TRESSFX 4.1/ENGINE INTEGRATION DEVELOPER GUIDE

For AMD TressFX/Epic Games Unreal Engine Integration and AMD TressFX/Radeon™ Cauldron (DirectX® 12/Vulkan® framework) implementation

THIS GUIDE:

- Describes new features of the TressFX integration into Unreal Engine 4.22 (branch available to developers with an Epic Games EULA/GitHub account) and AMD Radeon Cauldron (GPUOpen);
- Documents shader parameters, architecture, new features, Maya Exporter, and more;
- Includes basic tutorials for creating hair, using TressFX Exporter, and importing into Unreal.

AMD Immersive Technology

Contents

TressFX 4.x – Overview

The TressFX library implements AMD TressFX hair/fur rendering and simulation technology. The TressFX technology is designed to use the GPU to simulate and render high-quality, realistic hair and fur. TressFX makes use of the processing power of high-performance GPUs to do realistic rendering, and utilizes DirectCompute (DirectX® 11) and compute shaders (DirectX® 12/Vulkan®) to physically simulate each individual strand of hair.

This document discusses the new TressFX 4.1/Unreal 4.22 engine integration as well as the TressFX 4.1/Radeon™ Cauldron framework sample code. This version of TressFX — 4.1 — is designed for improved performance compared to earlier versions, including the simulation functionality as well as additional rendering and simulation features (shader enhancements) in both the Unreal integration and the Cauldron-based implementation. The TressFX Exporter plugin for Autodesk Maya, used to turn NURBS curves (splines) into TFX data files needed for hair and collision import, has also been re-designed to be easier to use, as well as improved with new features for greater export control.

A detailed explanation of the new features and the architecture is included, as well as some tutorials on using TressFX/Unreal and TressFX Exporter.

Note: Most of this document centers on TressFX/Unreal usage and components, including pictures of components and how to use them within Unreal 4.22.

Accessing TressFX via the Cauldron implementation will, of course, be different, because that implementation is focused on the code itself rather than integration into a commercial engine. However, a basic UI lets you update and change shader parameters in real time.

Differences with shaders, architecture, shader parameter usage and so forth are highlighted as appropriate per section. Most often, shader parameters are the same across the two implementations, especially in the physical simulation, but there are differences.

TressFX – from 3.0 to 4.0 to 4.1

Hair and fur simulating require that the hair stay attached to a skinned mesh. In TressFX 3.0, AMD introduced an animated bear model and used a triangle-based skinning system to animate root vertices. In 3.0, TressFX 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 — by accommodating skinning or blend shapes, for example.

In TressFX 4.0, AMD introduced a new bone-based skinning system for hair and fur. This new system was designed to allow developers to integrate TressFX into their engine more seamlessly. It also separated the hair or fur from its skin mesh so that multiple hair/fur objects could be used for the same character more easily. Since the bone skinning transforms are an input to the simulation, it meant that the hair/fur simulation could produce the same animated results as the character.

In terms of collision, TressFX 4.0 introduced Signed Distance Field (SDF). For a simple hair simulation, it might be suitable to use a few spheres or capsules to represent collision areas for the head and neck. However, long fur on a skinned mesh is more complex and makes it more difficult to model using this

simpler method. Additionally, the simpler model doesn't scale well. A signed distance field more generally represents an arbitrary, deforming mesh with a single data structure.

Lastly, TressFX 4.0 introduced Velocity Shock Propagation (VSP). In real-time games, it is hard to predict a character's movement, and the fast animation transition often causes an extremely high and sharp velocity change. In the previous TressFX version, this kind of excessive acceleration was addressed by increasing stiffness, damping and the number of iterations for constraints. In addition to the obvious difficulty of tuning so many interacting physics parameters, increasing the number of iterations and resulted in a significant computational cost. With VSP, the velocity of an animated root vertex is propagated throughout the rest of vertices in the strand. Also, VSP checks pseudo acceleration and increases the VSP value when the pseudo acceleration is higher than a certain threshold. It's important to note that this doesn't entirely put the burden on VSP. It is often still desirable to tune the other physics parameters in order to achieve a desired 'look'.

TressFX 4.1 is designed to be a performance and feature enhancement of TressFX 4.0. The physics simulation shaders have been improved, a level of detail (LOD) system has been added for the hair, and some new rendering features (shader parameters) have been added. The TressFX Exporter plugin for Autodesk Maya has also been updated with new features and a new UI to allow more control and error checking when exporting splines (NURBS curves) into the required TressFX file formats (.tfx, .tfxbone, .tfxmesh) for hair, bone and collision mesh data. TressFX 3.0 export has been left in the Exporter, but should be considered untested, unsupported and generally deprecated.

TressFX 4.1 is also designed to be a demonstration of engine integration/implementation. There is a TressFX/Unreal minimal engine integration, as well as a simpler demonstration of how to implement/wrap TressFX so that it can be easily dropped into an existing codebase, such as the AMD Cauldron framework.

TressFX Engine Integrations and Installation

In order to use the TressFX/Unreal 4.22 engine integration, a developer must sign the Epic Games EULA and get access to the Epic Games Unreal GitHub site. Then the developer will have access to the AMD TressFX/Unreal 4.22 branch (as a patch to the engine). Installation requirements may vary, but developers should assume that they need a system capable of building Unreal 4.22. TressFX is a hair rendering and simulation technology, so can be very demanding on any system. All testing and development of TressFX/Unreal was done on AMD Ryzen™ Threadripper™ (1950X 16-core) processor/Radeon™ RX Vega systems. An equivalent development system should be assumed. For a target gaming device, such a system will vary depending on the demands of the game/application, including how much hair is needed and expected to be simulated. Therefore, precise numbers cannot be given.

The TressFX/Cauldron implementation, by contrast, is based on AMD technology based and is available on GPUOpen. Again, developer requirements will depend on the end goal, but a system capable of building and running DirectX® 12 and/or Vulkan® is the minimum.

Note: To be precise, we released our patch and did our Unreal 4.22 testing using Unreal 4.22.3.

Microsoft® Visual Studio 2017 was used for development of TressFX/Unreal 4.22 and an equivalent environment is recommended for TressFX/Unreal.

Visual Studio 2019 was used for development of TressFX/Cauldron and an equivalent environment is recommended for TressFX/Cauldron.

The TressFX/Unreal Engine Integration (the Basics)

The TressFX/Unreal engine integration demonstrates the minimum integration needed for TressFX 4.x to run with Unreal Engine. Because TressFX needed access to the rendering pipeline, it was not sufficient to have TressFX as a 3rd party plugin, such as AMD FemFX. Developers wishing to further the integration or customize it for their own requirements should find this basic level of integration a helpful first step in that process.

TressFX 4.x (Unreal Engine 4.22 integration and Cauldron) includes a number of new shader parameters as well as earlier ones. This Guide documents each shader parameter as well as architectural elements particular to the engine being used (Unreal or Cauldron). Some topic areas include links to other pages with more information and/or example images.

Note: Earlier (TressFX 3.x/4.0) shader parameters, however, may not function the same way as before, so no assumptions about previous usage should be applied.

In TressFX/Unreal, the shader parameters are split across the TressFXComponent and the two TressFX materials, the TressFX Rendering Material and the TressFX Simulation Material. The TressFX component holds the hair asset as well. For collision meshes and the SDF field used in collisions, there are separate components in addition to the basic TressFXComponent and are referred to as the TressFX Collision Mesh and TressFX SDF components.

Multiple TressFX components would be a common use case, such as one for the top of the head and another for the eyebrows or beard — or even multiple overlapping components for a layered hair look. As for the materials, rendering and simulation, these materials should not be confused with Unreal's traditional materials, a part of its rendering system. These TressFX materials are simply ways to manipulate shader parameters and use them in a grouped manner — as in having multiple rendering materials for different situations, or the ability to share a simulation (or rendering) material across multiple TressFX components.

Because Lighting and the number of bones in a skeletal mesh are also important to the functioning of TressFX, these areas are also summarized in this document.

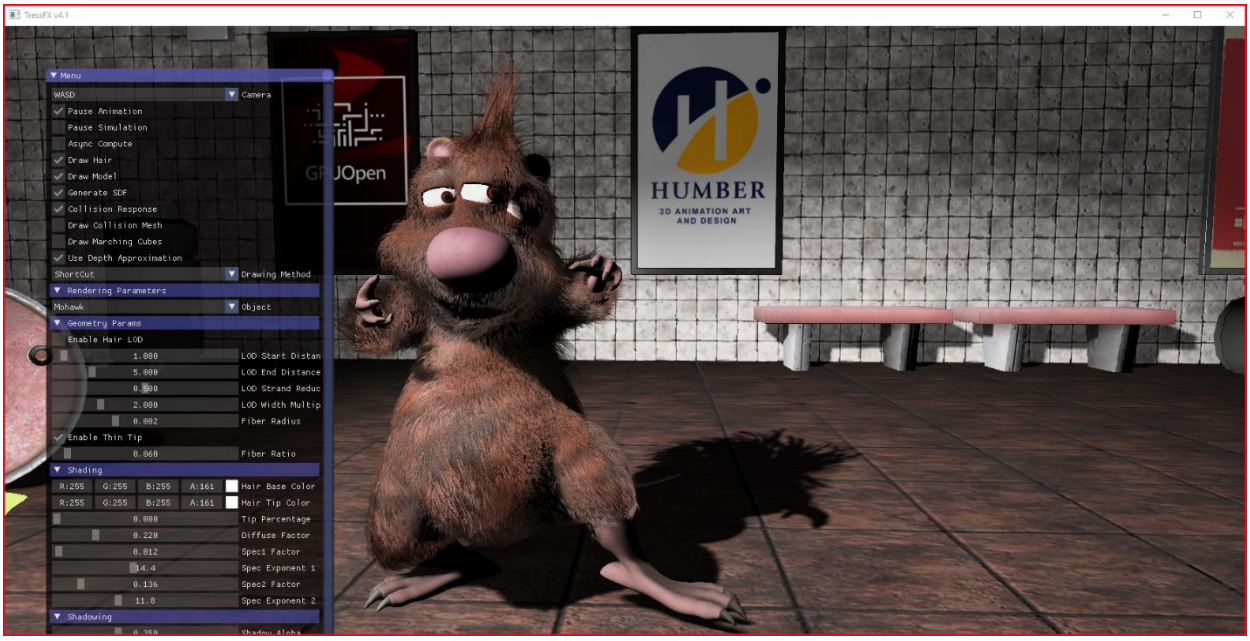
The TressFX/Radeon Cauldron Framework (the Basics)

Cauldron is a framework library for rapid prototyping using either the Vulkan® or DirectX® 12 APIs. It is developed by AMD and used internally by several teams. Cauldron is open source and has been designed to be simple and easy to extend. Please see this GPUOpen link for more information on Cauldron.

TressFX/Cauldron is not an engine integration per se, although the TressFX/Cauldron developers worked closely with the Cauldron developers when some TressFX requirements for Cauldron were needed. TressFX uses a DirectX® and Vulkan® wrapper to bridge between TressFX and Cauldron.

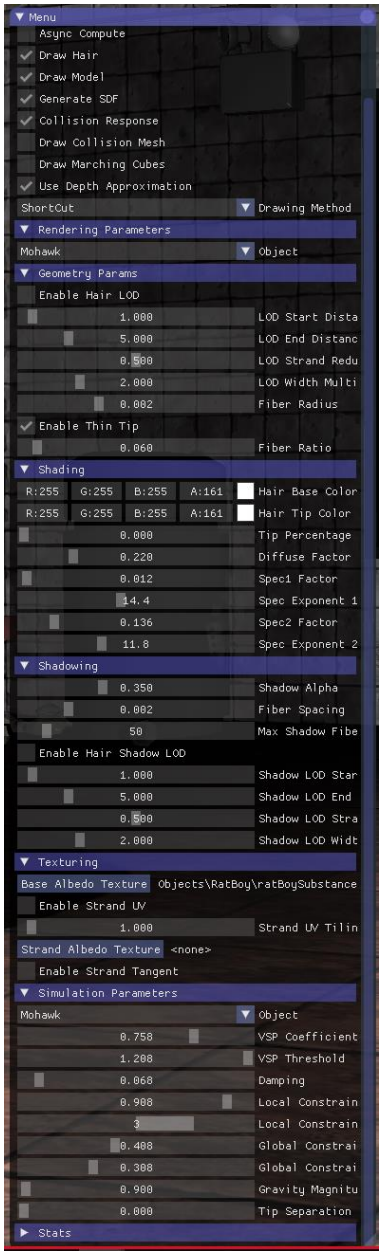
Because Cauldron is highly simplified, compared to production quality game engines, it is much easier to show the 'bare metal' requirements of TressFX without getting bogged down in specific engine details or architectural needs. The goal of this TressFX/Cauldron 'integration' is to demonstrate how TressFX can be cleanly dropped into an existing codebase. Therefore, the focus of TressFX/Cauldron is on the code requirements and suggested implementation, rather than on production use/ease of use (such as with TressFX/Unreal).

TressFX/Cauldron User Interface (UI)



The UI for TressFX/Cauldron is basic when compared to the complexity of TressFX/Unreal. To be loaded, files should be hard coded into the source itself at this time, although as Cauldron matures, this may change.

However, shader parameters are easily accessible with a menu and fundamental control system (checkboxes, dropdowns, sliders and so forth). You can change shader values at runtime and see the effects happen immediately in the scene.



TressFX Architecture

TressFX 4.0/4.1 is a bone-based skinning system for hair/fur simulation and rendering, and uses SDF for collision. The general process flow is to model hair as splines (NURBS curves), export the splines using the rigged (skinned) mesh as a reference into the require data files, import those files and turn them into data on the CPU side, then simulate and render using compute shaders and shaders. This includes getting the required information from the engine/rendering pipeline in order to properly react and simulate (bone information, lighting, etc.)

Collision Mesh

A collision mesh is a triangle mesh and an input to the SDF. It does not need to be the same as a 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.

Signed Distance Field (SDF)

SDF is a grid-based representation for a polygonal mesh.

In TressFX, the dense grid is used 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 the Rat Boy demo, for example, 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 required, it's best to minimize holes in the mesh that are larger than a grid cell unless this is a portion that won't interact with the hair.

Unlike TressFX 3.x skinning, no requirements are imposed on Maya mesh consistency.

Velocity Shock Propagation (VSP)

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

VSP has a control value ranged from 0 to 1. If the value is zero, there will be no VSP. If VSP 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 the VSP value to 1 when the pseudo-acceleration passes it. It is designed to be particularly effective when the character makes a sudden movement.

Marching Cubes (MC)

Marching cubes can be used to visualize the SDF for debugging purposes. If the 'Draw marching cubes' checkbox is selected, a visualization will appear in yellow.

Guide and Follow Hair System

This system was introduced in TressFX 2.0 and is used the same way in TressFX 4.0/4.1. In terms of collision with the SDF, both guide and follow hair will get a response from the system. It is still possible to redesign such that only guide hair would be affected by the SDF, but the overhead is small enough to use SDF collision for all hair.

Rendering

The implementation of the rendering part of TressFX 4.0/4.1 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 sufficient 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 designed to achieve good looking results. In TressFX 4.0/4.1, 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 designed to give 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 can be 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 can provide a softer look for the hair, similar to real hair. If transparency was not used, the strands of hair could 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 (PPLL), and the other uses a multi-pass method referred to as Shortcut.

The main decision to make is whether to use per-pixel linked lists or Shortcut. The Shortcut method can be generally faster, with an easier memory bound, but at the expense of some quality. The TressFX 4.1/Unreal 4.22 Engine integration uses Shortcut (although the shader for PPLL may still be present in the branch, it was/is unmodified/untested/unused). TressFX 4.1/Cauldron uses both Shortcut and PPLL.

TressFX/Unreal 4.22 Integration Architecture

The TressFX integration is primarily split into two modules, TressFXEditor & TressFXComponent. You can find these in **Engine/Source/Editor/TressFXEditor/** and **Engine/Source/Runtime/TressFX**, respectively.

TressFX Engine Hooks

As TressFX Unreal is implemented as an add-on to Unreal (and not fully integrated, which would mandate we maintain the full engine), there are a few locations in Unreal's code base that hooks must be added to call into various parts of our TressFX implementation, and for general scene and visibility tracking.

We've strived to keep these changes minimal, and have attempted to bookend all code changes in Unreal code with:

/ AMD TRESSFX CODE INSERT */**

As of the time of this writing, there are only five files within Unreal that are modified to accommodate TressFX. These files are the following:

- ***PrimitiveViewRelevance.h*** (to add hair-related relevances)
- ***DeferredShadingRender.cpp*** (to call into our simulation and rendering stages from Unreal's main rendering loop)
- ***SceneRendering.h*** (to add arrays of primitive scene information for TressFX related functionality)
- ***SceneVisiblity.cpp*** (to set up TressFX related views)
- ***ShadowSetup.cpp*** (to allow for hair to render as part of Unreal's shadow map rendering)

TressFXEditor Module

As the name implies, this is an *editor only* module. It contains the code required to import **TressFXAssets** and the asset actions for them. The **TressFXFactory** files contain the factories for the **TressFXHairAsset** as well as the **TressFXMeshAsset**.

Both factories use import data classes (**UTressFXImportData** and **UTressFXMeshImportData**) which will contain the user adjustable data at runtime. Widgets are created based from these classes. Users should make sure they set the proper skeleton they plan on using at runtime. The factory will go over the Bone data stored in the TFX files and compare it to the skeleton you selected. If the bones have improper indexes, it will try to find the correct bone by name and change the indexes, so the Assets match the skeleton.

The **TressFXHairAsset** factory will first import the data into the **TressFXAsset** class, then create a **TressFXHairAsset** from that data. The **TressFXMeshFactory** will put all the data into a string which will then get parsed for the information.

Finally, there are two more factories for the material classes (**UTressFXRenderMaterial** and **UTressFXSimulationMaterial**). These simply create new **UTressFXMaterial** and **UTressFXSimMaterial** UObjects. You can find these materials in the **Miscellaneous** section.

TressFXComponent Module

TressFXMaterial and TressFXSimMaterial

These are both standard **UObjects**. They are used to define a hair component's simulation behavior and hair rendering parameters. These are created through the **Miscellaneous** section in the content browser.

TressFXComponent

This is the primary interface for the hair and the editor. This class needs to be attached to the appropriate **SkeletalMeshComponnet** that is associated with the hair asset you plan to use with this component. This is the class where you can set individual settings for different hair assets.

TressFXSceneProxy

The scene proxy is responsible for creating and maintaining the dynamic render data known as **FTressFXHairObject**, and sending the dynamic constants to the renderer. TressFXRenderResources

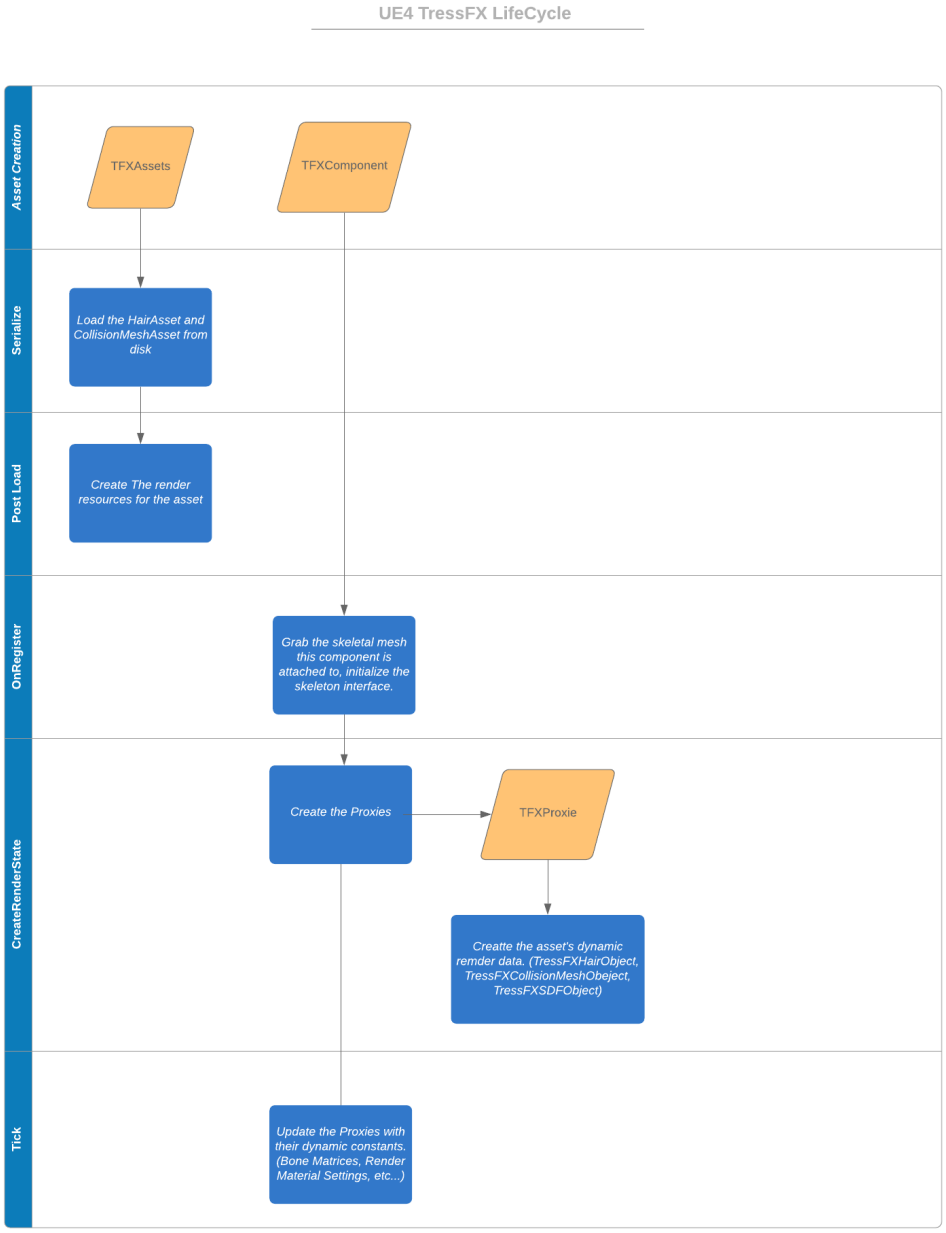
The TressFX render resources file contains our **FTressFXBuffer**. This is a custom **FRenderResource** we use for our buffers.

TressFXHairAsset and TressFXMeshAsset

These are the two data classes which will be serialized and saved to disk. They contain the static data for the hair assets and the collision mesh assets. The data will be read from disk during the serialize function, then the buffers get initialized during the post load function. These classes are used to create the dynamic data in their respective Proxies and are required for proxy creation.

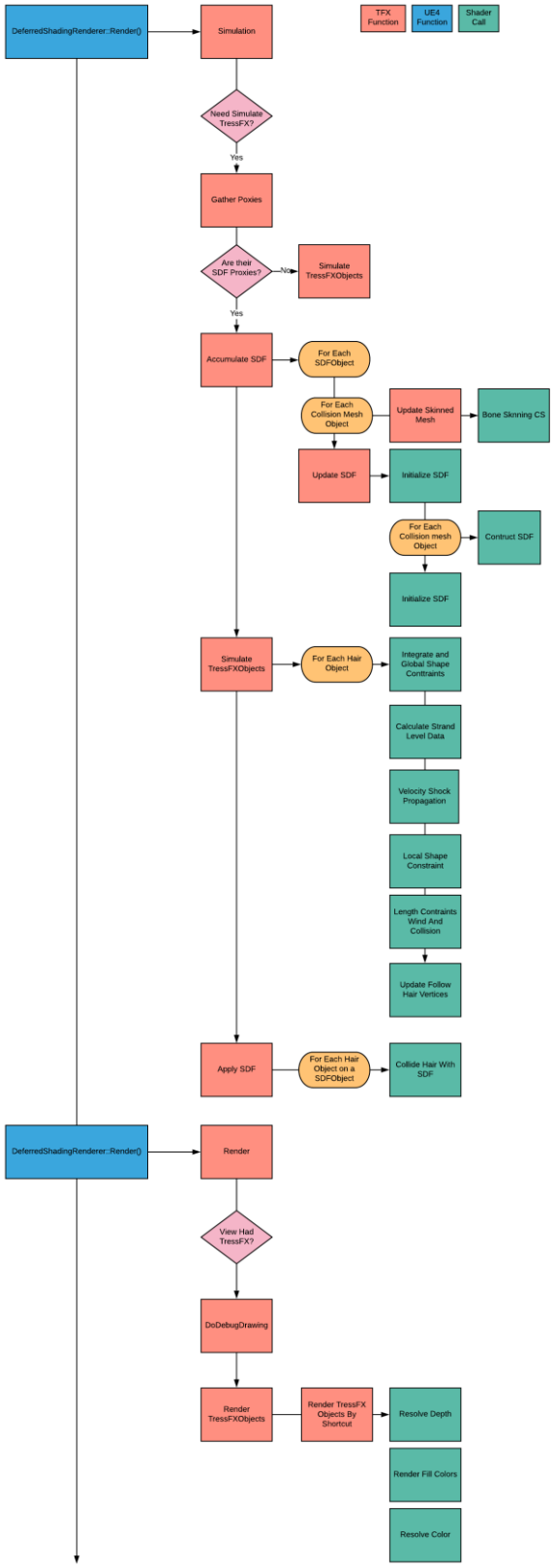
TressFXCollision

The collisions in TressFX are broken into two parts: the **TressFXCollisionMeshComponent** and the **TressFXSDFComponent**. The Collision mesh component needs to be attached to the same skeletal mesh it was exported from. The signed distance field component can be attached to anything. By default, it uses its parent's bounds, however, you can override the bounds by using the box component on the SDF component. It will generate its SDF by using the collision components that are added to its **TressFXCollisionComponents** array, then it will collide the hair with the SDF that's in its **TressFXHairComponents** array. You just need to add components to those arrays to get collisions to work.



TressFXRenderer

The renderer is responsible for all rendering and GPU activity for TressFX. The proxies will get collected by the engine, then placed inside their respective arrays inside ***SceneRendering.h***. Then the TressFXRenderer will go through these lists and call the appropriate shaders to simulate, collide and draw the hair.


TFX Function
UE4 Function
Shader Call
DeferredShadingRenderer::Render()
Simulation
Need Simulate TressFX?
Yes
Gather Poxies
Are their SDF Proxies?
No
Simulate TressFXObjects
Yes
Accumulate SDF
For Each SDFObject
For Each Collision Mesh Object
Update Skinned Mesh
Bone Skinning CS
Update SDF
Initialize SDF
For Each Collision mesh Object
Contruct SDF
Initialize SDF
Simulate TressFXObjects
For Each Hair Object
Integrate and Global Shape Contraints
Calculate Strand Level Data
Velocity Shock Propagation
Local Shape Constraint
Length Contraints Wind And Collision
Update Follow Hair Vertices
Apply SDF
For Each Hair Object on a SDFObject
Collide Hair With SDF
DeferredShadingRenderer::Render()
Render
View Had TressFX?
DoDebugDrawing
Render TressFXObjects
Render TressFX Objects By Shortcut
Resolve Depth
Render Fill Colors
Resolve Color

TressFX/Radeon Cauldron Architecture

A TressFX/Cauldron sample is built on top of Radeon™ Cauldron, which is an open sourced framework used at AMD for sample creation. It supports both DirectX® 12 (DX12) and Vulkan® implementations. For more information about Cauldron, or to download the source/examples, refer to https://gpuopen.com/gaming-product/caudron-framework/

With this release, we aim to clean up our previous TressFX 4.0 release to make it easier for developers to integrate TressFX.

On top of that, TressFX 4.1 provides over the previous releases: faster Velocity Shock propagation, simplified Local Shape Constraints; and reorganization of the order of dispatches, among other things.

Note: The TressFX GitHub site (GPUOpen) may also contain other living documents that are specifically targeted at the TressFX/Cauldron implementation. These documents may contain more recent updates and changes than this developer guide, which is shared across TressFX/Cauldron and TressFX/Unreal locations.

Abstraction of Graphics APIs and Engine Interface

TressFX/Cauldron provides two things:

- A self-contained solution for hair simulation
- A reference renderer for hair

Both aim to provide an easy starting point for developers that want to integrate TressFX with their own technology. To that end, it iterates on the Engine Interface that was introduced with TressFX 4.0.

Both the renderer and the simulation go through this interface, but we expect developers to want to implement their own hair rendering that integrates perfectly into their pipeline. They can use the renderer we provide as a guide for this.

The hair simulation consists of asset loading code and the TressFX compute shaders which should be self-contained enough to be integrated into another codebase without substantial changes to the source files.

To do this, a developer can implement all necessary functions in our Engine Interface (**EI_**), or replace it with their own calls and data structures (similar to how the TressFX/Unreal integration was done, which is tightly bound to Unreal's architecture).

The graphics hardware is accessed through the interface **EI_Device**, which is implemented for two (2) graphics APIs (Vulkan® and DirectX® 12) in ***VKEngineInterface.h*** and ***DX12EngineInterface.h*** respectively.

It is designed to map well to any modern graphics API.

The **EI_Device** interface is used for the following:

- Allocate device buffers, images and samplers (**EI_Resource**)
- Create Render Target Sets (**EI_RenderTargetSet**)
- Create Bind Layouts and Bind Sets (**EI_BindLayout, EI_BindSet**)

- Create Graphics and Compute PSOs (**EI_PSO**)
- Record and submit command buffers (**EI_CommandContext**)

Note: Please note that we use **std::unique_ptrs** for all resources we allocate through the interface to simplify memory management.

As Cauldron is a *barebones* framework, the implementation of this layer is straightforward.

The interface wraps all resource creation/management/destruction functionality into an **EI_Device** pointer that TressFX sample will use to set up, simulate, and render.

The scene management is accessed through the interface **EI_Scene**.

Cauldron implements a loader for GLTF files which contain the whole scene we use in the TressFX sample. That is why the **EI_Scene** interface is implemented in ***GLTFImplementation.h*** to interface the GLTF scene that Cauldron uses, including:

- Camera information
- Light information
- Skeleton transformations
- Bone IDs by name

Folder Structure

- **src/**
  Originally from the sushi-specific TressFX 4.0 code (from ***amd_tressfx_sample***). Now has Cauldron specific TressFX 4.1 code shared between DX12 and vk (DirectX® 12 and Vulkan®).
  - **src/Math**
    TressFX math libraries.
  - **src/Shaders**
    Shader sources in HLSL lie here. Cauldron supports HLSL for both DirectX® 12 and Vulkan®.
  - **src/TressFX**
    Platform agnostic TressFX files.
  - **src/Common**
    Cauldron specific but rendering API agnostic implementation.
  - **src/VK**
    Cauldron/Vulkan® specific code (contains an engine interface implementation for vk).
  - **src/DX12**
    Cauldron/ DirectX® 12 specific code (contains an engine interface implementation for DirectX® 12).
- **maya-plugin/**
  Contains the python Maya plugin. It is identical to the Unreal Maya plugin.
- **libs/cauldron/**
  Contains AMDs Cauldron open source library for demo creation.

TressFX Hair

The **TressFXAsset** contains the loading code to load a hair asset from storage. At runtime, a **TressFXHairObject** is created which allocates all necessary runtime resources through **EI_Device.** The runtime resources are split into dynamic buffers (**TressFXDynamicState**):

- Positions (this frame, previous frame, and the one before (previous previous))
- Tangents
- Strand level data

  Note: This (strand level data) is an optimization over TressFX 4.0 to speed up the velocity shock propagation and skinning.

As well as static buffers filled with data from the TressFX asset:

- Initial hair positions
- Rest lengths
- Strand type
- Follow hair root offset
- Bone skinning data
- Hair texture coordinates

Additionally, the hair object contains three (3) uniform buffers:

- **m_SimCB**
- **m_RenderCB**
- **m_StrandCB**

TressFX Simulation

The core simulation implementation is contained in ***TressFXSimulation.h/cpp***.

- The **Initialize** function creates the necessary PSOs.
- The **Simulate** function dispatches all simulation compute shaders:
  - **IntegrationAndGlobalShapeConstraints**

    Dispatched on hair segment level, it does the timestep for each of the hair segments, and resolves the global shape constraints.
  - **CalculateStrandLevelData**

    Dispatched at strand level. This is an optimization over TressFX 4.0, it precalculates the strand level data (rotation and translation for velocity shock propagation and the skinning quaternion).
  - **VelocityShockPropagation**

    Dispatched at segment level. This calculates the velocity shock propagation.
  - **LocalShapeConstraints**

    Dispatched at strand level. Resolves the Local Shape Constraints. This has been much simplified over TressFX 4.0.
  - **LengthConstriantsWindAndCollision**

Dispatched at segment level. This calculates the rest of the simulation, length constraint and applies collision from capsules (which we don't use in this sample).
Note: Yes, Constraints is spelled wrongly. (This is how it is spelled in the code at the time this document was written, so it is duplicated here). We are aware of this issue. Please be aware that some names, like this, may be changed.

- **UpdateFollowHairVertices**
  Dispatched at segment level. This updates the follow hair according to the guide hair.

The collision implementation is in ***TressFXSDFCollision.h/cpp***. It encapsulates the Signed Distance Field (SDF).

- **Update** dispatches three (3) compute shaders for SDF generation:
  - **InitializeSignedDistanceField**
    This initializes the signed distance field. If the signed distance field is too large around the object, this can become a bottleneck. That is why bounding boxes should be chosen as small as possible around the collision object.
  - **ConstructSignedDistanceField**
    This renders the collision object into the signed distance field.
  - **FinalizeSignedDistanceField**
    Finalizes the SDF.
- **CollideWithHair** dispatches
  - **CollideHairVerticesWithSdf**
    This collides the hair segments of a hair object with the SDF.

Physics Parameters

For a description of the shader physics (simulation) parameters, see The TressFX Simulation Material – Shader Parameters, later in this document. The simulation shader parameters are identical across TressFX 4.1/Cauldron and TressFX 4.1/Unreal 4.22.

TressFX Rendering

The core implementations for hair rendering are contained in ***TressFXShortcut.h/cpp*** (for the Shortcut algorithm) and ***TressFXPPLL.h/cpp*** (for the PPLL algorithm). Both files follow a similar flow:

- The **Initialize** function will take care of creating all resources for the implementation (render targets/textures, UAV/Constant buffers, render pass sets, and bind sets). Once resources have been created, all pipeline state objects (PSOs) are created.
- The **Draw** functions are called from the main sample loop and are responsible for the rendering of the hair into the sample's back buffer. This starts with a call to **Clear()** which clears all UAVs and buffers to needed initialized values, and proceeds to go through the steps needed to render

each hair algorithm. (For an overview of Shortcut and PPLL please see Algorithms Overview: ShortCut (Unreal and Cauldron) and PPLL (Cauldron).)

- The **UpdateShadeParameters** functions are where all settings needed for rendering are updated. This is called into from the main sample loop.

The main body of the sample can be found in ***TressFXSample.h/cpp***. It represents the main entry, update, and shutdown interface for Cauldron. The main functions of interest are:

- **OnCreate()**
  This is the entry point to the sample. The first thing done here is the initialization of the DX/VK interface layer that we use throughout the application. Once this is done, the scene is initialized, sample resources are created, the UI (courtesy of DearIMGui) is initialized, and the Scene/Models are loaded.
- **OnDestroy()**
  This is one of the last things called prior to the application terminating. At this point, the GPU is flushed of any remaining commands, and all resources are destroyed and their memory freed.
- **OnEvent()**
  Windows® event handler, and also redirects input to UI system for UI interaction.
- **OnResize()**
  Called when the window is resized by the user. Because of the number of resources that need to change when this occurs, this function will flush the GPU, destroy and re-create necessary resources, and re-create all PSOs and that are back buffer related.
- **OnRender()**
  This is the sample's "main loop" which consists of the following actions:
  - Calculate updated time delta and begin the frame (queries various data buffers for the frame from glTF and updates/uploads various constant buffer data).
  - Build out the updated IMGui UI elements for the frame.
  - Update and run the hair simulation.
  - Update lighting information and render shadow maps for necessary lights (calls glTF shadow map rendering functions and hair drawing functions).
  - Render the glTF scene. We make use of glTF for scene/models as well as camera, and lighting information.
  - Update hair parameters and render the hair with the proper implementation (Shortcut or PPLL).
  - Generate the Marching Cubes.
  - Draw the collision mesh if collision mesh debugging is enabled.
  - Draw the SDF if SDF debugging is enabled.
  - End the frame (renders the UI and submits command buffers).

