The TressFX 4.0 and Rat Boy Demo

The Rat Boy demo demonstrates the latest TressFX technology. This document describes new hair and fur technology.

Contents

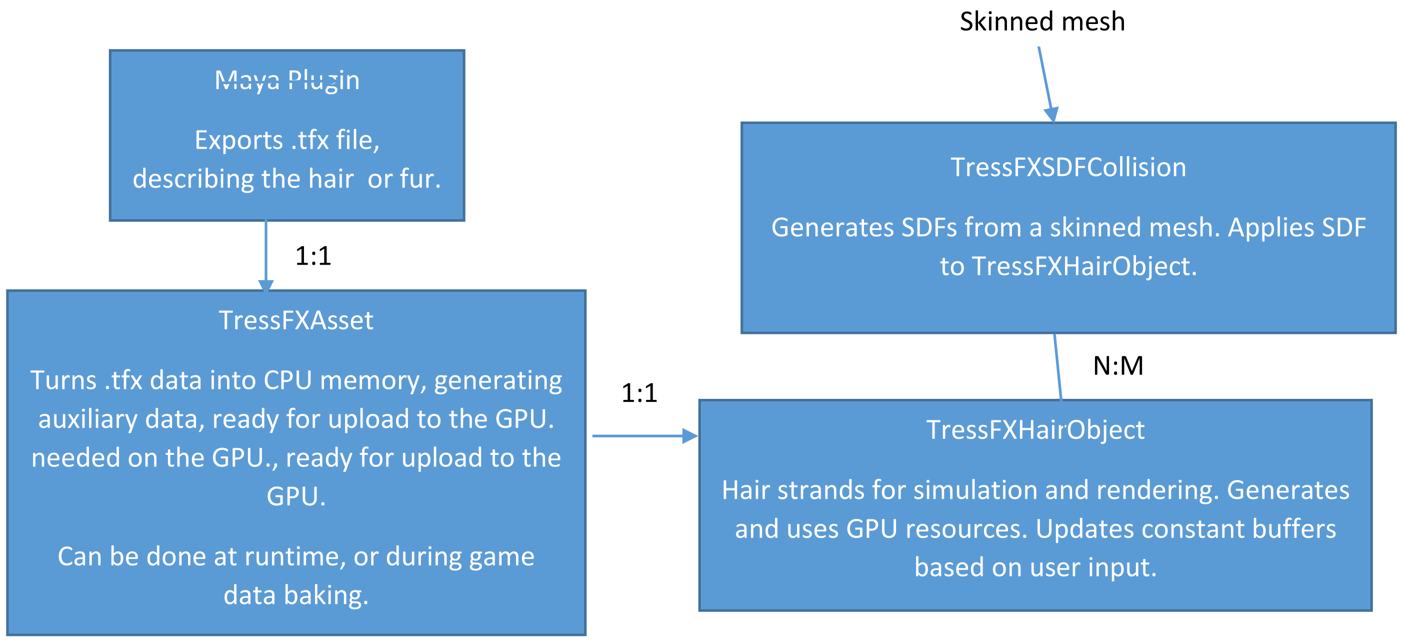
Introduction

This release introduces new simulation technology, as well as a different philosophy on how to organize the library.

In terms of simulation, we introduce three new changes. First, we now skin the hair roots directly, rather than relying on deriving a transform from streamed-out triangles that are required to match, or be mapped to, original Maya triangles. Second, we now support signed distance fields for collision, enabling more general collision surfaces. Third, we include “velocity shock propagation” to more robustly handle quick movement.

Our new philosophy in this release is to not provide an “open source library”, but instead source code for you to use as you see fit. Second, we have separated TressFX from low level implementation, both in C++ and shader code.

Code Overview


Skinned mesh
Maya Plugin
Exports .tfx file, describing the hair or fur.
1:1
TressFXAsset
Turns .tfx data into CPU memory, generating auxiliary data, ready for upload to the GPU. needed on the GPU., ready for upload to the GPU.
Can be done at runtime, or during game data baking.
1:1
TressFXSDFCollision
Generates SDFs from a skinned mesh. Applies SDF to TressFXHairObject.
N:M
TressFXHairObject
Hair strands for simulation and rendering. Generates and uses GPU resources. Updates constant buffers based on user input.

Calls Into TressFX

TressFXAsset takes a file stream containing output from the provided Maya plugin, and extracts the data into CPU memory in a format directly usable on the GPU. It also takes an interface to your skeletal system in order to map bone indices to your engine's ordering. It can be used at runtime, as in the included demo, or during asset processing.

HairObject represents a single hair "object" for simulation and rendering. It's created from a TressFXAsset.

TressFXSDFCollision takes a skinned mesh as input, generates a signed distance field (SDF) and creates a response in the TressFX hair/fur.

The mapping between HairObject and TressFXSDFCollision can be fairly arbitrary in terms of number. The demo includes two hair objects – one for the hair, and one for the fur, and three SDF objects – one for each hand, and one for the body overall.

