Advanced Techniques in Real-time Hair Rendering and Simulation

SIGGRAPH 2010 Course Notes

Lecturers:

Cem Yuksel
Texas A&M University
Cyber Radiance

Sarah Tariq
NVIDIA

The latest version of the course notes can be found at
http://www.cemyuksel.com/?x=RealtimeHairCourseNotesSiggraph2010
http://www.sarahtariq.com/HairCourseNotes_SIGGRAPH2010.pdf

Abstract

Hair rendering and simulation have always been challenging tasks, especially in real-time. Due to their high computational demands, they have been vastly omitted in real-time applications and studied by a relatively small group of graphics researchers and programmers. With recent advancements in both graphics hardware and software methods, real-time hair rendering and simulation are now possible with reasonable performance and quality. However, achieving acceptable levels of performance and quality requires specific expertise and experience in real-time hair rendering. The aim of this course is to bring the accumulated knowledge in research and technology demos to real world software such as video games and other commercial or research oriented real-time applications. We begin with explaining the fundamental techniques for real-time hair rendering and then present alternative approaches along with tips and tricks to achieve better performance and/or quality. We also provide an overview of various hair simulation techniques and present implementation details of the most efficient techniques suitable for real-time applications. Moreover, we provide example source codes as a part of our lecture notes.

Latest Version of the Course Notes

The latest version of the course notes can be found at

http://www.cemyuksel.com/?x=RealtimeHairCourseNotesSiggraph2010

http://www.sarahtariq.com/HairCourseNotes_SIGGRAPH2010.pdf

Lecturers

Cem Yuksel

Texas A&M University
Cyber Radiance

Cem Yuksel is the founder of Cyber Radiance LLC and is receiving his Ph.D. from Texas A&M University in 2010. He designed and programmed Hair Farm, a leading hair software plugin for 3ds Max, used by various production studios and individual artists. His published research work on hair includes hair modeling with hair meshes, curve formulations, real-time hair shadows, real-time computation of multiple scattering in hair, and efficient global illumination techniques for hair. His other published graphics research includes methods like mesh colors for efficiently storing color data on arbitrary meshes and wave particles for real-time water simulation.

cem@cemyuksel.com

www.cemyuksel.com
www.cyberradiance.com

Sarah Tariq

NVIDIA

Sarah Tariq is a software engineer on NVIDIA's Developer Technology team, where she works primarily on implementing new rendering and simulation techniques that exploit the latest graphics hardware, and helping game developers to incorporate these techniques. During her time at NVIDIA she has been involved in the development of several game titles for the PC, including Hellgate: London, Supreme Commander and Dark Void, and has helped optimize several other titles. She has presented talks at various conferences, including SIGGRAPH and GDC. Before joining NVIDIA, Sarah pursued graduate studies at Georgia Tech, where she worked on research projects including subsurface reflectance capture of skin.

stariq@nvidia.com

www.sarahtariq.com
www.nvidia.com

Course Overview

8:30 am **Introduction and Fundamentals of CG Hair** [Yuksel]

8:40 am **Data Management and Rendering Pipeline** [Yuksel and Tariq]
- Overview
- Rendering Hair as Poly Lines
- Rendering Hair as Camera Facing Polygons
- Managing Dynamic Hair Data
- Efficiently Sending Data to GPU
- Generating Hair on the GPU

9:15 am **Transparency and Antialiasing** [Yuksel and Tariq]
- Transparency and Blending
- Depth Sorting for Blending
- Avoiding Sorting
- Antialiasing
- Best Practices for Better Performance and Quality

9:30 am **Hair Shading** [Yuksel]
- Kajiya-Kay Shading Model for Hair
- Physically Based Hair Shading
- Improving Shading Performance on the GPU

10:10 am **Q & A** [Yuksel and Tariq]

10:15 am **Break**

10:30 am **Hair Shadows** [Yuksel and Tariq]
- Shadow Maps
- Transparent Shadow Mapping for Hair
- Shadow Filtering
- Simplified Shadow Maps for High Performance

11:05 am **Multiple Scattering in Hair** [Yuksel]
- Introduction to Multiple Scattering
- Dual Scattering Approximation
- Implementation Notes

11:25 am **Hair Dynamics for Real-time Applications** [Yuksel and Tariq]
- Overview of Hair Simulation Techniques
- Fast Simulation of Hair on the GPU
- Handling Hair-Hair Interaction
- Efficient Collision Detection and Handling for Hair
- Simulating Hair with Hair Meshes

12:10 pm **Conclusion / Q & A** [Yuksel and Tariq]

Introduction and Fundamentals of CG Hair

Hair is known to be an extremely important visual component of virtual characters. However, hair rendering has been avoided or substituted by extremely simplified and often highly unrealistic polygonal approximation in most real-time graphics applications. With the advancement of the graphics hardware, we believe that the time has come to handle hair rendering properly in real-time graphics applications. In this course, we overview crucial components of hair rendering and simulation for real-time applications and discuss how high performance results can be achieved with high quality images.

Human hair, especially when it is long, often forms an extremely complicated geometric structure. A person can have over 100 thousand hair strands and each them is an extremely thin fiber and can form rather complicated shapes. Therefore, a full hair model of a person presents various challenges in efficiently handling this complicated structure, in terms of rendering and simulation, as well as modeling.

The geometric complexity of hair also makes it difficult to realistically render it as a large surface, just like any other object. While such approaches are commonly used in practice, it is very difficult, if possible at all, to get realistic results with surface approximations of hair for most hair models. In this course, we do not talk about "fake" hair rendering approaches like these, but concentrate on properly rendering hair similar to the approaches used in offline hair rendering. We also explain how hair can be rendered efficiently to serve the needs of a real-time application.

In computer graphics a hair strand is often represented by a curve with some thickness. This representation ignores the details of the tubular shape of hair fibers, which is often unimportant, because, in most cases, the projected thickness of a hair strand is less than the size of a pixel on the screen. With this representation, a hair model in computer graphics is essentially a collection of curves.

Due to the large number of hair strands in a general hair model, it is often undesirable to explicitly model each and every hair strand. Therefore, most hair modeling techniques provide some form of modeling only a portion of all hair strands (key hairs) and populating the rest of the hair model based on the explicitly modeled ones. There are two main approaches for generating hair strands from the modeled subset.

Single Strand Interpolation creates hair strands around each explicitly modeled curve based on the shape of the curve. Wisps and generalized cylinder based techniques use this approach for generating a complete hair model.

Multi Strand Interpolation creates hair strands in-between a number of explicitly modeled curves by interpolating their shapes.

A recent approach for modeling and representing hair is the **hair mesh** structure, which permits modeling of hair similar to polygonal modeling. A hair mesh is essentially a volumetric structure and the hair strands are generated inside this volume following the shape and topology of the hair mesh.

Data Management and Rendering Pipeline

Content

- Rendering Hair as Poly Lines
- Rendering Hair as Camera Facing Polygons
- Managing Dynamic Hair Data
  - Wisps
  - Key Hair Interpolation
- Efficiently Sending Data to GPU
- Generating Hair on the GPU

Rendering Hair as PolyLines

- Advantages of Lines
  - Faster to render lines than camera facing triangles
  - Easy to implement

Rendering Hair as PolyLines

- Issues with lines:
  - Each draw call is limited to a single integer width (which is the number of pixels on screen)
  - Need to change the line width based on the distance of the head from the camera
    - Otherwise as hair zooms in it will look too thin and as it zooms out it will look too thick
  - To fake a floating point resolution for the line width you can modify the alpha of the lines in addition to their width

Rendering Hair as PolyLines

- Issues with Lines (continued):
  - Lines only support a 1D texture along their length, but you cannot map a texture across their width

Rendering Hair as Camera Facing Quads

- Hair strands can be rendered as Camera Facing Quads
  - Can create these quads by expanding each line into two triangles using the Geometry Shader
  - Or can render degenerate triangles as input and "expand" them in the Vertex Shader

The other option instead of rendering hair directly as lines is to expand the line segments into camera facing quads. That is, for each line segment, expand it into two triangles facing the camera.

This expansion is pretty easy to do in the Geometry shader. We specify the input primitive for the geometry shader to be a line (two vertices), from these we can figure out the tangent, and taking the cross product of that with the eye vector gives us the two axis to expand the quad:

//creating the four

Float3 Tangent = normalize(vertex[1].Position – vertex[0].Position);

Float3 sideVec = normalize(cross(eyeVec, tangent));

float4x3 pos;

float3 width0 = sideVec * 0.5 * width * vertex[0].width;

float3 width1 = sideVec * 0.5 * width * vertex[1].width;

pos[0] = vertex[0].Position - width0;

pos[1] = vertex[0].Position + width0;

pos[2] = vertex[1].Position - width1;

pos[3] = vertex[1].Position + width1;

Rendering Hair as Camera Facing Quads

- Advantages:
  - Hair strands have real world width
  - Can taper hair towards its end
  - Can apply a texture across the hair strand to simulate the look of multiple strands
- Disadvantages:
  - Rendering triangles is more expensive
  - Code is a bit harder/longer

Rendering hair as triangle strips has a number of advantages. The hair strips can have a real world thickness, and this thickness can vary both amongst strands and along the length of a given strand. This allows us to have different levels of detail in different parts of the head, for example the hair near the front of the face can be thinner (with more strands) and the hair near the center of the head (these are strands that will probably get occluded by other strands on top) can be thicker (to better occlude the scalp). Changing the thickness of the hair along its length allows us to taper it towards the bottom, which is important for a realistic look. Finally, we don't have to worry about changing the line width as the hair moves closer or further from the camera.

Rendering triangles can be costly however, both because of the additional amount of geometry that we have to either pass to or create on the GPU and because of the more costly rasterization.

Managing Dynamic Hair Data

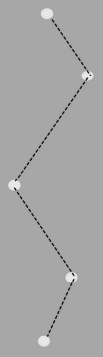
Creating Hair Dynamically

- In most use cases it makes sense to store and simulate only a subset of the hair and to create more hair dynamically for the purposes of rendering
  - Typical human head has ~100,000 strands – simulating all the vertices on all these strands is too much computation
  - Even if we are not simulating hair, depending on the GPU it might be faster to upload to the GPU only a subset of the hair and create the rest of the hair directly on the GPU

Many of the methods used in literature and in practice simulate hair interactions on only a subset of the hair, which are called guide strands, and then render a larger number of hair strands either by interpolating between the guide strands or by clumping rendered hair to guide strands.

Creating Hair from guide strands

Guide Strand

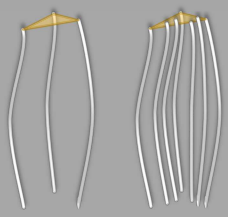


Simulated/Stored Vertices

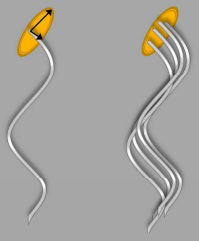
Tessellated Guide Strand

Smoothly Tessellated Hair by creating new vertices

Create New Strands



Multi Strand Interpolation



Single Strand Interpolation

Interpolation

- Multi Strand interpolation
  - Each interpolated strand is created by linearly interpolating the attributes of mutiple guide strands, for example three guide strands
    - The guide strands for a given interpolated strands are rooted at the vertices of the scalp triangle where the interpolated strand is created
    - Each interpolated Strand is assigned a random float2 to use as interpolating weights

Interpolation

- Single Strand Interpolation
  - Each interpolated strand is created by offsetting it from a single guide strand along that strand's local coordinate frame
  - Each interpolated strand is assigned a random float2 to offset it along the x and z coordinate axes (y axis points along the length of the hair)

Interpolating to create more Strands

Multi strand Interpolation | Single Strand Interpolation | Combination

Efficiently sending data to the GPU

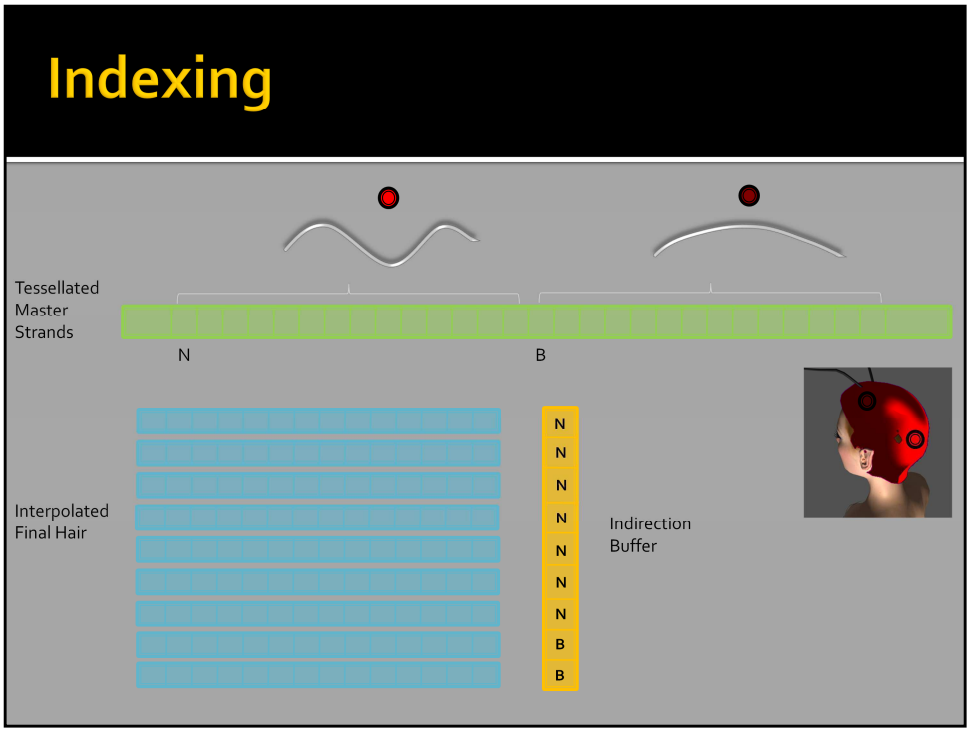
Process

- Create a tessellated dummy hair and render it N times, where N is the number of final hairs to be rendered
- In the Vertex Shader, load from Buffers or Textures storing guide strand attributes
  - Constant attributes like strand texcoords, length, width etc
  - Variable attributes like vertex positions, coordinate frames etc which are potentially calculated every frame

Process

- Stream out the data after each stage to minimize re-computation
  - Tessellate the simulated strands and Stream out
  - Interpolate the tessellated strands and Stream out
  - Render final hair for shading to shadow map
  - Render final hair for rendering
- Each stage uses data computed and streamed out from previous stage

diagrams



Indexing into the attribute buffers

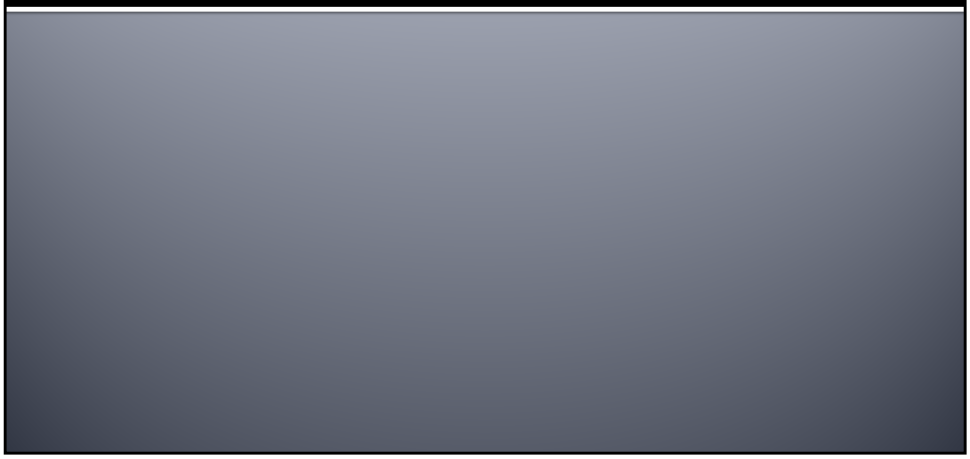
- Need additional buffers that map the instanceID of the rendered hair strand to the correct offset in the attribute buffers
- Variable density of hair across the head also has to be considered in this indexing
- Makes efficiently changing the amount of hair rendered (for example for LOD) hard

Additional Optimization details

- Don't use the GS for creating hair strands.
  - Can use the GS for expanding the lines to triangles but performance gain depends on pipeline load and type of GPU

Generating Hair on the GPU

Using Hardware Tessellation



Generating data on the GPU

- Sending the large amounts of hair geometry to the GPU can become bottlenecked in the DA (Data Assembly)
- Since we are already creating hair from a set of key strands, it makes sense to create this data directly on the GPU
- Direct3D11 class GPUs introduce a tessellation engine which is perfect for this task

We have talked about how to efficiently send all the hair vertices to be rendered on the GPU. However, given the large amount of data that we are trying to render, it is actually more efficient to generate this data directly on the GPU. The newest generation of GPUs introduce functionality to create large amounts of data directly on the hardware; the tessellation engine. In this section we are going to be talking about how to use the tessellation engine to easily and efficiently create hair strands for rendering.

Note that GPUs have had functionality for creating data, the Geometry Shader, for some amount of time. However, the Geometry shader is meant for only small amounts of data expansion, for example for expanding a line into a quad. It is not meant for, and is certainly not efficient at creating the amounts of geometry that we would like to create for all the hair strands.

Benefits

- Faster
- Easy and intuitive
- More programmable
  - Can create geometry only where needed
  - Reduce detail where not needed
- Continuous LOD
- Saves memory and bandwidth

There are a number of reasons why the tessellation engine is useful for creating hair for rendering. The most important advantage is that it is faster to create data using the tessellation engine than it is to create data on the CPU and then upload it to the GPU, or even to render “dummy vertices” on the GPU and then evaluate them in the vertex shader as we were discussing in the previous section. It is also easier to create hair using the Tessellation engine than using the dummy vertex method. Using the Tessellation engine we can have very fine grained and continuous control over the level of detail.

Hardware Tessellation Pipeline

- Current generation of GPUs have support for programmable tessellation
- Two tessellation specific shader stages:
  - Hull Shader (HS) in Direct3D11 and Tessellation Control Shader in OpenGL
  - Domain Shader (DS) in Direct3D11 and Tessellation Evaluation Shader in OpenGL
- One fixed function stage:
  - Tessellator (TS) in Direct3D11 and Primitive Generator

The tessellation engine has three stages, two of them are programmable (the hull and domain shader) and one of the is fixed function (the tessellator).

The Hull Shader is the first new stage and it comes after the vertex shader. The Hull shader takes as input a "patch" - an input primitive which is a collection of vertices with no implied topology. In this stage we can compute any per patch attributes, transform the input control points to a different basis, and compute tessellation factors. Tessellation factors are floating point values which tell the hardware how many new vertices you would like to create for each patch, and are necessary outputs of the Hull Shader.

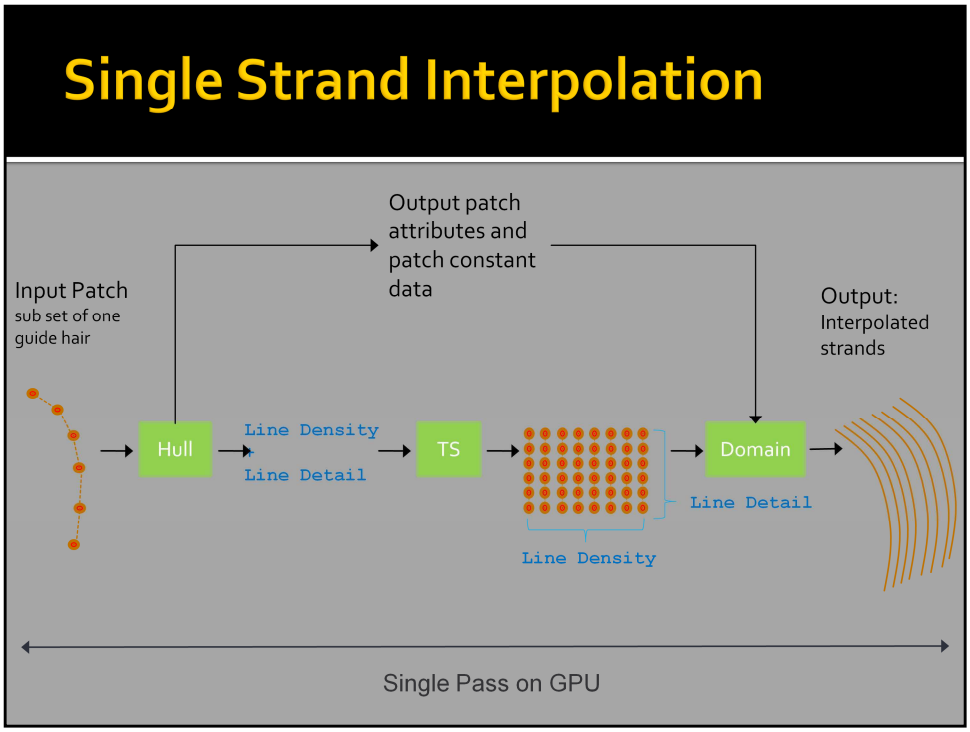
The next stage is the Tessellator, which is fixed function. The tessellator only takes as input the tessellation factors (specified by the Hull Shader) and the tessellation domain (which can be quads, triangles or isolines) and it creates a semi-regular tessellation pattern for each patch. Note that the tessellator does not actually create any vertex data – this data has to be calculated by the programmer in the Domain Shader.

The Domain shader is the last stage in the tessellation engine, and it is at this point that we actually create the vertex data for the tessellated surface. The Domain shader is run once for each final vertex. In this stage we get as input the parametric uvw coordinates of the surface, along with any control point data passed from the Hull shader. Using these inputs we can calculate the attributes of the final vertex.

ISO Line Domain

- Input an arbitrary patch
  - This patch will be different depending on what our method of creating new hair strands is
- For each patch the tessellator creates a number of lines with multiple segments per line
  - The number of lines output per patch and the number of segments per line are user controlled and can be different for each patch
  - The positions of the vertices of the line segments are evaluated in the domain shader
- We can render the output as lines, or expand to camera facing quads in the Geometry Shader

To render hair we are going to be using the Isoline tessellation Domain. In this mode the hardware tessellator creates a number of iso lines (connected line segments) with a multiple line segments per line. Both the number of isolines and the number of segments per isoline can be specified by the programmer in the Hull Shader. The actual positions and attributes of each tessellated vertex are evaluated using the Domain Shader. The output from the Tessellation engine is a set of line segments which can be rendered directly as lines, or they can be rendered as triangles by expanding them to camera facing quads using the geometry shader.



This is an overview of a pass to create and render data using the tessellation engine and the clump based interpolation method. Our input is a patch, which is a section of a guide strand. The hull shader computes the tessellation factors for this patch, which are the number of iso lines that we would like, and the number of line segments per isoline. This data is passed to the tessellator. The Hull shader also calculates any per patch data which might be needed by the Domain Shader. The Tessellator generates the topology requested by the Hull shader. Finally the Domain Shader is invoked once per final tessellated vertex. It is passed the parametric uv coordinates for the vertex, and all the data output from the Hull Shader.

Single Strand Interpolation

- Our input data is simulated guide strands that have been tessellated
  - These strands are appended back to back in one vertex buffer
- Each input vertex contains position, tangent, length and coordinate frame information
  - These vertex attributes are stored separately on the hardware as buffers

The input to our rendering pass is a set of guide strands that have been simulated and tessellated (to make them smooth). The reason we choose to tessellate the strands first is that we are going to be using these tessellated strands for creating both types of final interpolated strands - computing this tessellated data once and reusing it for both types of interpolation saves time.

The data that we have for each vertex includes its position, tangent, length (specified as distance from the root in world space units) and local coordinate frames. This data is stored on the GPU as buffers using structure of arrays format (we have separate buffers for position, tangent etc).

Single Strand Interpolation

- Our input data is going to be read from buffers directly by the shaders, and so we are not going to be sending any real data through the Input Assembler
  - Thus on the CPU side we set the primitive topology to be a patch with a single control point

    3D11_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST

Since we are going to be binding our data as texture buffers which can be sampled in the Hull or Domain shader we don't need to bind any data to the input assembler as vertex buffers (or for that matter index buffers or input layout):

```
unsigned int stride = 0;
unsigned int offset = 0;
ID3D11Buffer* buffer[] = { NULL };
pd3dContext->IASetVertexBuffers(0, 1, buffer, &stride, &offset);
pd3dContext->IASetInputLayout(NULL);
```

In addition, we set the primitive topology to be a patch with a single control point.

After this we call just call draw, with the total number of guide strands as input. We might have to call this draw call more than one time depending on how many isolines and line segments we want per patch (covered in the next slide).

Single Strand Interpolation

- We partition each guide strand is partitioned into one or more patches
- For each patch we create new strands of hair which follow the guide strand
  - The number of new strands created depends both on the LOD (based on distance of head from the camera) and the local density of hair

Dividing input guide strands into patches:

Ideally we would like to specify a patch to be one complete guide hair (and the tessellation engine would then create the specified number of final hair for this patch). However, there is a hardware limit of maximum 64 isolines per patch and maximum 64 segments per isoline, so if we want to create more than 64 segments per isoline or more than 64 isolines per guide strand we have to partition the guide strand into multiple patches.

For each patch that we render the hardware will create a specified number of output isolines. In this example the number of isolines created for a patch is based on two things – the floating point LOD specified globably (calculated based on the distance of the head to the camera) and the local density of hair that an artist has created. This local density is provided as a texture that is mapped to the scalp. Each guide strand has texture coordinates that can be used to lookup the local density.

Hull Shader Code

Patch Constant Shader

```
HS_CONSTANT_DATA_OUTPUT InterpolateConstHSClump(uint iPatch : SV_PrimitiveID) // PatchID is the index of the patch that we are working on
{
    HS_CONSTANT_DATA_OUTPUT output;

    output.nHairsBefore = g_iFirstPatchHair;
    if (iPatch > 0)
        output.nHairsBefore += g_CStrandsPerMasterStrandCumulative.Load(iPatch - 1);

    output.Edges[0] = min(64, g_CStrandsPerMasterStrandCumulative.Load(iPatch) - output.nHairsBefore);
    output.MasterStrandLength = g_tessellatedMasterStrandRootIndex.Load(iPatch);

    output.tessellatedMasterStrandRootIndex = 0;
    if (iPatch > 0)
        output.tessellatedMasterStrandRootIndex = g_tessellatedMasterStrandRootIndex.Load(iPatch - 1);

     //calculate the number of segments that we would want for each line
    output.MasterStrandLength -= output.tessellatedMasterStrandRootIndex;
    output.Edges[1] = min(64, output.MasterStrandLength - g_iSubHairFirstVert - 2);

    output.texcoords = g_Attributes.Load(iPatch); //these are the texture coordinates on the scalp for the root of the guide hair
    output.totalLength = g_Lengths.Load(iPatch);

    // the number of hair that we create is based on a density  texture map
    float4 hairDensityThickness = densityThicknessMapClump.SampleLevel(samLinear,output.texcoords,0);
    output.hairDensity = hairDensityThickness.r;

    // g_fNumHairsLOD  specifies the LOD that we would like, based on the distance of the head from the camera
    output.Edges[0] = max(output.Edges[0] * g_fNumHairsLOD, 1);

    return output;
}
```

The Hull Shader is split into two parts, the main shader which operates once on each input control point, and the patch constant function which is invoked once per patch. In our implementation we are loading control point data for a patch from a buffer, so the input to the hull shader is a dummy patch consisting of a single control point. Since we have only one input control point we are using the patch constant function for all of the computation and our main shader is null.

This listing shows the Patch Constant Shader, which calculates the tessellation factors and other data for each patch.

Output.Edges is the tessellation factors, its semantic is SV_TessFactor. Edges[0] is the amount of iso lines that we would like created for this patch, and Edges[1] is the amount of segments that we would like in each line. As we mentioned before, the number of isolines per patch and the number of segments per isoline cannot exceed 64, so the calculation of both Edges[0] and Edges[1] has to take this into account.

Hull Shader Code

Main Hull Shader

```
[domain("isoline")]  //specify that the tessellation domain is isoline (vs triangle or quad)

[partitioning("integer")] //specify an integer partitioning (vs pow2 etc)

[outputtopology("line")] //specify that the output topology should be lines (vs point or triangle)

[outputcontrolpoints(1)] //we want just one output control point, since all our real data is in the patch constant output

[patchconstantfunc("InterpolateConstHSClump")] // specify the function name of the patch constant function which should be used

void InterpolateHSClump(InputPatch<DUMMY, 1> inputPatch)
{
   //there is no calculation here in our case, since we have computed everything in the patch constant function
}
```

Domain Shader

- The domain shader invoked once per final vertex
- Each vertex gets its parametric location in the tessellated patch
  - For isolines this is a float2
    - uv.x [0,1) the index of the vertex in the isoline
    - Uv.y [0,1) the index of the line this vertex belongs to
- Using these uv along with the patch ID we can figure out the unique identifiers for the vertex and hair strand that we are operating on, and load data accordingly

The Domain Shader is invoked for each final vertex that is created. As input to the domain shader we get the patch constant data and the per patch control point data that we had output in the Hull shader. We also get as input a number of system generated values, including SV_DomainLocation which gives the parametric uv coordinates of the current vertex in the tessellated patch. We also get the id of the patch that we are operating on (SV_PrimitiveID). Using these values we can figure out which vertex and which strand we are operating on. These indices can then be used to look up the guide strand attributes and also the random offsets that we are going to use to offset this generated strand from the input guide strand.