Optimization Details

Simplified Local Shape Constraints

- Removed iterative generation of transformations along the strand — instead, we are able to reconstruct the parent's transformation directly.
- **globalRotation**, **localRotation** and **refVecsInLocalFrame** buffers from TressFX 4.0 are removed, thus memory is saved.
- **refVecs** are generated from the bind pose.
- For each segment, we look at the parent segment's orientation, and calculate the rotation that transforms the parent's bind pose into the parent's current pose. We then apply that rotation to the current segment's bind pose, and use that as the target position for the constraint.

The results are very close. However, it seems more robust than the previous implementation. Numerically, this new approach may introduce less error, which, upon reflection, shouldn't be that surprising.

Rendering Algorithms Overview: ShortCut and PPLL

At the core of the TressFX rendering technology is the Order Independent Transparency (OIT). TressFX has two options for the OIT rendering, the original Per Pixel Link List (PPLL) and ShortCut. ShortCut is the newer Order Independent Transparency (OIT) option. It's inspired by the method presented by Eidos-Montréal and Hybrid Transparency. Whereas our original Per Pixel Linked List (PPLL) method focused on the front $k = 8$ or so layers of hair, ShortCut is good for cases when you can get away with $k = 2$ or 3, and you are looking for reduced memory usage. .

It does require some forethought on how to build your models, however, as it comes with different performance characteristics, and a quality trade-off. But between the simpler memory bounds and the potential for higher performance, we expect it to be a popular choice.

The four main steps are outlined below.

1. Render hair geometry, using a sequence of InterlockedMin calls to update the list of k nearest fragments while computing an overall alpha.
2. Screen space pass that puts the kth nearest depth in the depth buffer for early z culling in the next step.
3. Render hair geometry again. Shade the fragment and write or blend the color (depending on variant). [earlydepthstencil] focuses shading cost on the front k.
4. Screen space pass that does the final blending.

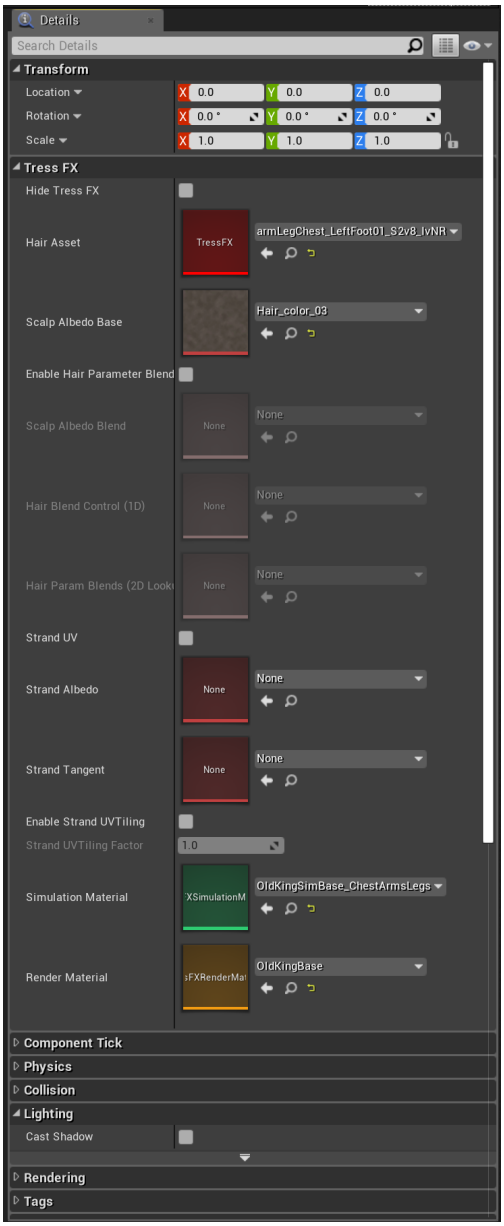
With the original PPLL method, you needed to allocate a single memory pool that is large enough for all hair fragments, not just the front k. With ShortCut, you only need space for the front k layers: 4 bytes for each depth, 4 bytes for each color, and 4 bytes for an accumulated alpha term for each pixel. Another difference ShortCut has with the PPLL method, is that although you still get the performance advantage of only shading the front k layers, you don't need to store the shader inputs in screen space.

The main drawback with ShortCut is in the extra geometry passes. It may still be a performance improvement when the depth complexity is high relative to the geometry cost, it trades the improved performance for a slightly lower quality result than the PPLL method.

Note: ShortCut is implemented in both TressFX/Unreal and TressFX/Cauldron. PPLL is only implemented in Cauldron.

Using TressFX/Unreal Components and Materials

TressFXComponent



TressFX Section

This section of a TressFX Component Details panel contains all the relevant TressFX parameters and settings for the component. This includes the Hair Asset and the TressFX Simulation and Rendering Materials. The collision mesh and SDF field settings are handled by their own components and used via the Unreal blueprint system (during construction).

See The TressFX Component for an in-depth discussion of the TressFX Component. There is a lot to cover.

Lighting Section

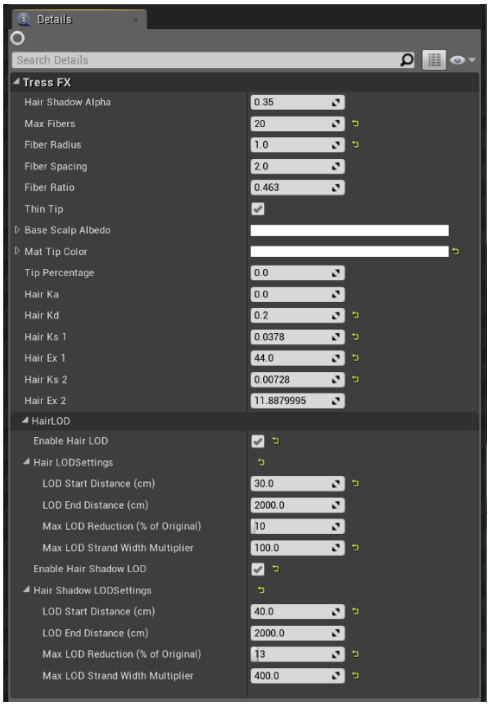
This section of the TressFX Component details panel is common to many other Unreal actors. It is mentioned because in order to see the hair cast shadows, not counting hair self-shadowing, you should select the Cast Shadow checkbox.

Cast Shadow (checkbox) — select it to have the hair cast shadows from Point Lights, Spot Lights or Directional Lights (for *Stationary* Directional Lights, Cascaded Shadow Mapping should be enabled on the Directional Light, see Lights Heading on this page).

Note: TressFX only supports Movable lights for shadow casting.

TressFX Materials

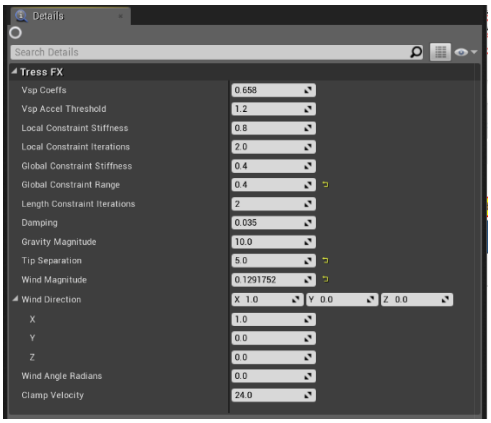
TressFXRendering Material



The TressFX Rendering material contains all rendering shader parameters and settings. This way you can use one or more materials across TressFX Components. If you do not use a Rendering Material on your TressFX Component, default shader settings will be used instead.

See The TressFX Render Material – Shader Parameters for more details on each parameter in the material.

TressFXSimulation Material



The TressFX Simulation material contains physical simulation shader parameters and settings. This way you can use one or more materials across TressFX Components. If you do not use a Simulation Material on your TressFX Component, default shader settings will be used instead.

See The TressFX Simulation Material – Shader Parameters for more details on each parameter in the material.

Lights

TressFX 4.x/Unreal Integration supports three Unreal light types: Point Light, Directional Light and Spotlights. TressFX is currently limited (in code) to twenty (20) non-shadowed lights (lights that do not cast shadows), five (5) spotlights (shadow casting), one (1) point light (shadow casting), and 1 directional light (shadow casting). This can be changed in the code, see TressFXLighting.h. So in summary, TressFX limits all lights to 27, and shadow casting lights to seven (7).

Because of architecture changes made by Unreal in 4.22, TressFX shadow casting lights must be Movable. Also, to avoid banding effects (streaks of light and dark shadows, or speckling of light and dark areas), move the lights in closer and reduce the coverage area to just the hair. This forces Unreal to create a shadow map that is finer grained and just focused on the hair. Additionally, set your shadow resolution as high as your performance allows (such as very high-level quality, i.e. 2048 max resolution or even higher if your applications' performance can handle it).

For example,

r.ShadowQuality=5
r.Shadow.MaxResolution=2048

Note: Lights are added to the list of lights applying to TressFX as they are added to the scene (level). So, to remove a light, either use light channels, or delete it (not hide it). Using Light Channels to specify which lights are for TressFX is good practice. A light can broadcast on multiple light channels, to keep your mesh and hair (for example) in sync, but as TressFX limits the total number of lights, and even limits the number of shadow casting lights further, this can enable you to have regular lights in your scene, as well as those specific to TressFX.

Point Lights

These lights can be very expensive (performance-wise within TressFX) so should only be used when necessary, and it is recommended to only use one Point Light. This limit is simply because of the performance cost.

Spotlights

These can be the preferred (lowest cost) lights to use with TressFX. Currently, the number of allowed spotlights is limited to five shadow casting lights (5). Shadow casting lights must be Movable lights. If banding (streaks of dark or speckling dark and light areas) appear, move the light closer and attenuate the spotlight (smaller cone).

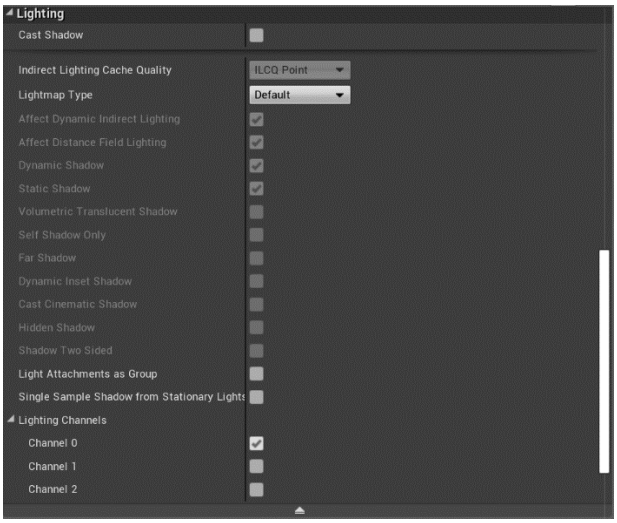
Directional Lights

These lights can be also expensive, though not as expensive as point lights. TressFX supports static, stationary and mobile (shadow casting). Note that Movable Directional Lights are automatically set to 20000.0. If banding (streaks of dark or speckling dark and light areas) appear, move the light closer and attenuate the distance.

Lighting Channels

TressFX supports lighting channels. To enable, on the light itself, under the Lighting section, set the Lighting Channels. Do the same on the TressFX Component (Lighting section, Lighting Channels). *Hint: click the Advanced Arrow (and the Up arrow at the base of the section, if not all of the items are showing).*

It is generally recommended that you put TressFX based lighting on a different channel than general lighting, since TressFX shadow casting lights require close-in handling and TressFX has a limited number of lights allowed (overall).



Boned Based Simulation and Tracking

TressFX 4 uses bones for hair simulation and tracking. This is different from earlier TressFX versions, which used the mesh. This is why, on import of Hair Assets (TFX/TFXBONE) and Collision Mesh Assets (TFXMESH) you are required to specify a skeletal mesh. The skeletal mesh should be using the same mesh and rig as the one used to export the hair/collision mesh files. You can use a different mesh, but TressFX will look for the bone names (joint names) for tracking, so the hair may not align or behave as expected if a different but similar rig is used.

For greater accuracy, TressFX animates the first two vertices to maintain the position as well as direction of the strand by assigning zero inverse masses to the first two vertices. The fourth component (w) of the vertex position is the inverse mass.
The bone-based skinning code is in the **IntegrationAndGlobalShapeConstraints** compute shader kernel. In TressFX 4 and 4.1, the demos have used 4x4 matrices for bone transforms, taking an array of 4x4 bone matrices in world space.

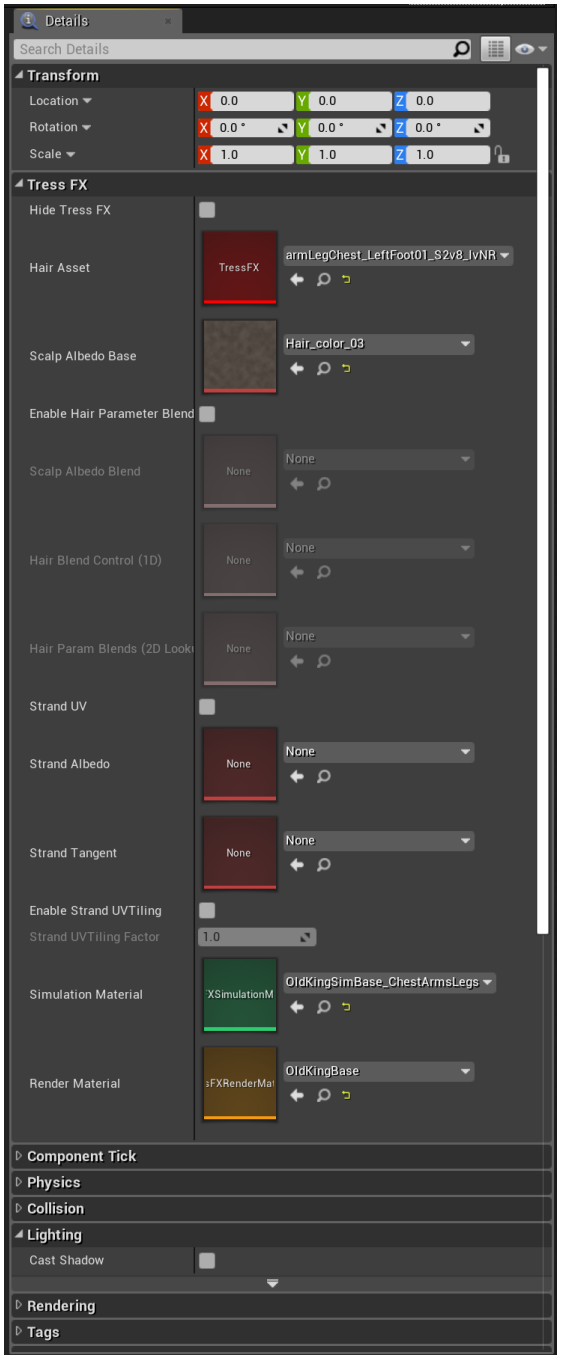
The maximum influential bones per hair strand are set to four. This value is hard-coded and should always match the .tfx file and the simulation compute shader.

TressFX currently limits the number of bones to 512. It is important to make sure that your skeletal mesh bone total is within this limit for proper bone/hair tracking. This limit can be changed (in code).

See ***TressFXCommon.h*, TRESSFX_MAX_NUM_BONES**

The TressFX Component

Summary



Basic Parameters

Hide Tress FX (checkbox)

Hides TressFX hair in the editor. Sometimes useful when working with multiple components, or if performance is slow.

Hair Asset (uasset drag/drop or dropdown selection)

The hair asset to use with this component. This is the uasset that is generated upon import of a TFX/TFXBONE file set (against a skeletal mesh). You can use the same hair asset on multiple TressFX components.

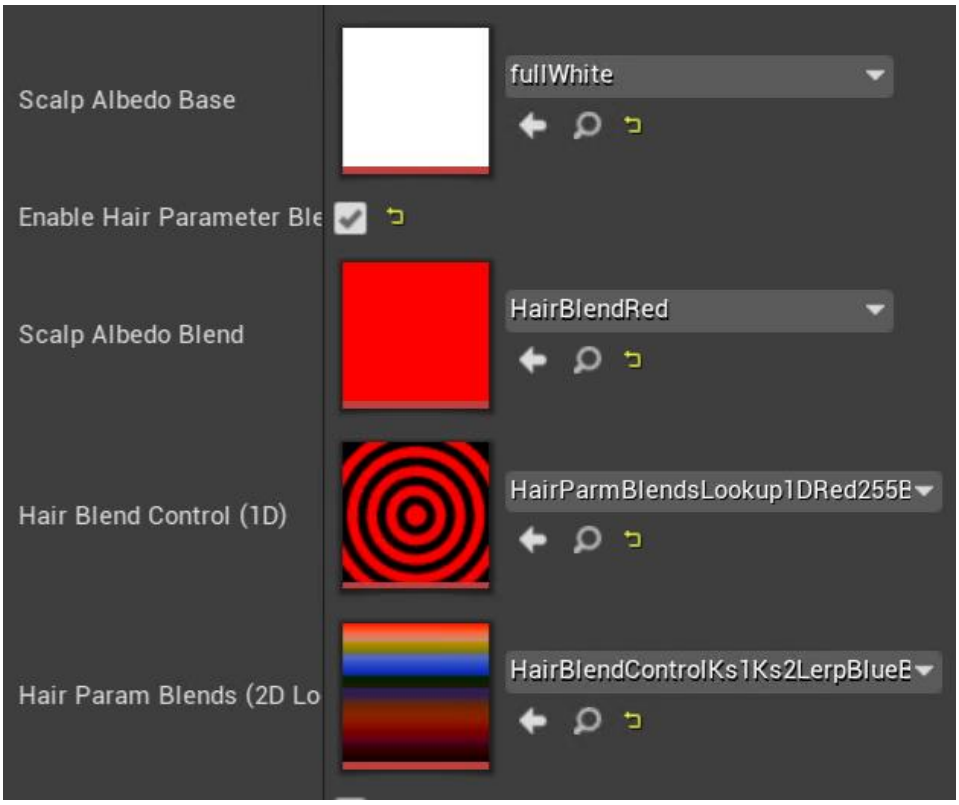
Scalp Albedo Base (uasset drag/drop or dropdown selection)

This texture is the UV mapped color texture to use for the 'scalp'. This is the base color of the hair (UV mapped). It is blended with the Base Scalp Albedo color on the Render material. And it is blended with any of the blend results (Hair Parameter Blending or Scalp UV blending). When only this albedo is used (texture or Render material color), specular values follow the Render material settings — they are not modified (modulated) by either the Hair Parameter Blending results or the Scalp UV normal map.

Hair Parameter Blending

This is a new feature for TressFX 4.1/Unreal engine integration.

See Hair Parameter Blending - More Information for examples of Hair Parameter Blending use.



Enable Hair Parameter Blending (checkbox)

Turns on usage of Blending textures with Scalp Albedo Base. All three blending textures must be present for correct results. Scalp Albedo Base must be present (not empty) to enable blending.

Note: This is one of the most complicated texture blending features of TressFX since the control textures require encoding per channel information.

Scalp Albedo Blend (uasset drag/drop or dropdown selection)

This texture holds the UV mapped texture to use for Hair Parameter blending results. How it is blended, including specular modulators, is controlled by the information contained in the channels (RGB) from the Hair Blend Control (1D texture) and the Hair Param Blends (2D lookup texture).

Hair Blend Control (1D) (uasset drag/drop or dropdown selection)

This is a UV mapped lookup using the same UV map as the Scalp Albedo Base/Scalp Albedo Blend textures, so that you can specify which 1D lookup entry (RGB) to use based on UV mesh position. Currently, only the R channel is used in this texture. The R channel contains an index into the 1D lookup texture. This specifies which 1D lookup entry (Ks1 modulator, Ks2 modulator, Lerp blend value) to use for which UV mapped position. For example if the UV mapped texture had R=0 at that position on the scalp UV map, then the first entry in the 1D lookup texture would be used for Ks1&2 modulation and the amount of lerping to do between the Scalp Albedo Base value and the Scalp Albedo Blend value (at that same UV mapped position). In this way, each strand (or set of strands) corresponding to a particular UV mapped coordinate (u,v) can have unique base and 'blend' colors, with unique specular values and lerp amount running the length of the strand.

Hair Param Blends (2D Lookup) (uasset drag/drop or dropdown selection)

This **2D texture** is used as a **1D lookup table**, where each channel's (R, G, B) 'entry' contains information to be used in the blend result. The U value from the scalp UV mapping is used to pick which set of values to use when going down the length of that strand. Think of this as a table, where the UV mapping dictates which column (at U) to use, and the column values per row (V) are used to determine the specular modulation and lerp blending to use on that section of the strand. Since the strand could be longer or shorter than the number of values in that column, the strand would be divided into sections (one per column value).

The R channel contains a modulation value for Ks1 specular gain term. The G channel contains a modulation value for the Ks2 specular gain term. The B channel contains lerp value to use (between Scalp Albedo Base and Scalp Albedo Blend). The texture itself is R8 so each channel value can range from 0 to 255 (256 possible values.) For example, a default for this texture should correspond to R=1, G=1, B=0 (multiply specular values by 1, giving no change to their current settings, and have no lerping).

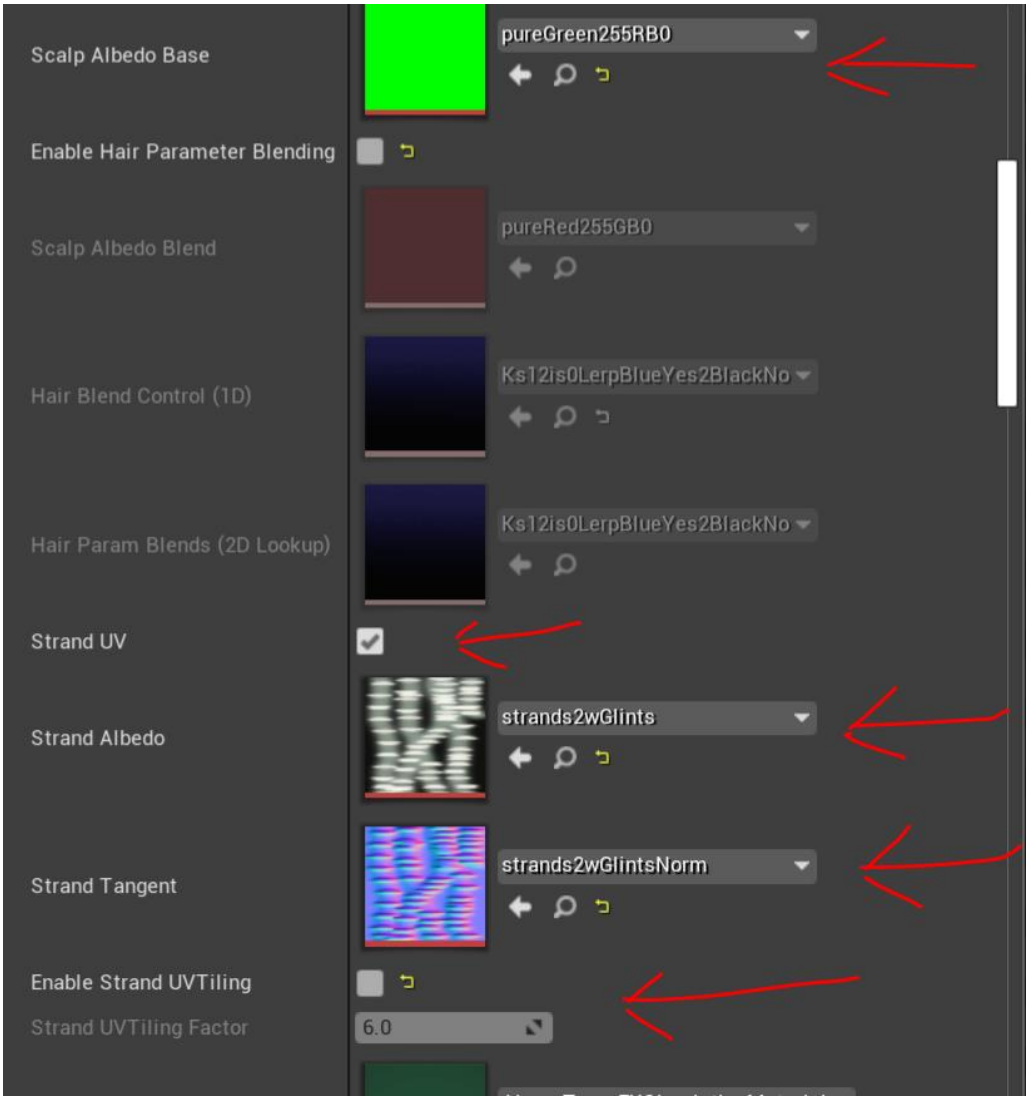
This means it can be possible to create a mapping that allows you to lerp different sections of the hair strand, for example, having both ends be the base color and the middle be the blend color. It also means you can modulate the specular gain values per section of strand, as well as having each strand have a unique mapping.

Strand UV (StrandUV)

This is a new feature for TressFX 4.1.

TressFX/Cauldron implements only the diffuse (Strand Albedo) part of StrandUV.

See StrandUV - More Information for examples of StrandUV use.



Strand UV (checkbox)

Turns on usage of Strand Albedo and Strand Tangent (if present).

Strand Albedo (uasset drag/drop or dropdown selection)

Albedo texture to use along the strand. This will be blended with other textures and/or colors already applied to the strand.

Strand Tangent (uasset drag/drop or dropdown selection)

Tangent-based normal map texture to use along the strand for normal mapping.

Enable Strand UV Tiling (checkbox)

If selected, enables tiling of the strand albedo and strand tangent textures.

Strand UVTiling Factor (numeric)

Amount of tiling to use (1D) along the strand.

Note: Using transparency on hair strands can result in unpredictable and erroneous shadow results (both hair self-shadowing and casting shadows). You should use long thin hairs or wider hair strips with non-transparent 'painted' strands as textures in most situations. This may also improve performance.

Render and Simulation Shared Materials

Simulation Material (uasset drag/drop or dropdown selection)

The TressFX simulation material to use for this TressFX component. A TressFX simulation material can be used across many TressFX components, and any change to the material would be applied to all TressFX components using that material.

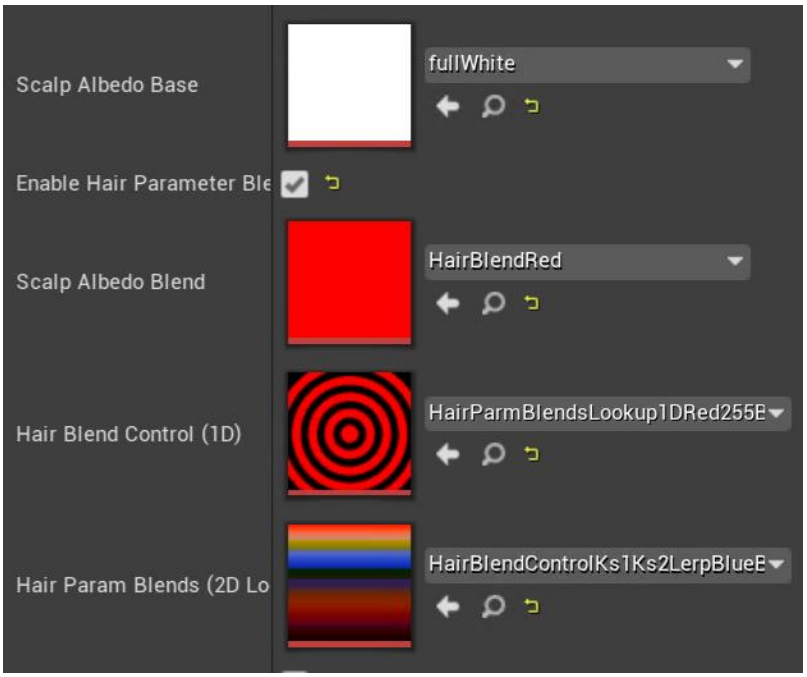
Render Material (uasset drag/drop or dropdown selection)

The TressFX render material to use for this TressFX component. A TressFX render material can be used across many TressFX components, and any change to the material would be applied to all TressFX components using that material.

Hair Parameter Blending – More Information

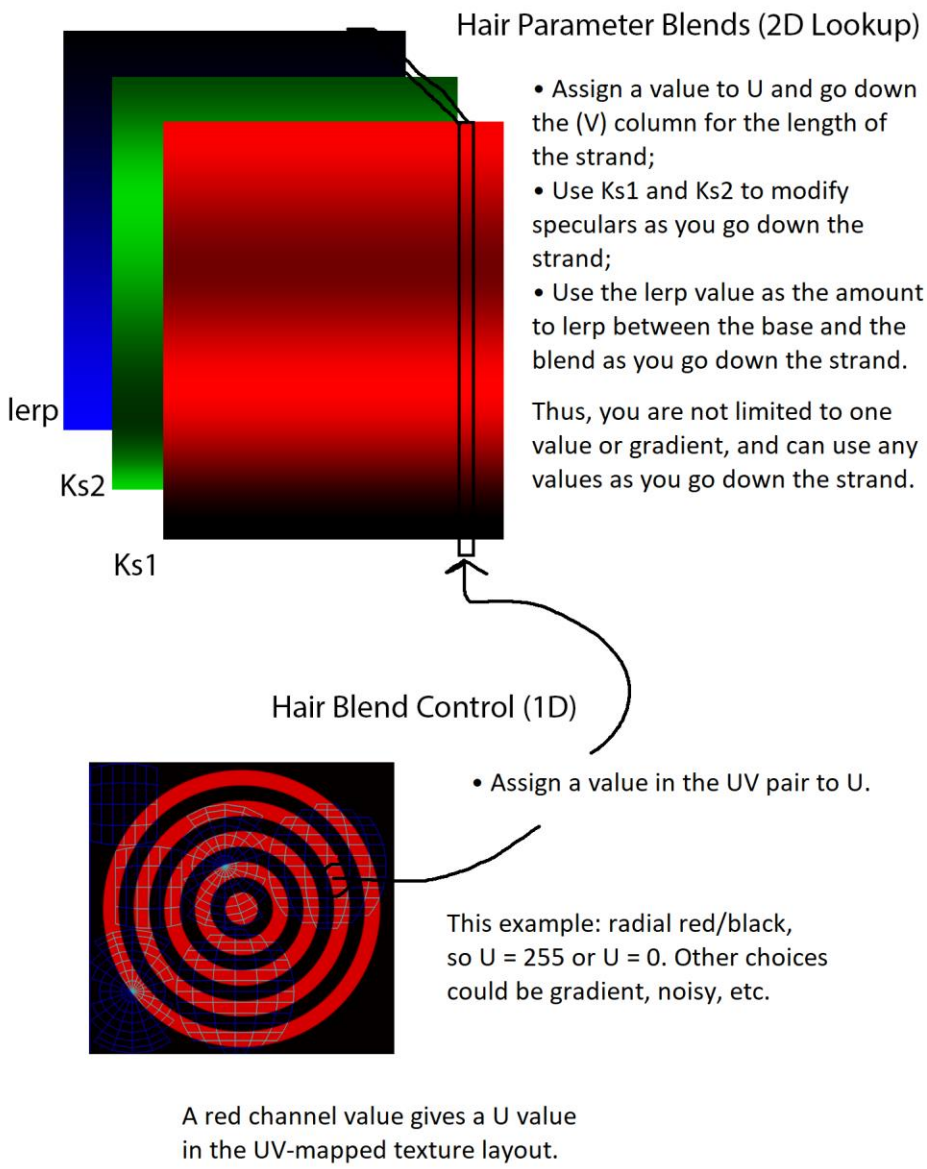
Hair Parameter Blending is a new feature of TressFX, starting with the TressFX/Unreal engine integration. Unfortunately, it is also a complicated feature but one that offers powerful control over the hair at a strand and sub-section of strand level.

Note: Currently, the Hair Parameter Blending feature is only available with the TressFX/Unreal integration.



All three blending textures must be present for correct results. Additionally, Scalp Albedo Base (texture) must be present (not empty) to enable blending. Scalp Albedo Blend (texture) holds the UV mapped texture to be used for Hair Parameter blending results. How it is blended, including specular modulators, is controlled by the information contained in the channels (RGB) from the Hair Blend Control (1D texture) and the Hair Param Blends (2D lookup texture).

;



The Hair Blend Control (1D) is a **2D texture** that is used as a **1D lookup table**, where each channel's (R, G, B) entry contains information to be used in the blend result. The U value from the scalp UV mapping is used to pick which set of values to use when going down the length of that strand. Think of this as a table, where the UV mapping dictates which column (U) to use, and the column values per row (V) are used to determine the specular modulation and lerp blending to use on that section of the strand. Since the strand could be longer or shorter than the number of values in that column, the strand would be divided into sections (one per column value).

The R channel contains a modulation value for Ks1 specular gain term. The G channel contains a modulation value for the Ks2 specular gain term. The B channel contains lerp value to use (between

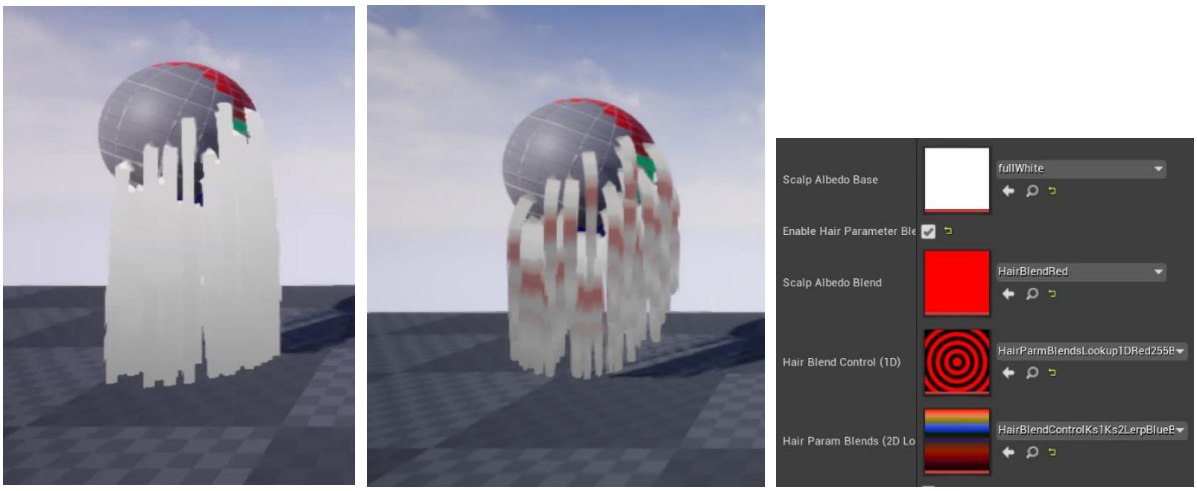
Scalp Albedo Base and Scalp Albedo Blend). The texture itself is R8 so each channel value can range from 0 to 255 (256 possible values.) For example, the default for this texture should correspond to R=1, G=1, B=0 (multiply specular values by 1, giving no change to their current settings, and have no lerping).

This means it is possible to create a mapping that can allow you to lerp different sections of the hair strand, for example, having both ends be the base color and the middle be the blend color. It also means you can modulate the specular gain values per section of the strand, as well as having each strand have a unique mapping.