Calls Into the Engine and Graphics APIs

We also provide a low-level interface. This is new with TressFX 4.0. Everything we've described above makes calls to low-level APIs or engines through two header files. We used the "EI" prefix to indicate "engine interface" pieces we expect you'll want to go through your engine, or would be implemented by the various graphics APIs (DirectX, Vulkan, etc). This differs from previous releases, which made raw calls to DirectX 11 and required substantial rewrites to accommodate others.

For the most part, we've exposed this interface as something you can dictate at compile time, rather than the runtime callbacks or abstract interfaces that are more common for closed-source systems.

TressFXEngineInterface.h defines GPU-independent pieces, such as how to allocate and deallocate system memory and how to handle error messages.

TressFXGPUInterface.h defines GPU-related pieces. Instead of making direct calls to a low-level graphics API, we define what's needed in one place, making the code more portable between low-level graphics APIs, whether DirectX or engine abstractions.

We've also carried this principal into the shader code. Although simulation shader code is mostly self-contained, we've provided functions to work with your lighting and shadowing system, rather than providing a complete, yet highly constrainted pixel shader, as in past versions.

File Locations

The **tressfx** directory contains the library code we don't expect you'll want to change that often. It's roughly equivalent to what we've released in previous versions, but broken into more independent components, and without dictating a specific graphics API.

The **Sample/src** directory contains code illustrating how to use TressFX and examples of how to integrate with an engine.

Hair/Fur Simulation

New features in Simulation

Fur requires that the hair stays attached to a skinned mesh. In TressFX 3.0, we introduced an animated bear model and we used triangle-based skinning system to animate the root vertices. In 3.0, we relied on post-skinned triangles of the mesh as inputs to the simulation through stream-out. This was a very general approach in the sense that it didn't matter how the triangle was generated, accommodating skinning or blend shapes, for example. However, it imposed constraints on the engine we felt were too cumbersome.

In TressFX 4.0, we are introducing a new bone-based skinning system for hair/fur. This allows developers to integrate TressFX into their engine more seamlessly. Also it can separate the hair or fur from its skin mesh so that multiple hair/fur objects can be used for the same character more easily. Since the bone skinning transforms (matrices or dual quaternions) are an input to the simulation, we can make sure that the hair/fur produces the same animated results as the character.

In terms of collision, we are introducing SDF (Signed Distance Field). For simple hair simulation, it might be suitable with a few spheres or capsules to represents head and neck. However, long fur on a skinned mesh is tough to model this way, and doesn't scale well. A signed distance field more generally represents an arbitrary, deforming mesh with a single data structure.

Lastly, we are introduce VSP (Velocity Shock Propagation). In real-time games, it is hard to predict the character movement and often, the fast animation transition causes extremely high and sharp velocity change. In the previous TressFX, this kind of excessive acceleration was addressed by increasing stiffness, damping and number of iterations for constraints. Not to mention the difficulty of tuning physics parameters, increasing number of iterations results computational cost. With VSP, the velocity of animated root vertex is propagated throughout the rest of vertices in the strand. Also VSP checks pseudo acceleration and increases VSP value when the pseudo acceleration is higher than certain threshold.

Physics Parameters

Here are brief descriptions of the physics parameters. The best way to get a feel for what they do is to simply play with them. Some parameters are particular to a hair object. In Rat Boy demo, fur is divided into two objects. One is Mohawk and the other is Short.

VSP (Velocity Shock Propagation) makes the root vertex velocity propagate through the rest of vertices in the hair strand. The amount of propagation is controlled by a value ranged from 0 to 1. If it is 0, then there will be no velocity propagation. If it is 1, then the full velocity of the root vertex will be passed to the rest of vertices. In this case, hair would act as if it is rigid and the rest of simulation would not affect the result. So in case you want to animate the hair or fur but do not want to simulate it, then VPS is a perfect solution.

VSP acceleration threshold makes VSP value increase when the pseudo-acceleration of root vertex is greater than it. This is particularly effective when the character makes a sudden movement because VSP value becomes to the highest amount when the pseudo-acceleration reaches the user input value. The acceleration is pseudo because it is not divided by time.

Damping smooths out motion of the hair. It also slows down the hair movement, making it more like hair under water, for example.

Local and Global Stiffness makes each strand stiffer – meaning that if the hair started out straight, it will try to keep that shape. If it was curved, it will try to keep the curve. The difference between the two is that **global stiffness** tries to move the hair back to where it was originally in a global way – so if there is something pushing a straight hair in the middle, the tip will try to go back around the ball to where it started, giving it a curved shape. The **local stiffness** only looks at the shape locally, so if that same straight hair is pushed in the middle, the hair will get out of the way of the ball, but try to stay straight, instead of curving around it.

Global Shape Range controls how much of each hair strand is affected by the global shape stiffness. If the value is 0, then the stiffness is off (as if it also has value of 0). If it is 1, then it affects the whole hair strand, and the tip of the hair will try to get back to where it was as described above. If the value is 0.5, then only the first half of the hair strand (from the root) tries to get back to its original position, so the tip will not try to get back to its original position in the previous example.

Length Constraint Iterations allocates more simulation time to keeping the hair the right length. Try to keep this number as low as possible.

Local Shape Constraint Iterations allocates more simulation time to keeping the hair shape. It can make the hair seem stiffer, but also greatly increases simulation time.

