figure 16, left). In this case we start at the closest (to point P) intersection point, and go along the circular path, following the camera ray direction, until another intersection point is met. This way we can ensure that the fibers that are farther away, will not occlude those that are closer to P and thus, the correct order will be maintained.

3.5. *Fiber position randomization*

By default, our algorithm places fibers on the circular path with a center that coincides with main cylinder center C (Figure 17). Even if several internal circular paths are used, that is not enough to ensure the natural (random, that is) distribution of fibers inside of the cylindrical volume. Instead, this would produce a very symmetrical and artificial look. (Figure 18).

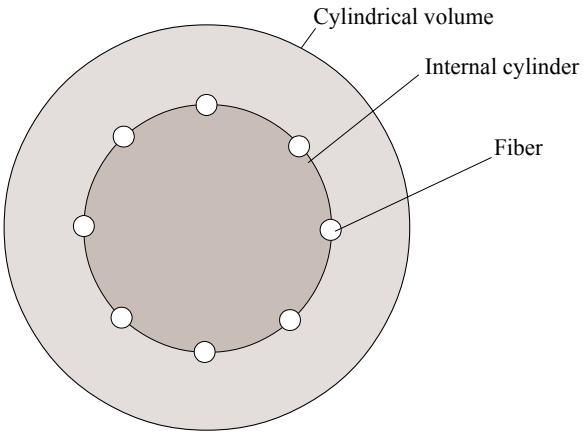


Figure 17. Single circular path with fibers

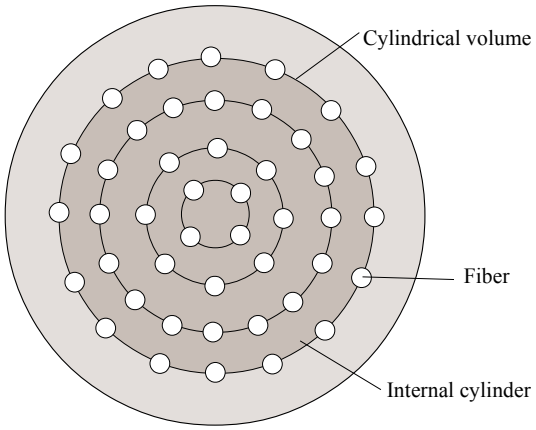


Figure 18. Multiple circular paths with fibers

To fix that, we propose a randomization (Figure 19) that will offset a position of every fiber, away from or toward the center C by some value, as well as, an angular random offset - clock-wise or counter-clock wise, for each fiber along the path (Figure 20).

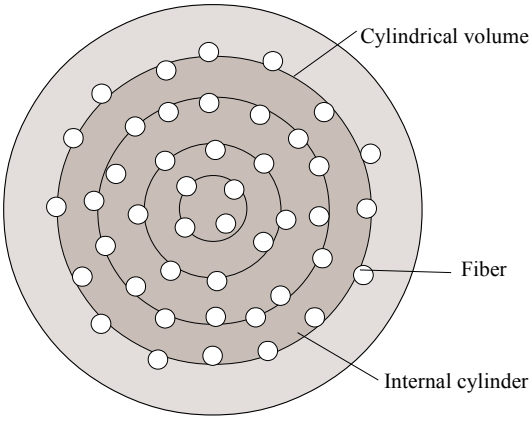


Figure 19. Multiple fibers with randomization

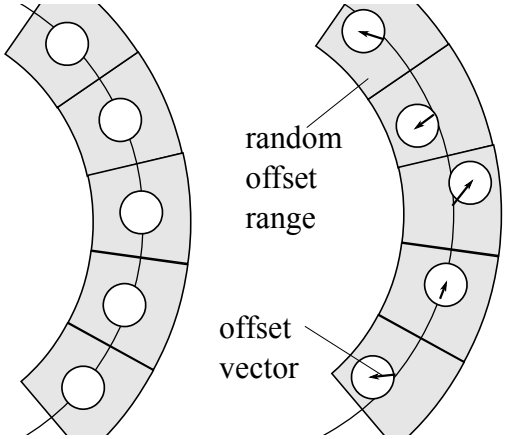


Figure 20. Random offset vectors for fibers

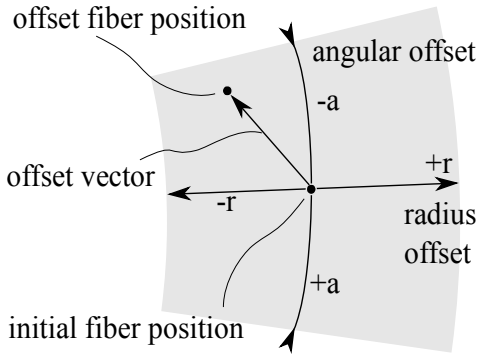


Figure 21. Randomization area for a single fiber

Both of these offsets (angular- based and radius-based) should be bounded by some reasonable numbers (Figure 20). If that random number range is chosen to be too large, then the correct order of traversal will be violated, due to fibers from one circular path overlapping/occluding a fiber from another circular path. Due to a strict order in which we traverse and check different internal circular paths, some fiber form a neighboring circular paths might end up in a wrong spot and it will be computed incorrectly, due to the assumption, mentioned above.

Hence, another rule for rendering fibers in a correct order should be this: for each area of fiber's probable position (or possible offset ranges) (Figure 19) there should be only one fiber among all generated fibers on all circular paths, that can be placed anywhere inside of that area.