Hair Param Blends (2D Lookup) is a UV mapped lookup using the same UV map as the Scalp Albedo Base and Scalp Albedo Blend textures, so that you can specify which 1D lookup entry (RGB) to use based on the UV mesh position. Currently, only the R channel is used in this texture. The R channel contains an index into the 1D lookup texture. This specifies which 1D lookup entry (Ks1 modulator, Ks2 modulator, Lerp blend value) to use for which UV mapped position.

For example, if the UV mapped texture (Hair Param Blends (2D Lookup)) had R=0 at a particular UV position on its UV map, then the first entry, entry 0, in the 1D lookup texture (Hair Blend Control (1d)) would be used for Ks1&2 modulation (R channel, G channel) and the amount of lerping (B channel) to do between the Scalp Albedo Base value and the Scalp Albedo Blend value (at that same UV mapped position). In this way, each strand (or set of strands) corresponding to a particular UV mapped coordinate (U, V) can have unique base and 'blend' colors, with unique specular values and a lerp amount running the length of the strand.

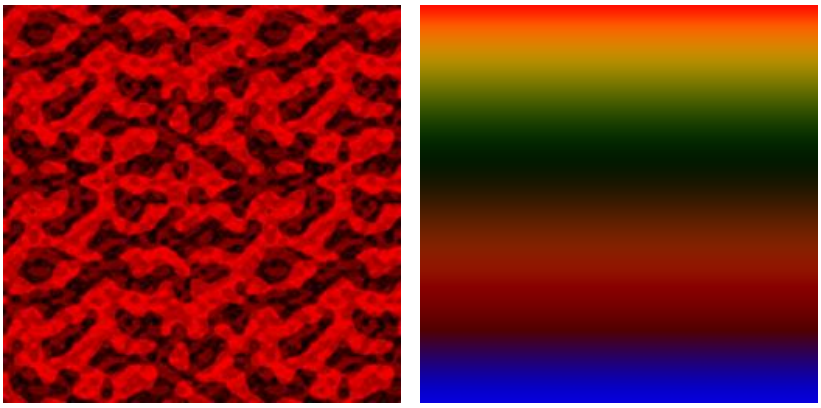
Examples:



Using the four textures: off, on, shot showing textures involved (left to right)

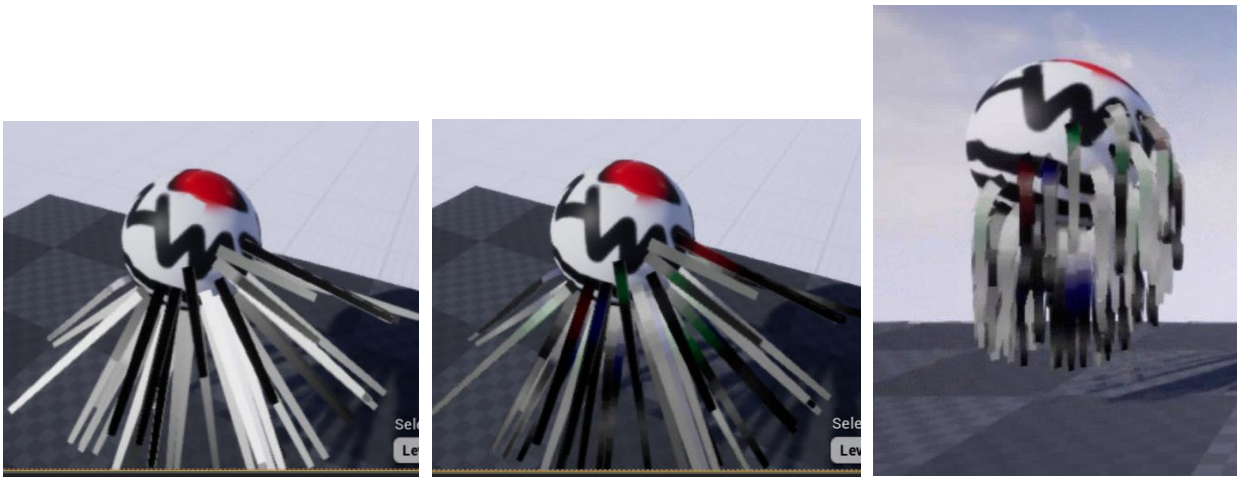
No color lerping, just gradient textures in Red (Ks1 modifier) and Green (Ks2 modifier) channels.

Blue (color lerp) is set to black. You can see that the red Scalp Albedo Blend texture is not used. But compared to the original (no Hair Parameter Blending) which does have a RenderMaterial with specular values set, the reflections on the original versus the extra patterning you get from Hair Parameter Blending can be quite different under the same lighting conditions.



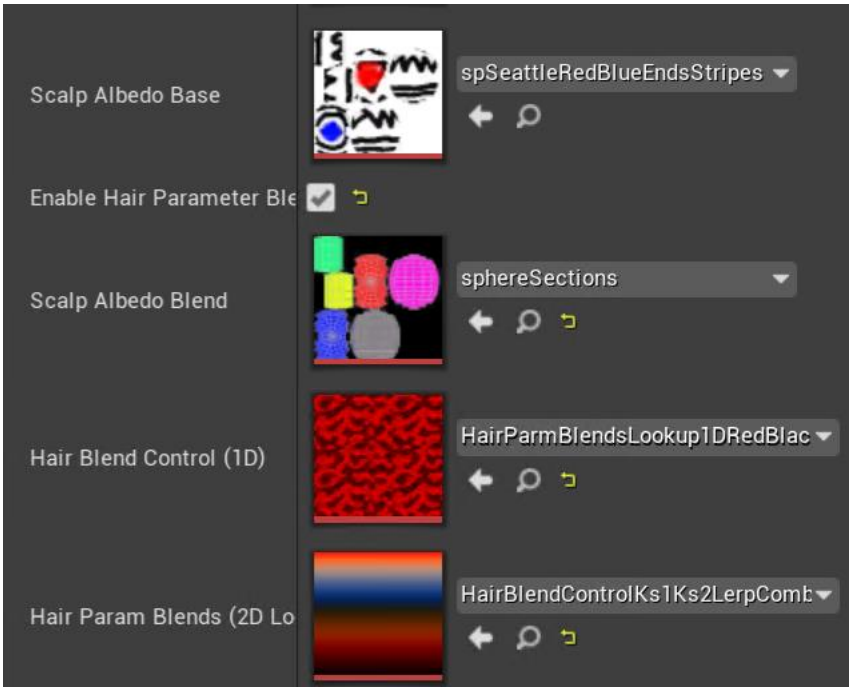
Control textures: Hair Blend Control, Hair Param Blends (left to right)

The next images show a more complicated example using two UV mappings for the sphere, one for the base and one for the blend.



All using default lighting: Hair Parameter Blending turned off/hair in Bind Pose (left), Hair Parameter Blending turned on/hair in Bind Pose (middle), Hair Parameter Blending turned on/hair allowed to simulate (right)

Note: The lighting in this scene also tends to wash out some of the strand color, especially in the normal (no Hair Parameter blending) case.



Textures used.

StrandUV – More Information

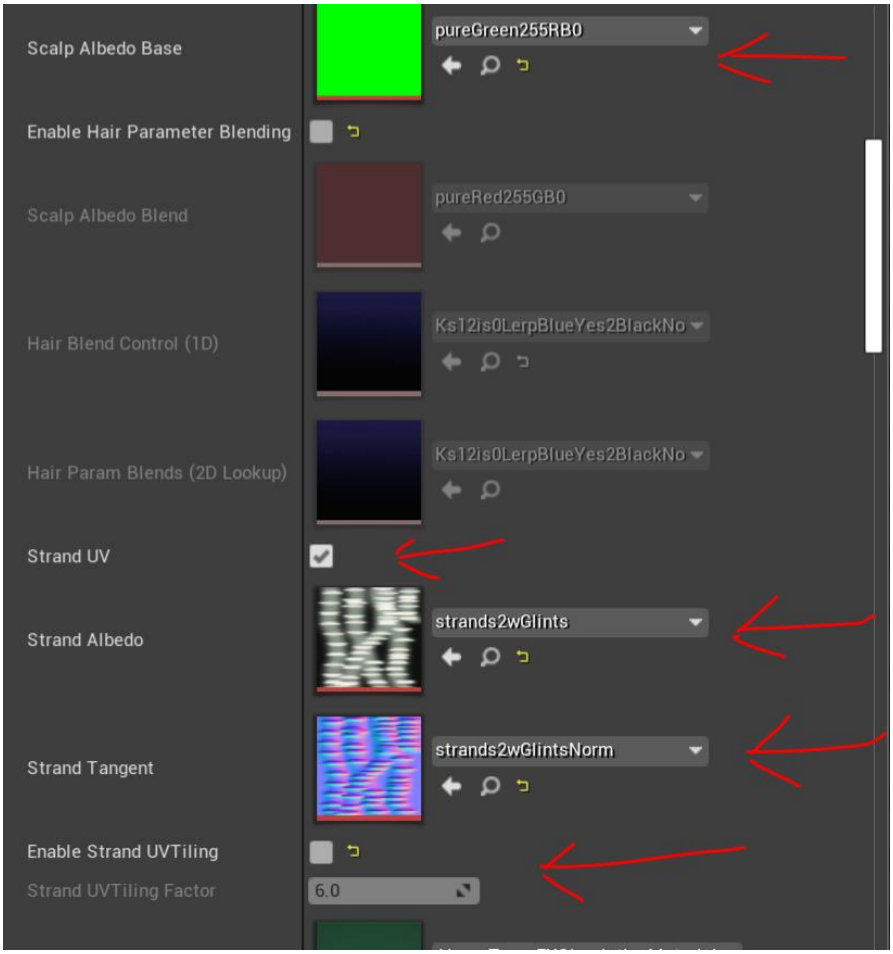
StrandUV is a new feature of TressFX 4.1, starting with the TressFX/Unreal engine integration. It allows you to apply an albedo texture and normal map along the strands, including the ability to tile those textures (along the length of the strand). Currently, StrandUV can only be applied at a macro level (all

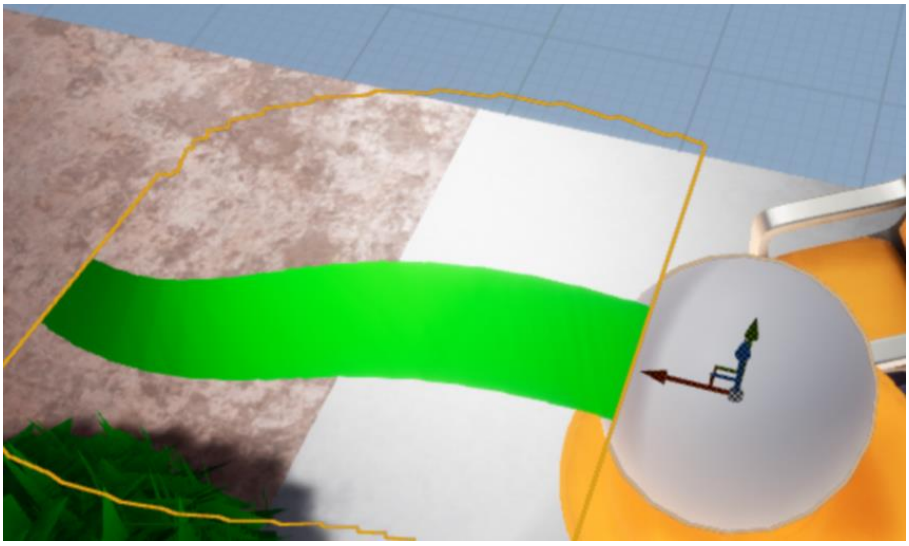
strands in a TressFXComponent) and does not support randomly offsetting the texture (only tiling). However, the textures are stretched to fit each strand in the TressFXComponent HairAsset group, so if different strands have different lengths, then some randomization will naturally occur.

Also, as with most shader parameters, for TressFX/Unreal, these values can be modified by blueprint at runtime to create other effects.

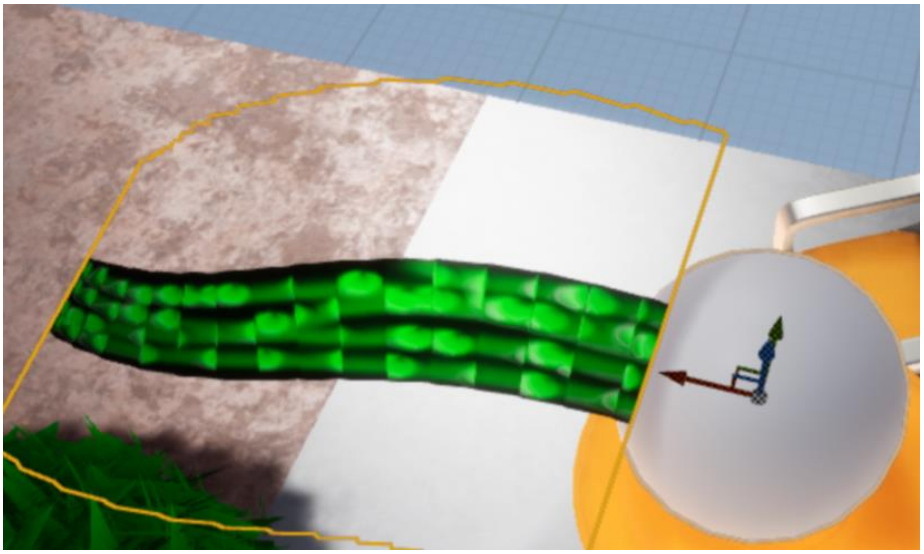
The diffuse part (Strand Albedo) of StrandUV is also implemented in TressFX/Cauldron, but **not** the tangent normal (Strand Tangent).

The following images show various uses of StrandUV.

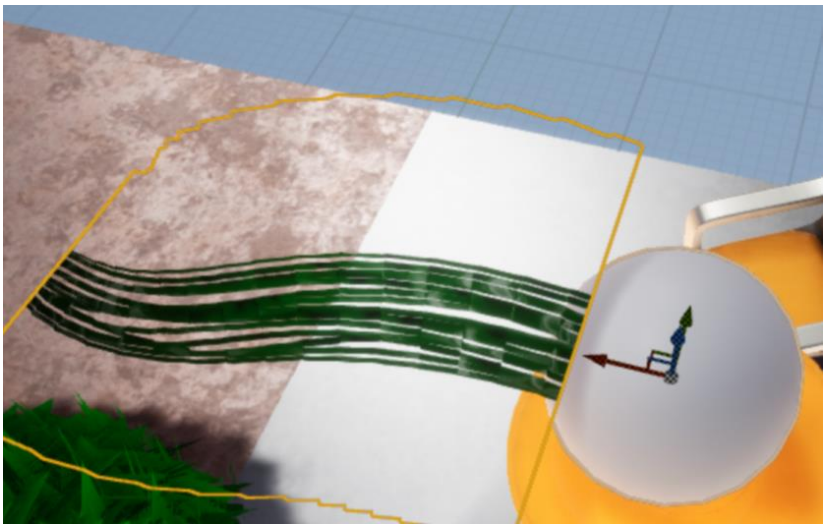




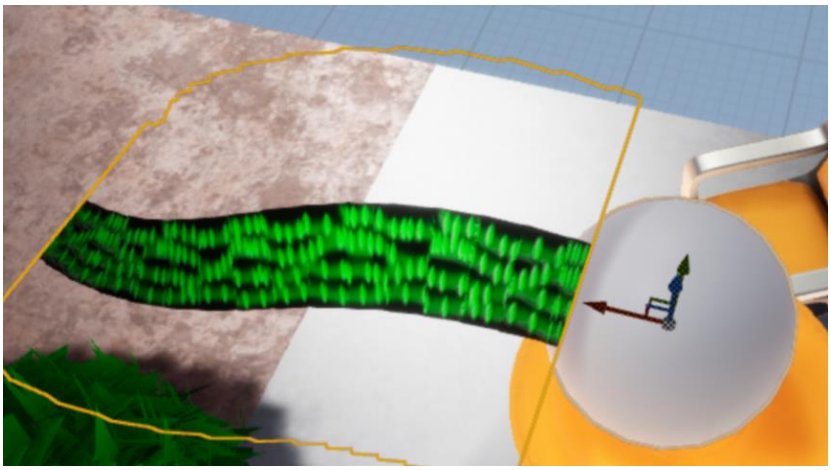
Before: no StrandUV



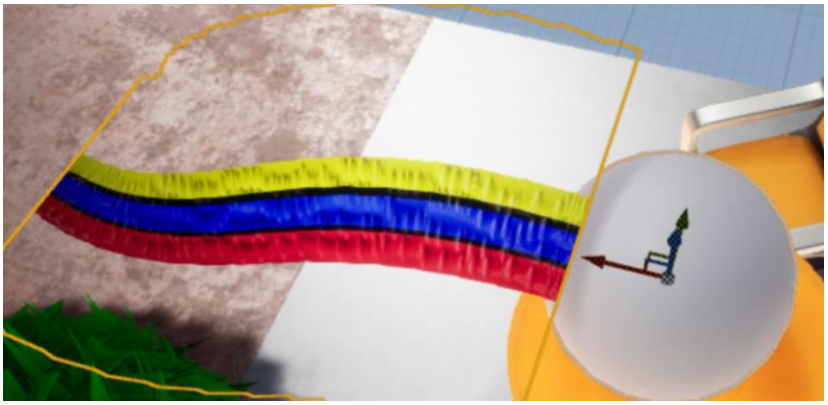
After: StrandUV turned on



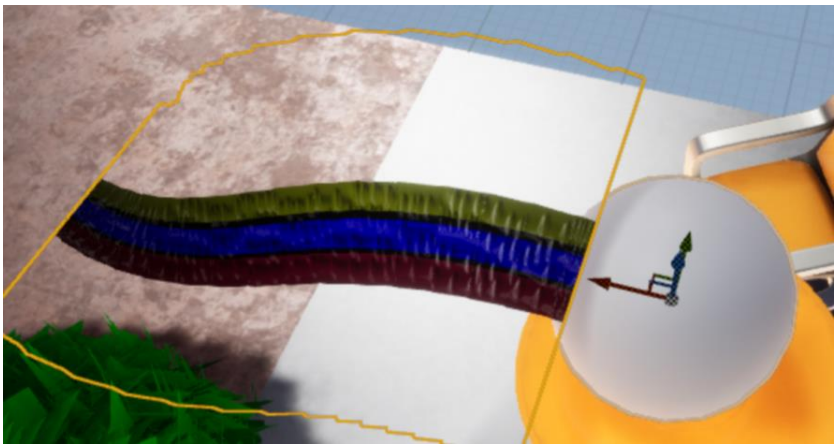
With transparency



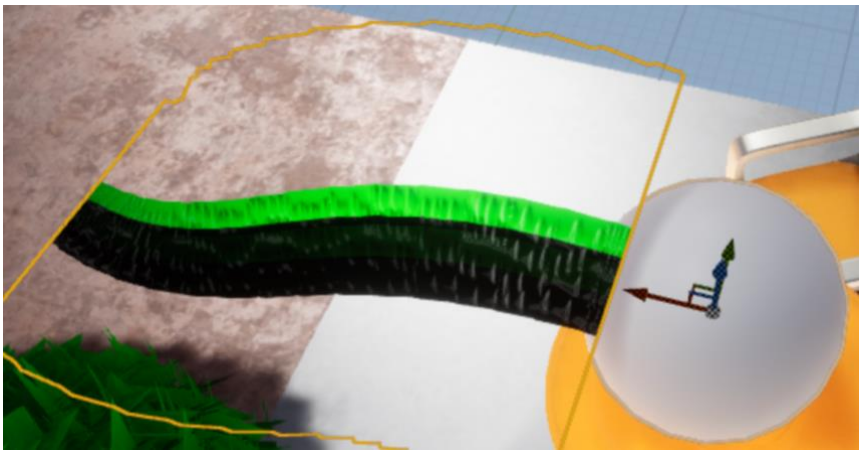
With tiling (6x)



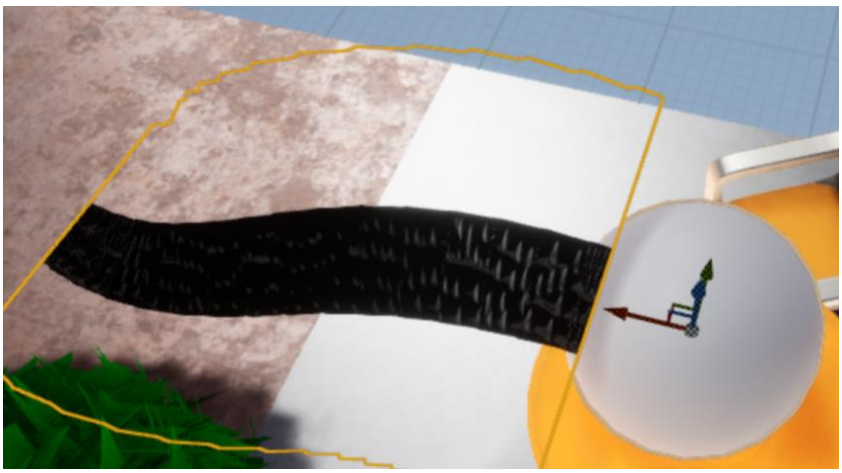
With Scalp Albedo Base off (clear) and only Strand Albedo + normal map (a different albedo texture was used)



Note that the Base Scalp Albedo Color (from Render Material) is multiplied in (color is dark blue here)



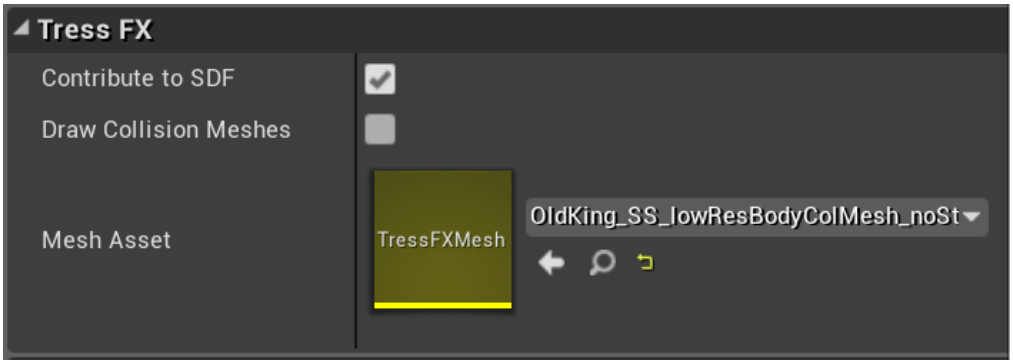
Scalp Albedo Base (texture) + Strand Albedo (texture) + normal map + Base Scalp Albedo Color (white)



Same combo but with Base Scalp Albedo Color as black

The TressFX Collision Mesh Component

Summary



The TressFX Unreal Collision Mesh Component holds a single TressFX collision mesh, which was imported using the .tfxmesh data file.

Contribute to SDF (checkbox)

Enables whether this collision mesh should contribute to the SDF.

Draw Collision Meshes (checkbox)

When collisions are enabled, draws a visualization of the collision mesh(es) being used.

Note: Visualizations of the collision meshes may have alignment issues at this time. This is a known issue. However, the collision meshes are working properly. It is only the visualization that is misaligning.

Mesh Asset (uasset drag/drop or dropdown selection)

The collision mesh to be used with this component. You can have multiple collision meshes assigned to the same hair asset (TressFXComponent) using blueprints (in TressFX/Unreal).

The TressFX SDF Component

Summary

SDFCollision Margin (float)

Value that lets you expand or contract the size of the SDF field. In essence, you can make the collision field slightly bigger than its original size (usually the same size as the render mesh (skinned mesh)) by setting the value greater than 0.0 or contract it by setting a value less than 0.0.

Num Cells in XAxis (integer)

SDF resolution. Increasing the number of cells in the SDF (x-direction) gives you greater resolution but comes at a performance cost.

Note: The marching cubes visualization is currently affected by this value, and a resolution above 50 may not be capable of visualizing the complete SDF field. This is a known issue. However, the SDF is working properly. It is only the visualization that is incomplete.

Draw Marching Cubes (checkbox)

When collisions are enabled, draw a visualization of the marching cubes (calculation) boundary.

The TressFX Render Material – Shader Parameters

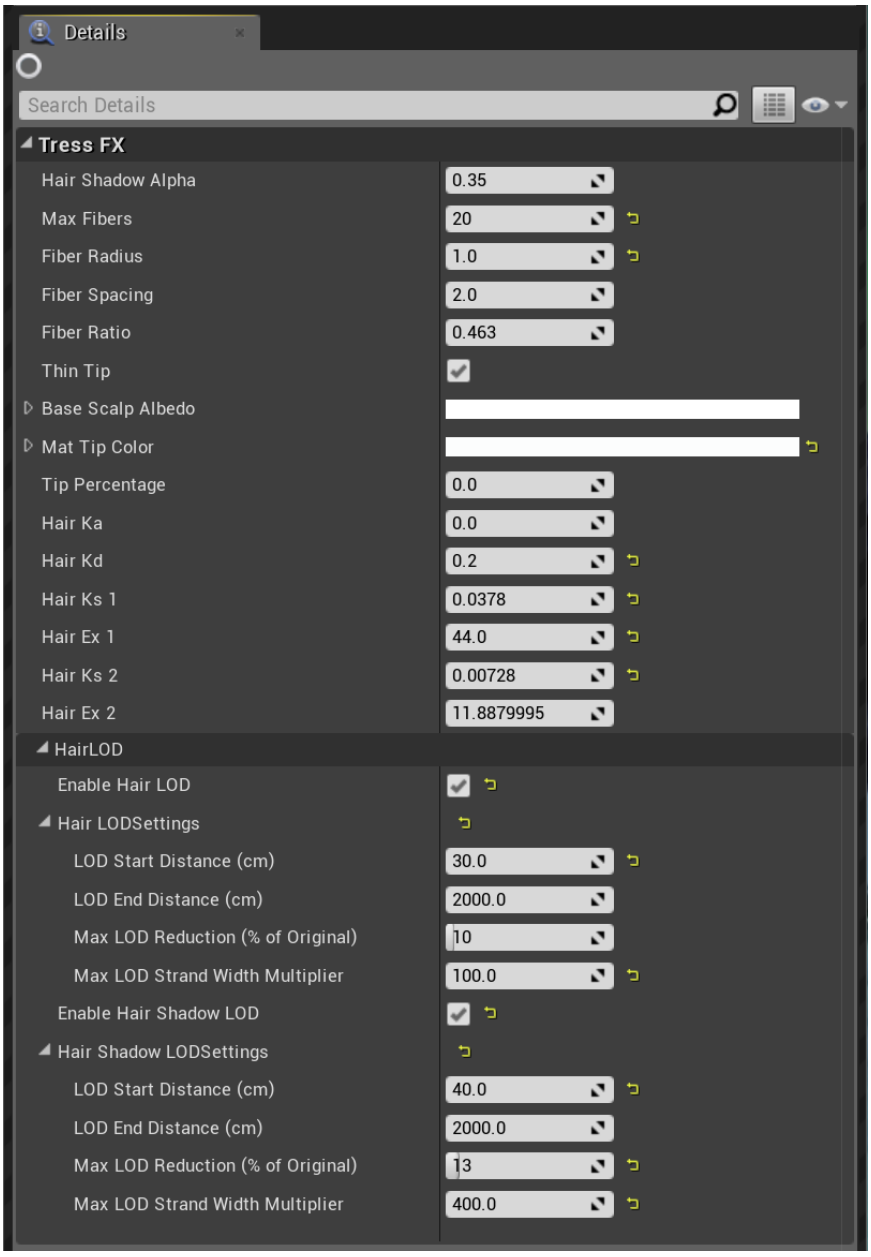
Summary

These shader parameters are included in both the TressFX/Unreal integration and the TressFX/Cauldron implementation, although the images and descriptions here are based on the TressFX/Unreal integration. Accessing these parameters in Cauldron will not be the same as with Unreal. For basic information on the Cauldron UI, see TressFX/Cauldron User Interface (UI).

TressFX 4.0/4.1 references the Kajiya-Kay lighting model for determining how light interacts with hair (Siggraph 1989, Kajiya & Kay). While the Kajiya-Kay model did not address the secondary (colored) specular highlight, this highlight was recreated using Thorsten Scheuermann's methodology (Siggraph 2004, ATI Research) where the two highlights are generated by shifting the hair tangent direction toward the root and toward the tip, respectively. As a result, both the primary and secondary specular highlights are simulated along with a diffuse component. See Marschner's 2003 Siggraph presentation (Siggraph 2003, Marschner, Jensen, Cammarano, Worley & Hanrahan) for more information on the Marschner specular model and the now common terminology used when discussing hair reflectance.

For TressFX, hair is modeled as having a tiled cone shape. The lighting equations use 10 degrees as the angle of the cone (5-10 degrees is typical), which is converted to radians in the lighting equations. To change this value, you need to modify the shader (for TressFX/Unreal branch, see ***TressFXLighting.ush***, **TressFX_ComputeKajiyaDiffuseSpecFactors**).

TressFX (section)



Hair Shadow Alpha (float)

Used to attenuate hair shadows based on distance (depth into the strands of hair). The more transparent the hair, the deeper shadows can penetrate into the strands (fibers) of that hair group. This value is not used to make the hair transparent, but rather to help determine how much attenuation of the color will occur because of shadowing.

Max Fibers (integer)

Used as a cutoff value for the shadow attenuation calculation, if the calculated number of fibers for a given depth is greater than the maximum allowed, then the amount of shadow attenuation for the

shadow casting light in question will be so small as to be negligible and no further calculations are required.

Fiber Radius (float)

Diameter of the fiber (human hair would generally be around .01 - .005). Used in the calculation to determine how much to attenuate the shadows in the hair based on the number of fibers contained within a given depth, the number of fibers is calculated by dividing the depth by (fiber spacing * fiber ratio). The more densely packed the fibers, the more likely shadows will **not** penetrate very far — although the transparency of the hair will allow greater shadow penetration.

For the exact code implementations/calculations, please see (in TressFX/Unreal branch) ***TressFXLighting.ush*** and ***TressFXRendering.ush***.

Fiber Spacing (float)

How much spacing between the fibers (should include fiber radius when setting this value). Used in the calculation to determine how much to attenuate the shadows in the hair (see Fiber Radius). This value should generally be set to the same value as Fiber Radius (hair hanging in a cluster is generally strand-to-strand with little extra space).

Fiber Ratio (float)

Used with thin tip. Sets the extent to which the hair strand will taper.

Thin Tip (checkbox)

If selected, the very end of the hair will narrow to a tip, otherwise it will stay squared.

Base Scalp Albedo (Color)

RGB color to be used for the base color of the hair. Will multiply with any textures applied (see TressFXComponent). The alpha value is not used.

Mat Tip Color (Color)

RGB color to use for a blend from root to tip. The alpha value is not used. The amount of Tip (lerp) is determined by the Tip Percentage. This is a quick way to get an overall tip tinting effect. It is not used if albedo textures are used (see TressFXComponent).

Tip Percentage (float)

Value between zero (0) (no lerping) and one (1) (entirely tip color). Dictates the amount of lerp blend between Base Scalp Albedo and Mat Tip Color.

Hair Ka (float)

Ambient coefficient, think of it as a gain value. Currently unused. Normally, this could be used as a way to handle any environmental lighting. However, it was not considered a part of the Kajiya-Kay model so was deprecated in TressFX 4. As an alternative, it is suggested to use a fill light to simulate any environmental/ambient effects.

Hair Kd (float)

Diffuse coefficient, think of it as a gain value. The Kajiya-Kay model diffuse component is proportional to the sine between the light and tangent vectors.

The diffuse component of the hair reflection model is obtained by integrating a Lambert surface model along the circumference of the half cylinder facing the light source (the back of the surface is not illuminated). This value is multiplied with the calculated diffuse value. Mathematically, the diffuse value resolves to Kd * sin(tangent, light direction), which can be rewritten as the square_root of (1 - square(dot_product(tangent direction, light direction))). Kd absorbs all the quantities that are independent of the tangent and light vectors.

Hair Ks1 (float)

Primary specular reflection coefficient (shifted toward the root). The Kajiya-Kay model primary specular can be derived in the same spirit as the Phong specular model but with modifications to approximate some diffraction around the hair. In Marschner modeling terms, this would be the gain on the primary specular, or R, value. It is the white specular reflection directly off the surface of the hair. If the light is colored, this reflection takes the color of the light like a dielectric specular term. This highlight is shifted towards the root of the hair.

Note: Despite using Marschner specular model terms to describe the reflection produced, the light equation is still using the Kajiya-Kay model as its primary theoretical base.

The reflected light forms a cone whose angle at the apex of the cone is equal to the angle of incidence. The actual highlight intensity is equal to Ks1 multiplied by cosine(eye direction, specular reflection vector in cone closest to eye vector) taken to the power Ex1. The highlight is shifted toward the root by 2 radians (cone angle radians). For a detailed look at how the mathematics that results from following the Kajiya-Kay model is converted into this implementation's resulting equation, which uses the tangent and the light direction rather than eye-related vectors, see, in TressFX/Unreal branch, ***TressFXLighting.usf***, **TressFX_ComputeKajiyaDiffuseSpecFactors()**.

Note: If a developer wanted to alter the equations to get a more noisy result on both the specular intensity and/or the amount of shift of the highlight toward the root or tip, then they could change the code. Ks1, Ks2 could be modulated with a noise texture, and the current equation for the shifts (currently hard-coded) could be modulated by use of a texture lookup in the shader code.

Hair Ex1 (float)

Specular power to use for the calculated specular root value (primary highlight that is shifted toward the root). The result of this will be multiplied with Ks1. i.e. root value taken to the power of Ex1, then multiplied with Ks1. This exponent specifies the sharpness of the highlight. The highlight is at a maximum when the eye vector is contained in the reflected cone and will fall off with a Phong model-based dependence.

Hair Ks2 (float)

Secondary specular reflection coefficient (shifted toward the tip). The Kajiya-Kay model secondary specular gain value. In Marschner model terms, this would be the gain on the secondary specular, or TRT, value. It is the colored specular. The colored specular term, aka Transmit Reflect Transmit, is caused by light passing through the hair, bouncing off the back, and reflecting back out. In doing this, this reflection picks up the resulting color. If the light is colored, it would also pick up that color. This highlight is shifted towards the tip of the hair.

Note: Despite using Marschner specular model terms to describe the reflection produced, the light equation is still using the Kajiya-Kay model as its primary theoretical base.

The reflected light forms a cone whose angle at the apex of the cone is equal to the angle of incidence. The actual highlight intensity is equal to Ks1 multiplied by cosine(eye direction, specular reflection vector in cone closest to eye vector) taken to the power Ex1. The highlight is shifted toward the tip by 3 radians (cone angle radians). For a detailed look at how the mathematics that results from following the Kajiya-Kay model is converted into this implementation's resulting equation, which uses the tangent and the light direction rather than eye-related vectors, please see, in TressFX/Unreal branch, ***TressFXLighting.usf***, **TressFX_ComputeKajiyaDiffuseSpecFactors()**.

Note: If a developer wanted to alter the equations to get a more noisy result on both the specular intensity and/or the amount of shift of the highlight toward the root or tip, then Ks1, Ks2 could be modulated with a noise texture, and the current equation for the shifts (currently hard-coded) modulated by use of a texture lookup in the shader code.

Hair Ex2(float)

Specular power to use for the calculated specular tip value (secondary highlight, colored highlight that is shifted toward the tip). The result of this will be multiplied with Ks2. i.e. tip value taken to the power of Ex2, then multiplied with Ks2. This exponent specifies the sharpness of the highlight. The highlight is at a maximum when the eye vector is contained in the reflected cone and will fall off with a Phong model-based dependence.

Hair LOD (subsection)

This is new to TressFX 4.1 and is present in both TressFX/Unreal and TressFX/Cauldron. Accessing and using these shader parameters will vary depending on which integration you are using, but the basic functionality is the same.