Domain Shader Code

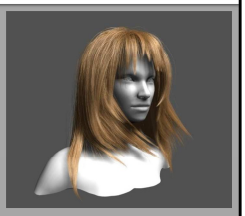
```
[domain("isoline")]
HairVertex InterpolateDSClump_NORMAL(OutputPatch<DUMMY, 1> inputPatch,
HS_CONSTANT_DATA_OUTPUT input,  // this is data written out by the patch constant function
float2 uv : SV_DomainLocation, //this the parametric location of the current vertex in the tessellated patch
uint MasterStrandNumber : SV_PrimitiveID) //this is the ID of the current patch
{
    HairVertex output;

    //calculate the index of this vertex from the root, and the index of the strand it belongs to in the patch
    //using the hardware generated parametric uv and the total tessellation factors output from the Hull Shader
    uint iHairInsideCurMasterStrand = (int)(uv.y * input.Edges[0] + 0.5);
    uint vertexID = (int)(uv.x * input.Edges[1] + 0.5) + g_iSubHairFirstVert;
    uint InstanceID = input.nHairsBefore + iHairInsideCurMasterStrand;

    //calculate the location that we have to load data from
    int vertexIndex = input.tessellatedMasterStrandRootIndex+vertexID;

    //load the random offset that this interpolated strand is going to use to offset itself from the guide curve
    float2 clumpCoordinates = g_StrandCircularCoordinates.Load(InstanceID & g_NumInterpolatedAttributes);

    //load the attributes for the vertex
    float lengthToRoot = g_TessellatedLengthsToRoots.Load( vertexIndex );
    float4 masterVertexPosition = g_TessellatedMasterStrand.Load(vertexIndex);
```

Domain Shader Code

```
//jitter the position slightly to create realisic random deviations in the strands
float2 jitter = g_strandDeviations.Load((InstanceID & g_NumInterpolatedAttributes)
                * g_TessellatedMasterStrandLengthMax + vertexID);

//if we are creating curly hair then add pre calculated curl deviations
if(doCurlyHair)
   jitter += g_curlDeviations.Load(MasterStrandNumber*g_TessellatedMasterStrandLengthMax + vertexID);

clumpCoordinates += jitter;

//load the coordinate frames
coordinateFrame4 cf;
cf.yAxis = g_tessellatedCoordinateFrames.Load( vertexIndex*3 + 1);
cf.zAxis = g_tessellatedCoordinateFrames.Load( vertexIndex*3 + 2);

//calculate the position for the newly created hair vertex
float radius = g_clumpWidth * ( g_topWidth*(1-lengthToRoot) + g_bottomWidth*lengthToRoot );
output.Position.xyz = masterVertexPosition.xyz
                      + cf.yAxis.xyz*clumpCoordinates.x*radius
                      + cf.zAxis.xyz*clumpCoordinates.y*radius;


//more code below to calculate other attributes like tangent, width, texture etc.
```

Level of Detail

- Can use the size of a projected segment or distance to the head to decide on the LOD
  - Low LOD levels would use less number of hair, less segments per hair, and thicker lines



Full Level of Detail

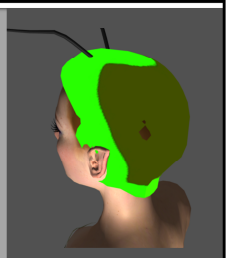
0.5 LOD, 2x faster to render. Image on Left is the actual image. Image on the right shows the actual quality of hair that we are rendering at 0.5 LOD

Being able to dynamically reduce the complexity of the rendered hair (and thus increase the performance) is very important for real time applications. The level of detail for hair can be based on the distance of the hair/head from the camera, the importance of the character, the probable occlusion of the patch or any other factor. Using the tessellation engine we can change the LOD per patch by changing the number iso-lines or the number of segments per line.

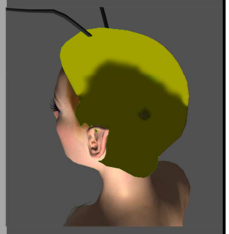
In the images on the right we are showing the same hair style under two levels of detail. At the top we have the hair rendered at full LOD, and at the bottom we have scaled down the rendering and increased performance by 2x. In this case we are creating and rendering less hair strands, although we are making the strands a bit wider so that there is no visible reduction in the density of hair. At the bottom right we show what that lower level of detail would look like if you zoomed in.

LOD

- Can also use artist defined LOD maps
  - Artists can create different looks for different LOD of the hairstyle.
    - For example a lower LOD can have large strips just on the top of the head along the parting of the hairstyle.
  - These LODs can then be transcribed into textures
  - The Hull Shader will blend between appropriate LOD textures to decide on the line density, and line thickness



Near

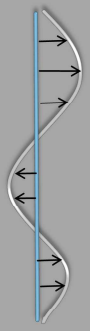


Far

Instead of linearly decreasing the amount of hair strands as LOD decreases we can also have an artist create density and thickness maps for different discrete LODs, and then blend between them to determine the amount of hair and its thickness for a particular LOD. This way we can use the computational resources available (the limited number of hair that we can create) in the region where the artists feel they will have the highest impact.

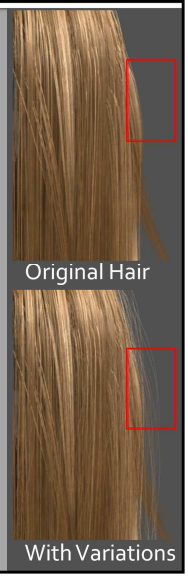
Curly Hair

- Pre-create and encode additional curl offsets into constant buffers or textures
- These offsets are added to the strand interpolation offsets
- Can either be created procedurally, or artists can create example curls and the offsets can be derived from those



Random Variations

- Random variations can similarly be added to interpolated hair vertices to give hair a more realistic look
- outputVertex.Position.xyz = guideVertexPosition.xyz + coordinateframe.xAxis.xyz*(clumpCoordinates.x + randomVariation.x)*ClumpRadius + coordinateframe.zAxis.xyz*(clumpCoordinates.y + randomVariation.y)*ClumpRadius;



An important part of creating realistic hair is having randomness between hair strands. Without this we get a look that is too smooth and synthetic, as shown in the top figure. Adding randomness to strands gives a more natural look (bottom). Since our model for rendering hair is based on interpolating many children hair from a small set of guide hair, we need to introduce this randomness at the interpolated hair level. The method we outline here is similar to that presented by [Choe and Ko 2005]. We pre-compute a small set of smooth random deviations and apply these to the interpolation coordinates of the interpolated hair. (Without loss of generality) the slide above shows how deviations are applied for clump based hair.

Creating Deviations

We create two types of deviations for the hair. The first, applied to a large number of the hair strands, is small deviations near the tips. In our images we have applied this deviation to 30% of the hair strands. The second, applied to only a small percent of the hair strands (for example 10%), is deviation all along the strands. This second type of deviation is what you see highlighted in the red box.

Transparency and Antialiasing

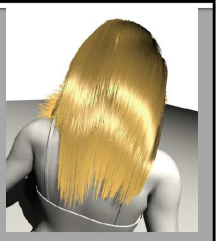
Contents

- Overview
- Transparency
- Depth sorting for Blending
- Avoiding sorting
- Antialiasing
- Doing Transparency and Shadowing together

Overview

Overview

- Hair strands are very **thin**, and projected onto a screen they are usually much thinner than a pixel
  - If we try to render very thin geometric strands we run into aliasing
  - can either use antialiasing techniques, or use thicker lines with transparency
- Hair strands, especially light colored hair, are semi **transparent**
  - Need a way to render transparent geometry



No Transparency

With Transparency, α=0.2

Images from Sintorn08

Transparency

Dealing with Transparent Geometry

Alpha To Coverage

- Also called screen door transparency
  - Used in the NVIDIA Nalu demo
- Benefits:
  - requires no sorting, works with MSAA (multi-sample anti-aliasing), trivial to use in GPUs
- Drawbacks:
  - Using alpha to coverage disables earlyZ in hardware (in order to get earlyZ, which is useful for culling extraneous shading work, you have to render all the geometry to the depth buffer in a separate pass)
  - Do not get very smooth images, alpha to coverage is better suited for "cut out" transparency, e.g. in foliage

Alpha blending

- Commonly used to render transparent objects
- The result of blending a fragment with color c_{in} and transparency α_{in} with the current frame buffer color c_{old} is

$$c_{new} = \alpha_{in} * c_{in} + (1 - \alpha_{in}) * c_{old}$$

- For alpha blending to work properly need to render the transparent geometry sorted back to front (from the camera's perspective)

Blending with Correct Sorting

Blending with Incorrect Sorting

Images from Kim03

Tae-Yong Kim, "Algorithms for Hardware Accelerated Hair Rendering" Game Tech 2003

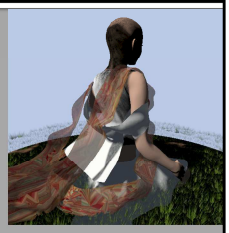
Sorting

Sorting

- Can sort on the CPU, but this will limit the amount of geometry that can be rendered in real time
- Since we want to render up to a 100,00 strands we would instead like to sort the hair directly on the GPU

Depth Peeling

- Depth peeling [Everitto1] can be used to sort fragments
- Process:
  scene is rendered many times, each time the depth buffer from the last pass is read in the pixel shader to reject any fragments with depth <= value in depth buffer
- However, depth peeling is not practical for the large depth complexity typical of hair



Depth peeling with 5 layers

Reference

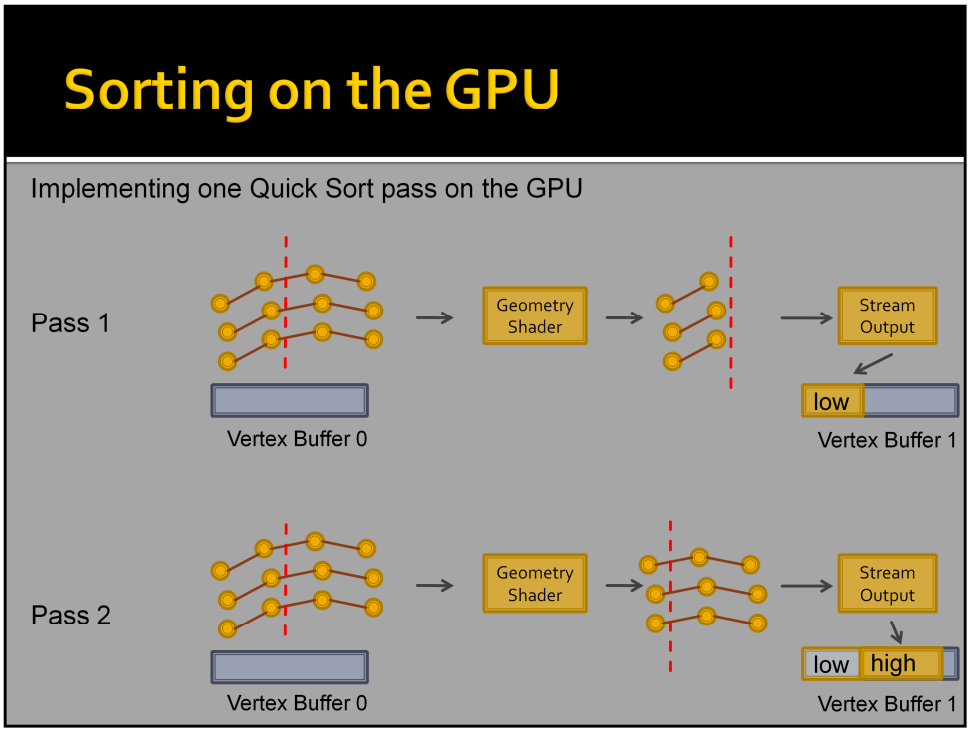
Images from Enderton10

Sorting on the GPU

- Can sort hair line segments into buckets using Quick Sort [Sintorno8]
  - Implemented on the GPU using Geometry Shader and Stream Output (Transform Feedback) and two vertex buffers (for ping ponging)
  - One pass of quick sort requires partitioning the line segments into two subsets; those nearer to the camera than a pre-determined middle, and those further.

Sorting on the GPU

Implementing one Quick Sort pass on the GPU



Sorting on the GPU

- Line segments spanning buckets can be split using the Geometry Shader
  - however this leads to variable amounts of geometry and more cost, and the improvement in the resultant image is almost negligible
- Can also sort all the hair line segments completely using GPU based Radix Sort [Satish09], [Sintorn09]
  - Implemented in CUDA
  - Radix Sort is available in CUDPP (CUDA Data Parallel Primitives Library)

Satish, Harris and Garland. “Designing Efficient Sorting Algorithms for Manycore GPUs”

Sintorn, Assarson, Olsson and Billeter. “Radix sort of line primitives in CUDA for real-time Self-Shadowing and Transparency of Hair”

Avoiding Depth Sorting

- "Fake" Transparency in Final Hair Image

subset 1 subset 2 subset 3 Final blended image

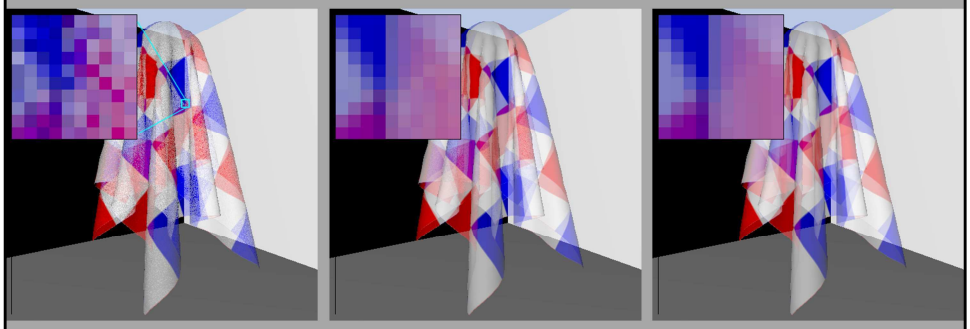
Stochastic Transparency

- Enderton01. Extend screen-door transparency with randomly chosen sub-pixel stipple patterns
- Does not require sorting, can be done in a single or couple of passes
- Results in correct color on average, but introduces noise
  - Noise can be reduced by more

15,000 transparent hairs, 6,000 transparency-mapped Grass cards and cloth rendered with transparency and shadows at 26fps.
Enderton01

Stochastic Transparency, Eric Enderton, Erik Sintorn, Peter Shirley, David Luebke. I3D 2010

Stochastic Transparency



Basic stochastic transparency

Depth-sampled stochastic transparency

Reference image

All images use 8 samples per pixel, no anti-aliasing, and an alpha of 40%.
Images from Enderton 01

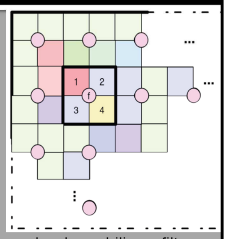
Antialiasing

Antialiasing

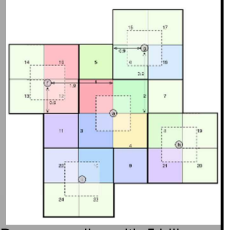
- Sources of aliasing in Hair rendering:
  - Geometric aliasing – thin geometric primitives
  - Shader aliasing – sharp changes in shading, mostly arising from specular highlights and shadows
- Antialiasing options:
  - Pre-filter
    - works well for points and lines, not for triangles
    - Does not work well with depth buffer
  - Over-sample:
    - Super Sampled AA (SSAA)
    - Multi Sampled AA (MSAA)

Super Sampled AA

- Simply render scene at a higher resolution and then down sample to a lower resolution output
- Incurs a large texture bandwidth and fillrate hit
- Provides geometric and shader antialiasing
- Can use different reconstruction filters



hardware bilinear filter based down sample



Down sampling with 5 bilinear tap rotated filter
Images from Lorach07

High Quality Antialiasing. Tristan Lorach, 2007. http://developer.download.nvidia.com/SDK/10.5/opengl/src/FroggyAA/doc/FroggyAA.pdf

Muti Sampled AA

- Optimization of super sampling where we shade each fragment once but evaluate depth and stencil at the sample rate
  - Example:
    - Two triangles go through this pixel
    - Color for each triangle is evaluated only once – at the pixel center (using the pixel shader)
    - Depth tests are performed at the four x locations
    - Samples that pass the depth test get the color from the pixel shader

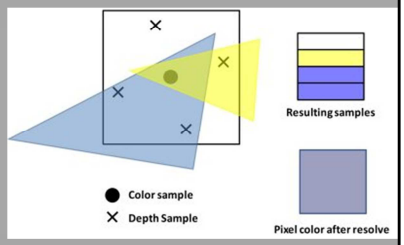


Image from Thibieroz, ShaderX7

Note that with MSAA edges of polygons are antialiased but interiors of polygons are not.

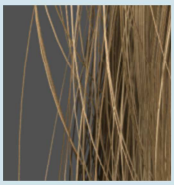
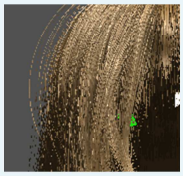
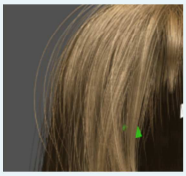
Nick Thibieroz, Deferred Shading with Multisampling Anti-Aliasing in DirectX10. ShaderX7

Multi Sampled AA

- Implemented in hardware and very easy to invoke using graphics APIs:
  - D3d11:
  - pDeviceSettings->d3d11.sd.SampleDesc.Count = D3DMULTISAMPLE_8_SAMPLES; //8x MSAA
  - OpenGL :
  - glEnable(GL_MULTISAMPLE);
  - glRenderbufferStorageMultisampleEXT(GL_RENDERBUFFER_EXT, numSamples, ...);//create an MSAA color and depth buffer

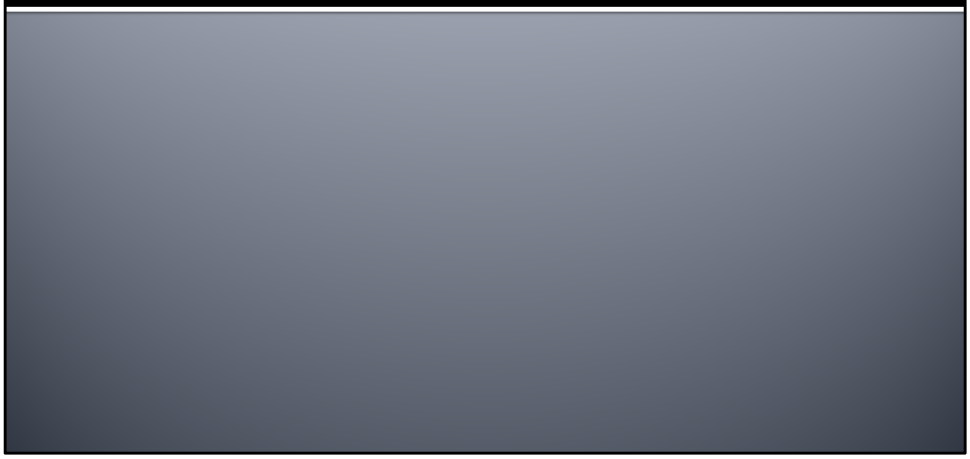
Comparison

Original image
1024x768

No AA	8x MSAA	2x SSAA
51 fps	48fps	25 fps
		
74 fps	71 fps	63fps
		

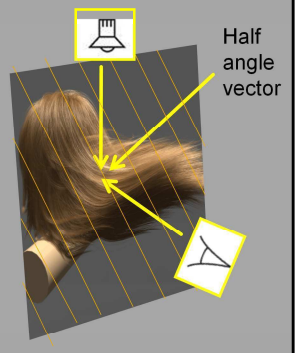
Detailed views

Doing Transparency and Shadowing together



Doing Transparency and Shadowing together

- Can also try to handle rendering transparency and shadowing at the same time [Ikits04]
- The main advantage is that memory requirements are reduced to a single 2D shadow buffer

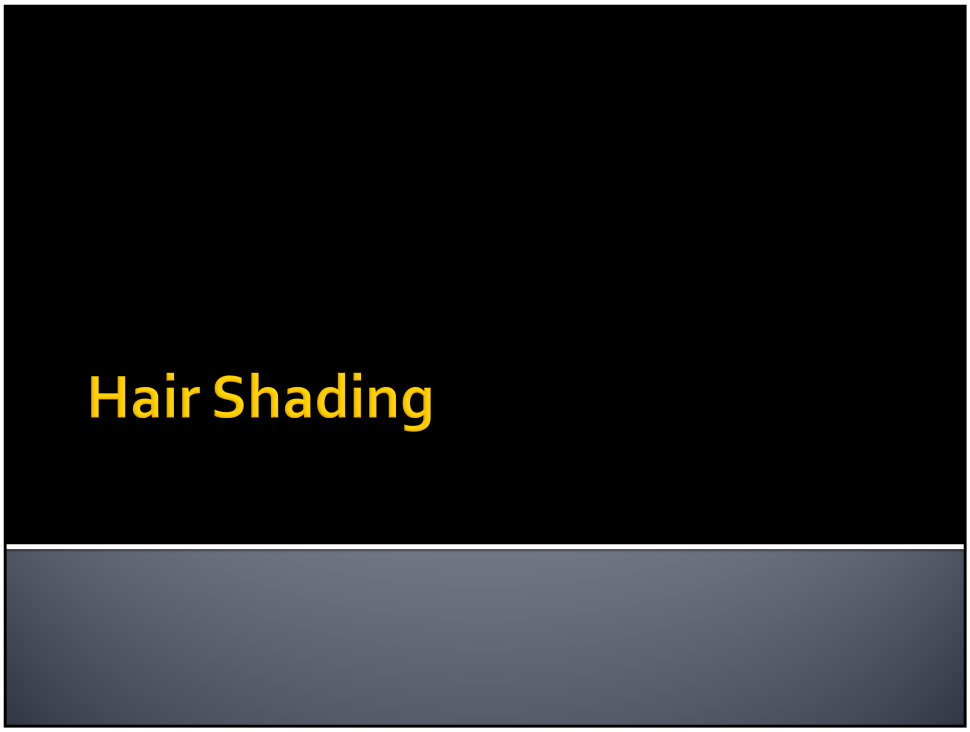


Volume Rendering Techniques, Milan Ikits, Joe Kniss, Aaron Lefohn, Charles Hansen. Chapter 39, section 39.5.1, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics(2004).*

Half Angle Slice Rendering

- Compute vector half way between view and light direction
  - Render hair to light and camera buffers as series of slices perpendicular to this vector
- For each slice
  - First render slice from the eye's point of view, using the results of the previous light pass to shadow the samples
  - Then render the same slice from the light's point of view to calculate the intensity of light arriving at the next slice

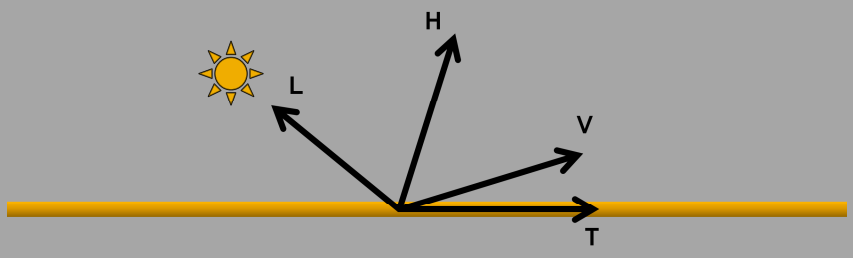
This technique allows us to accumulate the shadowing from the light at the same time at that we are alpha blending the slices



Since hair strands are very thin, when shading hair we assume that the projected thickness of a hair strand on the screen is smaller than the size of a pixel. With this assumption, we ignore the shading variations along the thickness of a hair strand and compute the shading function for the whole thickness of the hair strand. Therefore, we cannot use the surface normal of a hair strand fiber for shading hair with this assumption. Instead, we use the tangent direction of the hair strand and consider the hair strand as a thin tube that is aligned with this direction.

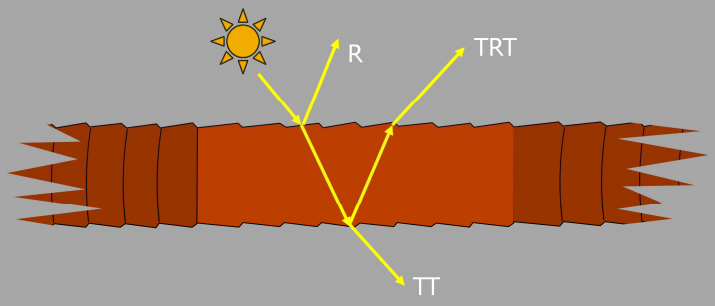
Kajiya-Kay Shading Model

- Kajiya-Kay Model (1989)



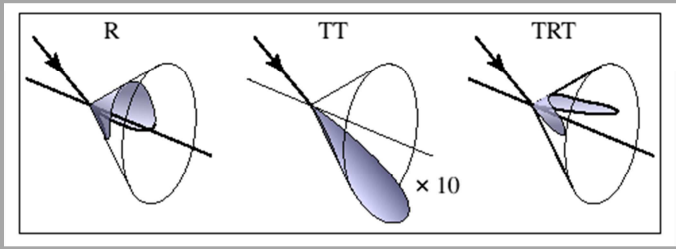
Physically Based Hair Shading

- Marschner et al. (2003)



Physically Based Hair Shading

- Marschner et al. (2003)



Improving Shading Performance

- Use precomputed tables
- Compute shading on vertices only

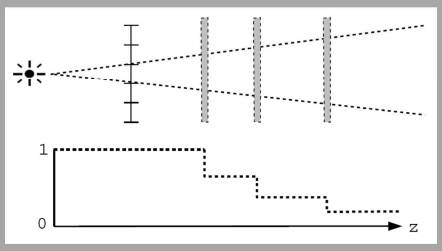
Hair Shadows

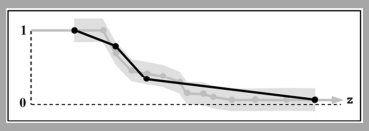
Shadow Maps

- Shadow Maps [Lance Williams, 1978]
  - Depth Map
  - Binary Decision

Transparent Shadow Mapping for Hair

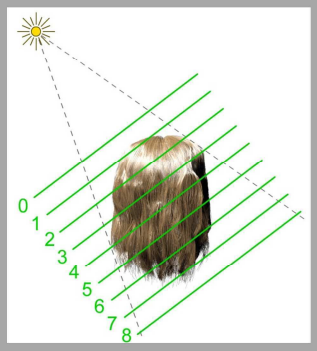
- Deep Shadow Maps [Lokovic and Veach 2000]
  - Multiple depths per pixel
  - Multiple opacities per pixel
  - Compress for efficiency
  - **Offline**

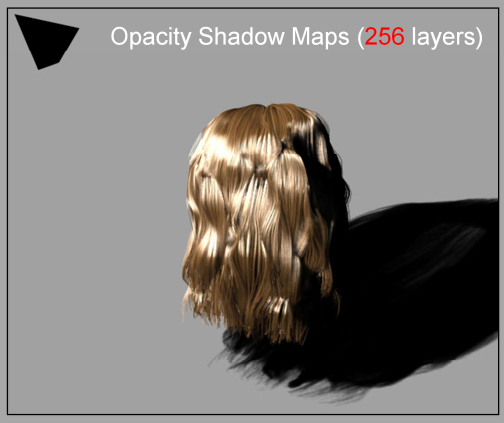




Transparent Shadow Mapping for Hair

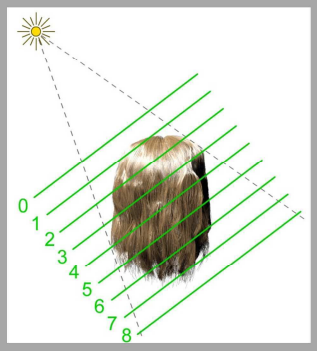
- Opacity Shadow Maps [Kim and Neumann 2001]
  - Opacity Layers
  - **Interactive**
  - **Layering Artifacts!**

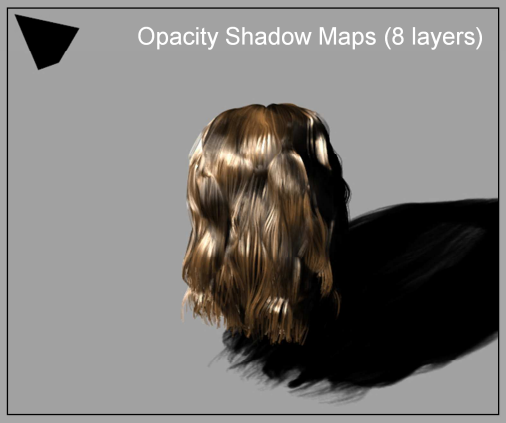




Transparent Shadow Mapping for Hair

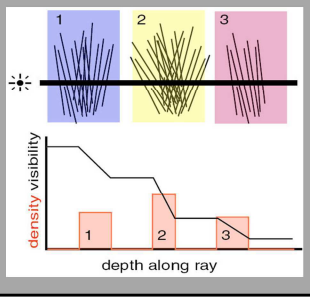
- Opacity Shadow Maps [Kim and Neumann 2001]
  - Opacity Layers
  - **Interactive**
  - **Layering Artifacts!**

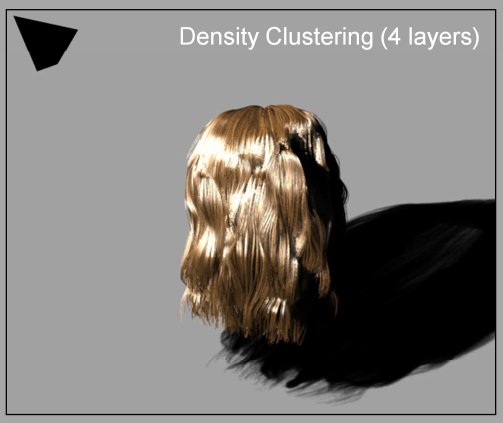




Transparent Shadow Mapping for Hair

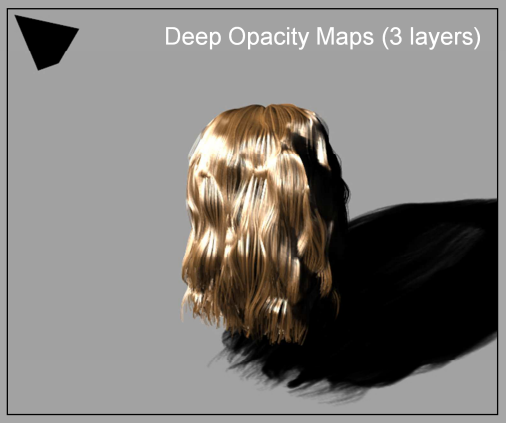
- Density Clustering [Mertens et al. 2004]
  - Per pixel layering
  - K-means clustering
  - **Real-time**
  - **Inaccuracy Artifacts!**



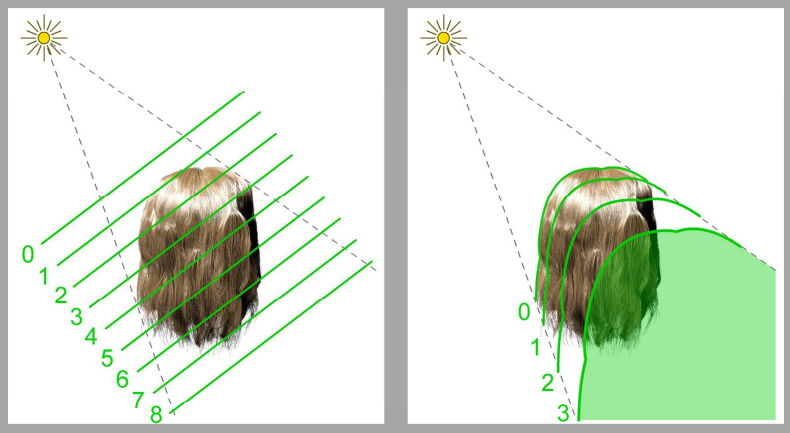


Transparent Shadow Mapping for Hair

- **Deep Opacity Maps** [Yuksel & Keyser – Eurographics 2008]
  - Depth Map
  - Opacity Map
  - **Real-time**
  - Artifact Free!