To do that without overlapping fibers, proper random value ranges should be chosen. Due to our set up, having a possibility of several concentric circles (internal cylinders), for each internal cylinder, that range of offset needs to be different. Mainly it should depend on how many fibers are placed on that circular path, individual fiber radius, and the diameter of the circular path. This should ensure that in no circumstances neighboring circular paths have overlapping fibers (Figure 18). Also, the shaft width needs to be controlled too, so that it includes enough space to accommodate random offsets.

3.6. *Varying inter-fiber distance*

An additional way to make fiber distribution less even would be to use incrementing angular distance at which fibers can be placed along their corresponding paths. The larger cylinder radius is, the smaller inter-fiber distance need to be. This should eliminate regions with higher fiber density that are formed at the center of a hair cylinder otherwise.

3.7. *Fiber length randomization*

Another randomization that can be implemented is randomization of the length of individual virtual fibers. This can be done by introducing a random component into the code where we compute true height of the fiber. Since the length needs to be stable (should be independent from position of the camera, hair system orientation, etc.) we can use angular position of a fiber for determining length of each individual fiber with the help of Linear congruential generator.

3.8. *Linear congruential generator*

For the verification purposes, LCG (Expression 3.4) was used, where φ is a unique number for each fiber and it is fiber's angular position.

$$rand = mod(1207.523 \cdot \varphi, 1) \tag{3.4}$$

3.9. Fiber local coordinate system

In order for the technique to work on deformed fibers and fibers that have various tilt, a local coordinate system needs to be constructed, where local Z axis is parallel to a hair cylinder axis of rotation and hence, it can change orientation at a particular location depending on how deformed a cylinder is (Expression 3.5). Local X-axis is created by calculating a cross-product of a global X-axis with local Z axis (Expression 3.6). Local Y-axis is obtained by computing a cross-product of local Z axis and local X-axis (Expression 3.7). The origin of the coordinate system is at each hair cylinder

origin – which is bottom of the cylinder, (0,0) coordinates.

$$vector\ Z_{local} = dPdu \tag{3.5}$$

$$vector\ X_{local} = cross\left(Z_{local}, X_{global}\right) \tag{3.6}$$

$$vector\ Y_{local} = cross\left(Z_{local}, X_{local}\right) \tag{3.7}$$

3.10. *Fibers with more complexity*

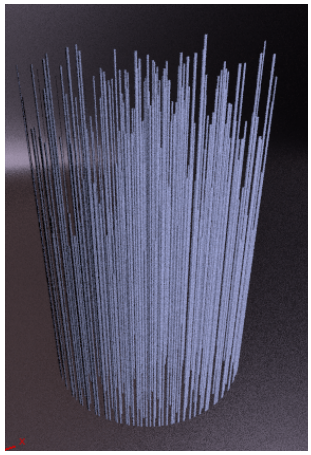


Figure 22. Fibers in a straight container

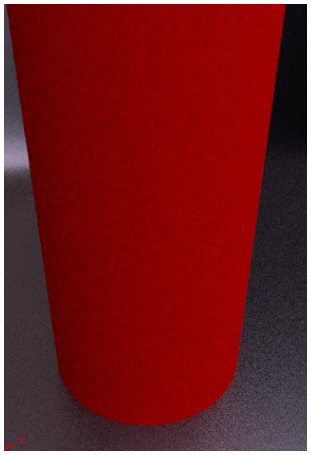


Figure 23. Shape of a straight container

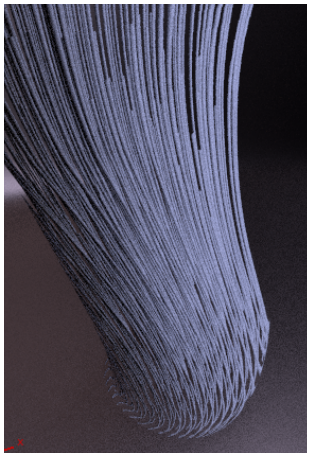


Figure 24. Fibers in a deformed container

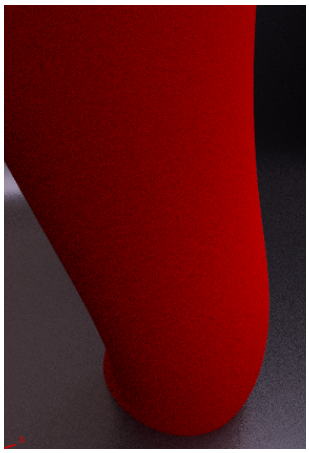


Figure 25. Shape of a deformed container

The initial version of the algorithm assumes that the fibers are always straight in the local coordinate space. That is true if a hair container is not bent, however fibers might end up deformed. This depends on the hair container deformation that is provided by Blender 3D. Figure 22 – Figure 25 give examples of how undistorted vs. distorted hair cylinder might look.

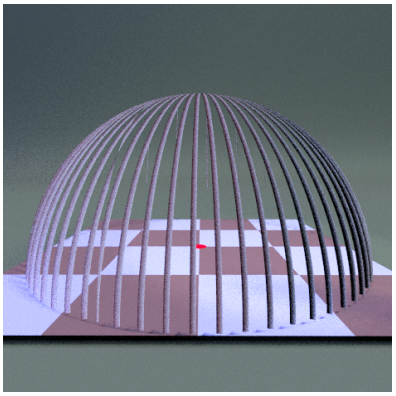


Figure 26. Degenerate ellipsoid is used to compute placement of fibers

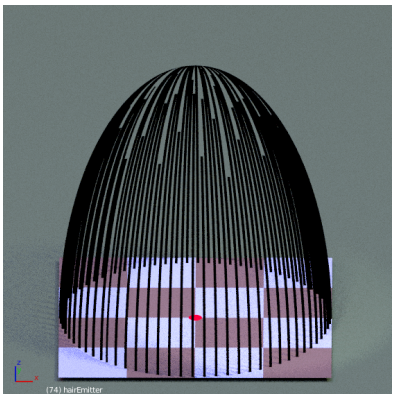


Figure 27. Ellipsoid is used to compute placement of fibers

Alternatively additional methods of finer fiber positioning might be used to make fibers look more complex. Such methods might include curving fibers as if they were located on ellipsoids as opposed to cylindrical surfaces. Since an ellipsoid is a more generalized form of a sphere, we can get deformation shapes (including a cylindrical shape) by using it. The result of such deformation can be seen in Figure 26, Figure 27.