Note: For performance reasons, the TressFX/Unreal version includes some throttling of the LOD rate of change per frame, as well as forces the shadow LOD settings to be within the same range as the hair LOD settings. The Cauldron implementation does not do any throttling or other limitations.

Enable Hair LOD (checkbox)

Turn on Level of Detail usage for the hair. This does not include or depend on shadow LOD for the shadows cast by the hair. The two LOD sets are independent. (See note about TressFX/Unreal version differences above.)

Hair LOD Settings (group)

LOD Start Distance (cm)

Distance to begin LOD. Distance is in centimeters between the camera and hair.

LOD End Distance (cm)

Distance where LOD should be at its maximum reduction/multiplier values, in centimeters (camera to hair). The distance from start to end is used in the percent reduction and width multiplier values.

Max LOD Reduction (% of Original)

Maximum amount of reduction as a percentage of the original. For example, 5% would mean the hair would be reduced (at the maximum distance) by 95% but no more than that.

Max LOD Strand Width Multiplier

Maximum amount the strand width would be multiplied by (see Render Material - Fiber Radius).

Enable Hair Shadow LOD (checkbox)

Turn on Level of Detail usage for shadows cast by the hair. This does not include or depend on hair LOD activation. The two LOD sets are independent. (See note about TressFX/Unreal version differences above.)

Hair Shadow LOD Settings (group)

LOD Start Distance (cm)

Distance to begin LOD. Distance is in centimeters between the camera and hair.

LOD End Distance (cm)

Distance where LOD should be at its maximum reduction/multiplier values, in centimeters (camera to hair). The distance from start to end is used in the percent reduction and width multiplier values.

Max LOD Reduction (% of Original)

Maximum amount of reduction as a percentage of the original. For example, 5% would mean the shadow cast by the hair would be reduced (at the maximum distance) by 95% but no more than that.

Max LOD Strand Width Multiplier

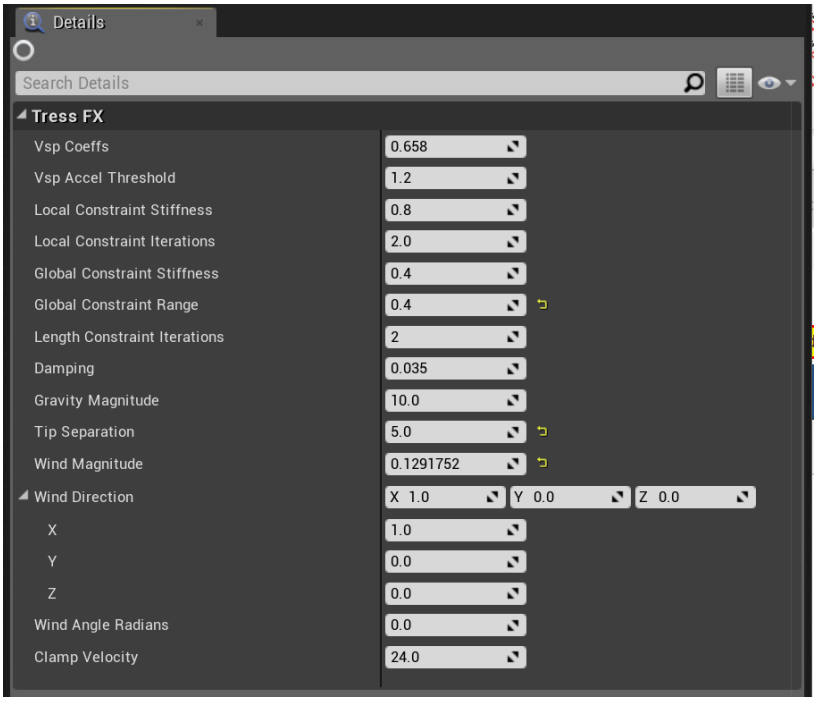
Maximum amount the shadow width cast by the strand would be multiplied by (see Render Material - Fiber Radius).

The TressFX Simulation Material – Shader Parameters

Summary

The TressFX/Unreal Simulation Material groups the physical simulation shader parameters. These parameters modify how the hair simulates, and can allow you to create more realistic effects or to help offset issues that might naturally arise with such complex interactions or performance heavy requirements.

These shader parameters are included in both the TressFX/Unreal integration and the TressFX/Cauldron implementation, although the images and descriptions here are based on the TressFX/Unreal integration. Accessing these parameters in Cauldron will not be the same as with Unreal.



TressFX (section)

Vsp Coeffs (float)

VSP (Velocity Shock Propagation) value. VSP 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 set to 0, there will be no velocity propagation. If set to 1, the full velocity of the root vertex will be passed to the rest of vertices. In the coefficient value = 1 case, hair would act as if it is rigid and the rest of simulation would not affect the result. So, in the case where you want to animate the hair or fur but do not want to simulate it, VPS would be a useful solution.

Vsp Accel Threshold (float)

VSP acceleration threshold makes the VSP value increase when the pseudo-acceleration of the root vertex is greater than the threshold value. This is particularly effective when the character makes a

sudden movement because the VSP value is set to its maximum when the pseudo-acceleration reaches this threshold.

Note: The acceleration is ***pseudo*** because it is not divided by time.

Local Constraint Stiffness (float)

Controls the stiffness of a strand, meaning both global and local stiffness are used to keep the original (imported) hair shape. If the hair was straight, stiffness values try to keep the hair straight, if curly, then curliness is attempted to be maintained.

Local stiffness only looks at the curve shape locally, rather than caring about the original world space position. If the hair is straight, for example, and the hair collides with a sphere, the hair will move out of the way of the sphere, but will try to maintain its straightness as it does so, instead of curving around the contour of the sphere.

Local Constraint Iterations (integer)

Allocates more simulation time (iterations) toward keeping the local hair shape. It can lead to better results, even make the hair seem stiffer, but it comes at a performance cost since it requires more simulation time, and could even lead to the hair moving slower/less realistically.

Global Constraint Stiffness (float)

Controls the stiffness of a strand, meaning both global and local stiffness are used to keep the original (imported) hair shape. If the hair was straight, stiffness values try to keep the hair straight, if curly, then curliness is attempted to be maintained.

Global stiffness tries to maintain the local shape (curly or straight, for example) as well as the original world space position (how the hair draped over the body). For example, if you have straight hair and it collides with a sphere (in the middle area of the strand), the tip of the strand will follow the contour of the sphere, hugging it, as it tries to move back to its original 'bind' position. Global stiffness helps the hair naturally flow over the body, adapting the curvature to the form it is colliding with.

Global Constraint Range (float)

This is a range value. It controls how much of the hair strand is affected by the global shape stiffness requirement. It is not a gradient but rather a cut-off value. If the value is set to 1, then the whole hair strand will be affected by the global constraint stiffness value (for example, the tip of the strand would move around the sphere to try to return to its original position). If the value is set to 0.5, then only the first half of the strand (root to the halfway point) will be affected and try to return to its original position, letting the other half move freely under the simulation.

Note: Global and local stiffness are useful but have obvious limitations. First, global is a cut-off value rather than a gradient of effect, making it more like very stiff 'hairspray' that does not let the hair move naturally in any way if the stiffness is set at a high value. But not using these values, or setting them too low results in loss of hair form (curl). These are known limitations with TressFX 4.x and are considered fast workarounds to more robust or realistic hair movement simulation and cohesion.

Length Constraint Iterations (integer)

Allocates more simulation time (iterations) toward keeping the global hair shape. It can lead to better results, even make the hair seem stiffer, but it comes at a performance cost since it requires more simulation time, and could even lead to the hair moving slower/less realistically.

Damping (float)

Damping smooths out the motion of the hair. It also slows down the hair movement. For example, you could increase the damping value and make the hair seem more like it was underwater.

Gravity Magnitude (float)

Gravity pseudo value. A value of 10 closely approximates regular gravity in Unreal Engine and Cauldron (despite the unit difference).

Tip Separation (float)

Forces the tips of the strands away from each other. This is the same as the hair import setting for tip, but here you can control the tip separation at runtime.

Wind Magnitude (float)

Wind multiplier value. It allows you to see the effect of wind on the hair.

Note: This is just a multiplier, using the wind direction vector, on the hair position. Unless you change it over time, it will just look like a static value. To see a naturalistic wind effect, it is recommended that you create a blueprint (in TressFX/Unreal) that varies the amount of wind cyclically. For TressFX/Cauldron you would need to change the wind magnitude (and possibly wind direction) writing code that interfaces with the shader parameters.

Wind Direction (vector3)

xyz-vector (world space) for the wind direction.

Wind Angle Radians (float)

Wind angle in radians.

Clamp Velocity (float)

Position Delta Clamp. New for TressFX 4.1. To increase stability at low or unstable framerates, this parameter limits the displacement of hair segments per frame. Decreasing the value (from the default of 24) will force the hair back to its starting pose faster (fewer cycles). This does not affect the other values, per se, such as local or global stiffness, but it does help keep the simulation in check.

Hair Asset and Collision Mesh Import Settings

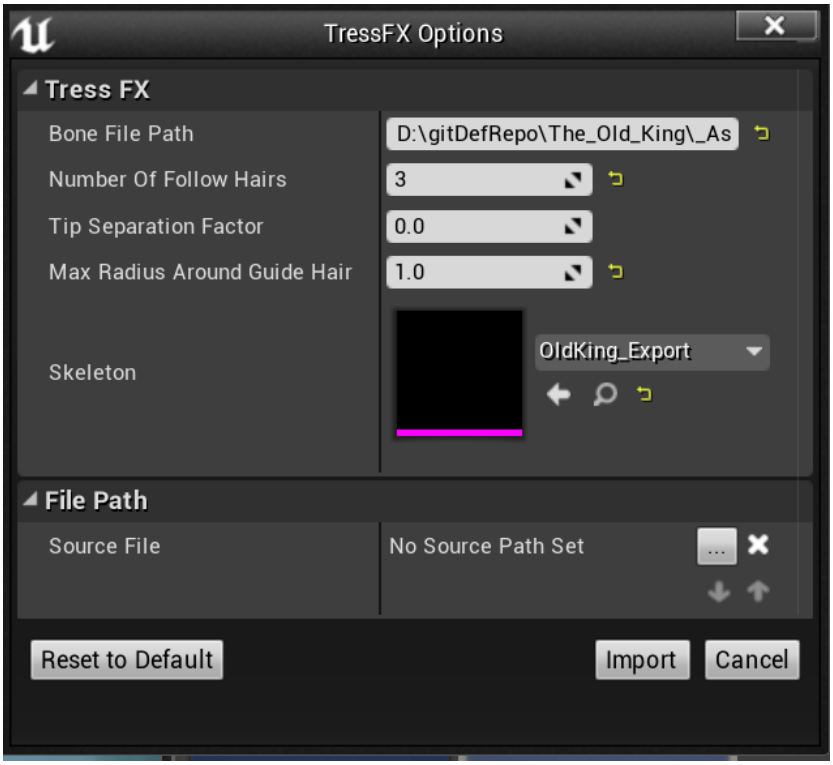
Summary

Hair and collision meshes that have been exported into TFX files (.tfx, .tfxbone, .tfxmesh) using the TressFX Exporter need to be imported into the engine/framework and that data turned into a form ready for runtime usage. This process involves importing the data, processing it into guide hairs with all the necessary buffers, bone information and parameters, and follow hair generation, where follow hairs are created and assigned to their 'guide' hair. This process will vary depending on the engine/framework being used. See the architectural information on TressFX/Unreal and TressFX/Cauldron at the beginning of this document for more information.

Hair Asset Import Settings

To start, in TressFX/Unreal, drag a .tfx (TressFX Hair) file into the Content Browser. This opens the following dialog box. You do not need to worry about the .tfxbone (TressFX Bone) file unless it has a different name than the .tfx file (save for file extension).

Note: This process will be different for TressFX/Cauldron.



Bone File Path (textbox)

If the filename of the TressFX bone file is different than that of the TressFX hair file (save for extension, of course), then you need to provide a full path to the actual file to use. The dialog will automatically populate this textbox with a path to the same directory as the .tfx file to an (assumed) .tfxbone file of the same name as the .tfx file. You only need to update this field if your bone file name and/or location is different than that of the .tfx file.

Number of Follow Hairs (integer)

This value will tell the import system the number of follow hairs you want the system to generate for each strand in the .tfx file. The hair data in the .tfx file will be pure guide hair, i.e. fully simulated hair. Follow hairs 'follow' their guide hair and therefore require less simulation processing, but do still increase performance costs overall.

Note: In TressFX 4.0/4.1, follow hair is generated at creation time (import time) and cannot be changed at runtime. If you want more follow hair, you need to re-import the TFX files and set a new number of follow hairs.

Tip Separation Factor (float)

If set greater than 0.0, the tips of follow hair are forced away from their guide hair, creating a small amount of separation between hairs. If you want hair that clumps together, like spiky hair, then you would have a large radius but 0.0 tip separation.

Note: You do not need to set this value on import. You can also change tip separation at runtime via the Simulation shader parameter (see Simulation Material - Tip Separation).

Max Radius Around Guide Hair (float)

When generating follow hair, TressFX will use a spherical radius around the guide hair root position. This value controls the radius size.

Note: This is a known issue. TressFX 4.0/4.1 follow hair generation uses a 3D sphere, rather than creating follow hairs that track to the surface of the skinned mesh. It is generally recommended to use a radius less than 1.0 to minimize the chance of hair that is obviously floating above or off the edge of the skin.

Skeleton (uasset drag/drop or dropdown selection)

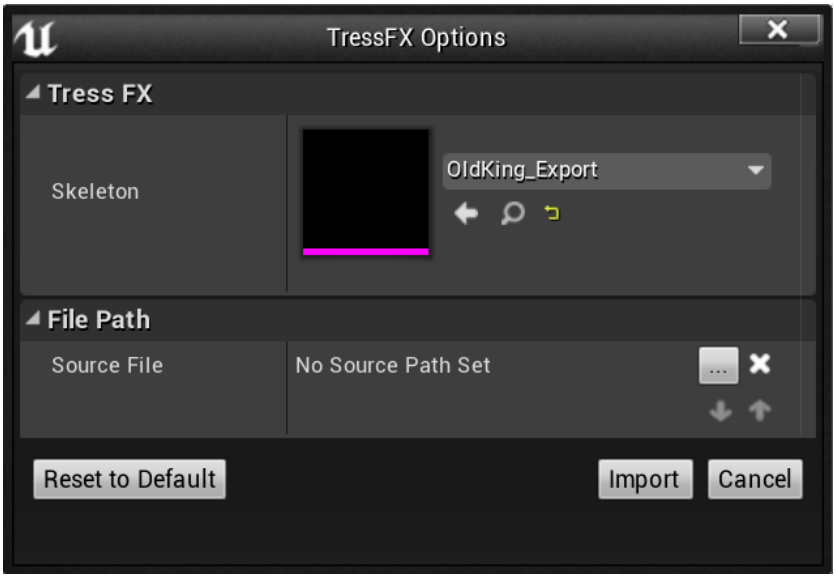
Select the skinned mesh (skeleton) that was used to create the hair. Usually, you have already loaded the skinned mesh (via FBX or other mechanism) into the engine/framework. TressFX uses the named joints in this assigned skeleton to match up with the information that was used during hair creation (export).

Note: TressFX assumes all joints have a unique name.

Collision Mesh Import Settings

To start, in TressFX/Unreal, drag a .tfxmesh (TressFX Collision Mesh) file into the Content Browser. This opens the following dialog box.

Note: This process will be different for TressFX/Cauldron.



Skeleton (uasset drag/drop or dropdown selection)

Select the skinned mesh (skeleton) that was used to create the hair. Usually, you have already loaded the skinned mesh (via FBX or other mechanism) into the engine/framework. TressFX uses the named joints in this assigned skeleton to match up with the information that was used during hair creation (export).

Note: TressFX assumes all joints have unique names.

Art Export: Exporting TFXxxx files (TressFX Exporter plugin for Autodesk Maya)

Summary

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 (NURBS curves) at the end. For demos using TressFX 4.0/4.1, the TressFX team has used Shave and a Haircut for Maya (4.0/Ratboy), Autodesk Maya XGen (4.0 and 4.1 demos) and, in the past, the Autodesk 3ds Max native hair modeler, although the 3ds Max plugin is no longer being developed and has not been included. However, the team did experiment with the use of SideFX Houdini to create the splines, then importing and exporting those splines using the Maya-based exporter plugin. Shave and a Haircut has now been acquired by Epic Games.

The Maya-based TressFX Exporter was tested on Maya 2015 and Maya 2017. Maya 2019 was not tested since at that time, Maya 2019 had a critical bug with the XGen Guides menu, a needed element during XGen-based hair creation. Maya 2017 is recommended and was the version used to create TressFX 4.1 art assets.

For a tutorial that walks you through creating and exporting hair using the Maya TressFX Exporter, see this tutorial: Creating and Exporting TressFX 4 Hair from Maya.

Installation of the Maya TressFX Plugin

The Maya-based TressFX exporter is a single python file residing in the following locations:

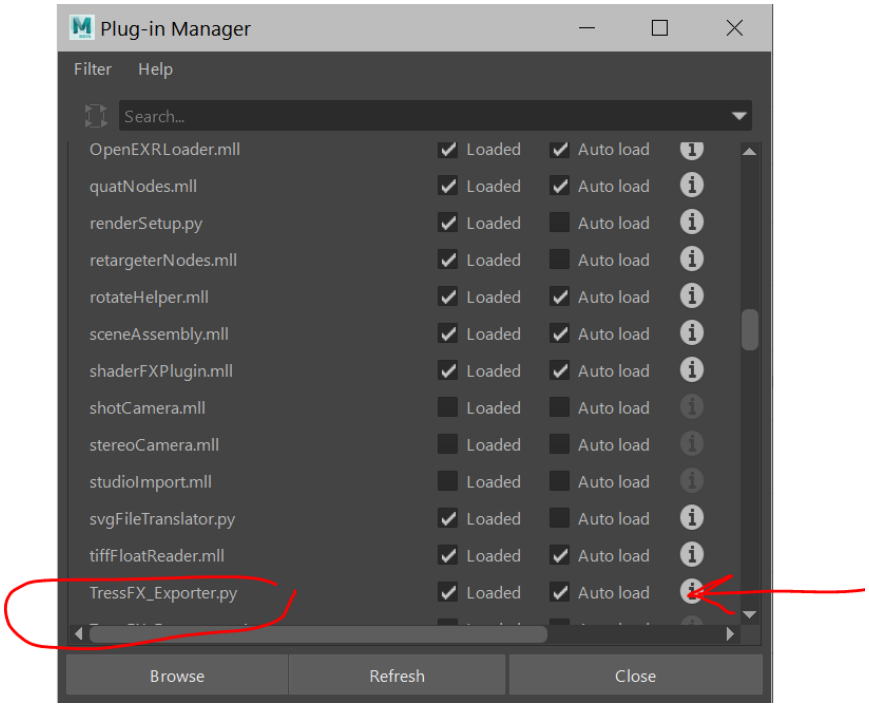
- TressFX/Unreal: **AMD_Tools_Docs\TressFX_Exporter\TressFX_Exporter.py**
- TressFX/Cauldron: **maya-plugin/TressFX_Exporter.py**

To enable the plugin, follow these steps:

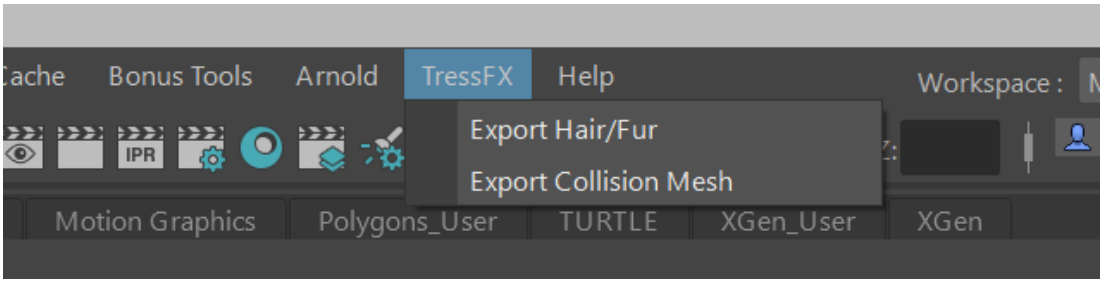
1. Copy TressFX_Exporter.py into Maya's plug-ins folder such as **C:\Users\USER_NAME\Documents\maya\plug-ins** or the main plug-in folder, such as **C:\Program Files\Autodesk\Maya2018\bin\plug-ins**

2. Launch Maya.

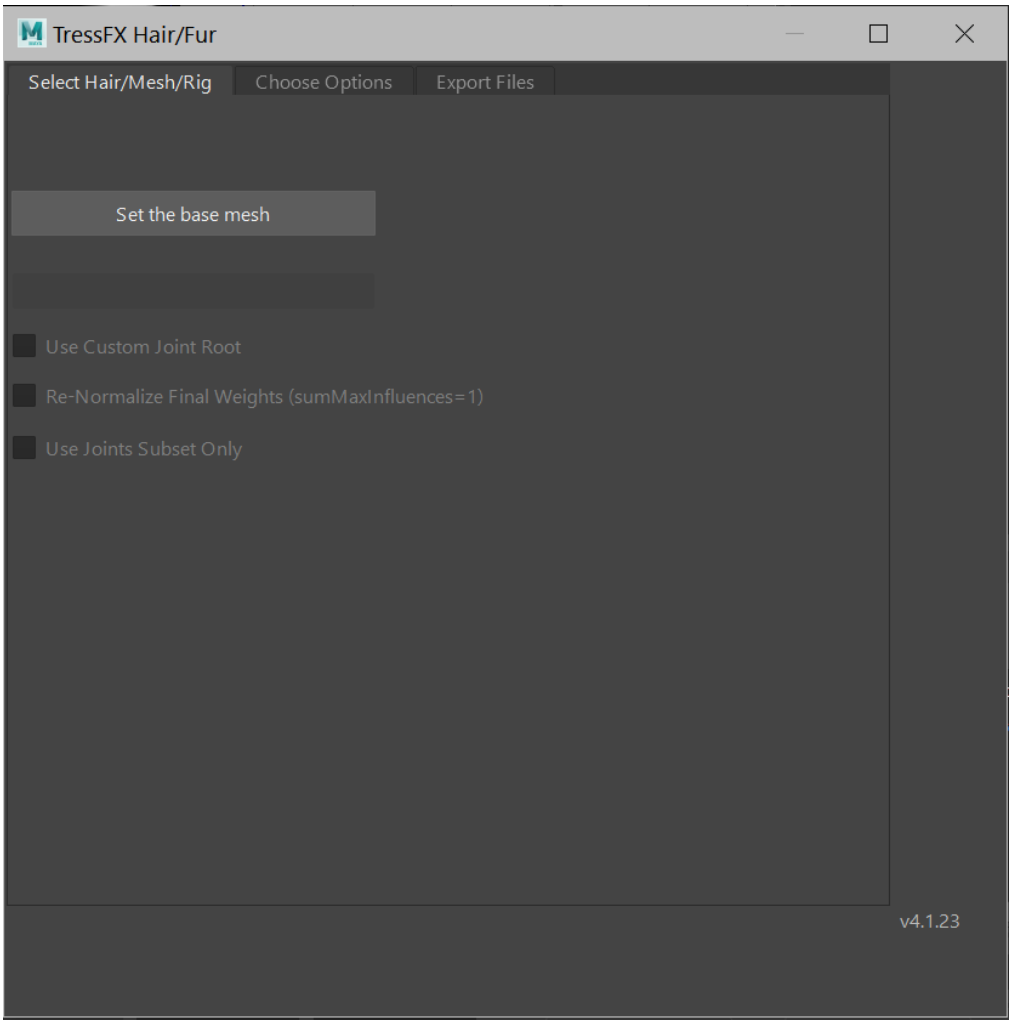
3. Open Plug-in Manager and enable the Loaded and Auto load options for TressFX_Exporter.py.



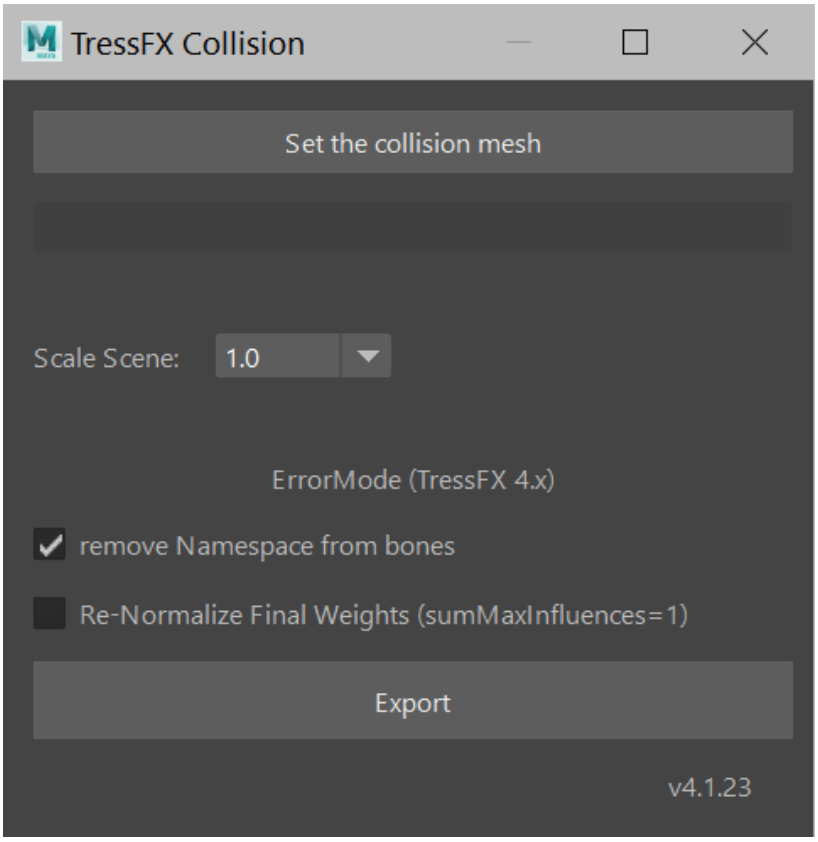
4. Now, a **TressFX** menu should appear on the main menu bar. Underneath it, there should two sub menu items: **Export Hair/Fur** and **Export Collision Mesh**.



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


TressFX Hair/Fur
Select Hair/Mesh/Rig
Choose Options
Export Files
Set the base mesh
Use Custom Joint Root
Re-Normalize Final Weights (sumMaxInfluences=1)
Use Joints Subset Only
v4.1.23

6. Export Collision Mesh will open the **TressFX Collision** window.



Exporter Settings

Since TressFX 4.0, the UI has changed, especially the new tabbed format for hair/fur export. Also, the exporter includes more error checking and new control features.

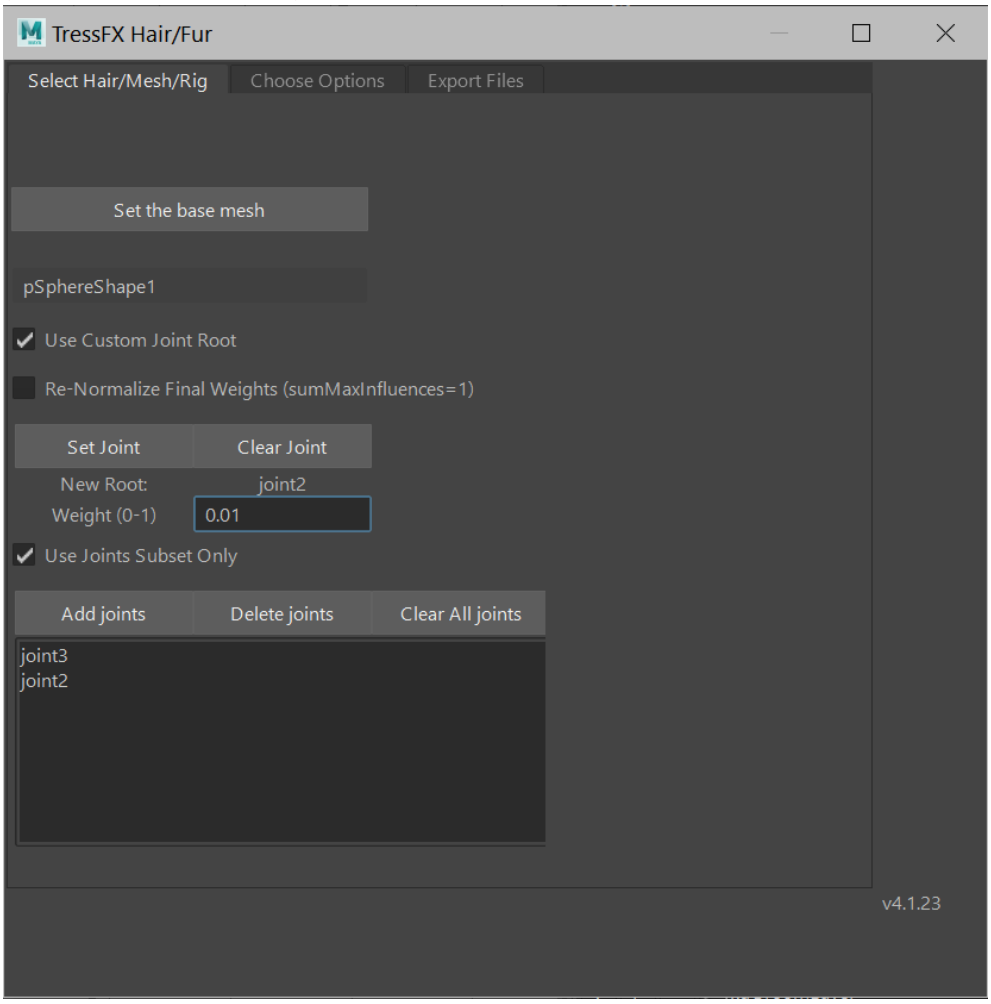
Warnings are displayed in the status bar as well as the script window. Print messages (informational) are also displayed in the script window.

Many settings will not work unless a base mesh is selected. Export will also not work unless a set of splines is selected for export. You typically do not need to select all the splines individually (i.e. Select Hierarchy in Maya) – the exporter will do a recursive search during export, so selecting a grouped set of splines (a group) is simply a matter of selecting the containing group. Tools like XGen, when converting XGen groomable hairs into splines, will automatically 'group' the selection.

See the tutorial for a walk-through on this process. A basic knowledge of Maya is assumed, including how to use the Outliner window and find the Plug-in Manager menu item.

Hair Settings

(SELECT HAIR/MESH/RIG) TAB GROUP 1:



Set the base mesh (button)

Select a mesh, then click the button to set the reference mesh (shape) that will be used for .tfx and .tfxbone (hair data) export. The selected mesh name will appear in the text box below the button.

Use Custom Joint Root (checkbox)

If selected, you can choose a different joint in the base mesh's skeleton to use as the default root joint. You can also set the value for this weight. This joint and its weight will only be used if there is no other joint found for a given point, filling out the 4 joint influencer list for each position. This is useful in the case where a hair strand is not tracking, typically because the joints nearest the root position of the strand are zero (or because of barycentric calculations sending the weight to zero). Often many joints in a complex skeleton are not used for the body animation (such as facial bones).

Note: This custom root joint also needs to be **added** to the Joint Subset ***if the Joint Subset is being used***. It is not added automatically at this time.

Set Joint (button)

Select a joint in the Outliner, then click this button to set that joint as the default root.

Clear Joint (button)

Clear the default root back to the default (joint 0 in the hierarchy).

Weight (0-1) (float)

Set the default weight for the chosen root. If you also check **Re-Normalize Final Weights**, then all four weights will be normalized so that they sum to one (1).

Re-Normalize Final Weights (sumMaxInfluences = 1) (checkbox)

Forces the four bone weights per strand vertex to sum to one. Divides each weight by the sum of the four weights. Often, this is not needed, but can be used to ensure normalization.

Use Joints Subset Only (checkbox)

Use only a selected subset of the joints attached to the base mesh. This is useful for hair tracking issues where a nearby joint is influencing the overall weighting but shouldn't. For example, you might want to include only the arms and legs for a particular hair asset group (arm and leg hair as one hair asset) and exclude other nearby bones (like facial bones or the chest or even a belt/clothing items) or bones that should only be used for non-skin movement (like a weapon).

Add joints (button)

Select a joint in the Outliner, then click this button to add that joint to the subset to be used.

Note: If you are also using a custom root, be sure to add that joint to the subset as well (currently, it does not happen automatically).

Delete joints (button)

Select a joint (or joints) from the joint list, then click this button to remove them from the list.

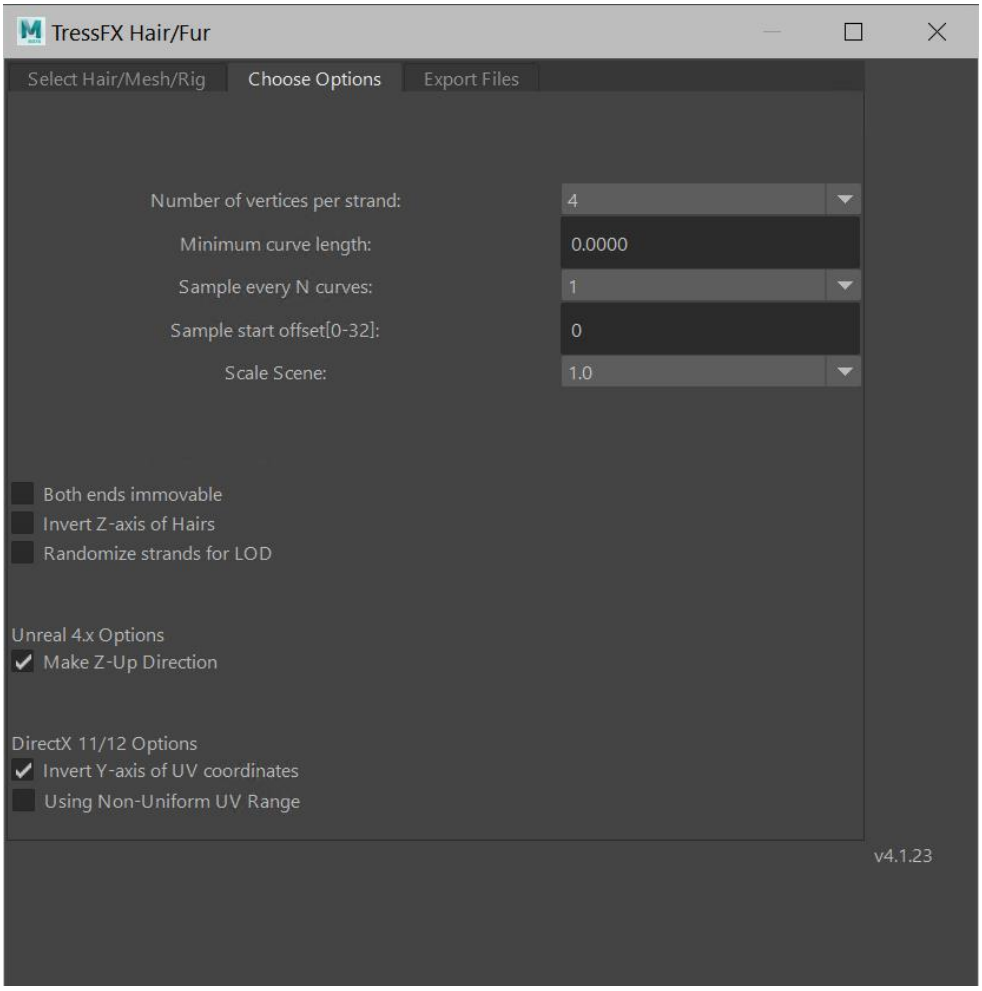
Clear All joints (button)

Click this button to clear all the joints from the joint list.

Joint list area (multi-selection listbox)

This is a multi-selection listbox (single column currently) that shows the current joints in the joint subset to be used.