Wind Magnitude allows you to see the effect of wind on the hair.

Bone-based Skinning

In hair on the head, we apply the single head transform to the entire hair strands. In fur on the skin mesh, we need to animate the root vertices of strands. To be accurate, we animate the first two vertices to maintain the position as well as direction of the strand. In TressFXAsset::LoadAsset function, you can find the code to assign zero inverse masses to the first two vertices. The forth component (w) of vertex position is the inverse mass.

The bone-based skinning code is in IntegrationAndGlobalShapeConstraints compute shader kernel. Since our demo uses 4x4 matrices for bone transforms, it takes array of 4x4 bone matrices in world space. In case your engine uses dual quaternion instead, you can use g_BoneSkinningDQ and change the code properly.

The maximum influential bones per hair strand are set to four. This value is hard-coded and should match the .tfx file and simulation compute shader (TressFXSimulation.hlsl). Although it's possible to change this value through TRESSFX_MAX_INFLUENTIAL_BONE_COUNT in TressFX_Exporter.py and TressFXAsset.h and IntegrationAndGlobalShapeConstraints in iTressFXSimulation.shl, we have not tested other values.

Collision Mesh

Collision mesh is a triangle mesh and an input to the SDF. It does not need to be the same as rendering mesh but is recommended to be closed without open holes. It is also recommended to be non-overlapping. However practically, it may be hard to avoid all overlapping or small holes. In our Rat Boy demo, the collision mesh is the same as rendering one and has some overlapping and holes in the face.

TressFXSDFInputMeshInterface is an interface class showing how the collision mesh can be linked to the SDF object.

Signed Distance Field (SDF)

SDF is a grid-based representation for polygonal mesh. For more details, you can get info from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch34.html.

In TressFX, we use the dense grid and the memory gets allocated up front. Because of this pre-allocation of memory, the grid and cell sizes should be chosen carefully to be big enough to enclose all possible animations but should not be too excessive. This might cause potential memory waste. So it would be wise to break down mesh parts and use different grid and cell sizes. In Rat Boy demo, the character has three body parts (main body and two hands).

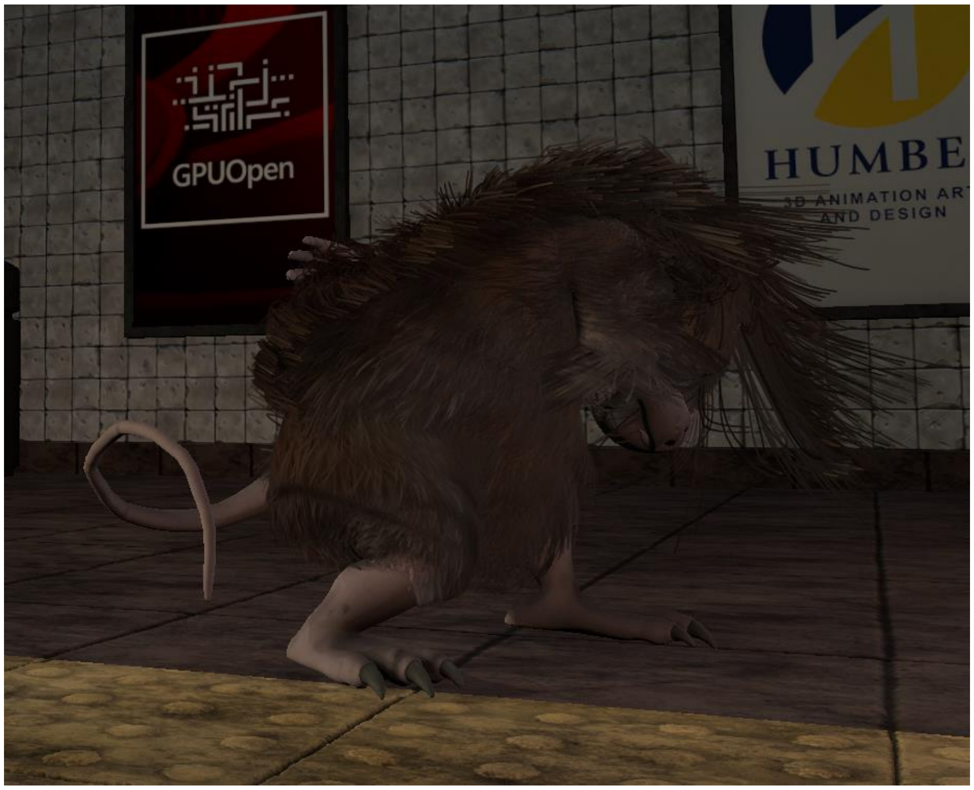
The signed distance field is best defined using closed meshes. Although a water-tight mesh is not strictly required, it's best to minimize holes in the mesh that are larger than a grid cell unless it's a portion that won't interact with the hair.

Unlike TressFX 3.x skinning, we impose no requirements on mesh consistency with Maya.

