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

9:00 am **Introduction and Fundamentals of CG Hair** [Yuksel]

9:10 am **Data Management and Rendering Pipeline** [Yuksel and Tariq]
- Overview
- Rendering Hair
- Efficiently Sending Data to GPU
- Generating Hair on the GPU
- Final Details
- Hair Meshes

9:40 am **Transparency and Antialiasing** [Yuksel and Tariq]
- Overview
- Antialiasing
- Transparency and Depth Sorting
- Avoiding Sorting for Transparency

9:55 am **Hair Shading** [Yuksel]
- Simple Hair Shading
- Physically-based Hair Shading
- Improving Shading Performance on the GPU

10:30 am **Break**

10:45 am **Hair Shadows** [Yuksel and Tariq]
- Shadow Maps
- Transparent Shadow Mapping for Hair
- Shadow Filtering
- Simplified Shadow Maps for High Performance

11:20 am **Multiple Scattering in Hair** [Yuksel]
- Introduction to Multiple Scattering
- Dual Scattering Approximation
- Implementation Notes

11:35 am **Hair Dynamics for Real-time Applications** [Yuksel and Tariq]
- Overview of Hair Simulation Techniques
- Fast Simulation of Hair on the GPU
- Handling Inter Hair Collisions

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

Introduction

Advanced Techniques in Real-time Hair Rendering and Simulation
SIGGRAPH 2010 Course

Cem Yuksel
Cyber Radiance and Texas A&M University

Sarah Tariq
NVIDIA

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

Advanced Techniques in Real-time Hair Rendering and Simulation
SIGGRAPH 2010 Course

Sarah Tariq
NVIDIA

Cem Yuksel
Cyber Radiance and Texas A&M University

Image from the game Heavenly Sword

What is the difference between these two images?

What is the difference between these two images, besides the fact that one of them is a character from a real game?

The main difference that I see in the hair is the amount of geometric complexity. As we move towards achieving more and more realism in real time, one of the most important obstacles that we have to overcome is how to efficiently render lots of geometry.

We know how to efficiently render hair like the one on the left - we represent the hair by a couple of hundred texture mapped triangles, and rendering this amount of geometry efficiently is very straight forward on a GPU. We could, for example, upload the entire hair style to the GPU, modify the vertex positions in the vertex shader to animate the hair, and then render.

However, if we want to render a lot more hair strands, for example instead of a few hundred triangles we want to render almost 3 million triangles, like for the image on the right, then we have to think differently and carefully about how to be efficient.

These are the questions we are going to be talking about in this section – if our goal is to render relatively large amounts of hair geometry then how do we represent the data and how do we render it, in order to get the best performance?

Data Management and Rendering Pipeline

- Rendering Hair
- Managing Dynamic Hair Data
- Efficiently Sending Data to GPU
- Generating Hair on the GPU
- Final Details
- Hair Meshes

Rendering Hair as PolyLines

- Advantages of Lines
  - Faster to render lines than camera facing triangles
  - Easy to implement
- Issues with lines:
  - Lines only support a 1D texture along their length, but you cannot map a texture across their width
  - Disconnecting different strands in a single draw call is not trivial

The most obvious way to render hair strands is to render them as lines – lines are easy to implement and fast to render.

However, lines have a number of restrictions.

For example, you can have a 1D texture on a line, but not a 2D texture which is more versatile. A 2D texture would allow texture variation across the width of the line.

In addition, if you want to render all the hair strands in one call, which is the efficient thing to do, then you have to get rid of the line segments that connect the end of one hair strand to the beginning of the next hair strand. You cant get rid of these connecting lines by setting their width to zero, so you either have to use the Geometry Shader to kill these segments, or you have to introduce explicit line segments connecting these strands and then set their alpha to zero.

Rendering Hair as PolyLines

- Lines have integer width
  - number of pixels on screen
  - each draw call is limited to a single integer
- Why do we need varying widths?
  - Change line width based on distance to camera
  - taper width near the tips of the hair

Lines can only have fixed integer width, leading to

Hair looking too thin when zoomed in

And too thick when zoomed out

Another issue with lines is that we can only have an integer width for a line, which is specified in number of pixels on screen, and this width has to be set per draw call.

So, if you would like to have different line segments with different widths you will have to have that many different draw calls, and increasing draw calls reduces performance.

In addition, you lose some programmability by having to specify the width per draw call, rather than dynamically in the shader.

There are a couple of reasons why varying widths are useful – first of all, you need to vary hair width as the hair moves closer or further from the camera, otherwise the hair will appear too sparse when it is near the camera, and too thick when its far away. In addition, variable width is useful for allowing hair to taper near its ends, and in implementing LOD.

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, we expand it into two triangles that are facing the camera.

This expansion is pretty easy, and can be done for example 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.

There are times where you might not want to use the Geometry Shader however, maybe because it is not available in the GPU you are using, or because enabling it causes a performance degradation, which would happen with some earlier GPUs. In this case camera facing quads can be rendered by rendering degenerate triangles and "expanding" them by using the Vertex Shader. This approach has a couple of drawbacks, including an increase in the amount of computation that the GPU has to do (we introduce redundant duplicated calculations).

Rendering Hair as Camera Facing Quads

- Disadvantages:
  - Rendering triangles is more expensive
  - Code is a more complicated
- 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

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.

Data Management and Rendering Pipeline

- Rendering Hair
- Managing Dynamic Hair Data
- Efficiently Sending Data to GPU
- Generating Hair on the GPU
- Final Details
- Hair Meshes

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
  - Simulation: Typical human head has ~100,000 strands – simulating all the vertices on all these strands is too much computation
  - Rendering: Even if we are not simulating hair, it is faster to upload to the GPU only a subset of the hair and create the rest of the hair directly on the GPU

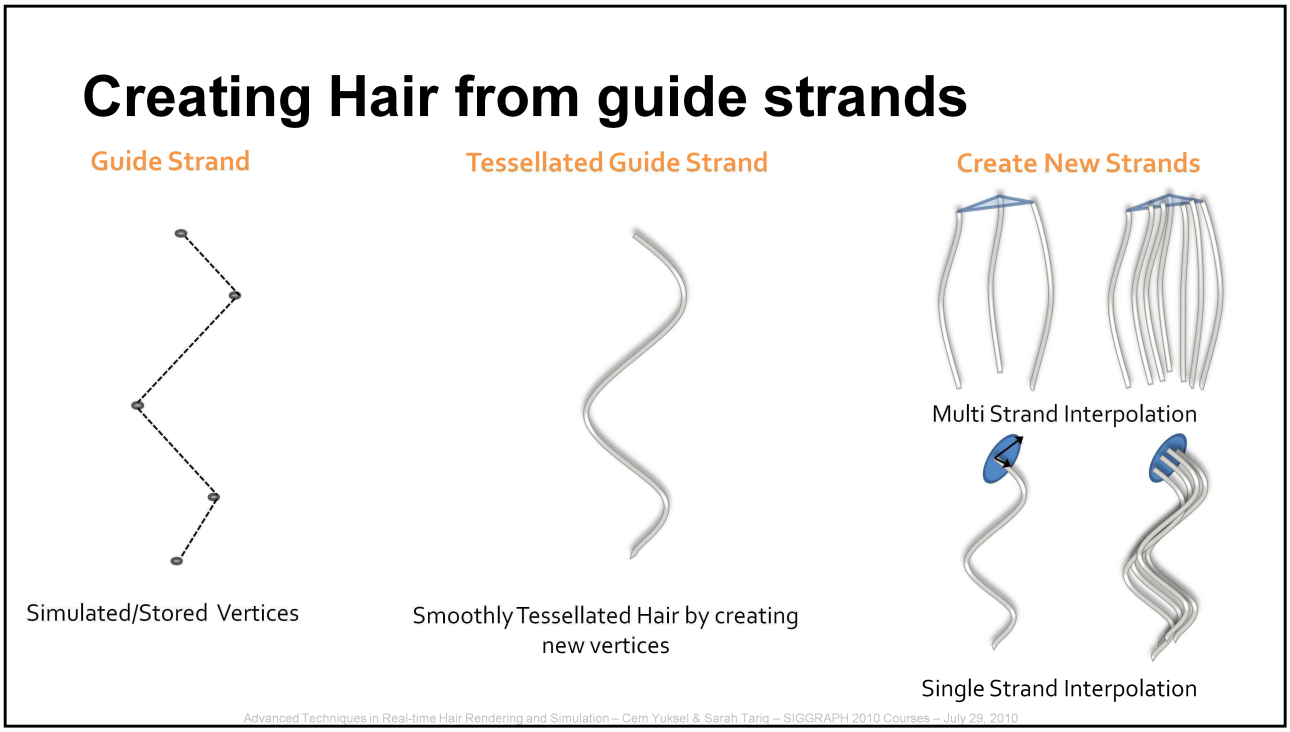
A typical human head has around a 100,000 strands – rendering and simulating all these strands is a lot of computation, especially for real time.

Many of the methods used in literature and practice simulate hair dynamics for only a subset of the hair, which we will call 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.

This approach has a number of advantages. Firstly, simulating every last hair can be time consuming - it is better to use the expensive simulation computations for only a small subset of the strands and use simple interpolation to generate the final strands.

Secondly, in most cases it is faster to upload a small amount of data to the GPU each frame and use the on-chip processing power of the GPU to recreate the rest of the Hair. GPUs are great at doing large amounts of computation, and not so fast at transferring data back and forth from the CPU.

By simulating only a subset of the hair we save substantial cost of simulation. In addition, we get a speedup by uploading only this subset of hair from the CPU to the GPU, and letting the GPU create the rest of the hair.



Here is an overview of the hair rendering that we are going to be talking about in the rest of this section. We start with guide strands which are just sets of simulated vertices. Next these strands are tessellated to make them smooth. We then create many more hair from these smooth guide strands by interpolation.

There are two types of interpolation that we will be talking about.

The first is Multi Strand Interpolation, where we average the attributes of many guide strands in order to create the attributes of a rendered strand.

The second type is Single Strand Interpolation, where attributes of only a single guide stand are used to create new rendered strands.

Multi Strand Interpolation

- Each interpolated strand is created by linearly interpolating the attributes of multiple guide strands
- Example use case, 3 guide strands:
  - The guide strands for a given interpolated strand are rooted at the vertices of the scalp triangle where the interpolated strand is created

Locations of desired rendered strands along with the three guide strands which will be used to create them

Lets start with Multi strand interpolation. In this approach a final rendered strand is computed by a weighted average of multiple guide strands.

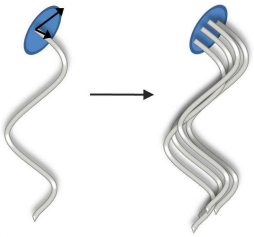
The example that we are going to talk about uses three guide strands. We define the scalp as the triangulation of the roots of these guide strands.

Lets assume that we are given a set of locations on the scalp that we want to create rendered hair on. These locations can for example be driven by a density map that an artist creates, specifying how much hair he wants in particular areas. For each location we find which scalp triangle it falls in, and then we determine its barycentric coordinates for the scalp triangle and the three guide strands that are rooted at the vertices of the triangle. We can then compute the attributes of the rendered strand by computing a weighted sum of the three guide strand attributes with the barycentric coordinates.

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)



Single Strand Interpolation uses only a single strand's attributes in creating its rendered child strands. In our example we are going to be offsetting rendered strands from the guide strand vertices along their coordinate frames.

Given a smooth guide strand with a consistent coordinate frame defined along the vertices we make an interpolated strand by assigning to it a single float2 offset and then offsetting the guide strand's vertex locations along their coordinate frames by this offset.

Interpolating to create more Strands

Multi strand Interpolation Single Strand Interpolation Combination

Here is what the two different interpolation modes look like. Multi Strand Interpolation provides a smooth even coverage of the scalp, while Single Strand interpolation produces clumps which are also an important part of the look of most hair. Depending on the type of hair desired we can use more or less of each type of interpolation.

Data Management and Rendering Pipeline

- Rendering Hair
- Managing Dynamic Hair Data
- Efficiently Sending Data to GPU
- Generating Hair on the GPU
- Final Details
- Hair Meshes

Efficiently Sending Data to the GPU

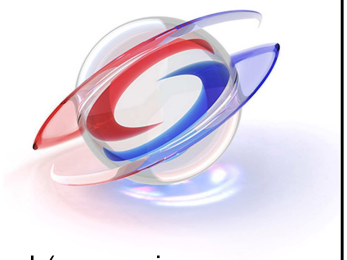
- Goal
  - use a small set of guide strands to create many more interpolated strands on the GPU
- How do we create these interpolated strands?
  - Can use tessellation hardware feature of latest generation of GPUs
  - No way to actually *create* massive amounts of geometry on earlier GPUs
    - Cannot use the Geometry Shader for creating massive amounts of hair, since it will be extremely inefficient!
    - Geometry Shader is only meant for relatively small amounts of data expansion.

As we talked about earlier, we are only going to upload to the GPU a small number of guide strands and then use interpolation to create the entire set of rendered hair on the fly. How do we actually create the interpolated strands on the GPU?

The latest generation of GPUs have a new feature called tessellation which can be used for this, and we will talk about later.

However, earlier generations of GPUs didn't have a way to *create* massive amounts of geometry on the GPU. We could try using the Geometry Shader for this, but that is almost certainly a bad idea. The Geometry Shader is only meant for relatively small amounts of data expansion, like expanding a point to a quad, and gets pretty inefficient for large amounts of expansion. This performance degradation is by design – this shader stage is meant for modest expansion and so the hardware is engineered to only make that use case fast.

Interpolating Hair on the GPU

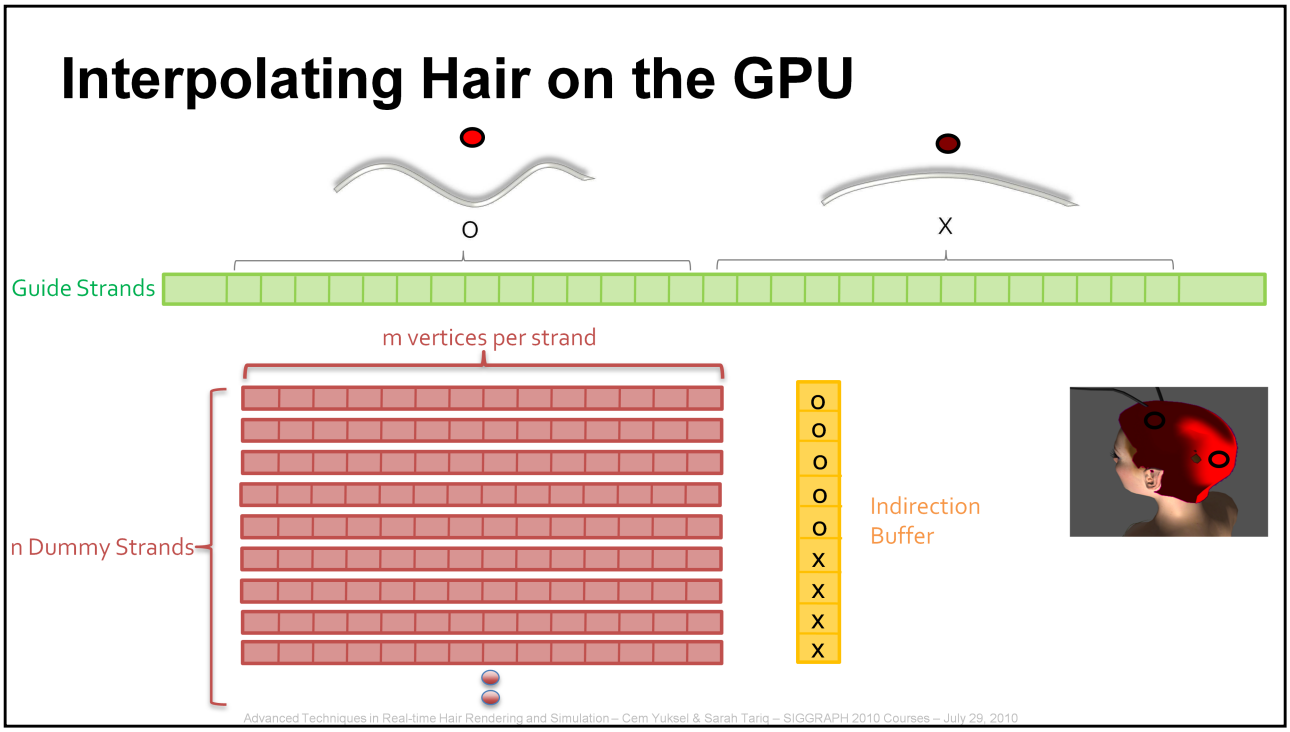


- Render dummy hair strands to the GPU
  - Render for example a line strip with m*n vertices per strand (m=maximum number of vertices in a hair strand, n=number of strands)
    - `pd3dContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP);`
    - `pd3dContext->Draw(maxVerticesPerStrand * numStrands, 0);`
  - These strands have no real attributes, so rendering them is reasonably fast
- Evaluate vertex attributes in GPU
  - load appropriate guide strand attributes from Buffers/Textures in the Vertex Shader
  - Interpolate between these attributes to determine the attributes of the rendered vertex

So, lets start with how we would render all these interpolated strands on these earlier GPUs.

In order to render the interpolated strands we have to send some representation of this geometry to the GPU, but we would like to keep it to a minimum, because as we talked about earlier, sending data between the CPU and GPU is not so fast. So, we render dummy vertices to the GPU. This means we just tell the GPU to render for us a certain amount of vertices, but we don't bind any vertex buffer or index buffer. The amount of vertices we render is the maximum number of vertices in a strand, multiplied by the total number of strands that we want to render.

We can compute the actual attributes of each vertex in the Vertex Shader, by loading the guide strand data that we need from textures and interpolating.



Here is a diagrammatic representation of how this works.
We upload to the GPU as a texture the guide strand attributes that we need, for example positions.
We then render to the GPU "dummy strands".
Each vertex has certain system generated values that we can use. For example, if we render the dummy strands as a line strip of n*m vertices, then in the Vertex Shader we can identify the particular vertex and hair strand by using its "vertex ID".
Once we know the id of the rendered strand and the id of the particular vertex within that strand that we are trying to render we can look up the attributes of the particular guide strands that we want to interpolate. In order to find the correct offsets into the guide strand texture or buffer we use an indirection texture, which basically tells us that given the id of the rendered strand and vertex we are trying to process, what is the offset of the guide strand or strands that it has to interpolate.

Once we compute the attributes of each vertex in the hair strand we can optionally expand the strands into camera facing quads by using the Geometry Shader.

Interpolating Hair on the GPU

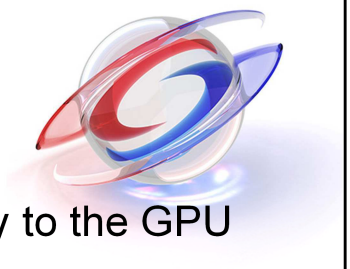
- We can optionally use “Stream Output” to store the data to memory between stages, for example:
  - Tessellate the simulated strands and Stream out
  - Interpolate the tessellated strands and Stream out
  - Render final hair for shading to shadow map
  - Render final hair for rendering
- Each stage uses data computed and streamed out from previous stage

There are a couple of different steps that we outlined for rendering the hair. We tessellate the guide strands to create smooth strands, we interpolate between the guide strands, we render the final hair to the shadow buffer and then to the camera. We can either perform all these operations any time we want to render the hair, or optionally we can save the intermediate results from these steps using Stream Out, and the resultant re-use can help increase performance. For example, if we don’t have a lot of guide strands it makes sense to tessellate them once and save them to the frame buffer, and then reuse these tessellated strands when we are trying to create the interpolated strands.

Data Management and Rendering Pipeline

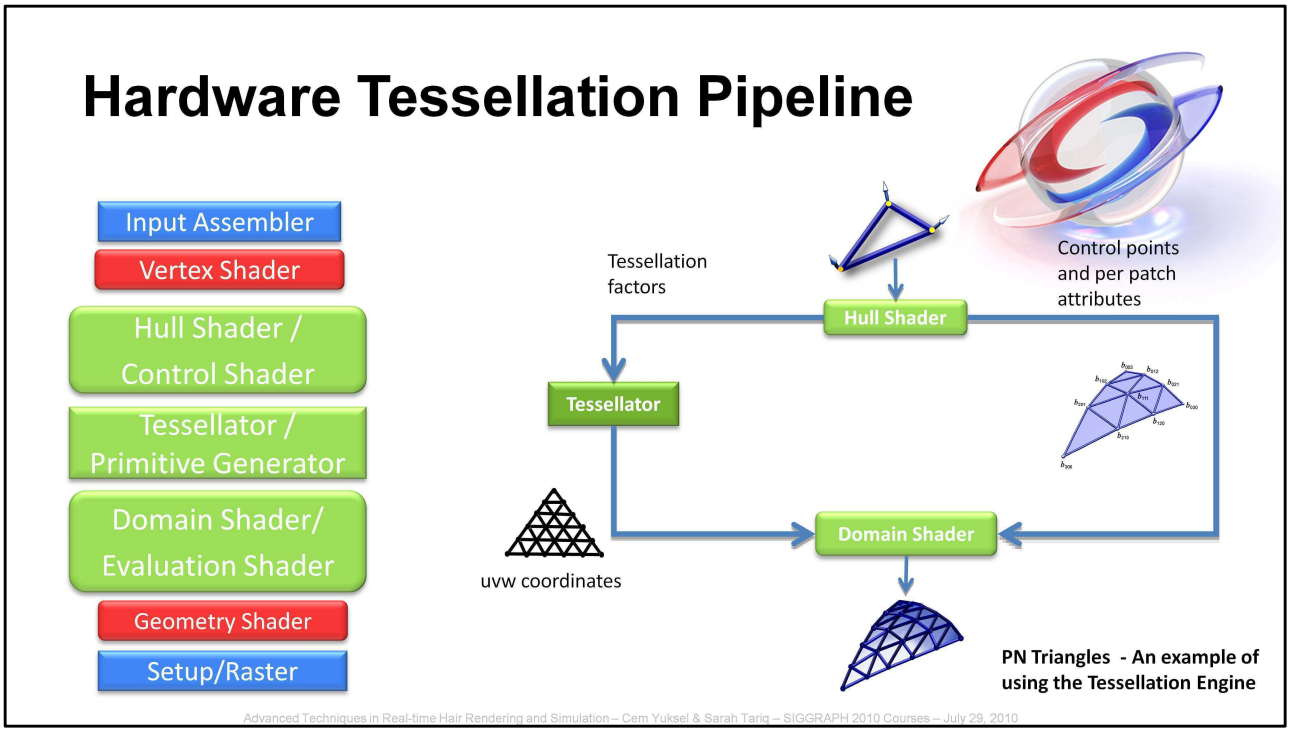
- Rendering Hair
- Managing Dynamic Hair Data
- Efficiently Sending Data to GPU
- Generating Hair on the GPU
- Final Details

Generating data on the GPU



- Sending the large amounts of hair geometry to the GPU can be slow
- 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 have a tessellation engine, which gives us functionality to create large amounts of data directly on the hardware. 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.



This pipeline has three stages, two of them are programmable (the hull and domain shader) and one of the is fixed function (the tessellator). Lets go over what each of these stages does. To help explain the functionality I'm going to use a very simple example – PN triangles. Basically, this tessellation method takes as input a triangle with normals, it fits a bezier patch to this triangle and then triangulates the bezier patch into a specified number of sub triangles.

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 up to 32 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. In our PN Triangles example the Hull shader will compute the tessellation factors, which it sends to the tessellator, it will calculate the control points for the bezier patch that it fits to the input triangle, and any other per patch attributes.

The next stage is the Tessellator, which is fixed function, and this stage produces the actual data expansion on the GPU. The tessellator only takes as input the tessellation factors 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 calculate any vertex attribute data – this data has to be calculated by the programmer in the Domain Shader.In our PN Triangles examples the Tessellator just creates a set of vertices with connectivity and these vertices have uvw coordinates that map to a triangle.

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 or per patch data passed from the Hull shader. Using these inputs we can calculate the attributes of the final vertex. In our PN triangles example a Domain Shader will be invoked for each final vertex created by the tessellator. For each vertex we will evaluate its attributes using the uvw coordinates passed by the tessellator, and the control points passed by the hull shader.

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 or evaluate 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. Also, using the Tessellation engine we can have very fine grained and continuous control over the level of detail.