Making fibers go around their respective local centers will produce curly fibers as it is shown in Figure 28, Figure 29.

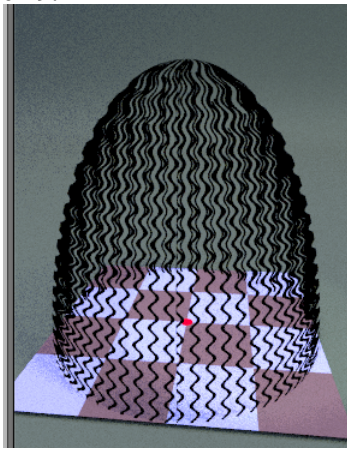


Figure 29. Example of a cylinder with curly fibers. Pattern 1

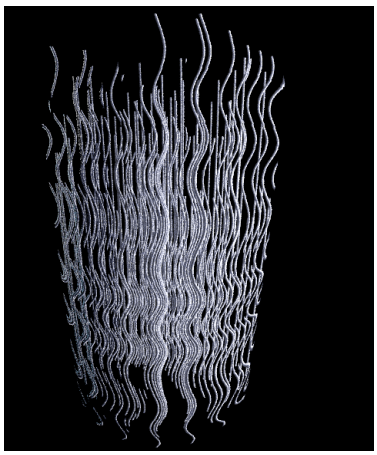


Figure 28. Example of a cylinder with curly fibers. Pattern 2

Rotating the origin of fibers in relation to the height will produce a group of fibers that not only go up but also around the main hair cylinder central axis, as in Figure 30–Figure 32

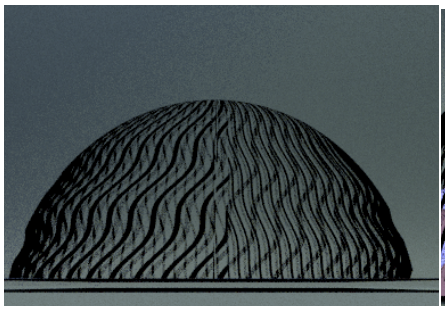
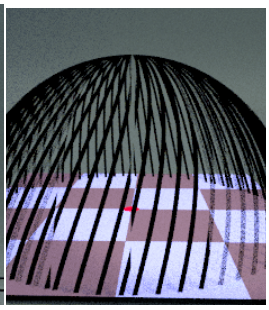


Figure 30. Example of a cylinder with twisted fibers. Pattern 1



ıre 31. Example of a cyli th twisted fibers. Pattern

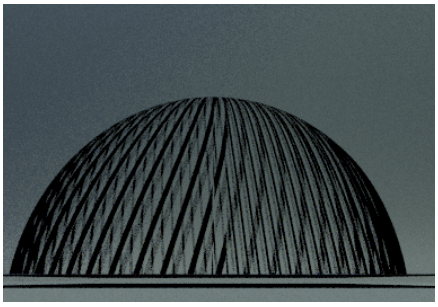


Figure 32. Example of a cylinder with twisted fibers. Pattern 3

These techniques can be combined to produce complicated fiber patterns, like in Figure 34 - Figure 33.

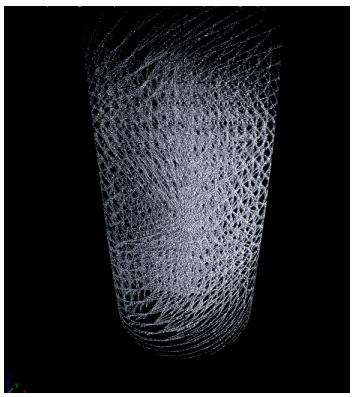


Figure 34. Twisted and curled fibers. Pattern 1

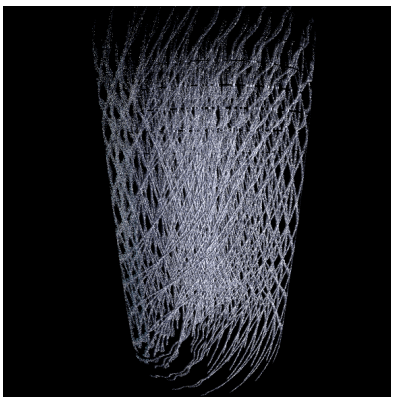


Figure 33. Twisted and curled fibers. Pattern 2

3.11. *Data structure for maintaining right order of traversal.*

Since our method relies heavily on the correct order of traversal, it is important to come up with a good and efficient way of doing that. We propose an additional data structure that gets created every time a shader is invoked (for each intersection of a camera ray with a hair cylinder, that is). The data structure is an array of tuples, where each tuple has several fields. When the shader runs, it first checks if the fibers that are placed on the front arc, intersect with a camera ray. If there is no intersection, before going one level deeper, the shader needs to store parameters for the back arc (as

the front arc that we just mentioned) into the data structure so that it can be computed later without recomputing the whole set of parameters again. The data structure acts as a stack, and so, if no fiber was hit in one of the front arcs, we start popping elements of the data structure top. Because we started with the most distance back arc (in relation to the camera) that we pushed into the stack first, and finished pushing the most inner back arc (closest to the center of the hair cylinder), when we return to the data structure and pop the top element out of it for checking if it has intersecting with a ray fibers, the correct order of traversal is preserved – from the most inner cylinder, away from the center of the hair cylinder toward the back (in relation to the camera) side of the hair cylinder. In our implementation, data structure is a generic C-array with 40 entries and because it is not a vector or another dynamically allocated array, it should have small enough overhead that will not spoil the performance.