(CHOOSE OPTIONS) TAB GROUP 2



Number of vertices per strand (dropdown)

The number of vertices to use to define the curve shape of a strand, the larger the value, the more definition the strand with have but at a greater hair asset size and processing (performance) cost. Generally, shorter hair looks better at 8 vertices per strand and long hair at 32, although in case of the Old King demo, the dreadlocks used 64 vertices per strand to keep a smoother curly look.

Minimum curve length (float)

The exporter uses this value to filter out hair that is shorter than this length. In some situations, it may be difficult to get rid of short hair within the modeling tool. If the value is set to 0.0 (default), no filtering will take place.

Sample every N curves (dropdown)

The exporter will only export every Nth strand. This can be useful when you have a lot of hair (splines) modeled, often more than really needed or wanted for good performance, and just a need a quick way to decimate the number of strands exported.

Note: all hair exported using the Exporter will be guide hair on import to TressFX/Unreal and (currently) to TressFX/Cauldron. Follow hairs are generated on import into those engines, not here (at this time).

Sample start offset[0-32] (integer)

Used in conjunction with subsampling (**Sample every N curves**). Lets you specify where to start the subsampling process. The default is zero (0), i.e. the first strand.

Scale Scene (dropdown)

Scales the points by multiplying the scale value against each .x, .y, .z position value. A value of 1.0 is fine for Unreal (cm), but Cauldron uses meters, so needs a scaling value of .01 to scale the data from cm->m.

Both ends immovable (checkbox)

Sets both ends of the strand (the endpoint vertices) to use zero inverse mass, which is kept in the (w) position coordinate (.w). This forces both ends, not just the 'root', to be immobile – although both will still track with the mesh skin itself. One example would be a loop on a mesh, where the middle of the loop can move freely but both ends are firmly fixed to the mesh itself.

Invert Z-axis of Hairs (checkbox)

Inverts the Z component of the hair vertices. This may be useful if dealing with an engine that uses a left-handed coordinate system.

Randomize strands for LOD (checkbox)

Randomizes hair strand indices so that any LOD done on strands in-engine would uniformly reduce hair. Not generally needed if using a hair creation tool like XGen, since it can randomize as it distributes groomable hair.

Make Z-Up Direction (checkbox)

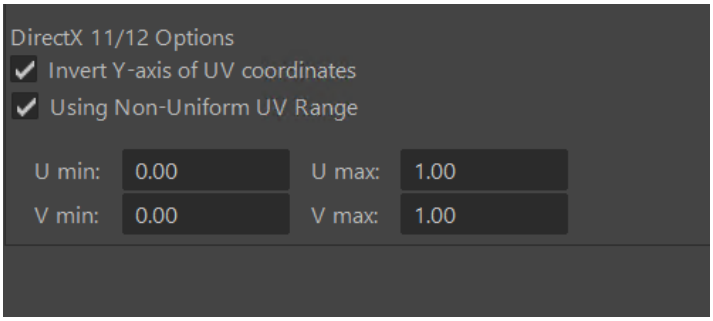
Commonly used/needed when exporting for Unreal Engine use. Maya typically uses Y-up for the up direction, while Unreal uses Z for the up direction. If Maya is not set to use Z as the up direction, this will swap the Y and Z values for you before export. It will not alter Maya settings or the Maya scene/DAG.

Invert Y-axis of UV coordinates (checkbox)

Inverts the Y (i.e. v) component of UV coordinates. DirectX® 11/12 needs the Y-axis inverted for proper UV alignment. This is a required option when exporting for use in the TressFX/Cauldron implementation.

Using Non-Uniform UV Range (checkbox)

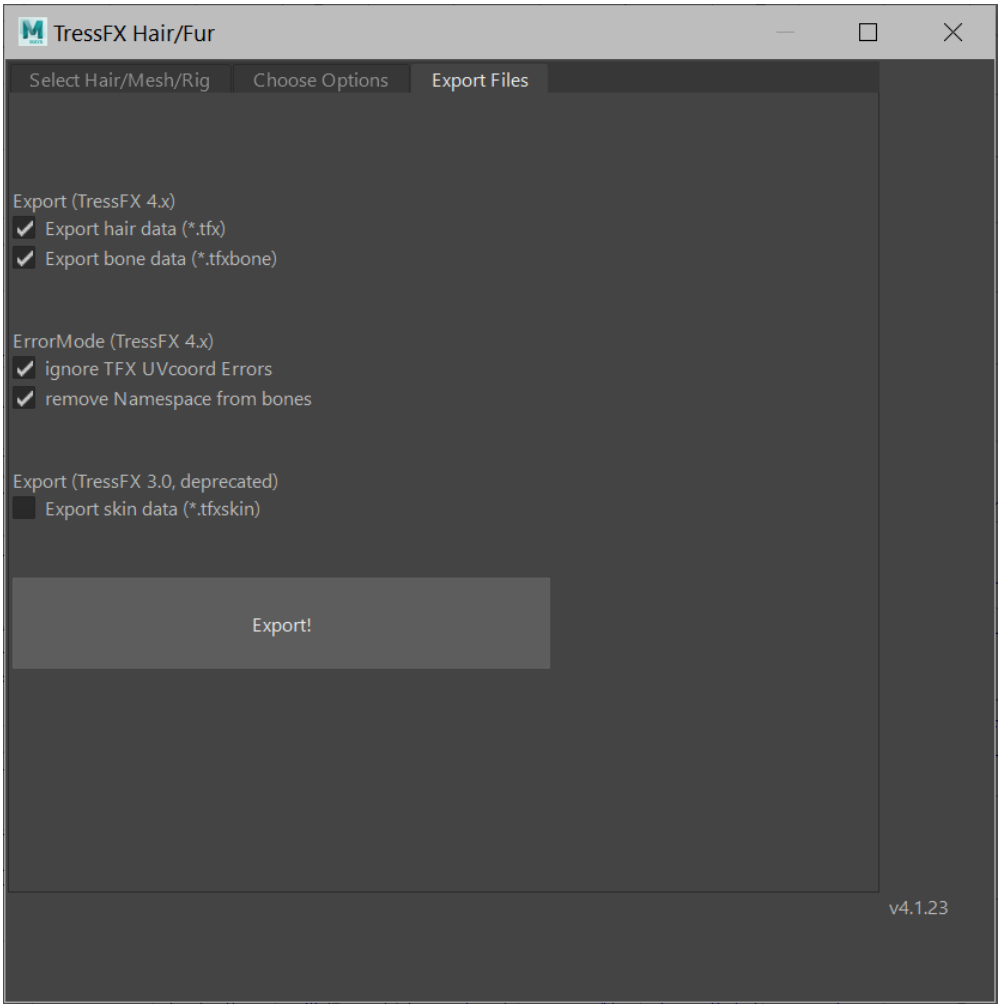
If inverting the Y-axis component of UV coordinates is selected, but the UV mapping is non-uniform (u:0-1, v:0-1 is uniform), this control lets you specify the UV mapping that your mesh is using. The v min and max values from this will be used during the Y-axis invert (instead of the default values of (0,1) for v.



Note: This user defined V min/max range is only used if **Invert Y-axis of UV coordinates** is selected**.**

Note: The U values are also taken but not used at this time. However, an informational print statement to the Maya Script window will reflect these changes, as well as a cmds warning that tells you that you are choosing to use a non-uniform UV range.

(EXPORT FILES) TAB GROUP 3



Export hair data (*.tfx) (checkbox)

Export a binary TFX file that contains hair strand and vertices data. Usually paired with a TFXBONE file, since both are needed for import and should match, with matching names (save for different extensions).

Export bone data (*.tfxbone) (checkbox)

Export a TFXBONE file that contains bone animation data (see .tfx above).

ignore TFX UVcoord Errors (checkbox)

Do not warn or fail if there are any issues with the UV coordinate look up during export (see the Exporter code for more information). Helps when trying to determine any issues with a skinned mesh model.

remove Namespace from bones (checkbox)

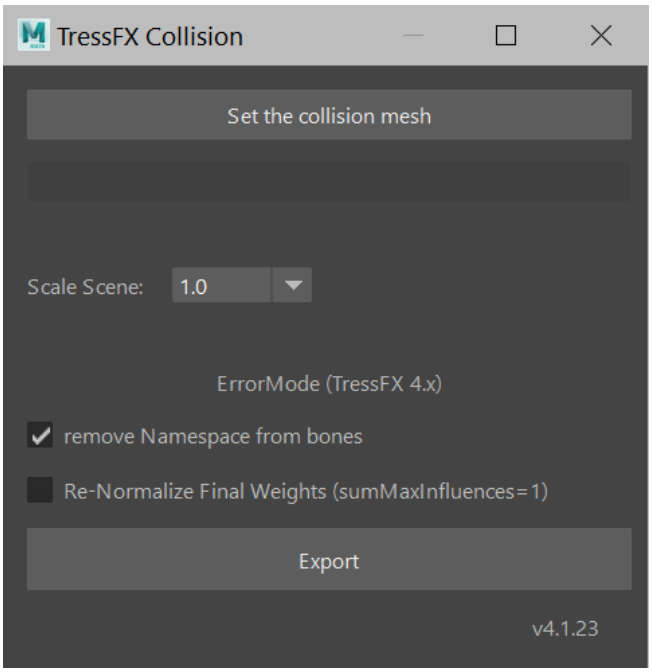
If there is a namespace on joint names, such as *myImportedBones:joint0*, this will remove the namespace from the string, leaving simply, the joint name, i.e. *joint0*. Important if the skinned mesh being used in the engine does not have any namespace on the joints. This situation can happen when importing an FBX or another Maya file into an open Maya file.

If the joint names do not match between the skinned mesh and the exported TFX files, then the simulation will not be able to find a joint name, or might find the wrong one if there is a match. (Joint names should be unique within a skeleton.)

Export skin data (*.tfxskin) (checkbox) (deprecated since TressFX 3.0)

This export process has been left in, but has not been tested nor worked on since TressFX 3.0 (not supported). Exports a file containing mesh data for triangle transform-based animation.

Collision Settings



Set the collision mesh (button)

Select a mesh, then click the button to set the reference mesh (shape) that will be used for .tfxmesh (collision mesh data) export. The selected mesh name will appear in the text box below the button.

Scale Scene (dropdown)

Same as for hair settings, will scale the points by multiplying the scale value against each .x, .y, .z position value. A value of 1.0 is fine for Unreal (cm), but Cauldron uses meters, so needs a scaling value of .01 to scale the data from cm->m.

Remove Namespace from bones (checkbox)

Same as for hair settings, if there is a namespace on joint names, such as *myImportedBones:joint0*, this will remove the namespace from the string, leaving simply, the joint name, i.e. *joint0*. Important if the skinned mesh being used in the engine does not have any namespace on the joints. This situation can happen when importing an FBX or another Maya file into an open Maya file.

If the joint names do not match between the skinned mesh and the exported TFX files, then the simulation will not be able to find a joint name, or might find the wrong one if there is a match. (Joint names should be unique within a skeleton.)

Re-Normalize Final Weights (sumMaxInfluences = 1) (checkbox)

Same as for hair settings. Forces the four bone weights per strand vertex to sum to one. Divides each weight by the sum of the four weights. Often, this is not needed, but can be used to ensure normalization.

Tutorials

A Quick Tutorial on Creating a Basic Skeletal Mesh in Maya

Introduction

TressFX 4 hair is bound to a skeletal mesh, meaning a mesh that is bound to a skeletal rig (bones). Most users of TressFX will hire experienced modelers and animators to create their rigs, and ultimately create guide hairs attached to that skeletal mesh. However, for testing purposes it is useful to be able to create the minimum required skeletal mesh. This tutorial will walk you through the basics of creating a sphere mesh bound (skinned) to a simple joint chain (a hierarchy of five joints...aka bones). This is the exact same skeletal mesh that is used in the Maya tutorial on creating and exporting TressFX hair.

Related Links:

Creating and Exporting TressFX 4 Hair from Maya
Adding TressFX 4 (hair/fur) to skeletal meshes (TressFXComponent to UE4 Blueprint)

Requirements

Autodesk Maya 2015 or higher (this tutorial uses Maya 2017, but 2015 is the minimum based on TressFX/XGen usage)

A basic understanding of how to launch and navigate in Maya (using the move/rotate/scale tools, zooming and panning in the main perspective view)

Let's Begin!

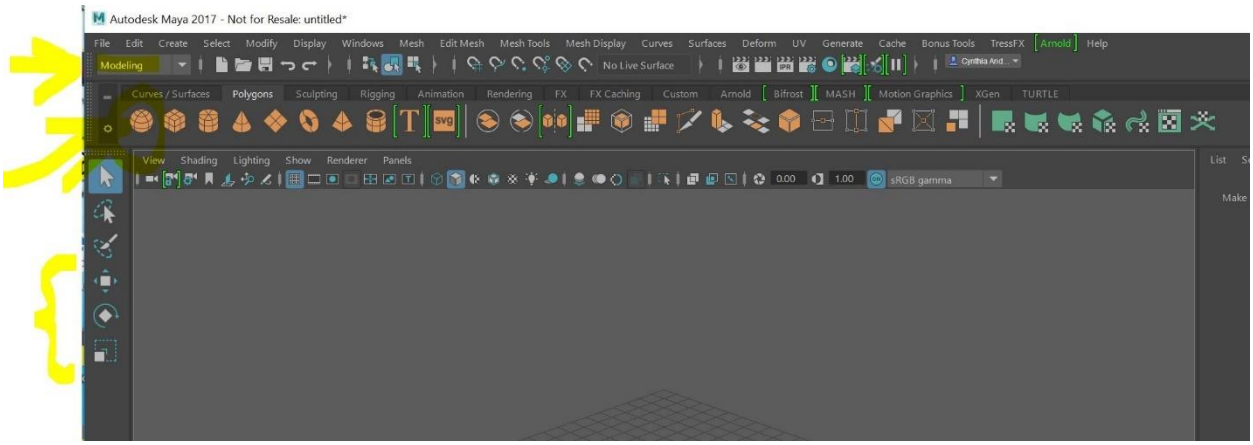
Step 1

Open a new scene in Maya. (Basically, open Maya and a new scene is automatically generated for you.)

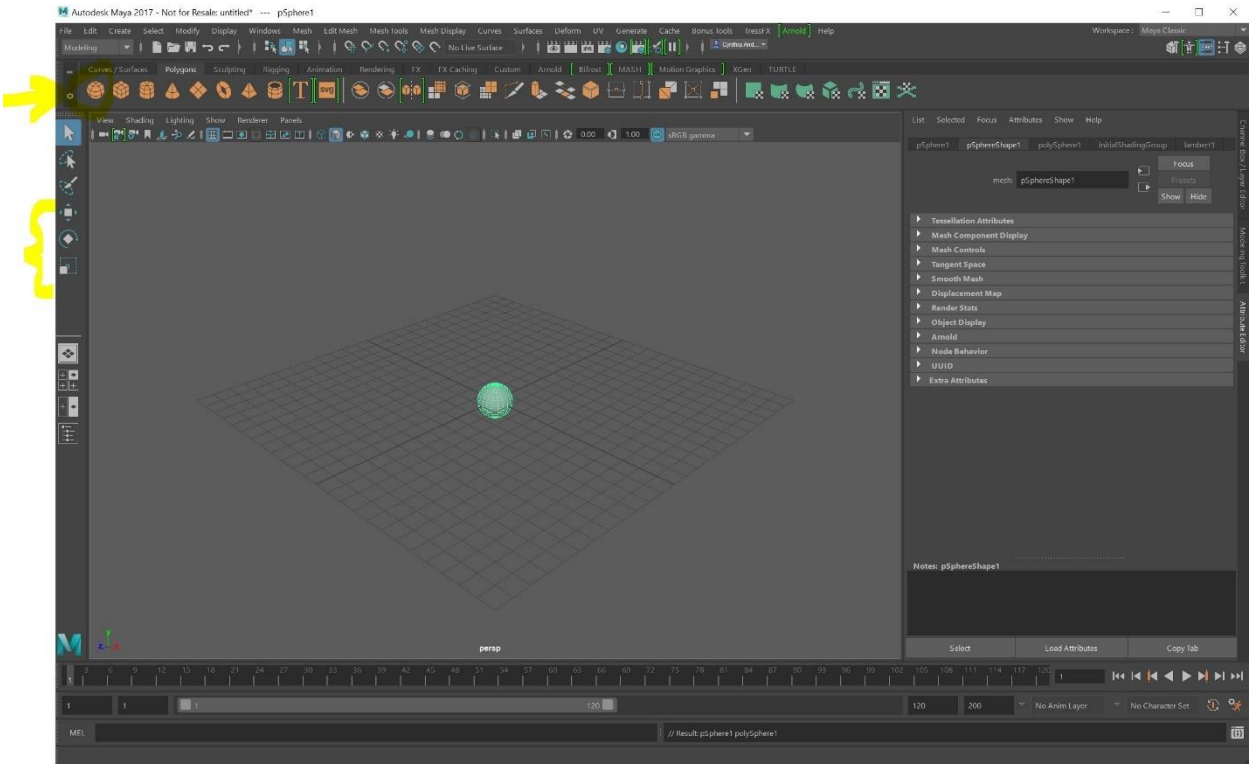
Make sure you are in **Modeling** mode.

Open the **Polygons** tab.

Click **Sphere**.

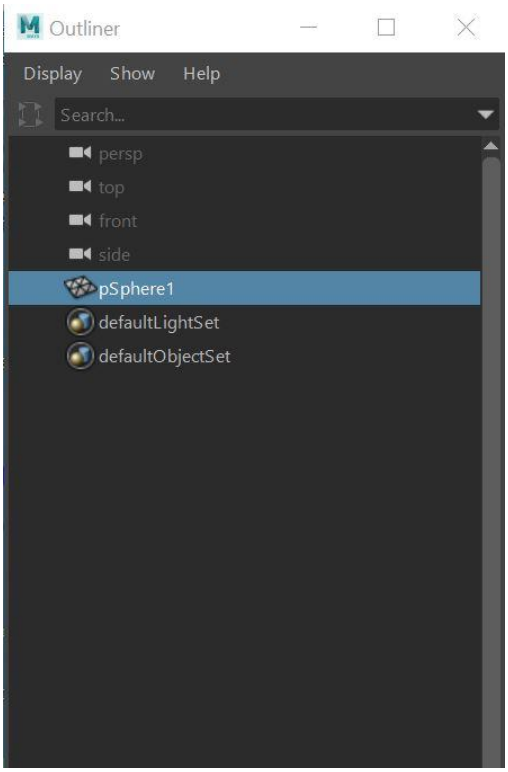


You should see in the main *perspective* view that a sphere is created.



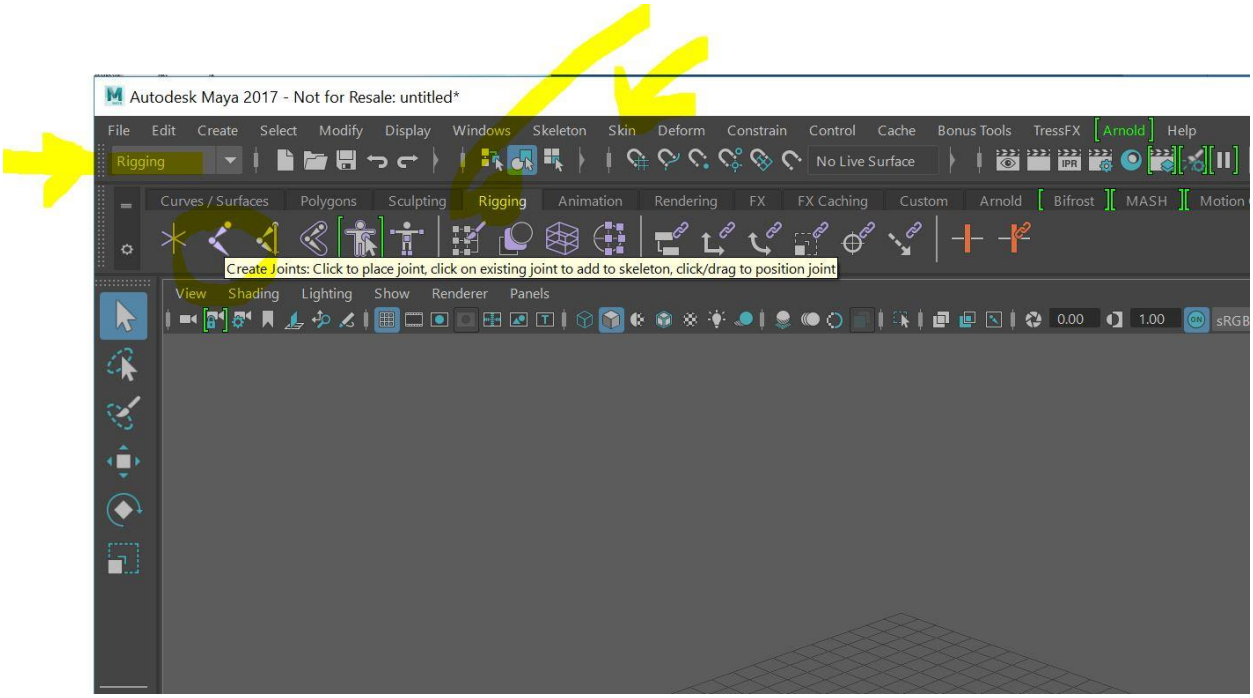
You can either make the sphere larger using the **Scaling** tool, or you can *zoom in* so you can see the sphere easier.

Open the **Outliner Window** if it is not already open: on the top level menu, choose **Windows > Outliner.** You will need it to move your joint chain, select your joint hierarchy and mesh for binding, etc.

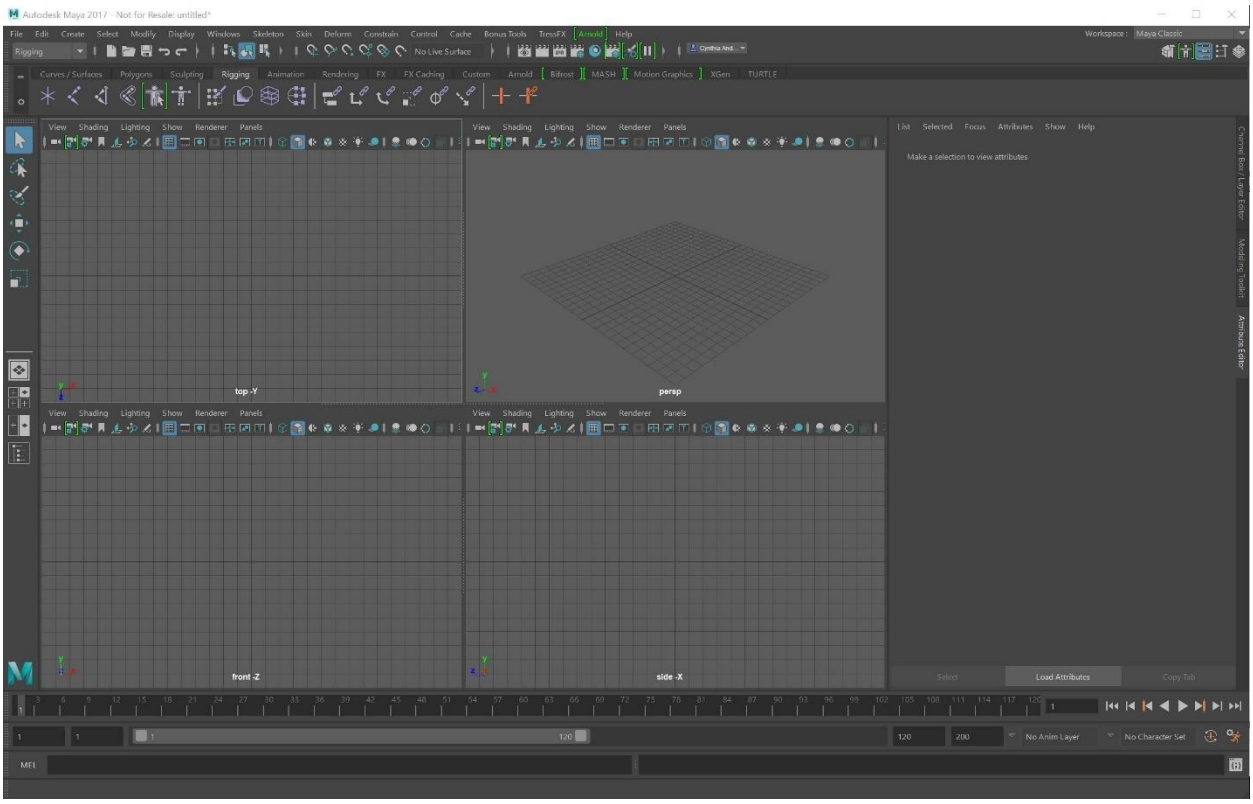


Step 2

Change to the **Rigging** mode.



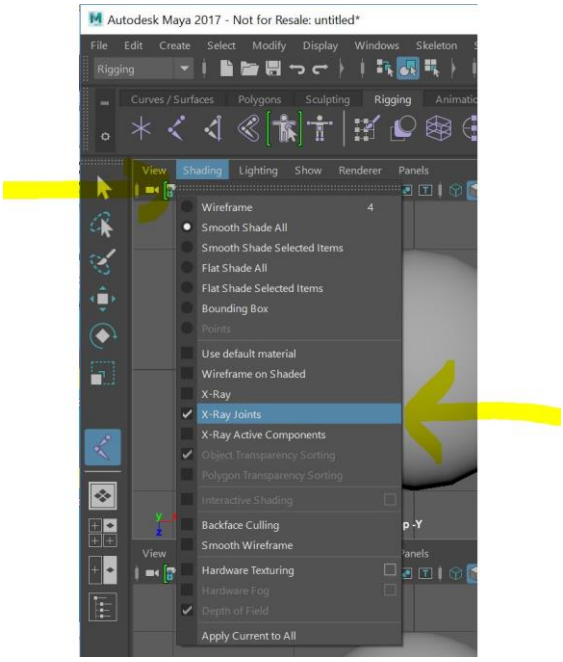
In the main *Perspective* view, tap the view to make sure it has the focus, then tap the **Spacebar**. This should send you into a four-way view: *Perspective, top, side, front.*



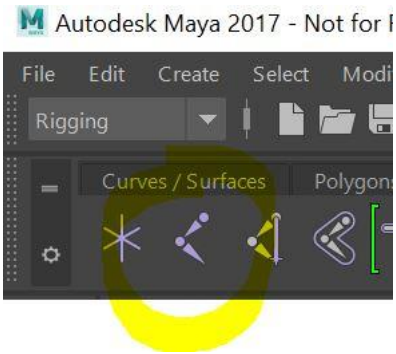
Click anywhere in the *front* view. This will activate the tool in that view.

On the **Shading** tab of the view, choose **X-Ray Joints**.

You *may* need to do this for each view you want to see the joints through a solid mesh. You can try to set the perspective view to X-Ray joints, when it is the only view — not four way — and see if this setting propagates to all the views. If it doesn't, set each one individually.

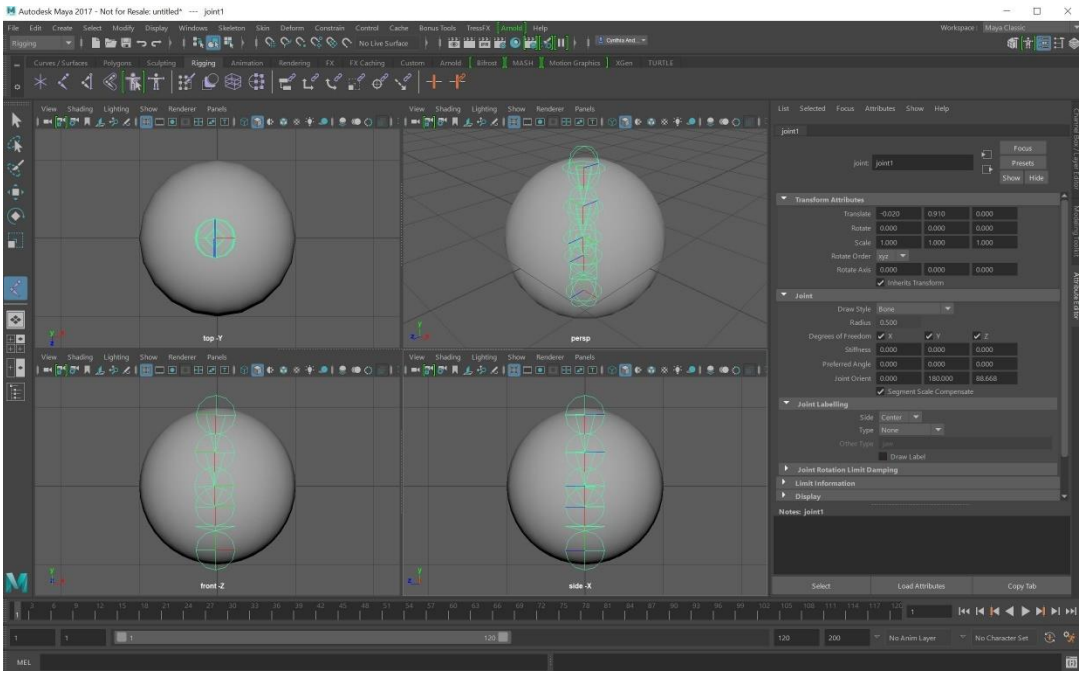


Click the **Joint** tool.



Now each click you make will create a bone. The first click will be the first bone, the next click will be the second that is automatically parented (a child of) the first bone, and down the chain.

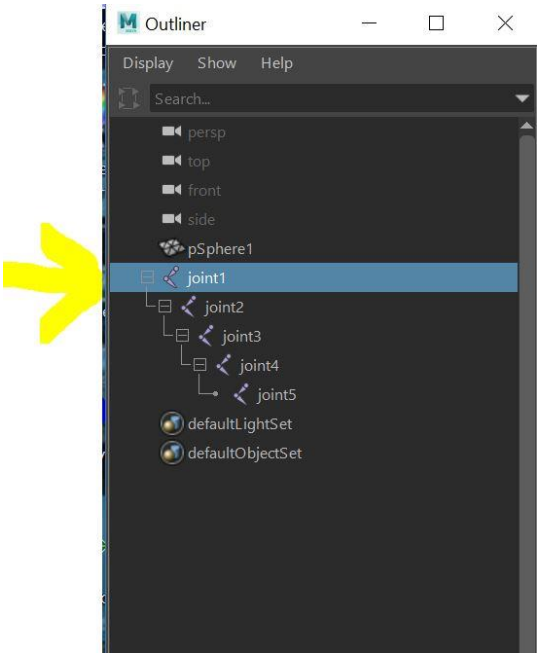
In the view, along the sphere, make five bones in roughly a long line. Keep them within the bounds of the sphere.



Click **Return** to stop the chain creation.

Look at the *side* and *top* views. If the bones are not aligned inside the sphere. First click the *top of the joint chain* in the **Outliner**, then click the **Move** tool and then click in the appropriate view window (*top, side, front*) to move the joints to where you want them.

Generally, it's better to have joints within a mesh. However, experienced riggers will often do different configurations. But this is just a simple skeletal mesh, so we will stay with the basics and let Maya do the rest...and use its defaults.

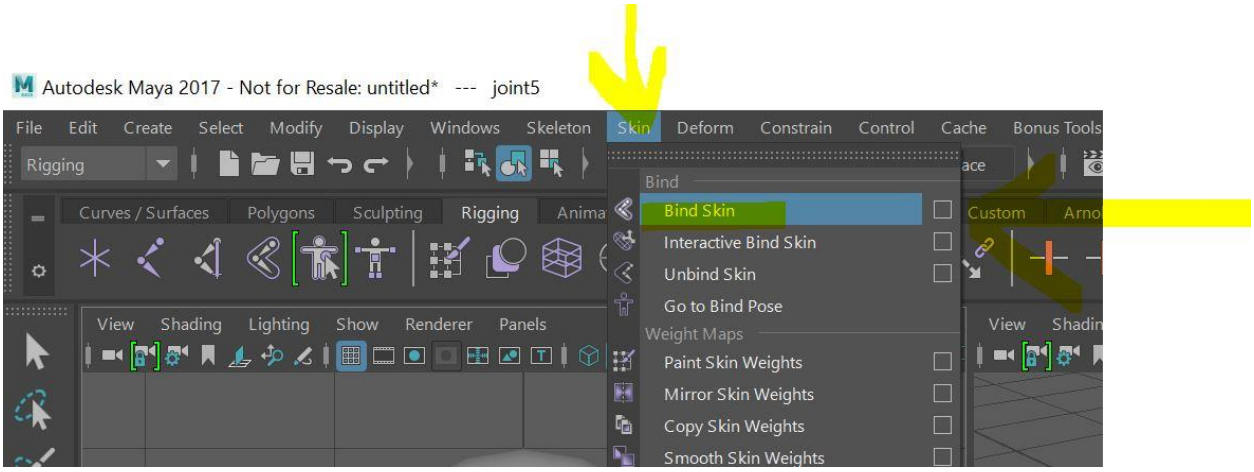


Step 3

When you are ready, click the sphere mesh in the **Outliner**, then press and hold *Shift* and click the top joint in the joint hierarchy.

You are still in **Rigging** mode.

Now go to the menu item **Skin**, and select **Bind Skin**.

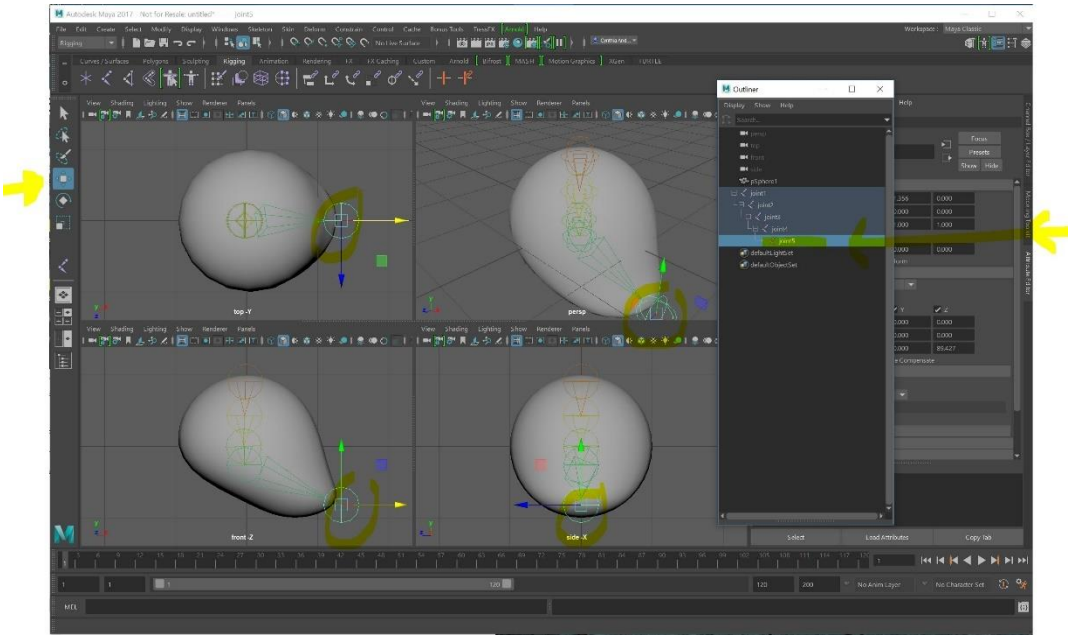


Your joint chain should be bound to your mesh now.

To test this, in the **Outliner**, select one of the joints, and use the **Move** tool to move the joint. Not only should the children of that joint in the chain move (if any children), but also the skin should warp and move as well.

In this picture, joint 5 is being moved. The skin stretches and deforms. Try other joints and other ways to manipulate it (move or rotate, for the most part).