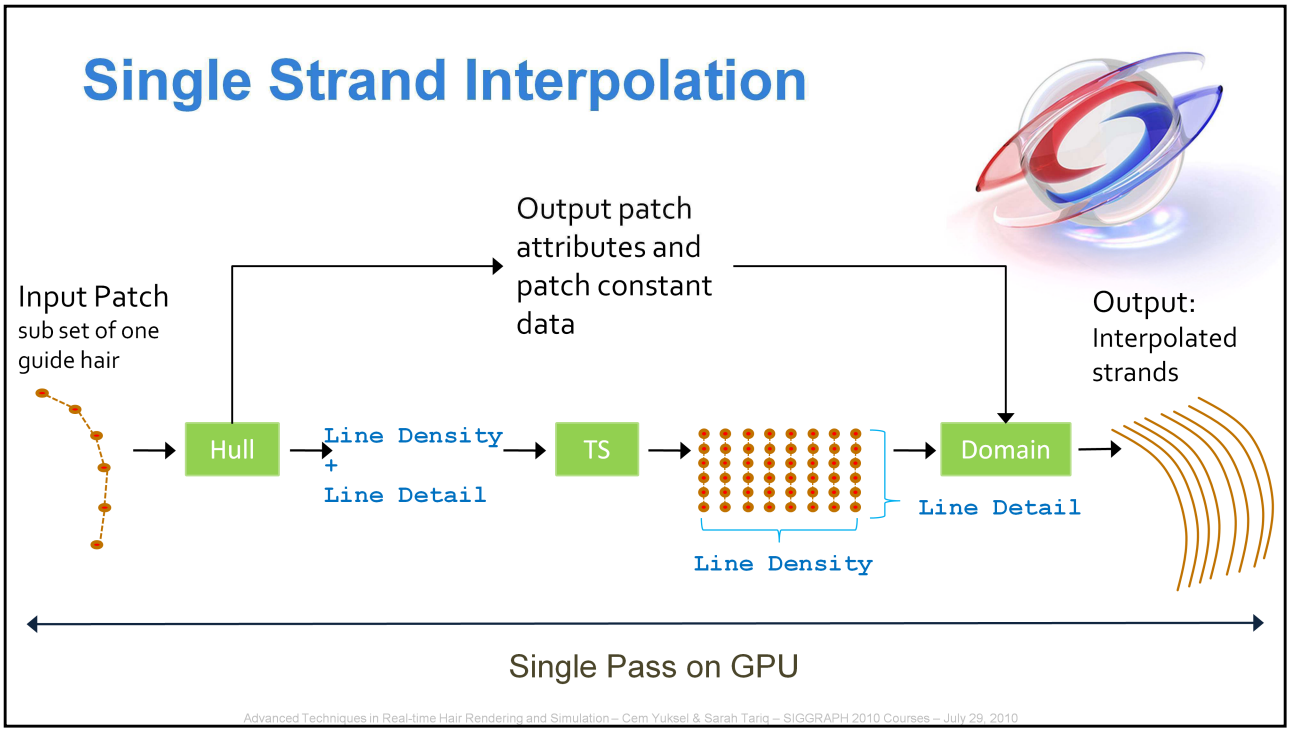
ISO Line Domain

- 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
- Input an arbitrary patch
  - This patch will be different depending on what our method of creating new hair strands is
- 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 specified amount of iso lines. Each isoline is just a set of connected line segments.

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 single strand interpolation method.

Our input is a patch, which represents a section of a tessellated 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, like scalp texture coordinates, which might be needed by the Domain Shader.

Then the Tessellator generates the topology requested by the Hull shader, which is a number of iso lines.

Finally the Domain Shader is invoked once per final tessellated vertex, and is passed the parametric uv coordinates for the vertex, and all the data output from the Hull Shader. The domain shader evaluates the position and other attributes of each vertex by loading the appropriate guide strand data from textures or buffers.

Data Management and Rendering Pipeline

- Rendering Hair
- Managing Dynamic Hair Data
- Efficiently Sending Data to GPU
- Generating Hair on the GPU
- Final Details
- Hair Meshes

Avoiding interpolated hair collisions

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

Interpolated hair intersect collision volumes

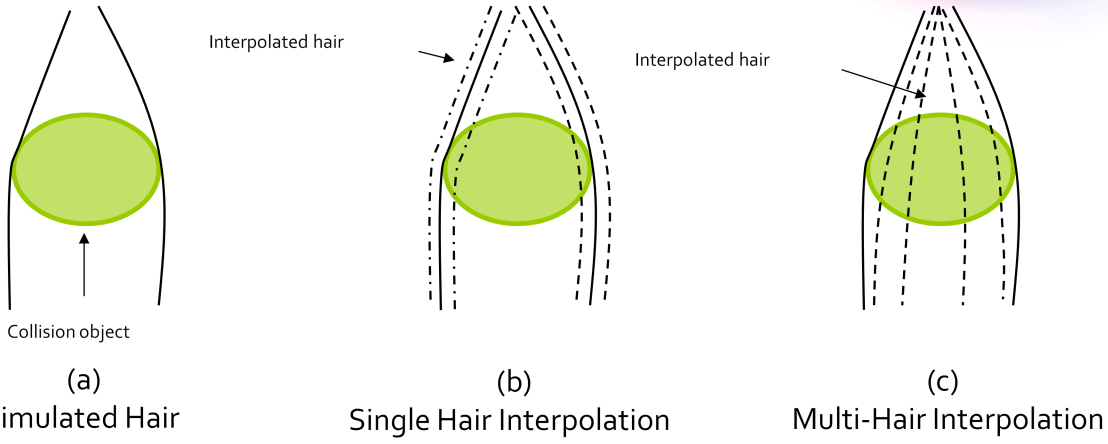
Fixing collisions without doing extra simulation

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.

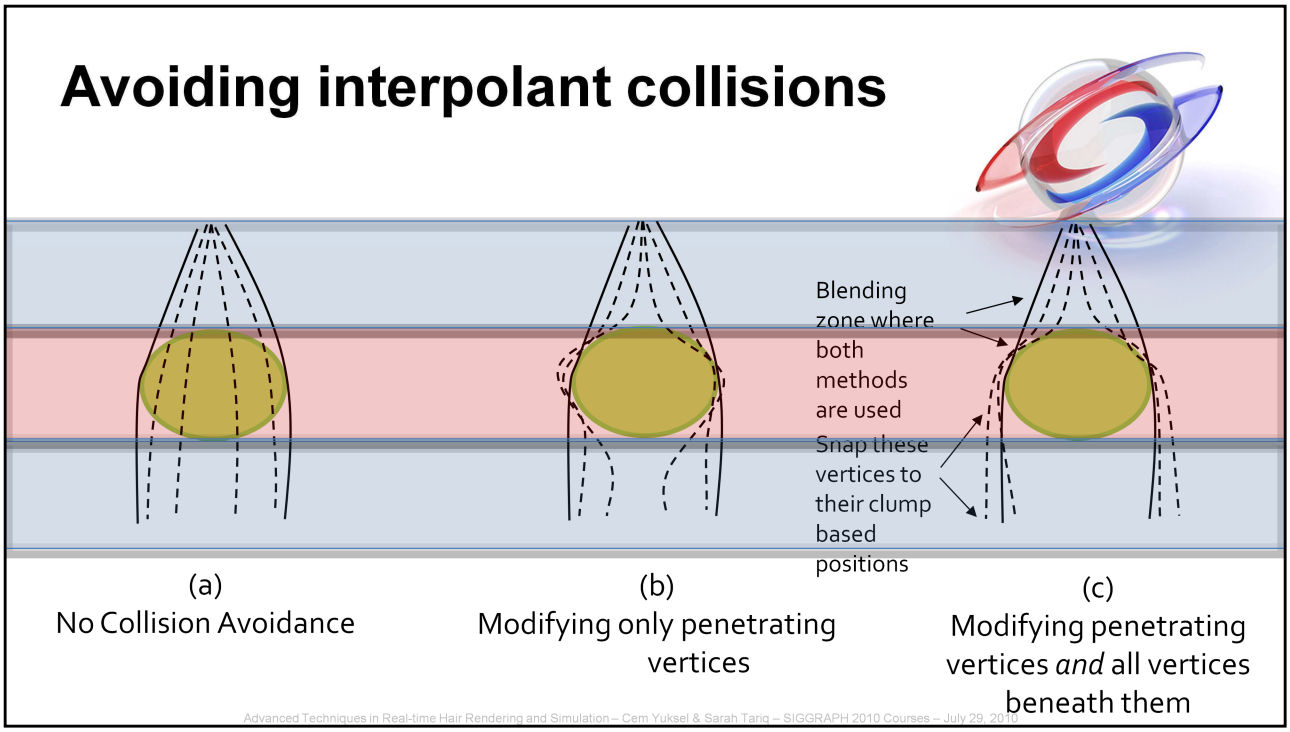
Avoiding interpolated hair collisions



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.

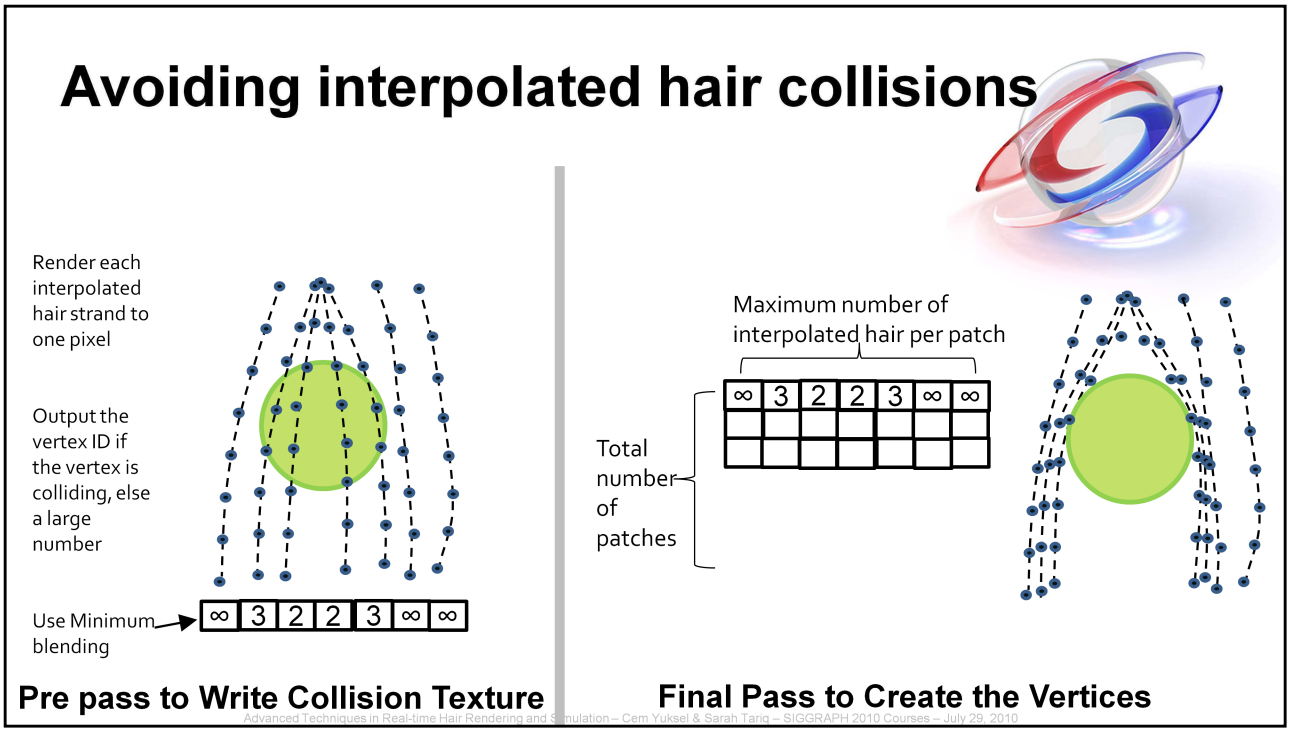
In Figure(c) however we see much worse penetration of interpolated hair even though the guide hairs are not penetrating the object. This is because the interpolated hairs are created by averaging the positions of many guide hair and this process gives us no guarantees that the interpolated positions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

Level of Detail

- Dynamically controlling LOD is important for real time applications
- Reducing amount of hair geometry and making each hair thicker makes rendering faster
  - In the limit this approaches the few large texture mapped polygon look of current games
- Can use the size of a projected segment or distance to the head to decide on the LOD

Full Level of Detail

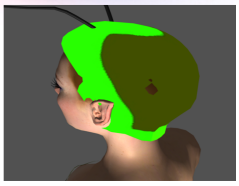
0.5 LOD, more than 2x faster (full frame). Image on Left is the actual image. Image on the right shows the actual quality of hair that we are rendering at 0.5 LOD

Being able to dynamically reduce the complexity of the rendered hair to increase the performance is very important for real time applications. The level of detail for hair can be based on the distance of the 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 Level of Detail on a per patch level by changing the number iso-lines or the number of segments per line.

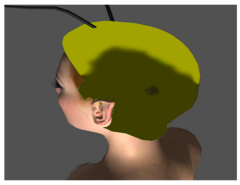
In the images on the right we are showing the same hair style under two levels of detail. At the top we have the hair rendered at full Level of Detail, and at the bottom we have scaled down the rendering and increased performance by 2x. In this case we are creating and rendering less hair strands, although 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.

Level Of Detail

- 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 part of the hairstyle.
  - These LODs can then be transcribed into textures
  - The Hull Shader can be used to blend between appropriate LOD textures to decide on the line density and line thickness



LOD map for Near

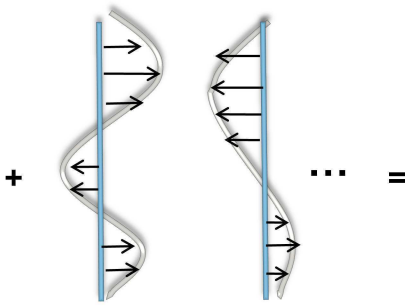


LOD map for Far

There are different ways we can map the Level of Detail to the actual number and thickness of hair strands. We can for example define a simple linear mapping between the desired Level of Detail and the number of hair strands to create.

Or, we can allow an artist to define the actual look of the hair at different discrete levels of detail by creating density and thickness maps for each of them. At run time we can blend between these maps to determine the amount of hair and its thickness for any given LOD. This way we can use the limited number of hair that we can create in the region where the artists feel they will have the highest impact.

Curly Hair



Set of curl deviations, created either procedurally or from curls modeled by artists

Simulating curly hair is quite difficult, but faking the look of curly hair is actually quite easy. In order to render curly hair we add pre-computed curl offsets to the interpolated strand vertex positions along the vertex coordinate frames. The curl offsets can be computed procedurally, or they can be computed by having artists create a set of desired curls, and deriving from those curls the offsets.

Random Variations

Original Hair

With Variations

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, as in the bottom). Since our model for rendering hair is based on interpolating many children hair from a small set of guide hair, we introduce this randomness at the interpolated hair level. We pre-compute a small set of smooth random deviations and apply these to the interpolation coordinates of the interpolated hair.

Data Management and Rendering Pipeline

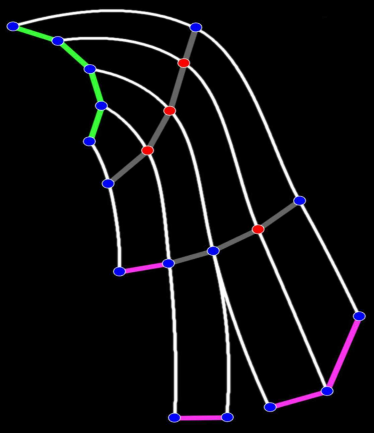
- Rendering Hair
- Managing Dynamic Hair Data
- Efficiently Sending Data to GPU
- Generating Hair on the GPU
- Final Details
- Hair Meshes

Hair Meshes

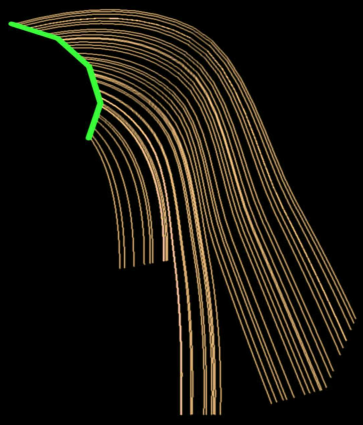
Hair Meshes

- Volumetric Structure
  - Topological connections
  - Can uniquely trace a path from root to tip
- Editable as a surface
  - Polygonal modeling concepts
  - Obey topological constraints

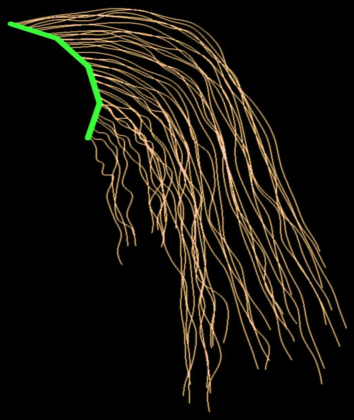
Hair Modeling with Hair Meshes



Hair Mesh Modeling

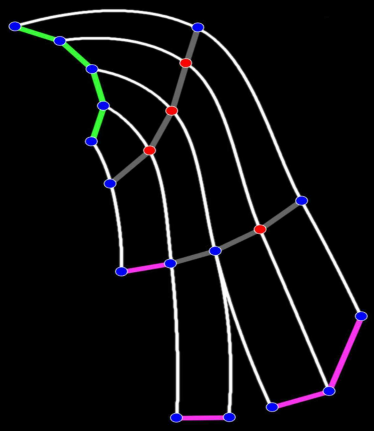


Hair Generation

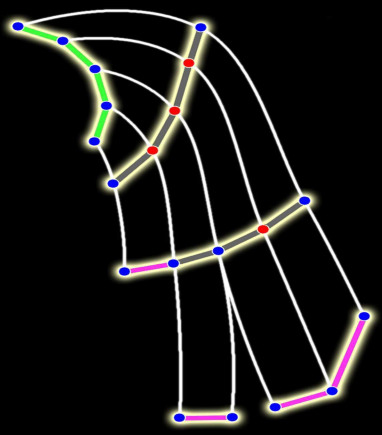


Hair Styling

Hair Meshes

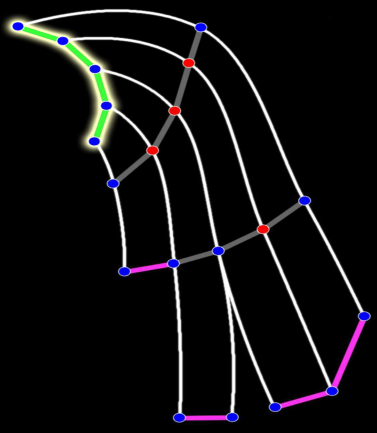


Hair Meshes



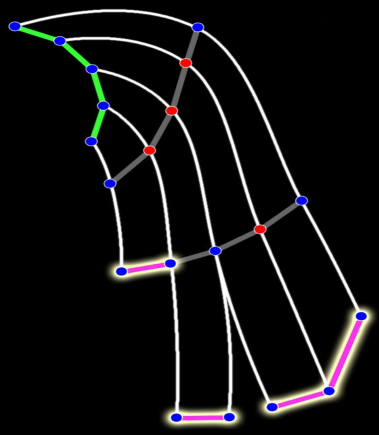
- Layers

Hair Meshes



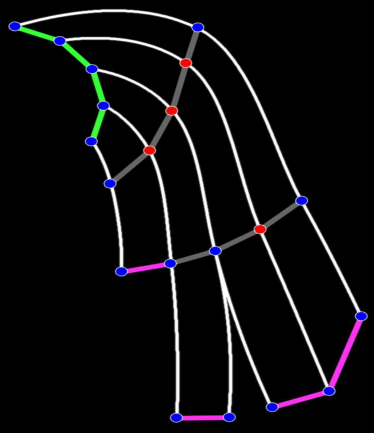
- Layers
  - **Root Layer**

Hair Meshes



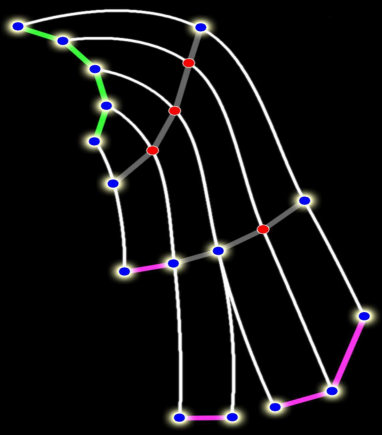
- Layers
  - Root Layer
  - **Tip Layer**

Hair Meshes



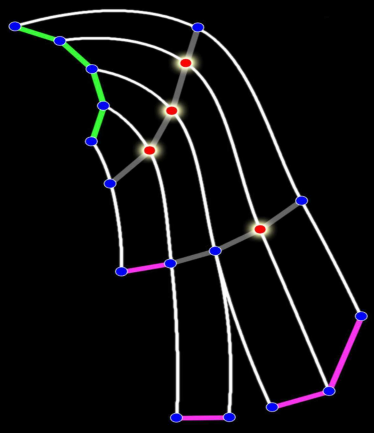
- Layers
  - Root Layer
  - Tip Layer
- Vertices

Hair Meshes



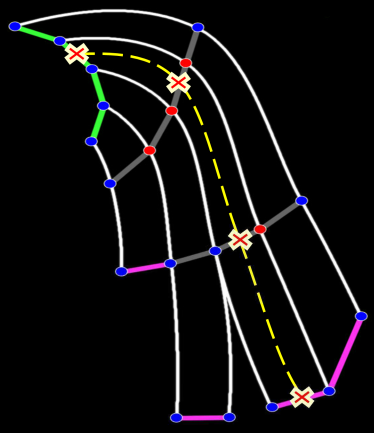
- Layers
  - Root Layer
  - Tip Layer
- Vertices
  - **External Vertices**

Hair Meshes



- Layers
  - Root Layer
  - Tip Layer
- Vertices
  - External Vertices
  - **Internal Vertices**

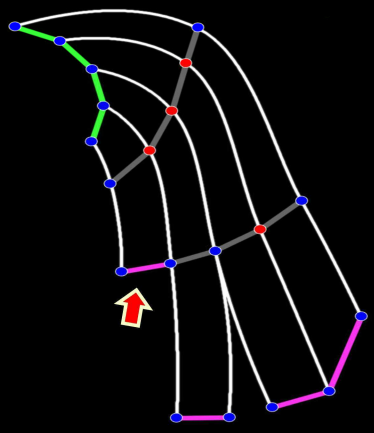
Hair Meshes



Hair Generation

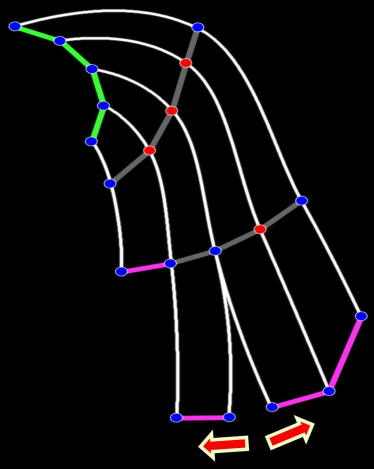
- Pick a point at root layer
- Find corresponding points on all layers
- Connect them with a curve (ex. Catmull-Rom splines)

Hair Meshes



- Tip layer can be different for each face

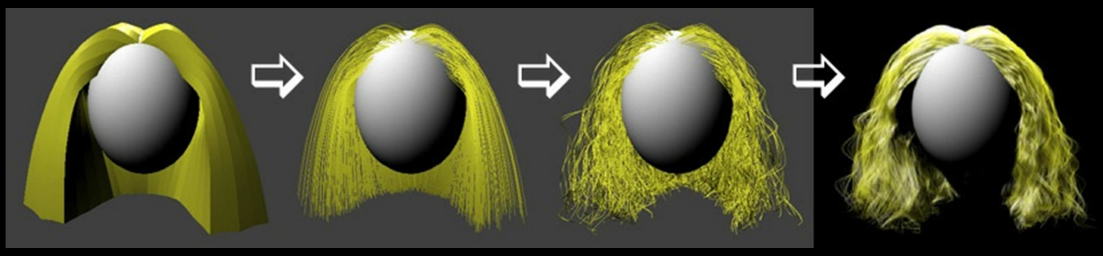
Hair Meshes



- Tip layer can be different for each face
- Topology can change between layers

Hair Styling with Hair Meshes

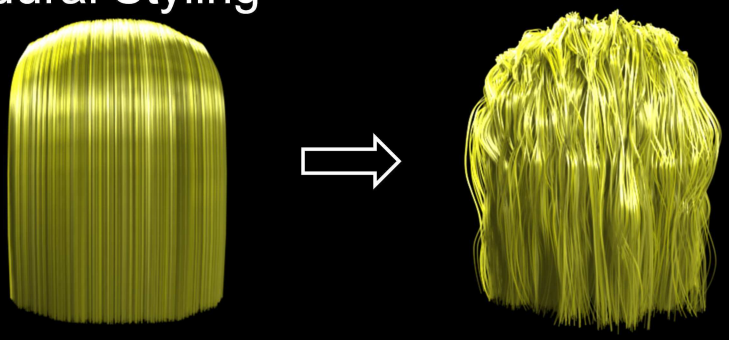
- All operations on hair strands are *styling* operations



Hair Mesh Modeling Hair Generation *Styling* Rendering

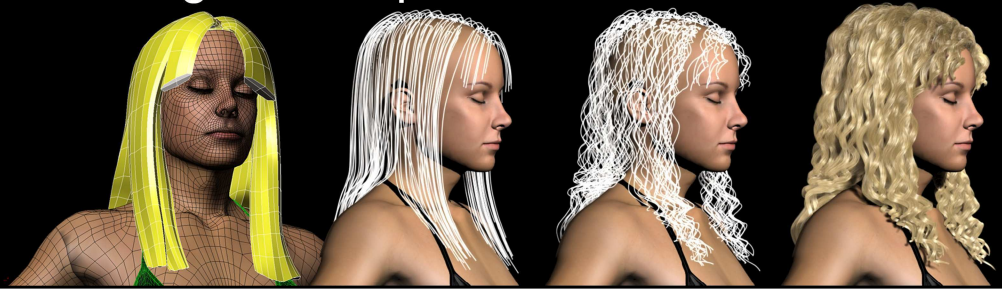
Hair Styling with Hair Meshes

- All operations on hair strands are *styling* operations
- Procedural Styling



Hair Styling with Hair Meshes

- All operations on hair strands are *styling* operations
- Procedural Styling
- Combining with wisp-based methods




HAIRFARM™
THE ULTIMATE HAIR PLUG-IN FOR 3DS MAX
www.hair-farm.com

Hair Meshes for Real-time Rendering

- Send hair mesh to GPU
- Generate hair strands on the GPU
- Styling variations
  - Procedural functions
  - Explicitly prepared texture maps

Transparency and Antialiasing

Advanced Techniques in Real-time Hair Rendering and Simulation
SIGGRAPH 2010 Course

Sarah Tariq
NVIDIA

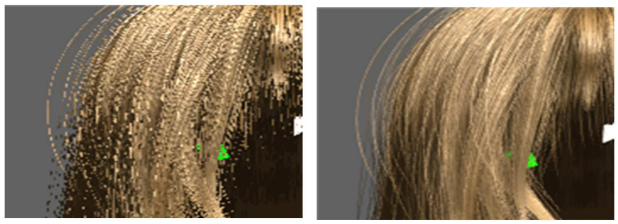
Cem Yuksel
Cyber Radiance and Texas A&M University

Transparency and Antialiasing

- Overview
- Antialiasing
- Transparency and Depth sorting
- Avoiding Sorting for Transparency

Overview

Hair Strands are **Thin**



No Antialiasing | With Antialiasing

Hair Strands are **Semi-Transparent**

No Transparency | With Transparency

There are two main issues that we are going to deal with in this section.

