# Hair

Sarah Tariq
stariq@nvidia.com

March 2010

# Document Change History

| Version | Date | Responsible | Reason for Change |
|---|---|---|---|
| | 25/03/2010 | Sarah Tariq | Initial release |
| | | | |
| | | | |
| | | | |

## Contents

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

# Abstract

Simulating and rendering realistic hair with tens of thousands of strands is something that until recently was not feasible for games. However, with the increasing programmability and power of Graphics Hardware not only is it possible to simulate and render realistic hair entirely on the Graphics Processing Unit (GPU), but it is also possible to do it at interactive frame rates. Furthermore, new hardware features like the Tessellation and Compute Shaders make the creation and simulation of hair easier and faster. This sample shows how to simulate and render hair on the GPU using both D3D11's new shader stages (tessellation and compute shader) and using older APIs like D3D10.

# How Does It Work?

## Overview

In this sample we simulate a small number of hair strands (166 strands) and then when rendering the final hair we interpolate between these strands to create many more strands (over 18,000). Since the amount of final rendered hair is a lot larger than the amount of simulated strands, this method saves the considerable cost of having to simulate every final hair that is rendered while producing very acceptable results. The creation of these additional strands can be done using the tessellation engine, or can be done using other approaches as outlined in [Tariq 08] (code for both paths is provided in the sample).

## Rendering Hair Using Tessellation

**Generating Data on the GPU**

Given the large amount of data that we are trying to render, it is more efficient to generate this data directly on the GPU than to send it at each frame from the CPU to the GPU. DirectX 11 introduce functionality to create large amounts of data directly on the hardware; the tessellation engine. In this section we are going to be talking about how to use the tessellation engine to easily and efficiently create hair strands for rendering.

Note that functionality for creating data on the GPU existed even in D3D10, via the Geometry Shader. However, the Geometry Shader is meant for only small amounts of data expansion, for example for expanding a line into a quad. It is not meant for, and is certainly not efficient at, creating the amounts of geometry that we would like to create for all the hair strands.

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

**Hardware Tessellation Pipeline**

The tessellation engine has three stages, two of them are programmable (the Hull and Domain Shader) and one of them is fixed function (the Tessellator).

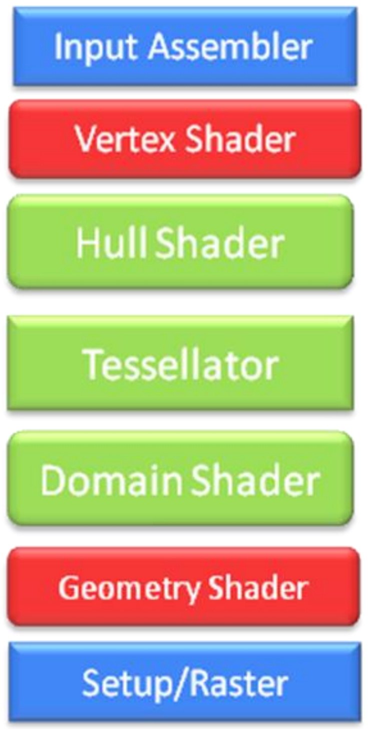


Figure 1 D3D11 Tessellation Pipeline

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

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

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

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

**Interpolation Methods**

In the remainder of this section we are going to discuss how the tessellation engine can be used to efficiently create new hair strands. We are going to be using two methods of creating strands as example use cases (although the tessellation engine is programmable enough to support other methods).