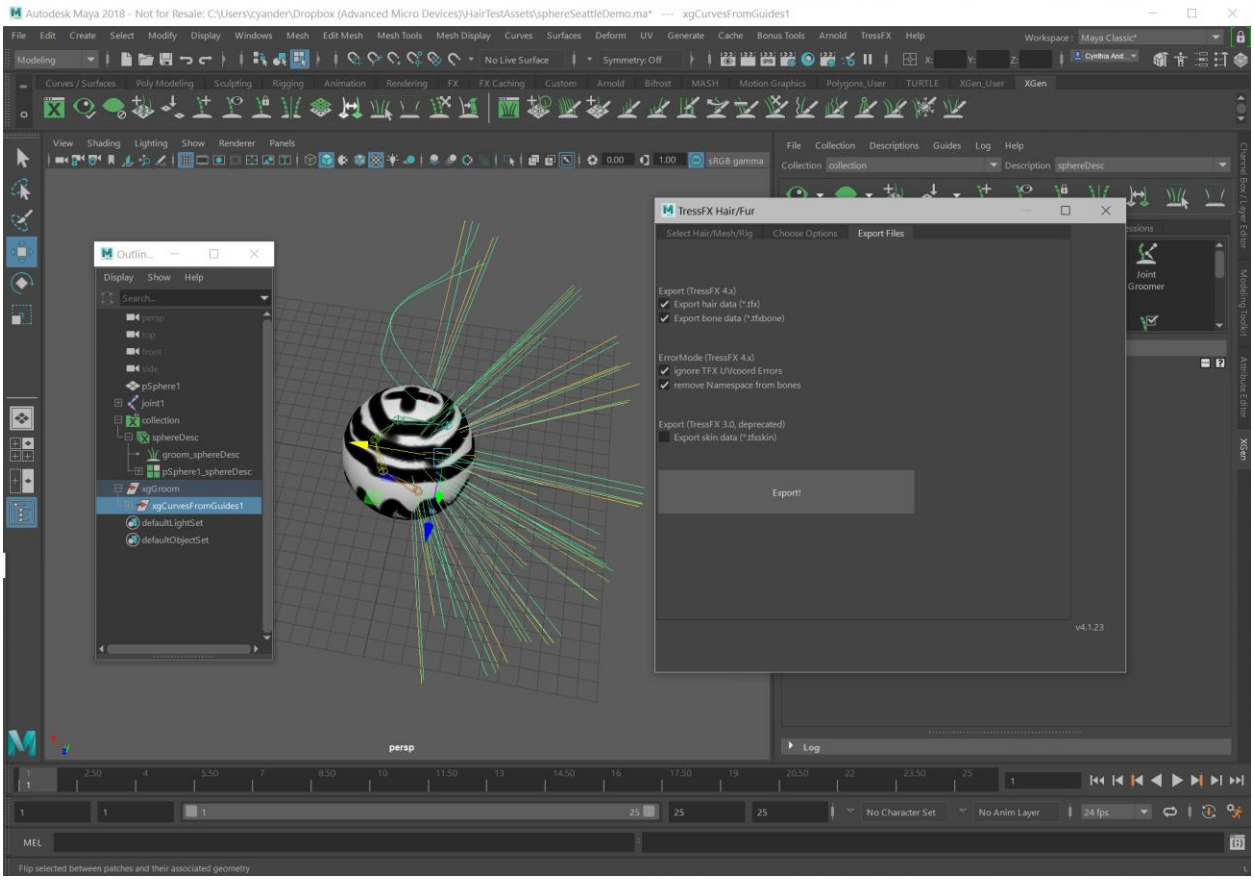
Note: If you scaled your sphere larger, you may notice the bones appear smaller than in these pictures. That is normal.



You are now ready to use this simple skeletal mesh — to animate it, add hair, and so on.

Creating and Exporting TressFX 4 Hair from Maya

Introduction



The following is a quick walk-through on how to use Maya's XGen to create 'guide hairs'. Then to export those guide hairs using the TressFX for Maya Exporter. The files produced (.tfx, .tfxbone and .tfxmesh) can then be used, along with the skeletal mesh, in the TressFX Unreal 4 build to create a skeletal mesh asset 'with hair'.

TressFX shader parameters have been set in that Unreal skeletal mesh to be wide strands, few strands, and globally stiff so the original shape in Maya is easy to discern.

Related Links

Adding TressFX 4 (hair/fur) to skeletal meshes (TressFXComponent to UE4 Blueprint)
A Quick Tutorial on Creating a Basic Skeletal Mesh in Maya

Requirements

Maya 2017 (you can use Maya 2015 or Maya 2018 or later as well). This tutorial uses Maya 2017.

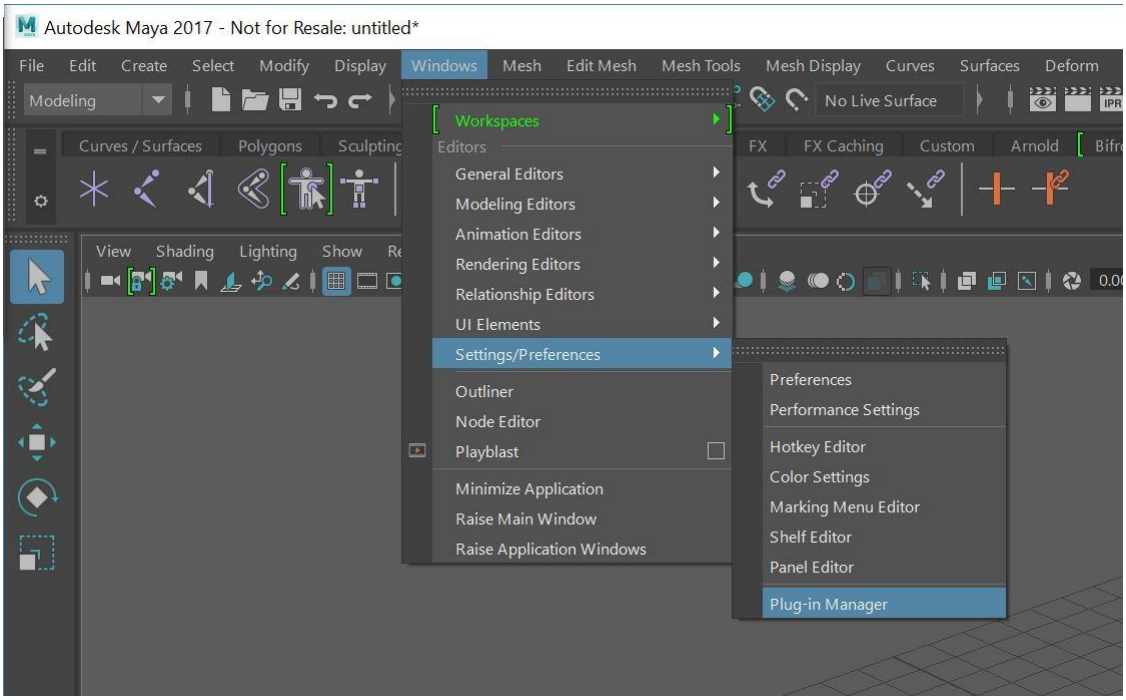
The TressFX Exporter for Maya (TressFX_Exporter.py).

Note: The Exporter was updated. Be sure to use the most recent Exporter (TressFX 4.1).

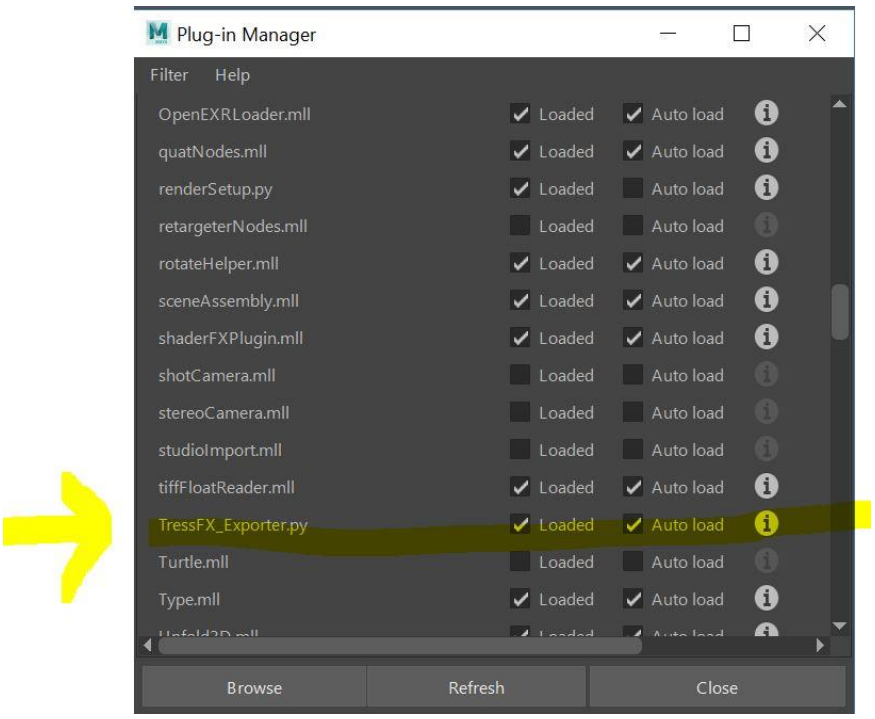
Let's Begin!

Step 1

Install the TressFX Exporter as a plugin. An easy way to do this is to take the Exporter (a python script), and copy it to the **bin\plugins** directory of Maya. For example, if you installed Maya in **C:\Program Files\Autodesk\Maya 2017**, you would copy the exporter to **C:\Program Files\Autodesk\Maya 2017\bin\plugins**.

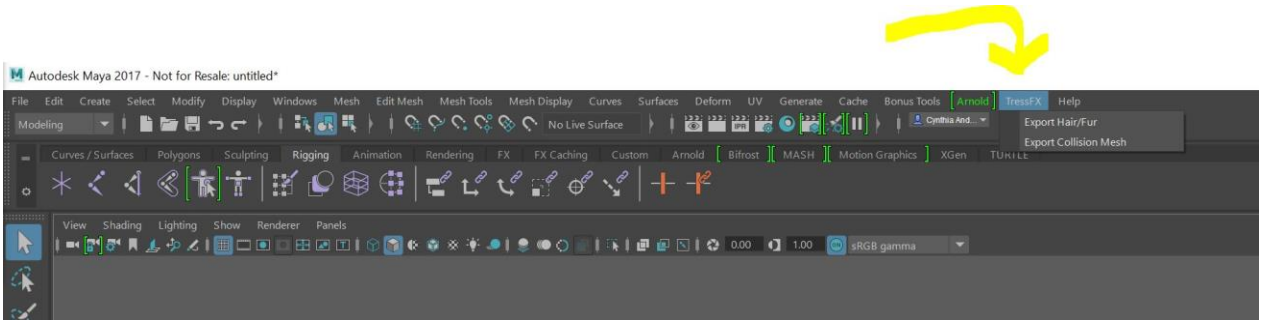


Launch Maya and open the Plugin Manager. From the top menu in Maya, choose **Windows > Settings/Preferences > Plug-in Manager**.



Scroll down until you find the TressFX plugin. Select both the Load and Autoload options. Then close the window.

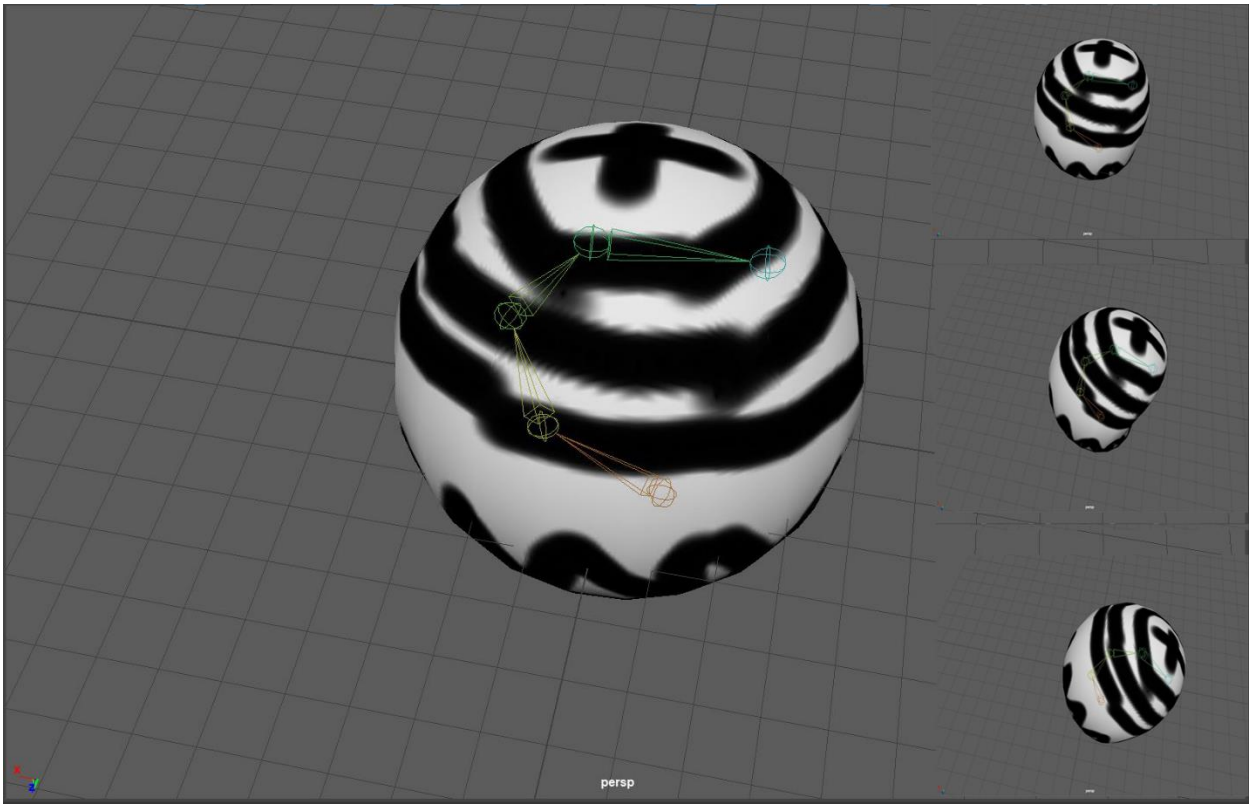
You should now see the TressFX menu on the top-level menu bar.



Step 2

Load or create a skeletal mesh model. In other words, you need a well-behaved, well-formed mesh ("quad modeled" has proved to be the best so far) that has been skinned to a rig with at least four joints. You might be able to get away with fewer joints, but for testing purposes, having five joints has proven easiest. TressFX 4 binds the hair to the bones (influences) and uses a maximum of four influences (bones) per vertex. (TressFX 3, which bound the hair to mesh vertices is deprecated.)

For this tutorial, we are using a basic sphere (with a ‘not artistic but useful’ texture) that is rigged (and animated).



You will be using the mesh as the reference mesh that the exporter requires. The rig needs to be skinned to the mesh before export.

Step 3

What you will be doing. You will be creating hair and then exporting three files.

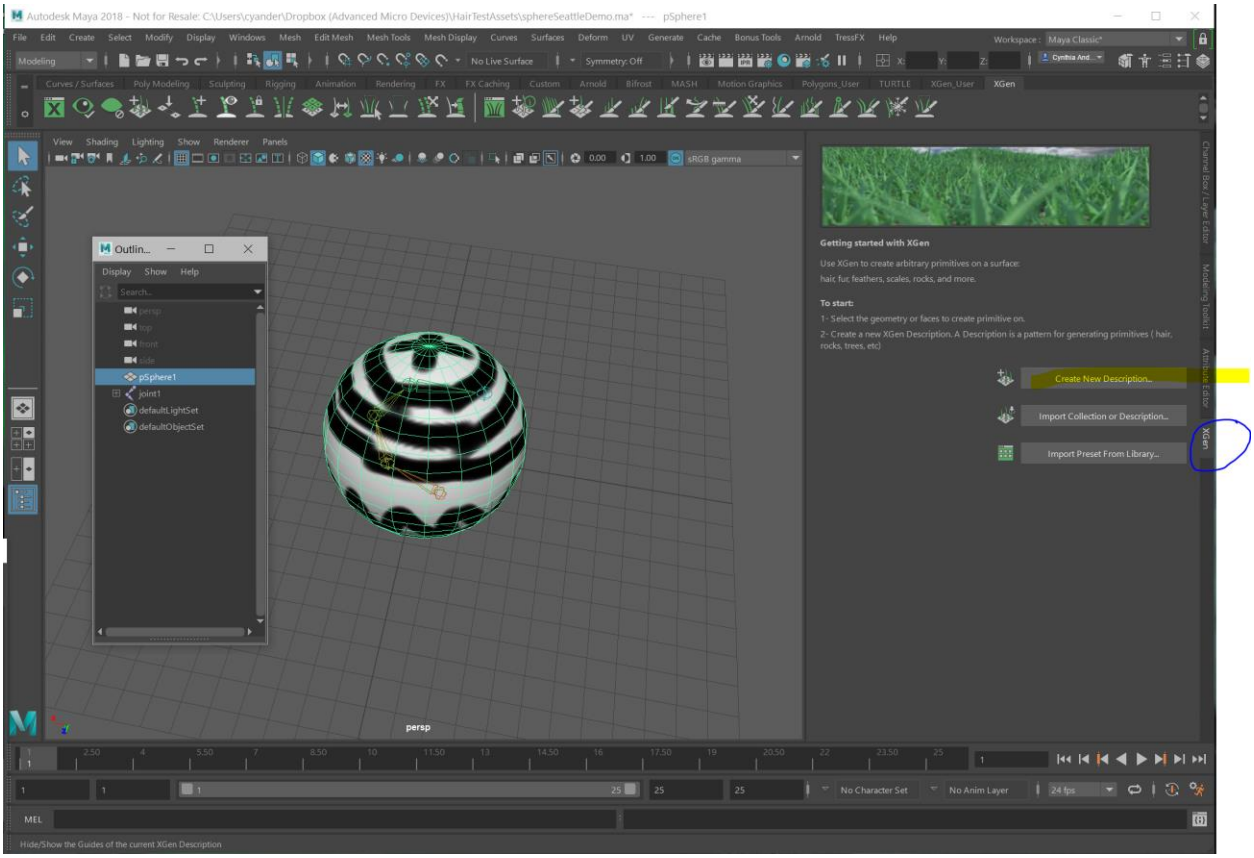
- A TFX file, which contains the guide hairs (as spline curves).
- A TFXBONE file, which contains the bone (influencer) information required.
- A TFXMESH file, which contains the collision mesh to use for this TFX/TFXBONE file set. Typically, you will use a simplified but fairly form hugging mesh that mimics your skeletal mesh. You can use multiple collision meshes (and TFX/TFXBONE sets) for a single model. The RatBoy example uses three collision meshes.

The collision mesh is used to prevent the hairs from penetrating a surface (such as the head or body mesh). Having a basic collision mesh that is simpler and then detailed collision meshes for smaller parts of the model, such as the hands, is a normal use of collision meshes.

Note: See the tutorial on hooking up collision meshes and SDF fields to hair assets for more information on collision/SDF/hair asset interactions.

Step 4

You can see the XGen settings tab on the main display. (To the far right, typically). Under that tab (far left) you can click and bring up the XGen settings window. If you haven't got the ChannelBox/Attribute/... Panel open, you will need to open the panel to see XGen.

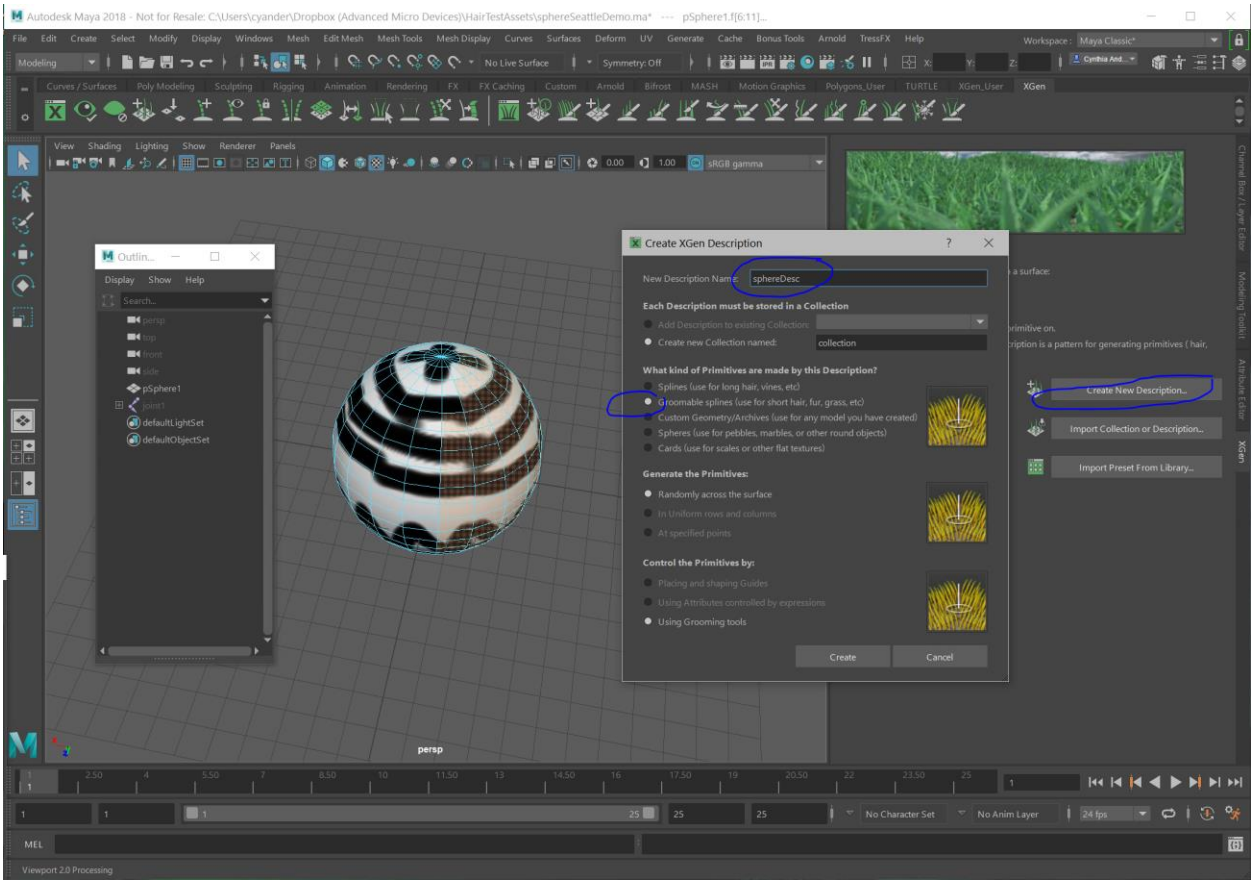


Step 5

Put your Mesh in the Bind Pose.

TIP: Try to make the Bind Pose as close to location 0,0,0 as possible. This is to avoid any possible conversion errors that may result in animation offset errors.

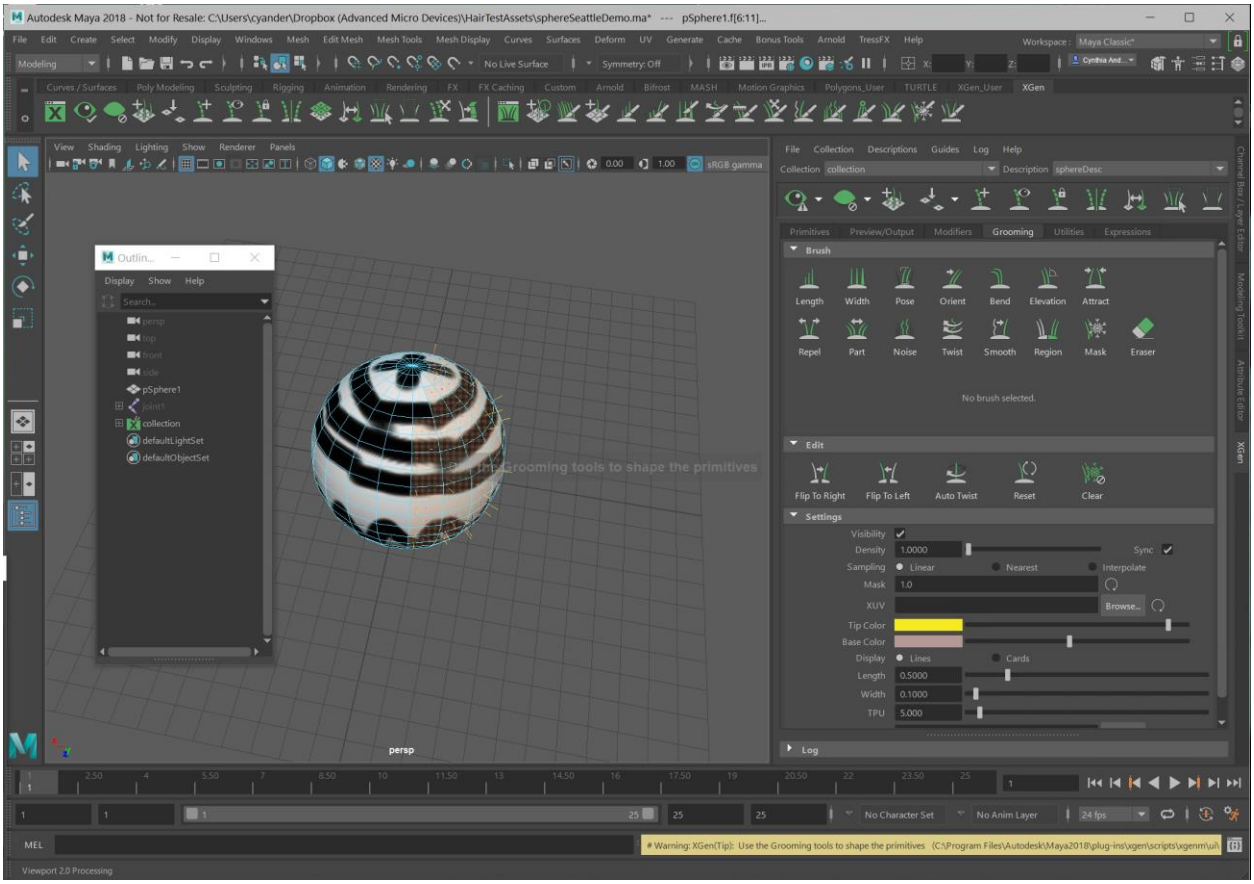
Select the entire mesh or a subset of Faces of the mesh.



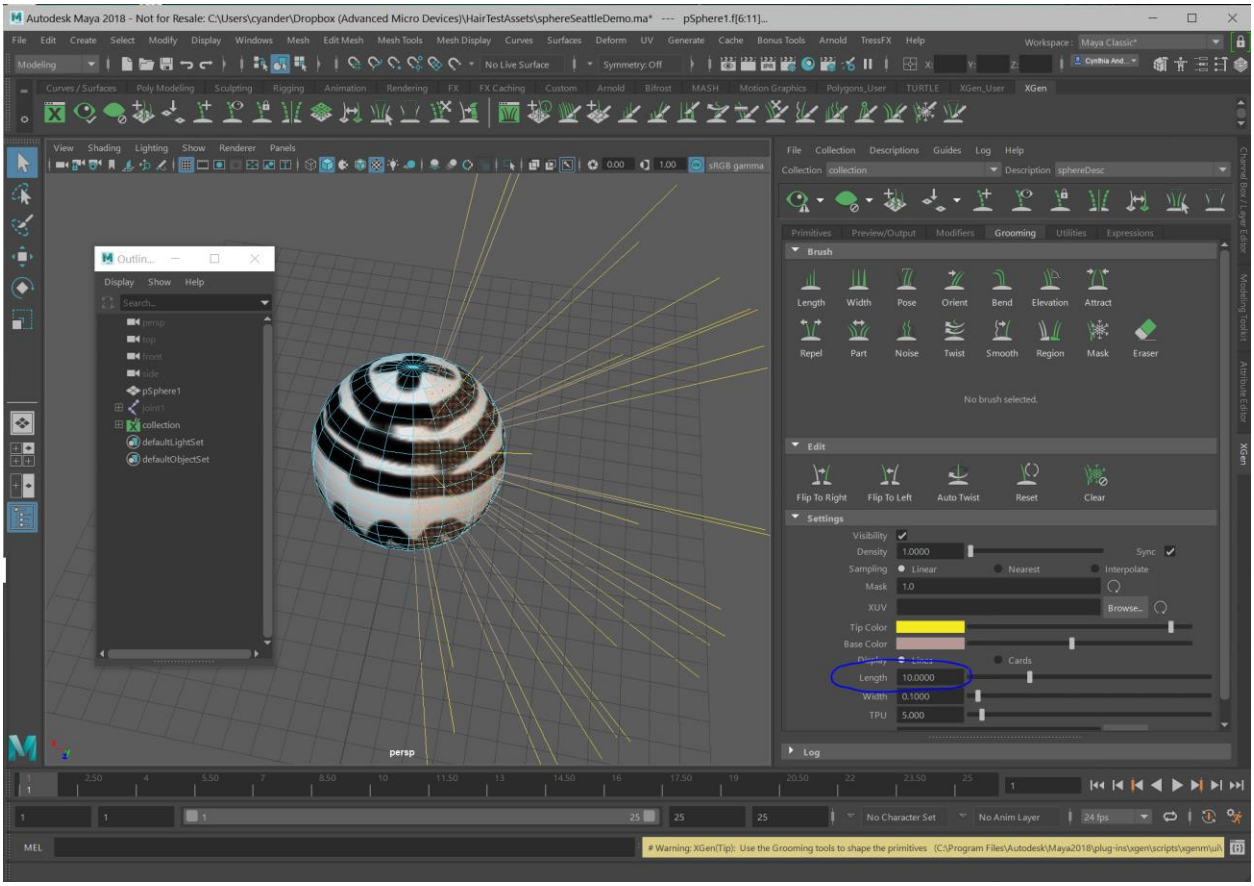
Then create a new description, click either the button in the Panel or the button in the row of buttons under the XGen tab.

Name your description, for example, *sphereSection*. And your collection, which holds multiple descriptions. For example, *sphereHair*.

You can choose Splines or Groomable Splines. Splines you would place individually. For this tutorial, we are doing the quicker (and slightly more complicated) groomable splines. Choose **Groomable Splines.**



You will see tiny guide hairs randomly distributed across the faces you selected.



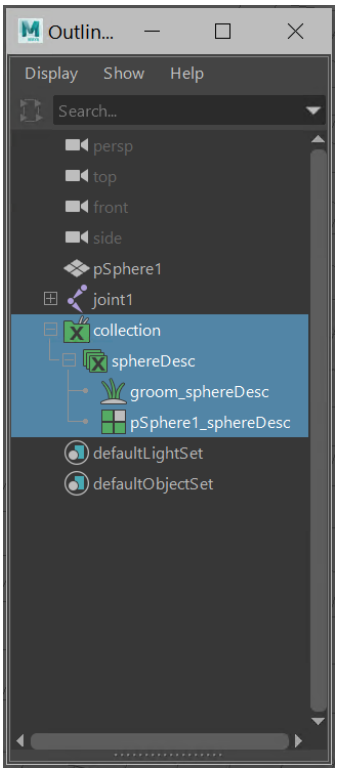
Change the **Length** setting (Grooming Tab, Settings area...far right) to something more dramatic, like 10 or 20. (Type it in, as the slider has a limited range based on the current value.)

You can use the grooming tools to shape the hair. ***(Which we do not do here, but feel free to play before continuing.)***

Note: You cannot edit the grooming splines directly via control vertices. See the next step.

Step 6

Now we are going to convert those groomable splines to actual curves.

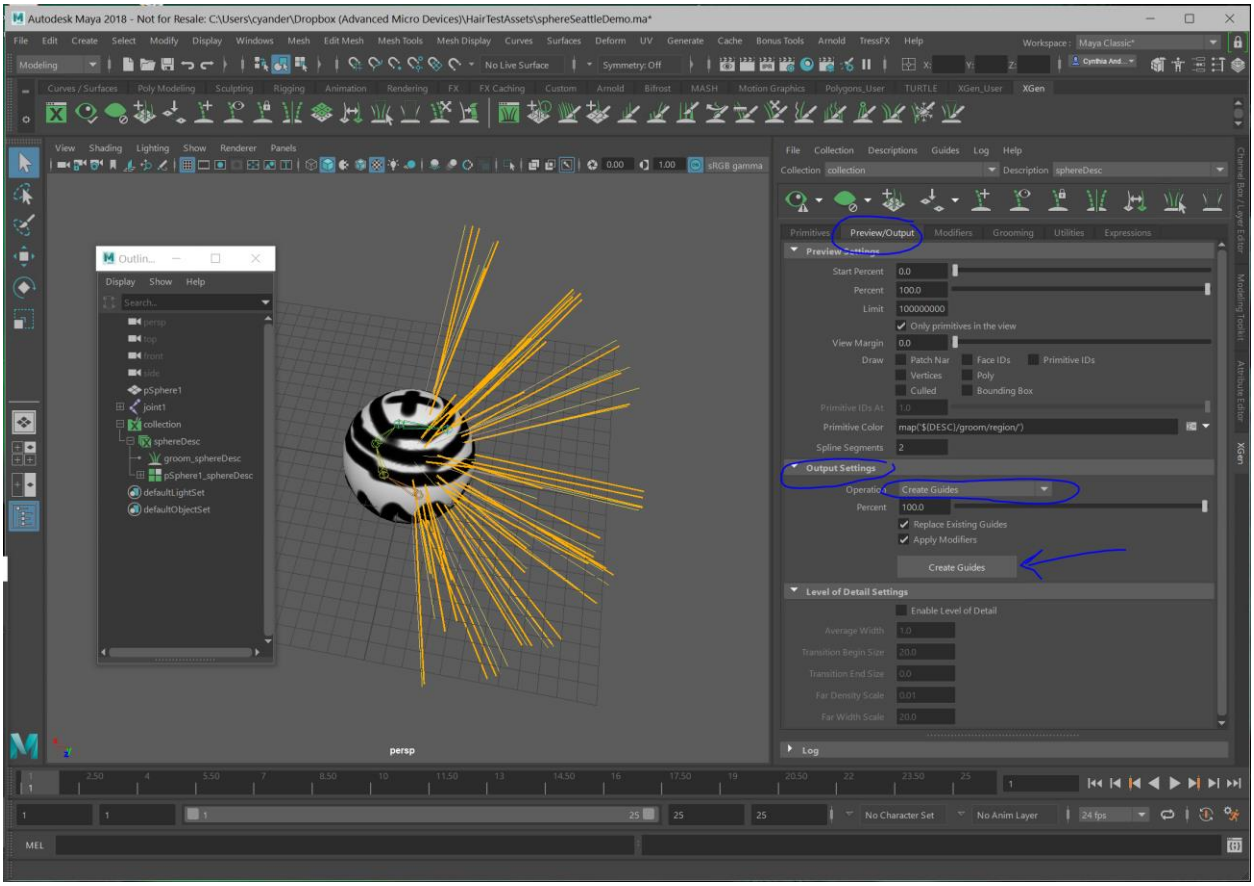


If you look in the Outliner, under the collection you created, you will notice that you only have descriptive information. You do not have any actual splines. That is why you cannot edit these groomable splines any direct way. You must use the grooming tools. So now, since we require actual splines in order to export, we will convert these groomable splines into individual splines.