The first is that 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. We can either use antialiasing techniques to combat this, or use thicker lines with transparency.

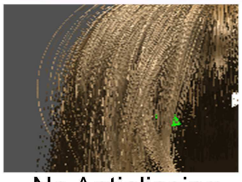
The second is that hair strands, especially light colored hair, are slightly **transparent.** So we need a way to render transparent geometry. Transparency also helps convey the look of thinner hair – useful for subpixel sized hair.

Transparency and Antialiasing

- Overview
- Antialiasing
- Transparency and Depth sorting
- Avoiding Sorting for Transparency

Antialiasing

- Antialiasing options:
  - Pre-filter
    - works well for points and lines, not for triangles
    - Requires alpha blending which is costly
  - Over-sample:
    - Super Sampled AA (SSAA)
    - Multi Sampled AA (MSAA)

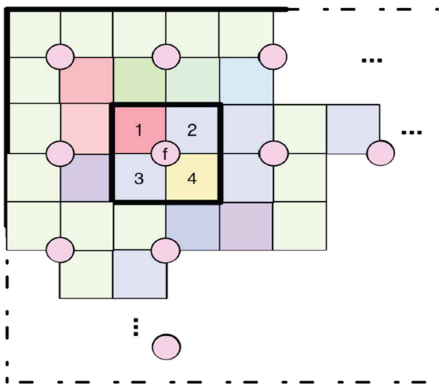


No Antialiasing

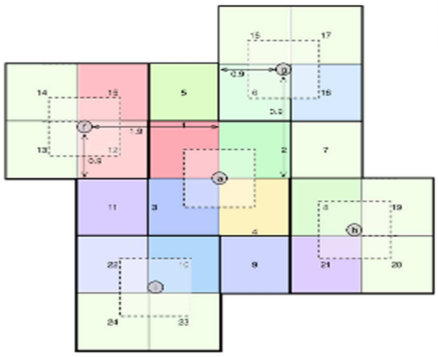
With Antialiasing

There are a number of ways that we can deal with geometric aliasing of thin hair strands; for example we can use pre-filtering or oversampling. In the remainder of this section we are going to be talking about two over sampling techniques, SSAA and MSAA.

Super Sampled AA



hardware bilinear filter based down sample



Down sampling with 5 bilinear tap rotated filter

Images from Lorach07

We start with super sampling ,which simply means render the scene at a higher resolution and then down sample to a lower resolution output.

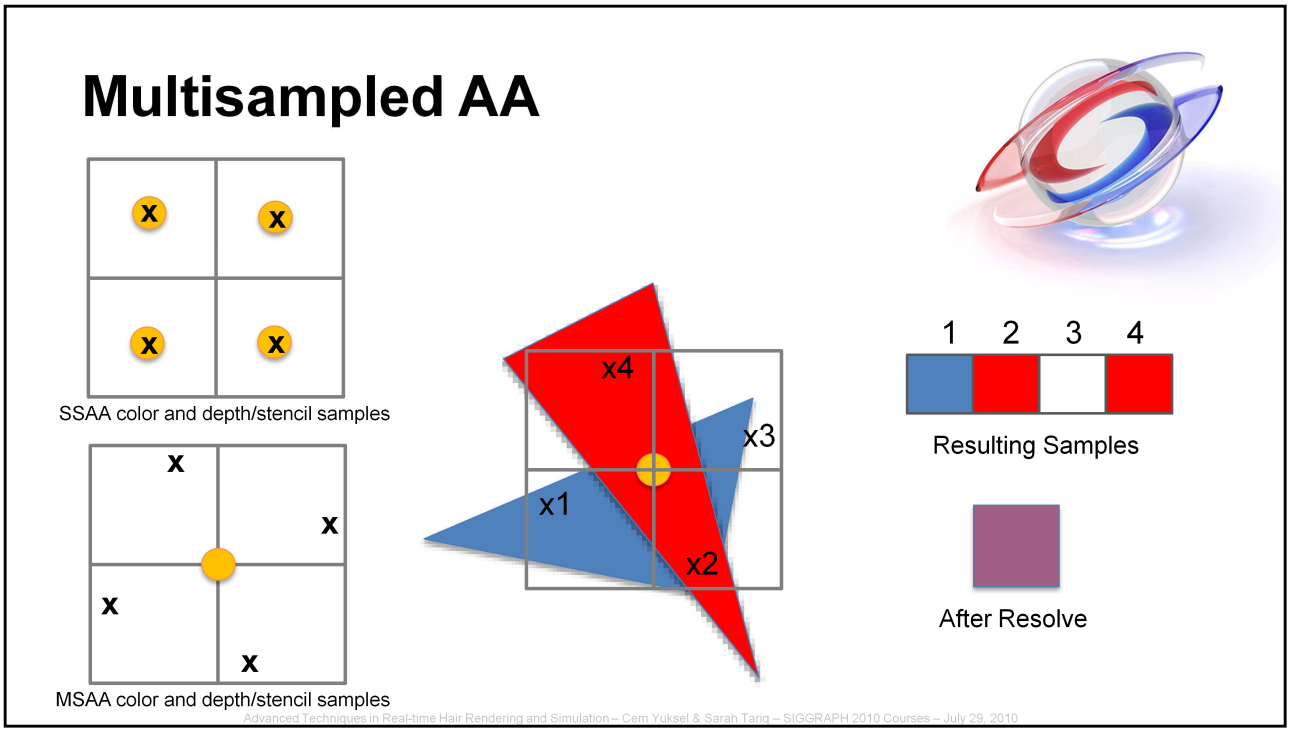
SSAA incurs a large bandwidth and shading cost, increasing amount of memory resident on the GPU which can lead to more paging.

When downsampling we can use different reconstruction filters, for example instead of a simple average of four samples we can use a larger filter like the one on the right.

This 5 tap takes a weighted average of 20 samples and gives a softer look – since it is more blurry it is something that some people might prefer and others not.

This filter also fakes a more transparent look because we are sampling from neighbors and are potentially pulling more color from the background.

The cost is not that much more than the one tap, since you are using the same color buffer, just taking more samples.



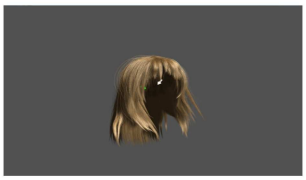
The next antialiasing option that we have is Multisampled Antialiasing, or MSAA. MSAA is a variation of SSAA, which is implemented in hardware. MSAA decouples the rate at which we evaluate the pixel shader and the depth/stencil tests.

So, for example in the case of 4xSSAA we would have evaluated both the pixel shader and the depth/stencil tests 4 times for each pixel. In MSAA we will evaluate the pixel shader only once, for example at the center. We will still perform 4 depth stencil tests, although their locations will be a bit offset.

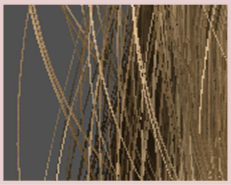
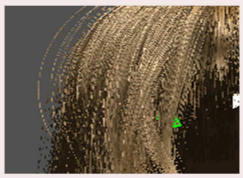
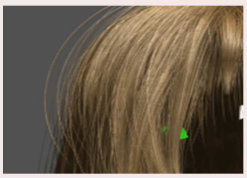
Here is an example of how MSAA would work. When we render this blue triangle we evaluate the pixel shader once at the center of the pixel, and we perform the depth stencil test at all the covered samples. For each sample that passes the depth/stencil test we write the evaluated color at that sample location. We do the same thing for the other triangles, and at the end we have the color of all 4 samples, and we can down sample by a simple averaging to get the final pixel color.

Comparison

Original image
1024x768



Detailed views

No AA	8x MSAA	4x SSAA
51 fps	48fps	25 fps
		
74 fps	71 fps	63fps
		

Transparency and Antialiasing

- Overview
- Antialiasing
- Transparency and Depth sorting
- Avoiding Sorting for Transparency

Alpha blending

- Commonly used to render transparent objects
- Fragments should be sorted (from the camera's perspective), either back to front or front to back
- Back to Front

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

- Front to back

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

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

Blending with Correct Sorting

Blending with Incorrect Sorting

Images from Kim03

Sorting

- Can sort on the CPU, but this will limit the amount of geometry that can be rendered in real time
  - Want to render many strands
  - Will be generating strands on GPU, so minimize transfer

Depth Peeling

Opaque dragon

Transparent dragon rendered with 4 depth peeled layers

Depth peeling [Everitt01] can be used to render correctly sorted fragments with transparency on the GPU.

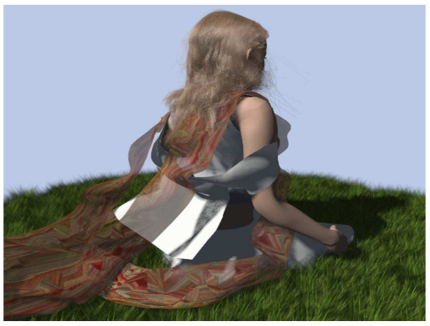
In depth peeling the scene is rendered many times, each time the depth buffer from the last pass is read in the pixel shader to reject any fragments with depth less than or equal to the value in depth buffer. This way we can get fragments at successive depth layers. With traditional depth peeling we have to render the scene once for each layer that we peel, but improvements of this method, for example stencil routed k buffer can peel up to 8 layers at a time, which is significantly faster.

Depth peeling is a very efficient way of handling transparency for relatively few depth layers - here we have an image rendered with just 4 depth layers.

However, depth peeling is not practical for the large depth complexity typical of hair

Depth Peeling

Depth peeling with 5 layers

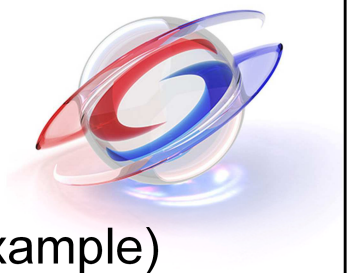


Reference

Images from Enderton10

As we can see in the image on the left depth peeling using only 5 layers is hopelessly inadequate for rendering transparent hair, due to its large depth complexity.

Sorting on the GPU



- Sort fragments (using directx11 for example)
  - Either per-pixel linked lists or one large sorted list of fragments
- Sort line segments

Instead, a better approach is to try to sort either the fragments or the line segments of the hair on the GPU.

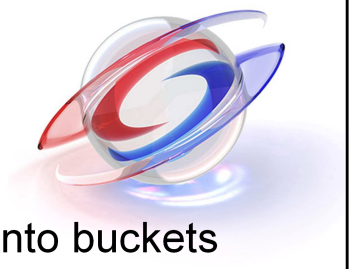
Sorting on the GPU

Image from Sintorn

- ~0.5 million line segments
  approximate alpha sorting
  256 Opacity Map slices
- 2008: 7.5fps (8800GTX, quick sort) [Sintorn08]
- 2009: 37 fps (GTX280, radix sort) [Sintorn09]

Here is an example of the sort of performance you can get if you render hair with almost half a million line segments, with sorting blending and shadows.

Sorting on the GPU



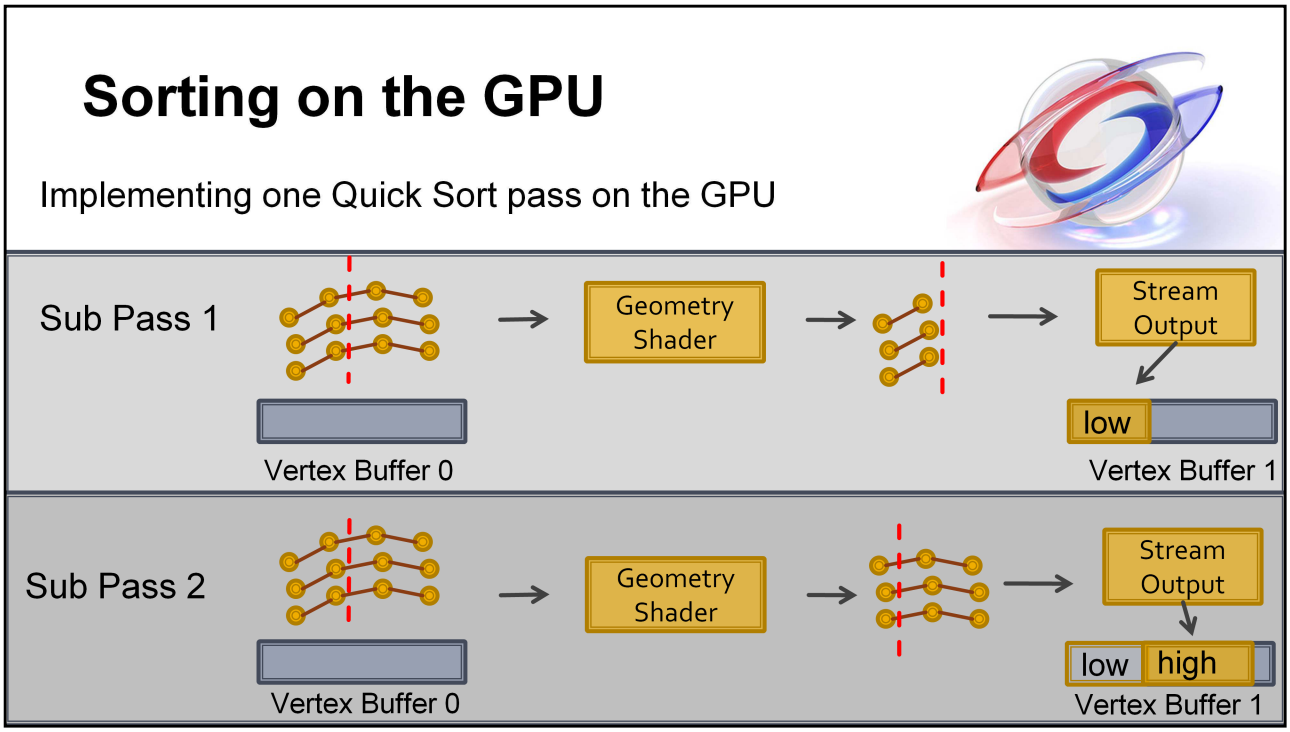
- Can approximately sort hair line segments into buckets along depth using partial sorting
  - Quick Sort [Sintorn08]
  - Radix Sort [Sintorn09], [Satish09]

- Quick Sort:
  - Choose pivot element in list. Re-order list to place all elements smaller than pivot before it, and larger than pivot after it. Recursively apply to two sublists
  - Complexity of O(nlog(s))

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)



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.

In order to do quick sort we need to know the position of the pivot – so that we can recursively sort the lower and higher sections. We can do this by using a query (below) which will give the amount of primitives written. This query is asynchronous, so we can issue it around the first pass, and read the results after pass2, before the next iteration. Note that this might still cause a stall.

typedef struct D3D10_QUERY_DATA_SO_STATISTICS { UINT64 NumPrimitivesWritten; UINT64 PrimitivesStorageNeeded; } D3D10_QUERY_DATA_SO_STATISTICS;

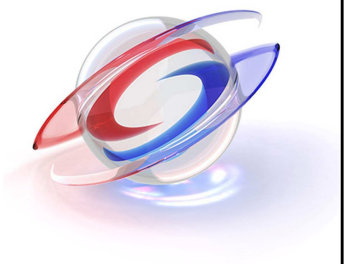
Or, we can just do the test in the GS to figure out which previous bucket you are in based on a LUT.

Pass2 starts appending data at the place where pass 1 ended writing, and this is done by passing an offset of -1 to StreamOut: An offset of -1 will cause the stream output buffer to be appended, continuing after the last location written to the buffer in a previous stream output pass.

Transparency and Antialiasing

- Overview
- Antialiasing
- Transparency and Depth sorting
- Avoiding Sorting for Transparency

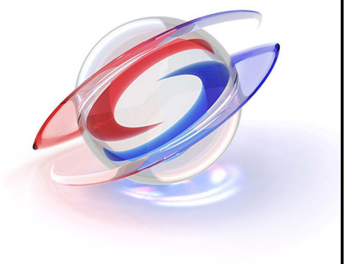
Alpha Test

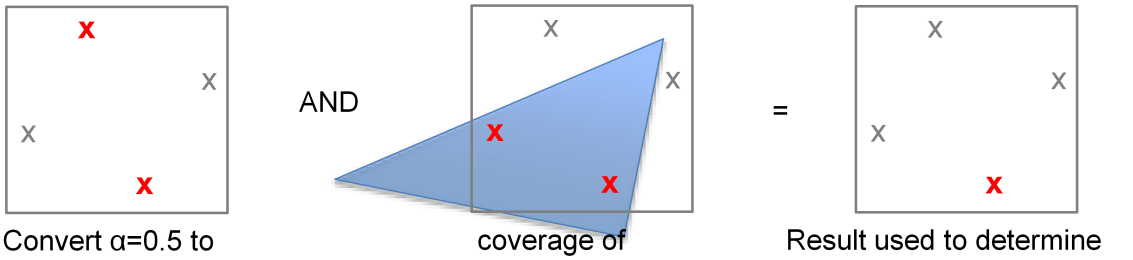


- Simplest way to deal with Alpha
- Use computed alpha to decide whether or not to kill the fragment in the pixel shader
- Only a binary test, not very useful for transparency, better suited for “cut out” transparency

The simplest way to deal with the alpha of a fragment without any sorting is to use the Alpha test, but since it is only a binary test it is more useful for “cut out” transparency.

Alpha to Coverage





The next option is a hardware implemented feature called Alpha to Coverage, which results in something similar to screen door transparency.

Alpha-to-coverage works by taking the alpha value and converting it into an n-step coverage mask, where n is the sample count. This n-step coverage mask is then AND'ed with the multisample coverage mask and the result is used to determine which samples should get updated for all of the render targets currently bound.

1. computed alpha -> n-step coverage mask
   - n is the number of samples
2. AND with multisample coverage mask
3. Use result to determine which samples should get updated

There is no precise specification of exactly how alpha gets converted to a coverage mask by the hardware, except that alpha of 0 (or less) must map to no coverage and alpha of 1 (or greater) must map to full coverage (before ANDing with actual primitive coverage). As alpha goes from 0 to 1, the resulting coverage should generally increase monotonically, however hardware may or may not perform area dithering to provide some better quantization of alpha values at the cost of spatial resolution and noise.

Alpha To Coverage

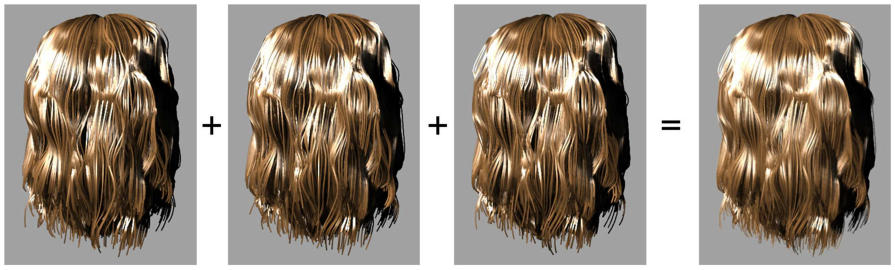
- Benefits:
  - requires no sorting, trivial to use in GPUs
- Drawbacks:
  - usually disables earlyZ in hardware
  - Do not get very smooth images
  - Quality depends on number of MSAA samples

The original alpha value in the output of the pixel shader is not changed when that alpha value is used to create the n-step coverage mask (alpha blending will still occur on a per-sample basis).

Alpha-to-coverage will work regardless of whether or not multisampling is also turned on in the rasterizer state. Alpha-to-coverage multisampling is essentially the same as regular multisampling except that the n-step coverage mask is generated and ANDed with the multisample coverage mask.

Fake Transparency

- Fake the look of transparency by dividing the hair into 3 subgroups and blending the images together



Finally, yet another way to add a bit of transparency to hair without incurring the overhead of sorting is to render hair as separate disjoint sets to different render targets and then blend the results together. This approach does not incur a lot of performance overhead, although it does necessitate more storage.

Summary of Transparency and Antialiasing

- Recommended approach
  - For fastest implementation use multi layer fake transparency with MSAA
  - For a better looking approach use MSAA in combination with alpha blending and GPU sorted line fragments

References

- Enderton10. *Stochastic Transparency.* Eric Enderton, Erik Sintorn, Peter Shirley, David Luebke. Symposium on Interactive 3D Graphics and Games (I3D), 2010
- Everitt 02. *Interactive Order-Independent Transparency.* Cass Everitt. Technical report, NVIDIA Corporation, 2002.
- Ikits 04. *Volume Rendering Techniques.* Milan Ikits, Joe Kniss, Aaron Lefohn, Charles Hansen. Chapter 39, GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, 2004.
- Kim03. *Algorithms for Hardware Accelerated Hair Rendering.* Tae-Yong Kim. Game Tech 2003
- Lorach07. *High Quality Antialiasing.* Tristan Lorach, 2007.
http://developer.download.nvidia.com/SDK/10.5/opengl/src/FroggyAA/doc/FroggyAA.pdf
- Satish09. *Designing Efficient Sorting Algorithms for Manycore GPUs.* Satish, Harris and Garland. Proc. 23rd IEEE Int'l Parallel & Distributed Processing Symposium, May 2009
- Sintorn09. *Radix sort of line primitives in CUDA for real-time Self-Shadowing and Transparency of Hair.* Sintorn, Assarson, Olsson and Billeter. Research Poster GPU Technology Conference, 2009.
- Sintorn08. *Real-Time Approximate Sorting for Self Shadowing and Transparency in Hair Rendering.* Symposium on Interactive 3D Graphics and Games (I3D), 2008

SIGGRAPH 2010

WONDER!