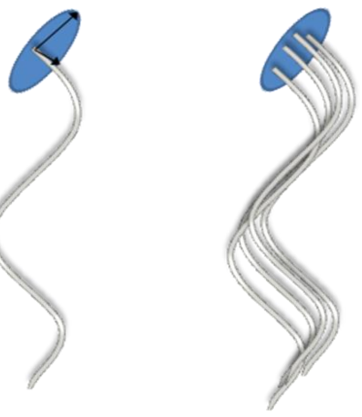


Figure 2
Single Strand
Based
Interpolation

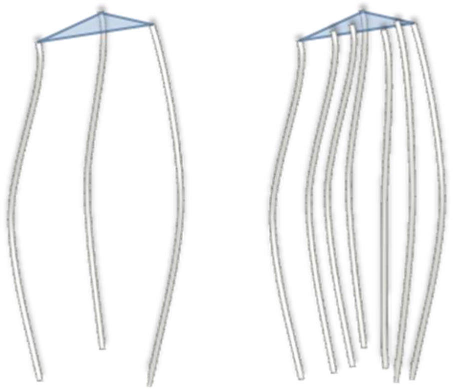
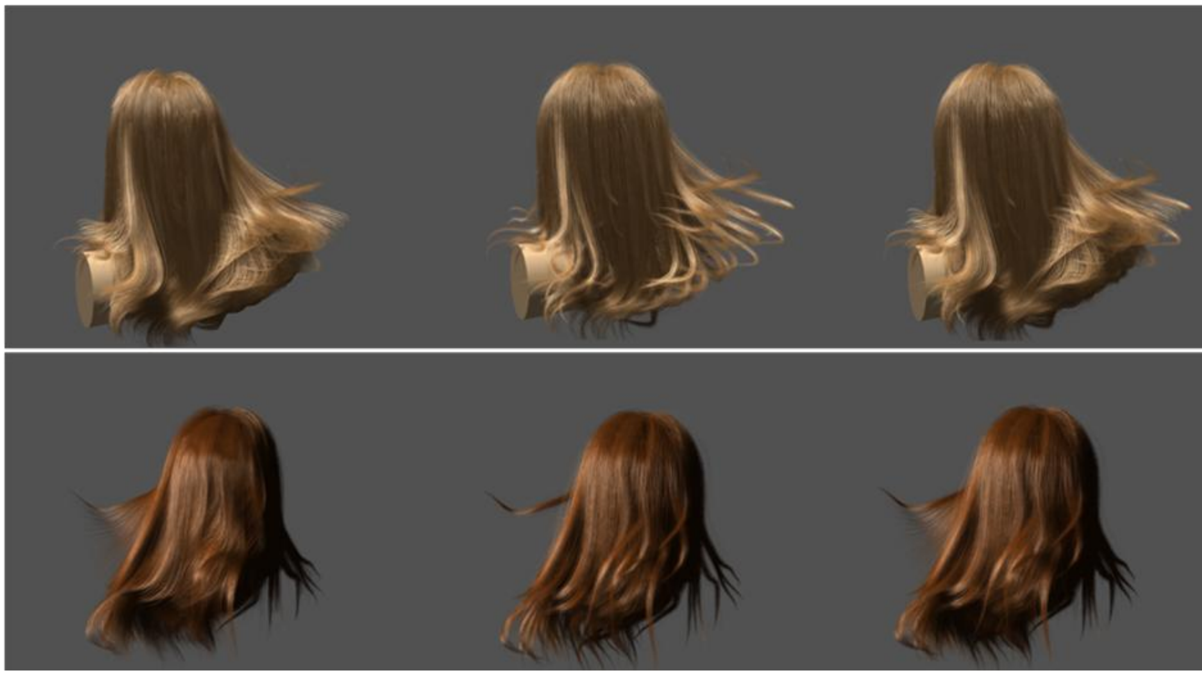


Figure 3
Multi Strand
Based
Interpolation

The first is Single Strand Based Interpolation. Using this 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).

The second type of interpolation is Multi Strand interpolation. Each interpolated strand is created by linearly interpolating the attributes of three 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. Each interpolated Strand is assigned a random float2 to use as interpolating weights.

shows what the two interpolation modes are able to create for the same input data. Each has a different look, but they are complementary. The Multi Strand interpolation method creates a full and uniform look which can be a bit boring sometimes. It also has problems with hair penetrating into collision obstacles (we will deal with this issue and its solution in another section). The Single Strand based interpolation method does not have significant collision issues, although sometimes when the guide strands are relatively far from each other the resultant interpolated hair can look like ropes.



Multi Strand Interpolation Single Strand Interpolation Combination

Figure 4 Different Interpolation Modes For Hair

**Single Strand Based Interpolation**

Figure 5 shows an overview of a pass to create and render data using the tessellation engine and the single strand based interpolation method.