CHAPTER IV

FINDINGS

4.1. *Preliminary findings*

A preliminary test was done by using a simplified version of the algorithm, so that instead of a multiple coaxial internal cylinder structure, we placed hairs in the cells of a grid and used a grid traversing algorithm. Fibers were placed only in those grid cells, which were hit by the camera ray. While the algorithm was not optimal in terms of CPU workload, it had a very similar memory footprint to our current algorithm in terms of optimality of CPU workload, however it shouldn't affect the memory foot-print whatsoever.

# of fibers	Conv. alg, memory consumed, MB	Test alg, memory consumed, MB	Conv. alg., CPU time, seconds	Test alg, CPU time, seconds
100	314	171	9.25	32.4
400	314	171	14.7	32.4
2500	338	179	21.6	32.3
10000	460	180	25.5	34.8
40000	920	182	30.26	43.6
160000	2900	188	39.5	65
360000	6300	190	54	85
402000	crash	190	crash	88
490000	crash	192	crash	90
1000000	crash	192	crash	133

Table 1. Preliminary results of a test algorithm run vs conventional

Table 4.1 shows the memory consumption of our early algorithm compared to Blender's standard hair algorithm using different numbers of fibers.

We believe that even the small rise in memory consumption for the test algorithm was due to (most likely) a memory leak in Blender3D. It was determined that if a test 3D scene was used progressively through all test cases without relaunching Blender3D application, then this leads to increase in memory consumption, but if Blender3D is shut down and relaunched again and tested immediately with, say, # of fibers is 1000000, then instead of 192MB consumption, it will give only 167-170 megabytes.

We believe there must be a memory leak in the way Blender3D uses OSL shaders. This memory leak can't be attributed to the shader or algorithm itself, due to the fact that there is no notion of memory allocation/deallocation commands within the OSL environment.

This test algorithm demonstrates also, that in this unpolished version, the time spent computing the image is 157% of what the conventional algorithm offers, however the memory foot-print is much small than what conventional method consumes. During renderings of a test scene with a very large number of fibers, our test algorithm continues to render, while the conventional one fails (application crashed) due to the lack of RAM.

The test run was performed on a Toshiba A505-S6035 laptop that has:

- Intel CPU i7 1.6 GHz (quad-core + hyper-threaded = 8 threads);
- 8 GB of RAM (no swap file);
- OS Ubuntu 14.04 64 bit;
- Blender3D 2.73.

The conventional algorithm starts crashing Blender3D when the hair system has approximately 380000 fibers and more, with the largest successful size of the hair system - 360000 fibers, and memory consumption of 6300 megabytes.

By contrast, the algorithm doesn't crash and, as expected, consumes much less memory: for the 360000-fiber scene it used 190 megabytes (33 times less).

4.2. Experiment setup

In order to measure performance of the proposed algorithm, we would need to create a 3d scene that would have the following:

- a similar number of rendered fibers;
- would use enough resources of the computer so that the performance difference as well as memory consumption difference is measurable;
- lighting conditions as well as material used should be the same;
- preferably a curved surface from which fibers grow. That should test how well a proposed method works with a local fiber coordinate system.

4.3. *Number of fibers*

Since fibers in our method are generated by our algorithm, the number of fibers is a multi-variable parameter, that number can be computed by Expression 4.1, where *phi_step* is a value in integer degrees between two adjacent fibers, when they are placed on one of the internal arcs. For example, value of 1 would give us 360 fibers per internal cylinder, whereas 2 would give us 180 fibers per internal cylinder.

So, if we have 10 internal cylinders in a single hair cylinder, that would give us 3600 fibers per cylinder. Things get more complicated if phi_step changes for every internal cylinder. In our experiment, we set it to be incremented by 1 every time the next internal cylinder is computed. That is done because phi_step acts as a cheap version of a randomizer. So, because of that change, the total number of fibers per hair system can be computed by using Expression 4.1

$$Total\,number\,of\,fibers = \left(\sum_{\varphi_{step}=1}^{N_{internal\,cylinders}} \frac{360}{\varphi_{step}} \right) \cdot N_{internal\,cylinders} \cdot N_{hair\,cylinders} \qquad (4.1)$$

4.4. Conventional and proposed method. Total number of fibers and performance. Test scene 1

We set up two scenes to test difference in performance between a conventional and the proposed method. The number of actual fibers for the proposed method stayed the same, and number of internal cylinders wasn't changed either, while the number of fibers per each internal cylinder was changed. Results for the test scene 1 (renderings of how scene1 looks like for each of hair rendering methods can be found in Figure 35, Figure 36) are provided in Table 2, where RAM and RAM2 are two different memory consumption counters. RAM count is provided by the OS and RAM2 is provided by Blender3D itself.