"***The People Behind The Pixels***"

Community … Clarity … Content

Hair Shading

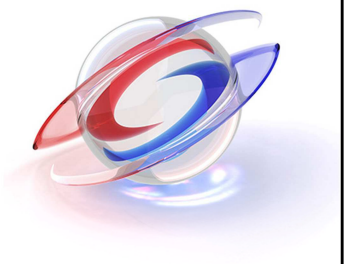
Advanced Techniques in Real-time Hair Rendering and Simulation
SIGGRAPH 2010 Course

Cem Yuksel
Cyber Radiance and Texas A&M University

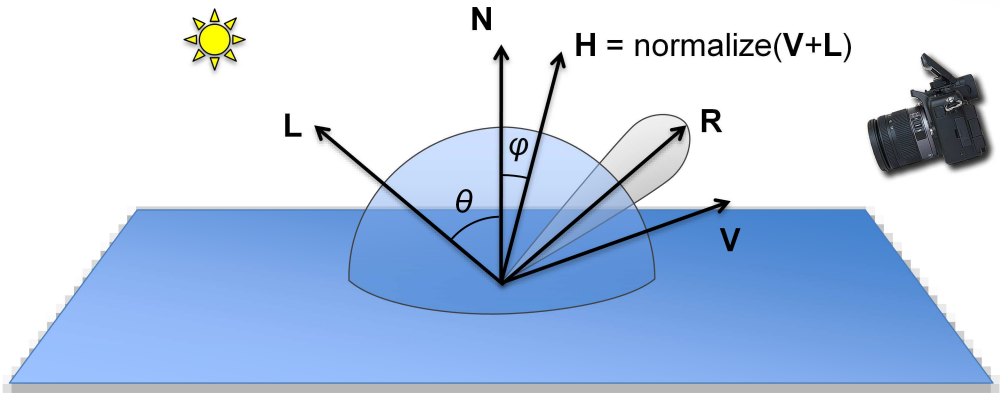
Hair Shading

- Simple Hair Shading
- Physically-based Hair Shading
- Improving Shading Performance

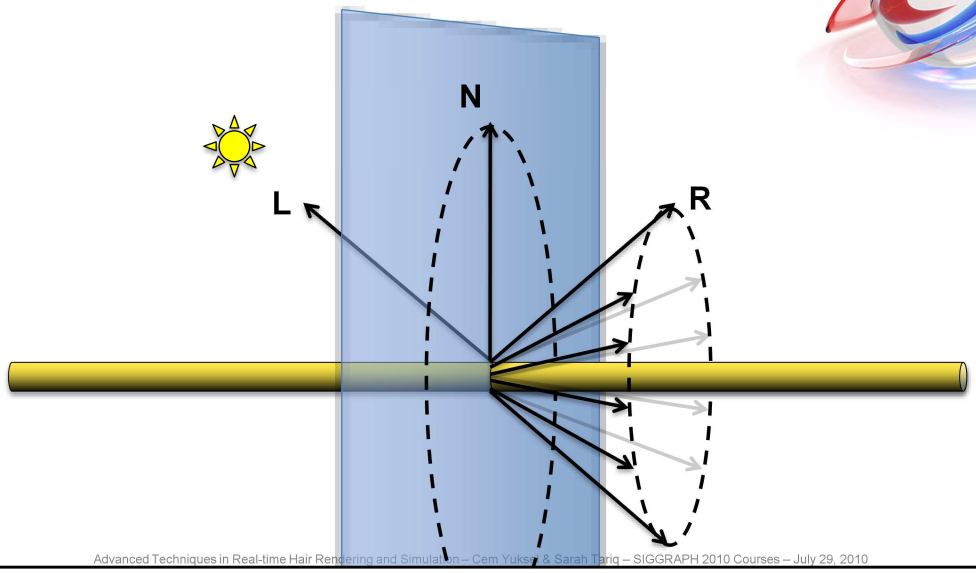
Shading Surfaces



$$L_{out} = L_{in}\left(C_{diffuse}\cos\theta + C_{specular}\cos(\mathbf{N},\mathbf{H})^{\alpha}\right)$$

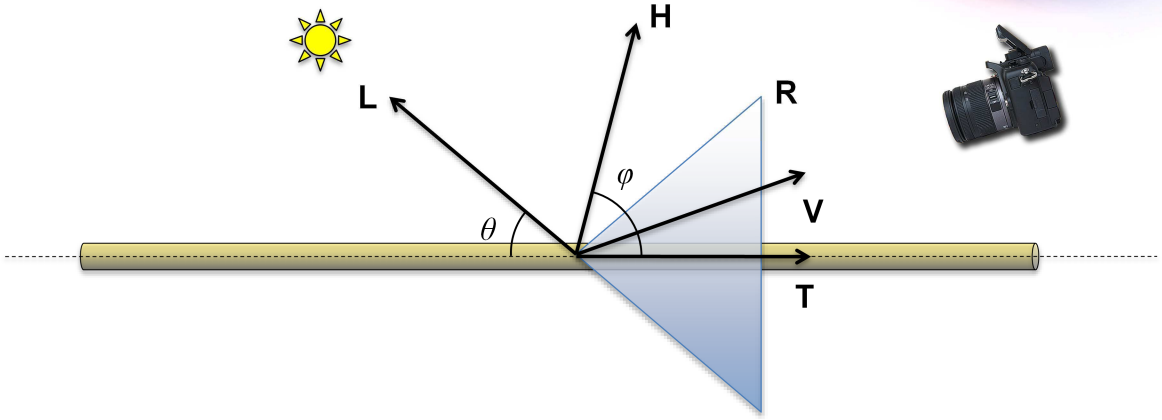


Simple Hair Shading



Kajiya-Kay Hair Shading

$$L_{out} = L_{in}\left(C_{diffuse}\sin\theta + C_{specular}\ \sin(\mathbf{T},\mathbf{H})^{\alpha}\right)$$



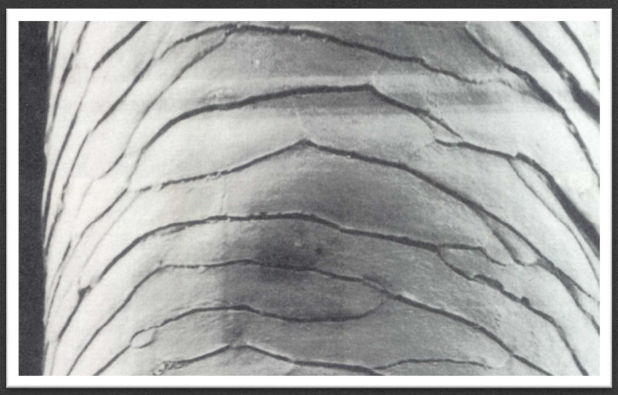
Kajiya-Kay Hair Shading

- Simple
- Ignores hair transparency
- Good for very dark hair
- Does not match measurements/observations for lighter hair colors

Hair Shading

- Simple Hair Shading
- Physically-based Hair Shading
- Improving Shading Performance

The Hair Surface



ROBBINS, C. R. 1994. Chemical and Physical Behavior of Human Hair, third ed. Springer-Verlag, New York.

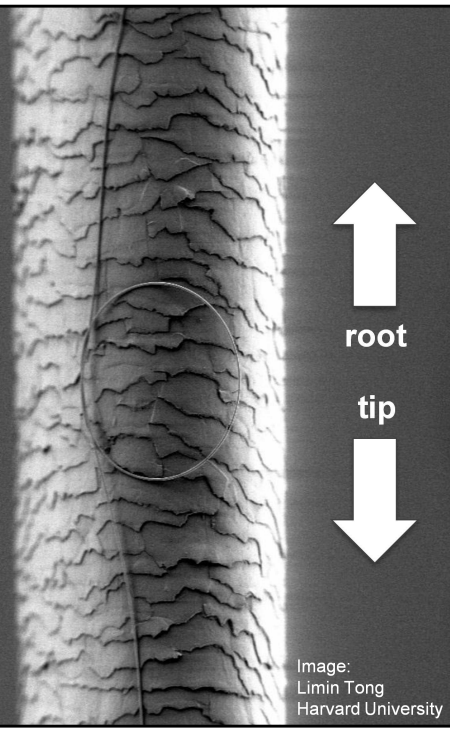
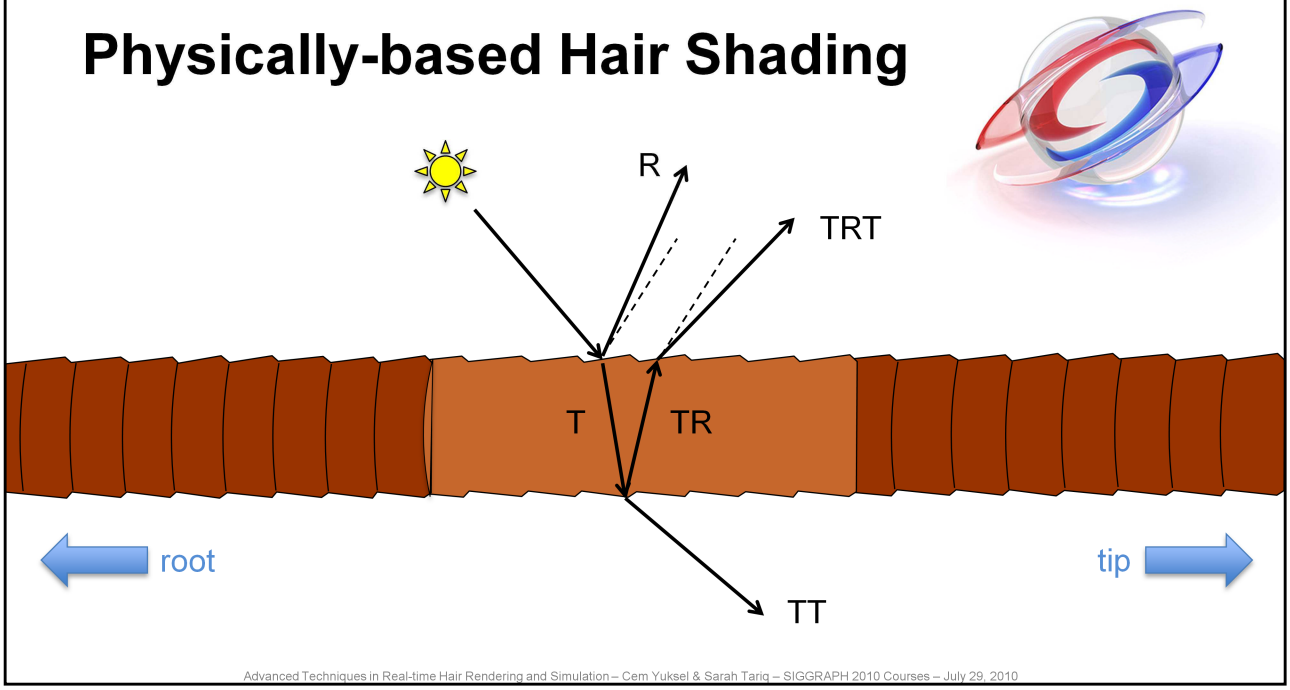
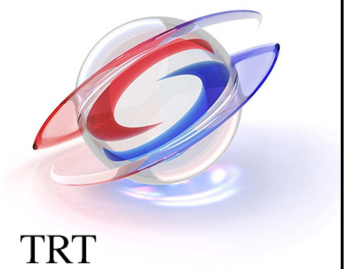


Image:
Limin Tong
Harvard University

Physically-based Hair Shading



Physically-based Hair Shading



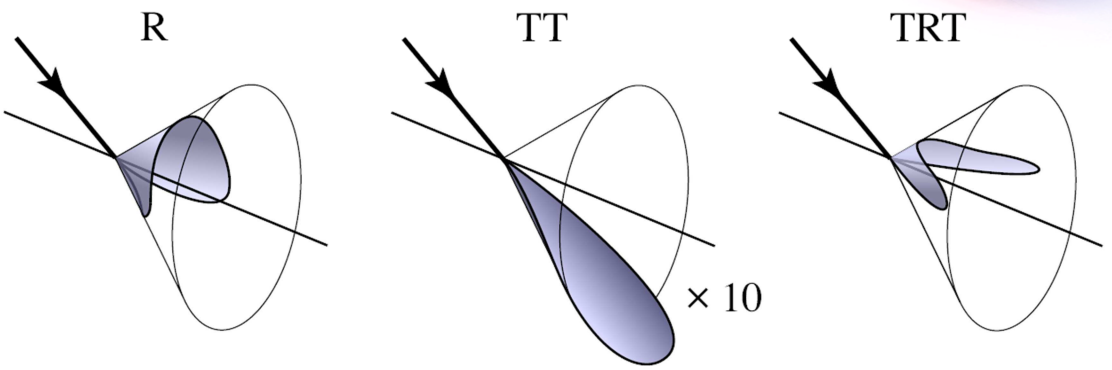
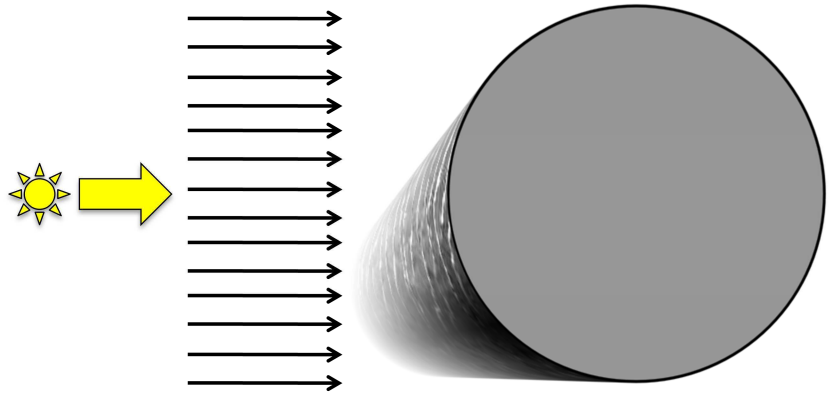
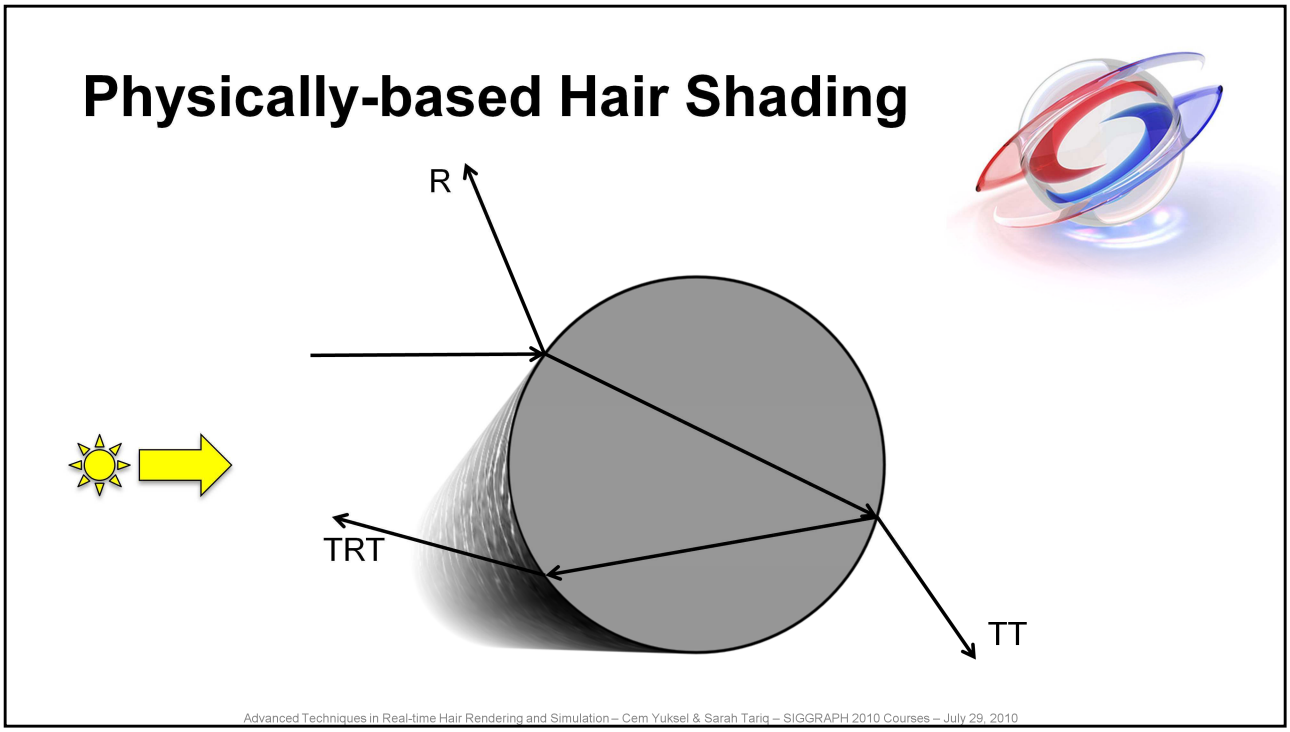


Figure: Moon and Marschner 2006

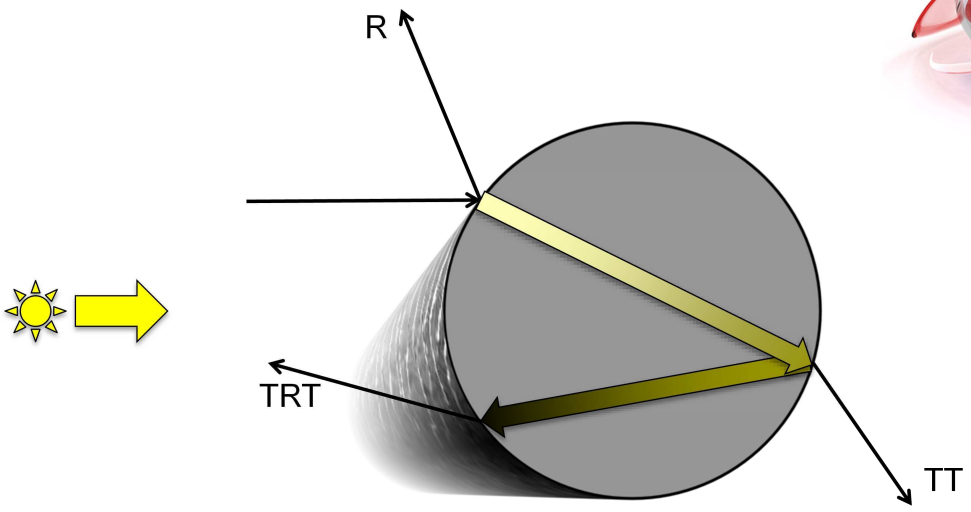
Physically-based Hair Shading

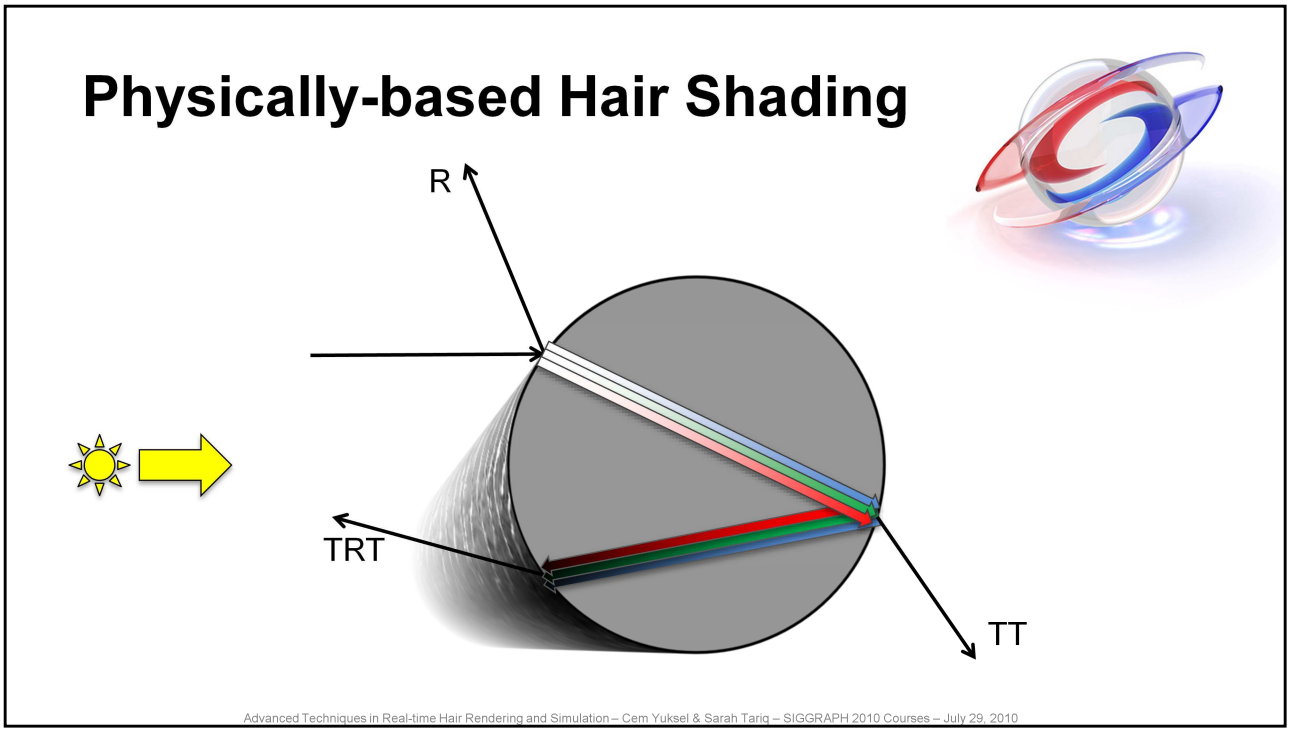




Index of refraction is 1.55

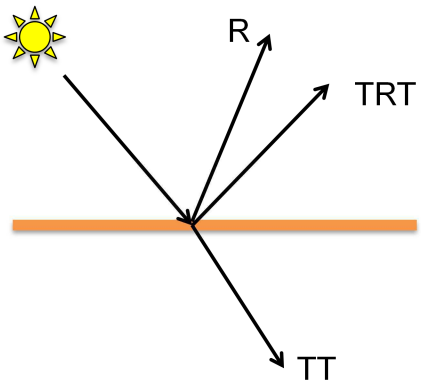
Physically-based Hair Shading



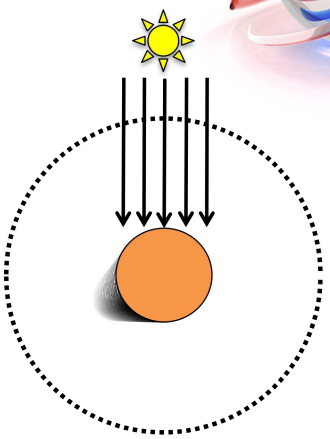


Absorption coefficients σ_a for red, green, and blue are 0.2 to infinity, and their unit.

Physically-based Hair Shading



Longitudinal



Azimuthal

Physically-based Hair Shading

- Shading equation

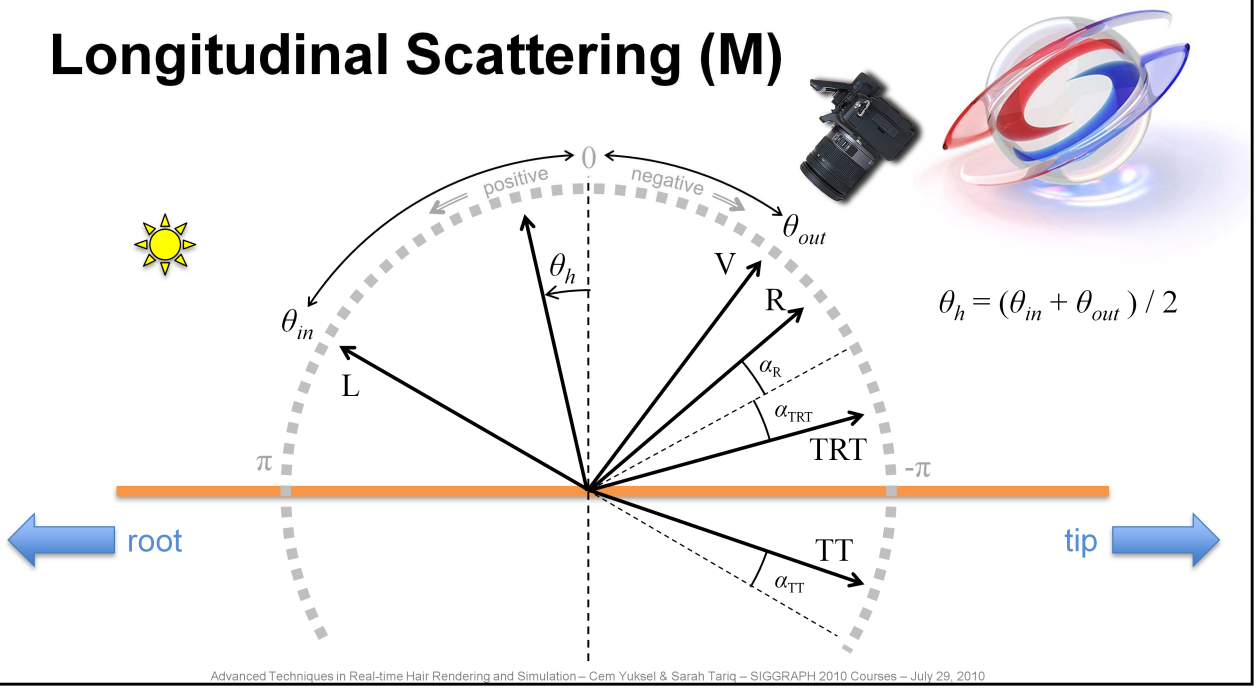
$$L_{out} = D \cos\theta_{in}\ L_{in}\ S(\mathbf{L},\mathbf{V})$$

Thickness factor ← → Scattering function

$$S(\mathbf{L},\mathbf{V}) = \frac{M_R N_R + M_{TT} N_{TT} + M_{TRT} N_{TRT}}{\cos\left(\frac{\theta_{out} - \theta_{in}}{2}\right)}$$

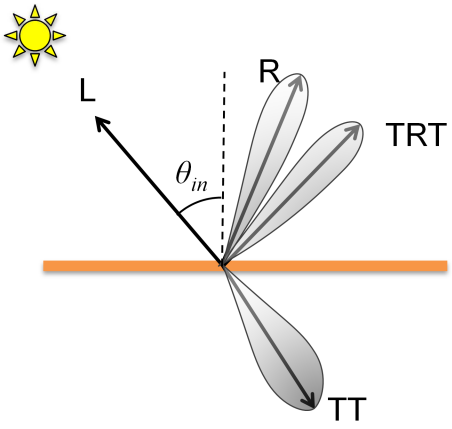
$M_R,\ M_{TT},\ M_{TRT}$ are longitudinal scattering and $N_R,\ N_{TT},\ N_{TRT}$ are azimuthal scattering

Longitudinal Scattering (M)



$$\theta_h = (\theta_{in} + \theta_{out}) / 2$$

Longitudinal Scattering (M)



$$\theta_h = (\theta_{in} + \theta_{out}) / 2$$

$$M_R = \text{Gauss}\,(\beta_R, \theta_h - \alpha_R)$$
$$M_{TT} = \text{Gauss}\,(\beta_{TT}, \theta_h - \alpha_{TT})$$
$$M_{TRT} = \text{Gauss}\,(\beta_{TRT}, \theta_h - \alpha_{TRT})$$

$$\text{Gauss}(\beta, \theta) = \frac{1}{\beta\sqrt{2\pi}} e^{-\theta/\beta}$$

Longitudinal Shifts:

α_R = -10˚ to -5˚

α_{TT} = - α_R / 2

α_{TRT} = - 3α_R / 2

Longitudinal Widths (stdev.):