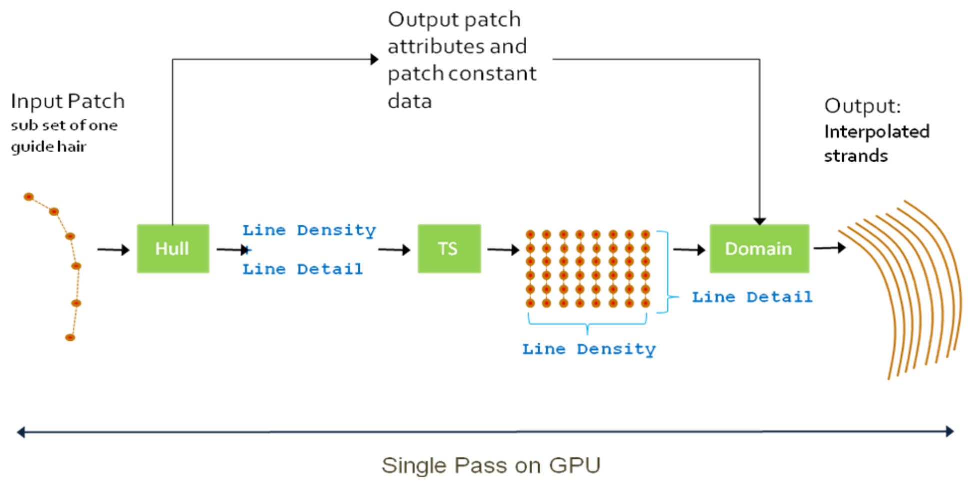


Figure 5 Interpolation using the Tessellation Engine

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

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

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

Since we are going to be binding our data as texture buffers which can be sampled in the Hull or Domain Shader we don't need to bind any data to the input assembler as vertex buffers (or for that matter index buffers or input layout). In addition, we set the primitive topology to be a patch with a single control point.

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

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

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

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

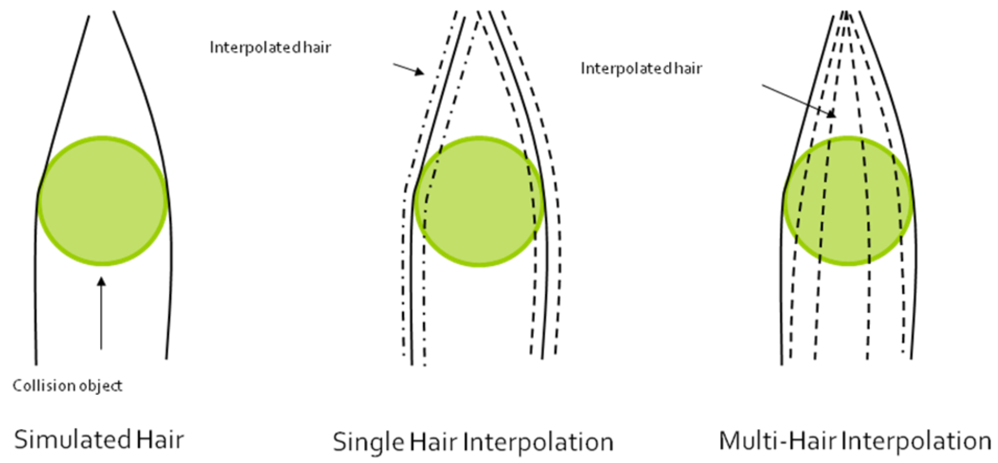
The Hull Shader is split into two parts, the main shader which operates once on each input control point, and the patch constant function which is invoked once per patch. In our implementation we are loading control point data for a patch from a buffer, so the input to the hull shader is a dummy patch consisting of a single control point. Since we have only one input control point we are using the patch constant function for all of the computation and our main shader is null. The code for the Patch Constant Function, which calculates the tessellation factors and other data for each patch, can be found in the function InterpolateConstHSSingleStrand in Hair.fx.

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

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

## Interpolated Hair Collisions

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 (single strand based interpolation), or they can be created by combining multiple guide hair (for example 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 below Figure 6.



### Figure 6 Interpolated hair collisions arising from multi hair Interpolation

As we see in Figure 6(a) the guide hairs avoid the collision obstacle since they are explicitly simulated. The interpolated hairs on the other hand are only produced using the positions of the guide hair and have no physical simulation. In Figure 6(b) we see the results of interpolation using only a single guide hair to create a given interpolated hair. The created interpolated hairs penetrate the collision object slightly, but largely follow their guide hair. In Figure 6(c) however we see much worse penetration of interpolated hair even though the guide hairs themselves 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 about the positions of the interpolated vertices – they might go straight through the collision object, for example the head. In this section we describe a method for efficiently detecting and avoiding cases where multi-guide hair interpolation leads to hair penetration into objects. Figure 7 gives an example of such interpolation artifacts which are obviously undesirable. Figure 8 shows the results after applying our technique to detect and avoid these problems.