Velocity Shock Propagation (VSP)

VSP is a new feature in TressFX 4.0. The main purpose of this is to handle fast moving animations. When the character changes its speed or direction, it generates a high acceleration and consequently a big external force. Since TressFX hair simulation uses iteration-based constraint solver, when a high acceleration gets applied, hair can easily lose its physically-correct shapes and shows unpleasant elongation.

Following images compares the simulation with the same physics parameters with and without VSP. The first image is without it and the second one is with it. In the scene, the Rat Boy is moving from left to right quickly. With VSP, the fur maintains it shape well and reacts to the character movement elegantly.



GPUOpen
MBE
MATION ART
ESIGN

VSP has a control value ranged from 0 to 1. If it is zero, the will be no VSP. If it is 1, then full velocity of root vertex can be propagated to the rest of vertices in the strand. VSP can handle both linear and rotational velocities.

In addition to propagating velocity, there is a VSP acceleration threshold value which increases VSP value to 1 when the pseudo-acceleration passes it. It is quite effective when the character makes a sudden movement.

Marching Cubes (MC)

Marching cubes can be used to visualize the SDF for debugging purposes. By checking 'Draw marching cubes' checkbox, it will appear in yellow as below.

Guide and Follow Hair System

This system was introduced in TressFX 2.0 and is being used the same in this demo. In terms of collision with SDF, both guide and follow hair get response from it. It is still possible to make only guide hair affected by SDF but the overhead is small enough to use SDF collision for all of them.

Rendering

The implementation of the rendering part of TressFX is based on three main features required to render good looking hair:

- Antialiasing
- Self-Shadowing
- Transparency

When used with a realistic shading model these features provide realism needed for natural looking hair. The sample uses a modification of the well-known Kajiya-Kay hair shading model (Kajiya, 1989). Two specular highlights are rendered (Sheuermann), similar to the Marschner model (Marschner, Jensen, Cammarano, Worley, & Hanrahan, 2003). The hair is rendered as tens to hundreds of thousands of individual strands stored as thin chains of polygons.

Since the width of a hair is on the order of pixel size, antialiasing is required to achieve good looking results. In TressFX this is done by computing the percentage of pixel coverage for each pixel that is partially covered by a hair.

The second main feature, Self-Shadowing, is necessary for giving the hair a realistic looking texture. Without it, the hair tends to look artificial and more like plastic on a puppet than as opposed to real hair. Shadowing is achieved through a variant of shadow mapping. Rather than a binary in shadow / out of shadow, the light is attenuated using the depth difference.

Transparency provides a softer look for the hair, similar to real hair. If transparency was not used the strands of hair would look too coarse, especially at the edges. In addition to that, real hair is actually translucent, so rendering with transparency is consistent with simulating the lighting properties of real hair. Unfortunately transparent hair is difficult to render because there are thousands of hair strands that need to be sorted. To help with this, TressFX technology uses order independent transparency (OIT). There are two variants. One uses a per-pixel linked list, and the other uses a multi-pass method we refer to as “ShortCut”.

Further detail for how each of these components work is provided here. Details on how to enable these systems are provided in the Integration Steps section of this document.

The main decision to make is whether to use per-pixel linked lists, or “Shortcut”. The Shortcut method is generally faster, with an easier memory bound, but at the expense of some quality. Tomb Raider used the per-pixel linked list method. Deux Ex used something more like Shortcut.

Integration Steps

You will want to start by getting strands to draw on the screen as a baseline. From there, you can choose what to add next based on your needs: simulation, fur (skinned TressFX), transparency, and signed distance field collisions (static or dynamic).

Baseline

This section covers the steps necessary to load a .tfx file and display it on the screen.

Baseline	Loading TressFX TressFX Object Layouts TressFX Vertex Shading	Draw hairs on the screen without transparency or lighting.
Simulation	Simulation	To see motion, you will need to
Fur		
PPLL Transparency	PPLL	
Shortcut Transparency	Shortcut	
Signed distance field collision		

Loading TressFX

The data on disk for hair/fur geometry is in a .tfx file. TressFXAsset is responsible for loading data from this file into CPU memory, and further processing needed to put the data into a form ready for runtime usage.

These are the basic functions:

TressFXAsset::LoadAsset(EI_Stream& fileStream)

Loads data from a .tfx filestream. The filestream uses only two functions: read and seek. Implementation can be provided by engine, or by the default implementation, which simply reads from a stdio FILE. We also provide an example that works with a memory buffer.

TressFXAsset::GenerateFollowHairs(…);

Optional stage that creates "follow hairs" in addition to hairs in the .tfx file. A follow hair is not simulated, but follows the motion of its "guide hair".

TressFXAsset::ProcessAsset();

Prepares data for runtime use, such as m_pRefVectors, m_pGolobalRotations, m_pLocalRotations, and m_pRestLengths for shape matching constraints.

An example looks like this:

```
FILE *fp = fopen("Ratboy_short.tfx", "rb");
TressFXAsset asset;
asset.LoadAsset(fp);
fclose(fp);
asset.ProcessAssset();
```

Layouts

Layouts specify how slots and constant buffers are set up for each kernel or shader. You will at least need CreateRenderPosTanLayout to start. Basic categories are listed here.