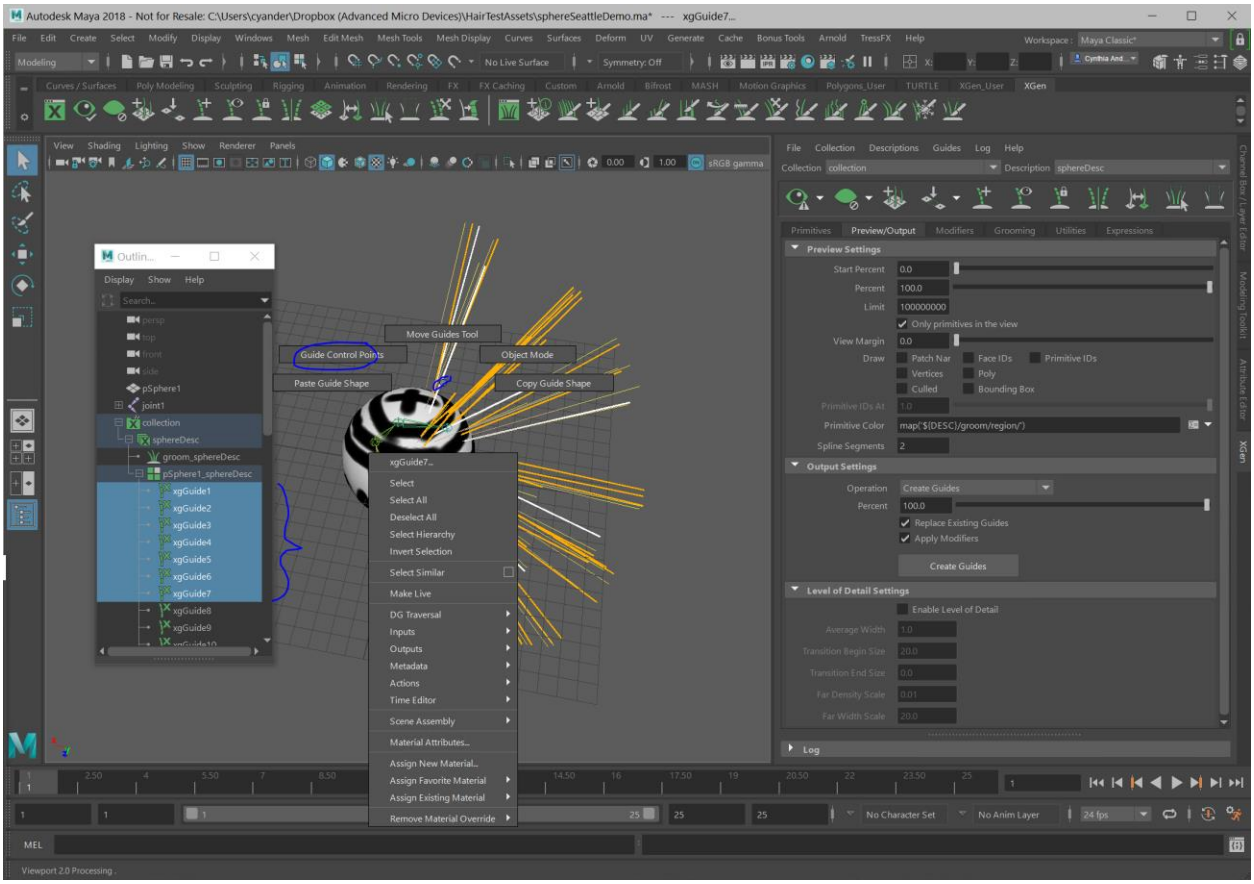
Go to the **Preview/Output** tab in the XGen Panel.

Go down to the *Output Settings* section.

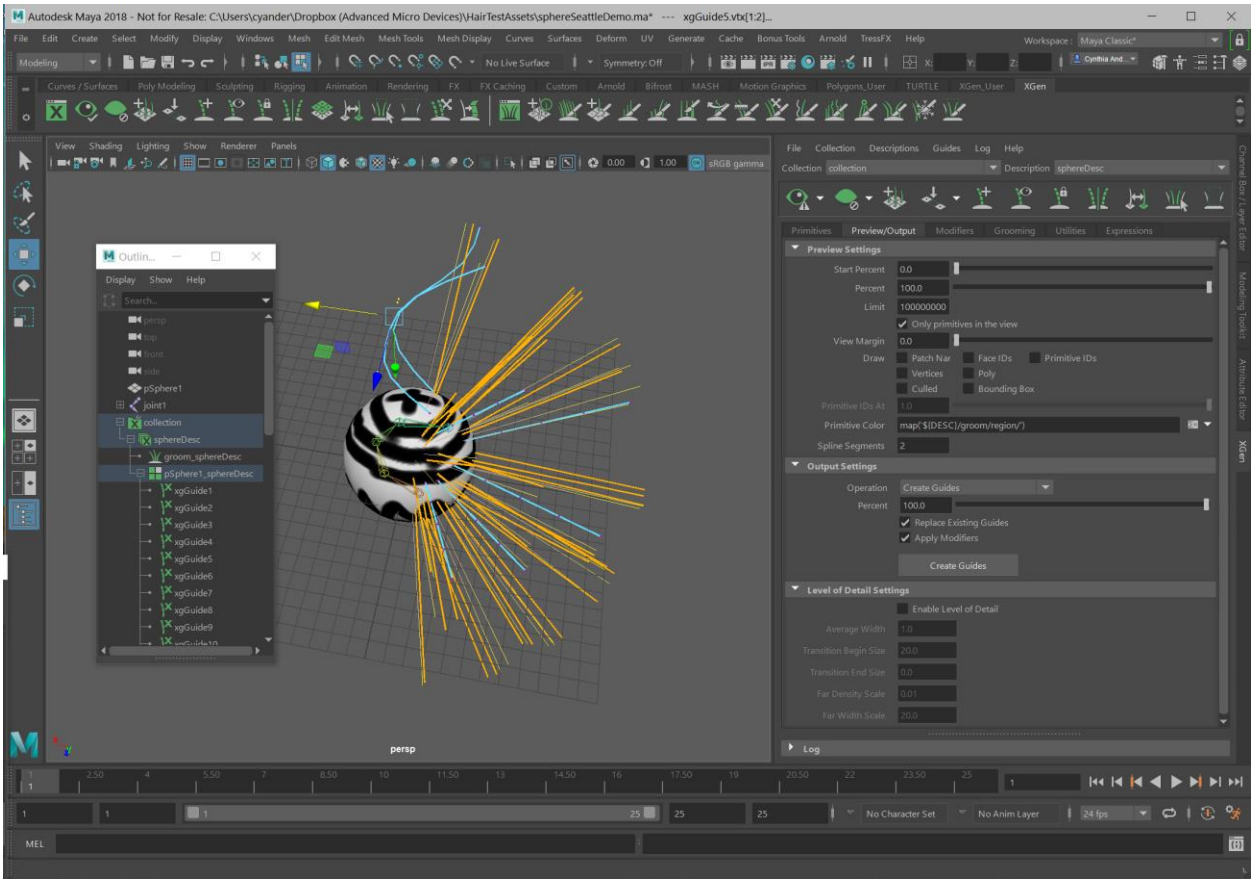
Click the dropdown for **Operation**. Change it from the default *Render* to *Create Guides*. Then click the **Create Guides** button that appears.



Guides that you can directly manipulate are now created. You can see them in the Outliner.



And if you want, select them (right click then hold for markup menu) and ask to manipulate them using Guide ControlPoints.



Then select and move the control points with the standard transformation tools.

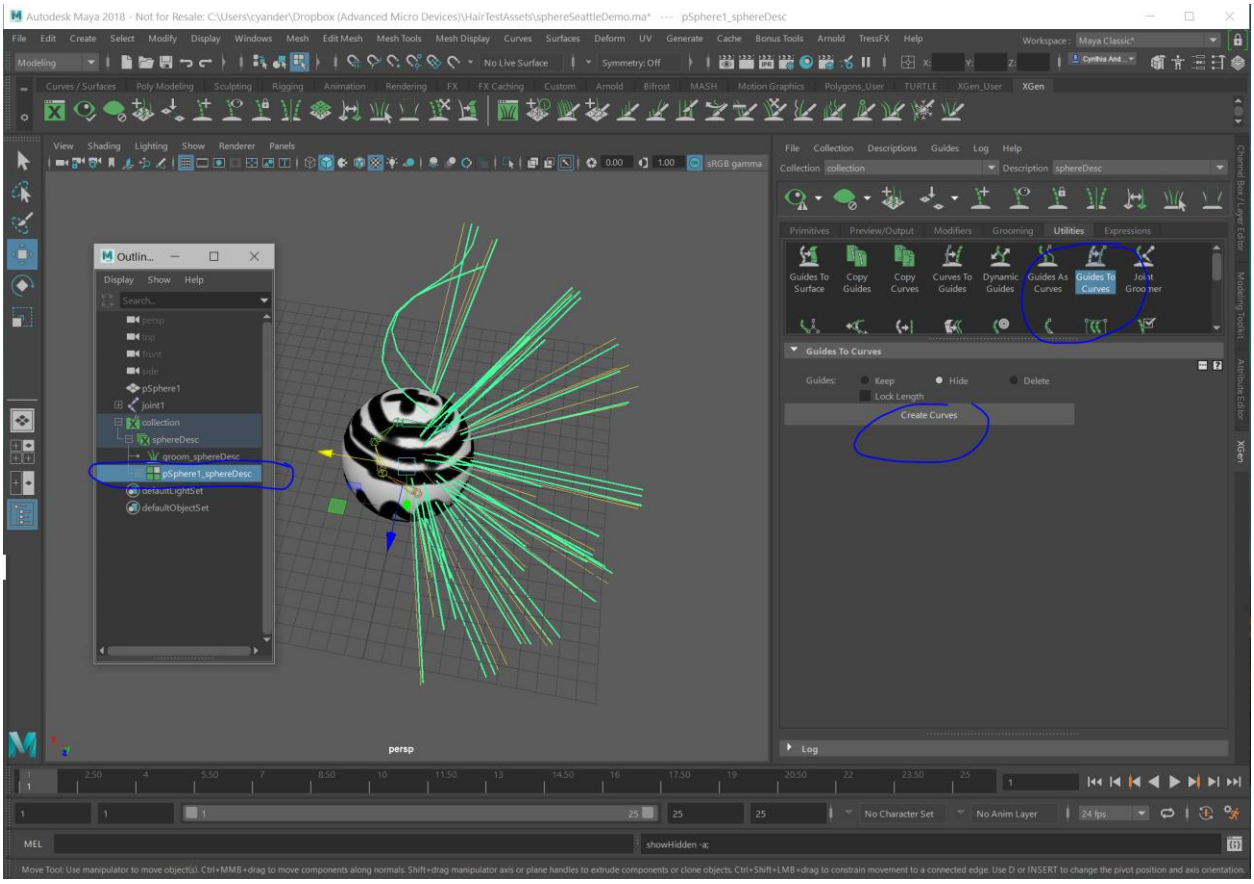
Step 7

Now we need to convert these guides into curves so we can export them with the TressFX Exporter.

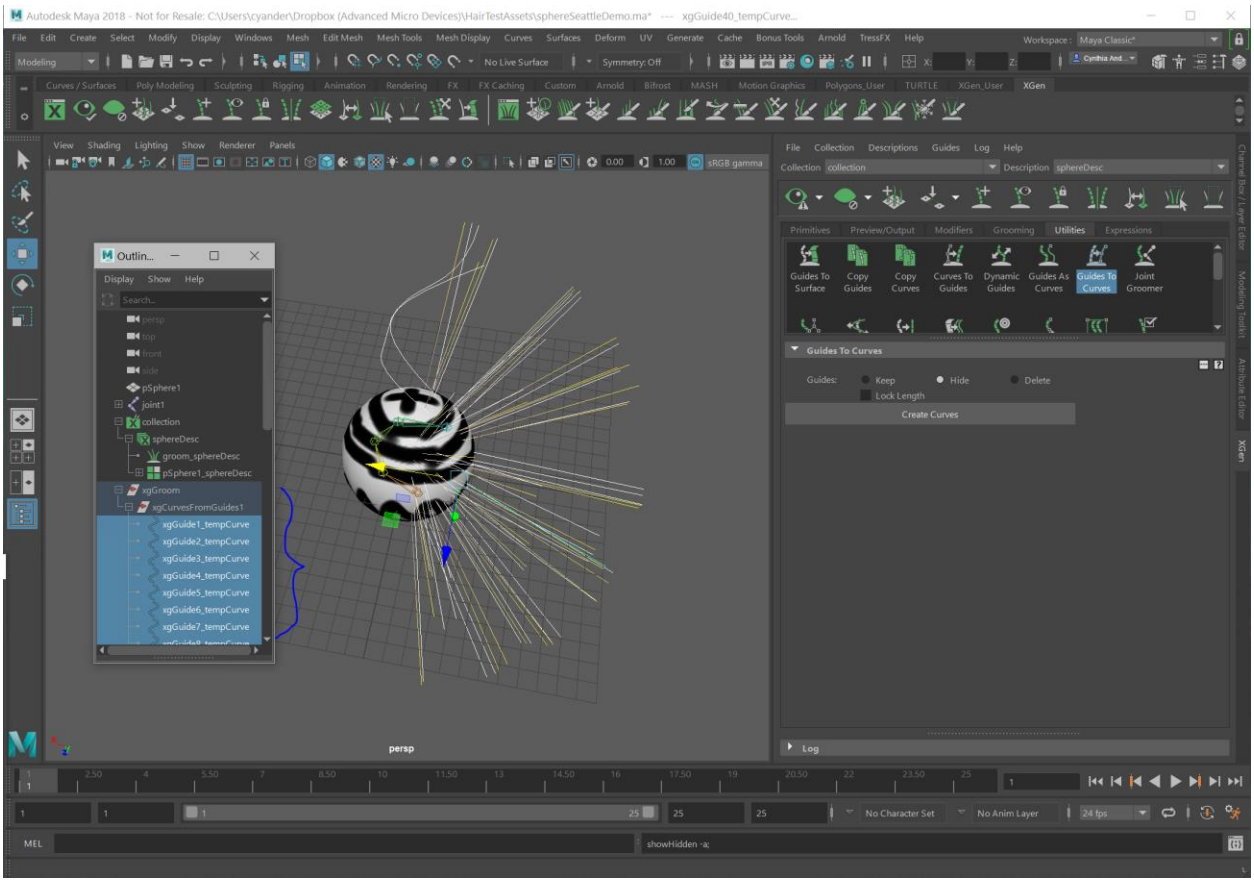
Go to the **Utilities** tab in the XGen Panel.

Click the **Guides to Curves** tool. This makes it available for use (in the area below the tools list.) You should see the section called *Guides To Curves* appear. Make sure it is open, so you can see the options and button.

In the Outliner, press and hold Shift and select all the individual guides. (You should select all the guides you want to convert.)



Click the **Create Curves** button in the *Guides To Curves* subsection of the XGen Panel.



The guides should be converted to curves now.

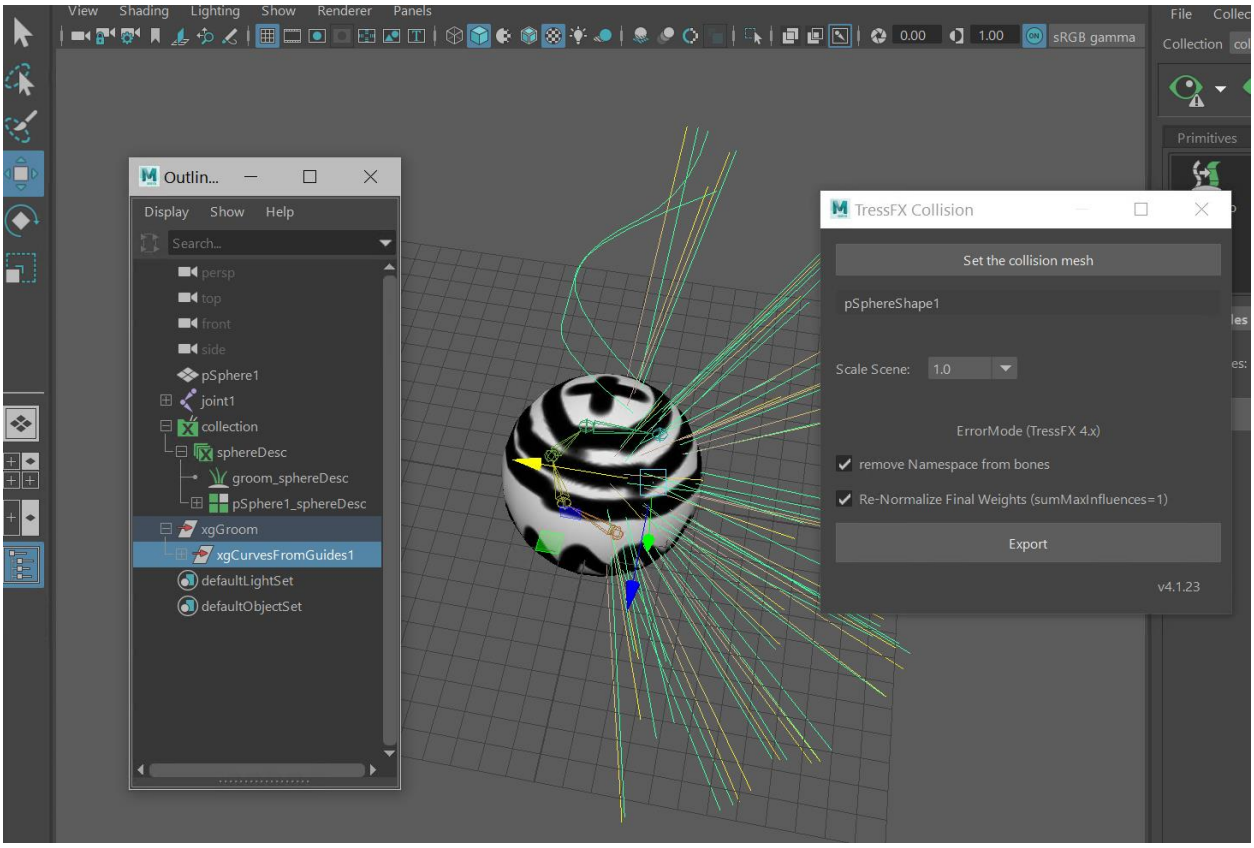
The curves will be in their own group in the Outliner, under the default name of **xgGroom**.

Step 8

Open the menu item: **TressFX →Export Collision Mesh**

In the Outliner, click on the sphere mesh (not the joint hierarchy).

Note: Make sure your joint hierarchy is skinned to the mesh, though!



In the TressFX Collision dialog, click **Set the collision mesh** button. You should see the shape of the mesh now set.

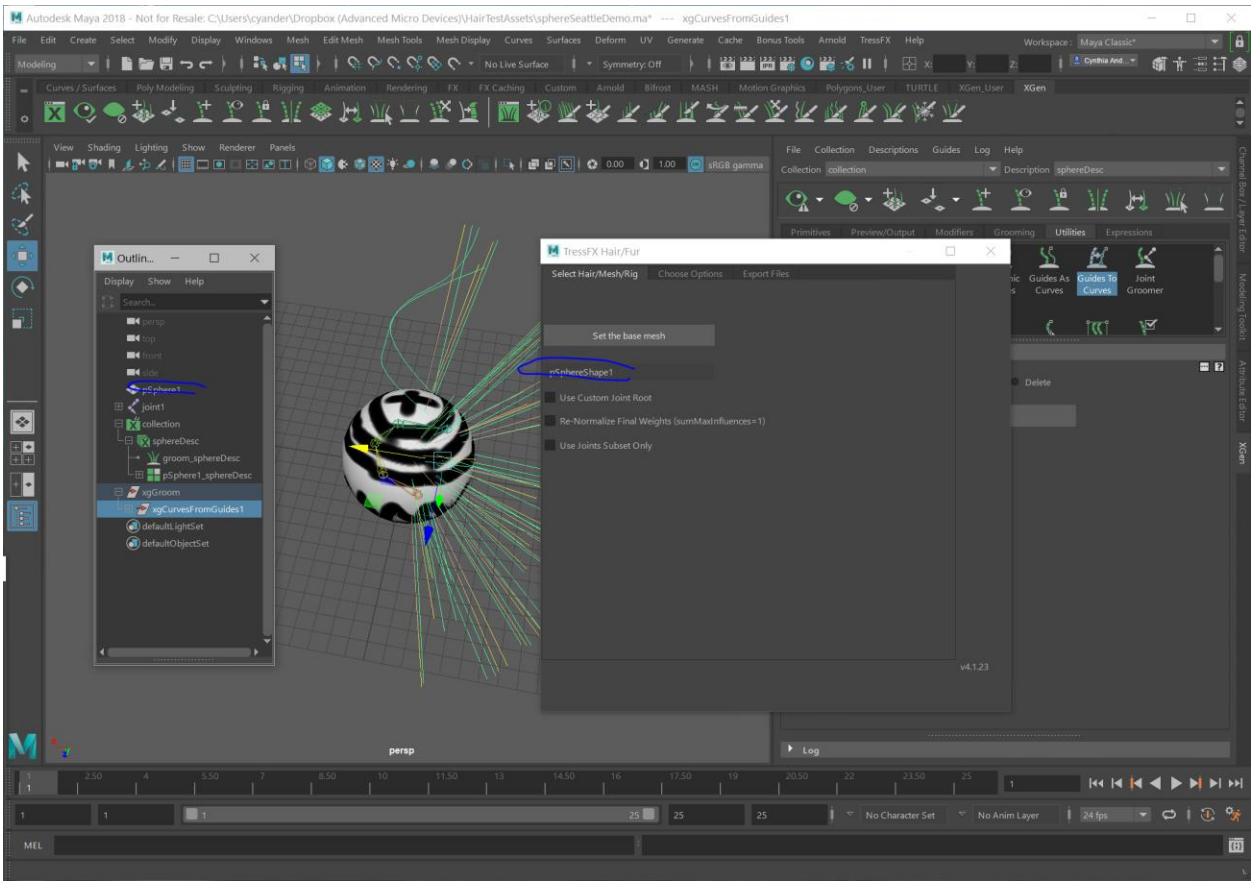
Go ahead and **Export**. You will be asked for a name and a file location. Try to keep all your files together. So I would recommend having a folder that contains your Maya file.

The TFXMESH file has now been exporter. Close the TressFX Collision dialog.

Step 9

Open the menu item: **TressFX→Export Hair/Fur**

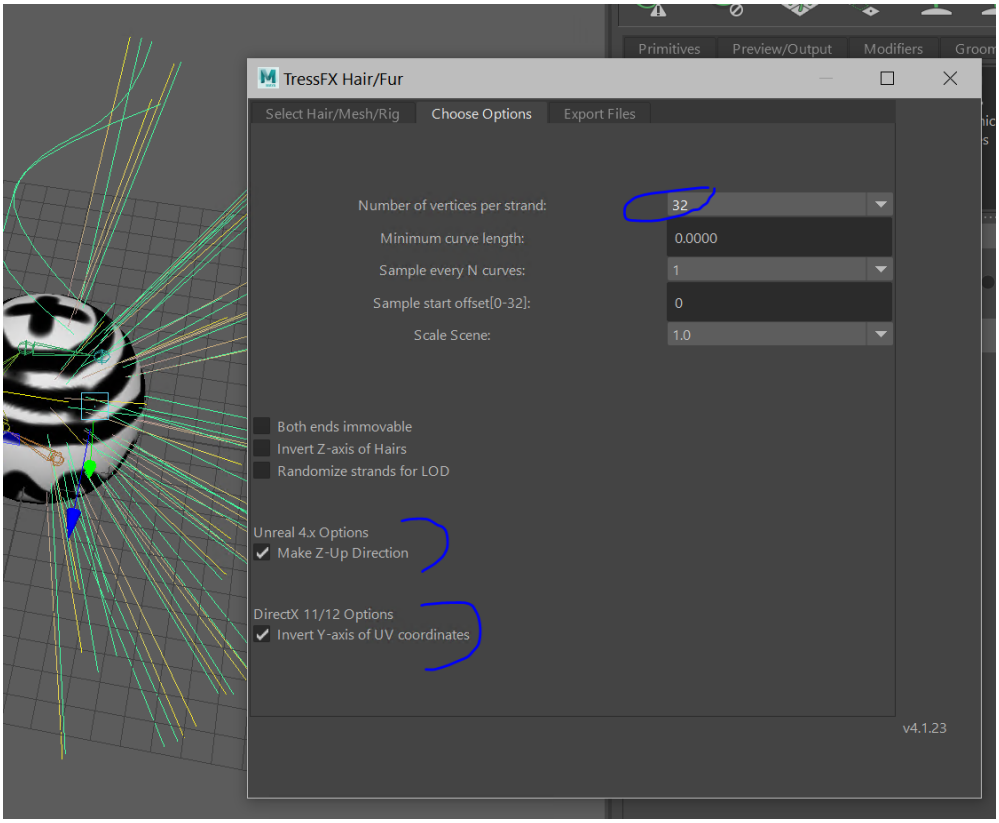
Again in the Outliner, click the sphere mesh. And if the TressFX Hair/Fur dialog, click **Set the base mesh**.



In the TressFX Hair/Fur dialog, in all the three tabs, keep most of the defaults. For this example, set the **Number of vertices per strand** to 32 (Choose the Options tab). Leave the **Minimum curve length** at 0.0000.

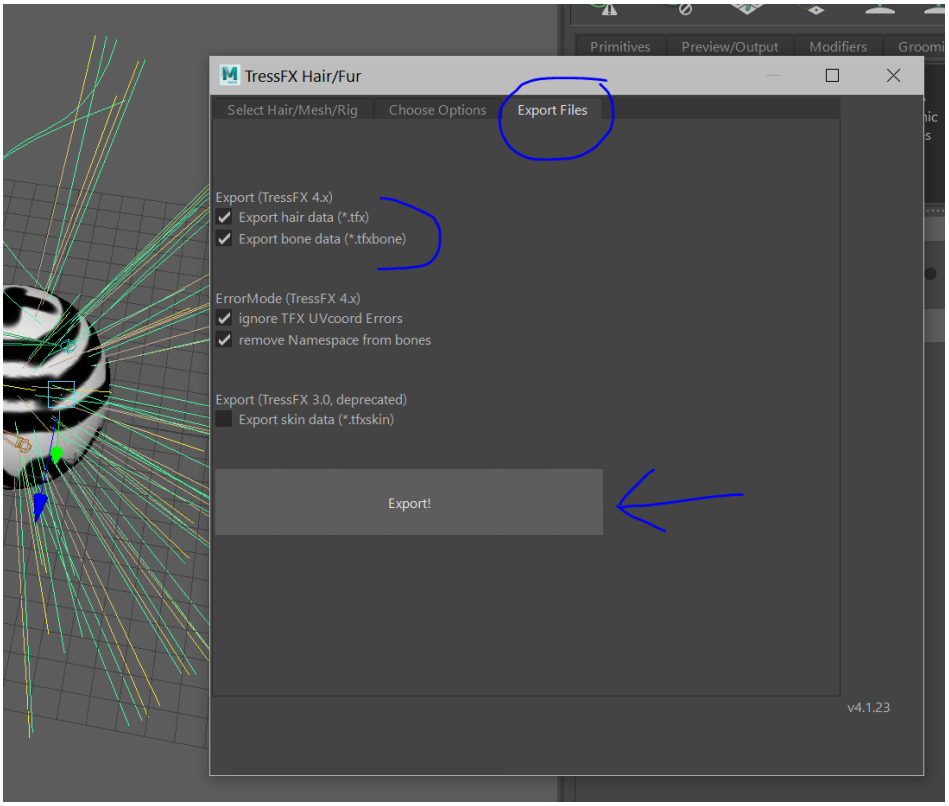
For Unreal: Make sure the option to have Z as the UP axis is selected. Unreal requires this. (It should be a default setting.) For engines that use Y as the UP axis, uncheck this box.

If exporting to Windows® (Cauldron and Unreal on Windows), make sure the DirectX® options are selected as well.



Click the checkbox **Export bone data (*.tfxbone)**. This is the third tab in the dialog.

Both **Export hair data (*.tfx)** and **Export bone data (*.tfxbone)** should be selected. We are exporting both.



Note: If you needed/wanted to export more hair groupings on the same mesh (with the same bound rig), for now, you will need to re-export the bone data. This is because the current Unreal TressFX build looks for a TFXBONE file that has the same name as the specified TFX file. (You could just copy your TFXBONE file and rename it, but just keep in mind the current naming requirements for a TFX/TFXBONE set.)

Important: You need to select the curves that you want to export. You are selecting the curves that you **converted** from grooming splines into NURBS curves (splines). In other words, the curves that are under **xgGroom**.

Note: You should be able to just select the hair group (or groups). If that does not seem to get all the hair (when importing and inspecting), then try shift-selecting all the individual splines. The exporter is designed to do a recursive search for splines (nurb curves), but it relies on the OpenMaya API.

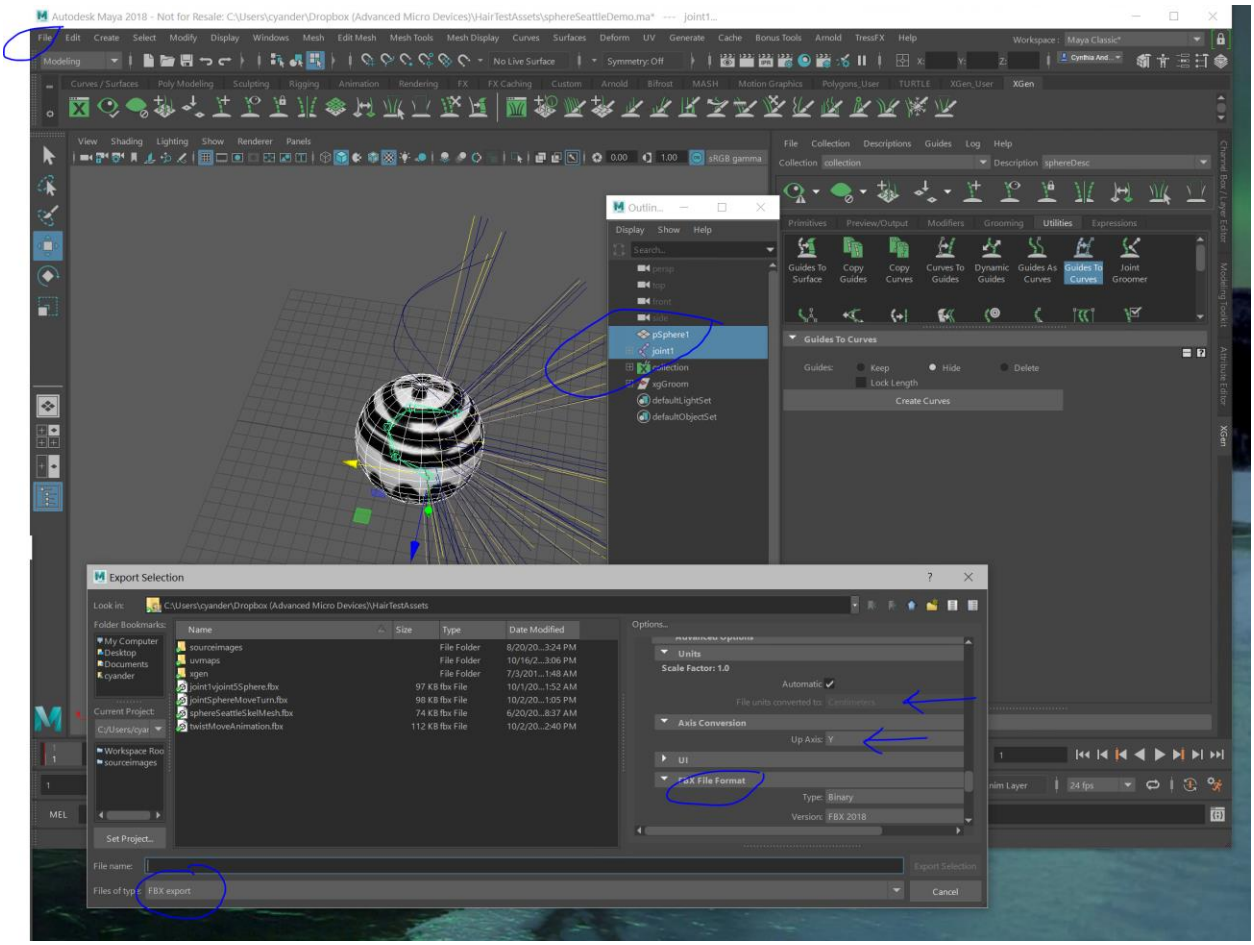
After you select the curves, and you have set your options in the dialog, go ahead and click **Export!**

Again, you will be asked for a filename and location for both the TFX and the TFXBONE files.

You have now exported your TFX (.tfx), TFXBONE (.tfxbone) and TFXMESH Collision (.tfxmesh) files.

Step 10

Export your mesh and rig as an FBX file. Make sure (if using Unreal) that you set the units value to 1 unit = 1 centimeter.



And make sure that your Bind Pose is in the frame 0 slot of your animation.

You are now ready to import your FBX and TressFX files into either Cauldron or Unreal, and to create a TressFX aware skeletal mesh.

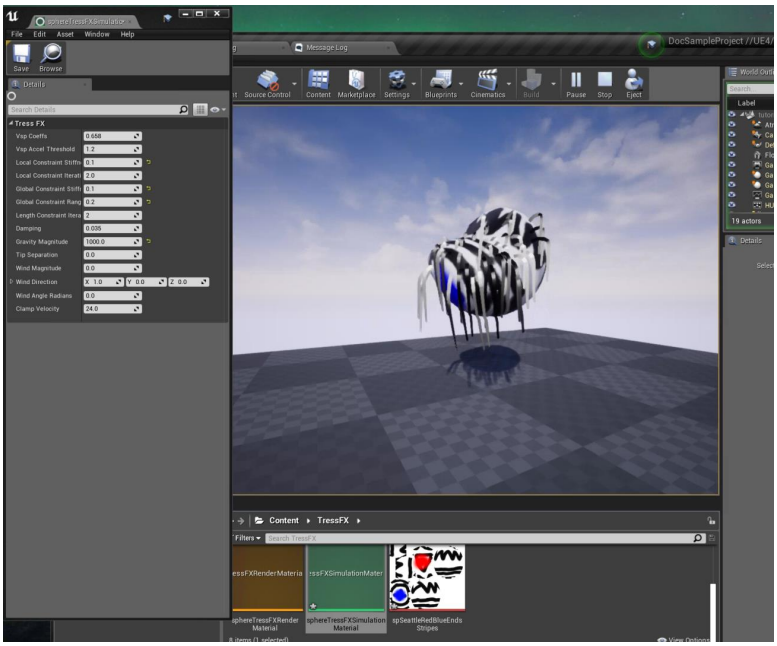
See the quick tutorial on how to create a TressFX aware skeletal mesh in related links.

Note: Tutorial results may vary and there may be other, easier ways to make sure your Bind Pose is exported out of Maya and into Unreal. Note that the exporter wants you to model the hair in Bind Pose, and that not doing that, and not having the Bind Pose imported as your skeletal mesh in Unreal, will lead to problems with hair offsetting (very dramatic errors in some cases.) So the current rule is use the Bind Pose, make the Bind Pose as close to 0,0,0 as you can in Maya, and export the Bind Pose to Unreal (if you make it frame 0, you can easily set an option during import to take T0 as your Bind Pose in Unreal).

Adding TressFX 4 (hair/fur) to skeletal meshes (TressFXComponent to UE4 Blueprint)

Introduction

The following short tutorial walks you through the process of adding a skeletal mesh that has a TressFX component and TressFX materials (Render and Simulation). In other words, adding hair or fur to a skeletal mesh in UE4.



Related Links:

Creating and Exporting TressFX 4 Hair from Maya
A Quick Tutorial on Creating a Basic Skeletal Mesh in Maya
Adding Collision to TressFX Hair Components (Unreal Blueprint)

Requirements

Microsoft® Visual Studio 2017 Community Edition or higher or equivalent (currently you need to build the project and engine to run).

The Unreal TressFX 4 engine build (available on GitHub, a branch of Unreal. You will need to have signed the Epic Games EULA and followed their processes in order to get access to the Unreal GitHub site. TressFX/Unreal is one branch within that site.)

You should have built the TressFX/Unreal branch and have a working engine build, and Unreal editor.