Deep Opacity Maps

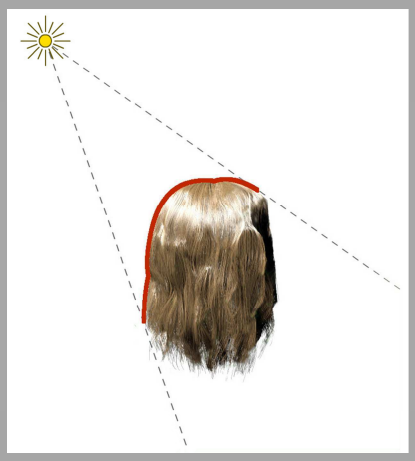


Opacity Shadow Maps

Deep Opacity Maps

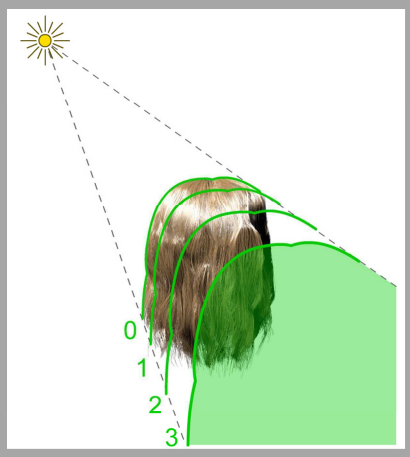
Deep Opacity Maps

- **Pass 1:** Depth Map
  - z_o per pixel



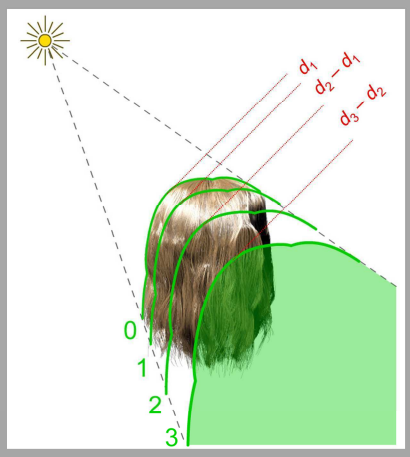
Deep Opacity Maps

- **Pass 2:** Opacity Map
  - Layers:
  - z_0 → $z_0 + d_1$
  - $z_0 + d_1$ → $z_0 + d_2$
  - $z_0 + d_2$ → $z_0 + d_3$
  - …
  - d_1, d_2, d_3… are user defined



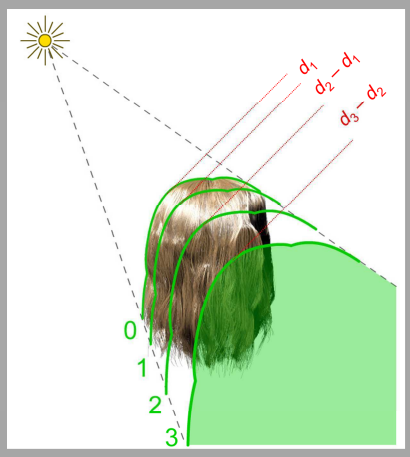
Deep Opacity Maps

- Layer Sizes
  - d_1
  - $d_2 - d_1$
  - $d_3 - d_2$
  - …
  - can be different!



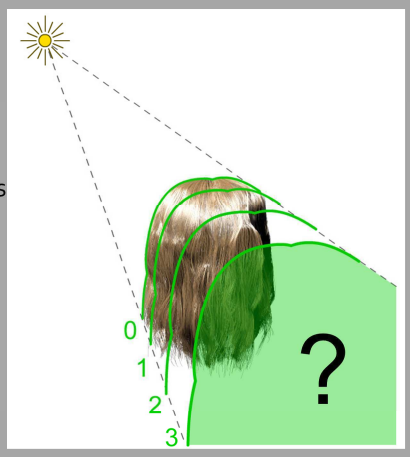
Deep Opacity Maps

- Layer Sizes
  - s = d_1
  - Alternatives:
    - s, s, s, s, … (constant)
    - s, 2s, 4s, 8s, … (powers of 2)
    - s, s, 2s, 3s, 5s, .. (Fibonacci)
    - **s, 2s, 3s, 4s, … (linear)**



Deep Opacity Maps

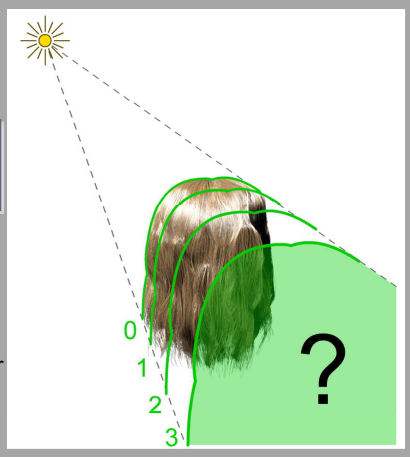
- Beyond the last layer
  - Ignore?
    - Won't cast shadows
  - Add to the last layer?
    - Cast shadows onto themselves
  - Increase the last layer size?
    - Reduce accuracy



Deep Opacity Maps

- Beyond the last layer
  - Ignore?
    - Won't cast shadows
  - Add to the last layer?
    - Cast shadows on themselves
  - Increase the last layer size?
    - Reduce accuracy

Transmittance beyond the last layer should be close to zero anyway!



Deep Opacity Maps

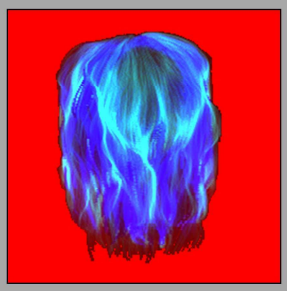
- Depth Map
  - can be **8-bit**, 16-bit, or 32-bit
- 3 opacity layers
  - **Single Texture**

    R: depth (z_0)

    G: layer 1 opacity

    B: layer 2 opacity

    A: layer 3 opacity

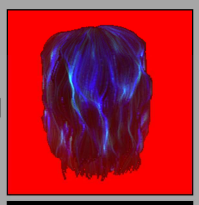


Deep Opacity Maps

- 7, 11, 15… opacity layers
  - **Multiple Draw Buffers**

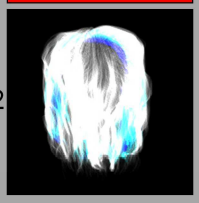
R_1: depth (z_o)
G_1: layer 1 opacity
B_1: layer 2 opacity
A_1: layer 3 opacity

Texture 1

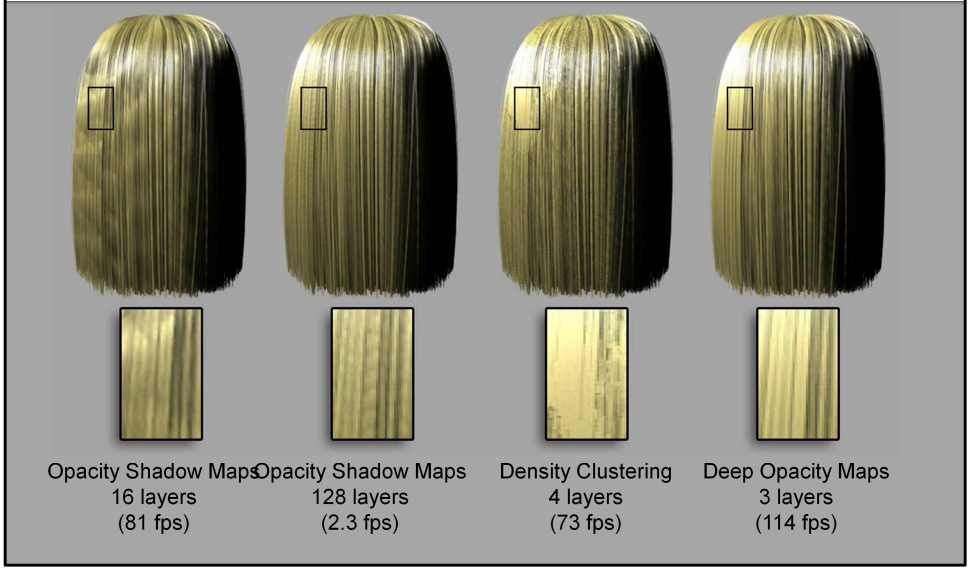


R_2: layer 4 opacity
G_2: layer 5 opacity
B_2: layer 6 opacity
A_2: layer 7 opacity
…

Texture 2

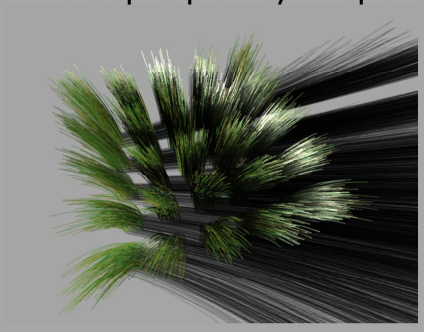


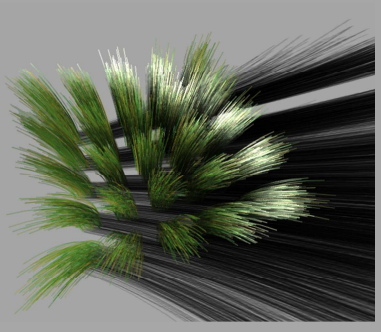
Deep Opacity Maps



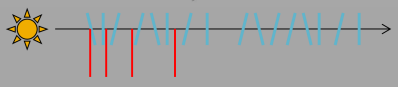
Deep Opacity Maps

- Deep Opacity Maps

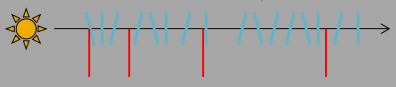




3 layers

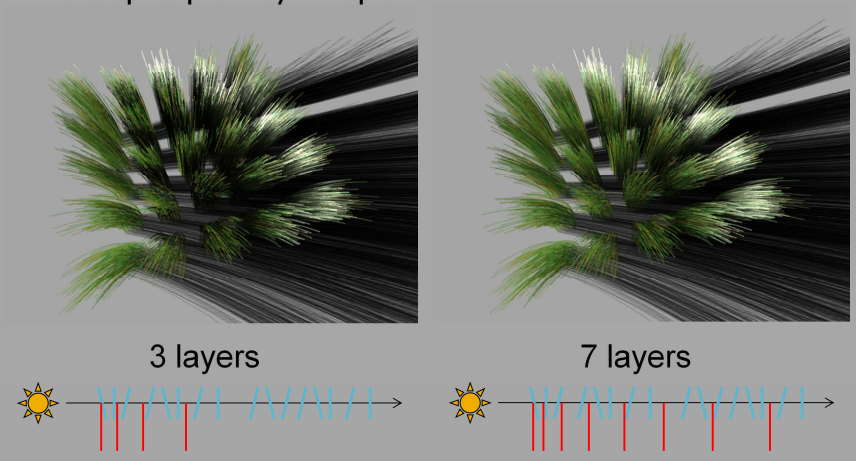


3 LARGER layers



Deep Opacity Maps

- Deep Opacity Maps



Results

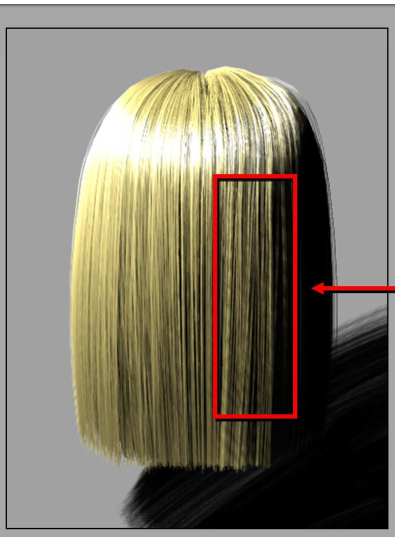
Deep Opacity Maps + Shadow Map

Deep Opacity Maps

- Direct illumination (no shadow) captured correctly
- Concentrate accuracy to where the shadow begins
- Interpolation is moved to within hair volume
- Layering artifacts are hidden
- Fewer layers (less memory)
- 2 pass shadow generation (fast)

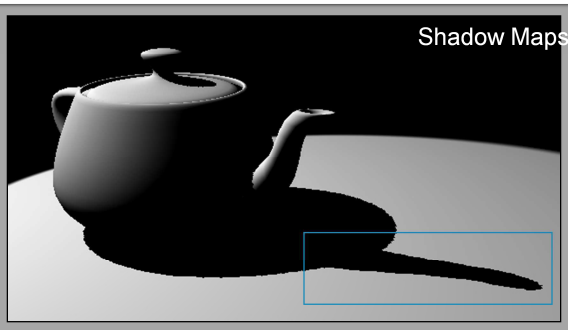
Shadow Filtering

- Flickering?



Shadow Filtering

- Flickering?
  - Same as shadow maps



single look-up

multiple look-up

Shadow Filtering

single look-up

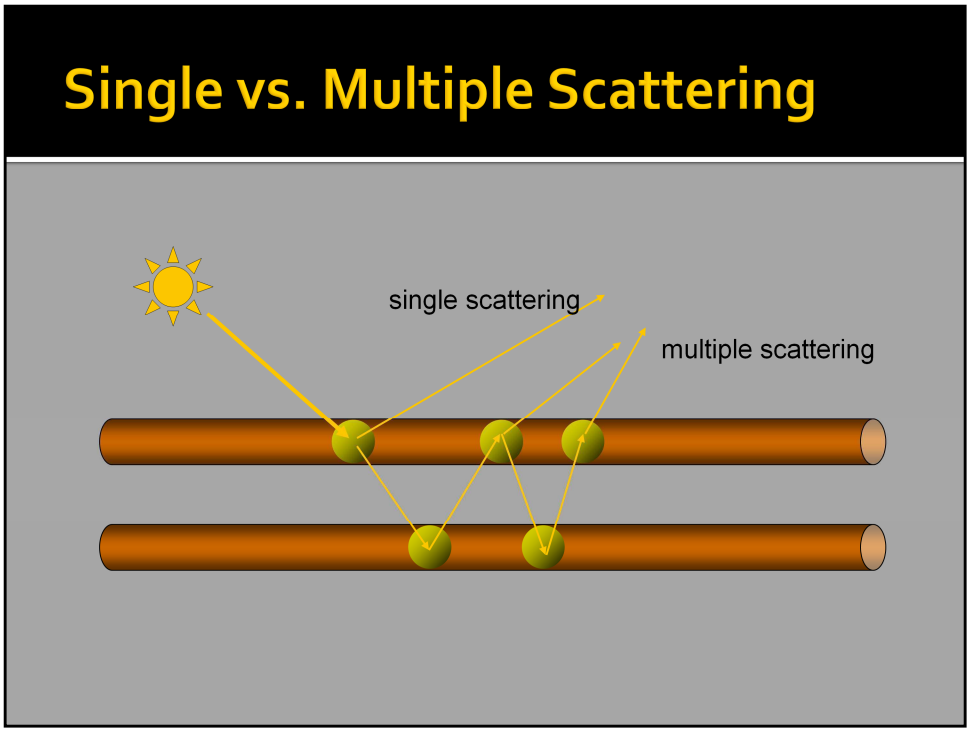
multiple look-up

Multiple Scattering in Hair

Multiple Scattering in Hair

Fiber Scattering Only

Full Solution



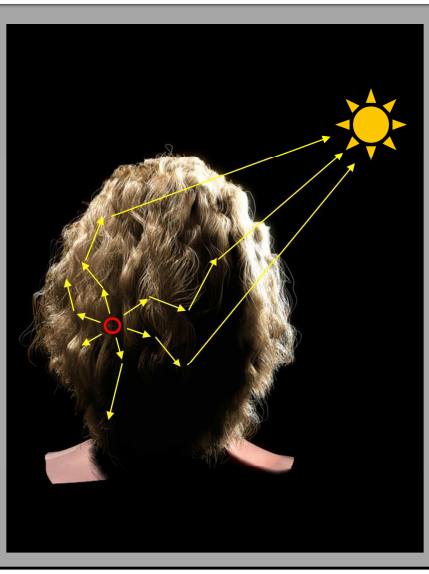
Bounces between

Alternative Methods

- **Path Tracing**

  [Zinke, Weber – VMV 2004]

  ~100 hours



Alternative Methods

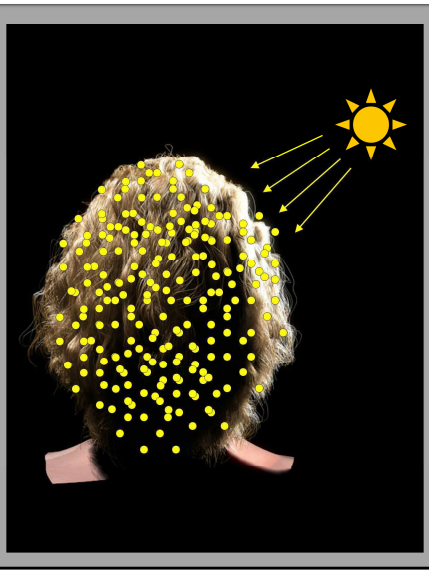
- Path Tracing

  [Zinke, Weber – VMV 2004]

  ~100 hours
- **Photon Mapping**

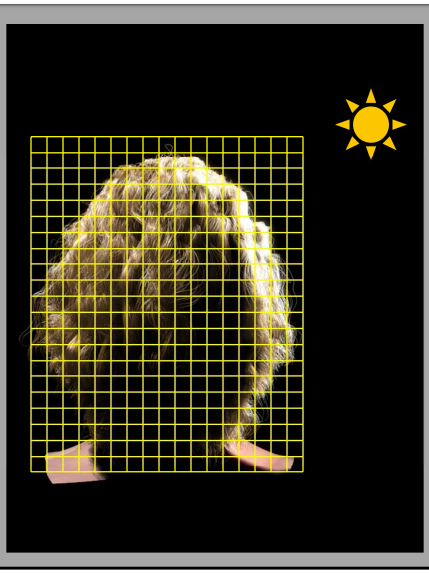
  [Moon, Marschner – SIGGRAPH 2006]

  ~1.5 hours



Alternative Methods

- Path Tracing
  [Zinke, Weber – VMV 2004]
  ~100 hours
- Photon Mapping
  [Moon, Marschner – SIGGRAPH 2006]
  ~1.5 hours
- **Spherical harmonics in a grid**
  [Moon, Walter, Marschner – SIGGRAPH 2008]
  ~30 minutes



Alternative Methods

- Path Tracing

  [Zinke, Weber – VMV 2004]

  ~100 hours
- Photon Mapping

  [Moon, Marschner – SIGGRAPH 2006]

  ~1.5 hours
- Spherical harmonics in a grid

  [Moon, Walter, Marschner – SIGGRAPH 2008]

  ~30 minutes
- **Dual Scattering**

  [Zinke, Yuksel, Weber, Keyser – SIGGRAPH 2008]

  ~20 fps

Examining Scattering in Hair

- Uniformity in hair
- Uniformity in fiber scattering
- Multiple scattering is smooth

Dual Scattering

- Multiple Scattering

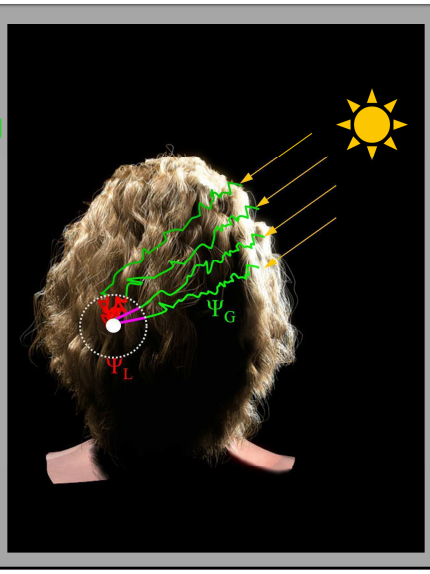
(Ψ_G) Global Multiple Scattering

(Ψ_L) Local Multiple Scattering

$$L_o = \int_\Omega L_i \cdot f_s \cdot \cos\theta \, d\omega_i$$

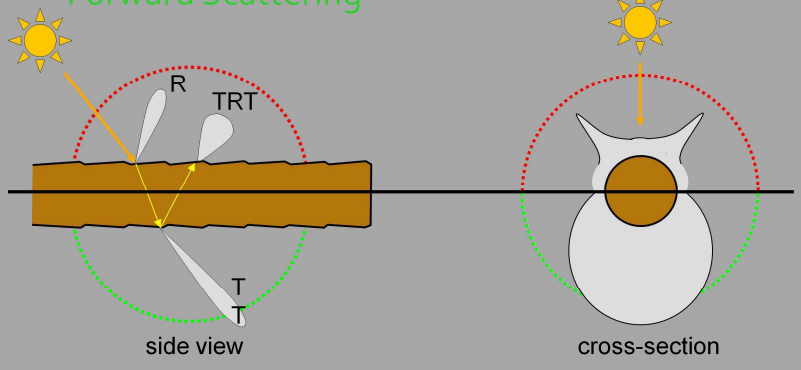
$$L_i = \int_\Omega L_d \cdot \Psi \, d\omega_d$$

$$\Psi = \Psi_G (1 + \Psi_L)$$



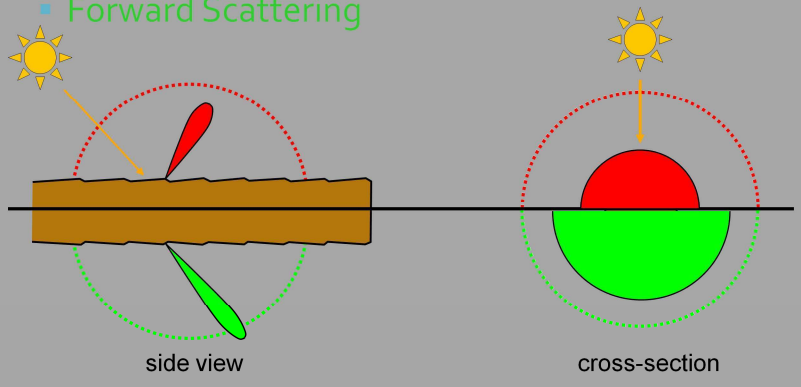
Dual Scattering

- Fiber Scattering
  - Backward Scattering
  - Forward Scattering

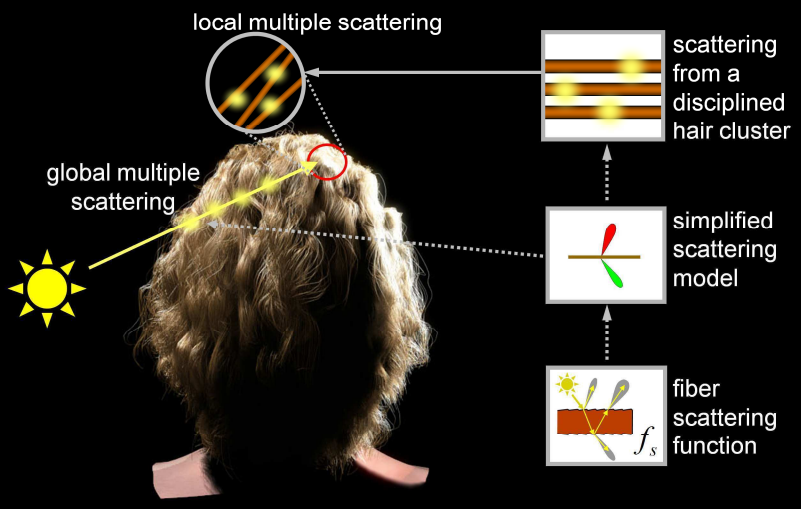


Dual Scattering

- Fiber Scattering
  - Backward Scattering
  - Forward Scattering



Dual Scattering



Dual Scattering

Path Tracing
Reference
22 hours

Dual Scattering
(offline ray shooting)
9.6 minutes

Dual Scattering
(real-time GPU-based)
5.8 fps

Hair Dynamics for Real-time Applications

Contents

- Overview of Hair Simulation Techniques
- Fast Simulation of Hair on the GPU
- Handling Inter Hair Collisions
- Efficient Collision Detection and Handling for interpolated Hair
- Simulating Hair with Hair Meshes

Overview of Hair Simulation Techniques

Overview of Hair Simulation Techniques

- Hair is a complex material, consisting of hundreds of thousands of thin inextensible strands
- Number of different issues to address
  - Dynamics of an individual strand
    - Individual strands have a complex nonlinear behavior
  - Collective dynamics of strands
    - Hair self interaction leads to changes in motion and shape of hair
  - Efficiency
    - Simulating realistic numbers of hair can be quite costly

Overview of Hair Simulation Techniques

- Many different techniques, ranging in complexity, stability and accuracy, for example:
  - Mass spring system [Rosenblum91,Selle09]
  - Rigid body chains [Hadap06]
  - Super helices [Bertails08]

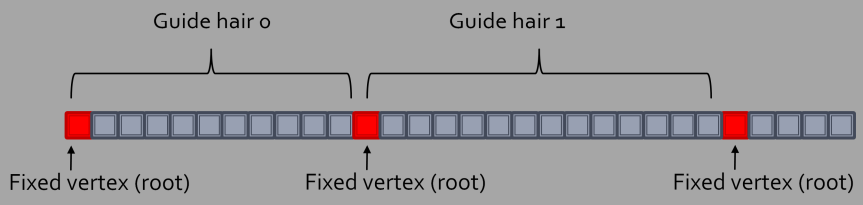
Fast Hair Simulation on the GPU

Simulation of Hair on the GPU

- Hair simulated as a particle constraint system
  - Hair vertices are simulated as particles
  - Links between hair vertices are treated as Distance constraints
    - these constraints maintain hair length
  - Angular forces at each hair vertex maintain hair shape
  - Collision constraints keep hair particles outside obstacles

Representation

- All the guide hair vertices are appended together to form one buffer



Dynamics on the GPU

- Directx11 Class Hardware (for example GTX480, HD 5870)
  - Simulation is done in the compute shader
  - All simulation steps can be done in a single Compute Shader invocation
  - Shared memory is used to transfer intermediate results between vertices
  - Particle positions and attributes are stored in Structured Buffers which can be both read from and written to

Dynamics on the GPU

- Directx10 Class hardware
  - Simulation is done using the Vertex Shader and Stream Output
  - Using Stream Out we can write directly from the Vertex Shader to another Vertex Buffer, skipping rasterization
  - StreamOut prevents reading from and writing to the same buffer, so we have to use two buffers
  - We use the ping-ponging technique – bind VB1 to input and VB2 to output, write to VB1 and then swap the buffers for the next step

Dynamics on the GPU

- Directx9 Class Hardware
  - Use the same approach for ping ponging as for Dx10 class hardware
  - There is no Stream Out so we can use Render To Texture
  - For each iteration we render a quad to the screen, each fragment shader updates one particle, reading the previous particle attribute from one texture and writing the other texture

One simulation step

- Simulate wind force
- Add external forces and integrate
- Repeat for num_iterations
  - Apply distance constraints
  - Apply angular constraints
  - Apply collision constraints
- Apply inter hair collisions

Add forces and integrate

- There are no inter dependencies between particles in this stage, so we can update all particles in parallel
- Each particle is simulated in a separate thread

Add Forces and Integrate

If !isFree(particle) transform particle position by Root transform and return.