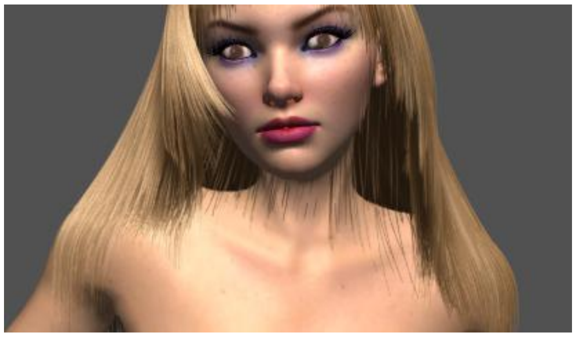


Figure 7 Interpolated hair can intersect collision volumes

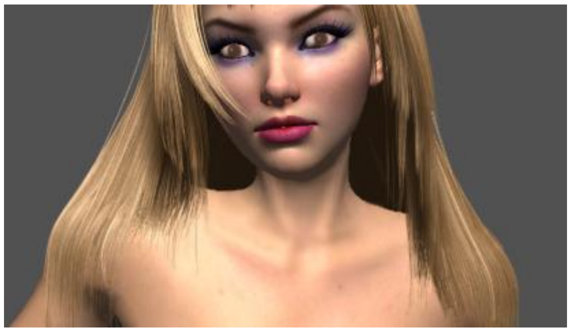


Figure 8 fixing the collisions without extra simulation

**Technique**

We first detect strands where multi-strand interpolation leads to penetration and then we switch the interpolation mode of such strands to 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 Figure 9. If, as in figure Figure 9(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 9(c).

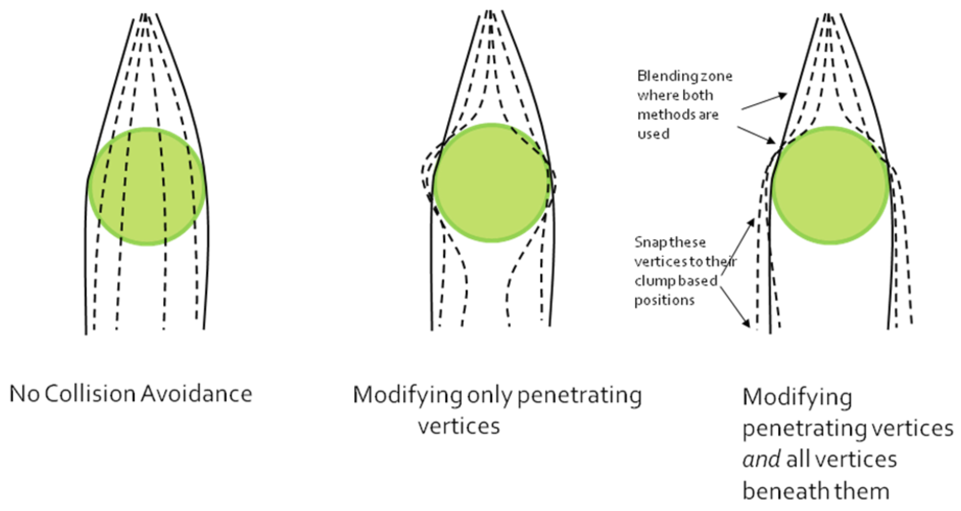

Blending zone where both methods are used
Snap these vertices to their clump based positions
No Collision Avoidance
Modifying only penetrating vertices
Modifying penetrating vertices *and* all vertices beneath them

Figure 9 Correctly modifying penetrating hair

In order to identify hair vertices that are below other object-penetrating vertices we need to 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. To test if a vertex is inside a collision obstacle we perform a single lookup into a 3D texture of obstacles, which we create once per frame (for a sufficiently high number of obstacles such as in this demo we have found that pre-voxelizing the obstacles into a texture is faster than evaluating for each vertex whether it is undergoing a collision).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 correctly switch the interpolation mode of interpolated hair when we are creating them. When we calculate the interpolated hair position we look up into 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.