β_R = 5˚ to 10˚

β_{TT} = β_R / 2

β_{TRT} = 2β_R

Physically-based Hair Shading

- Shading equation

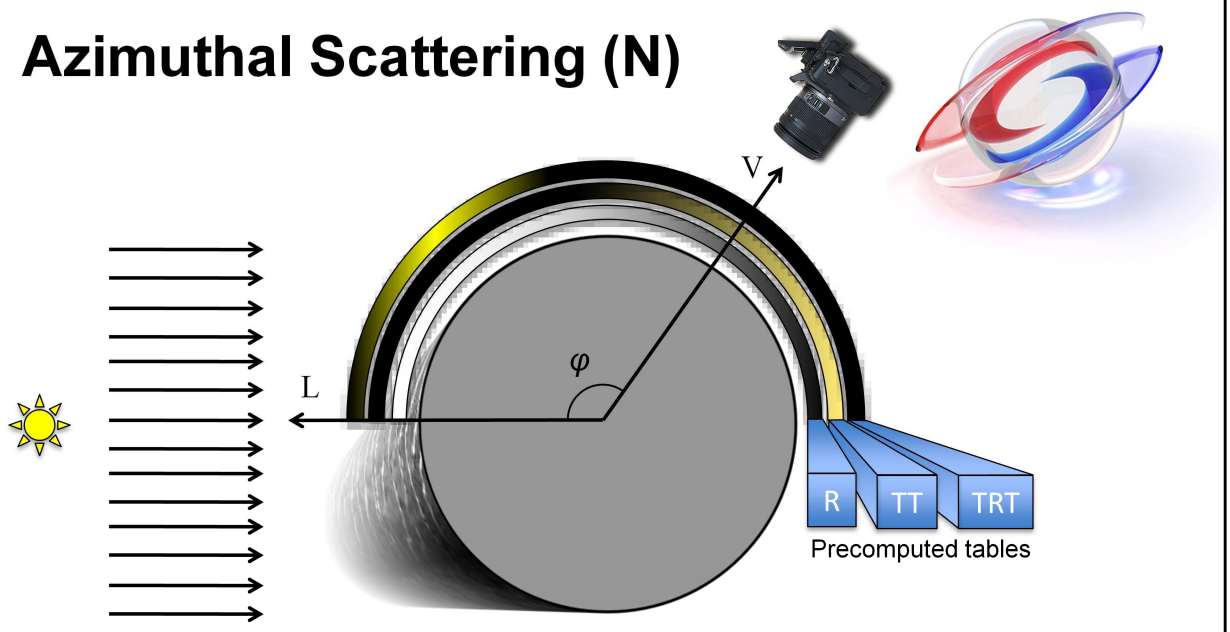
$$L_{out} = D \cos\theta_{in}\ L_{in}\ S(\mathbf{L},\mathbf{V})$$

Thickness factor ← → Scattering function

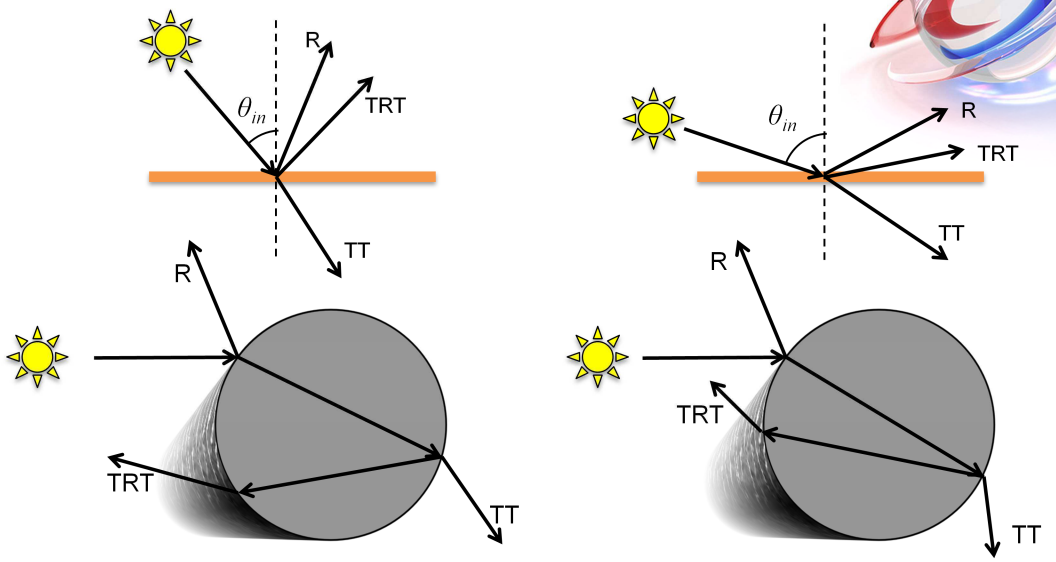
$$S(\mathbf{L},\mathbf{V}) = \frac{M_R N_R + M_{TT} N_{TT} + M_{TRT} N_{TRT}}{\cos\left(\frac{\theta_{out} - \theta_{in}}{2}\right)}$$

$M_R,\ M_{TT},\ M_{TRT}$ are longitudinal scattering and $N_R,\ N_{TT},\ N_{TRT}$ are azimuthal scattering

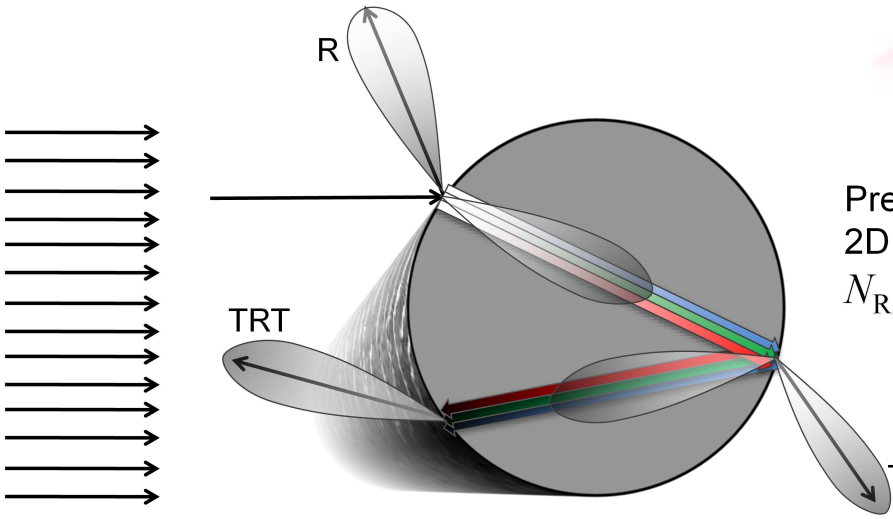
Azimuthal Scattering (N)



Azimuthal Scattering (N)



Azimuthal Scattering (N)



Precompute
2D tables for
$N_{\mathrm{R,TT,TRT}}(\theta_{in}, \varphi)$

Azimuthal Scattering (N)

- Precomputing $N_{\mathrm{R,TT,TRT}}(\theta_{in}, \varphi)$
  - For all θ_{in} ,
  - For all light entry positions along the hair thickness,
  - Trace a ray through the circular hair strand
  - Compute fraction of R and T using Fresnel factors at each hit
  - Find R, TT, and TRT positions on the tables
  - Compute the distance travelled in hair d for TT and TRT
  - Attenuate TT and TRT based on colored absorptions and d
  - Distribute TT and TRT to neighboring cells based on d to approximate scattering inside the hair volume.

Physically-based Hair Shading

- Shading equation

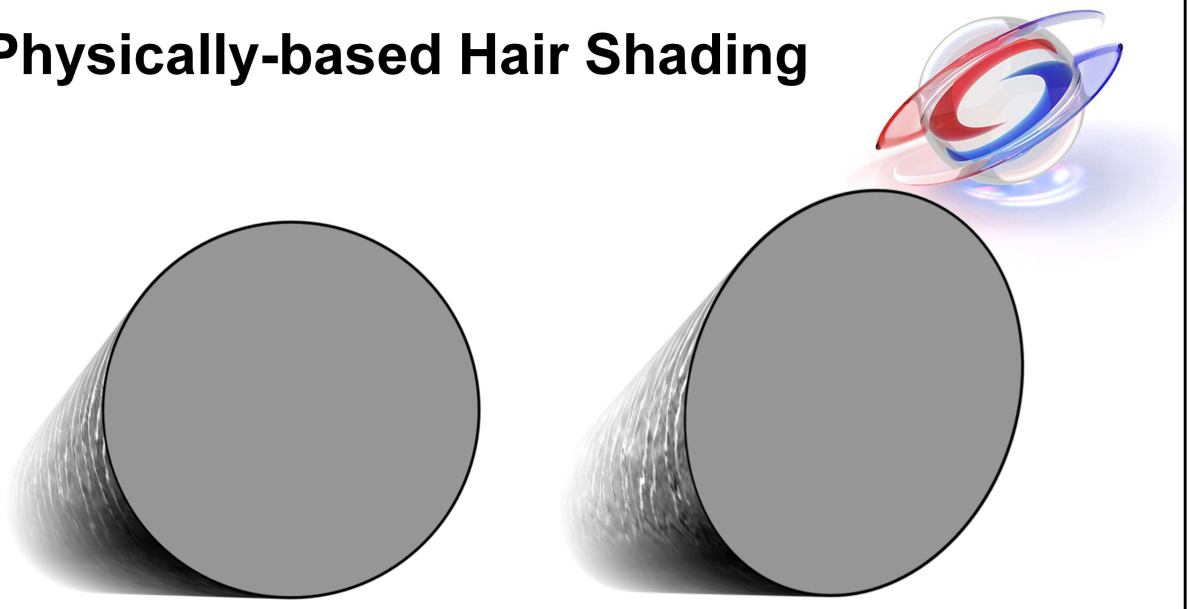
$$L_{out} = D \cos\theta_{in}\ L_{in}\ S(\mathbf{L},\mathbf{V})$$

Thickness factor ← → Scattering function

$$S(\mathbf{L},\mathbf{V}) = \frac{M_R N_R + M_{TT} N_{TT} + M_{TRT} N_{TRT}}{\cos\left(\frac{\theta_{out} - \theta_{in}}{2}\right)}$$

$M_R,\ M_{TT},\ M_{TRT}$ are longitudinal scattering and $N_R,\ N_{TT},\ N_{TRT}$ are azimuthal scattering

Physically-based Hair Shading



Circular cross-section

Elliptic cross-section

Elliptic Cross-Section

- Shifts the locations of TRT highlights
- TRT highlights depends on
  - Eccentricity
  - Hair orientation
- Simple approximation
  - Shifting φ values using a noise function


Physically-based Shading
+ Diffuse Color from Kajiya-Kay


Fake Diffuse Color
Multiple Scattering

Hair Shading

- Simple Hair Shading
- Physically-based Hair Shading
- Improving Shading Performance

Improving Shading Performance

- Precomputed tables for N
  - Combine R, TT, and TRT into one/two textures
  - Use very small textures
- Precomputed tables for M
  - May or may not be beneficial depending on
    - GPU hardware
    - The number of other math and texture operations in the fragment shader

Improving Shading Performance

- Computation in vertex shader
  - Some values can be computed in the vertex shader instead of the fragment shader
- Fake physically-based shading by
  - Using the simple model of Kajiya-Kay
  - Adding TT and TRT highlights
  - **DOES NOT WORK WELL!**

Summary of Hair Shading

- Recommended method
  - Simple hair shading for dark (black) hair
  - Physically-based shading for light colored hair
    - Precomputed tables for N
    - On the fly computation for M

Hair Shadows

Advanced Techniques in Real-time Hair Rendering and Simulation
SIGGRAPH 2010 Course

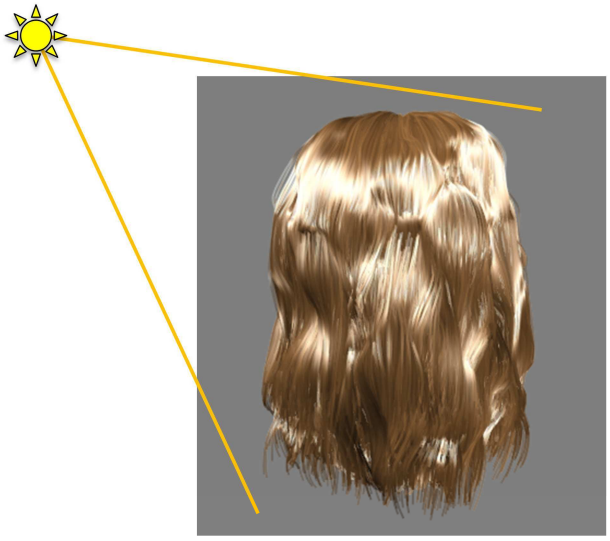
Cem Yuksel
Cyber Radiance and Texas A&M University

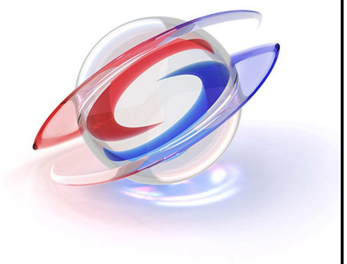
Hair Shadows

Hair Shadows

- Shadow Maps
- Transparent Shadow Mapping for Hair
- Shadow Filtering
- Simplified Shadow Maps for High Performance

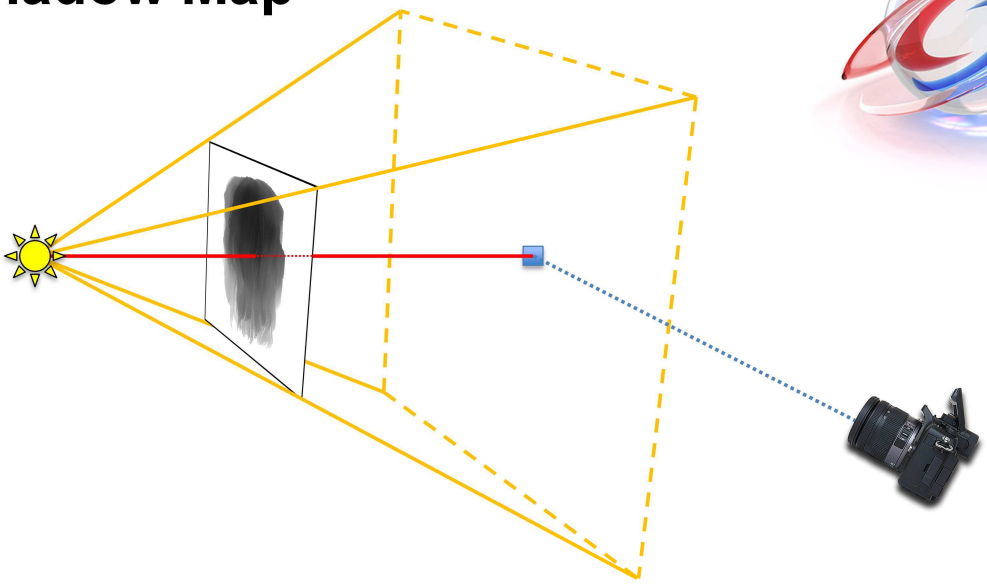
Shadow Maps



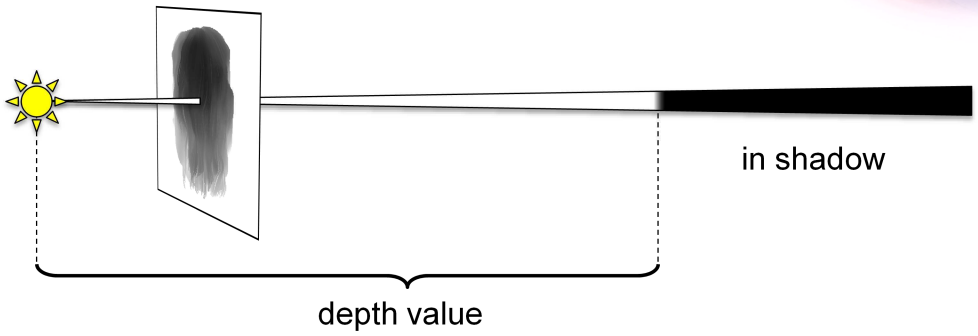


Depth Map

Shadow Map

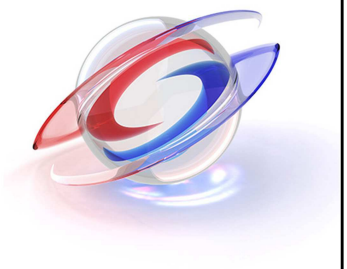


Shadow Map



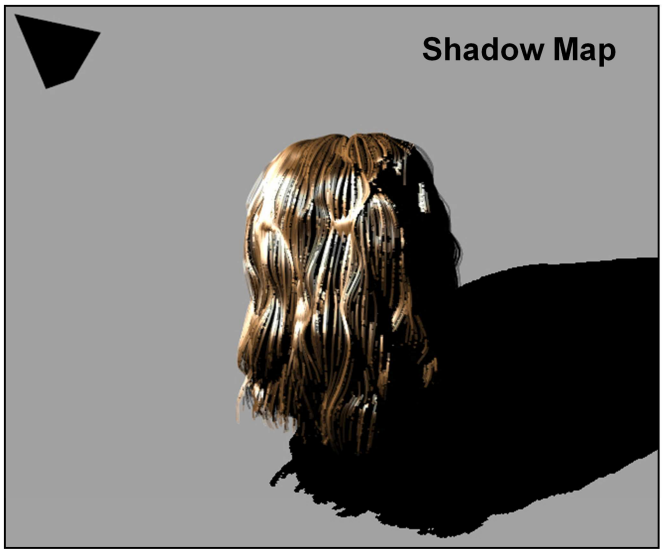
depth value

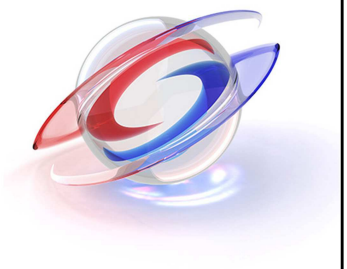
Shadow Maps



Depth Map

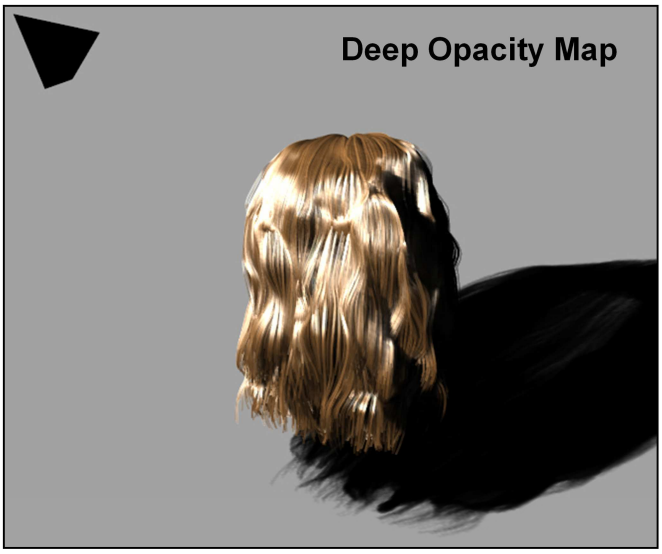
Shadow Maps

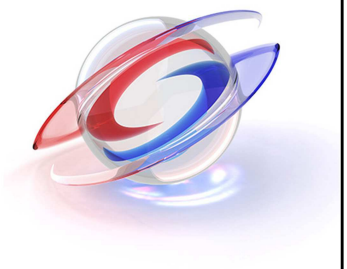




Depth Map

Shadow Maps



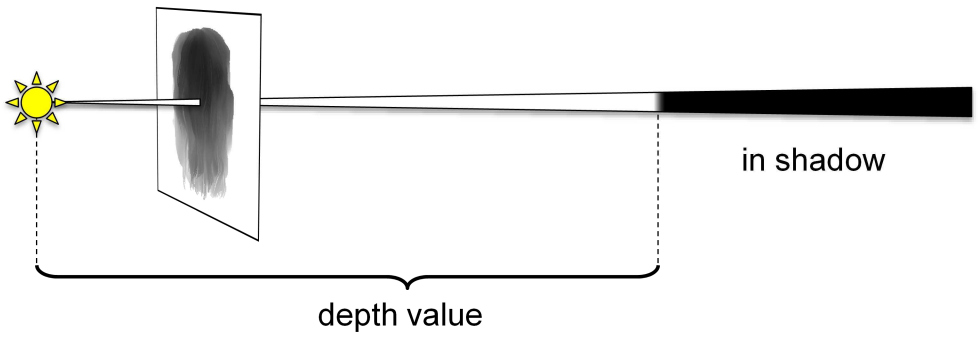


Depth Map

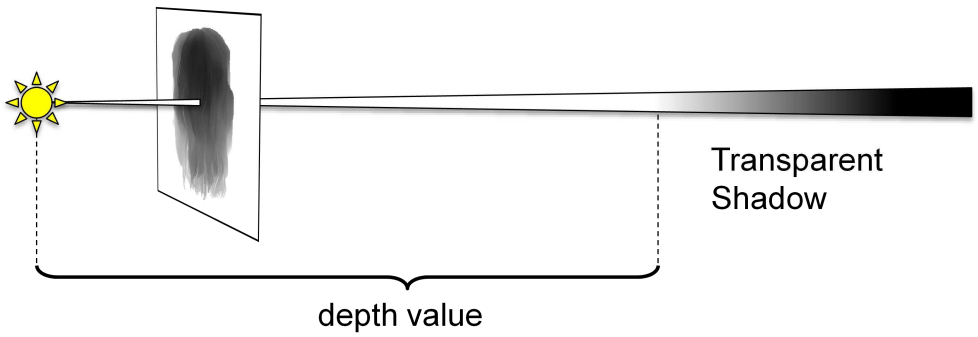
Hair Shadows

- Shadow Maps
- Transparent Shadow Mapping for Hair
- Shadow Filtering
- Simplified Shadow Maps for High Performance

Shadow Maps

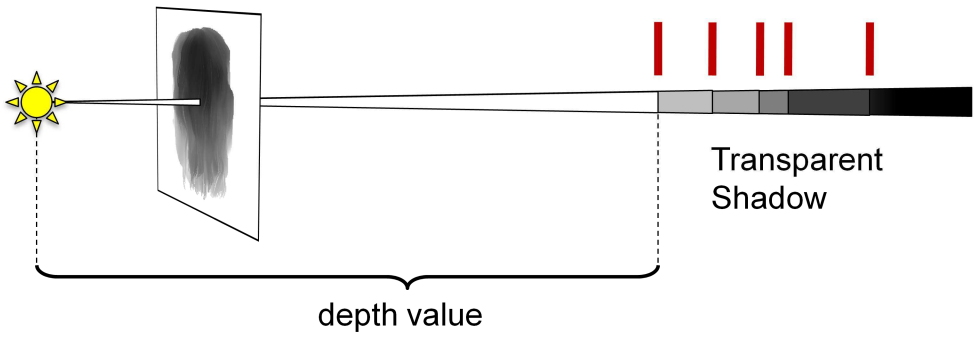


Transparent Shadows



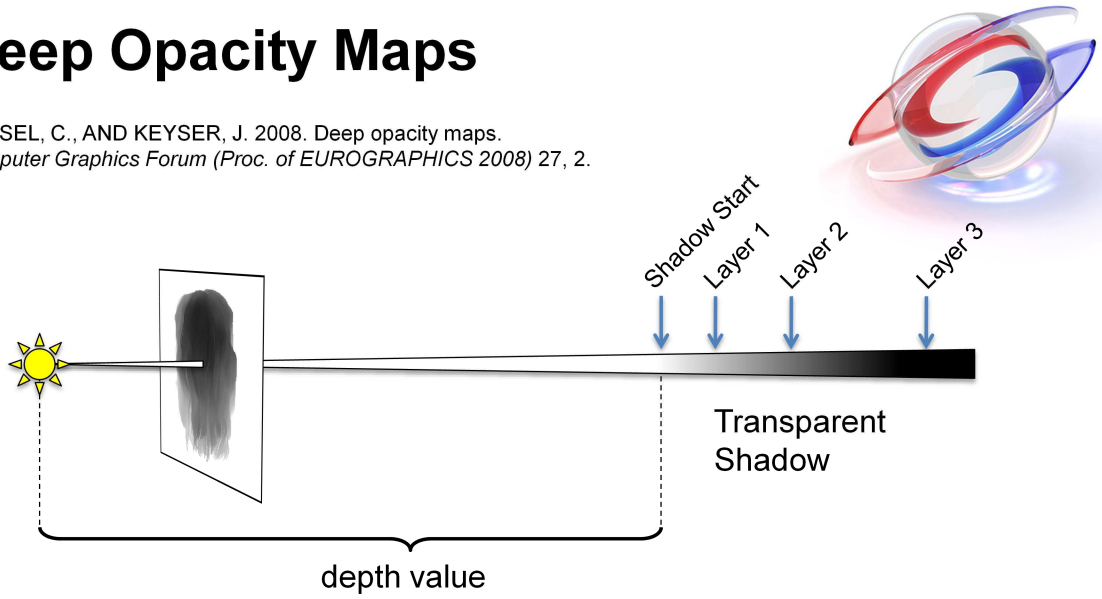
Deep Shadow Maps

LOKOVIC, T., AND VEACH, E. 2000. Deep shadow maps.
In Proceedings of ACM SIGGRAPH 2000.