CreateRender*Layout	Used for drawing strands
CreatePPLL*Layout	For PPLL usage
CreateShortcut*Layout	For Shortcut usage
CreateSimLayout	Simulation
Create*SDFLayout	For signed distance field use.

Arguments include a "layout manager". This is what, from your engine or API, assigns slots and constant buffers based on names in the shader.

TressFX Object

TressFXHairObject contains the GPU data for an instance of a hair object, including per-instance structured buffers and constant buffer data. It's constructed from TressFXAsset. You must also provide an EI_CreateContext, which is the context required to allocate GPU resources. In DX11 would be the ID3D11Device, for example. For initializing and updating data, you need to provide the EI_CommandContext, which in DX11 would be ID3D11DeviceContext.

The calls you need just to render are the following:

TressFXHairObject::Create(…) Creates the GPU resources, and initializes them.

TressFXHairObject::DrawStrands(…) Binds resources and draws the strands.

TressFX Vertex Shading

TressFXStrands.hlsl contains the HLSL required in a vertex shader to put hair on the screen. There is a single HLSL function that generates the vertex data for export from a vertex shader.

As mentioned in the previous section, DrawStrands binds the necessary resources, and invokes the draw call.

Below is a complete vertex shader, using code supplied by TressFXStrands.hlsl. GetExpandedTressFXVert takes arguments listed, and generates the output vertex data in clip space. Eye postion is in world space. vertexId is, as indicated, the SV_VertexID system value. The third argument is the window size in pixels (x and y). The final argument is the view projection matrix.

The projection matrix is used in column-major style: `mul(mat, vec)`

If your projection matrix is set up to be row major, you can simply change the order by changing `MatrixMult` in TressFXStrands.hlsl.

```
float4x4 g_mVP;
float3 g_vEye;
float4 g_vViewport;

struct PS_INPUT_HAIR
{
    float4 Position    : SV_POSITION;
    float4 Tangent     : Tangent;
    float4 p0p1        : TEXCOORD0;
    float3 strandColor : TEXCOORD1;
};


PS_INPUT_HAIR VS_RenderHair_AA( uint vertexId : SV_VertexID )
{
    TressFXVertex tressfxVert =
        GetExpandedTressFXVert(vertexId, g_vEye, g_vViewport.zw, g_mVP);

    PS_INPUT_HAIR Output;

    Output.Position = tressfxVert.Position;
    Output.Tangent  = tressfxVert.Tangent;
    Output.p0p1     = tressfxVert.p0p1;
    Output.strandColor = tressfxVert.strandColor;

    return Output;
}
```

This covers the vertex shader.   A simple pixel shader outputting a constant color should be enough to at least see hair in the world.

Art Exporter

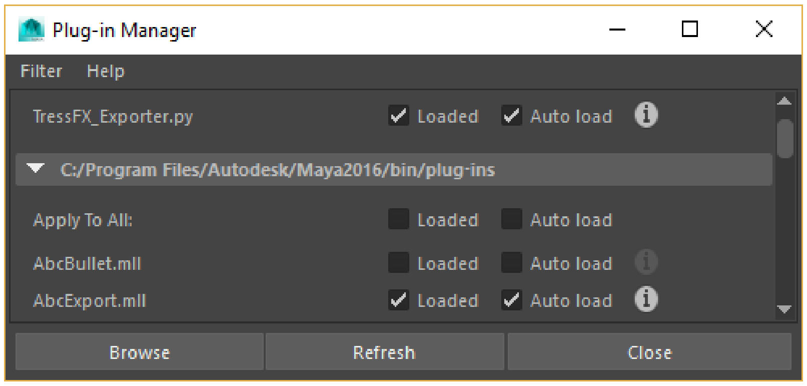
TressFX is designed to be compatible with your favorite hair modeler. All modeling is done within the modeling system of your choice, as long as you can turn them into splines at the end. We've used Shave and a Haircut, XGen and the 3ds Max native hair modeler, although the 3dS Max plugin is not included in this release.

In this section, we describe how to export your hair geometry as files through our Maya exporter, beginning with instructions on how to install the plugin.

Installing the Plugin

The Maya TressFX exporter is a single python file located in the following location: TressFX\Tool\Maya \TressFX_Exporter.py. To enable this plugin, follow these steps.

1. Copy TressFX_Exporter.py into Maya's plug-ins folder such as C:\Users\USER_NAME\ Documents\maya\plug-ins
2. Launch Maya.
3. Open Plug-in Manager and set Loaded and Auto load for TressFX_Exporter.py as below.