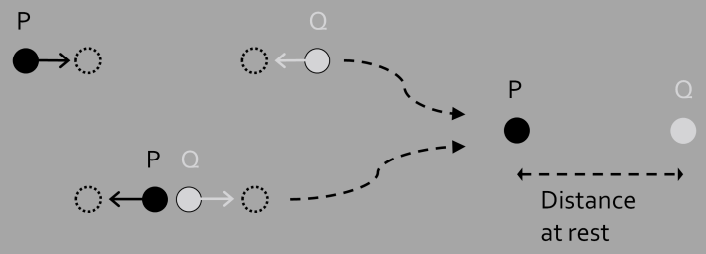
move particle out of any obstacles

accumulate forces (for example from wind, gravity and grooming constraints)

integrate position using verlet integration

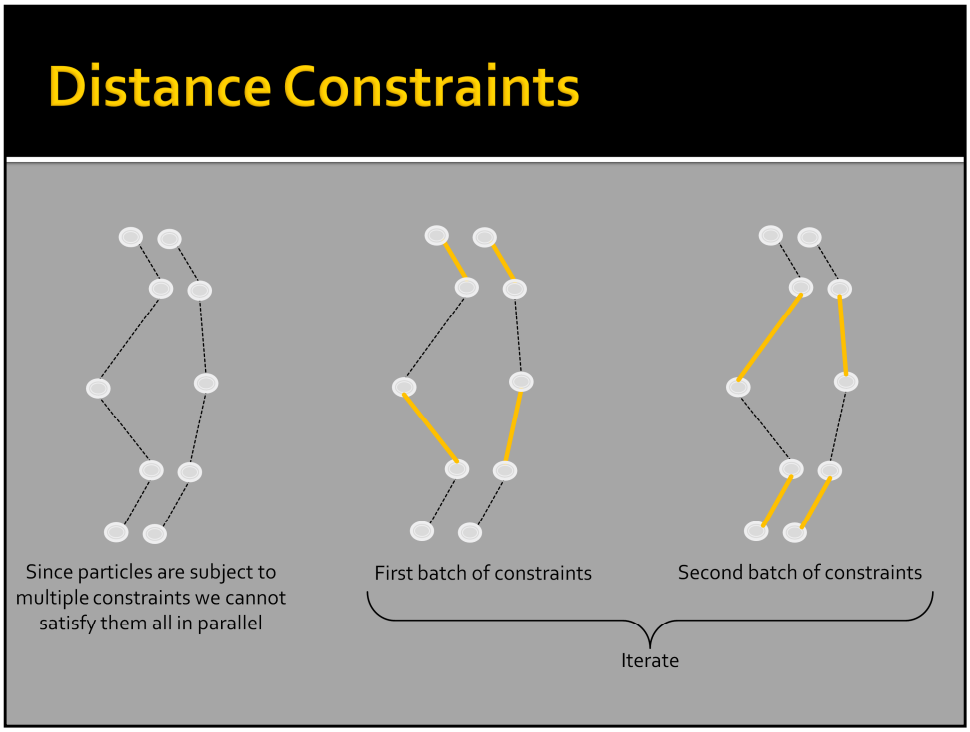
Distance Constraints

- A distance constraint DC(P, Q) between two particles P and Q is enforced by moving them away or towards each other:



Distance Constraints

- To satisfy a distance constraint we need to move the positions of two vertices
- A single particle is subject to two distance constraints (to the particle above and the particle below), thus we cannot update all constraints in parallel
- Instead we batch up the constraints into two non-overlapping sets



Iteratively satisfy distance constraints between vertices

To satisfy one distance constraint we need to modify positions of the two vertices it connects

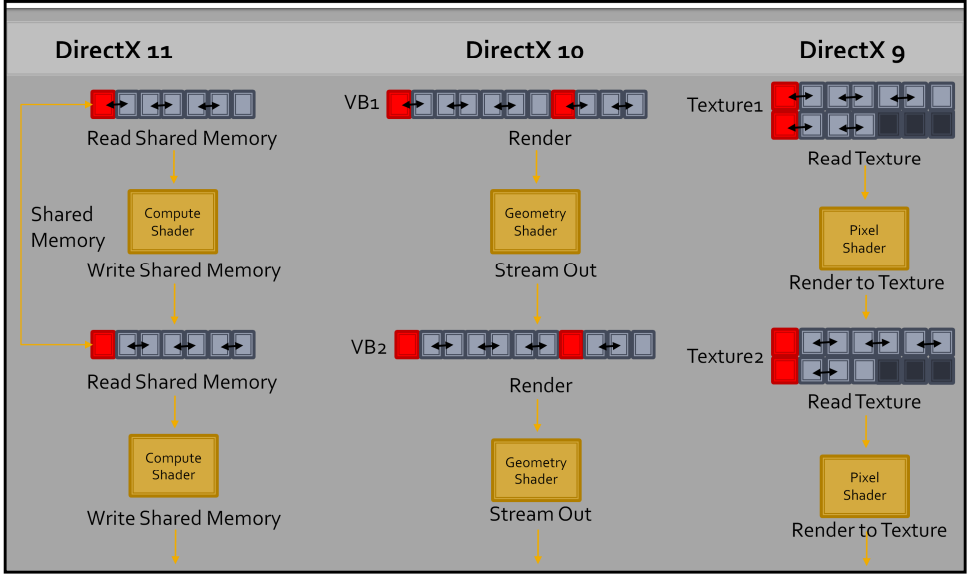
To satisfy many springs in parallel we partition VB into disjoint sets of vertices by using two index buffers

IndexBuffer 1: (0,1) (2,3) (4,5) ….
IndexBuffer 2: (0,0) (1,2) (3,4) ….

One iteration consists of two passes;

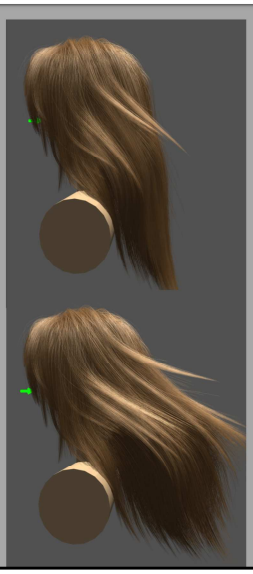
first satisfying spring conditions between pairs of vertices given by IndexBuffer 1
then the same for IndexBuffer2

Distance Constraints



Wind Forces

- Wind forces simulated using semi-lagrangian fluid simulation on a coarse grid (for example 32 cubed in these images)
- Voxelized hair and mesh also added to grid as obstacles to wind
- Similar to [Hadap02]



Simulation using Compute

- For e.g. Dx11 Compute, CUDA, OpenCL
- Using compute provides a number of advantages:
  - Simpler model for abstract computational tasks – don't have to deal with mapping the graphics pipeline
  - Can read from and write to the same buffer, saving memory
  - Can communicate intermediate results between threads using Shared Memory

Hair Simulation Using Compute

- We run one CTA for each hair strand
  - All threads in a CTA work together to update the strand
  - One thread will update a single particle or a single vertex
  - Threads working on the same strand use Shared Memory to communicate intermediate results
    - For example, threads can communicate updated particle positions to each other using shared memory.

Compute Shader Code

```
//buffers that we can write to and read from
RWStructuredBuffer<float4> particlePositions : register(u0);

//declare the shared memory that we will be using
groupshared float4 sharedPos[BLOCK_SIZE];
groupshared float sharedLength[BLOCK_SIZE];

//the compute shader
[numthreads(64,1,1)]
void UpdateParticlesSimulate(uint threadId               : SV_GroupIndex,
                             uint3 groupId               : SV_GroupID,
                             uint3 globalThreadId        : SV_DispatchThreadID)
{
    //calculate the index of where we want to read in the buffer (threadReadIndex), and how long the hair strand is (n)
    …

    //read the attributes into shared memory
    if(threadId < n)
    {
      originalPosition = sharedPos[threadId] = particlePositions[threadReadIndex];
      sharedLength[threadId] = particleDistanceConstraintLengths[threadReadIndex];
      …
    }

    //synchronize after reading the data into shared memory
    GroupMemoryBarrierWithGroupSync();
```

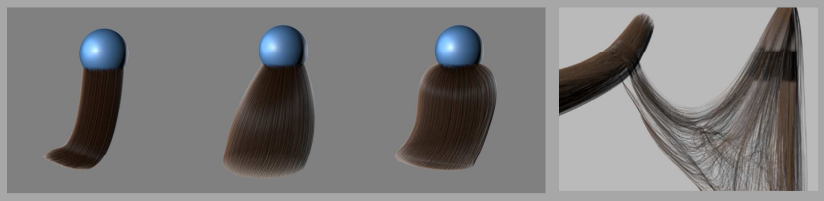
Compute Shader Code

```
    //calculate forces, add them to the particles and integrate
    …

    //iterate through the constraints for a specified number of times
    for(int iteration=0; iteration<g_numConstraintIterations; iteration++)
    {
        //apply distance constraints to first subset
        if(threadId<half)
            DistanceConstraint(sharedPos[threadId*2],sharedPos[threadId*2+1],sharedLength[threadId*2].x);

        GroupMemoryBarrierWithGroupSync();

        //apply distance constraints to second subset
        if(threadId<half2)
            DistanceConstraint(sharedPos[threadId*2+1],sharedPos[threadId*2+2],sharedLength[threadId*2+1].x);

        GroupMemoryBarrierWithGroupSync();

        //apply angular constraints
        //apply collision constraints
    }

    //and finally write back the data to the global buffer
    if(threadId < n)
    {
        particlePositions[threadReadIndex] = sharedPos[threadId];
        …
    }
}
```

Handling Inter Hair Collisions

Inter -Hair collisions

- Collision between hair strands is very important:
  - Gives shape to the hair, causing it to occupy a larger volume (especially frizzy hair)
  - Changes the motion of a full head of hair



Hair self collision helps preserve volume and leads to differences in the look and movement of strands (Images from McAdams 09)

Issues

- The large number of hair strands makes processing hair self-interactions very challenging
- Can often have issues with instability and jittering, which is more obvious when hair is at rest

Approaches to handing collisions

- Explicit collision detection and resolution of each hair
  - Slow and prone to jitter
- Wisp model - reduce the amount of interaction by only doing collisions at the wisp
  - level Layered Wisp Model [Plante01], [Choe05]

Approaches to handing collisions

- Continuum framework
  - Assume hair is a fluid medium and use fluid simulation techniques to handle self collisions efficiently
- Some approaches
  - Hadap01 Lagrangian
  - Bertails05 Eulerian
  - McAdams09 Hybrid

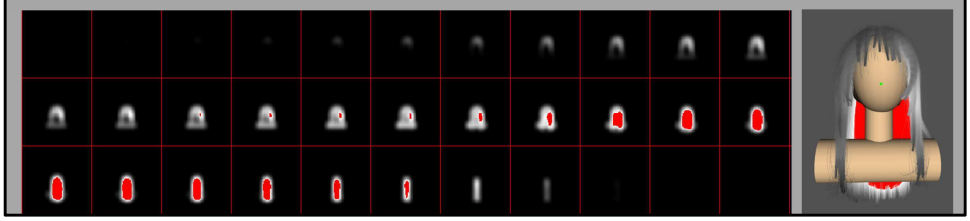
McAdams 09

Inter-hair collisions

- We will outline a simple approach here which tries to achieve volume preserving quality of inter-hair collisions. (It is based on Bertails05 and can be extended to McAdams09)
  - Inter-hair collisions dealt with in grid based framework
  - Hair strands and obstacles (like head/body) are voxelized into a low resolution grid
  - Hair vertices are pushed out of high density areas

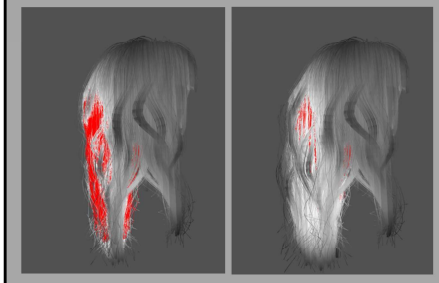
Detecting areas and avoiding collisions

- Force is applied to each vertex in a high density region
- Force is applied in the direction of the negative gradient of the density
  - blur the voxelized density, then for each vertex falling in a high density area find the gradient of the density field at that point



Results of applying Inter-Hair Collision Forces

Visualizing density

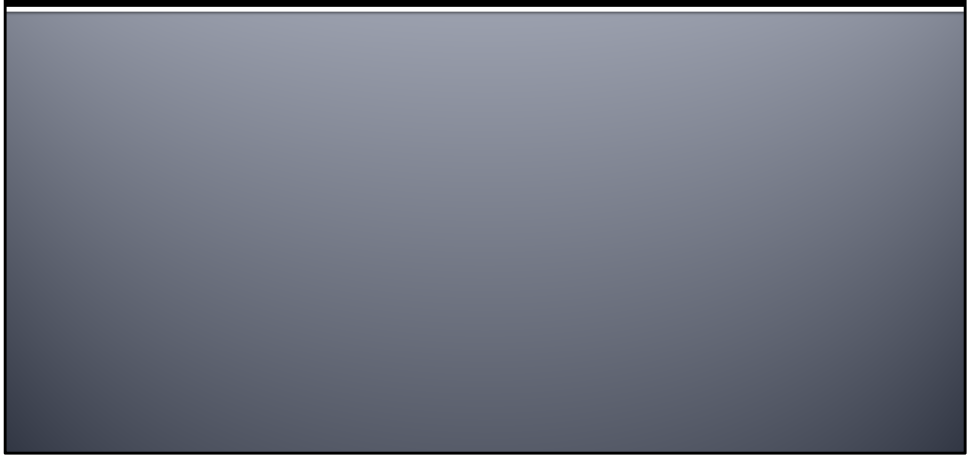


Before After

Final Rendering

Before After

Efficient Collision Detection and Handling for interpolated Hair



Avoiding interpolated hair collisions

- While rendering we
- Interpolating between multiple guide strands can lead to some hair going through collision obstacles

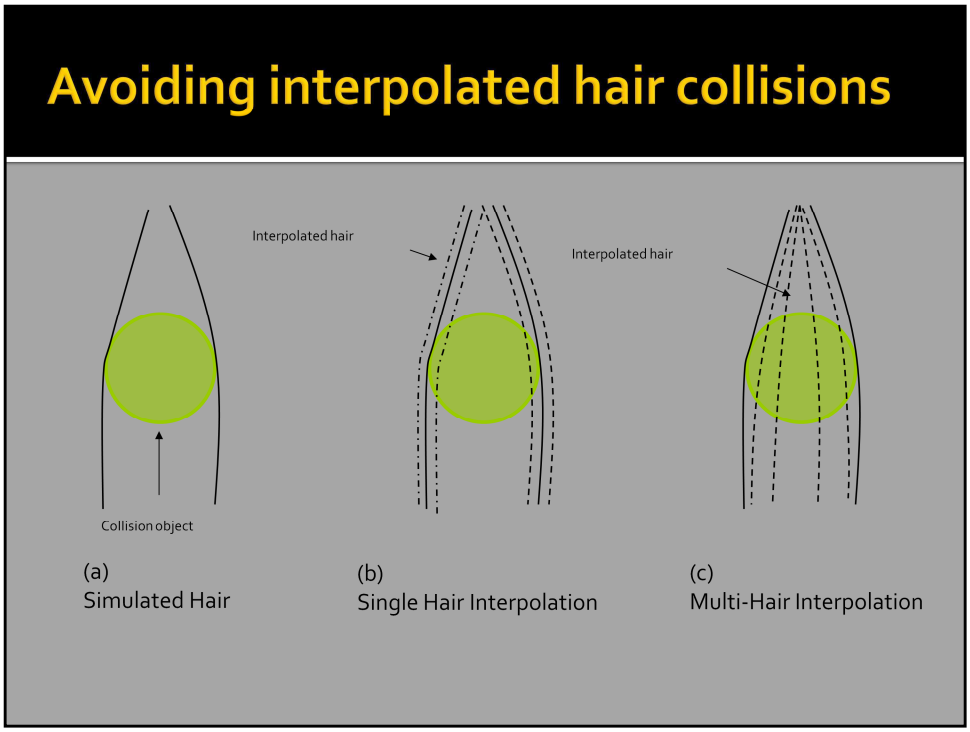
Interpolated hair intersect collision volumes

Fixing collisions without doing extra simulation

As discussed previously, there are different methods available for interpolating hair each with its own advantages. Interpolated hairs can be created along the length of a single guide hair (clump based interpolation), or they can be created by combining multiple guide hair (for example barycentric interpolation between three guide hair).

The advantage of using the latter approach is that it provides a good coverage of the scalp and hair volume with relatively few hairs. Unfortunately, one of the drawbacks of this approach is that it is likely to produce interpolated hairs that penetrate the body or other collision objects. This is demonstrated in the top figure, where interpolated strands are going through the collision obstacles of the head and body.

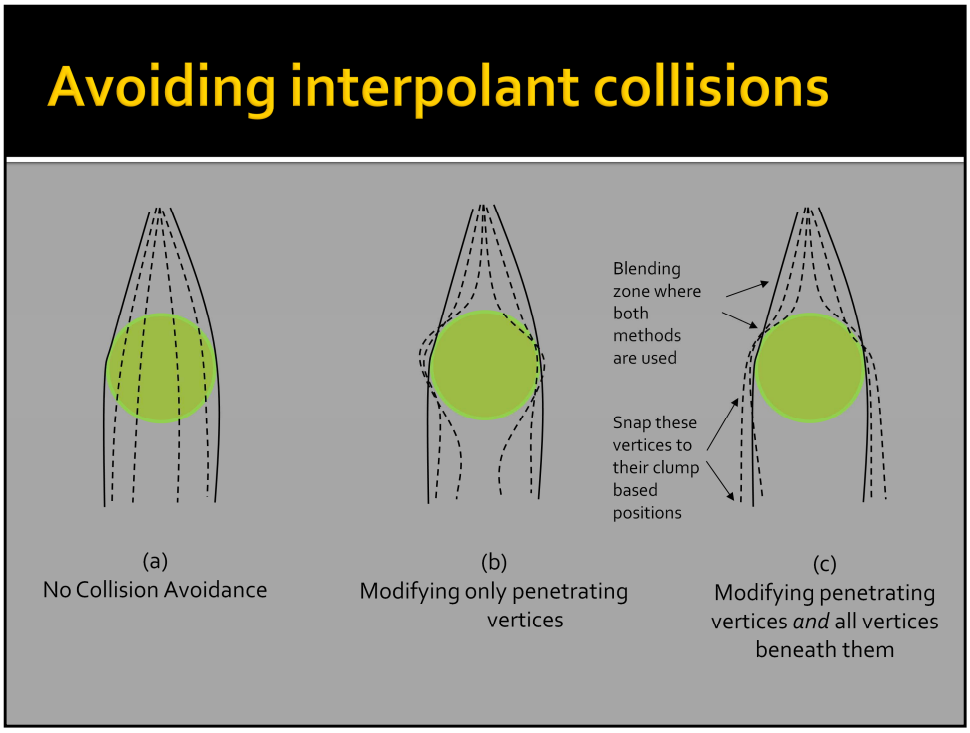
In this section we describe a method for efficiently detecting and avoiding cases where multi-guide hair interpolation leads to hair penetration into objects.



As we see in Figure(a) the guide hairs avoid the collision obstacle since they are explicitly simulated. The interpolated hairs however are only produced using the positions of the guide hair and have no physical simulation.

In Figure(b) we see the results of interpolation using only a single guide hair to create a given interpolated hair. The interpolated hairs penetrate the collision object slightly, but largely follow their guide hair.

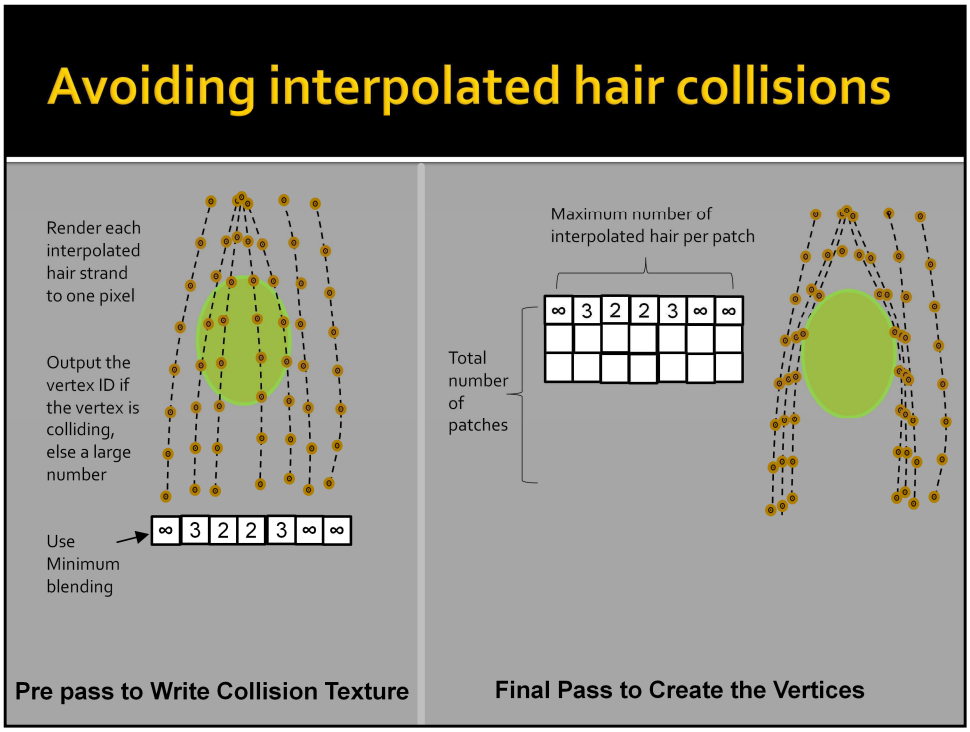
In Figure(c) however we see much worse penetration of interpolated hair even though the guide hairs are not penetrating the object. This is because the interpolated hairs are created by averaging the positions of many guide hair and this process gives us no guarantees that the interpolated positions will not go straight through the collision object (for example the head).



In order to avoid these collisions we first detect the vertices where collisions occur and then switch those vertices to a more conservative interpolation approach, like the single strand interpolation.

It is important to note that it is not sufficient to address only those vertices in the interpolated hair strands that actually undergo a penetration; altering their positions necessitates that we alter the positions of all vertices beneath them (and some above them) as well.

This is demonstrated in figures above. If, as in figure(b), we only change the interpolation method of those vertices directly undergoing a collision the remaining hair strand *below* the modified vertices looks un-natural in its original position. Instead, we need to modify the interpolation mode of all vertices that are undergoing a collision or are below such vertices, as in Figure(c).



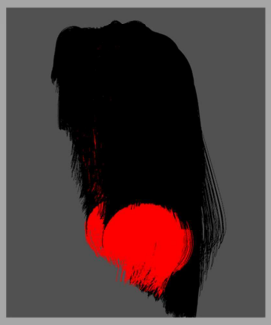
In order to identify hair vertices that are below other object-penetrating vertices we do a pre-pass. In this pre-pass we render all the interpolated hair to a texture; all vertices of an interpolated hair strand are rendered to the same pixel. For each hair vertex we output its ID (number of vertices that separate the current vertex from the hair root) *if* that vertex collides with a collision object. Otherwise we output a large constant.

This rendering pass is performed with minimum blending. The result of the pass is a texture that encodes for each interpolated hair strand whether any of its vertices intersect a collision object, and if they do, what is the first vertex that does so.

We can then use this texture to switch the interpolation mode of any vertex of an intersecting hair strand below the first intersecting vertex, as in the figure on the left.

Avoiding interpolated hair collisions

The original interpolated hair.



The original hair, annotated with red to indicate the vertices which are intersecting

The modified hair; all strands with red vertices are switched over to single-guide interpolation.

In the Figures above we can see the result of this step visualized on the hair. Here we render each vertex as red if its ID is greater than or equal to the minimum ID that we wrote out in the pre-pass.

We can then use our pre-calcuated collision texture to correctly switch the interpolation mode of interpolated hair when we are creating them. In the shader where we calculate the interpolated hair position we read the texture to determine if the current vertex is above or below the first intersecting vertex of the strand.

If the current vertex is below the first intersecting vertex we use the single strand interpolation method to calculate its position. We also employ a blending zone of several vertices above the first intersecting vertex to slowly blend between the two interpolation modes hence avoiding a sharp transition

Hair Meshes

Cem Yuksel*
Cyber Radiance

Scott Schaefer†
Texas A&M University

John Keyser‡
Texas A&M University

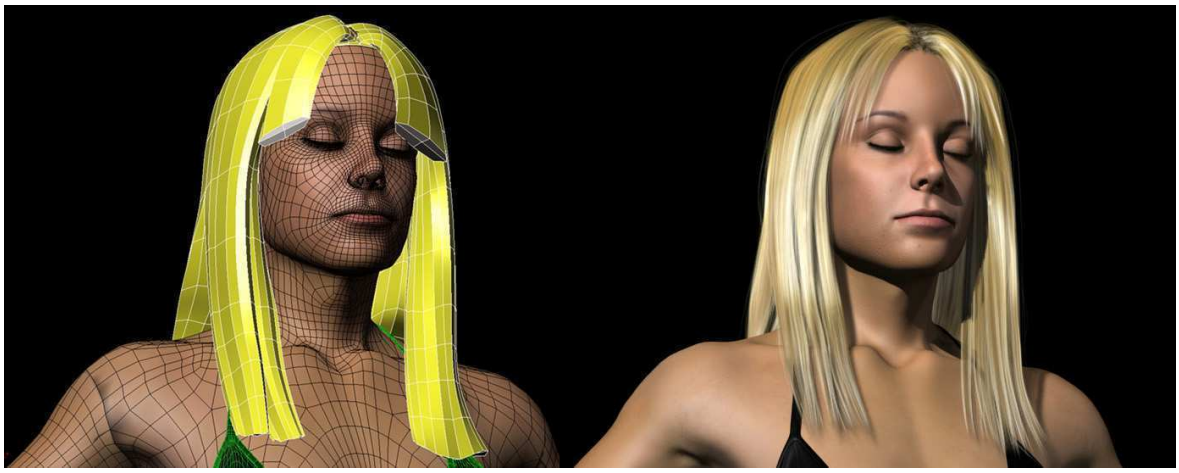


Figure 1: *An example hair mesh model and the final hair model generated using this hair mesh and procedural styling operations.*

Abstract

Despite the visual importance of hair and the attention paid to hair modeling in the graphics research, modeling realistic hair still remains a very challenging task that can be performed by very few artists. In this paper we present *hair meshes*, a new method for modeling hair that aims to bring hair modeling as close as possible to modeling polygonal surfaces. This new approach provides artists with direct control of the overall shape of the hair, giving them the ability to model the exact hair shape they desire. We use the hair mesh structure for modeling the hair volume with topological constraints that allow us to automatically and uniquely trace the path of individual hair strands through this volume. We also define a set of topological operations for creating hair meshes that maintain these constraints. Furthermore, we provide a method for hiding the volumetric structure of the hair mesh from the end user, thus allowing artists to concentrate on manipulating the outer surface of the hair as a polygonal surface. We explain and show examples of how hair meshes can be used to generate individual hair strands for a wide variety of realistic hair styles.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations;

Keywords: Hair modeling, hair mesh, volume modeling

*e-mail: cem@cemyuksel.com
†e-mail: schaefer@cse.tamu.edu
‡e-mail: keyser@cse.tamu.edu

1 Introduction

Hair is an extremely important visual component of virtual characters. Therefore, it is crucial to equip artists with powerful tools that can help them sculpt the exact hair model they desire. Unfortunately, realistic hair models may require hundreds of thousands of hair strands, formed into exceptionally complicated geometric structures. The characteristics of individual hairs, the styling products applied to hairs, and the physical forces affecting the hairs all impact the overall look. Hence, a hair modeling tool should be powerful enough to handle a wide range of hair styles, simple enough that the tremendous complexity of the model is hidden from the user, and controllable such that artists can easily express their desired outcome.