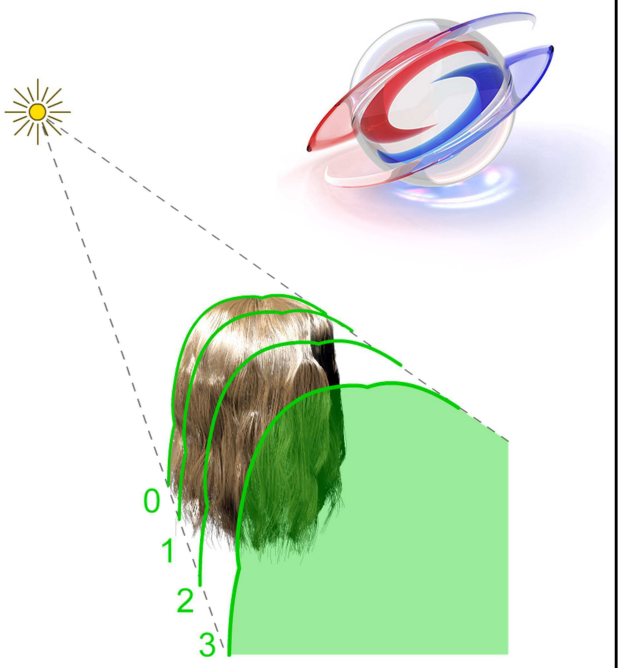
Deep Opacity Maps

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



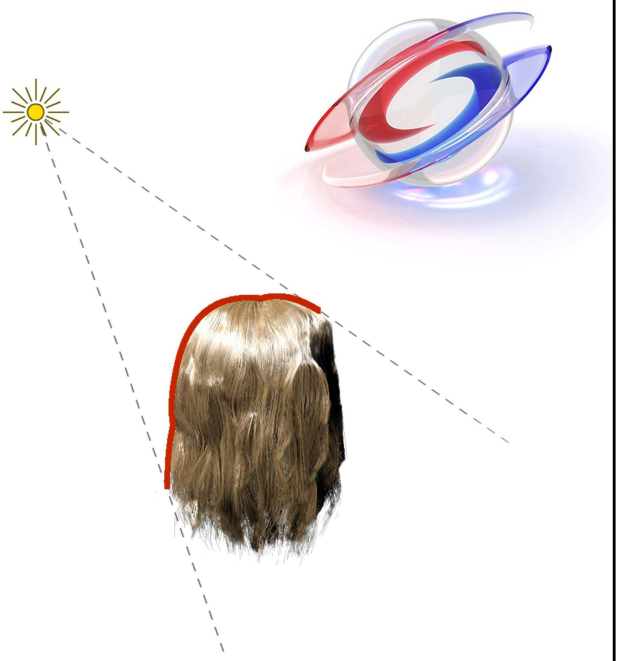
Deep Opacity Maps

- Layers take hair's shape
- High quality hair shadows
- Computation
  - Two pass algorithm
  - High performance
  - Memory efficient



Deep Opacity Maps

- **Pass 1:** Depth Map
  - z_0 per pixel



Deep Opacity Maps

- **Pass 2:** Opacity Map

  Layers:

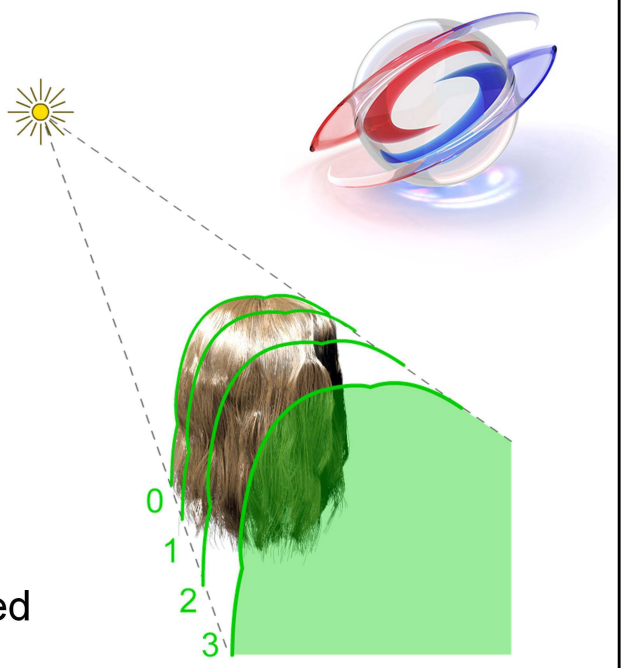
  $z_0 \rightarrow z_0 + d_1$

  $z_0 + d_1 \rightarrow z_0 + d_2$

  $z_0 + d_2 \rightarrow z_0 + d_3$

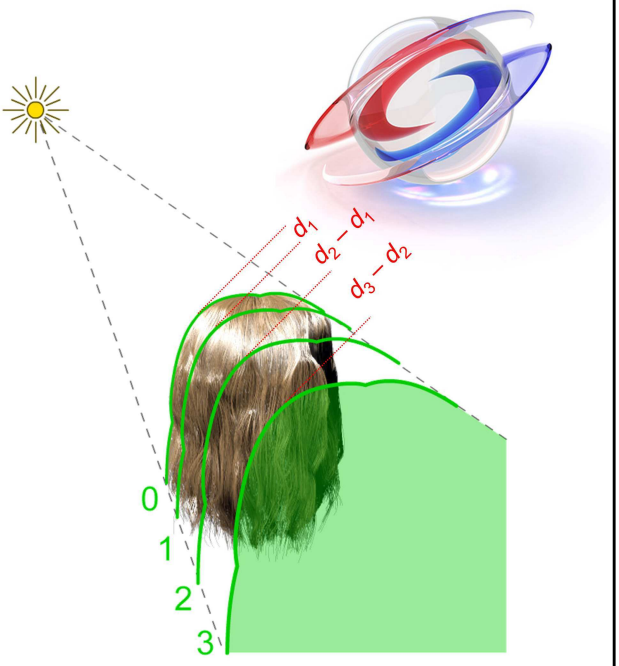
  …

  $d_1, d_2, d_3, \ldots$ are user defined



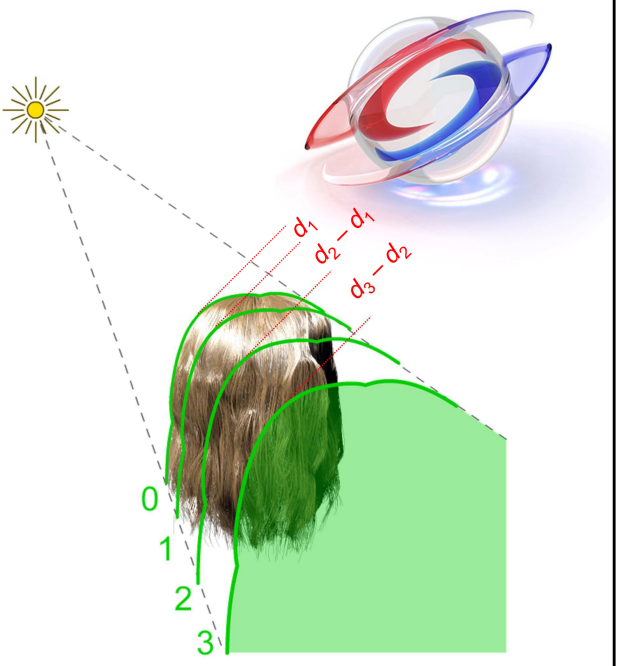
Deep Opacity Maps

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



Deep Opacity Maps

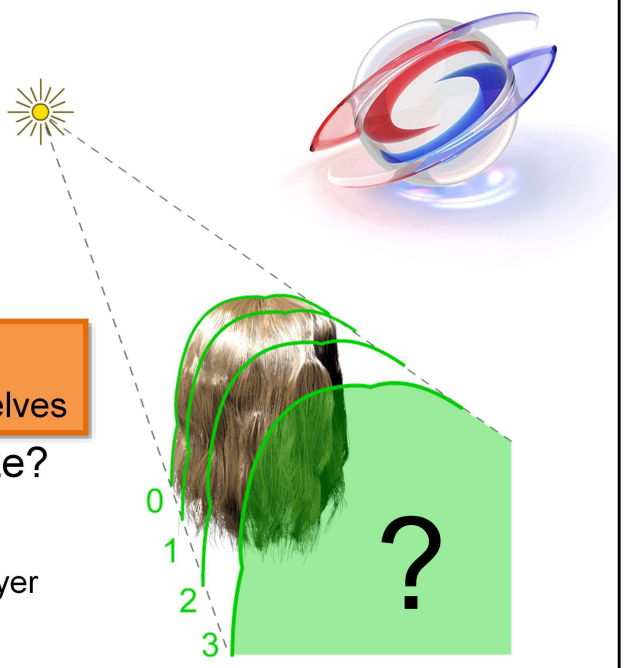
- Layer Sizes
  - s = d_1
  - Alternatives:
    - s, s, s, s, … (constant)
    - s, 2s, 4s, 8s, … (powers of 2)
    - s, s, 2s, 3s, 5s, .. (Fibonacci)
    - s, 2s, 3s, 4s, … (linear)



Deep Opacity Maps

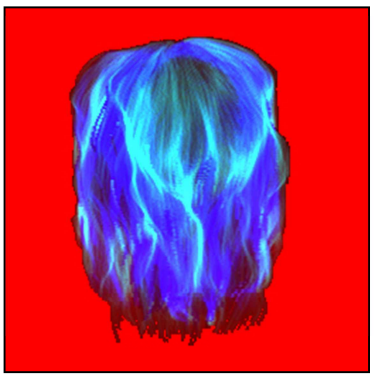
- Beyond the last layer
  - Ignore?
    - Won't cast shadows
  - Add to the last layer?
    - Cast shadows onto themselves
  - Increase the last layer size?
    - Reduce accuracy

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

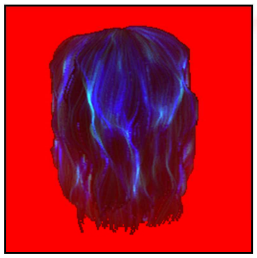


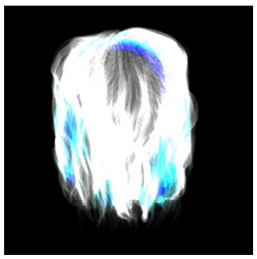
Deep Opacity Maps

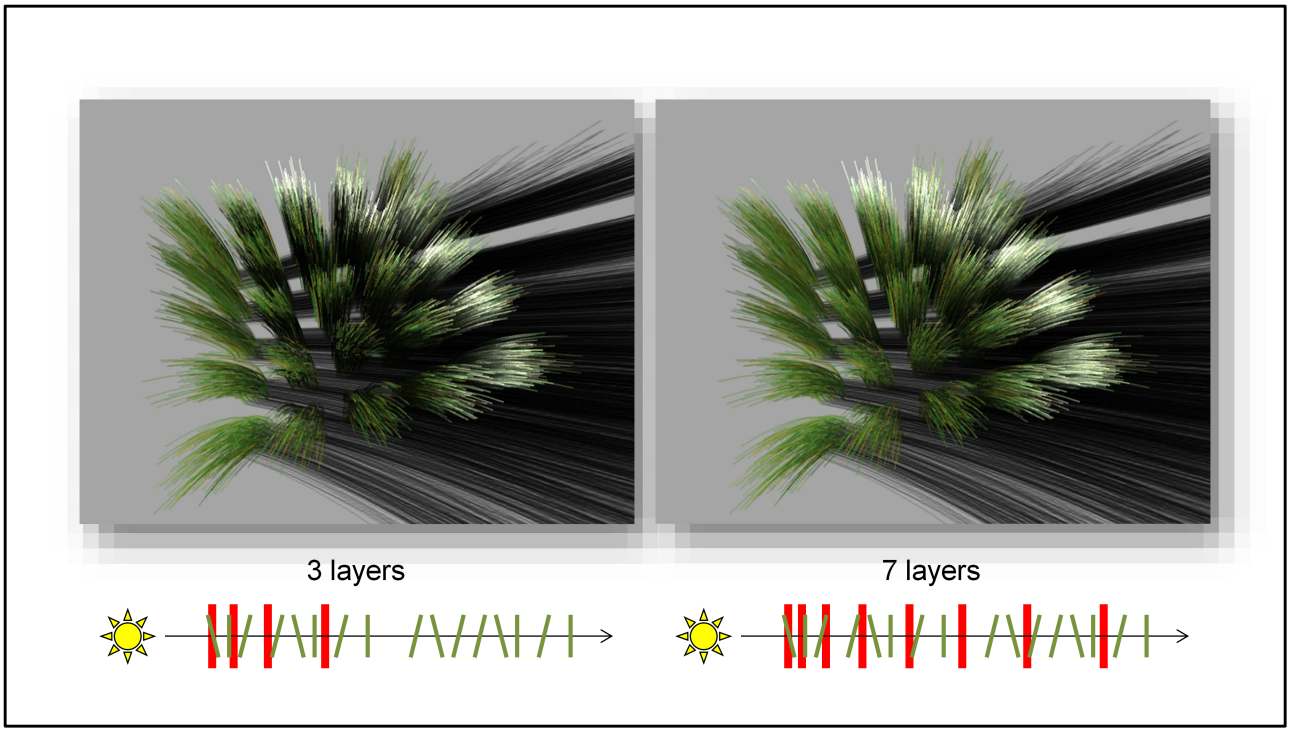
- 7, 11, 15… opacity layers
  - **Multiple Draw Buffers**

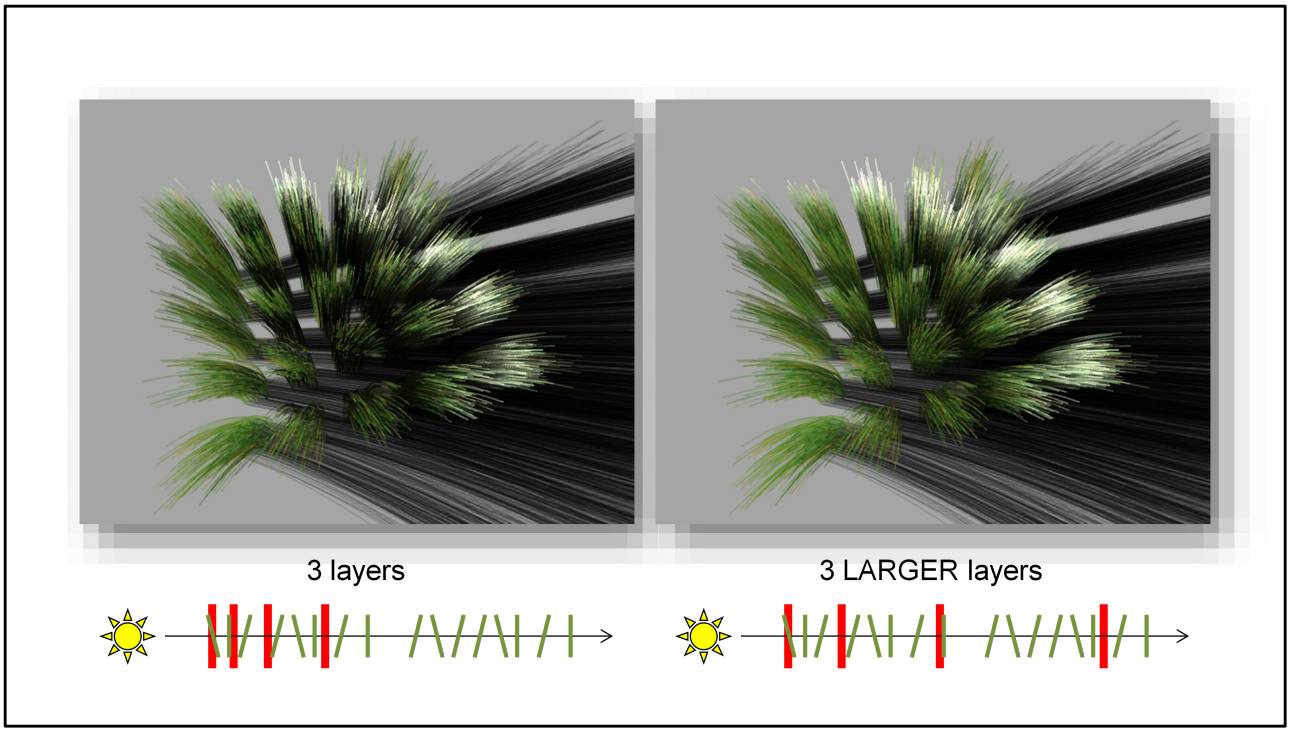
R_1: depth (z_0)
G_1: layer 1 opacity
B_1: layer 2 opacity
A_1: layer 3 opacity
} Texture 1

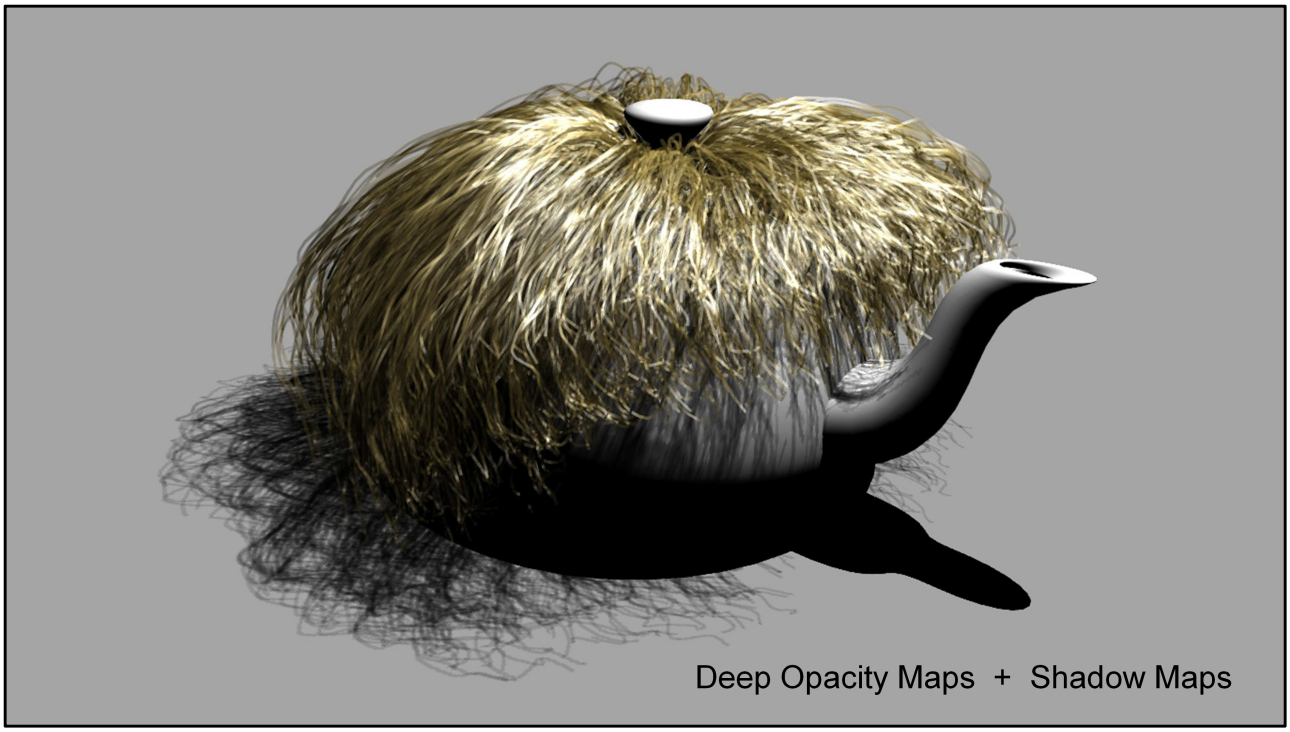
R_2: layer 4 opacity
G_2: layer 5 opacity
B_2: layer 6 opacity
A_2: layer 7 opacity
} Texture 2






3 layers
7 layers


3 layers
3 LARGER layers


Deep Opacity Maps + Shadow Maps

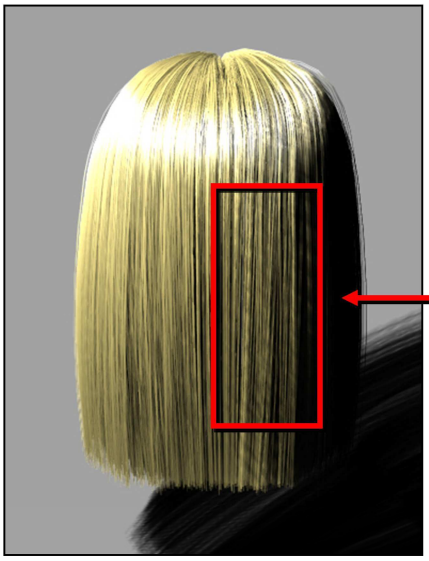
Shadow Map & Deep Opacity Map

- Separate
  - Deep Opacity Map for hair shadows
  - Shadow Map for object (non-hair) shadows
- Combined
  - Deep Opacity Map for both hair and object shadows
  - The depth map should include both hair and objects
  - The opacity map should include object opacity

Hair Shadows

- Shadow Maps
- Transparent Shadow Mapping for Hair
- Shadow Filtering
- Simplified Shadow Maps for High Performance

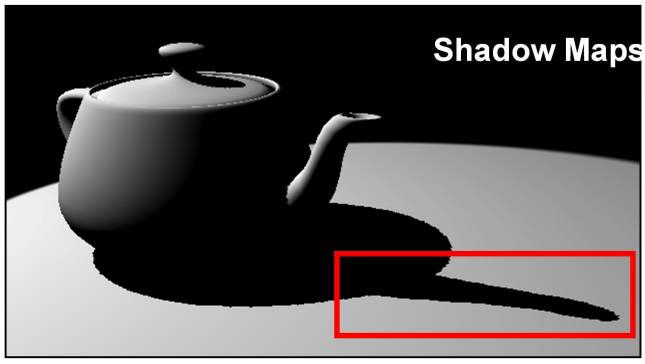
Shadow Filtering



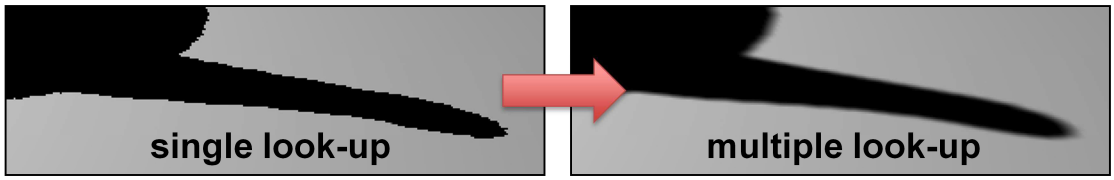
- Flickering in animation

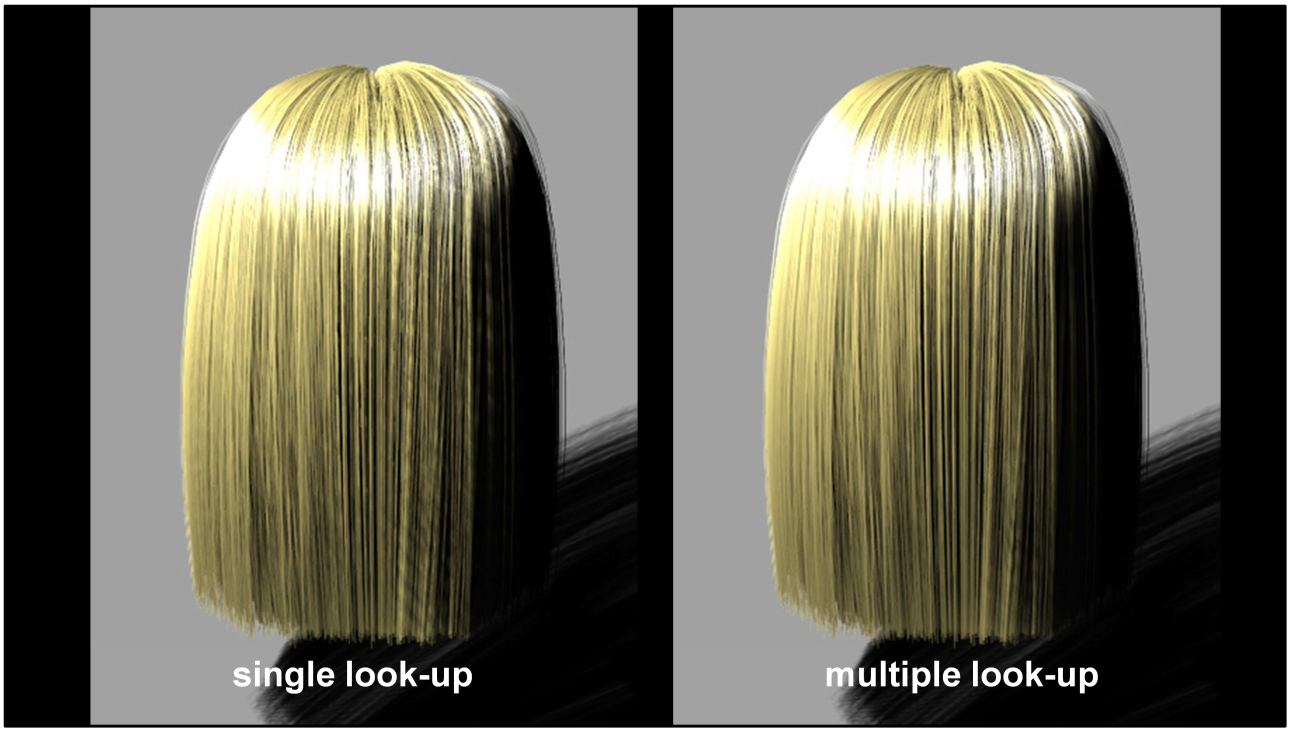
Staircase Artifacts!