TressFX_Exporter.py [✓ Loaded] [✓ Auto load]

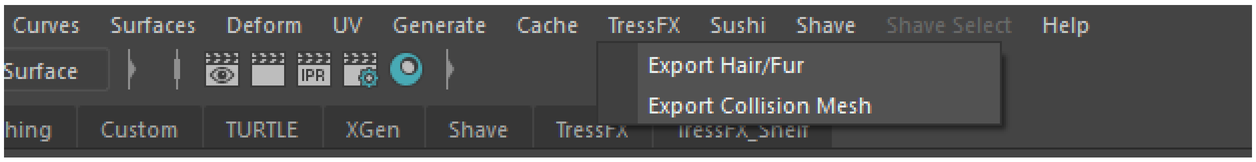
C:/Program Files/Autodesk/Maya2016/bin/plug-ins

Apply To All: Loaded Auto load

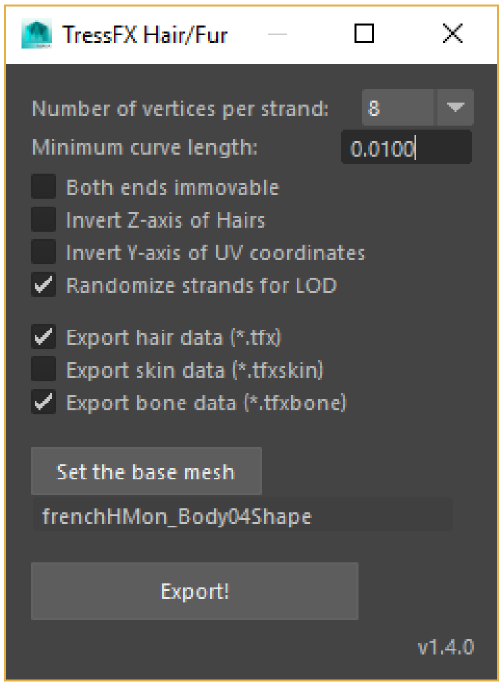
AbcBullet.mll Loaded Auto load

AbcExport.mll [✓ Loaded] [✓ Auto load]

4. Now, **TressFX** menu should appear on the main menu bar. Underneath it, there should two sub menu items: **Export Hair/Fur** and **Export Collision Mesh**.



5. Export Hair/Fur menu item will bring up the **TressFX Hair/Fur** window as below.


TressFX Hair/Fur

Number of vertices per strand: 8

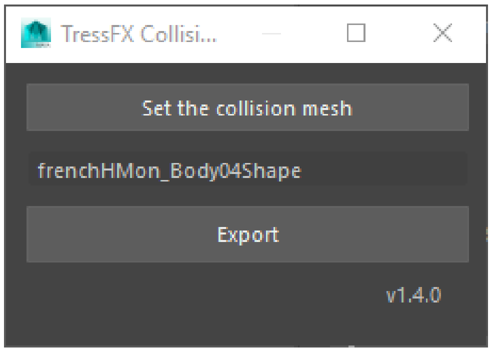
Minimum curve length: 0.0100

☐ Both ends immovable
☐ Invert Z-axis of Hairs
☐ Invert Y-axis of UV coordinates
✓ Randomize strands for LOD

✓ Export hair data (*.tfx)
☐ Export skin data (*.tfxskin)
✓ Export bone data (*.tfxbone)

frenchHMon_Body04Shape

6. Export Collision Mesh will bring up the **TressFX Collision** window as below.


TressFX Collisi...

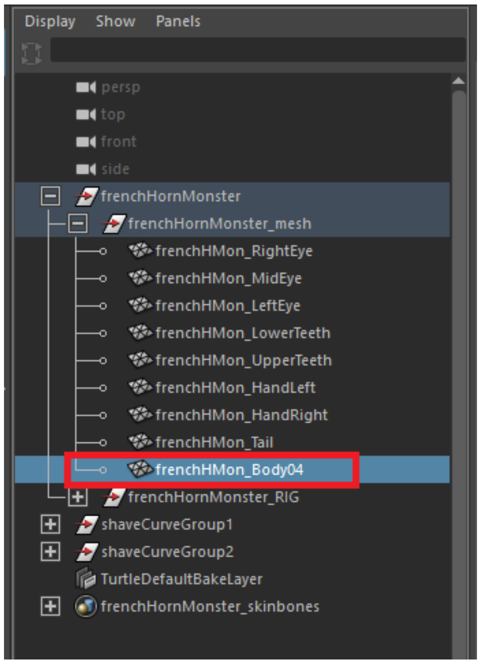
frenchHMon_Body04Shape

Exporting

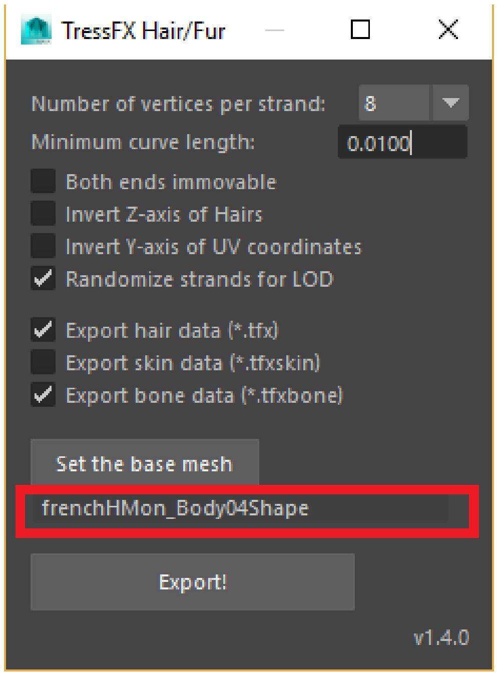
As mentioned, you are free to use any hair modeling tool that creates nurbs curves (splines). In this walkthrough, we will be using Maya. We tested the python exporter in Maya 2015 and 2016.