## Simulating Hair

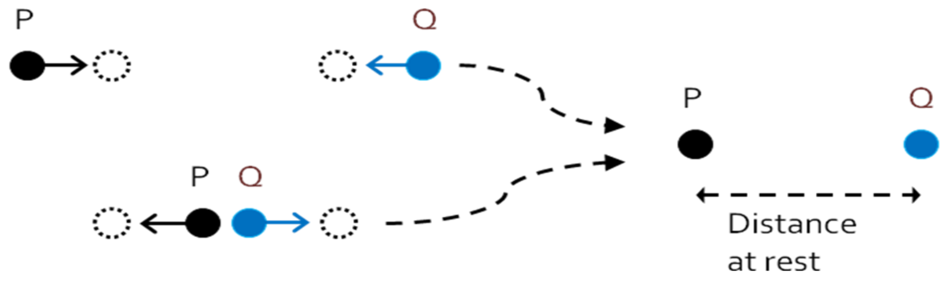
In order to keep our simulation simple and also parallel we simulate the hair as a Particle Constraint System:

- Hair vertices are simulated as particles
- Links between hair vertices are treated as distance constraints – these constraints maintain the length of the hair, preventing it from stretching or compressing
- Angular forces and angular constraints help maintain the shape of the hair
- Collision constraints keep the hair outside of collision obstacles

One time step of the simulation looks something like this:

- Simulate wind force (using fluid simulation, see [Tariq07] for more details)
- Add external forces and integrate using verlet integration
- Repeat for num_iterations
    - Apply distance constraints
    - Apply angular constraints
    - Apply collision constraints

- Example: Distance Constraints.
  A distance constraint DC(P, Q) between two particles P and Q is enforced by moving them away or towards each other so that they are at exactly a pre-specified distance apart:



### Figure 10 A Distance Constraint between two particles

Since a given particle can be affected by two distance constraints (one to the hair particle above it, and the other to the one below) we cannot apply all distance constraints in parallel. Instead we split the hair particles into two groups and apply the distance constraints in two steps:

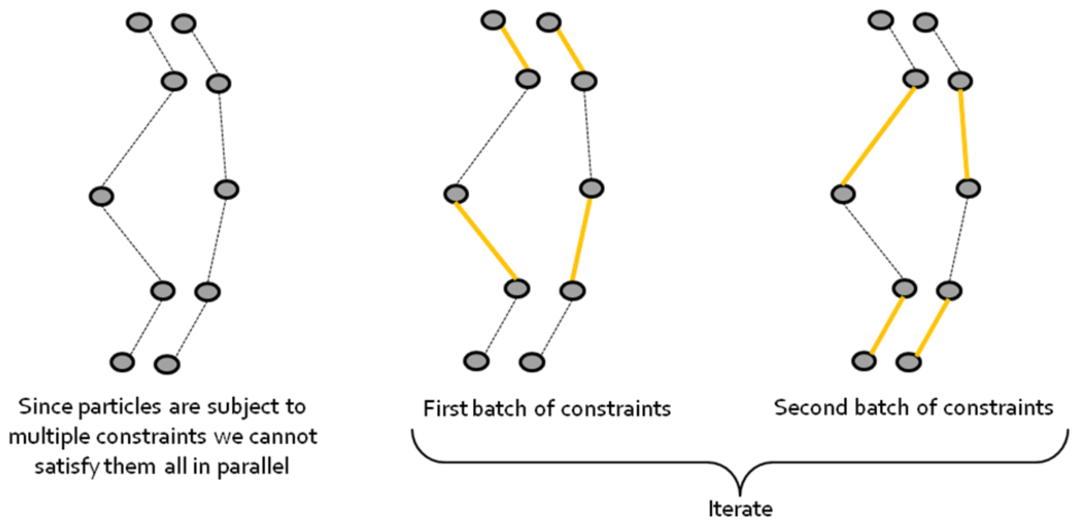

Since particles are subject to multiple constraints we cannot satisfy them all in parallel
First batch of constraints
Second batch of constraints
Iterate

Figure 11 Applying Distance Constraints in Parallel

The code sample provides two methods of simulating hair; using D3D11's Compute Shader and using Streamout.