Shadow Filtering

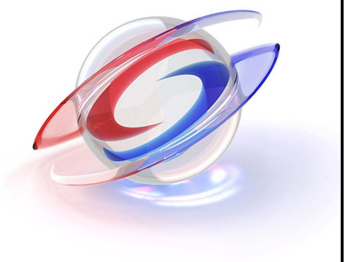


- Can’t pre-filter depth maps!

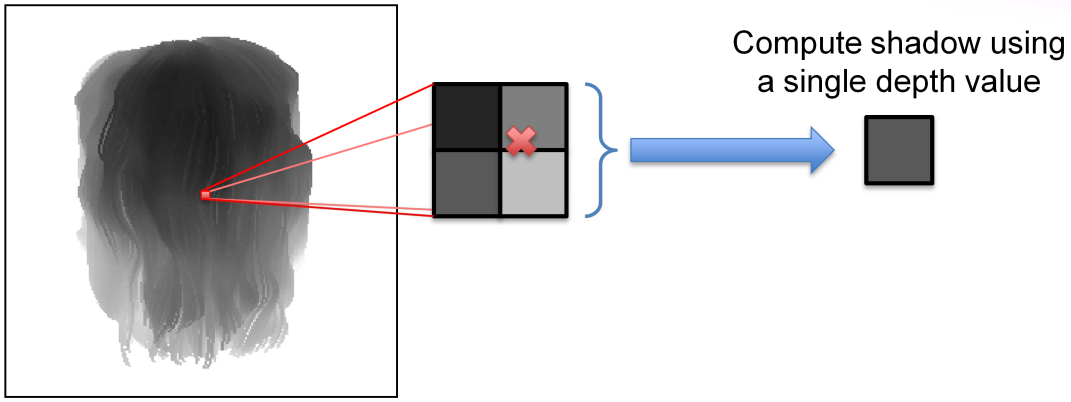



single look-up
multiple look-up

Shadow Filtering

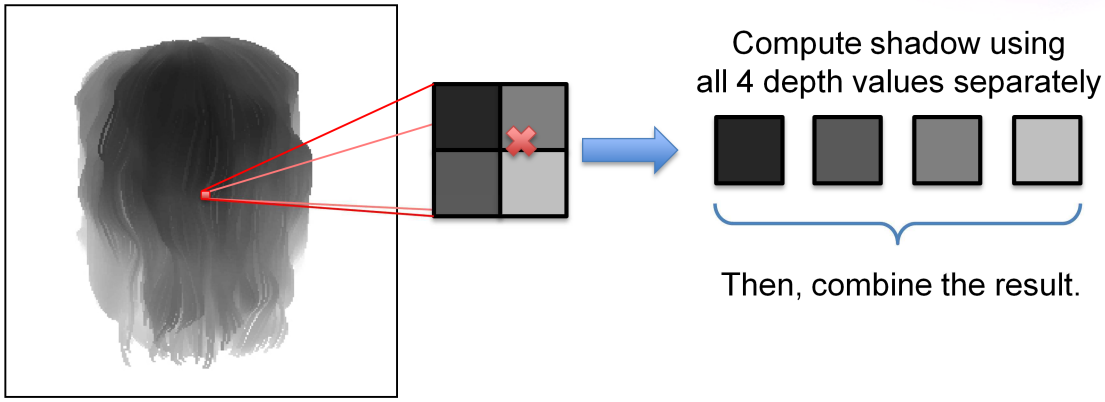


- No Filtering

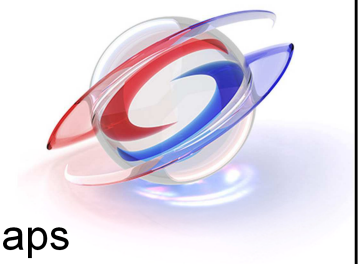


Shadow Filtering

- Percentage closer filtering
  - Standard filtering method for shadow maps



Shadow Filtering

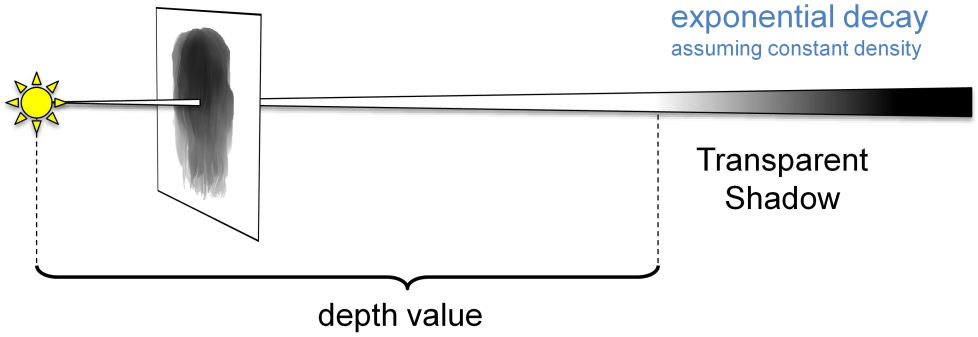


- Percentage closer filtering
  - Standard filtering method for shadow maps
- For better results use more samples (such as 16).
  - Performance vs. quality

Hair Shadows

- Shadow Maps
- Transparent Shadow Mapping for Hair
- Shadow Filtering
- Simplified Shadow Maps for High Performance

Simplified Shadow Maps for Hair



Simplified Shadow Maps for Hair

- Depth map only
- No opacity Maps
- Higher performance
- Lower accuracy
- Lower quality – more flickering

Summary of Hair Shadows

- Recommended method
  - Deep Opacity Maps
  - 3 opacity layers only
    - Combined with depth map, placed on a single texture
  - Percentage closer filtering if desired
  - Avoid computing shadows for each fragment
    - Improves performance
    - Reduces shadow filtering artifacts

Multiple Scattering in Hair

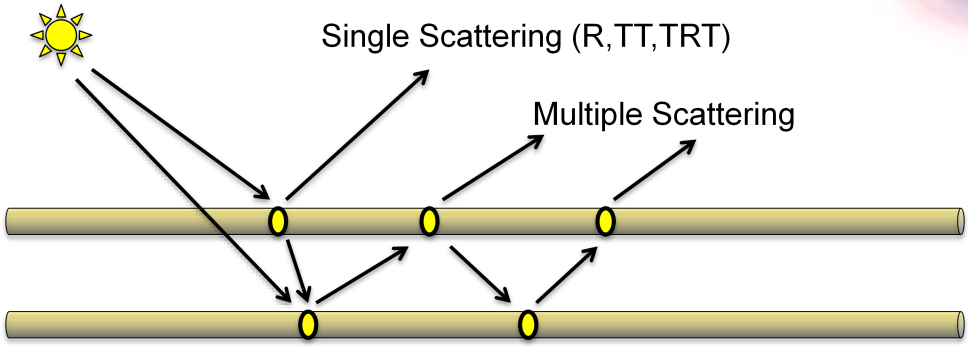
Advanced Techniques in Real-time Hair Rendering and Simulation
SIGGRAPH 2010 Course

Cem Yuksel
Cyber Radiance and Texas A&M University

Multiple Scattering in Hair

- Introduction to Multiple Scattering
- Dual Scattering Approximation
- Implementation Notes

Introduction




Physically-based Shading
Multiple Scattering


Fake Diffuse Color
Multiple Scattering

Multiple Scattering

- Important for computing hair color
- Subtle color variations
- Color changes due to changes in
  - Lighting
  - View direction
  - Hair orientation
- Natural look for hair

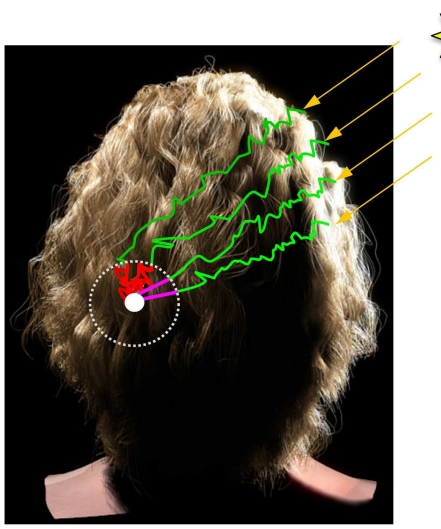
Multiple Scattering in Hair

- Introduction to Multiple Scattering
- Dual Scattering Approximation
- Implementation Notes

Dual Scattering Approximation

- Fast multiple scattering computation
- Implementation is similar to Deep Opacity Maps for hair shadows
- Various *theoretical simplifications*
- Small pre-computed tables.

Dual Scattering Approximation



- Multiple Scattering
  - Global Multiple Scattering
    - ≈ Colored shadows
  - Local Multiple Scattering
    - ≈ Extra factor in the shader

Dual Scattering Approximation

- Shading equation

$$L_{out} = \underbrace{L_{direct}\ f_{direct}} + \underbrace{L_{global} f_{global}} + \underbrace{(L_{direct} + L_{global}) f_{local}}$$

direct light **X** *single scattering*

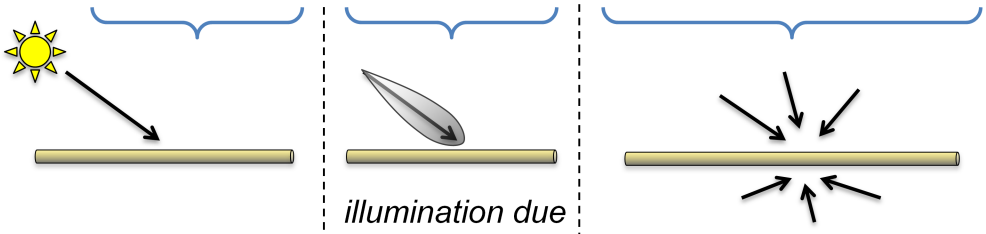
total light **X** *local scattering*

scattered light **X** *multiple scattering*

Dual Scattering Approximation

- Shading equation

$$L_{out} = \underbrace{L_{direct}\ f_{direct}} + \underbrace{L_{global} f_{global}} + \underbrace{(L_{direct} + L_{global}) f_{local}}$$

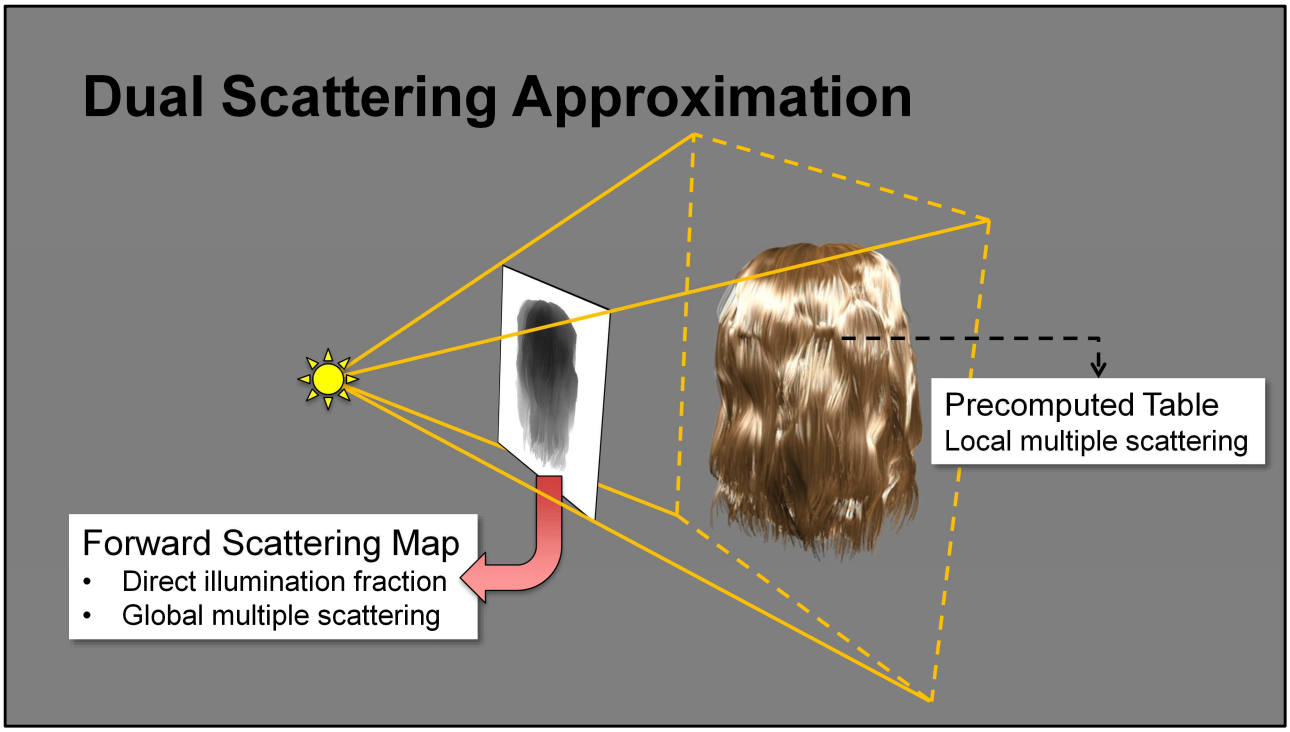


direct illumination

illumination due to global multiple scattering

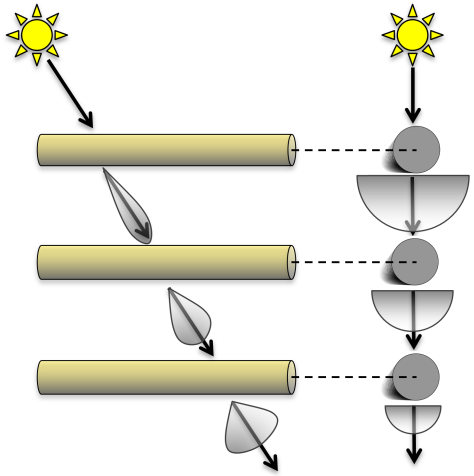
Illumination due to local multiple scattering

Dual Scattering Approximation



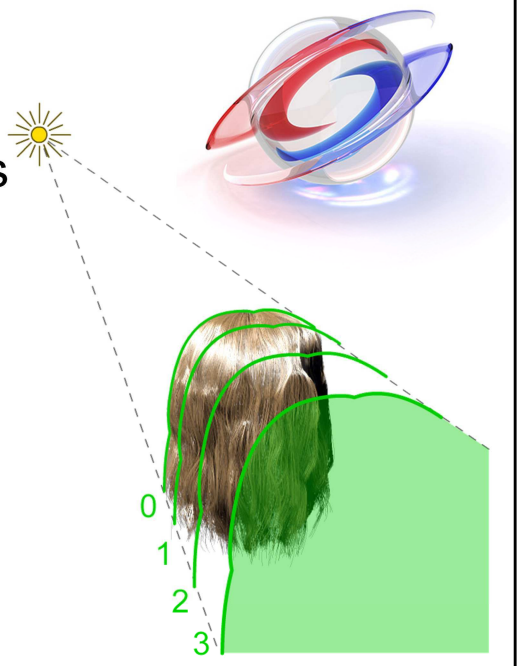
Global Multiple Scattering

- Each time light goes through a hair strand
  - Intensity decreases
  - Longitudinal spread slightly increases
  - Azimuthal spread drastically increases



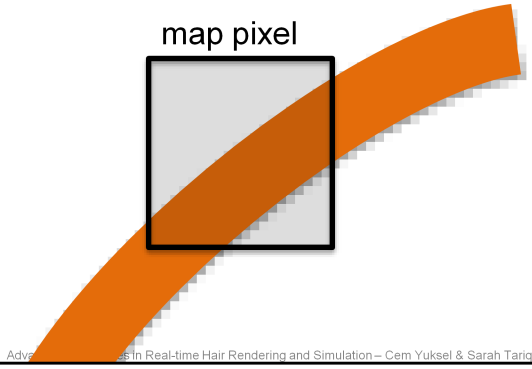
Forward Scattering Map

- Similar to Deep Opacity Maps
- At each pixel
  - Depth value
  - For each layer
    - Direct illumination fraction
    - Global scattering intensity
    - Longitudinal spread

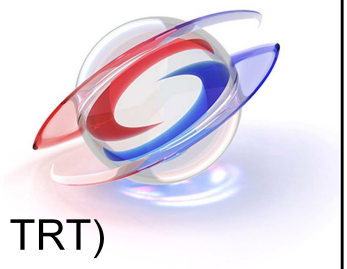


Global Multiple Scattering

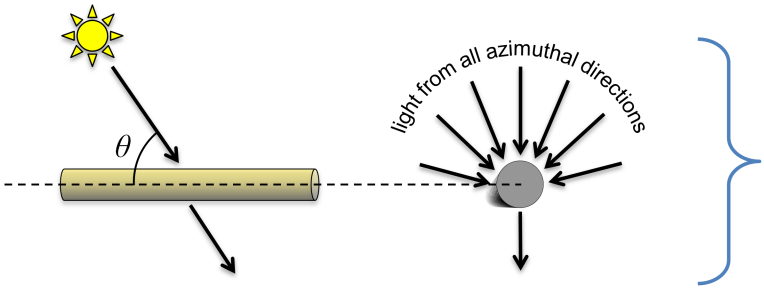
- Direct illumination fraction
  - Fraction of a shadow pixel that is not occupied by the shadowing hair strand



Global Multiple Scattering



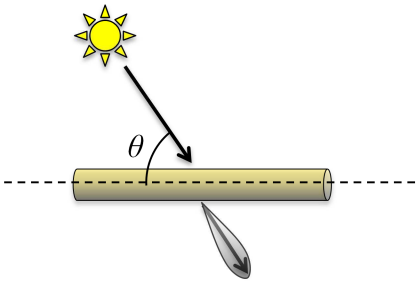
- Global scattering intensity
  - Forward scattering (including R, TT, and TRT)
  - Due to light from all azimuthal directions



Using precomputed scattering function depends on θ only

Global Multiple Scattering

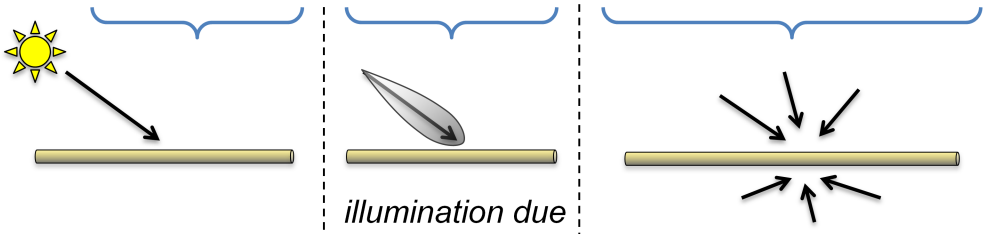
- Longitudinal spread
  - Average spread value for a given θ
  - Precomputed using spread of M_R, M_{TT}, and M_{TRT}



Dual Scattering Approximation

- Shading equation

$$L_{out} = L_{direct}\ f_{direct} + L_{global} f_{global} + (L_{direct} + L_{global}) f_{local}$$



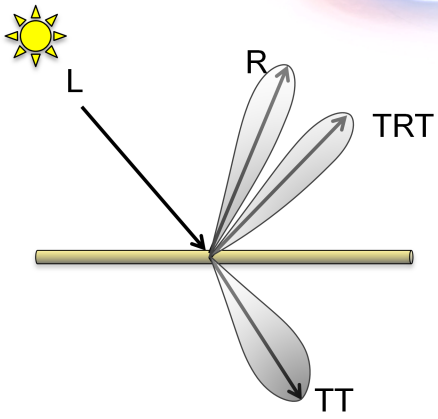
direct illumination

illumination due to global multiple scattering

Illumination due to local multiple scattering

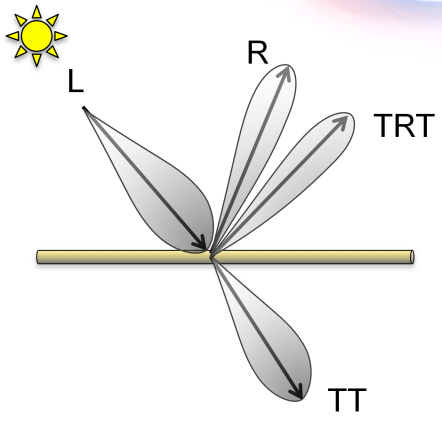
Dual Scattering Approximation

- Direct illumination
  - Standard hair shading
  - Using direct illumination fraction



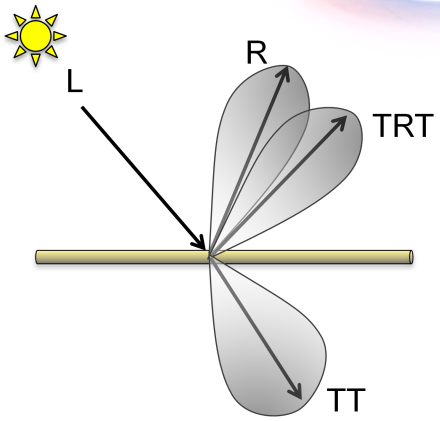
Dual Scattering Approximation

- Illumination due to global multiple scattering



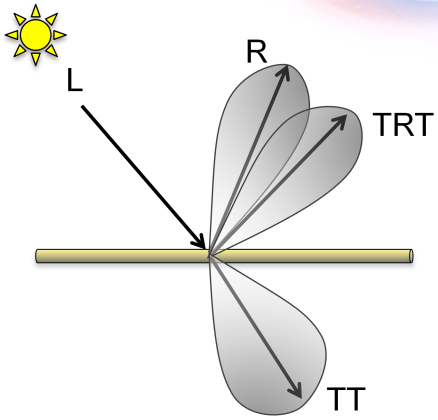
Dual Scattering Approximation

- Illumination due to global multiple scattering
  - Increase spread of M with spread of global scattering



Dual Scattering Approximation

- Illumination due to global multiple scattering
  - Increase spread of M with spread of global scattering
  - Different tables for N using isotropic azimuthal illumination



Dual Scattering Approximation

- Illumination due to local multiple scattering
  - Assuming disciplined hair strands
    - i.e. hair strands are parallel
  - Precomputed 3D table based on
    - θ_{in} : longitudinal light direction
    - θ_{out} : longitudinal view direction
    - Longitudinal spread due to global scattering

Multiple Scattering in Hair

- Introduction to Multiple Scattering
- Dual Scattering Approximation
- Implementation Notes

Implementation Notes

- Single Scattering Component
  - Compute $M_{R,TT,TRT}$ using the Gaussian function
  - Lookup $N_{R,TT,TRT}$ from precomputed tables

Implementation Notes

- Global Mult. Scattering Component
  - Compute $M^G_{R,TT,TRT}$ using the Gaussian function and global multiple scattering spread (s)

$$M^G_R = \text{Gauss}\left(\beta_R + \mathbf{s},\ \theta_h - \alpha_R\right)$$

  - Lookup $N^G_{R,TT,TRT}$ from precomputed tables
    - Prepared using isotropic azimuthal illumination
    - Smaller tables are sufficient

Implementation Notes

- Local Mult. Scattering Component
  - Precomputed table
  - 3 independent variables
    - 3 x 1D table
    - 1 x 3D table
    - 1 x 1D table & 1 x 2D table

Implementation Notes

- Varying hair color
  - All precomputed tables depend on hair color (i.e. absorption coefficients)
  - Precompute tables for a few colors only
  - Interpolate in-between values
  - No need to have separate tables for red, green, blue

Summary of Multiple Scattering in Hair

- Multiple scattering is important for realism
- Dual scattering is
  - Fast
  - Yet, slower than simple shadow computation
  - Uses significantly more memory
    - Precomputed tables
    - Forward scattering map
- Use multiple scattering if you need it

Hair Dynamics for Real-time Applications

Advanced Techniques in Real-time Hair Rendering and Simulation
SIGGRAPH 2010 Course

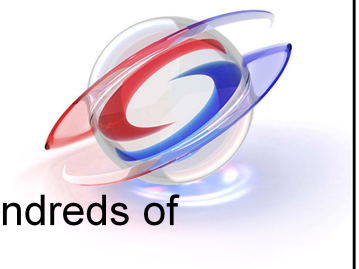
Sarah Tariq
NVIDIA

Cem Yuksel
Cyber Radiance and Texas A&M University

Hair Dynamics for Real-Time Applications

- Overview of Hair Simulation Techniques
- Fast Simulation of Hair on the GPU
- Handling Inter Hair Collisions

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

Hair is a difficult material to simulate - a typical hairstyle consists of a hundred of thousand of thin inextensible strands.

There are a number of major issues that we have to deal with.

The first is the dynamics of an individual strand. Each hair strand has a complex non-linear behavior

Then we have to deal with the collective behavior of all the strands. Once we put hair strands together they have complex interactions which lead to distinctive changes in both the motion and shape of the hair style

Finally, we have to deal with both these simulation constraints in real time

Overview of Hair Simulation Techniques

- Many different techniques, ranging in complexity, stability and accuracy, for example:
  - Rigid multi body serial chains [Hadap06]
  - Super helices[Bertails06]
  - Mass spring system [Rosenblum91,Selle08]
- For real time we are using a particle constraint system

There are many different methods for simulating hair, ranging in complexity, stability and accuracy. Some examples of simulation methods include mass and spring systems, rigid body chains and super helices.