All hair curves should be converted into polylines. In our case, we use Shave and a Haircut's converting feature.

First, select the skin mesh and click **Set the base mesh** button. The selected mesh name will appear on the text box as below.



persp
top
front
side
frenchHornMonster
frenchHornMonster_mesh
frenchHMon_RightEye
frenchHMon_MidEye
frenchHMon_LeftEye
frenchHMon_LowerTeeth
frenchHMon_UpperTeeth
frenchHMon_HandLeft
frenchHMon_HandRight
frenchHMon_Tail
frenchHMon_Body04
frenchHornMonster_RIG
shaveCurveGroup1
shaveCurveGroup2
TurtleDefaultBakeLayer
frenchHornMonster_skinbones



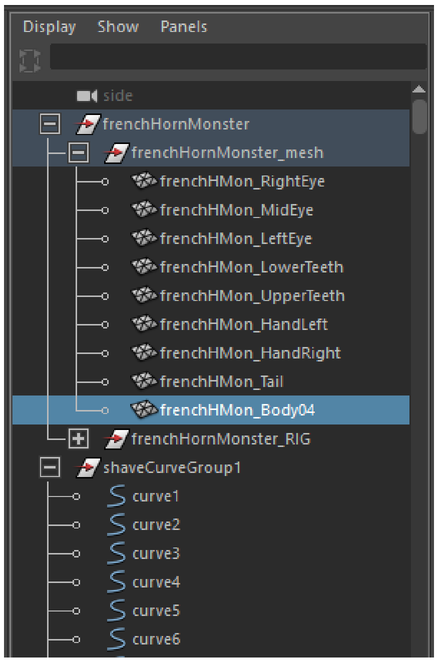
Number of vertices per strand: 8
Minimum curve length: 0.0100
☐ Both ends immovable
☐ Invert Z-axis of Hairs
☐ Invert Y-axis of UV coordinates
✓ Randomize strands for LOD
✓ Export hair data (*.tfx)
☐ Export skin data (*.tfxskin)
✓ Export bone data (*.tfxbone)
frenchHMon_Body04Shape

Now, choose the **Number of vertices per strand** option. In Rat Boy demo, we used 8 for the Short and 32 for the Mohawk. **Minimum curve length** is to filter out hair shorter than the input length. In some case, it is hard to get rid of short hair using modeling tool. This option will be handy in that case. If it is set to zero, then there will be no filtering.

Both ends immovable makes the both ending vertices get zero inverse mass. **Invert Z-axis of Hairs** inverts the Z component of hair vertices. This may be useful to deal with some engines using left-handed coordinate system. **Invert Y-axis of UV coordinates** inverts Y component of UV coordinates. **Randomize strands for LOD** randomizes hair strand indices so that LOD can uniformly reduce hair strands.

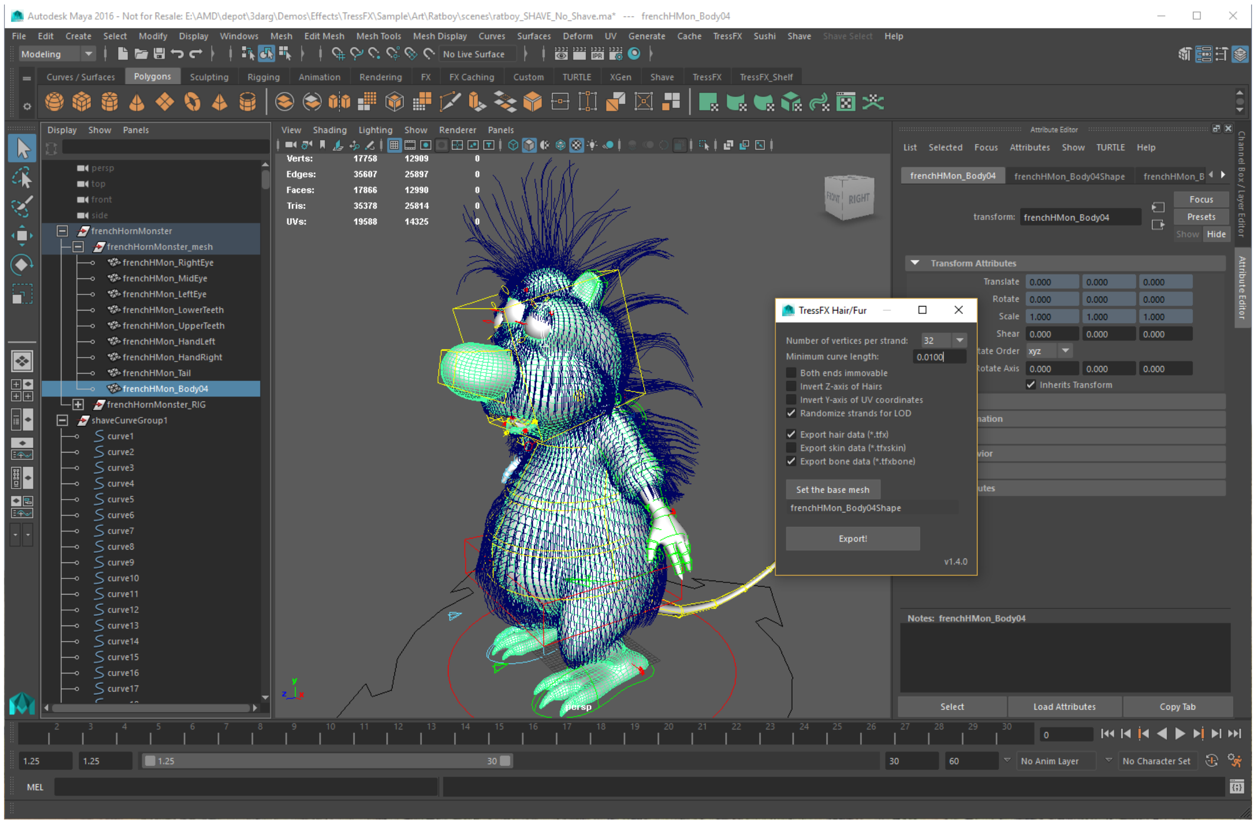
If you check **Export hair data (*.tfx)**, hair geometry file (*.tfx) will be exported as a file. It is a binary file containing hair strands and vertices data. **Export skin data (*.tfxskin)** exports a file containing mesh data for triangle transform-based animation. It was introduced in TressFX 3.0 but is being replaced with bone-based animation in 4.0. **Export bone data (*.tfxbone)** is to export bone animation data for the skin mesh. The last two options work only when the base mesh is set.