Despite a considerable amount of research and a variety of implementations over the past two decades, hair modeling still remains an open challenge; there is no solution that is widely accepted in the graphics industry. To reduce the complexity of hair modeling, almost all existing approaches generate fine details of the hair model through procedural techniques. These procedural tools relieve the burden of dealing with every individual hair strand, allowing artists to concentrate on the overall look of the hair model. Thus, the main modeling effort on the part of the artist is in defining the global shape of the hair.

Even though hair is made up of many thin strands, we often interpret hair models as a surface. Therefore, the shape of this outer surface is important when modeling a particular hair style. Existing hair modeling approaches either define the shape of the hair model indirectly through various parameters or concentrate on the shapes of individual hair strands or bundles. In either case, the outer surface of the hair model is not explicitly defined. For this reason many skilled artists first model the outer surface with standard surface modeling tools and then use this surface as a guide while modeling hairs, attempting to place hairs in positions that will match that surface. This indirect control can be rather time consuming, especially when an artist desires to change the shape of the hair surface later.

In this paper we present a new alternative, *hair meshes*, that aims to bring hair modeling as close as possible to modeling polygonal meshes. Figure 1 shows an example hair mesh and the hair model

created using this hair mesh. A hair mesh represents the entire volume of hair with topological constraints that allow us to easily trace the path of hairs from the scalp through the volume. The user has the ability to explicitly control the topological connections within the hair mesh, thus allowing creation of a wide range of possible hair models. A user will typically only interact with the external surface of the mesh volume, using this surface to explicitly control the shape of the hair. Internal vertices of the mesh volume are automatically placed based on the external surface. Since internal vertices do not need to be manipulated directly, artists who are already skilled at polygonal surface modeling can easily model hair with the flexibility and direct control of polygonal structures.

2 Related Work

There is a large body of previous research on virtual hair. In this section we briefly overview most related methods. We recommend the reader refer to Ward et al. [2007a] for a recent, extensive survey of hair methods in Computer Graphics.

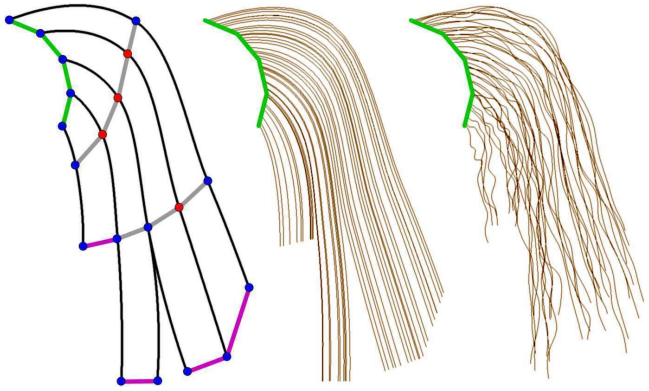
The high geometric complexity of hair and the wide variety of real-world hair styles make hair modeling a challenging task. Therefore, most hair modeling techniques are based on controlling collections of hair strands at once. Perhaps the simplest approach to hair modeling is representing hairs as parametric surfaces (e.g. NURBS) called *strips* [Koh and Huang 2001; Liang and Huang 2003; Noble and Tang 2004]. Using texture mapping with alpha channels, these surfaces look like a flat group of hair strands. Even though these techniques can be improved by adding thickness to this surface [Kim and Neumann 2000], they are very limited in terms of the models these methods can represent and are not suitable for realistic hair modeling.

A common hair modeling technique is to use *wisps* or *generalized cylinders* to control mostly cylindrical bundles of hairs with 3D curves [Chen et al. 1999; Yang et al. 2000; Xu and Yang 2001]. While these approaches are especially good at modeling hair styles with well defined clusters, it is often difficult and time consuming to shape a collection of wisp curves. Even making simple changes to an existing hair model can be exhausting depending on the number of curves to be edited. Multi-resolution approaches [Kim and Neumann 2002; Wang and Yang 2004] can improve the modeling process, yet this improvement depends on the complexity of the hair style and how close the desired hairstyle is to the types supported in the system.

Researchers have also tried using different physically-based techniques to shape hair strands. Anjyo et al. [1992] simulated the effect of gravity to find the rest poses of hair strands. Hadap and Magnenat-Thalmann [2000] modeled hairs as streamlines from a fluid dynamics simulation around the head. Yu [2001] used 3D vector fields to shape hairs by placing vector field primitives, and Choe and Ko [2005] applied vector fields with constraints to shape wisps. While various hair types can be modeled with these approaches, just like other simulation methods, they can be difficult to control in a precise manner.

Capturing a hair model from images [Kong et al. 1997; Grabli et al. 2002; Paris et al. 2004; Wei et al. 2005] is another alternative used to automate the virtual hair modeling process. Even though the recent methods [Paris et al. 2008] are very promising in terms of the visual realism of the results, these methods do not incorporate any artistic control.

Sketch based interfaces are also used for modeling hair [Malik 2005], both for cartoon hairstyles [Mao et al. 2005] and more realistic models [Wither et al. 2007]. Recently, Fu et al. [Fu et al. 2007] proposed a sketch based interface to build a vector field, which is then used to generate individual hair strands. While these techniques are practical for quickly generating a hairstyle, they are very difficult to control for achieving a desired outcome precisely.



(a) Hair Mesh (b) Generated Hairs (c) Styled Hairs

Figure 2: *(a) 2D representation of a hair mesh, (b) hairs generated from this hair mesh, and (c) hairs after procedural styling operations. The green lines correspond to faces of the root layer, the tip layer is colored purple. Blue hair mesh vertices are external, and red are internal.*

Other hair modeling approaches include explicitly modeling individual hair strands [Daldegan et al. 1993; Lee and Ko 2001] or a number of guide hairs [Alter 2004]. An interesting recent approach aims to simulate a real-world hair dressing session using haptic controls and physically-based simulation [Ward et al. 2007b]. Recently, Wang et al. [2009] proposed a method for generating a new hair model based on a given hair model.

3 Hair Mesh Modeling

In our approach, hair modeling begins by defining the outer surface of the hair model. This outer surface is used to create a meshed volume, the *hair mesh* (see Figure 2a). The hair mesh structure is then used to generate individual hair strands (Figure 2b). As in most existing techniques, fine details of the hair strands are subsequently defined through procedural styling operations (Figure 2c), which we will discuss at the end of this section.

3.1 The Hair Mesh Structure

A hair mesh is a 3D mesh describing the volume of space enclosing the hair. It consists of layers of polygonal meshes, which typically contain quadrilateral or triangular faces, but we place no restrictions on the types of polygons used. Let F^k be a set of faces for layer k. We refer to the faces in F^0 as the *root layer* of the hair mesh (highlighted green in Figure 2a) and the polygons in this layer exactly match the scalp model. This is the surface we will grow hairs from and each face in F^0 will correspond to a bundle of hairs.

To create a path for each hair, we place a number of additional layers on top of the root layer, such that each face F_j^k at layer k has a one-to-one correspondence to a face F_j^{k+1} at the next layer. Connecting the two corresponding faces F_j^k and F_j^{k+1}, we form a prism such that these faces are the two base faces of the prism. We refer to the collection of such prisms starting at F_j^0 and connecting to faces F_j^k (where $k \geq 0$) as a *bundle*, F_j. If for any face F_j^k the corresponding face F_j^{k+1} of the next layer does not exist, hair strands of this bundle terminate at layer k. We refer to the termination layer of bundle F_j as n_j and call the surface composed of all of the $F_j^{n_j}$ the

Figure 3: *A hair knot model and its hair mesh. The explicit control of the hair shape provided by hair meshes makes modeling such hairstyles as easy as modeling any other surface.*

tip layer, which is highlighted in purple in Figure 2a. Even though there is a one-to-one mapping between faces at different layers, the faces adjacent to a given face may change from one level to the next (e.g. the mesh can split, separating bundles as in Figure 2a).

Given this correspondence between layers, we can create a path for each hair from the root layer. For each point on the root layer, we compute the barycentric coordinates of that point with respect to the face F_j^0 containing it using mean value coordinates [Floater 2003]. We then trace the path of a hair growing from that point through the volume by applying these barycentric weights at every corresponding face in the bundle up to the tip layer. Finally, we connect these points together using C^1 Catmull-Rom splines [1974] to generate the final hair.

In the simplest case, all layers of a hair mesh have exactly the same topology. However, as illustrated earlier, this is not a requirement. The mesh is valid as long as each face has a corresponding face on all subsequent layers. Therefore, a *vertex* can be connected to one vertex (if the topology is locally the same) or to multiple vertices (if the topology changes) on a neighboring layer.

To keep the hair mesh structure simple, we permit only a one-to-many mapping of *vertices* from one layer to the next layer (note: faces are always a one-to-one mapping), though a many-to-many mapping of vertices can potentially generate a valid hair mesh. This simple restriction ensures many useful properties, such as edges of faces at one layer cannot collapse at the next layer, two faces at any layer can be neighbors (i.e. share a vertex) only if they are neighbors at the root layer, and the genus of the root layer is the same as that of the set of bundles.

3.2 Topological Operations

Users must be given controls that provide a wide variety of modeling operations, but at the same time these modeling operations must preserve the topological constraints on the mesh. The input to our system is a polygonal object that we would like to grow hairs on. This object forms the root layer F^0 of our hair mesh. In the beginning (when the hair mesh has no layers), the root and the tip layers coincide (i.e. $n_j = 0 \; \forall j$).

A user interacting with the mesh will typically model the hair by "growing" the layers out from the root layer, specifying geometric and topological changes in each layer. To perform this modeling, the following operations are supported for creating and modifying the hair meshes:

Face Extrude: This is our primary operation to create new layers. Face extrusions are only permitted from the current tip layer faces, as the extrusions of side faces would not generate valid hair meshes. For each face $F_j^{n_j}$ to be extruded, a new face $F_j^{n_j+1}$ is created, thereby generating a new prism in the hair mesh. This is typically the very first operation we use to extend the root layer.

Face Delete: This operation deletes the face at the tip layer of a bundle, thereby removing the last prism. When the root and the tip layers coincide ($n_j = 0$), a face delete operation is equivalent to deleting a particular face from the root (and thus no hairs will be created for that region).

Layer Insert: We use this operation to create new layers in between two intermediate layers. Though layer insert could be defined as a local operation, to enforce one-to-many mapping of vertices we perform the same operation on all faces *that are topologically connected* in the layer at which the operation is applied. The new layer is inserted before the layer that is selected.

Layer Remove: Similar to layer insertion, layer removal affects all the topologically connected faces of a layer. When the layer to be removed is the tip layer, this operation is identical to face delete(s). The root layer cannot be removed as it would mean deleting the root object.

Edge and Vertex Separate: Vertices and edges shared by more than one face in a single layer can be topologically separated. This topological separation creates multiple edges/vertices that are topologically separated but are geometrically coincident. Subsequent modeling operations may move these points geometrically. If the separated vertex or edge is not at the tip layer, all corresponding vertices or edges above this layer are also separated to ensure one-to-many mapping of vertices.

Edge and Vertex Weld: The weld operation is the inverse of a separate operation, and topologically joins the vertices or edges at the same layer. To respect one-to-many mapping of vertices, this operation can only weld vertices that correspond to the same vertex at the root layer. Furthermore, all corresponding vertices below this layer are also welded.

Face and Edge Divide and Subdivision: Splitting and subdivision of faces can be easily defined over the hair mesh. This includes standard approaches such as Catmull-Clark or Loop subdivision. Any subdivision operation applied to a face must be propagated throughout the entire bundle for that face. In addition, since subdivision may modify adjacent faces, all bundles adjacent to that bundle in the root layer may also be affected. Note that subdivision is supported only on the layers of the hair mesh, not on the quadrilateral faces that form the sides of the prisms in the hair mesh surface (layer insertion provides a similar effect, there).

3.3 Geometrical Operations

The vertices of the hair mesh are described as either external vertices, which lie on the outer surface of the mesh, or internal vertices. This classification is illustrated in Figure 2a, where external vertices are drawn in blue, and internal vertices in red. Note that several

Figure 4: *A simple hair bun modeled using hair meshes.*

topological operations can generate new vertices (both external and internal), or convert vertices between external and internal vertices.

In general, the user can explicitly position all these vertices. However, a large number of internal vertices may be generated during construction of the hair mesh. Since the number of external vertices is proportional to surface area and the interior to volume, the number of internal vertices may dominate the total number of vertices in complex hair models. These vertices are necessary to determine the path of a hair and provide adjacency information for hair bundles. However, these internal vertices are problematic for the user because they lie inside the enclosed volume of the hair mesh, making them hard to see, especially when the hair mesh is visualized as a surface. Therefore, we provide the option to hide these internal vertices and instead place them automatically based on the positions of the external vertices.

Internal vertex placement is a part of the modeling process and is executed every time the user moves or creates a group of external vertices. Some of these operations (moving external vertices) can affect a large number of internal vertices. For this reason, it is critical that the internal vertex placement algorithm be performed very quickly, so as not to interrupt the modeling process.

While many techniques may be used to place internal vertices, we choose a simple constrained quadratic minimization. We will define an error metric in terms of the positions of all mesh vertices, fix the external vertices, and solve for the positions of internal vertices that minimize this metric. Our choice of error metric is motivated by physical properties of hair. Namely, hair strands should have a similar shape as nearby hair strands (later we will apply styling operations to differentiate nearby hairs as explained in Section 3.4).

For each extruded prism between faces F_j^k and F_j^{k+1}, we can approximate the hair direction locally using the edges of the prism in the extrusion direction. Let $V_{j,i}^k$ be the i^{th} vertex of the face F_j^k. Then, an edge of the prism is given by the vertices $V_{j,i}^k$ and $V_{j,i}^{k+1}$, and the hair direction locally along the edge is $V_{j,i}^{k+1} - V_{j,i}^k$. We want to minimize the difference in the local hair direction between adjacent edges along the extrusion direction of the prism.

If we sum over all of the quad faces that form the sides of the prisms in the volume, the resulting minimization is of the form

$$\min \sum_j \sum_{k=1}^{n_j} \sum_i \left\| (V_{j,i}^{k+1} - V_{j,i}^k) - (V_{j,i+1}^{k+1} - V_{j,i+1}^k) \right\|^2 ,$$

subject to the constraint that the external vertices (or any others the user wishes to fix) remain unchanged. We can easily minimize this quadratic, which has a unique minimum, using Conjugate Gradients [Shewchuk 1994]. Since we use the current positions of the internal vertices as a starting point for this minimization, Conjugate Gradients typically converges to a solution in only a few iterations and is quite fast. The red vertices in Figure 2a show the result of a 2D version of this optimization with the blue, external vertices as constraints.

Figure 5: *A futuristic hair bun model and its hair mesh.*

3.4 Hair Styling

Hair mesh modeling can be thought of as an initial stage of modeling hair. The hair mesh defines the overall shape of the hair model and the hair strands we generate conform to that model. However, realistic hair is not always straight and many existing hair modeling techniques can be applied to the hair strands to improve the realism of the hair or reproduce specific hair styles. In our system, all hair modeling operations applied to the hair strands after they are generated from the hair mesh are called *styling* operations.

Procedural hair styling forms one group of such operations. These operations typically deform the hair by moving the vertices of the hair strands using a combination of procedural noise and trigonometric functions with various parameters. The functions can be directly computed using the 3D position of each hair strand vertex. However, this makes the hair style, which is applied using these procedural operations, very sensitive to the initial 3D positions of hair strand vertices. As a result, even minor modifications to the hair mesh may significantly alter the shapes of some hair strands. To avoid this undesired behavior, one can define these procedural operations in the canonical space of a hair strand as in [Yu 2001] or using the barycentric embedding of a hair strand within the hair mesh. In our system we use the later approach.

In addition to procedural operations, we can easily combine our hair modeling system with some previous hair modeling techniques that use wisps. We achieve this by generating wisp curves from the hair mesh similar to generating hair strand curves. In this case, individual hair strands are not directly generated from the hair mesh, but the wisp curves along with a number of parameters are used to populate final hair strands. Note that wisp curves themselves can go through procedural styling operations or explicit user modifications before generating the hair strands. Similarly, we can combine our hair mesh modeling approach with multiresolution hair modeling [Kim and Neumann 2002] by generating first level generalized cylinders using the hair mesh. Higher level generalized cylinders and finally individual hair strands are then generated from the first level generalized cylinders as described in [Kim and Neumann 2002]. This approach replaces the most laborious stage of multiresolution hair modeling (as stated by Kim and Neumann [2002]) with hair mesh modeling.

4 Results and Discussion

To demonstrate the capabilities of our hair mesh modeling approach we present various hair models produced using our system. Figure 1 shows a typical hair model with its hair mesh. While similar hair models can be prepared with many previous techniques, the main advantage of the hair mesh is the ability to control the hair shape by directly manipulating the outer surface.

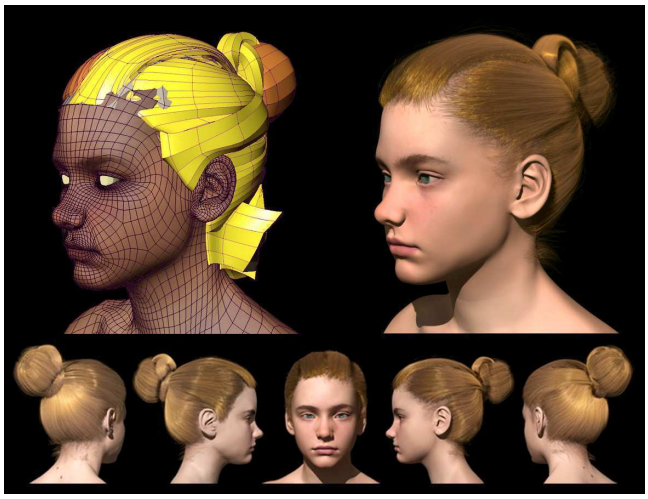


Figure 6: *A complicated hair bun model and its hair mesh.*

Figures 3, 4, and 5 show different hair models with buns and knots. Such models are very difficult to prepare with most previous techniques, but with hair meshes, modeling these hairstyles are no more difficult than modeling the outer surface using any standard polygonal modeling tool. Figure 6 shows a more complicated bun model. Notice the fine detail of the bun and the explicit control of the hair shape and direction available to the artist.

Depending on the complexity of the desired hair model, modeling using hair meshes can take as short a time as a couple of minutes. For example, the simple hair model shown in Figure 7 can be prepared in a couple of minutes. While preparing a more complicated hair model can take significantly longer, the explicit control provided by hair meshes makes it easy to edit the model and produce the desired variation.

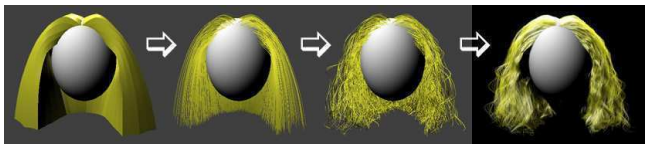


Figure 7: *A simple hair model prepared within a couple of minutes.*

Even though hair meshes are designed to model hair by letting the user specify only the outer surface of the hair, they are also useful when an artist desires to control the hair shape within the hair volume. Figure 8 shows such a complicated hair mesh model where the artist explicitly shapes each hair bundle.

Figure 9 shows an example of combining hair mesh modeling with wisp based hair modeling. Here the hair mesh in Figure 1 is used to generate 200 wisp curves, instead of individual hair strands. Final hair strands are then generated from these wisp curves as in Choe and Ko [2005].

One useful property of our hair mesh modeling approach is the complete separation of large and small scale details. While large scale details that define the global shape of the hair model are controlled using the hair mesh, fine details are introduced during the styling process. Therefore, the same hair mesh can be used to generate different types of hair styles as shown in Figure 10. This technique can also be used for introducing significant style variations within a hair model by generating one set of hair strands from a hair mesh and applying one style, then using the same hair mesh to generate another set of hair strands with a different style applied. By generating multiple sets of hairs with this procedure, style

Figure 8: *A complicated hair mesh model and the hair generated from this hair mesh.*

variations of a hair model can be easily represented. The union of these sets form the final hair model, and the global shape of the hair model is explicitly controlled by a single hair mesh. This feature is especially useful when modeling realistic hairs with rich variations such as frizzy hair and fly-aways. Figure 11 shows such an example. Note that in many previous hair modeling techniques, introducing these frizzy strands can be difficult or even impossible.

We designed the hair mesh modeling approach such that styling operations are reserved for small scale details only and larger details are explicitly modeled using the hair mesh. However, in our system there is no restriction on the user side to forbid using styling operations for large variations as well. When the style variations are exaggerated, the perceived surface of hair formed by the final hair strands can deviate from the surface defined by the hair mesh. This deviation is especially undesirable when the hair mesh is used for explicitly avoiding intersections of hairs with surrounding objects. Note that undesired intersections can also be automatically avoided at the hair strand level using the technique described by Kim and Neumann [2002].

Hair mesh modeling merely provides a high level structure to easily define the global shape of a hair model. Unfortunately, it is not possible to claim for any hair modeling technique that it can produce hair models that cannot be modeled using previous techniques, since theoretically speaking all hair models can be produced by ex-

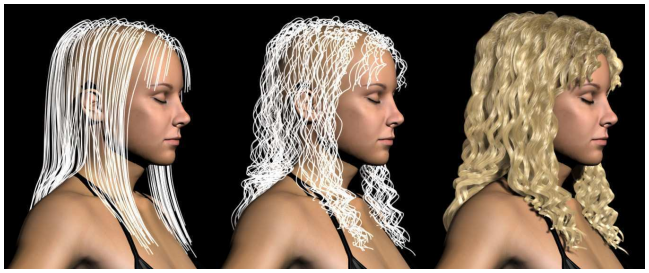


Figure 9: *(Left) wisp curves generated from a hair mesh, (middle) wisp curves after styling operations, and (right) final hair strands generated from these wisp curves.*

Figure 10: *A complicated hair mesh model and two different hairstyles generated from the same hair mesh via different styling operations.*

plicitly modeling the hair strands. Moreover, the real power of hair meshes is not the fact that various different hair models can be prepared with this approach, either. Most existing techniques permit a wide variety of hair styles to be generated. However, the lack of explicit control over the global hair shape makes the existing techniques difficult, if not impossible, to use to achieve the exact hair model one aims for. On the other hand, hair meshes convert the volumetric hair modeling problem to a surface modeling problem. This significantly reduces the high complexity of volume modeling and brings hair modeling closer to standard polygonal surface modeling. As a result, hair meshes offer a familiar interface to experienced modelers and make it very easy for them to sculpt the exact hair models they desire.

We have also tried using hair meshes for simulating hair. We follow an approach similar to that introduced by Chang et al. [2002]. Instead of picking representative hair strands (i.e. guide hairs), we form an articulated rigid body chain directly from the edges of the hair mesh that connect the vertices of one layer F^k to the next layer F^{k+1}. Figure 12 shows example frames captured from our hair mesh simulation system. We observed that physical simulations using hair meshes can produce seemingly natural hair-hair interactions with high performance. The hair mesh in Figure 12 includes 30 rigid body links and 120 chains, and the simulation runs at 92 fps on a 2.14 GHz Intel Core 2 Duo processor with a single thread (note that such a simulation can be trivially multi-threaded).

5 Conclusions and Future Directions

We have introduced hair meshes for modeling hair using polygonal surface tools. Our technique allows an artist to create complex hair styles easily by providing explicit control over the overall shape of the hair surface. By automatically placing internal vertices, the artist can concentrate on the outer surface shape of the hair, which significantly simplifies the hair modeling process without limiting direct control over the hair model shape.

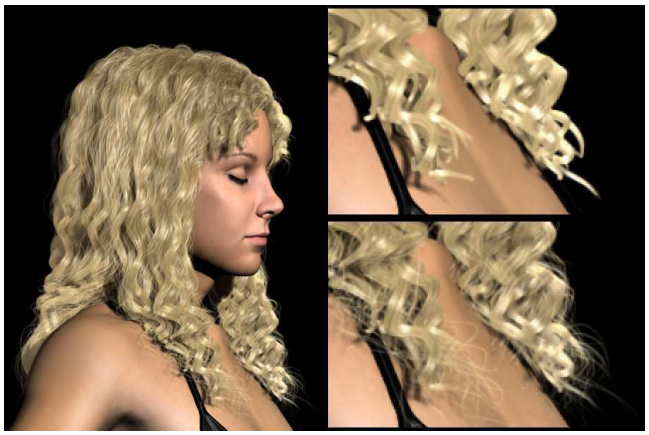


Figure 11: *Frizzy strands generated directly from the hair mesh as an additional hair group on top of the hairs from Figure 9. The close-up view on the right shows the effect of frizzy strands.*

We believe there are more applications of hair meshes than just modeling. An example of using hair meshes for hair simulation is presented in the previous section. Furthermore, real-time rendering of deforming hair such as the animation produced by our hair simulation can be accelerated using hair meshes. Since the hair is completely determined by the geometry of the hair mesh, the hair geometry can be synthesized on the GPU simply by sending the deformed positions of the hair mesh vertices, thereby significantly saving graphics bus bandwidth. Hair meshes may also be used to approximate shadow and ambient occlusion computations.

Acknowledgements

We would like to thank Lee Perry-Smith (www.ir-ltd.net) for providing the models and producing most of the hairstyles in this paper. This work was supported in part by NSF grant CCF-07024099.

References

ALTER, J. S., 2004. Hair generation and other natural phenomena with surface derived control volumes in computer graphics and animation. U.S. Patent 6720962.

ANJYO, K., USAMI, Y., AND KURIHARA, T. 1992. A simple method for extracting the natural beauty of hair. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 111–120.

CATMULL, E., AND ROM, R. J. 1974. A class of local interpolating splines. In *Computer Aided Geometric Design*, Academic Press, Orlando, FL, USA, 317–326.

CHANG, J. T., JIN, J., AND YU, Y. 2002. A practical model for hair mutual interactions. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM, New York, NY, USA, 73–80.

CHEN, L.-H., SAEYOR, S., DOHI, H., AND ISHIZUKA, M. 1999. A system of 3d hair style synthesis based on the wisp model. *The Visual Computer 15*, 4, 159–170.

Figure 12: *Sample frames captured from our real-time hair mesh simulation system. The last frame shows the structure of the simulated hair mesh, which has 150 vertices in 5 layers. The simulation runs at 92 fps on a 2.14 GHz Intel Core 2 Duo processor with a single thread.*

CHOE, B., AND KO, H.-S. 2005. A statistical wisp model and pseudophysical approaches for interactive hairstyle generation. *IEEE Transactions on Visualization and Computer Graphics 11*, 2, 160–170.

DALDEGAN, A., THALMANN, N. M., KURIHARA, T., AND THALMANN, D. 1993. An integrated system for modeling, animating and rendering hair. In *Eurographics '93*, Blackwell Publishers, Oxford, UK, R. J. Hubbold and R. Juan, Eds., Eurographics, 211–221.

FLOATER, M. S. 2003. Mean value coordinates. *Computer Aided Geometric Design 20*, 1, 19–27.

FU, H., WEI, Y., TAI, C.-L., AND QUAN, L. 2007. Sketching hairstyles. In *SBIM '07: Proceedings of the 4th Eurographics Workshop on Sketch Based Interfaces and Modeling*, ACM, New York, NY, USA, 31–36.

GRABLI, S., SILLION, F., MARSCHNER, S. R., AND LENGYEL, J. E. 2002. Image-based hair capture by inverse lighting. In *Proc. Graphics Interface*, 51–58.

HADAP, S., AND MAGNENAT-THALMANN, N. 2000. Interactive hair styler based on fluid flow. In *Eurographics Workshop on Computer Animation and Simulation 2000*, Springer, 87–99.

KIM, T.-Y., AND NEUMANN, U. 2000. A thin shell volume for modeling human hair. In *CA '00: Proceedings of the Computer Animation*, IEEE Computer Society, Washington, DC, USA, 104.

KIM, T.-Y., AND NEUMANN, U. 2002. Interactive multiresolution hair modeling and editing. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2002) 21*, 3, 620–629.

KOH, C. K., AND HUANG, Z. 2001. A simple physics model to animate human hair modeled in 2d strips in real time. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, Springer-Verlag New York, Inc., New York, NY, USA, 127–138.

KONG, W., TAKAHASHI, H., AND NAKAJIMA, M. 1997. Generation of 3d hair model from multiple pictures. In *Proceedings of Multimedia Modeling*, 183–196.

LEE, D. W., AND KO, H. S. 2001. Natural hairstyle modeling and animation. *Graphical Models 63*, 2, 67–85.

LIANG, W., AND HUANG, Z. 2003. An enhanced framework for real-time hair animation. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, IEEE Computer Society, Washington, DC, USA, 467.

MALIK, S. 2005. A sketching interface for modeling and editing hairstyles. In *SBIM '05: Proceedings of the 2nd Eurographics Workshop on Sketch Based Interfaces and Modeling*, 185–194.