Using the compute shader has several advantages – the code is easier to write, and can be faster. We can satisfy all constraints in a single function call by using Shared Memory (all vertices of a single strand are in the same thread group). The code for the compute shader is given in HairSimulateCS.hlsl. All the simulation for the hair is encapsulated in a single function, `UpdateParticlesSimulate`.

In order to simulate hair without using the Compute Shader we use traditional GPGPU ping-ponging techniques – every time we update particle attributes that have to be read by other particles we write out the attributes to a buffer. For example, in order to satisfy distance constraints we write the positions of the particles in a vertex buffer, and satisfy the distance constraint in the Geometry Shader, and then stream out this data to another vertex buffer. In the next iteration we swap these buffers and repeat the process:

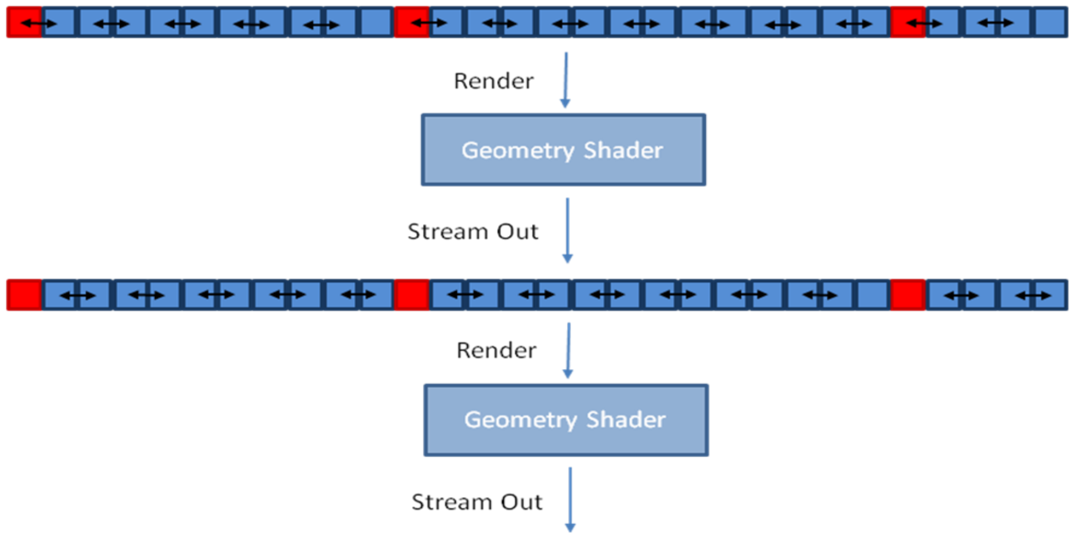


Figure 12 Satisfying Distance Constraints using Stream Out

## LOD

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

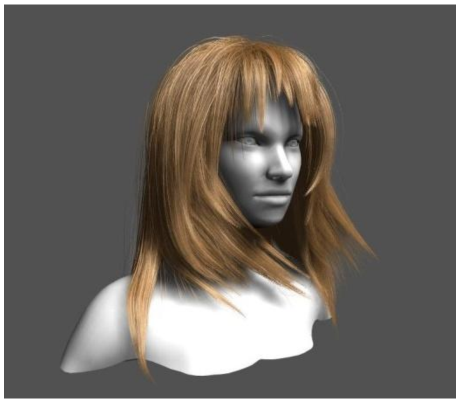


Figure 13 Hair at full Level of Detail

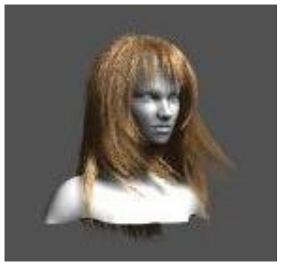


Figure 14 Hair at lower Level of Detail

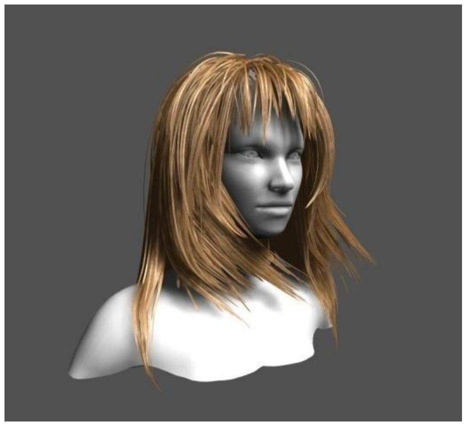


Figure 15 Hair at lower LOD, zoomed in

In the figures above we are showing how hair LOD can be scaled as the character moves further away, by reducing the amount of hair that we create. Figure 13 and Figure 14 show the same hair style under two levels of detail. At the top we have the hair rendered at full LOD, and at the bottom we have scaled down the rendering and increased performance by 2x. In this case we are creating and rendering less hair strands, although we are making the strands a bit wider so that there is no visible reduction in the density of hair. Figure 15 shows what that lower level of detail would look like if you zoomed in.

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

In addition to rendering LOD we can also change the simulation cost of the hair. In this demo we have implemented a very simple LOD system for simulation – at full LOD we simulate the hair at each frame, and as LOD decreases we simulate only once every n frames (where n is inversely proportional to the LOD).

## Additional Details

**Curly Hair**

In order to create curly or wavy hair we pre-create and encode additional curl offsets into constant buffers or textures. These offsets are then added to the clump offsets when creating the final interpolated hair. In this demo the curl offsets are being created procedurally, but we can also have artists create example curls and the offsets can be derived from those. This process of creating curly hair lets us retain the relatively simple hair simulation scheme that we have which will not work well if the simulated hair is actually curly.