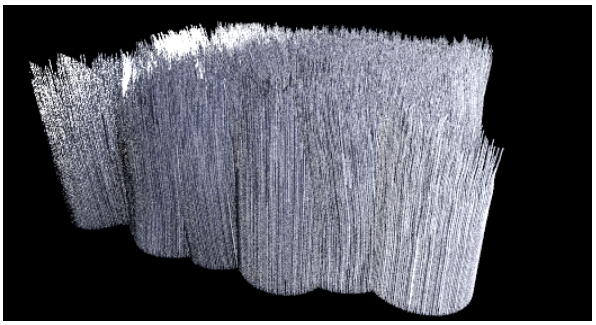


Figure 36. Virtual fibers in hair cylinders. Proposed method

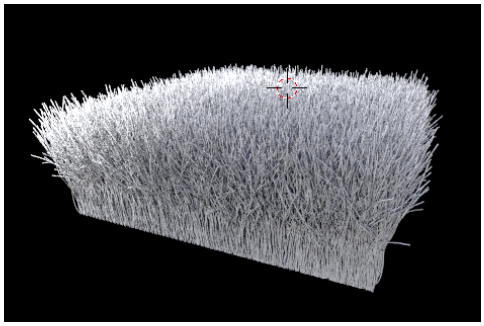


Figure 35. Blender fibers. Standard hair system

# of fibers	Proposed method			Blender Fibers		
	RAM, MB	RAM2, MB	Time, sec	RAM, MB	RAM2, MB	Time, sec
10000	147	22.9	22.3	321	241	1.24
20000	147	22.9	23.7	321	241	1.3
50000	147	22.9	17.4	575	552	1.5
100000	147	22.9	16.48	974	1074	1.93
200000	147	22.9	21.55	1740	2117	2.73
350000	147	22.9	22.7	2686	3696	3.9
500000	147	22.9	20.5	3891	5077	7.7
1000000	147	22.9	22.3	crash	crash	crash

Table 2. Comparison of the proposed method and conventional hair system

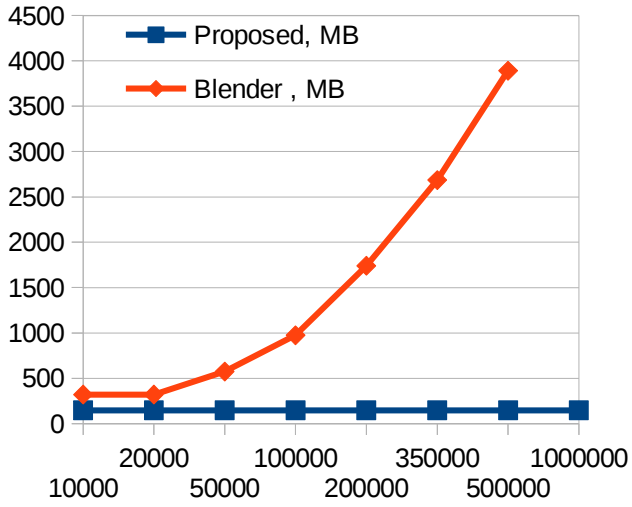


Figure 37. Memory consumption, MB

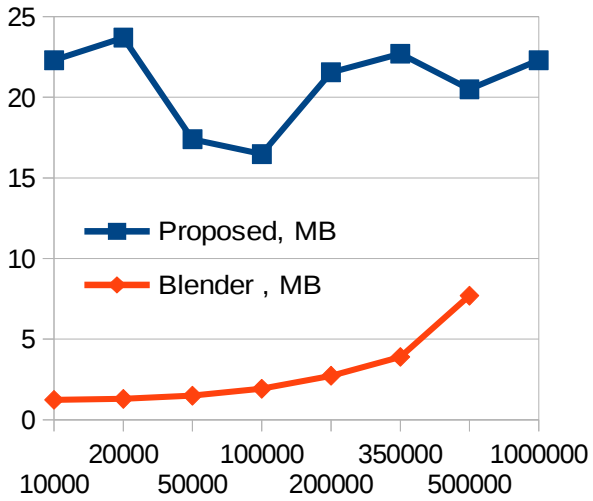


Figure 38. Rendering time, sec

As we can see in Figure 37 (blue line), the memory foot-print for the proposed method stays unchanged no matter how many virtual fibers are computed. Even the time of rendering in Figure 38 (blue line) stays the same pretty much, although oscillations are noticeable, the average would give a constant time of rendering. On the other hand, memory consumption for the conventional algorithm on Figure 37, Figure 38(red lines) changes dramatically and in accordance with the size of the data structure Blender has to keep for the hair system.

Figure 39 shows a rendering of Scene2 – a carpet made using our algorithm. The image was rendered at a resolution of 1920x1080 pixels, 32 samples, full global illumination enabled, 336 000 virtual fibers, 320 blender fibers were used, 1050 virtual fibers per hair cylinder. During rendering, Blender consumed about 118 MB of memory totally and it took 33 minutes 38 seconds to render at 1920x1080 pixel image, standard path tracing method, full global illumination enabled, 32 samples.

Scene2 with a standard blender hair system is rendered (Figure 40) in 2 min, 23 seconds and 3277 megabytes of memory were consumed. It used 336 000 fibers, 1920x1080 pixels, 32 samples, full global illumination enabled.