MAO, X., ISOBE, S., ANJYO, K., AND IMAMIYA, A. 2005. Sketchy hairstyles. In *CGI '05: Proceedings of the Computer Graphics International 2005*, IEEE Computer Society, Washington, DC, USA, 142–147.

NOBLE, P., AND TANG, W. 2004. Modelling and animating cartoon hair with nurbs surfaces. In *CGI '04: Proceedings of the Computer Graphics International*, IEEE Computer Society, Washington, DC, USA, 60–67.

PARIS, S., HECTOR M. BRICE N., AND SILLION, F. X. 2004. Capture of hair geometry from multiple images. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2004) 23*, 3, 712–719.

PARIS, S., CHANG, W., KOZHUSHNYAN, O. I., JAROSZ, W., MATUSIK, W., ZWICKER, M., AND DURAND, F. 2008. Hair photobooth: geometric and photometric acquisition of real hairstyles. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2008) 27*, 3, Article 30.

SHEWCHUK, J. R. 1994. An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., Pittsburgh, PA, USA.

WANG, T., AND YANG, X. D. 2004. Hair design based on the hierarchical cluster hair model. *Geometric modeling: techniques, applications, systems and tools*, 330–359.

WANG, L., YU, Y., ZHOU, K., AND GUO, B. 2009. Example-based hair geometry synthesis. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2009) 28*, 3, Article 56.

WARD, K., BERTAILS, F., KIM, T.-Y., MARSCHNER, S. R., CANI, M.-P., AND LIN, M. C. 2007. A survey on hair modeling: Styling, simulation, and rendering. *IEEE Transactions on Visualization and Computer Graphics 13*, 2, 213–234.

WARD, K., GALOPPO, N., AND LIN, M. 2007. Interactive virtual hair salon. *Presence: Teleoperators and Virtual Environments 16*, 3, 237–251.

WEI, Y., OFEK, E., QUAN, L., AND SHUM, H.-Y. 2005. Modeling hair from multiple views. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2005) 24*, 3, 816–820.

WITHER, J., BERTAILS, F., AND CANI, M.-P. 2007. Realistic hair from a sketch. In *International Conference on Shape Modeling and Applications*, IEEE, Lyon, France, IEEE, 33–42.

XU, Z., AND YANG, X. D. 2001. V-hairstudio: An interactive tool for hair design. *IEEE Computer Graphics and Applications 21*, 3, 36–43.

YANG, X. D., XU, Z., WANG, T., AND YANG, J. 2000. The cluster hair model. *Graphical Models 62*, 2, 85–103.

YU, Y. 2001. Modeling realistic virtual hairstyles. In *PG '01: Proc. of the 9th Pacific Conference on Comp. Graphics and Applications*, IEEE Computer Society, Washington, DC, USA, 295.

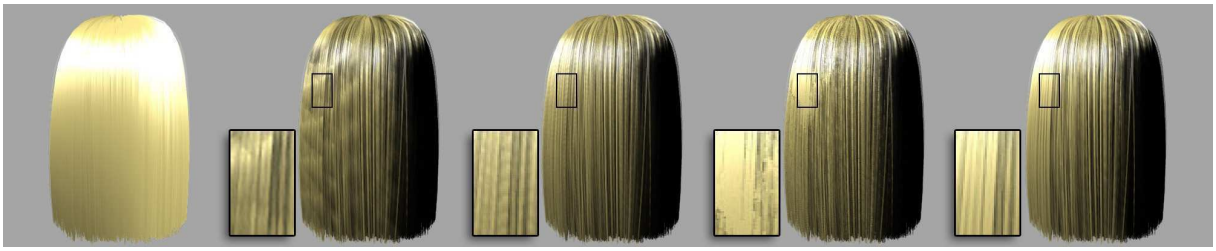
EUROGRAPHICS 2008 / G. Drettakis and R. Scopigno
(Guest Editors)

Volume 27 (2008), Number 2

Deep Opacity Maps

Cem Yuksel[1] and John Keyser[2]

Department of Computer Science, Texas A&M University
[1]cem@cemyuksel.com [2]keyser@cs.tamu.edu



No shadows	Opacity Shadow Maps	Opacity Shadow Maps	Density Clustering	**Deep Opacity Maps**
–	16 layers	128 layers	4 layers	**3 layers**
(154 fps)	(81 fps)	(2.3 fps)	(73 fps)	**(114 fps)**

Figure 1: *Layering artifacts of Opacity Shadow Maps are visible even with 128 layers, while Density Clustering has artifacts due to inaccuracies. Deep Opacity Maps with only 3 layers can generate an artifact free image with the highest frame rate.*

Abstract

We present a new method for rapidly computing shadows from semi-transparent objects like hair. Our deep opacity maps method extends the concept of opacity shadow maps by using a depth map to obtain a per pixel distribution of opacity layers. This approach eliminates the layering artifacts of opacity shadow maps and requires far fewer layers to achieve high quality shadow computation. Furthermore, it is faster than the density clustering technique, and produces less noise with comparable shadow quality. We provide qualitative comparisons to these previous methods and give performance results. Our algorithm is easy to implement, faster, and more memory efficient, enabling us to generate high quality hair shadows in real-time using graphics hardware on a standard PC.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Color, shading, shadowing, and texture

Keywords: shadow maps, semi-transparent shadows, hair shadows, real-time shadows, GPU algorithms

1. Introduction

Self-shadowing is an essential visual element for rendering semi-transparent objects like hair, fur, smoke, and clouds. However, handling the transparency component is either inefficient or not possible for simple shadowing techniques. Various algorithms have been proposed to address this issue both for offline rendering [LV00, AL04] and interactive/real-time rendering [KN01, MKBvR04]. In this paper we present the *deep opacity maps* method, which allows real-time hair rendering with dynamic lighting and semi-transparent shadows. This new method is faster than the previous ones and produces artifact free shadows (Figure 1). Even though we focus on hair shadows, our method is applicable to other semi-transparent objects.

The deep opacity maps method combines shadow mapping [Wil78] and opacity shadow maps [KN01] to give a

better distribution of opacity layers. We first render the hair geometry as opaque primitives from the light's view, recording the depth values on a shadow map. Next we render an opacity map from the light's view similar to opacity shadow maps. The novelty of our algorithm lies in the way that the opacity layers are distributed using the depth map to create opacity layers that vary in depth from the light source on a per-pixel basis. Unlike previous interactive/real-time transparent shadowing techniques [KN01, MKBvR04], this new layer distribution guarantees that the direct illumination coming from the light source without being shadowed is captured correctly. This property of deep opacity maps eliminates the layering artifacts that are apparent in opacity shadow maps. Moreover, far fewer layers are necessary to generate high quality shadows.

The layering artifacts of the previous methods are especially significant in animated sequences or straight hair models. Figure 1 shows a comparison of our deep opacity maps algorithm to the previous methods. Layering artifacts in opacity shadow maps [KN01] are significant when 16 layers are used, and even with 128 layers, diagonal dark stripes are still visible on the left side of the hair model. Density clustering [MKBvR04] produces a good approximation to the overall shadows, but still suffers from visible artifacts around the specular region. However, our deep opacity maps method can produce an artifact-free image with fewer layers and it is significantly faster.

The next section describes the previous methods. The details of our deep opacity maps algorithm are explained in Section 3. We present the results of our method in Section 4, and we discuss the advantages and limitations in Section 5 before concluding in Section 6.

2. Related Work

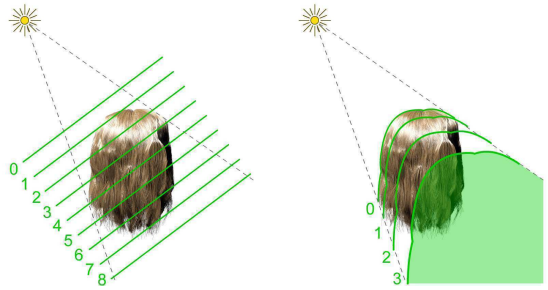
Most shadow computation techniques developed for hair are based on *Shadow Maps* [Wil78]. In the first pass of shadow mapping, shadow casting objects are rendered from the light's point of view and depth values are stored in a depth map. While rendering the scene from the camera view in the second pass, to check if a point is in shadow, one first finds the corresponding pixel of the shadow map, and compares the depth of the point to the value in the depth map. The result of this comparison is a binary decision, so shadow maps cannot be used for transparent shadows.

Deep Shadow Maps [LV00] is a high quality method for offline rendering. Each pixel of a deep shadow map stores a 1D approximate transmittance function along the corresponding light direction. To compute the transmittance function, semi-transparent objects are rendered from the light's point of view and a list of fragments is stored for each pixel. The transmittance function defined by these fragments is then compressed into a piecewise linear function of approximate transmittance. The value of the transmittance function starts decreasing after the depth value of the first fragment in the corresponding light direction. The shadow value at any point is found similar to shadow maps, but this time the depth value is used to compute the transmittance function at the corresponding pixel of the deep shadow map, which is then converted to a shadow value.

The *Alias-free Shadow Maps* [AL04] method is another offline technique that can generate high quality semi-transparent shadows. In this method, rasterization of the final image takes place before the shadow map generation to find the 3D positions corresponding to every pixel in the final image. Then, shadows at these points are computed from the light's point of view, handling one occluding piece of geometry at a time.

Opacity Shadow Maps [KN01] is essentially a simpler version of deep shadow maps that is designed for interactive hair rendering. It first computes a number of planes that slice the hair volume into layers (Figure 2a). These planes are perpendicular to the light direction and are identified by their distances from the light source (i.e. depth value). The opacity map is then computed by rendering the hair structure from the light's view. A separate rendering pass is performed for each slice by clipping the hair geometry against the separating planes. The hair density for each pixel of the opacity map is computed using additional blending on graphics hardware. The slices are rendered in order starting from the slice nearest to the light source, and the value of the previous slice is accumulated to the next one. Once all the layers are rendered, this opacity map can be used to find the transmittance from the occlusion value at any point using linear interpolation of the occlusion values at the neighboring slices. Depending on the number of layers used, the quality of opacity shadow maps can be much lower than deep shadow maps, since the interpolation of the opacities between layers generates layering artifacts on the hair. These artifacts remain visible unless a large number of layers are used.

Mertens et. al. proposed the *Density Clustering* approach [MKBvR04] to adjust the sizes and the positions of opacity layers separately for each pixel of the shadow map. It uses k-means clustering to compute the centers of the opacity layers. Then, each hair fragment is assigned to the opacity layer with the nearest center, and the standard deviation of the opacity layer times $\sqrt{3}$ is used as the size of the layer. Once the opacity layers are positioned, the hair geometry is rendered once again from the light's point of view and the opacity value of each layer is recorded. In general, density clustering generates better opacity layer distributions than opacity shadow maps, but it also introduces other complications and limitations. Density clustering's major limitation is that it cannot be extended to have a high number of layers due to the way that the layering is computed, and it is only suitable for a small number of clusters (the original paper [MKBvR04] suggests 4 clusters). Moreover, k-means clustering is an iterative method and each iteration requires a separate



(a) Opacity Shadow Maps (b) Deep Opacity Maps

Figure 2: *Opacity shadow maps use regularly spaced planar layers. Our deep opacity maps use fewer layers, conforming to the shape of the hair model.*

pass that renders the whole hair geometry. The efficiency of the clustering depends on the initial choice of the opacity layer centers. Even if only a single pass of k-means clustering is performed, the density clustering method requires 4 passes to generate the shadow map. Finally, like opacity shadow maps, density clustering cannot guarantee that unshadowed direct illumination is captured correctly since the first opacity layer can begin before the first hair fragment.

The deep opacity maps method presented in this paper has advantages over these prior methods. It guarantees that the direct illumination of the surface hairs is calculated correctly. Unlike opacity shadow maps, opacity interpolation occurs within the hair volume, thus hiding possible layering artifacts. Unlike density clustering, deep opacity maps can easily use arbitrary numbers of layers (though usually 3 layers are sufficient). Comparing to both density clustering and opacity shadow maps, deep opacity maps achieve significantly higher frame rates for comparable quality.

Other previous methods include extensions of opacity shadow maps [KHS04], voxel based shadows [BMC05, ED06, GMT05], precomputed approaches [XLJP06], and physically based offline methods [ZSW04, MM06]. For a more complete presentation of the previous methods please refer to Ward et al. [WBK*07].

3. Deep Opacity Maps Algorithm

Our algorithm uses two passes to prepare the deep opacity map, and the final image is rendered in an additional pass, using this map to compute shadows.

The *first step* prepares the separators between the opacity layers. We render a depth map of the hair as seen from the light source. This gives us, for each pixel of the depth map, the depth z_0 at which the hair geometry begins. Starting from this depth value, we divide the hair volume within the pixel into K layers such that each layer lies from $z_0 + d_{k-1}$ to $z_0 + d_k$ where $d_0 = 0$, $d_{k-1} < d_k$ and $1 \leq k \leq K$. Note that the spacing $d_k - d_{k-1}$ (layer size) does not have to be constant. Even though we use the same d_k values for each pixel, z_0 varies by pixel, so the separators between the layers take the shape of the hair structure (Figure 2). Note that the light source in this setup can be a point or a directional light.

The *second step* renders the opacity map using the depth map computed in the previous step. This requires rendering the hair only once and all computation occurs within the fragment shader. As each hair is rendered, we read the value of z_0 from the depth map and find the depth values of the layers on the fly. We assign the opacity contribution of the fragment to the layer that the fragment falls in and to all the other layers behind it. The total opacity of a layer at a pixel is the sum of all contributing fragments.

We represent the opacity map by associating each color channel with a different layer, and accumulate the opacities using additive blending on the graphics hardware. We reserve one color channel for the depth value, so that it is stored in the same texture with opacities. Therefore, using a single color value with four channels, we can represent three opacity layers. By enabling multiple draw buffers we can output multiple colors per pixel to represent more than three layers (n draw buffers allow $4n-1$ layers). Obviously, using more than three layers will also require multiple texture lookups during final rendering.

One disadvantage of using a small number of layers with deep opacity maps is that it can be more difficult to ensure all points in the hair volume are assigned to a layer. In particular, points beyond the end of the last layer $z_0 + d_k$ do not correspond to any layer (shaded region in Figure 2b). We have a few options: ignore these points (thus, they will not cast shadows), include these points in the last layer (thus, they cast shadows on themselves), or ensure that the last layer lies beyond the hair volume by either increasing the layer sizes or the number of layers. While the last option might seem "ideal," it can lead to unnecessary extra layers that add little visual benefit at more computational cost, since the light intensity beyond a certain point in the hair volume is expected to vanish. We found that the second option, mapping these points onto the last layer, usually gave reasonable results.

Note that our algorithm uses the depth map only for computing the starting points of layers, not for a binary decision of in or out of shadow. Thus, unlike standard shadow mapping, deep opacity maps do not require high precision depth maps. For the scenes in our experiments, we found that using an 8-bit depth map visually performs the same as a 16-bit floating point depth map.

4. Results

To demonstrate the effectiveness of our approach we compare the results of our deep opacity maps algorithm to optimized implementations of opacity shadow maps and density clustering. We extended the implementation of opacity shadow maps with up to 16 layers to simultaneously generate the opacity layers in a single pass by using multiple draw

No shadows	Opacity Shadow Maps	Opacity Shadow Maps	Density Clustering	**Deep Opacity Maps**
–	8 layers	256 layers	4 layers	**3 layers**
(140 fps)	(88 fps)	(0.6 fps)	(47 fps)	**(74 fps)**

Figure 3: *Dark hair model with over one million line segments. The Opacity Shadow Maps method requires many layers to eliminate layering artifacts, Density Clustering approximates the shadows with some noise, while the Deep Opacity Maps method generates an artifact free image with higher frame rate.*

Opacity Shadow Maps - 256 layers

Density Clustering - 4 layers

Deep Opacity Maps - 3 layers

Figure 4: *Enlarged images from Figure 3 comparison.*

buffers, as opposed to the multi-pass implementation proposed in the original method [KN01]. We also introduced an additional pre-computation pass to density clustering, which computes the layer limits before the final image rendering, and achieved higher performance by optimizing the shadow lookup. All images presented in this paper were captured from our real-time hair rendering system using a standard PC with a 2.13GHz Core2 Duo processor and GeForce 8800 graphics card.

We used line drawing for rendering the hair models, and the Kajiya-Kay shading model [KK89] because of its simplicity. Antialiasing in the final image was handled on the graphics hardware using multi-sampling. We did not use antialiasing when generating opacity maps. To achieve a fake transparency effect in the final image we divided the set of hair strands into three disjoint subsets. Each one of these subsets is rendered separately with no transparency on a separate image. The resulting images are then blended to produce the final frame.

Figure 1 shows a straight hair model with 150 thousand line segments. The opacity shadow maps method with 16 layers produces severe artifacts of diagonal dark stripes that correspond to the beginning of each opacity layer. Though significantly reduced, these artifacts are still visible even when the number of layers is increased to 128. Density clustering, on the other hand, produces a good approximation to the overall illumination using 4 layers, however it suffers from a different kind of layering artifact visible around the specular region due to its inaccuracy. Our deep opacity maps technique produces an artifact free image with plausible shadow estimate using only 3 layers, and it is significantly faster than the other methods.

Figures 3 and 5 show two different complex hair styles with over one million and 1.5 million line segments respectively. On both of these models, the opacity shadow maps method with 8 layers produces dark stripes as interpolation artifacts between layers. When 256 layers are used with opacity shadow maps, layering artifacts visually disappear and the resulting shadows approach the correct values, but rendering one frame takes about two seconds. On the other hand, density clustering manages to produce a close approximation using only 4 layers. However, the inaccuracies of density clustering produce some noise in the illumination that are clearly visible in the enlarged images (Figures 4 and 6) and animated sequences. The deep opacity maps method manages to create an artifact free image with smooth illumination changes over the hair surface with significantly higher frame rates and less memory consumption.

Figure 7 demonstrates that deep opacity maps can be used in conjunction with traditional shadow maps. In this image of a hairy teapot, the shadow map handles the opaque shadows due to the teapot and the deep opacity map handles semi-transparent shadows due to the hair strands. Both the hair model and the teapot cast shadows onto each other as well as on the ground plane.

No shadows	Opacity Shadow Maps	Opacity Shadow Maps	Density Clustering	**Deep Opacity Maps**
–	8 layers	256 layers	4 layers	**3 layers**
(104 fps)	(65 fps)	(0.5 fps)	(37 fps)	**(50 fps)**

Figure 5: *Curly hair model with over 1.5 million line segments. Layering artifacts of Opacity Shadow Maps with 8 layers are apparent on the top of the hair model. They disappear with 256 layers with low frame rates. Density Clustering generates a noisy approximation, while the Deep Opacity Maps method generates an artifact free image with higher frame rate.*

Opacity Shadow Maps - 256 layers

Density Clustering - 4 layers

Deep Opacity Maps - 3 layers

Figure 6: *Enlarged images from Figure 5 comparison.*

Figure 7: *A hairy teapot rendered using deep opacity maps with traditional shadow maps for opaque shadows.*

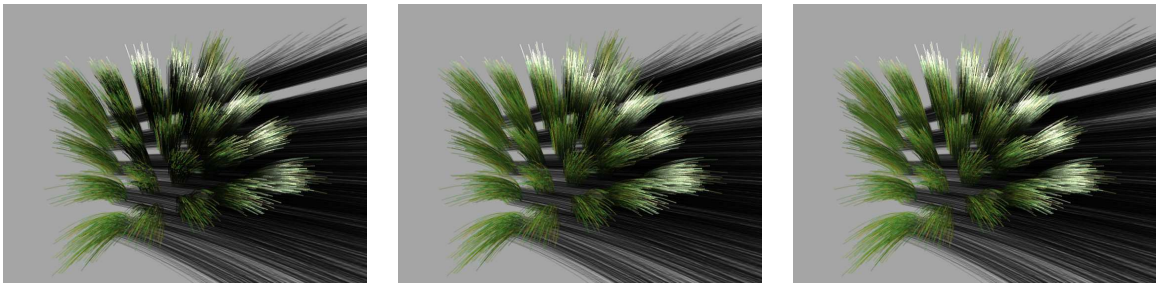
Figure 8 shows clustered strands rendered using deep opacity maps. In Figure 8a, 3 layers are not enough to cover the whole illuminated volume, and dark regions in appear where the strands beyond the last layer cast shadows onto themselves. By increasing the number of layers, as in Figure 8b, incorrect self-shadowing can be eliminated. This can also be achieved by increasing the layer sizes as in Figure 8c; however, this also reduces the shadow accuracy. As can be seen from these images, deep opacity maps can generate high quality shadows with non-uniform hair models.

5. Discussion

The main advantage of our method is that by shaping the opacity layers, we capture direct illumination correctly while eliminating visual layering artifacts by moving interpolation between layers to within the hair volume. This lets us hide possible inaccuracies while also allowing high quality results with fewer layers. Unlike density clustering, which tries to approximate the whole transmittance function, we concentrate the accuracy on the beginning of the transmittance decay (where the shadow begins). By doing so, we aim to be more accurate around the illuminated surface of the hair volume—the part of the hair that is most likely to appear in the final image and where inaccuracies would be most noticeable.

Since we require very few layers, all information can be stored in a small number of textures (a single texture for 3 layers). This makes our algorithm memory efficient and also reduces the load on the fragment shader.

The extreme simplicity of our approach allows us to prepare the opacity map with only 2 render passes, and only one of these passes uses blending. Opacity shadow maps of just a few layers can be generated in only a single pass, however visual plausibility requires many more layers.



(a) 3 layers (b) 7 layers (c) 3 (larger) layers

Figure 8: *Clustered strands with deep opacity maps using different number of layers and different layer sizes.*

For the examples in this paper we used linearly increasing layer sizes with deep opacity maps. This choice of layer distribution provides high accuracy around bright regions where the transmittance begins to decay, while keeping other layers large enough to cover the illuminated part of the hair volume with a few number of layers. Varying layer sizes can also be used for opacity shadow maps, but this would not provide any visual enhancement, since there is no heuristic that can reduce the layering artifacts by changing layer sizes without increasing the number of layers.

In our implementation we observed minor flickering due to aliased line drawing we used while rendering depth and opacity maps. Applying a smoothing filter to the depth and opacity maps reduced this problem, but did not completely remove it. In our experiments we found that using multi-sampling for shadow computations, a standard technique used for smoothing shadow maps, produced better results with additional computation cost.

6. Conclusion

We have introduced the deep opacity maps method, which uses a depth map to achieve per-pixel layering of the opacity map for real-time computation of semi-transparent shadows. We compared both quality and performance of our method to the previous real-time/interactive semi-transparent shadowing techniques. Our results show that deep opacity maps are fast and can generate high quality shadows with minimal memory consumption. Our algorithm does not have any restrictions on the hair model or hair data structure. Since it does not need any pre-computation, it can be used when rendering animated dynamic hair or any other semi-transparent object that can be represented by simple primitives.

References

[AL04] AILA T., LAINE S.: Alias-free shadow maps. In *Eurographics Symp. on Rendering* (2004), pp. 161–166.

[BMC05] BERTAILS F., MÉNIER C., CANI M.-P.: A practical self-shadowing algorithm for interactive hair animation. In *Proc. Graphics Interface* (2005), pp. 71–78.

[ED06] EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *Symposium on Interactive 3D Graphics and Games* (2006), pp. 71–78.

[GMT05] GUPTA R., MAGNENAT-THALMANN N.: Scattering-based interactive hair rendering. In *Comp. Aided Design and Comp. Graphics* (2005), pp. 489–496.

[KHS04] KOSTER M., HABER J., SEIDEL H.-P.: Real-time rendering of human hair using programmable graphics hardware. In *Proceedings of the Computer Graphics International (CGI'04)* (2004), pp. 248–256.

[KK89] KAJIYA J. T., KAY T. L.: Rendering fur with three dimensional textures. In *Proceedings of SIGGRAPH 1989* (1989), pp. 271–280.

[KN01] KIM T.-Y., NEUMANN U.: Opacity shadow maps. In *12th Eurographics Workshop on Rendering Techniques* (2001), pp. 177–182.

[LV00] LOKOVIC T., VEACH E.: Deep shadow maps. In *Proceedings of SIGGRAPH 2000* (2000), pp. 385–392.

[MKBvR04] MERTENS T., KAUTZ J., BEKAERT P., VAN REETH F.: A self-shadow algorithm for dynamic hair using clustered densities. In *Proceedings of Eurographics Symposium on Rendering 2004* (2004), pp. 173–178.

[MM06] MOON J. T., MARSCHNER S. R.: Simulating multiple scattering in hair using a photon mapping approach. In *Proceedings of SIGGRAPH 2006* (2006), pp. 1067–1074.

[WBK*07] WARD K., BERTAILS F., KIM T.-Y., MARSCHNER S. R., CANI M.-P., LIN M.: A survey on hair modeling: Styling, simulation, and rendering. *IEEE TVCG 13*, 2 (Mar-Apr 2007), 213–34.

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. In *SIGGRAPH '78* (1978), pp. 270–274.

[XLJP06] XU S., LAU F. C., JIANG H., PAN Y.: A novel method for fast and high-quality rendering of hair. In *Proc. EGSR'06* (2006), pp. 331–341.

[ZSW04] ZINKE A., SOBOTTKA G., WEBER A.: Photo-realistic rendering of blond hair. In *Vision, Modeling, and Visualization 2004* (2004), pp. 191–198.

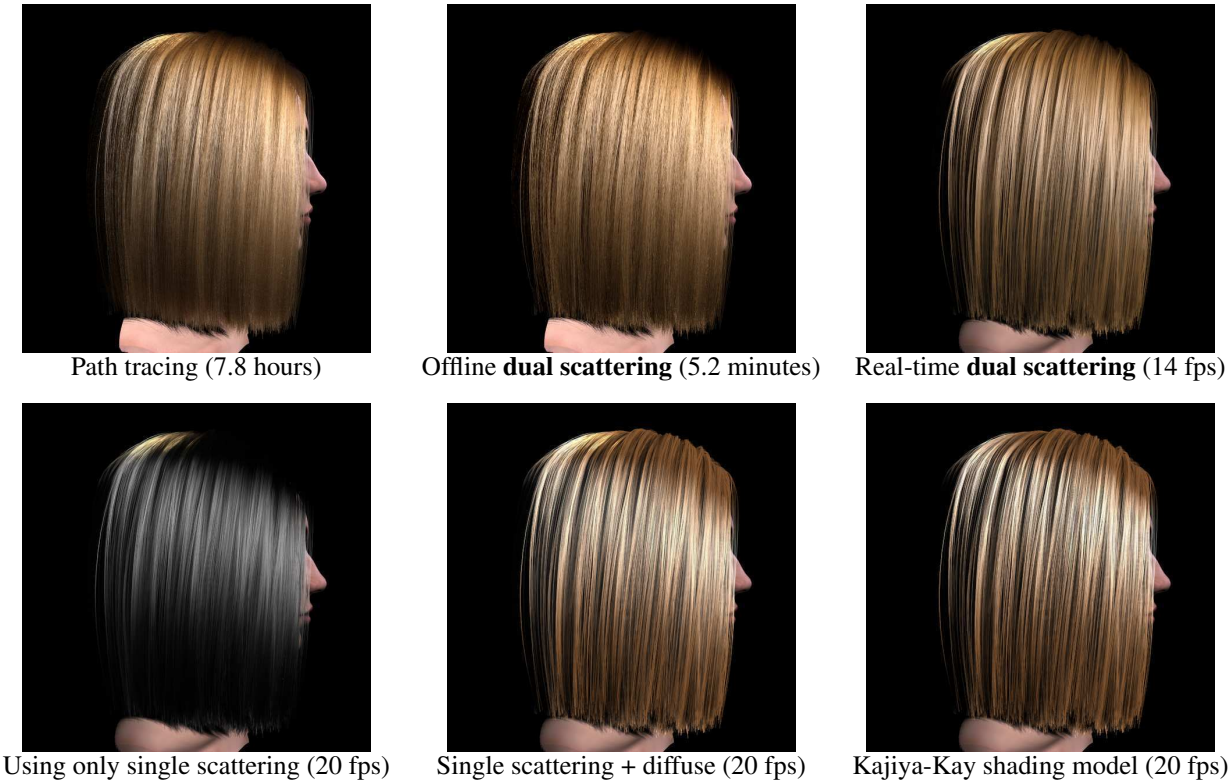
Dual Scattering Approximation for Fast Multiple Scattering in Hair

Arno Zinke*
Institut für Informatik II
Universität Bonn

Cem Yuksel†
Dept. of Computer Science
Texas A&M University

Andreas Weber‡
Institut für Informatik II
Universität Bonn

John Keyser§
Dept. of Computer Science
Texas A&M University



Path tracing (7.8 hours) | Offline **dual scattering** (5.2 minutes) | Real-time **dual scattering** (14 fps)

Using only single scattering (20 fps) | Single scattering + diffuse (20 fps) | Kajiya-Kay shading model (20 fps)

Figure 1: *Comparison of our method to path tracing and existing hair shading methods with deep opacity maps. Our dual scattering approximations (offline ray shooting and real-time GPU-based implementations) achieve close results to the path tracing reference without any parameter adjustment and with significantly improved rendering times. Using single scattering only fails to produce the correct hair color. Adding an ad-hoc diffuse component or Kajiya-Kay shading model also fails to achieve the same realism, even after hand tweaking the diffuse color and shadow opacity to match the reference. The hair model has 50K strands and 1.4M line segments.*

Abstract

When rendering light colored hair, multiple fiber scattering is essential for the right perception of the overall hair color. In this context, we present a novel technique to efficiently approximate multiple fiber scattering for a full head of human hair or a similar fiber based geometry. In contrast to previous ad-hoc approaches, our method relies on the physically accurate concept of the Bidirectional Scattering Distribution Functions and gives physically plausible results with no need for parameter tweaking. We show that complex scattering effects can be approximated very well by using aggressive simplifications based on this theoretical model. When compared to unbiased Monte-Carlo path tracing, our approximations preserve photo-realism in most settings but with rendering times at least two-orders of magnitude lower. Time and space complexity are much lower compared to photon mapping-based techniques and we can even achieve realistic results in real-time on a standard PC with consumer graphics hardware.