**Random Variations**

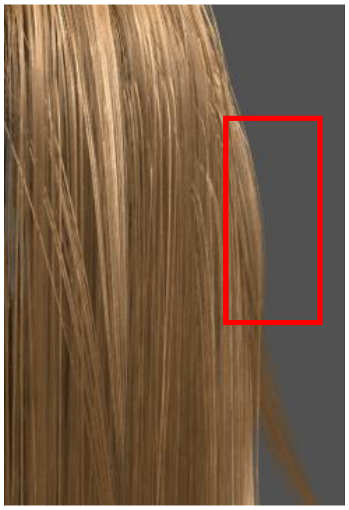


Figure 16
No variations

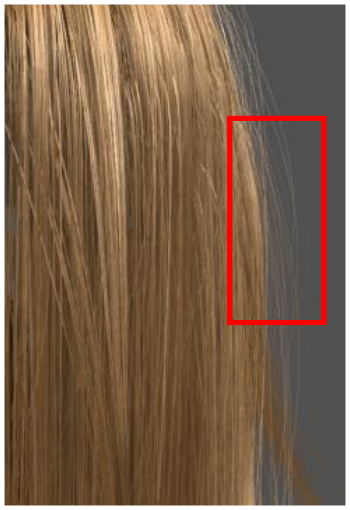


Figure 17
with random variations

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 **Error! Reference source not found.**. Adding randomness to strands gives a more natural look, **Error! Reference source not found.**. Since our model for rendering hair is based on interpolating many children hair from a small set of guide hair, we need to introduce this randomness at the interpolated hair level. The method we use is similar to that presented by [Choe and Ko 2005]. We pre-compute a small set of smooth random deviations and apply these to the interpolation coordinates of the interpolated hair.

We create two types of deviations for the hair. The first, applied to a large number of the hair strands, is small deviations near the tips. In our images we have applied this deviation to 30% of the hair strands. The second, applied to only a small percent of the hair strands (for example 10%), is deviation all along the strands. This second type of deviation is what you see highlighted in the red box **Error! Reference source not found.**.

## Performance

Default performance of the demo on GTX 480.

| | Using Tessellation Engine and Compute Shader | Not using Tessellation engine and using Dx10 simulation |
|---|---|---|
| Hair rendering and simulation with wind | 57 fps | 44 fps |
| Just hair rendering, no hair simulation | 77 fps | 60 fps |
| Just hair simulation, no hair rendering | 720 fps | 624 fps |
| Hair rendering and simulation with wind, far away (with LOD) | 930 fps | 77 fps* *note that this frame rate is exceptionally low because no rendering LOD has been implemented in this path |

## Integration

In order to integrate this demo the most important files to look at are Hair.cpp (functions RenderInterpolatedHair, StepHairSimulation, and OnD3D11FrameRender), HairSimulateCS.hlsl (shaders for hair simulation using the compute shader), and Hair.fx (techniques for hair rendering and simulation). Some useful techniques in Hair.fx are InterpolateAndRenderM_HardwareTess (render hair using multistrand interpolation), InterpolateAndRenderS_HardwareTess (render hair using single strand based interpolation) and

InterpolateAndRenderCollisions_HardwareTess (pass to render multi strand hair to a texture in order to find colliding vertices).