Select the top curve group containing individual curves as below. It is also possible to select each curve that you want to export, however in some case, it may take a while to select many curves in Maya.



Once the curve group is selected, click **Export** button and it will ask file names for *.tfx, *.tfxskin or *.tfxbone file based on the option selection.

For SDF, it takes an input mesh to generate SDF. The input mesh can be provided by the engine but it could be difficult to get it because most of engine is using vertex shader for skinning. In Rat Boy demo, we animate the collision mesh in compute shader. For this, we export the collision mesh using **TressFX Collision**. Select the collision mesh in Maya and click **Set the collision mesh** button. Then the selected mesh name will show up in the text box. **Export** button will let you save the *.tfxmesh as a file. Please note that one *.tfxmesh file is to generate one SDF. In Ray Boy demo, we used three collision meshes for body and two hands.


Autodesk Maya 2016 - Not for Resale: E:\AMD\depot\3darg\Demos\Effects\TressFX\Sample\Art\Ratboy\scenes\ratboy_SHAVE_No_Shave.ma* --- frenchHMon_Body04
File Edit Create Select Modify Display Windows Mesh Edit Mesh Mesh Tools Mesh Display Curves Surfaces Deform UV Generate Cache TressFX Sushi Shave Shave Select Help
Modeling
No Live Surface
Curves / Surfaces Polygons Sculpting Rigging Animation Rendering FX FX Caching Custom TURTLE XGen Shave TressFX TressFX_Shelf
Display Show Panels
persp
top
front
side
frenchHornMonster
frenchHornMonster_mesh
frenchHMon_RightEye
frenchHMon_MidEye
frenchHMon_LeftEye
frenchHMon_LowerTeeth
frenchHMon_UpperTeeth
frenchHMon_HandLeft
frenchHMon_HandRight
frenchHMon_Tail
frenchHMon_Body04
frenchHornMonster_RIG
shaveCurveGroup1
curve1
curve2
curve3
curve4
curve5
curve6
curve7
curve8
curve9
curve10
curve11
curve12
curve13
curve14
curve15
curve16
curve17
View Shading Lighting Show Renderer Panels
FRONT RIGHT
persp
Attribute Editor
List Selected Focus Attributes Show TURTLE Help
frenchHMon_Body04 frenchHMon_Body04Shape frenchHMon_B
Focus
Presets
Show Hide
Transform Attributes
Notes: frenchHMon_Body04
Select Load Attributes Copy Tab
No Anim Layer
No Character Set
MEL

```
Verts:   17758   12909   0
Edges:   35687   25897   0
Faces:   17866   12990   0
Tris:    35378   25814   0
UVs:     19588   14325   0
```

transform: frenchHMon_Body04

Translate	0.000	0.000	0.000
Rotate	0.000	0.000	0.000
Scale	1.000	1.000	1.000
Shear	0.000	0.000	0.000
Rotate Order	xyz		
Rotate Axis	0.000	0.000	0.000

✓ Inherits Transform

TressFX Hair/Fur

Number of vertices per strand: 32

Minimum curve length: 0.0100

- [ ] Both ends immovable
- [ ] Invert Z-axis of Hairs
- [ ] Invert Y-axis of UV coordinates
- [✓] Randomize strands for LOD
- [✓] Export hair data (*.tfx)
- [ ] Export skin data (*.tfxskin)
- [✓] Export bone data (*.tfxbone)

Set the base mesh

frenchHMon_Body04Shape

Export!

v1.4.0

Timeline: 1.25 1.25 1.25 30 30 60 0