*e-mail: zinke@cs.uni-bonn.de.de
†e-mail: cem@cemyuksel.com
‡e-mail: weber@cs.uni-bonn.de.de
§e-mail: keyser@cs.tamu.edu

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: Hair rendering, multiple scattering, GPU algorithms

1 Introduction

Accounting for multiple scattering is a key factor in the realistic rendering of human hair. Particularly in dense, light-colored hair, multiple scattering provides a critical component of the hair color, similar to the effect seen in subsurface scattering of translucent materials. Unfortunately, the high geometric complexity of hair

models coupled with the complexity of the light interaction in hair volumes makes computing this multiple scattering effect difficult. Even for the simplest case of computing illumination due to a single point light source, contributions of several different light paths need to be determined to achieve reasonable visual accuracy.

Recently, two different photon mapping based methods have been proposed to accelerate the multiple scattering computation in hair. Even though these methods present a significant speed improvement as compared to brute force path tracing, they still require hours to compute a single frame and a large amount of memory to store the photon map. On the other hand, in real-time graphics the multiple scattering property of hair is overly simplified and treated as transparent shadows (completely ignoring the effect of the circular shape of hair fibers) coupled with an ad-hoc diffuse component. This extreme simplification prevents the use of physically based hair shaders, and gives hair a dull appearance. Furthermore, rigorous parameter tweaking is necessary to achieve acceptable results, the realism of which is always questionable.

In this paper, we introduce the concept of dual scattering, which splits the multiple scattering computation into two components: *global multiple scattering* and *local multiple scattering*. The global multiple scattering component aims to compute the light traveling through the hair volume and reaching the neighborhood of the point of interest, while local multiple scattering accounts for the scattering events within this neighborhood. Exploiting physically based scattering properties of real hair fibers, we introduce several theoretical simplifications for computing both of these components; this permits extremely efficient multiple scattering computations with minimal accuracy compromise. As a result of our aggressive theoretical simplifications, the implementation of global multiple scattering becomes very similar to standard semi-transparent hair shadowing techniques, while local multiple scattering is modeled as a material property derived from hair fiber properties.

The top row of Figure 1 compares different implementations of our dual scattering method to path tracing. The path tracing image takes 7.8 hours to compute, while by using dual scattering we can reduce this time to 5.2 minutes in our offline implementation or even 14 frames per second with our GPU implementation. The precomputation time required for all dual scattering implementations is only a few seconds. As can be seen from these images, we can maintain visual accuracy with significant improvement in computation time. Just as important, all calculations are based on computable or measurable values, so there is no unintuitive parameter tweaking (such as smoothing radius) that existed in previous physically based techniques.

The bottom row of Figure 1 shows the results of single scattering only and existing hair shading methods with semi-transparent shadows using deep opacity maps. Even after rigorous parameter tweaking of ad-hoc diffuse color used by the shaders and the opacity value used for the shadow computation, the results still differ significantly from the path tracing reference. This is mainly because several important multiple scattering effects like directionality, color shifts, and successive blurring cannot be modeled by these oversimplified formulations. Therefore, the realism that we achieve using our dual scattering approximation goes beyond existing real-time hair rendering methods. Note that dual scattering is not an ad-hoc addition for enhancing real-time hair rendering; it is a physically-based simulation, the simplicity of which permits real-time implementation.

In the next section we provide a brief summary of the previous work and the terminology we use throughout the paper. The theory of dual scattering is introduced in Section 3. We present the details of our different implementations in Section 4 and our results in Section 5. Finally, we conclude with a short discussion in Section 6.

2 Background

There is a large body of work on hair modeling and rendering in computer graphics. In this section we only review the techniques that are most relevant to our approach and we concentrate on physically based methods. For a more comprehensive overview of hair rendering please refer to the recent survey paper of Ward et al. [2007].

2.1 Prior Work

Marschner et al. [2003] presented the seminal work in the physically-based rendering of human hair. Their paper developed a far-field scattering model for hair based on measurements of actual hair. They modeled hair fibers as dielectric cylinders with colored interiors, and their model was able to account for important single scattering effects, such as multiple highlights, and deliver realistic results for dark colored hair.

Further work showed that multiple fiber scattering is essential for correct perception of hair color, particularly for light colored hair. New models [Zinke et al. 2004; Moon and Marschner 2006; Zinke and Weber 2007] were developed to generalize the approach of Marschner et al. [2003] to account for these multiple scattering effects.

In order to solve this more general illumination problem, methods such as path tracing can be used, though this often leads to prohibitive running times. Both Moon and Marschner [2006] and Zinke and Weber [2006] use photon mapping approaches to estimate the multiple scattering. Although both methods can produce accurate results similar to path tracing in many situations, they are computationally costly and require high resolution (memory consuming) photon maps. These methods cannot be made interactive. Yuksel et al. [2007] have also proposed an alternative projection-based method for rendering global illumination for fibers. However, this approach makes several simplifications, e.g. neglecting inter-reflections, that reduce accuracy.

A number of other techniques have addressed some aspects of multiple scattering effects for hair volumes, especially in the context of self-shadowing for interactive hair rendering [Lokovic and Veach 2000; Kim and Neumann 2001; Mertens et al. 2004; Bertails et al. 2005; Xu et al. 2006; Hadwiger et al. 2006; Yuksel and Keyser 2008]. However, the main drawback of most of these methods is that they use ad-hoc simplifications and non-physical parameters, which cannot be derived from physical hair fiber properties. While plausible images can be produced by these methods in some situations, parameters need to be repeatedly tweaked to be appropriate for a particular scene and lighting condition. An exception is the work of Gupta and Magnenat-Thalmann [2005], which provides a more complex scattering-based approach. However, their approach is purely density-based and uses an ad-hoc volumetric scattering function without any physical basis. Therefore it cannot capture important phenomena such as directionality of multiple scattering, successive blurring, or subtle color shifting effects.

For simulating volumetric light transport, cheap analytical multiple scattering models have been presented [Kniss et al. 2003; Premoze et al. 2004]. By taking into account optical properties of a participating media Premoze et al. [2004] use practical approximations to efficiently compute important features of multiply scattered light, such as spatial and angular spread of an incident beam. The radiative transfer is computed based on only a few prototypic path samples. However, even though this work is related in spirit to our approach, it has not yet been demonstrated to work with spatially varying and highly anisotropic scattering of hair fibers.

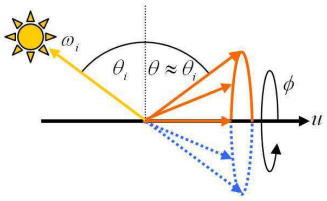


Figure 2: *Definition of angles and directions*

2.2 Terminology and Notation

The discussion in this paper includes numerous symbols and terms that might not be familiar to readers. We follow the terminology and notation used by Marschner et al. [2003] and Zinke and Weber [2007]. While we would refer the reader to those sources for more complete descriptions, we give a brief review and overview of terminology here.

Referring to Figure 2, consider the tangent to the hair fiber (i.e. a vector running down the central fiber axis), u. The normal plane is the plane perpendicular to u. Directions within this normal plane are referred to as *azimuthal* angles, and are expressed with the symbol ϕ. An angle formed with respect to the normal plane is called the *longitudinal* inclination, and is expressed with the symbol θ. For a point on the fiber, a direction is expressed by the symbol ω. Thus, any direction ω_a is equivalently expressed by the longitudinal inclination θ_a and the azimuthal angle ϕ_a.

The amount of incoming light from one given direction, ω_i, scattered in another given outgoing direction, ω_o, is expressed by a Bidirectional Scattering Distribution Function. For a hair fiber, Zinke and Weber [2007] defined a very general Bidirectional Fiber Scattering Distribution Function that incorporated fiber geometry in addition to incident and output angles. By assuming the viewer and light source are sufficiently far from the hair fiber, one can ignore the hair geometry and local (near-field) variations in geometry, resulting in a simplified Bidirectional Curves Scattering Distribution Function (BCSDF), $f_s(\omega_i, \omega_o)$. The scattering function defined by Marschner et al. [2003] is one possible BCSDF.

A hair fiber is generally thought of as a cylinder, and as light hits these cylinder boundaries, it can either be transmitted (T) or reflected (R). Thus, TT refers to the light passing into then out of the fiber in a generally forward direction, while R and TRT refer to backward reflections, either before (R) or after (TRT) passing through the fiber. Computing a BCSDF generally involves computing these three components, and more complex interactions (e.g. TRRT) are ignored.

We follow both Marschner et al. [2003] and Zinke and Weber [2007] in using a formulation for the BCSDF as a product of a longitudinal function, M_t, and an azimuthal function, N_t for each of the three reflection types $t \in (\mathrm{R}, \mathrm{TT}, \mathrm{TRT})$. M is modeled as a normalized Gaussian function $g(\alpha_t, (\beta_t)^2)$ with mean α_t and standard deviation β_t specified according to measured properties of hairs. N is precomputed in a 2D table of difference angles $\theta = (\theta_o - \theta_i)/2$ and $\phi = \phi_o - \phi_i$. See Marschner et al.'s paper [2003] for more details on these computations. Note that in this paper, $g(a, b)$ refers to a unit area Gaussian function defined in variable a with a zero mean and variance b.

3 Dual Scattering

In this section we present the theoretical foundations of the dual scattering approximation. We simplify the computation of the complicated physical multiple scattering phenomena using properties of real human hair fibers and realistic human hair models. As a result of these simplifications, we achieve a physically-based formulation for multiple scattering, which can be implemented in a very similar way to standard hair shadowing techniques.

Similar to Moon and Marschner [2006], in our rendering system we use one dimensional fibers, disregarding the illumination variation across the width of a hair strand. In this form, the general rendering equation for outgoing radiance L_o towards a direction ω_o at a point x can be written as

$$L_o(x, \omega_o) = \int_\Omega L_i(x, \omega_i)\, f_s(\omega_i, \omega_o)\, \cos\theta_i \, \mathrm{d}\omega_i \ , \qquad (1)$$

where $L_i(x, \omega_i)$ is the incident radiance from direction ω_i, $f_s(\omega_i, \omega_o)$ is the BCSDF of a hair fiber, and Ω is the set of all directions over the sphere. The incident radiance function $L_i(x, \omega_i)$ includes all light paths scattered inside the hair volume such that

$$L_i(x, \omega_i) = \int_\Omega L_d(\omega_d)\, \Psi(x, \omega_d, \omega_i)\, \mathrm{d}\omega_d \ , \qquad (2)$$

where $L_d(\omega_d)$ is the incoming radiance from outside the hair volume from direction ω_d (assuming distant illumination), and $\Psi(x, \omega_d, \omega_i)$ is the *multiple scattering function* denoting the fraction of light entering the hair volume from direction ω_d that is scattered inside the hair volume and finally arriving at point x from direction ω_i.

The main concept behind the dual scattering method is to approximate the multiple scattering function as a combination of two components: *global multiple scattering* and *local multiple scattering*. The global multiple scattering function Ψ^G is used to compute the irradiance arriving at the neighborhood of a point x inside the hair volume, and the local multiple scattering function Ψ^L approximates the multiple scattering of this irradiance within the local neighborhood of x (Figure 3). Therefore, the multiple scattering function becomes the sum of global multiple scattering and global multiple scattering that goes through further local multiple scattering

$$\Psi(x, \omega_d, \omega_i) = \Psi^G(x, \omega_d, \omega_i)\, \left(1 + \Psi^L(x, \omega_d, \omega_i)\right) \ . \qquad (3)$$

For the sake of simplicity we explain our dual scattering method assuming that the subject hair model is illuminated by a single directional light source. At the end of this section we explain how this approach can be extended for other light source types, multiple light sources, image based lighting, and global illumination.

3.1 Global Multiple Scattering

Global multiple scattering is especially important for light colored hair types as they permit outside illumination to penetrate

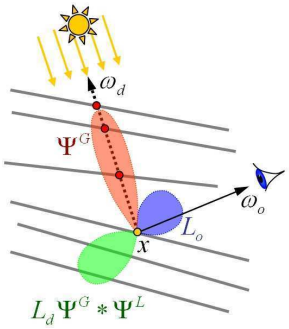


Figure 3: *Based on intersections (red dots) along the shadow path (dashed line), the multiple scattering distributions Ψ_L and Ψ_G and the resulting outgoing radiance L_o are locally estimated.*

deeper into the hair volume. According to the measurements of Marschner et al. [2003], light scattering from human hair fibers is strongly anisotropic in the longitudinal direction, while it is much less anisotropic in the azimuthal direction. Because of this rather wide azimuthal scattering property of human hair, the computation of global multiple scattering should handle several different rather complicated light paths. On the other hand, the energy of light arriving at a point does not depend on the actual path, but the quality of the scattering events along the path. Assuming statistically independent scattering events, all other geometrical properties of the cluster (such as the distance between fibers) can be neglected. Moreover, a realistic human hair model presents strong local similarity in the orientation of neighboring hair fibers, therefore the probabilities of different light paths exhibit this similarity.

Based on these observations, in our dual scattering method we simplify the computation of global multiple scattering by exploring the light scattering properties along only a single light path, namely the *shadow path* (in the direction ω_d), and use the information we gather for approximating the contributions of other possible paths (Figure 3). Along the shadow path, we classify all possible scattering directions of a scattering event into two groups: *Forward Scattering* and *Backward Scattering*, which correspond to all directions in the front and back half of the scattering cone relative to the original light source.

Note that the strong TT component of hair fiber scattering is included in the front half-cone. Therefore, for light colored hair models forward scattering is significantly stronger than backward scattering. Furthermore, light paths that go through a backward scattering before reaching the neighborhood of point x scatter away from this point, and they need another backward scattering to reverse their direction. As a result, for light colored hair types only a very small portion of global multiple scattering includes backward scattering. In our formulation we disregard these double (or more) backward scattered paths, approximating the global multiple scattering by only front scattered radiance. (Note that backward scattered paths are ignored only for the global multiple scattering computation and they will be included in local multiple scattering). As a result, we approximate the global multiple scattering as

$$\Psi^G(x, \omega_d, \omega_i) \approx T_f(x, \omega_d)\, S_f(x, \omega_d, \omega_i)\ , \tag{4}$$

where $T_f(x, \omega_d)$ is the total transmittance and $S_f(x, \omega_d, \omega_i)$ accounts for the spread of global multiple scattering into different directions. Therefore, to compute the global multiple scattering we need to evaluate these two functions.

3.1.1 Forward Scattering Transmittance

The transmittance function $T_f(x, \omega_d)$ gives the total attenuation of a front scattered light path arriving at the point x. Therefore, the transmittance function depends on the number of scattering events n along the shadow path and the average attenuation $\bar{a}_f(\theta_d)$ caused by each forward scattering event as

$$T_f(x, \omega_d) = d_f(x, \omega_d) \prod_{k=1}^{n} \bar{a}_f\big(\theta_d^k\big)\ , \tag{5}$$

where $d_f(x, \omega_d)$ is the density factor and θ_d^k is the longitudinal inclination at the k^{th} scattering event. Note that if $n = 0$, the point x is illuminated directly and the transmittance function is set to 1. We use the density factor $d_f(x, \omega_d)$ to account for the fact that not all points x are located inside a dense cluster. Thus, the front scattered irradiance on x comes from only a subset of all directions as shown in Figure 4. Although the density factor theoretically depends on the hair density and the specific location of x, in practice we simply

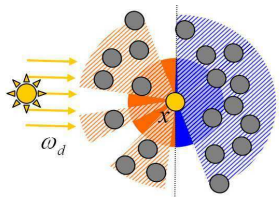


Figure 4: *Cross section of a hair cluster. The fiber at x is not fully surrounded by other strands, so it can receive multiple scattered radiance only from the shaded sections. d_f accounts for fibers in the orange region, while d_b accounts for fibers in the blue region.*

use a constant value (between 0 and 1) to approximate this factor based on the overall hair density. In our experiments we found that for realistic human hair models density factors between 0.6 and 0.8 give realistic results (as compared to a path tracing reference). For all examples in this paper, the density factor is set to 0.7.

We compute the average attenuation $\bar{a}_f(\theta_d)$ directly from the fiber scattering function f_s as the total radiance on the front hemisphere due to isotropic irradiance along the specular cone:

$$\bar{a}_f(\theta_d) = \frac{1}{\pi} \int_{\Omega_f} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} f_s\left((\theta_d, \phi), \omega\right) \cos\theta_d \, \mathrm{d}\phi \, \mathrm{d}\omega, \tag{6}$$

where Ω_f is all directions over the front hemisphere and θ_d is the inclination of direct illumination at the scattering event.

3.1.2 Forward Scattering Spread

The spread function $S_f(x, \omega_d, \omega_i)$ approximates the final angular distribution of front scattered light to find the probability of radiance coming to the point x from direction ω_i. Because of the wide azimuthal scattering property of hair fibers, front scattered radiance quickly becomes almost isotropic in the azimuthal direction after only a few scattering events. However, in the longitudinal direction front scattered spread is still rather anisotropic. Therefore, we represent our spread function using a constant term $\tilde{s}_f$ for the azimuthal spread and a narrow Gaussian distribution function g for the longitudinal component:

$$S_f(x, \omega_d, \omega_i) = \frac{\tilde{s}_f(\phi_d, \phi_i)}{\cos\theta_d}\, g\Big(\ \theta_d + \theta_i\ ,\ \bar{\sigma}_f^2(x, \omega_d)\ \Big)\ , \tag{7}$$

where $\tilde{s}_f(\phi_d, \phi_i)$ is $1/\pi$ for forward scattering directions and zero for backward scattering, and $\bar{\sigma}_f^2(x, \omega_d)$ is the total variance of forward scattering in the longitudinal directions. Since the BCSDF of a fiber is represented by a Gaussian distribution (M) in longitudinal directions, we can compute the total variance as the sum of variances of all scattering events along the shadow path

$$\bar{\sigma}_f^2(x, \omega_d) = \sum_{k=1}^{n} \bar{\beta}_f^2\big(\theta_d^k\big)\ , \tag{8}$$

where $\bar{\beta}_f^2(\theta_d^k)$ is the average longitudinal forward scattering variance of the k^{th} scattering event, which is directly taken from the BCSDF of the hair fiber. Note that for a single directional light source when $n = 0$, i.e. when the fiber is being illuminated directly, the spread function becomes a delta function $\delta(\omega_d - \omega_i)$.

3.2 Local Multiple Scattering

The local multiple scattering function accounts for the multiple scattering events within the neighborhood of the point x. Since light paths that go through only forward scattering are included in

the global multiple scattering function, light paths of the local multiple scattering function must include at least one backward scattering. Because of this backward scattering, local multiple scattering is mostly smooth with subtle changes over the hair volume, yet it significantly affects the visible hair color especially for light colored hair types.

In our dual scattering method, we combine local multiple scattering and the BCSDF of the hair fibers, approximating the result with a density factor d_b and backscattering function f_{back} as

$$\Psi^L(x, \omega_d, \omega_i)\, f_s(\omega_i, \omega_o) \approx d_b(x, \omega_d)\, f_{\text{back}}(\omega_i, \omega_o)\ . \quad (9)$$

Similar to forward scattering density factor d_f, backward scattering density factor d_b accounts for the hair density around the point x, and in practice we approximate it using a constant density term equal to d_f, which we set to 0.7. Note that f_{back} is not a function of x, which means that it is modeled as a material property. We formulate f_{back} as a product of an average backscattering attenuation function, $\bar{A}_b$, and an average spread function, $\bar{S}_b$, estimating the bidirectional multiple backscattering distribution function of a hair cluster as

$$f_{\text{back}}(\omega_i, \omega_o) = \frac{2}{\cos\theta}\ \bar{A}_b(\theta)\ \bar{S}_b(\omega_i, \omega_o)\ , \quad (10)$$

where $\theta = (\theta_o - \theta_i)/2$ is the difference angle of incident and outgoing inclinations and an additional $\cos\theta$ factor accounts for the fact that light is roughly scattered to a cone as in [Marschner et al. 2003]. To evaluate the effect of local multiple scattering we need to compute average backscattering attenuation $\bar{A}_b(\theta)$ and average backscattering spread $\bar{S}_b(\omega_i, \omega_o)$. Note that our formulation of Ψ^G and $\Psi^L f_s$ have a similar structure; however, these two expressions are conceptually different: Ψ^G models an angular radiance distribution whereas f_{back} serves as a curve scattering term (BCSDF).

3.2.1 Average Backscattering Attenuation

The average backscattering attenuation is computed for a point x inside a cluster of disciplined hair[1]. Since realistic hair models have strong similarity among neighboring hair strands, average attenuation computed for a disciplined hair cluster is a good approximation to be used in local multiple scattering. Furthermore, we ignore the slight change in the longitudinal inclination angle due to backward scattering events assuming that the absolute value of the longitudinal inclination $|\theta|$ is the same for all fibers in the cluster.

Local multiple scattering needs to account for the portion of light paths after they reach the neighborhood of point x. We consider only paths with an odd number of backward scattering events in this part of the light path. When there are an even number of backward scattering events in this part of the light path, the path points away from the light and does not return back to x; thus, such paths would not contribute to local multiple scattering. Average backscattering attenuation for all light paths that include a single backward scattering is

$$\bar{A}_1(\theta) = \bar{a}_b \sum_{i=1}^{\infty} \bar{a}_f^{2\,i} = \frac{\bar{a}_b\, \bar{a}_f^2}{1 - \bar{a}_f^2}\ , \quad (11)$$

where $\bar{a}_f(\theta)$ is the average forward scattering attenuation, and $\bar{a}_b(\theta)$ is the average backward scattering attenuation. It is computed similar to Equation 6 from fiber BCSDF for isotropic irradiance along the specular cone as

$$\bar{a}_b(\theta_d) = \frac{1}{\pi} \int_{\Omega_b} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} f_s\left((\theta_d, \phi), \omega\right) \cos\theta_d \,\mathrm{d}\phi\,\mathrm{d}\omega, \quad (12)$$

[1] All hair fibers of a disciplined hair cluster share the same tangent

where Ω_b is all directions over the back hemisphere. In other words, we consider all paths for which light forward scatters through i fibers, then backward scatters once, then again forward scatters all the way back through the same i fibers. Note that the special case with no forward scattering ($i = 0$) is not included here, since it is handled separately within the single scattering computation.

The average backscattering attenuation for light paths with three backward scattering is approximated by the analytical solution of the following triple sum:

$$\bar{A}_3(\theta) = \bar{a}_b^3 \sum_{i=1}^{\infty} \sum_{j=0}^{i-1} \sum_{k=j+1}^{\infty} \bar{a}_f^{2(i-j-1+k)} = \frac{\bar{a}_b^3\, \bar{a}_f^2}{\left(1 - \bar{a}_f^2\right)^3}\ . \quad (13)$$

Here, we consider each possible case where light forward scatters through i fibers, backscatters, forward scatters through $j < i$ fibers, backscatters, and forward scatters through $k > j$ fibers before backscattering one last time and forward scattering all the way back through the $i - j - 1 + k$ fibers again. Since $\bar{a}_b(\theta)$ is small for human hair fibers, we disregard the paths with more than 3 backward scattering events, approximating the average backscattering attenuation as the sum of equations 11 and 13

$$\bar{A}_b(\theta) = \bar{A}_1(\theta) + \bar{A}_3(\theta)\ . \quad (14)$$

3.2.2 Average Backscattering Spread

Similar to the forward scattering spread function in Equation 7, we represent backscattering spread as the product of a constant azimuthal term $\tilde{s}_b$ and a Gaussian function for the longitudinal spread as

$$\bar{S}_b(\omega_i, \omega_o) = \frac{\tilde{s}_b(\phi_i, \phi_o)}{\cos\theta}\ g\Big(\ \theta_o + \theta_i - \bar{\Delta}_b(\theta)\ ,\ \bar{\sigma}_b^2(\theta)\ \Big)\ , \quad (15)$$

where $\tilde{s}_b(\phi_i, \phi_o)$ is $1/\pi$ for backward scattering directions and zero for forward scattering, $\bar{\Delta}_b(\theta)$ is the average longitudinal shift caused by the scattering events, and $\bar{\sigma}_b^2(\theta)$ is the average longitudinal variance for backscattering.

As in the computation of average backscattering attenuation, we consider all possible paths with one and three backward scattering events, and we compute average longitudinal shift using a weighted average of shifts due to all possible light paths

$$\bar{\Delta}_b = \frac{\bar{a}_b}{\bar{A}_b} \sum_{i=1}^{\infty} \bar{a}_f^{2i}\left(2i\bar{\alpha}_f + \bar{\alpha}_b\right) + \frac{\bar{a}_b^3}{\bar{A}_b} \sum_{i,j,k} \bar{a}_f^m\left(3\bar{\alpha}_b + m\bar{\alpha}_f\right),$$

where $\sum_{i,j,k}$ denotes $\sum_{i=1}^{\infty}\sum_{j=0}^{i-1}\sum_{k=j+1}^{\infty}$, m is $2(i-j-1+k)$, and $\bar{\alpha}_f(\theta)$ and $\bar{\alpha}_b(\theta)$ are average forward and backward scattering shifts, taken from the BCSDF of the hair fiber. Similarly, the average backscattering standard deviation is computed as

$$\bar{\sigma}_b = \frac{\bar{a}_b}{\bar{A}_b} \sum_{i=1}^{\infty} \bar{a}_f^{2i} \sqrt{2\,i\bar{\beta}_f^2 + \bar{\beta}_b^2} + \frac{\bar{a}_b^3}{\bar{A}_b} \sum_{i,j,k} \bar{a}_f^m \sqrt{3\,\bar{\beta}_b^2 + m\bar{\beta}_f^2},$$

where $\bar{\beta}_f^2(\theta)$ and $\bar{\beta}_b^2(\theta)$ are average forward and backward scattering variances of the hair fiber BCSDF. In these formulations, we approximate the sum of the Gaussian functions of the individual scattering spreads by a single Gaussian function with a combined mean ($\bar{\Delta}_b$) and standard deviation ($\bar{\sigma}_b$). This is a good approximation since for realistic hair BCSDF, longitudinal shifts are small and individual scattering lobes have a comparable standard deviation. In practice, we use the following analytical approximations[2]

[2] These analytical approximations are numerical fits based on a power series expansion with respect to $\bar{a}_b$ up to an order of three.

to these sums given above:

$$\bar{\Delta}_b \approx \bar{\alpha}_b \left(1 - \frac{2\bar{a}_b^2}{(1-\bar{a}_f^2)^2}\right) + \bar{\alpha}_f \left(\frac{2(1-\bar{a}_f^2)^2 + 4\bar{a}_f^2\bar{a}_b^2}{(1-\bar{a}_f^2)^3}\right) \quad (16)$$

$$\bar{\sigma}_b \approx \left(1 + 0.7\bar{a}_f^2\right) \frac{\bar{a}_b\sqrt{2\bar{\beta}_f^2 + \bar{\beta}_b^2} + \bar{a}_b^3\sqrt{2\bar{\beta}_f^2 + 3\bar{\beta}_b^2}}{\bar{a}_b + \bar{a}_b^3\left(2\bar{\beta}_f + 3\bar{\beta}_b\right)}. \quad (17)$$

3.3 General Rendering Equation

The equations we derived up to here assume that the hair model is illuminated by a single directional light source such that ω_d is constant. However, our formulation can be extended to arbitrary light sources as long as estimating all global scattering based on the shadow path is still practicable. We can rewrite the general form of the rendering equation incorporating the terms for global and local multiple scattering as

$$L_o(x,\omega_o) = \int_\Omega L_i^G(x,\omega_i)\, f(\omega_i,\omega_o)\, \cos\theta_i \,\mathrm{d}\omega_i\,, \quad (18)$$

where

$$L_i^G(x,\omega_i) = \int_\Omega L_d(x,\omega_d)\, \Psi^G(x,\omega_d,\omega_i)\,\mathrm{d}\omega_d \quad (19)$$

is the globally multiple scattered light entering the hair volume from all directions before scattering towards x in direction ω_i, and

$$f(\omega_i,\omega_o) = f_s(\omega_i,\omega_o) + d_b f_{back}(\omega_i,\omega_o) \quad (20)$$

is the BCSDF including both single hair fiber scattering and backscattering from a collection of fibers. Hence, this general rendering equation can handle multiple light sources, general shaped area lights, image based lighting from all directions, and full global illumination solution including multiple scattering in hair.