In addition, you have to create and import a hair cut into the application. For this demo we have used a default haircut available in Maya, exported it as points and parsed and massaged in LoadHairFile.cpp.

Please note that a number of the parameters used for hair rendering and simulation are specific to this hair cut and the dimensions of our scene, and will have to be modified to fit different hair models and scenes.

# Running the Sample

In the default start up mode the sample exhibits hair being simulated using the compute shader and rendered using hardware tessellation. To reset all the options to the default use the "Reset" button near the top.

## Interaction

Camera: Use left mouse drag to move the camera, and mouse wheel to zoom the camera.

Light: To move the light use SHIFT + left mouse drag (the light is indicated by a white arrow)

Model: To move the model use right mouse drag. To rotate the model drag the middle mouse button / mouse wheel.

Wind: Use SHIFT + right mouse drag to move the direction of the wind (shown by the green arrow)

## Animation

The sample provides a way to move the head in a pre-animated motion (note that hair is still simulated in real time), which can be turned on by the "Play Animation" check box, and looped with the "Loop Animation" check box.

## Hair Rendering

There are a number of ways to change the look of the hair, including changing its color using the provided combo box, and making it short or curly using checkboxes.

To choose the type of interpolation to be used, Single Strand or Multi Strand, use the check boxes "Render M Strands" and "Render S Strands".

The checkbox "Hardware Tessellation" allows you to toggle between using D3D11 Tessellation for rendering the hair, or rendering them using D3D10 style "instanced tessellation" (se Tariq08).

## LOD

By default the sample uses dynamic LOD for the hair rendering, so that as you zoom the camera in or out different amount of hair are rendered. The dynamic LOD level is indicated by the slider under the "Dynamic LOD" checkbox – note that as you zoom the camera the  LOD and Hair Width sliders changes their values, indicating current LOD. If you wish to use manual LOD you can uncheck the "Dynamic LOD" checkbox and manually adjust the number of hairs and the width of each hair using the two sliders.

## Hair Simulation

By default the hair is simulated using the Compute Shader. In order to disable this you can uncheck the "Compute Shader" checkbox, in which case the sample uses multipass simulation on the vertex/geometry shader.

"Simulation LOD" allows the sample to drop the rate of simulation to once every n frames based on the LOD. In order to disable Simulation LOD, uncheck the checkbox. When simulation LOD is disabled you have the option of pausing the simulation using the "Simulate" checkbox.

The "Show Collision" checkbox allows the user to visualize the collision implicits that are being used in the hair simulation.

Wind is being applied to the simulation, which can be turned off using the "Add Wind Force" checkbox, or by pressing 'k'. The "Wind Strength" slider directly below this checkbox allows the user to adjust the amount of wind force.

# References

Bertails, F., Menier, C., and Cani, M.-P. 2005. A Practical Self-Shadowing Algorithm for Interactive Hair Animation. In *proceedings of Graphics Interface 2005*, 71-78.

Bertails, F., Audoly, B., Cani, M.-P., Querleux, B., Leroy, F., and Leveque, J.L. 2006. Super-Helices for Predicting the Dynamics of Natural Hair. In *Proceedings of ACM Transactions on Graphics (Proceedings of the SIGGRAPH conference).*

Plante, E., Cani, M.-P., and Poulin, P. 2001. A layered wisp model for simulating interactions inside long hair. *In EG CAS '01*, Springer, Computer Science, 139–148.

Choe, B., and Ko, H.-S. 2005. A Statistical Wisp Model and Pseudophysical Approaches for Interactive Hairstyle Generation. *In proceedings of IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 2, PP. 160–170, march 2005

Tariq, S and Bavoil, L 2008. Real Time Hair Simulation and Rendering on the GPU. Technical talk, SIGGRAPH 2008

Tariq, S and Llamas, I. Smoke. NVIDIA Direct3D10 Sample. http://developer.download.nvidia.com/SDK/10/direct3d/samples.html