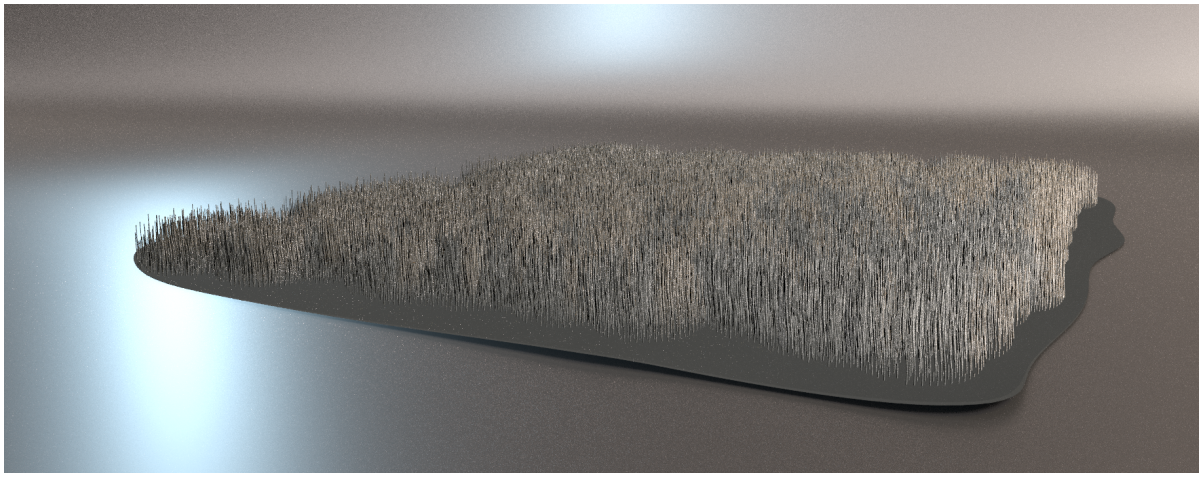


Figure 39. Fibers rendered with a proposed approach

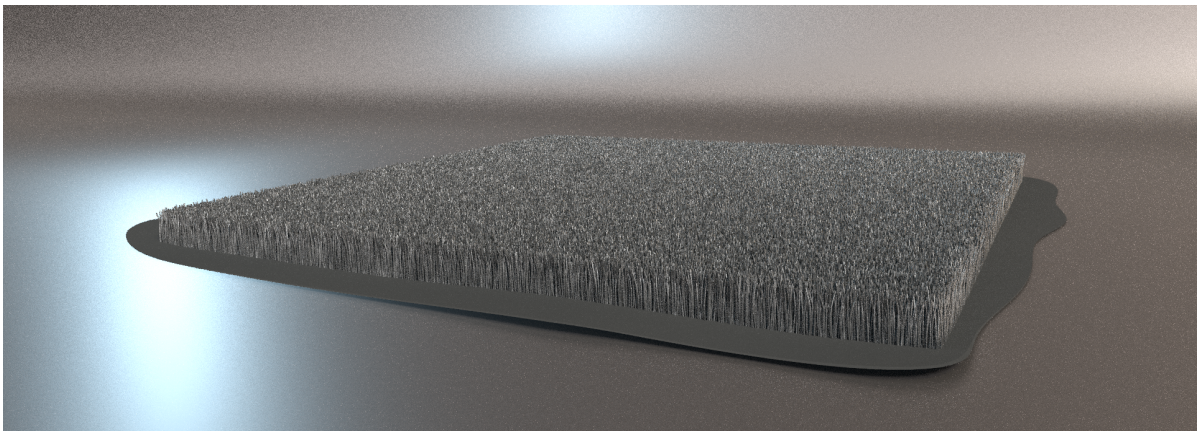


Figure 40. Fibers rendered with a standard Blender hair system

4.5. *Number of internal cylinders vs. number of fibers for one internal cylinder.*

According to our algorithm, it is expected that the number of internal cylinders does affect performance more significantly than a number of virtual fibers per each of those internal cylinders due to the way optimization is set up (Appendix A, page 34). Regardless of how many fibers need to be rendered on a single internal cylinder, the number of those that we check for the intersection with a camera ray is very small (several fibers at most) which can be considered as O(1). On the other hand, the number of internal cylinders linearly affects the performance (Expression 4.2).

$$O(1)\cdot N_{internal\,cylinders} = N_{internal\,cylinders} \qquad (4.2)$$

# int cylinders	# fibers per int. cylinder	2	4	10	30	45	90	180	360
	phi_step	180	90	36	12	8	4	2	1
1		6.9	6.7	7	7	6.86	6.8	7	6.7
5		8.9	8.7	8.8	8.8	8.7	8.6	8.5	8.1
10		11.2	11.4	11.6	11.1	10.9	10.6	10.1	9.7
20		16	16	16.1	15.4	14.8	14.6	12.8	11.8
40		25.3	25.1	24.6	22.8	21.6	20.0	17.9	15.6

Table 3. Number of internal cylinders vs. # of fibers per internal cylinder (seconds)