4 Implementation

For an efficient implementation of the dual scattering approximation we rewrite the general rendering equation, changing the order of the integrals in equations 18 and 19 as

$$L_o(x,\omega_o) = \int_\Omega L_d(x,\omega_d)\, F(x,\omega_d,\omega_o)\,\mathrm{d}\omega_d\,, \quad (21)$$

where

$$F(x,\omega_d,\omega_o) = \int_\Omega \Psi^G(x,\omega_d,\omega_i)\, f(\omega_i,\omega_o)\, \cos\theta_i \,\mathrm{d}\omega_i\,. \quad (22)$$

Since $L_d(x,\omega_d)$ is known (incoming radiance), all we need to compute is the function $F(x,\omega_d,\omega_o)$ to find the outgoing radiance. This computation has two main steps: the first step gathers the necessary information to compute the global multiple scattering function $\Psi^G(x,\omega_d,\omega_i)$ and the second one is the shading step that computes the above integral.

4.1 Computing Global Multiple Scattering

The global multiple scattering function can be computed in many different ways, with the choice of method determining the speed and accuracy of the overall implementation. Since global multiple scattering is estimated by analyzing a single prototype light path (along the shadow path), all implementations of global multiple scattering are very similar to different semi-transparent hair shadowing techniques. However, unlike these non-physical shadowing approaches that aim to compute a transmittance (or an opacity) function along the shadow path, in our dual scattering method we compute front scattering transmittance $T_f(x,\omega_d)$ and front scattering variance $\bar{\sigma}_f^2(x,\omega_d)$, along with the fraction of direct illumination that reaches the point of interest without being blocked (shadowed) by other hair strands.

4.1.1 Ray Shooting

Ray shooting is the simplest implementation of the global multiple scattering function. The procedure is very similar to ray traced shadow computation. To find the forward scattering transmittance and spread at point x due to illumination from direction ω_d, we shoot a ray from x in the direction ω_d. If the ray does not intersect with any hair strands, T_f is taken as 1, $\bar{\sigma}_f$ is set to zero, and the direct illumination fraction becomes 1. If there is an intersection with a hair strand, the direct illumination fraction becomes zero and we update the transmittance and variance values using equations 5 and 8. While computing the transmittance (Equation 5) we use a one dimensional lookup table for $\bar{a}_f(\theta_d)$, which is precomputed by numerical integration of Equation 6. The ray shooting method provides an accurate dual scattering approximation, however multiple ray samples per pixel are needed to eliminate sampling noise.

4.1.2 Forward Scattering Maps

To accelerate the global multiple scattering computation we implemented a two pass approach. We begin by generating a voxel grid that encloses the hair geometry. Each voxel in this grid keeps T_f and $\bar{\sigma}_f^2$ values along with a direct illumination fraction. In the first pass we trace rays from each light source towards the grid, computing T_f and $\bar{\sigma}_f^2$ values along the ray. The values of each voxel are determined by the average of the values along the rays that intersect with the voxel. In the second pass we render the hair geometry and find the global multiple scattering values using linear interpolation from the voxel grid. In our implementation we also applied multi-sampling to further smooth the results.

Using forward scattering maps significantly improves the rendering time as compared to the ray shooting method. Furthermore, this voxel grid can keep global multiple scattering information of multiple light sources within the same data structure; therefore, the performance gain of forward scattering maps becomes more significant as the number of light sources increase. On the other hand, the accuracy and performance of forward scattering maps depend on the map resolution (i.e. the extent of a voxel). For all examples presented in this paper the map resolution is 0.5 cm.

4.1.3 GPU Implementation

In our GPU implementation we used a similar approach to forward scattering maps based on the deep opacity maps method [Yuksel and Keyser 2008] developed for computing semi-transparent hair shadows. We chose the deep opacity maps method over other possible alternatives, since it generates high quality results with minimum computation and memory cost.

Similar to deep opacity maps, in the first pass we render a depth map from the light's point of view. This depth map is used in the second pass to shape the map layers conforming to the shape of the hair style. Therefore, the hair volume illuminated by the light source can be accurately sampled using a very small number of layers. In the third and final pass, we render the final hair image using the map to find the global multiple scattering values.

Instead of keeping a single opacity value for each map pixel as suggested by the deep opacity maps method, we store 7 values: T_f and $\bar{\sigma}_f$ values for each RGB color channel and the direct illumination fraction. Therefore, for every layer we need two textures to store all 7 values. Due to the efficiency of the deep opacity maps approach, we can achieve high quality results with as few as 4 layers, which are stored in 8 textures. The latest consumer graphics cards support generation of all these 8 textures in a single pass using multiple draw buffers. However, in our (earlier generation) GPU implementation we output 4 textures per pass and generate our maps using two passes (one additional pass as compared to the original deep opacity maps implementation).

4.2 Shading Computation

Once the global multiple scattering information is gathered, the shading step computes the rest of the integral in Equation 22. This step is the same for all implementations of dual scattering. We compute Equation 22 differently depending on whether or not the point x receives illumination directly without being blocked (shadowed) by other hair strands, which is determined by the direct illumination fraction value of the global multiple scattering information.

When point x receives illumination directly (i.e. no scattering events occur along the shadow path) the integral in Equation 22 evaluates to $f(\omega_d, \omega_o)\cos\theta_d$. Here $f(\omega_d, \omega_o)$ is the sum of the single scattering component and the average backscattering component f_{back} as given in Equation 20. The computation of the single scattering component is identical to [Marschner et al. 2003] and [Zinke et al. 2004], such that the longitudinal scattering function M, which is modeled as a Gaussian function, is multiplied by a precomputed azimuthal scattering function N. The backscattering component f_{back} is computed from Equation 10 using precomputed tables for $\bar{A}_b(\theta)$, $\bar{\Delta}_b(\theta)$, and $\bar{\sigma}_b^2(\theta)$.

When point x is blocked by other hair strands, it receives illumination via global multiple scattering. In this case the azimuthal forward scattering spread on x becomes isotropic and the longitudinal forward scattering spread is represented by a narrow Gaussian function as given in Equation 7. For efficient computation of Equation 22, instead of using numerical integration techniques, we approximate the result by combining azimuthal and longitudinal components of S_f and f. The combination of longitudinal components can be easily approximated for narrow Gaussian functions by using combined variances β_s^2 and β_{back}^2 for f_s and f_{back} respectively, such that

$$\beta_s^2(x, \omega_d, \omega_i) = \bar{\sigma}_f^2(x, \omega_d) + \beta^2(\theta)\ , \text{and} \quad (23)$$

$$\beta_{\text{back}}^2(x, \omega_d, \omega_i) = \bar{\sigma}_f^2(x, \omega_d) + \bar{\sigma}_b^2(\theta)\ , \quad (24)$$

where $\beta^2(\theta)$ is the variance of the scattering lobe of the BCSDF[3]. In the azimuthal direction, f_{back} is already isotropic; therefore, isotropic forward scattering does not affect this component. However, the azimuthal component N of f_s changes under isotropic azimuthal irradiance. We precompute this component N^G and store it in a 2D table similar to N using numerical integration of

$$N^G(\theta, \phi) = \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \tilde{s}_f(\phi)\, N(\theta, \phi')\, \mathrm{d}\phi' \ . \quad (25)$$

Note that this formulation ignores the elipticity of hair fibers. However, this approximation is still accurate enough even for elliptic hair fibers, since $\tilde{s}_f = 1/\pi$ is isotropic (constant) and N^G is averaging f_s along the front half cone.

[3]The BCSDF has three lobes, one for each component: R, TT, and TRT

Table 1: *Precomputed tables used for shading*

Function	Reference	Use
$\bar{A}_b(\theta)$	Equation 14	local multiple scattering
$\bar{\Delta}_b(\theta)$	Equation 16	local multiple scattering
$\bar{\sigma}_b^2(\theta)$	Equation 17	local multiple scattering
$N^G(\theta, \phi)$	Equation 25	BCSDF due to forward scattering

$\mathbf{F}(T_f, \bar{\sigma}_f^2, \text{directFraction})$

// Compute local multiple scattering contribution

$f_{\text{back}} \leftarrow 2\,\bar{A}_b(\theta)\, g(\theta_d + \theta_o - \bar{\Delta}_b(\theta), \bar{\sigma}_b^2(\theta) + \bar{\sigma}_f^2)/(\pi\,\cos^2\theta)$

// Compute BCSDF of the fiber due to direct illumination

$M_{\text{R,TT,TRT}} \leftarrow g(\theta - \alpha_{\text{R,TT,TRT}}, \beta_{\text{R,TT,TRT}}^2)$

$f_s^{\text{direct}} \leftarrow M_{\text{R}} N_{\text{R}}(\theta, \phi) + M_{\text{TT}} N_{\text{TT}}(\theta, \phi) + M_{\text{TRT}} N_{\text{TRT}}(\theta, \phi)$

$F^{\text{direct}} \leftarrow \text{directFraction}\left(f_s^{\text{direct}} + d_b\, f_{\text{back}}\right)$

// Compute BCSDF of the fiber due to forward scatterd illumination similarly

$M_{\text{R,TT,TRT}}^G \leftarrow g(\theta - \alpha_{\text{R,TT,TRT}}, \beta_{\text{R,TT,TRT}}^2 + \bar{\sigma}_f^2)$

$f_s^{\text{scatter}} \leftarrow M_{\text{R}}^G N_{\text{R}}^G(\theta, \phi) + M_{\text{TT}}^G N_{\text{TT}}^G(\theta, \phi) + M_{\text{TRT}}^G N_{\text{TRT}}^G(\theta, \phi)$

$F^{\text{scatter}} \leftarrow (T_f - \text{directFraction})\ d_f\left(f_s^{\text{scatter}} + \pi d_b\, f_{\text{back}}\right)$

// Combine direct and forward scattered components

return $(\ F^{\text{direct}} + F^{\text{scatter}}\ )\cos\theta_i$

Figure 5: *Pseudo code of our dual scattering shader. α and β values are measured hair characteristics (part of the BCSDF) and computation of M and N is discussed in Section 2.2. Other precomputed tables are given in Table 1.*

To accelerate most computations we use small precomputed tables that store complicated hair fiber properties. These tables are listed in Table 1. The precomputations are simply numerical integrations of the given equations. In addition to these tables, we use a precomputed table for the azimuthal part of single fiber scattering, $N(\theta, \phi)$, which can be computed using techniques described in [Marschner et al. 2003] or [Zinke and Weber 2007].

Figure 5 shows the pseudo code for computing Equation 22 using this procedure. This pseudo code is a simple extension to the BCSDF shading computation; therefore, our method can be easily integrated into existing physically based hair rendering systems.

5 Results

To test the validity of our simplifications, we compared the results of our dual scattering approximation directly to path tracing. Figure 6 shows such a comparison for a disciplined cluster of blond hair, where the single scattering component is excluded to compare the results of multiple scattering only. As can be seen from these images, our approximation generates very similar results to path tracing regardless of the incident light direction. Although there are subtle differences compared to the results obtained with unbiased path tracing, the general look is very similar and irregularities that occur in real hairstyles tend to conceal such errors.

The graph in Figure 7 shows the comparison of radiance values in the middle of the clusters in Figure 6. Here, the average BRDF of multiple scattered light is shown for longitudinal and azimuthal sweeps along the cluster for various absorption coefficients. As can be seen from this graph, the azimuthal component of backward scattering (white region of the left graph) is almost isotropic, while the longitudinal component resembles a Gaussian distribution as modeled in our formulation. Even though the approximation slightly overshoots the accurate simulation for very low coefficients, the results indicate that our approach is a viable approximation for scattering from a cluster.

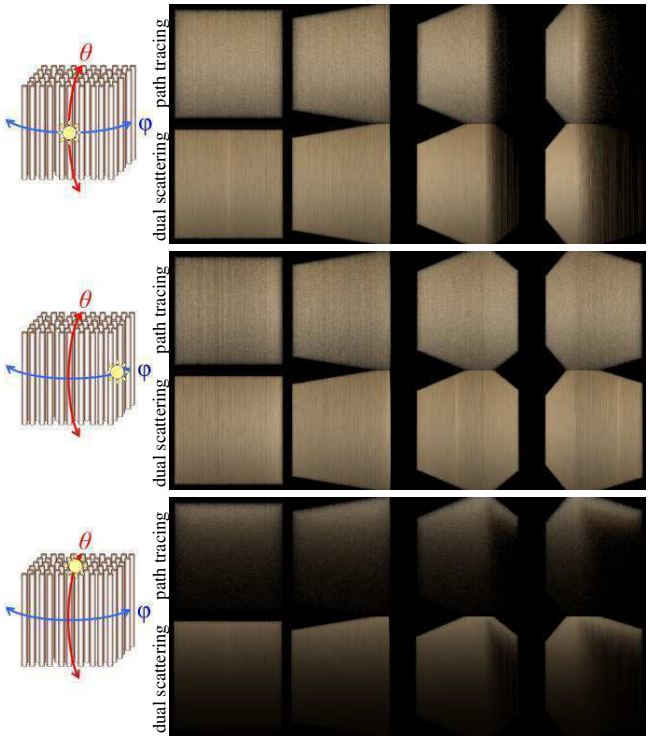


Figure 6: *Comparison (multiple scattering only) of dual scattering using ray shooting to path tracing for a cube shaped disciplined hair cluster consisting of 64K hair fibers illuminated by a single directional light source (light direction is shown on the left).*

We also tested the accuracy of our dual scattering method for different hair fiber scattering properties. Figure 8 and Figure 9 show that dual scattering provides a good approximation for various hair colors and longitudinal Gaussian widths (standard deviations). Slight changes of the optical properties of a fiber (such as absorption) may have a drastic impact on the overall hair color. As can been in these figures, our approximation handles such variations correctly.

Despite the common intuition, it is more difficult to perceive errors in complicated models (where detail can hide problems), while even the slightest errors show up on simple models. We show a complicated hair model example in Figure 10. Since dual scattering only exploits local symmetry in the hair model, most of our assumptions still hold and we can produce a close approximation to the path tracing reference with significantly improved rendering times.

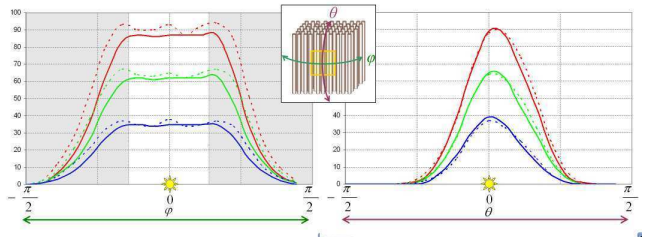


Figure 7: *The average BRDF (multiple scattering only) at a blond hair cluster computed for the red, green and blue components of light blond hair (see Figure 6) using path tracing (solid lines) and dual scattering approximation (dashed lines). The scene is illuminated by a directional front light source which means that for $\phi \approx 0$ (white area) the results are dominated by purely local multiple scattering (f_{back}) since no visible fibers are shadowed.*

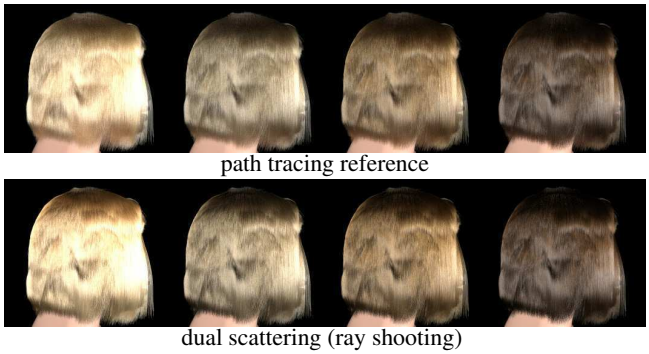


Figure 8: *Comparison for various hair colors with RGB absorption coefficients (from left to right) (0.03,0.07,0.15), (0.15,0.2,0.3), (0.2,0.3,0.5), and (0.3,0.6,1.2).*

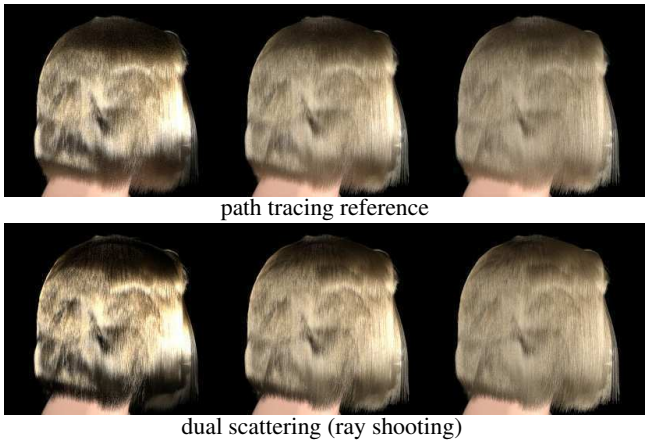


Figure 9: *Comparison for varying longitudinal widths β_R, β_{TT}, and β_{TRT} of the BCSDF (from left to right) (4, 5, 7.5), (8, 10, 15), (16, 20, 30)*

Figure 11 compares the offline implementations of dual scattering to path tracing and single scattering as well as to photon mapping (using ray based global illumination [Zinke and Weber 2006]) techniques. As can be seen from these images, while single scattering produces a dark image that fails to reproduce the correct color of hair, all other methods produce similar results to path tracing, while offline implementations of dual scattering significantly improve the rendering speed.

Figure 12 presents captured frames from our GPU based implementation of dual scattering, and comparisons to offline dual scattering (ray shooting) and path tracing. The figure shows three scattering components (single, global multiple, and local multiple) that produce the full dual scattering solution when combined. For this hair style the performance of the GPU implementation of dual scattering (12 fps) is comparable to the performance of a deep opacity maps [Yuksel and Keyser 2008] implementation (18 fps). Using extended multiple draw buffers, the additional pass we use for GPU-based dual scattering can be eliminated, which would significantly reduce the performance gap between our method and existing non-physical real-time solutions.

6 Discussion and Conclusion

Dual scattering offers a physically based simplification to the complicated phenomenon of multiple scattering in hair, exploiting scattering properties of hair fibers (such as narrow scattering along longitudinal directions and wide scattering along azimuthal directions) and general characteristics of human hair models (such as local sim-

Single Scattering Only (offline) 3 minutes	Path Tracing Reference (offline) 22 hours	**Dual Scattering** (ray shooting) 9.6 minutes	**Dual Scattering** (forward scattering map) 4.6 minutes	**Dual Scattering** (GPU-based) 5.8 fps

Figure 10: *Comparison of our dual scattering approximation method to path tracing ("ground truth") for a complicated hair style with 50K strands and 3.4M segments. The hair model is illuminated by two light sources, one of which act as a back light.*

0.0 0.1 0.2 0.3 0.4 0.5 **0.6** **0.7** **0.8** 0.9 1.0

Figure 13: *The effect of forward and backward scattering density factors $d_f(x, \omega_d)$ and $d_b(x, \omega_d)$. Values of 0.0 are equivalent to single scattering only. For most human hair styles values between 0.6 and 0.8 generate close approximations to path tracing.*

ilarity of fiber directions). By splitting multiple scattering into a local and a global part and using different approximations regarding the directionality of scattering from hair fibers, our dual scattering approach is orders of magnitude faster than other accurate techniques. We justify our simplifications by not only the theory but also several comparisons to ground truth (path tracing) in both experimental setups and realistic cases.

It is important to note that all of the computations described rely on physically-based values. All parameters are either fundamental to the virtual scene (e.g. directions such as ω_i), or are characteristics of real hair that can be accurately measured and described (e.g. the α and β values, and the BCSDF description). The only "user adjustable" term is the density factor (d_f and d_b), but even this has a physical meaning that limits the range of choices and, in theory, it could be computed precisely. Figure 13 shows the effect of changing density parameters.

The multiple scattering computation simplifications introduced in this paper are based on theoretical approximations rather than ad-hoc appearance-based formulations. As a result, despite the aggressive simplifications we have made, we obtain close approximations with no parameter adjustment and much faster computations.

On the other hand, one can come up with special hair models that would break some of our assumptions. For instance, if the mean path length is large (as in sparse hairstyles) or if attenuation coefficients are very small, the global structure of a hairstyle tends to play an important role that biases the results. Furthermore, the assumption that neighboring hair strands exhibit a similar structure can be violated in chaotic hair models. However, even for complicated cases our results look plausible and close to reference images with significantly improved computation times.

One limitation of our formulation arises from the fact that we use the shadow path as a prototype for all significant multiple scattered paths. Therefore, when there is a strong spatial variation in illumination, this prototype path assumption may be violated. For example, a hard shadow edge falling across the hair creates a sharp illumination change, hence our prototype path approximation would be less accurate along the shadow boundary.

We believe that separating local and global multiple scattering is a very general principle that is applicable not only in the realm of hair rendering, but also for other highly scattering quasi-homogeneous structures such as snow, clouds or woven textiles.

Acknowledgements

We would like to thank Murat Afsar for the head model, and Anton Andriyenko for the hair model in Figures 8, 9 and 11. We would also like to thank the anonymous reviewers for their helpful comments. This work is supported in part by *NSF* grant CCR-0220047 and *Deutsche Forschungsgemeinschaft* under grant We1945/3-2.

References

BERTAILS, F., MÉNIER, C., AND CANI, M.-P. 2005. A practical self-shadowing algorithm for interactive hair animation. In *Graphics Interface*, 71–78.

GUPTA, R., AND MAGNENAT-THALMANN, N. 2005. Scattering-based interactive hair rendering. In *Comp. Aided Design and Comp. Graphics*, 489–496.

HADWIGER, M., KRATZ, A., SIGG, C., AND BÜHLER, K. 2006. Gpu-accelerated deep shadow maps for direct volume rendering. In *Proceedings of Graphics Hardware 2006*, 49–52.

KIM, T.-Y., AND NEUMANN, U. 2001. Opacity shadow maps. In *Eurographics Rendering Workshop*, 177–182.

KNISS, J., PREMOZE, S., HANSEN, C., SHIRLY, P., AND MCPHERSON, A. 2003. A model for volume lighting and modeling. *IEEE Trans. on Vis. and Comp. Graphics 9*, 2, 150–162.

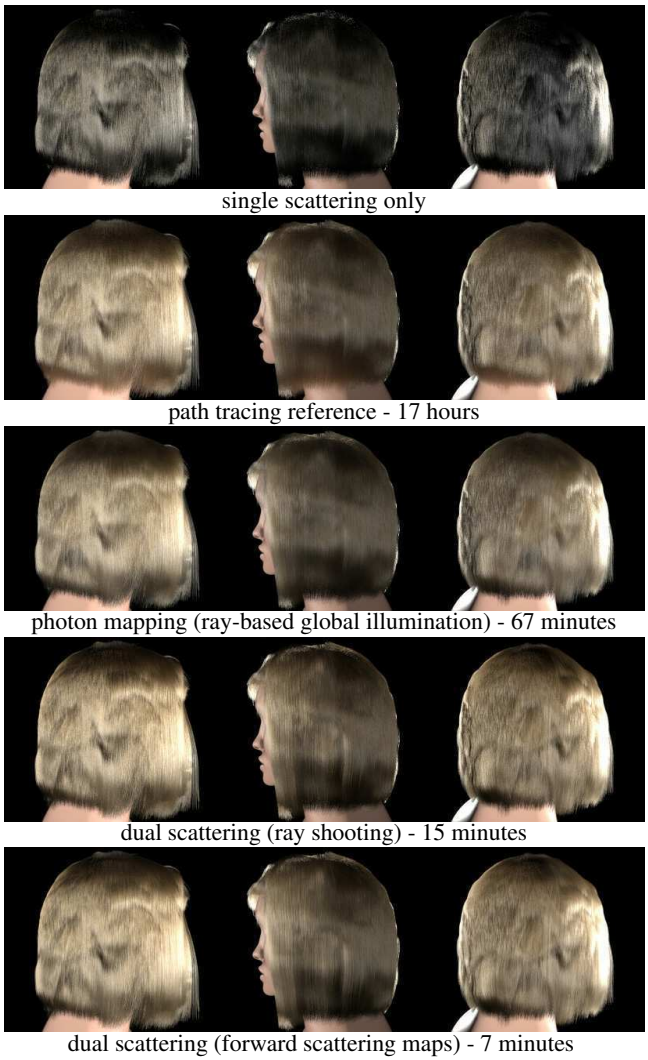


Figure 11: *Comparison of a blond hairstyle (87K strands) viewed from three different perspectives (illuminated by three directional light sources).*

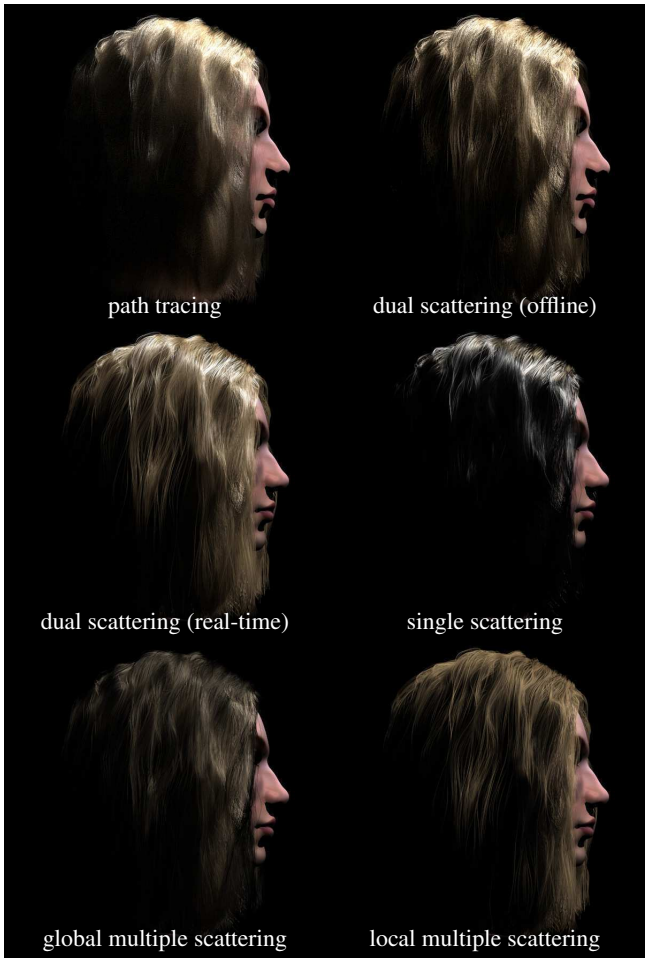


Figure 12: *Components of dual scattering method captured from our real-time implementation (12 fps) and comparisons to offline dual scattering using ray shooting (11.2 minutes) and path tracing reference (11.8 hours). The hair model has 50K strands and 2.4M line segments.*

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

MARSCHNER, S. R., JENSEN, H. W., CAMMARANO, M., WORLEY, S., AND HANRAHAN, P. 2003. Light scattering from human hair fibers. *ACM Transactions on Graphics 22*, 3, 780–791. SIGGRAPH 2003.

MERTENS, T., KAUTZ, J., BEKAERT, P., AND REETH, F. V. 2004. A self-shadow algorithm for dynamic hair using density clustering. In *Eurographics Symposium on Rendering*, 173–178.

MOON, J. T., AND MARSCHNER, S. R. 2006. Simulating multiple scattering in hair using a photon mapping approach. *ACM Transactions on Graphics 25*, 3, 1067–1074. SIGGRAPH 2006.

PREMOZE, S., ASHIKHMIN, M., RAMAMOORTHI, R., AND NAYAR, S. 2004. Practical rendering of multiple scattering effects in participating media. In *Eurographics Symp. on Rendering*.

WARD, K., BERTAILS, F., KIM, T.-Y., MARSCHNER, S. R., CANI, M.-P., AND LIN, M. 2007. A survey on hair modeling: Styling, simulation, and rendering. *IEEE Transactions on Visualization and Computer Graphics 13*, 2, 213–34.

XU, S., LAU, F. C., JIANG, H., AND PAN, Y. 2006. A novel method for fast and high-quality rendering of hair. In *Proc. of the 17th Eurographics Symp. on Rendering*, 331–341, 440.

YUKSEL, C., AND KEYSER, J. 2008. Deep opacity maps. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2008) 27*, 2.

YUKSEL, C., AKLEMAN, E., AND KEYSER, J. 2007. Practical global illumination for hair rendering. In *Pacific Graphics 2007*, 415–418.

ZINKE, A., AND WEBER, A. 2006. Global illumination for fiber based geometries. In *Electronic proceedings of the Ibero American Symposium on Computer Graphics (SIACG 2006)*.

ZINKE, A., AND WEBER, A. 2007. Light scattering from filaments. *IEEE Trans. on Vis. and Comp. Graphics 13*, 2, 342–356.

ZINKE, A., SOBOTTKA, G., AND WEBER, A. 2004. Photorealistic rendering of blond hair. In *Vision, Modeling, and Visualization (VMV) 2004*, 191–198.