Rigid multi body chains simulate hair as connected rigid joints, and you can solve this sort of system using featherstone's algorithm for example. They are great for getting natural motion of hair, but have some drawbacks, for example they have difficulty supporting implicit integration, the collision response is difficult, and it is difficult to parallelize them.

Mass spring systems are actually a method that a lot of people use for simulating hair, and the advantage these systems have is that they can be used with implicit integration, they have no restrictions with collision handling and they are well understood.

The approach that we are going to be using for real time is sort of similar to mass spring systems, and that is a particle constraint system. This representation is fast, simple and stable, but not very accurate. We choose it because of its speed and also its ease of implementation on a the parallel architecture of a GPU.

Hair Dynamics for Real-Time Applications

- Overview of Hair Simulation Techniques
- Fast Simulation of Hair on the GPU
- Handling Inter Hair Collisions

Simulation of Hair on the GPU

- Hair simulated as a particle constraint system
  - Each strand is represented as a number of particles
  - Distance constraints between hair particles maintain hair length
  - Angular forces at each hair vertex maintain hair shape
  - Collision constraints keep hair particles outside obstacles

This is an overview of the simulation that we are going to be talking about.

Each hair strand is simulated as a particle constraint system.

Each hair particle is subject to forces like gravity, wind etc.

Links between neighboring hair particles are simulated as distance constraints. These constraints help the hair maintain its length.

Angular forces at each hair vertex help the hair maintain its shape.

And collision constraints help the hair particles stay out side of collision obstacles.

One simulation step

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

This is a high level overview of the hair strand simulation.

We start off by simulating the external forces like wind in the scene.

Then for each hair particle we accumulate all the forces affecting it and integrate the particle forward, updating its position and velocity.

After this we iteratively satisfy a number of constraints, including distance, angular and collision constraints.

Finally we can optionally resolve or account for the inter-hair collisions.

Lets now talk about how we would implement some of these steps efficiently in parallel on the GPU.

Add forces and integrate

- no inter particle dependencies
  - Each particle is simulated in a separate thread
- Update:
  - If particle is attached then update by transform and return.
  - move particle out of any obstacles
  - accumulate forces (for example from wind, gravity and grooming constraints)
  - integrate position using verlet integration

We start off with integrating forward the position of each particle. Since there is no inter particle dependencies in the integrate step we can easily run each particle update on a separate thread.

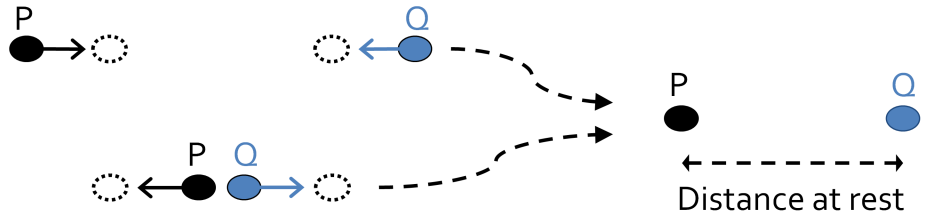
The steps that we follow in the integration code are very simple.

We start off by updating the positions of any vertices that are attached – for example if a particle is attached to the scalp of the model then this particle does not need to be integrated – all we need to do is update the position of the particle by the transform of the scalp and return.

For all free particles we accumulate all the forces acting on them and then we integrate the particles forward using verlet integration.

Distance Constraints

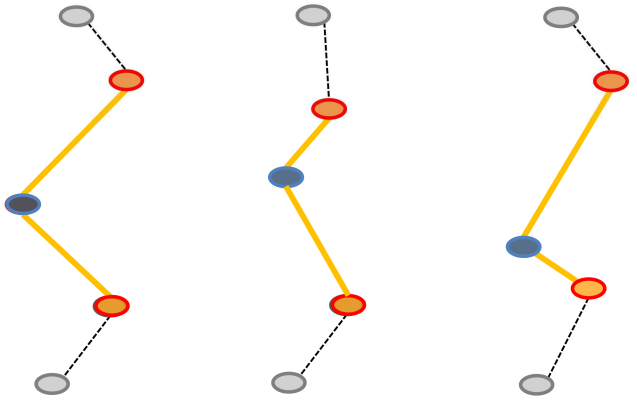
- Links between consecutive vertices in a strand are represented by Distance Constraints
- A distance constraint DC(P, Q) between two particles P and Q is enforced by moving them away or towards each other:

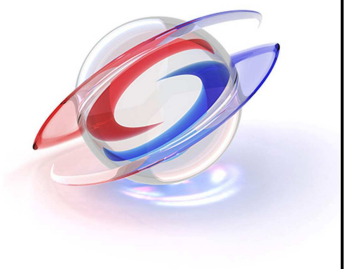


The link between two given vertices in a strand is simulated using a distance constraint.

A distance constraint of distance D between two particles is resolved by moving the two particles equally so that the distance between them is D.

Distance Constraints





?

Two constraints acting on one particle

Result of applying first constraint

Result of applying second constraint

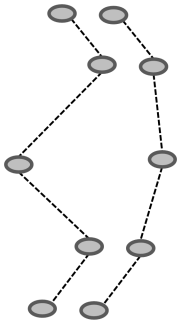
What happens if we apply them both in parallel?
Results are undefined

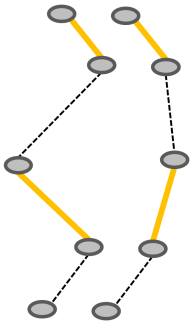
In order to satisfy a given distance constraint we need to move the positions of two vertices.

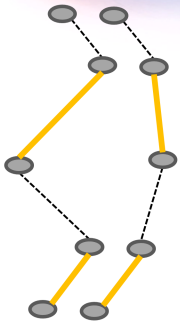
since each single particle is subject to two distance constraints, we cannot update all the constraints in parallel. Otherwise we would end up trying to modify the same particle from two parallel threads with undefined results.

Instead we need to batch up the constraints into non-overlapping sets and update these sets sequentially

Distance Constraints







Since particles are subject to multiple constraints we cannot satisfy them all in parallel

First batch of constraints

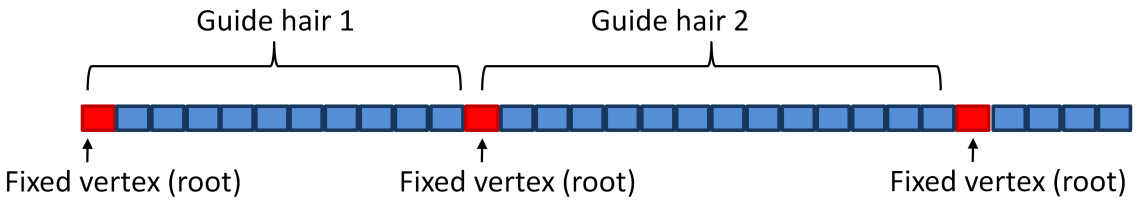
Second batch of constraints

Iterate

In order to properly update the constraints in parallel we divide them into non-overlapping sets. For this example set of strands the two sets of constraints could look like this. We solve the two sets one after the other, iterating a number of times to converge to the final solution.

Representation

- All the guide hair vertices are appended together to form one buffer, using array of structures



Our simulated hair strands are represented in one large buffer on the GPU, with all strands appended one after the other. Each strand is just a collection of vertices, and each vertex can have multiple attributes (including position, velocity, coordinate frames etc).
In order to represent all the attributes of the vertices we use an array of structures approach, i.e. we have separate buffers for position, velocity and any other attributes. This separation helps cache efficiency when we only need to access some but not all of the attributes of a hair vertex.
Note that at this point we are not stating what sort of buffer this is – that is, is it a vertex buffer, a texture buffer, a structured buffer or something else? The actual type of buffer will be based on the particular method that we use to do the actual simulation on the GPU, which we will be talking about next.

Dynamics on the GPU

- Issues:
  - How should we map the tasks to the GPU?
    - Should we run a thread in a pixel shader, vertex shader or something else? How will we write out updated data?
  - Need to be able to communicate data between threads.
    - For example if a thread updates two particles to satisfy a distance constraint then in the next iteration two more distance constraints need these updated positions.

When trying to map the simulation to the GPU there are a couple of issues that we have to resolve.

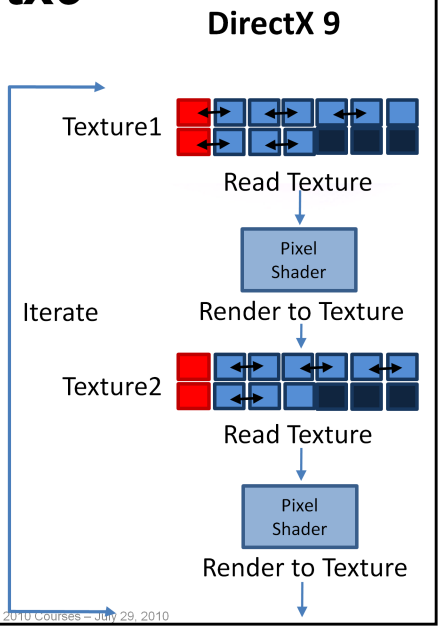
The first is, what shader stage are we going to use to run the blocks of computation. These blocks include integrating the particles forward, and updating the various constraints on them.
For example, if we want to integrate forward the positions of all the particles, should we update these positions in the vertex shader, the pixel shader, the geometry shader or something else like the compute shader?
Depending on the choice of shader stage we also have to decide how we are going to store the inputs and outputs to the simulation.

In addition, we need to share updated results between different blocks of computation, since particles in a hair strand can indirectly or directly affect each other. For example, once a distance constraint is done updating the position of two particles, two more distance constraints will need to know these updated positions. If we are using something like a pixel shader to update the constraints there is no way for the different shaders to communicate data other than writing this data out to memory.

In the next couple of slides we will go over several methods of mapping this work to the GPU. The different methods have their own advantages and disadvantages – some can only run on certain newer hardware, while others are slower or less elegant.

Dynamics on the GPU – directx9

- Directx9 Class Hardware
  - Use Render To Texture and Pixel shaders
    - Save particle attributes in a texture
    - For each iteration render a quad to the screen and run the Pixel Shader
    - each Pixel Shader updates one particle
    - Read previous particle attributes from one texture
    - write the updated particle attributes to the other texture which is bound as a Render Target



Lets start with a method that will work on any GPU with DirectX9 class support; the vast majority of GPUs in the wild today have atleast directx9 capabilities.

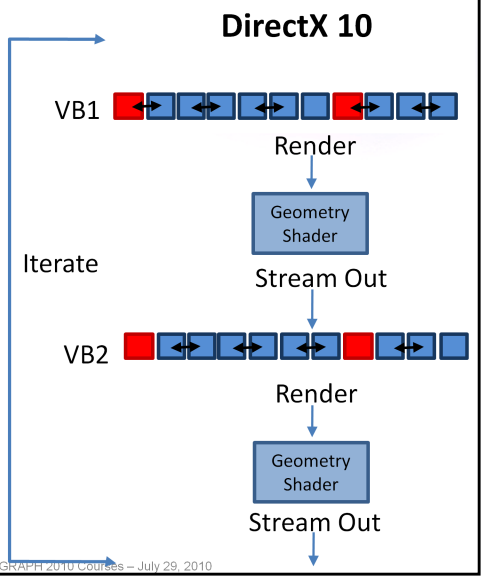
The approach we are going to be using here is a Render to Texture based approach.

In this approach we will represent the particle attributes in a texture, update the attributes in the pixel shader and write out the new attributes to another texture. The reason we have to use two textures, one to read the attributes from and the other to write the attributes to is that we are not allowed to bind the same texture to input and output. Instead we have to use two textures, in one step we read from one and write to the other and in the next step we swap the buffers. This technique is called ping ponging.

In order to run the pixel shaders we will render a quad to the screen which will contain as many pixels as the number of particles we want to update. Each pixel shader will update one particle, and it will write out the results to the Render Target, which is actually our second particle texture. At the end of an update step we swap the buffers.

Dynamics on the GPU – Directx10

- Simulation is done using the Vertex and Geometry Shader and written out using Stream Output
- Stream Out
  - Allows writing directly from Vertex/Geometry Shader to another Vertex Buffer, skipping rasterization
  - Prevents reading from and writing to the same buffer, so we have to use two buffers
- use the ping-ponging between two vertex buffers



The next method is one supported by any GPU that has Directx10 Class support. GPUs with this functionality have been around since 2006, for example NVIDIA Geforce 8 series, or ATI HD 2000 series. Our particle data will be represented as a vertex buffer, with each particle mapped to a separate vertex. We are going to be running our actual computation on the vertex shader or the geometry shader.

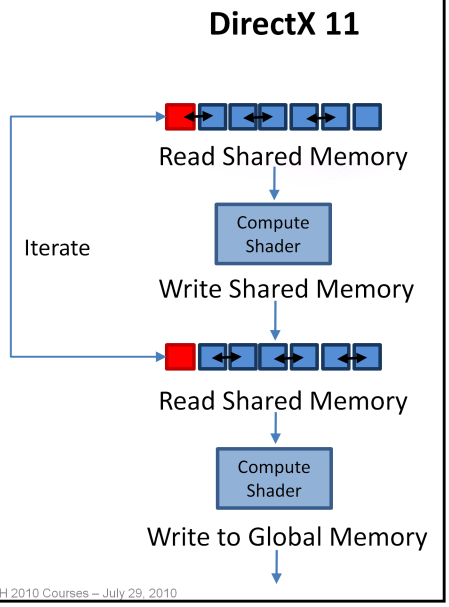
The choice of which shader to use will depend on the simulation step that we are doing – in particular, it will depend on the number of particles that our step is trying to update. For example, if we are trying to integrate the particles we only need to read one vertex and write out one vertex, which maps well to the vertex shader.

On the other hand, if we are trying to satisfy a distance constraint then our computation will modify the position of two vertices, and we cannot use a vertex shader to write out more than one vertex. Instead we will use the geometry shader for the step where we satisfy distance constraints.

The vertex or geometry shader will write out the modified data to another vertex buffer using Stream Output. Stream Out enables us to write directly from the Vertex Shader to another Vertex Buffer, skipping work like rasterization, pixel shading etc. However, we have to continue using the ping ponging approach, since Stream Out prevents us from reading from and writing to the same buffer

Dynamics on the GPU – Directx11

- Directx11 Class Hardware (for example GTX480, HD 5870)
  - Simulation is done in the compute shader
  - Particle positions and attributes are stored in Structured Buffers
    - Can read from and write to the same buffer
  - All simulation steps can be done in a single Compute Shader invocation since we can transfer intermediate results between particles by using Shared Memory



Finally, if you have access to Directx11 class hardware then you can use the compute shader. This functionality is exposed in GPUs that started shipping in the last year, for example the GTX480, and the HD 5870. The compute shader is a programmable shader designed to provide functionality to help general purpose computing.

Note that compute functionality on GPUs is not just a feature of graphics APIs like Dx11 and OpenCL. It has also been exposed through CUDA for a long time, and is now also available through OpenCL.

We will be using Structured buffers to store the particle attribute data, and these buffers can be updated in place by the compute shader – that is we can read from and write to the same buffer. This means that we don't have to do ping ponging between two buffers representing the particles, saving space.

In addition, we will be doing all the simulation steps – that is the integration, and all the iterative solving of the constraints in the same Shader invocation. This is because with compute shader we can use Shared memory.

Compute Shader

- A thread is the basic CS processing element
- Programmer gets to decide
  - how many threads are in a thread group
  - How many thread groups to execute
- All threads in a thread group can cooperate using shared memory

Hair Simulation Using Compute

- We map (one or more complete) hair strands to a CTA
  - threads in a CTA work together to update the strand, using shared memory
  - One thread will update a single particle or a single vertex

Shared memory allows all threads in a block of work can share data using on chip dedicated memory.

So, we will map all the particles of one hair strand to the same block of threads, and then the threads can run co-operatively and compute the final attributes of the particles.

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

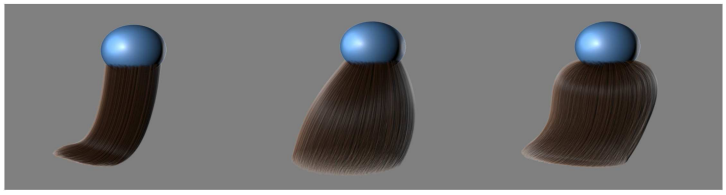
```
 //calculate forces, add them to the particles and integrate
 …

//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];
     …
}
}
```

Hair Dynamics for Real-Time Applications

- Overview of Hair Simulation Techniques
- Fast Simulation of Hair on the GPU
- 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)

After dealing with the dynamics of independent strands the next thing we have to look at is the behavior of strands in a group. The interaction of hair strands with each other gives the hair style many of its important characteristics, and is especially challenging to simulate.

Interaction between hair strands causes the hair to have more volume and gives the hair style some of its shape. In the image on the left we can see the change in the look of the hair as we increase its volume.

The motion of hair strands that are colliding is also different, and more damped, than of independent non-interacting strands.

Finally, the complex collisions and contacts between hair strands cause intricate webbing as illustrated by the image on the right.

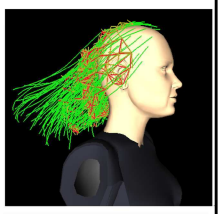
Issues

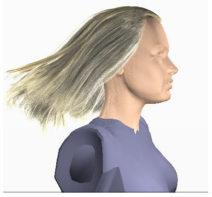
- The large number of hair strands makes processing hair self-interactions very challenging
- Can often have issues with instability and jittering, which is more obvious when hair is at rest

Simulating all these interactions for the large number of hair strands is challenging, especially when trying to get a robust solution that does not exhibit jittering or instability.

Approaches to handing collisions

- Explicit collision detection and resolution of each hair
- Wisp model - reduce the amount of interaction by only doing collisions at the wisp level
  - level Layered Wisp Model [Plante01], [Choe05]
- Inter guide hair links
  - Chang et al





Collision Resolution using inter-guide hair links

There are a number of approaches to handling inter-hair collisions.

We can explicitly simulate all the hair strands, and detect and resolve all the collisions of all the strands – this method can give good results, but is very slow and can also lead to jittering.

In order to make the simulation more tractable we can use the wisp model, which reduces the amount of interaction we have to handle by only doing collisions at the wisp level.

Approaches to handing collisions

- Continuum framework
  - Assume hair is a fluid medium and use fluid simulation techniques to handle self collisions efficiently
- Some approaches:
  - Simpler grid based smoothing
    - Bertails05 Eulerian density based
    - Petrovic Eulerian velocity based smoothing
  - Fluid simulation based approaches
    - Hadap01 Lagrangian
    - McAdams09 Hybrid

Petrovic

McAdams 09

A different approach to modeling the bulk behavior of strands is to model the hair mass as a fluid.

Hadap for example used Smooth Particle Hydrodynamics, Bertails used an Eulerian grid, and McAdams used a hybrid combination.

Inter-hair collisions

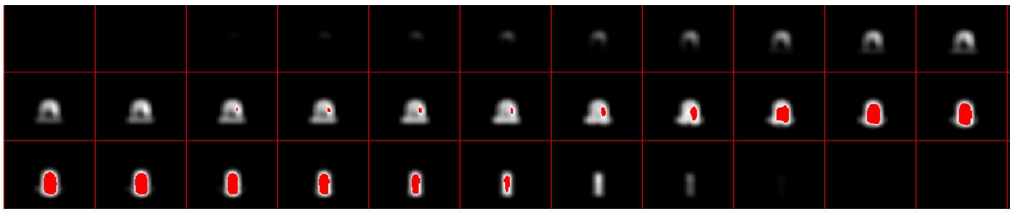
- Simple grid based density smoothing outlined here
  - tries to achieve volume preserving quality of inter-hair collisions.
  - based on Bertails05, can easily add Petrovic, and can extend to McAdams09
  - Hair strands and obstacles (like head/body) are voxelized into a low resolution grid
  - Hair vertices are pushed out of high density areas

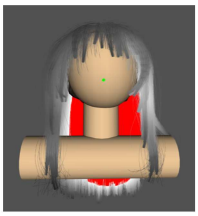
The approach that we are going to describe here is based on the approach of Bertails et al. This method only handles one of the aspects of inter-hair collision – its ability to give volume to the hair style. We choose this method because of its low simulation cost, and simplicity. Note however that this method can also be extended to the more complete, but more costly method of McAdams09.

We start by voxelizing the density of the head and hair strands to a low resolution grid. At the end of this step areas with high density represent areas where there are either lots of strands or there is an obstacle. In either case these are areas that we would want to push hair out of, because high density implies lots of hair strands in close proximity, and these are the regions that we want to expand out.

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

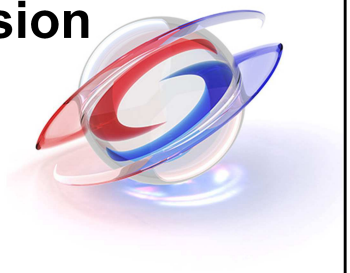


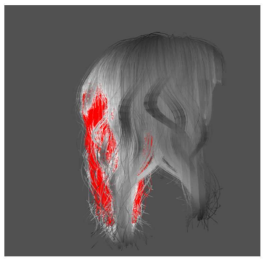
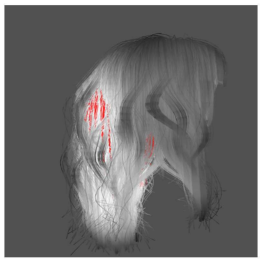


Once we have the voxelized density how do we use it to apply forces to the hair particles? The force on a hair particle is proportional to the density of the area it is in, and the force is in the direction of the negative gradient of the density. In order to get smooth and well defined forces everywhere we blur the voxelized density prior to using it to apply forces.

In the image at the bottom you can see the image of the blurred voxelized density of the hair style, which is to the right. The areas in red denote a minimum threshold of density – any region with less density does not experience intercollision forces.

Results of applying Inter-Hair Collision Forces



	Before	After
Visualizing density		
Final Rendering		

Here are some images showing the results of using this method. The images on the right show the initial hair, along with density of different regions. The images on the right show how the hair looks like after we apply the forces – we can see that regions of high density are reduced and that the hair has a fuller look.

References

- Bertails 06. *Super-Helices for Predicting the Dynamics of Natural Hair.* Florence Bertails, Basile Audoly, Marie-Paule Cani, Bernard Querleux, Frédéric Leroy, & Jean-Luc Lévêque. ACM SIGGRAPH 2006.
- Chang01. A Practical Model for Hair Mutual Interactions. Johnny T. Chang, Jingyi Jin, and Yizhou Yu. Pacific Graphics, 2001
- Hadap 06. *Oriented Strands - dynamics of stiff multi-body system.* Sunil Hadap.ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2006
- Hadap 01. *Modeling Dynamic Hair as a Continuum.* Sunil Hadap, Nadia Magnenat-Thalmann. Eurographics 2001
- McAdams 09.*Detail preserving continuum simulation of straight hair.* Aleka McAdams, Andrew Selle, Kelly Ward, Eftychios Sifakis, and Joseph Teran. SIGGRAPH 2009.
- Petrovic. *Volumetric Methods for Simulation and Rendering of Hair.* Lena Petrovic, Mark Henne, John Anderson
- Rosenblum 91. *Simulating the structure and dynamics of human hair: modeling, rendering and animation.* Rosenblum, R. E., Carlson, W. E., and Tripp III, E. J. Vis. and Computer Animation 1991.
- Selle 08. *A mass spring model for hair simulation.* Andrew Selle , Michael Lentine , Ronald Fedkiw. SIGGRAPH 2008.

SIGGRAPH 2010

WONDER!

"***The People Behind The Pixels***"

Community … Clarity … Content

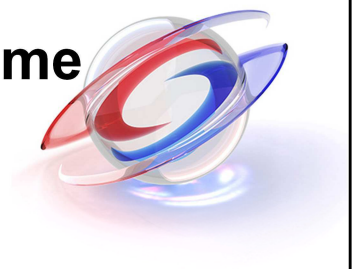
Conclusion

Advanced Techniques in Real-time Hair Rendering and Simulation
SIGGRAPH 2010 Course

Sarah Tariq
NVIDIA

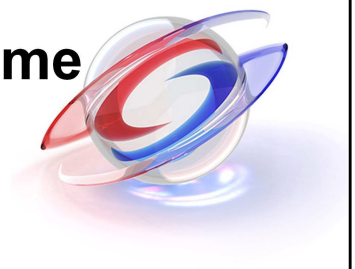
Cem Yuksel
Cyber Radiance and Texas A&M University

Advanced Techniques in Real-time Hair Rendering and Simulation



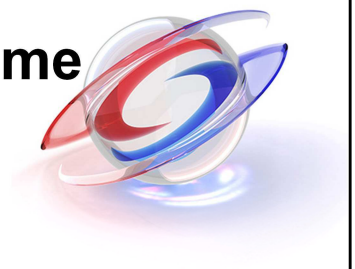
- Data Management and Rendering
- Transparency and Antialiasing
- Hair Shading
- Hair Shadows
- Multiple Scattering
- Hair Dynamics for Real-time Applications

Advanced Techniques in Real-time Hair Rendering and Simulation



- Acknowledgements
  - SIGGRAPH 2010
  - Cyber Radiance LLC
  - NVIDIA
  - NSF IIS Award #0917286

Advanced Techniques in Real-time Hair Rendering and Simulation



- Final Course Notes
  - Cem Yuksel's web-site

    http://www.cemyuksel.com/?x=RealtimeHairCourseNotesSiggraph2010
  - Sarah Tariq's web-site

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