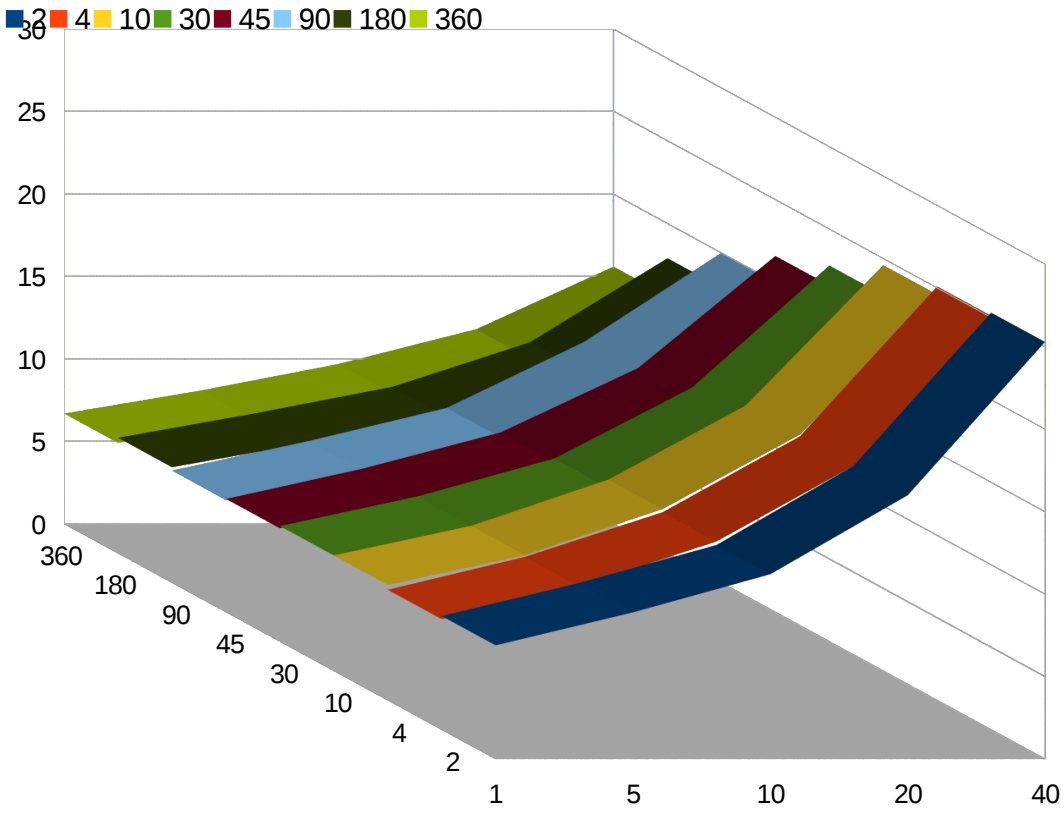


Figure 41. Speed of rendering of a single hair cylinder

The test scene 3 was done on a single hair cylinder and it was designed check performance of the algorithm in cases where there is a single hair cylinder, but number of internal cylinders as well as the number of virtual fibers per each internal cylinder change. It clearly can be seen that the larger the

number of internal cylinders is, the longer rendering time becomes. However, there is another trend which is -- the larger the number of virtual fibers, the faster rendering becomes. This can be explained by occlusion. The larger the number of fibers, the more fibers are there to occlude other fibers. The algorithm runs only until the first intersection between a fiber and a camera ray is detected. Hence more fibers will be computed faster, because they have a much higher probability to be front fibers and so the internal fibers will never be reached, meaning – internal cylinders are less likely to be computed. This means that the number of internal cylinders to be traversed during the computation is less than the total number of those cylinders that was requested. From the table as well as from our understanding of the algorithm, it is clear that the highest cost for algorithm is to compute multiple internal cylinders with very small number of fibers in them. No memory consumption numbers are needed for Table 3, since it is the same for any combination of parameters described in the Table 3 – regardless of how many virtual fibers are used, the only single hair cylinder is stored in memory and thus memory consumption doesn't change.

4.6. *Number of hair cylinders vs. rendering time.*

Here we were testing how well the algorithm can handle multiple hair cylinders when they occlude each other. Two extreme cases are considered:

- Empty - internal cylinders have almost no fibers inside so that they are almost transparent on average (2 fibers per internal cylinder). For this test the number of internal cylinders is 10.
- Full – hair cylinders are almost opaque because of the large number of fibers (360 fibers per internal cylinder). For this test the number of internal cylinders is 10.

# of hair cylinders	1	2	5	10	20	50	100
Empty (2)	4.4	6.6	15.8	31	61.3	228	685
Full (360)	3.2	5.2	13.7	24.6	50.5	134	360

Table 4. Number of hair cylinders vs. rendering time (seconds)

As we can see, 100 empty cylinders with 20 fibers are rendered twice as slow comparing to 100 cylinders with 3600 fibers per hair cylinder. That is the same effect that has been explained in Chapter 4.5 – the more fibers are requested, the less time it will take to render those. That is a proof that our method does what it should – conserve memory for high density hair/fur systems.

4.7. *Drawbacks of the method*

While our algorithm provides substantial memory savings, it does come with some limitations:

- no easy way to control individual virtual fibers inside of a single group;
- this drawback is a limitation of a current implementation of our method, rather than the drawback of the method itself. Our "pure shader" approach doesn't actually displace the surface of a hair cylinder, to which our shader is applied. That is, it doesn't change for the renderer the location of the intersection between a virtual fiber and a camera ray) and it is still located on the surface of a hair cylinder. Proper implementation of displacement can change the point of intersection P of the ray with a fiber, giving a renderer a true position of the fiber *inside* of a hair cylinder, as opposed to the one located on the surface of that cylinder. That should allow for more correct lighting and shadow computation;
- in the case a camera ray penetrates a cylinder through one of its bases, there is no way to use the shader, since Blender doesn't allow to obtain information about that case. Namely, there is no information that would allow to unambiguously determine the position of the cylinder from just a point of intersection of camera ray and base of the cylinder. Because of that, cylinders need to be long enough so that in most cases the base will not be hit by a camera ray. Of course, there are some cases in which a light source goes inside of one of this cylinders, and because of the way our shader works, the lighting gets computed incorrectly,

since light rays doesn't hit any fibers inside because the shader simply doesn't get invoked;

- A more efficient method of placing hair cylinders on a surface is needed, otherwise some cylinders overlap each other almost completely making computations unnecessary longer, while not contributing to the visual fidelity of renderings;
- Method is not suitable (in its current unoptimized state, that is) for fast renderings of scenes that easily fit into memory. In it's current form, our method does not beat Blender's hair rendering system in terms of speed for small hair systems that fit into memory. This is a testament the Cycles hair system, which is quite fast until it runs into memory limits.

4.8. *Conclusions*

Our algorithm demonstrates good results and shows much smaller and, in fact, constant memory footprint, when compared to the standard hair rendering method. Despite such memory savings, the quality of renderings is still on par with todays methods. Considering the fact that rendering of hair and fur systems is still a brute force approach, this algorithm shows very promising results in the task of lowering the memory footprint of such algorithms.

CHAPTER V

FUTURE WORK

5.1. Ways to improve the proposed algorithm

We have considered a number of ways to improve our hair rendering algorithm. One way to improve the algorithm would be an adaptive version of the algorithm that will compute curvature of the surface that spawns off hair cylinders (parent surface). When cylinders are placed, their positions and radii can be controlled by the value of curvature. The larger the curvature of the surface is, the smaller the radius of a hair cylinder should be. That would allow for smaller number of fibers with larger radii to be placed on relatively flat parts of the parent surface, while still retaining detail look of fur/hair at the curved parts of the surface thereby eliminating cases where large enough cylinders can create fibers that "float" above the surface.

Speed is another issue that could be addressed. The current shader has not been extensively optimized, and we feel that there is quite a bit of room for improvement.

REFERENCES

[1] L. Gritz, C. Stein, C. Kulla, and A. Conty, "Open Shading Language," in *ACM SIGGRAPH 2010 Talks*, New York, NY, USA, 2010, pp. 33:1–33:1.

[2] J. T. Kajiya and T. L. Kay, "Rendering Fur with Three Dimensional Textures," in *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1989, pp. 271–280.

[3] A. Meyer and F. Neyret, "Interactive Volumetric Textures," presented at the Eurographics Workshop on Rendering techniques (Rendering Techniques'98), 1998, pp. 157–168.

[4] F. Neyret, "Modeling, animating, and rendering complex scenes using volumetric textures," *IEEE Trans. Vis. Comput. Graph.*, vol. 4, no. 1, pp. 55–70, Jan. 1998.

[5] C. K. Koh and Z. Huang, "A Simple Physics Model to Animate Human Hair Modeled in 2D Strips in Real Time," in *Computer Animation and Simulation 2001*, P. D. N. Magnenat-Thalmann and D. D. Thalmann, Eds. Springer Vienna, 2001, pp. 127–138.

[6] W. Liang and Z. Huang, "An enhanced framework for real-time hair animation," in *11th Pacific Conference on Computer Graphics and Applications, 2003. Proceedings*, 2003, pp. 467–471.

[7] P. Noble and W. Tang, "Modelling and animating cartoon hair with NURBS surfaces," in *Computer Graphics International, 2004. Proceedings*, 2004, pp. 60–67.

[8] C. Yuksel, S. Schaefer, and J. Keyser, "Hair Meshes," in *ACM SIGGRAPH Asia 2009 Papers*, New York, NY, USA, 2009, pp. 166:1–166:7.

[9] K. Ward, F. Bertails, T.-Y. Kim, S. R. Marschner, M.-P. Cani, and M. C. Lin, "A Survey on Hair Modeling: Styling, Simulation, and Rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 2, pp. 213–234, Mar. 2007.

[10] D. Bhagvat, S. Jeschke, D. Cline, and P. Wonka, "GPU Rendering of Relief Mapped Conical Frusta," *Comput. Graph. Forum*, vol. 28, no. 8, pp. 2131–2139, Dec. 2009.

APPENDICES

Appendix A. Pseudo-code of the proposed algorithm

P – (3D point) point of intersection of a camera ray with main hair cylinder;

N – (3D vector) surface normal at point P;

I – (3D vector) direction of a camera ray;

isect – (boolean) value, keeps the result of intersection between a camera ray and a virtual fiber

fiberNormal – (3D vector) computed normal of the surface of a virtual fiber at the point of its intersection with a camera ray;

front_arc, back_arc– (float [2]) a set of values that describe limits of front and back arcs that is a result of an intersection between a shaft (that goes along the ray) and an internal cylinder;

radius – (float) radius of one of internal cylinders;

radius_step – (float) value that is used to decrement radius of an internal cylinder;

num_int_cylinders – (int) number of internal cylinders that need to be checked;

DS – stack-like data structure that stores a tuple of several values that allows to compute fibers for back arcs later.

```
hair_shader(P, N, I){
      isect = 0
      for(i = 0, i < num_int_cylinders and isect == 0, i++){
            compute_lims_for_single_cylinder(P, N, I, r, fN, front_arc,
                  back_arc)
            isect = walk_arc(P, N, I, fiberNormal, front_arc, r)
            r =  r – radius_step
            if (isect == 0) DS.push_back(back_arc, r)
      }
      for(i = 0, i < num_int_cylinders and isect == 0, i++){
            DS.pop_back(back_arc, r)
            isect = walk_arc(P, N, I, fiberNormal, back_arc, r)
      }
      return isect
}

walk_arc(P, N, I, fiberNormal, arc_limits){
      isect = fiber_intersection_routine (arc_limits,P,N,I,fiberNormal)
      if (isect == 1) { compute fiberNormal; return 1 }
      else return 0
}
```

VITA

ANTON ZUYKOV

Candidate for the Degree of

Master of Science

Thesis: A MEMORY CONSERVING RENDERING METHOD FOR HAIR/FUR SYSTEMS IN COMPUTER GRAPHICS

Major Field: COMPUTER SCIENCE

Education:

Completed the requirements for the Master of Science/Arts in your major at Oklahoma State University, Stillwater, Oklahoma in July, 2015.

Completed the requirements for the Engineering degree, Control in Information Automatic Systems (Computer Science) Penza State University, Penza, Russia, May 2006