TressFX 4 .tfx, .tfxbone and .tfxmesh files (exported from Maya)**

The FBX containing the mesh and rig (and animations) from Maya — the mesh and skeleton rig must be the ones used to build the TressFX hair files.***

**See the tutorial on creating hair in Maya (xGen) and exporting the files.

***The mesh must be rigged with a skeleton. TressFX 4 binds to the bones (influences), up to four. So the mesh must be a rigged mesh (skeletal mesh).

Let's Begin!

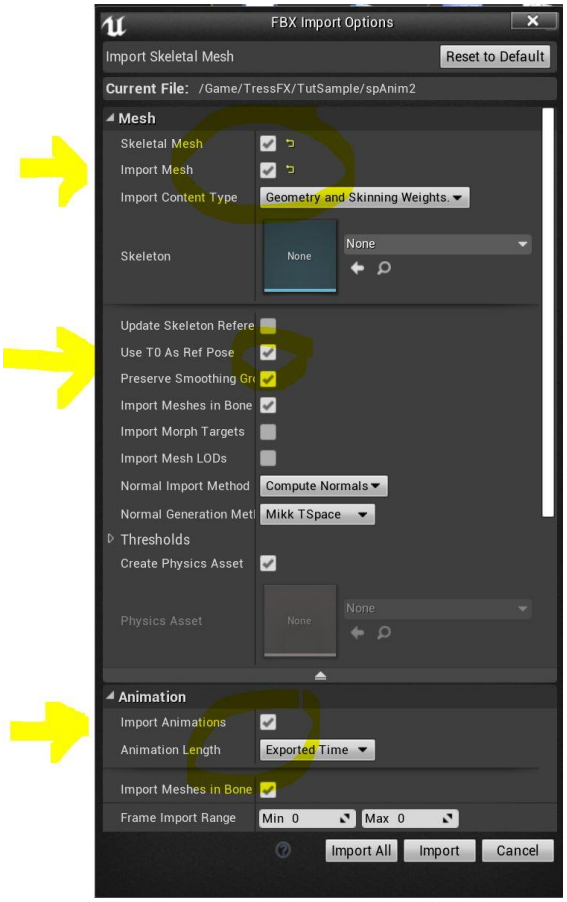
Step 1

Start up the Engine Editor (either by directly starting the UE4Editor or launching it from Visual Studio). Load your uproject or a new project.

When the editor is running and your project loaded, drag in your FBX that contains your mesh and rig. Optionally, it also contains your animations.

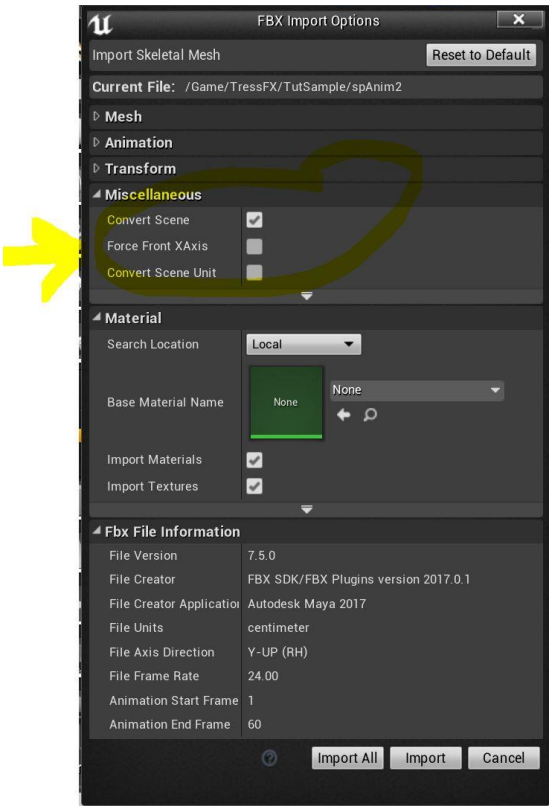
You can have the FBX already in the folder and when the Editor starts it will automatically start to load the FBX, but doing it the manual way gives you more control and is the preferred way for this tutorial.

When loading the FBX, you want the mesh and the skeleton. If you are reloading this same mesh/skeleton, you can select the skeleton previously imported instead of loading it as a new skeleton.



Using T0 as the bind pose is also typical for FBX importing. Make sure your Bind Pose is in the first frame of the animation (i.e. frame 0).

Make sure you have Scene Convert selected (a checkbox). You should have exported your FBX with 1 unit = 1 centimeter (which is usually the default setting for Maya FBX export). But scene convert handles the change of a typical Y-up axis to Unreal's Z-up axis setup.



You can also load Animations if you wish, or load them later in their own FBXs (see Unreal documentation on FBX importing for more information. You will need to have loaded your mesh/rig first.)

Note: If you want to keep your TFX asset files located with the uproject, it is recommended you create a parallel directory to Content and place them there.

Note: For TressFX items, you may want to create a new sub-directory for the incoming uassets you will be creating using blueprints or creating on import of a TFX file.

Step 2

The editor module of TressFX is not part of the regular engine, so you must add it to any uproject that aims to import TFX files or create TressFX Render/Simulation materials.

Close down your uproject and open the .uproject itself in a text editor. Add the following to your modules list (or add the Modules list, if you don't have other modules yet).

```
"Modules": [
      {
            "Name": "TressFXEditor",
            "Type": "Editor",
            "LoadingPhase": "Default"
      }
]
```

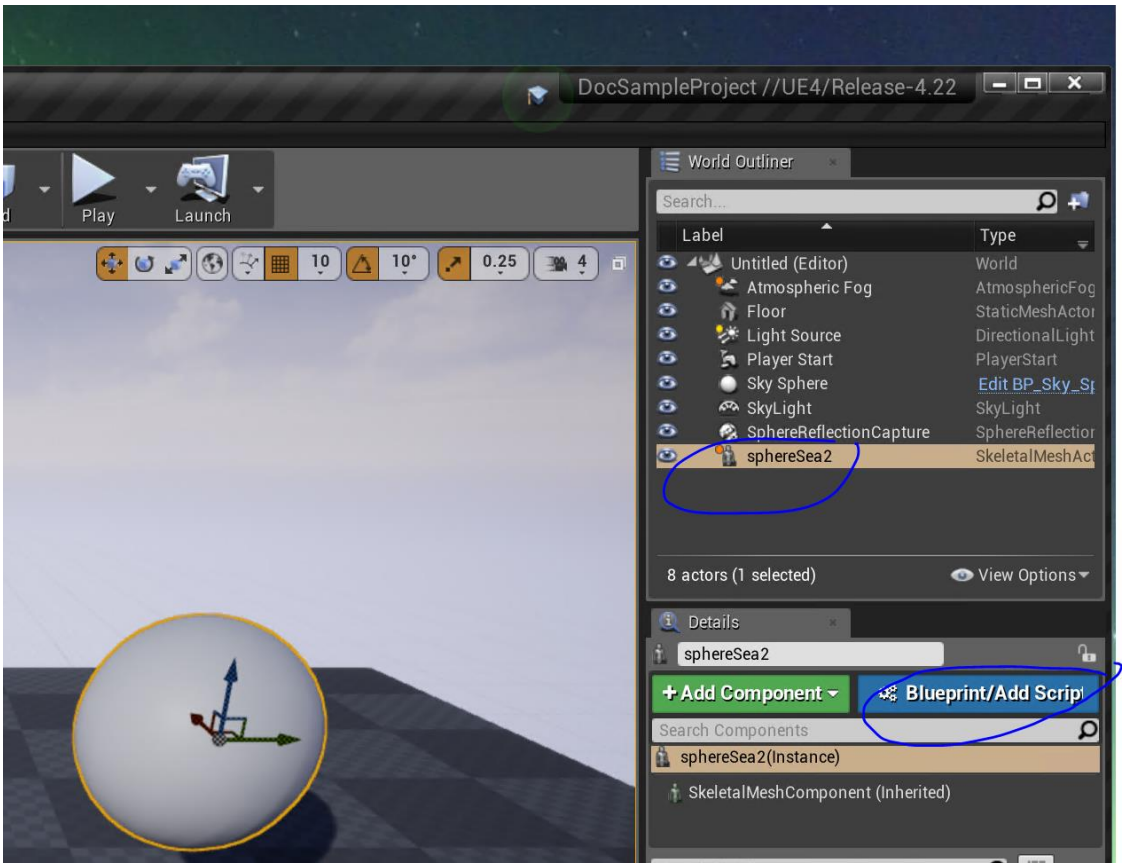
You will have a uproject that will look similar to the following (remember this is just a basic project).

```
{
      "FileVersion": 3,
      "EngineAssociation": "{EB2299E7-4574-
E5F6-77A8-4C8BD1EE28F9}",
      "Category": "",
      "Description": "",
      "Modules": [
            {
                  "Name": "TressFXEditor",
                  "Type": "Editor",
                  "LoadingPhase": "Default"
            }
      ]
}
```

Step 3

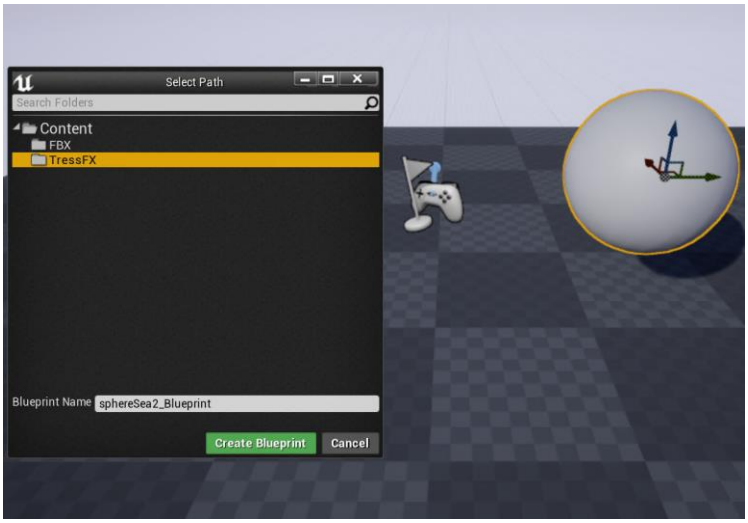
For this tutorial, we are going to set up the skeletal model (actor) first, turn it into a blueprint, then create TressFX components and materials, and then 'wire up' our blueprint with TressFX components and materials.

Drag your skeletal mesh into the scene/level. Select your mesh in the World Outliner Panel.



Then in the Details Panel (which shows the details of the mesh) click the **Blueprint/Add Script** button to turn your mesh into a Blueprint.

Select the same directory as your other TressFX assets as the location of the blueprint.

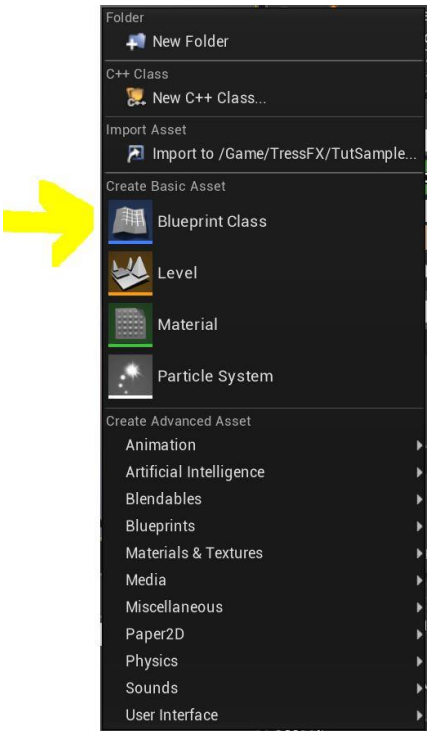


Then, in the Content Browser window, in your TressFX assets directory, change the name of the blueprint if you wish. For example, MeshTitle_BP where MeshTitle is the name of your mesh, i.e. MyModel or RatBoy or HB.

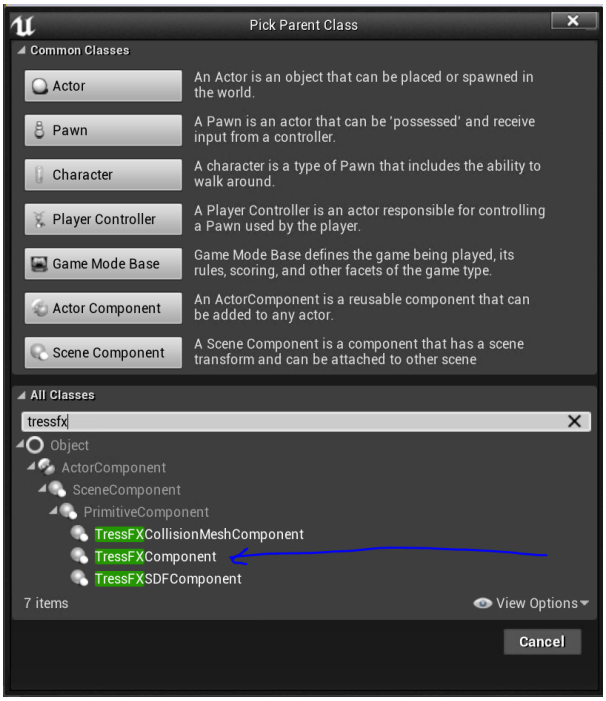
Step 4

In your TressFX assets directory, in the Content Browser window, right-click in an empty area. A window will appear.

Under Create Basic Asset heading, click Blueprint Class.



The following dialog will appear.

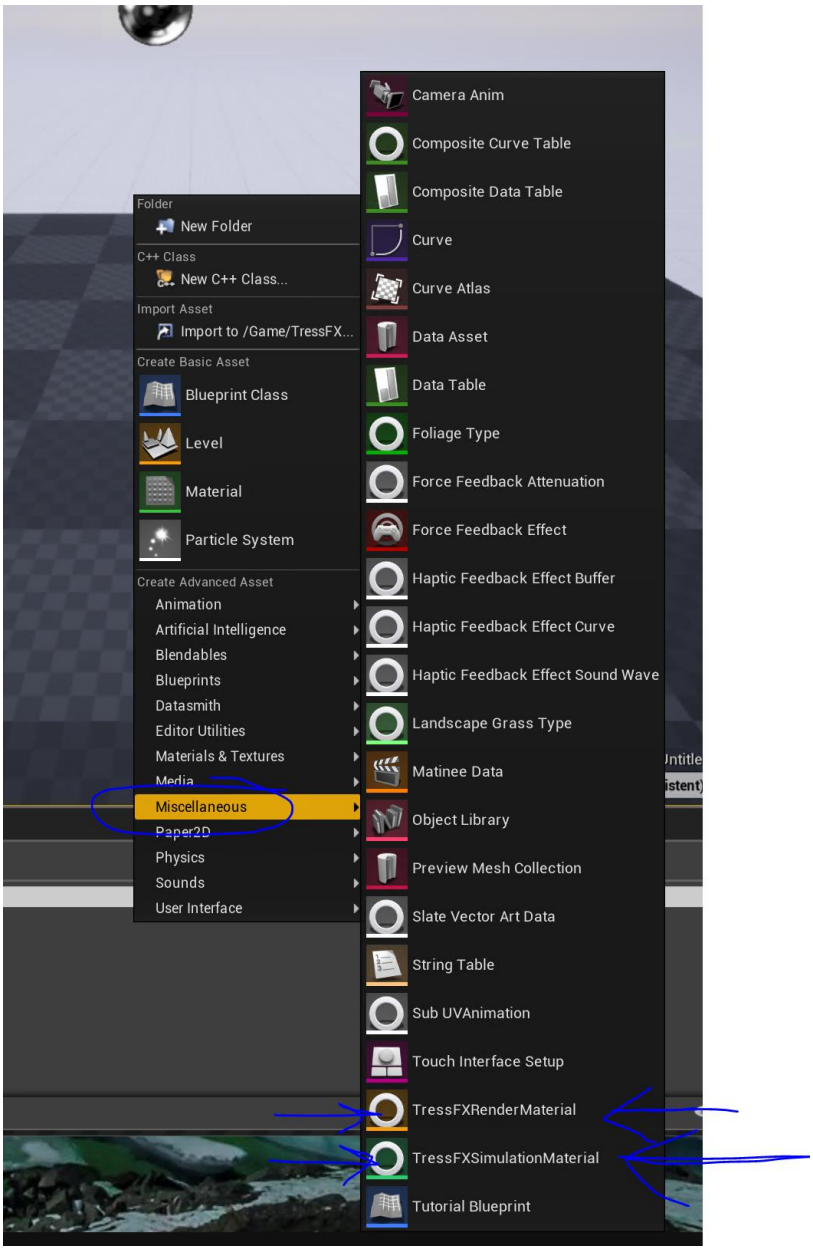


Instead of picking a parent class from the top. Type TressFXComponent in the search panel. You should see a TressFX Component (and possibly children of that class) pop up as you type. Select the TressFX Component (not a child, unless you know what you are doing). This creates another blueprint. Rename this MeshTitle_TressFXComp to make it easier to distinguish from your skeletal mesh's main blueprint.

The other two TressFX component types (CollisionMesh and SDF) are for adding collision ability to your TressFX aware skeletal mesh. There is a separate tutorial for adding a collision mesh and SDF field.

We will also go ahead and create a TressFX Render Material and a TressFX Simulation Material at this time.

Right-click again in the Content Browser. This time choose Miscellaneous and scroll down until you see TressFXRenderMaterial and TressFXSimulationMaterial.



Select TressFXRenderMaterial and it creates a material, name it appropriately.

Repeat the right-click and create a TressFXSimulationMaterial as well.

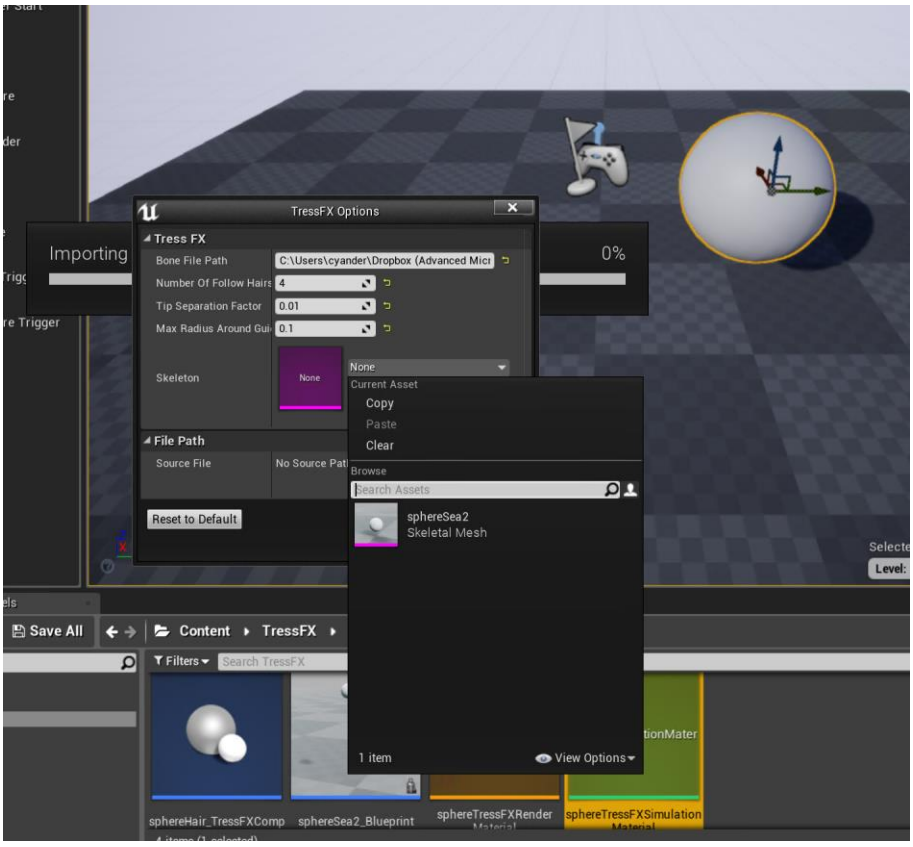
Note: If you do not see these Materials under Miscellaneous, then either you do not have your .uproject set to include the TressFX Editor module, or you are not running the TressFX/Unreal Engine. If you are sure that is not the issue, double-check the .uproject. Sometimes, using text editors leaves unwanted, hidden text in the file and this might be preventing the uproject module information from being read correctly.

Step 5

You can import TFX and TFXMESH files into the Unreal Editor as you would other files (for example, FBX). Either click **Import** in the Content Browser or simply drag and drop the TFX or TFXMESH file you wish to import into the Content Browser directory you want.

TFXBONE files should typically have the same name as their TFX companion file. This helps confirm that the TFX for a particular skeleton matches the TFXBONE file if both are generated at the same time in Maya. A TFX file should always be generated with a TFXBONE file to match, unless you are sure that the TFX file you are generating is a perfect match for a previously generated TFXBONE file.

Drag in your TFX file into the directory you want in the Content Browser.



You will see a dialog pop up.

From the dropdown, select the **skeleton** that you used to create this file. You should have already imported the skeleton as an FBX.

All of these options happen at creation time, not runtime. Currently, we create followhairs + guidehair HairAssets during import. This may change in the future to allow for more flexibility.

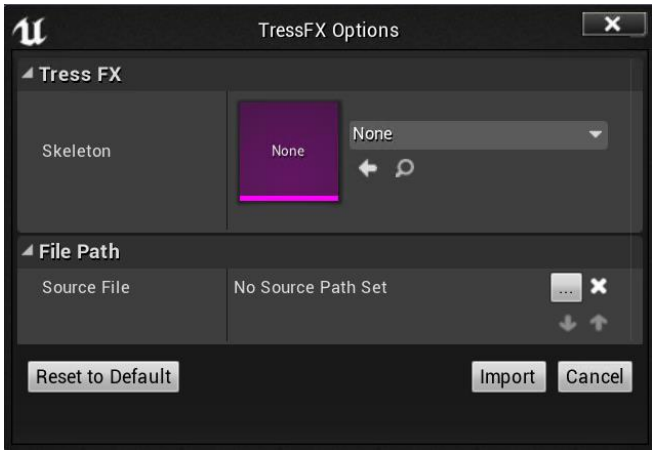
- The **Bone File Path** is currently set to the same directory as the TFX, with the name of the TFXBONE file being the same as the TFX (minus the file extension change). You can use a different name for the TFXMESH, in which case, you would change it here.
- The **Number of Follow Hairs** dictates how many follow hairs will be generated (hairs that 'follow' the imported guide hairs) per guide hair. Typically, a normal range is 0 to 32. Try to keep follow hairs a power of 2.
- The **Tip Separation Factor** dictates how much the tip of each follow hair should separate itself from the other follow hairs. Think of it the same as when you see hair separate out because of static electricity. You can also use this value to indicate that you want the tips close together or always far apart.
- The **Max Radius Around Guide Hair** option disperses the follow hairs randomly in a circle around the guide hair. You specify the radius of that circle here. A normal value is 0-10, although 0-1 generally gives the best and the most realistic results (see the Note below). If you think about it, you will see this is one way to create clumps of hair that start from a spherical radius around a guide hair, then taper down to a single point — rather like wet hair clumps.

Note: TressFX 4.1 still uses the radius method for distributing follow hairs. Follow hairs are distributed in a 3D radius around the guide hair, therefore, for large radii, it is very likely you will get 'floating' hairs above the skin, and embedded hairs below it. This is a current, known issue.

Now you will create a Collision Mesh Asset for your model, this allows the hair to interact with the model. In other words, it will collide with the collision meshes rather than penetrate the mesh.

Note: We will not be hooking up the collision mesh in this tutorial, but it's good to go through the process of importing the collision mesh that you created and exported in Maya.

Drag your TFXMESH file into the directory in the Content Browser. You will see the following dialog box.



From the dropdown, select the skeleton that you used to create this file. You should have already imported the skeleton as an FBX.

You can have multiple collision meshes per HairAsset (TFX/TFXBONE). For example, you can have a collision mesh for the body of a character, as well as collision meshes for the hands or other objects attached to the character.

To activate Collisions, you need to create a construction blueprint as part of your main skeletal mesh (actor) blueprint. Another tutorial will walk you through this. It is a more complicated process than simply adding a collision mesh to the component, but in keeping collision meshes and SDF fields separated, and as blueprint components, a great deal of power and flexibility in the use of collision mesh/SDF fields is possible – which may include better overall performance since the resolution of the collision mesh and SDF field can be tailored to the required situation.

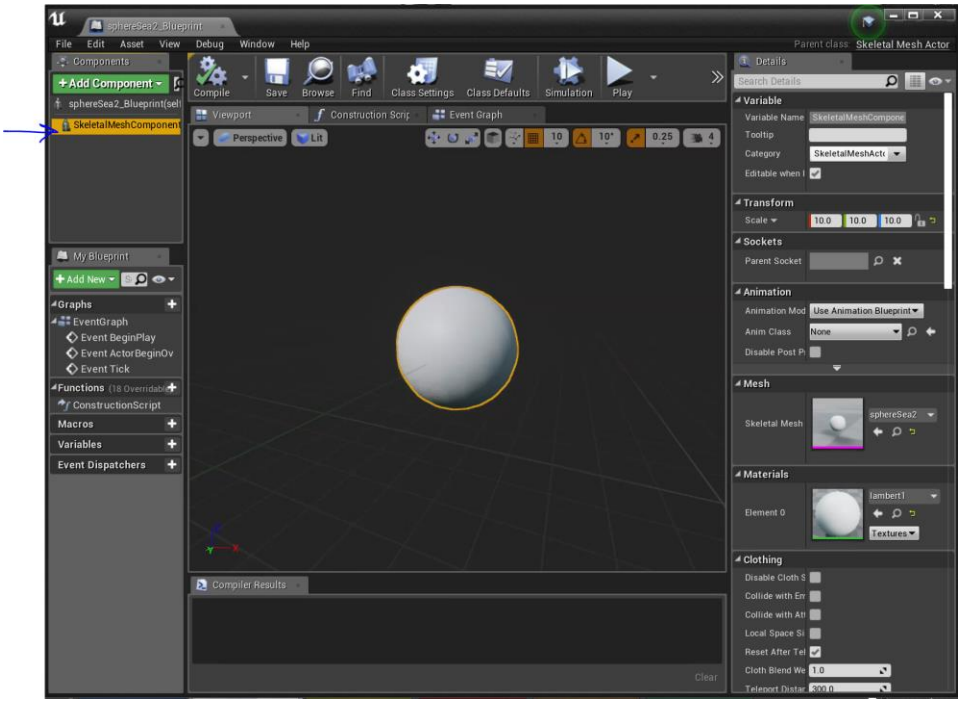
This is also why many shader parameters have been put into Render and Simulation materials, versus staying directly on a TressFXComponent. Parameters on the Render and Simulation materials are likely to be used and reused by many TressFXComponents (hair assets) on a single skeletal mesh (or across actors), so it made sense to put them into materials – which are then referenced by the TressFXComponent.

So next, we will be attaching the imported HairAsset (from the TFX and TFXBONE file combination) to the component.

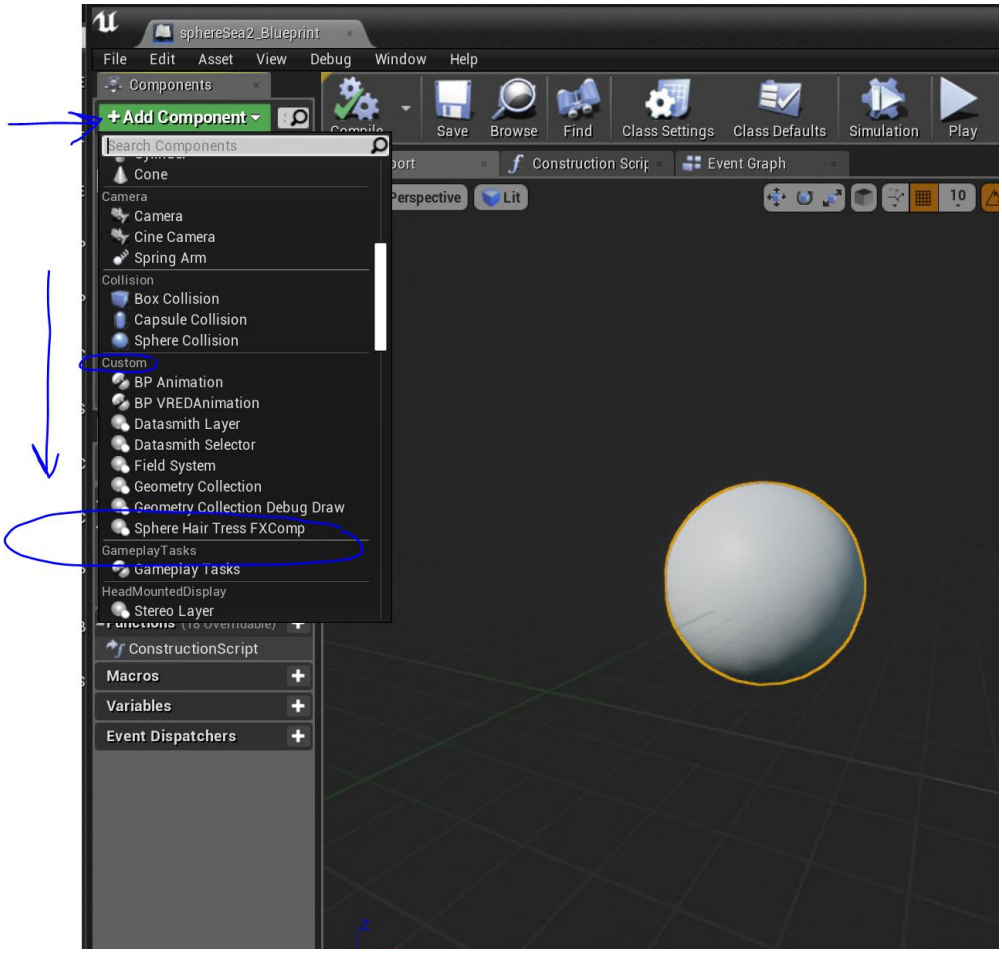
Step 6

In the World Outliner, select your skeletal mesh and click Edit MeshTitle_BP (your chosen blueprint name). This will bring up the Blueprint editor. Each Tab at the top of the editor is a blueprint. You should see your blueprint among them (or the only one if the editor was previously closed.) You can see your skeletal mesh in the Viewport tab in the main display area.

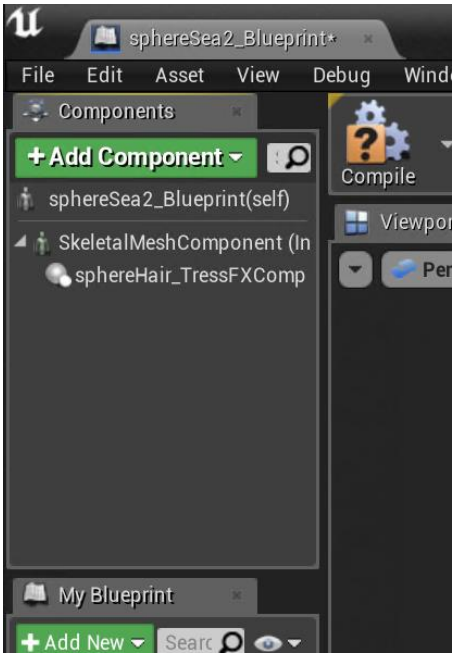
In the Components panel, select your SkeletalMeshComponent. ***You want the component, not the parent which is the blueprint itself.***



After ***selecting*** the *SkeletalMeshComponent*, click the green **AddComponent** button.

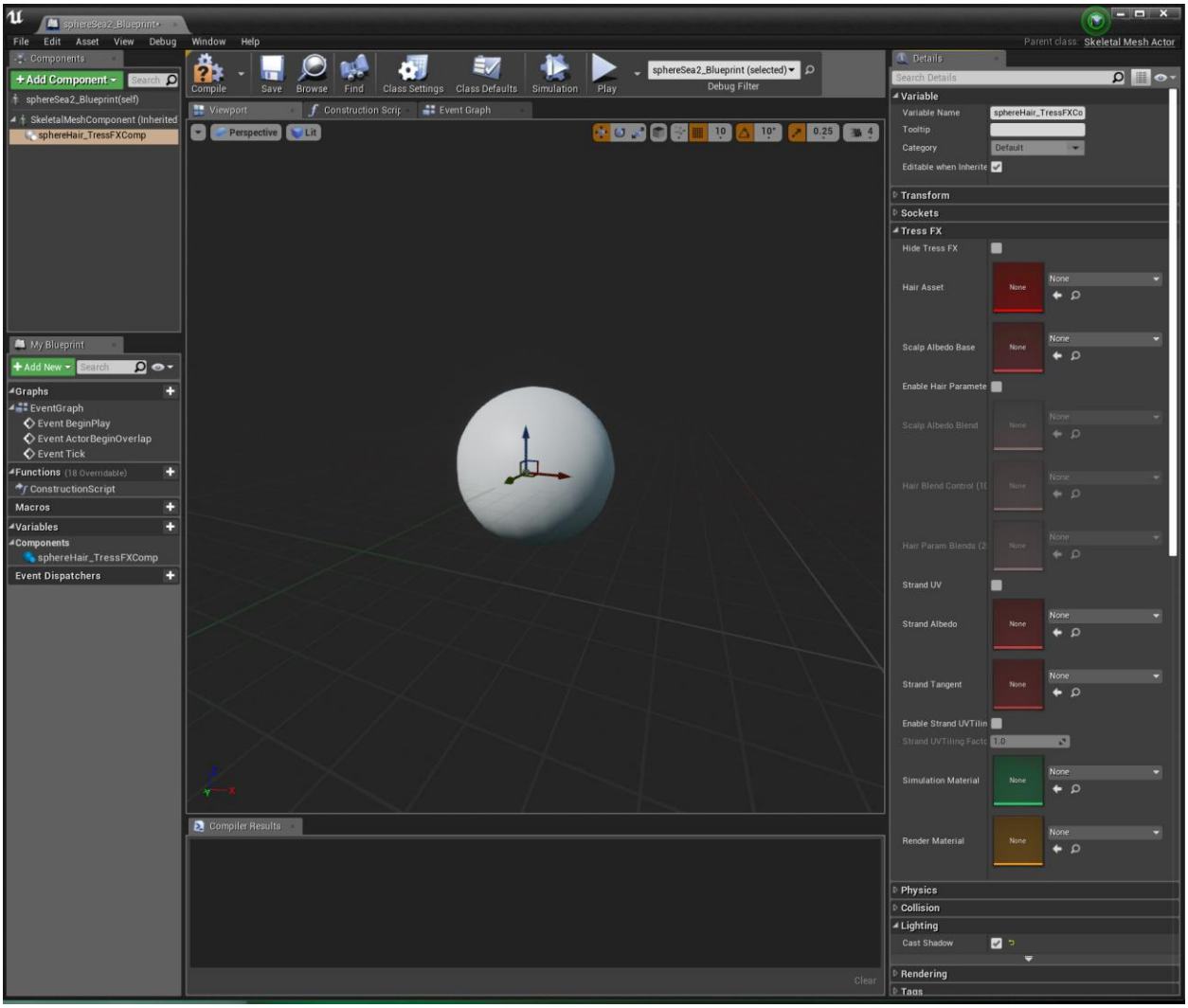


Scroll down the list to the *Custom* section and you should see the TressFXComponent that you created earlier. (i.e. your MeshTitle_TressFXComp but without the underscores. Here, it is *Sphere Hair Tress FXComp.*) Select it.



It should now be parented to the skeletal mesh component.

Select the TressFX Component to see the Details Panel for it.



The TressFX section of the details has many different settings, including checkboxes to turn on Hair Parameter Blending and/or StrandUV, which are beyond the scope of this tutorial. Please see the specific information about these features in other parts of this document.

For now, you will just add your hair asset as well as the Render and Simulation materials you created. The collision mesh you imported needs to be wired up to the SDF component and is in a separate tutorial.

Drag your HairAsset to the **Hair Asset** element (or select it from the drop-down).

Drag your Render material and drop it on the Render Material element (or use the drop-down). Do the same for your Simulation material.

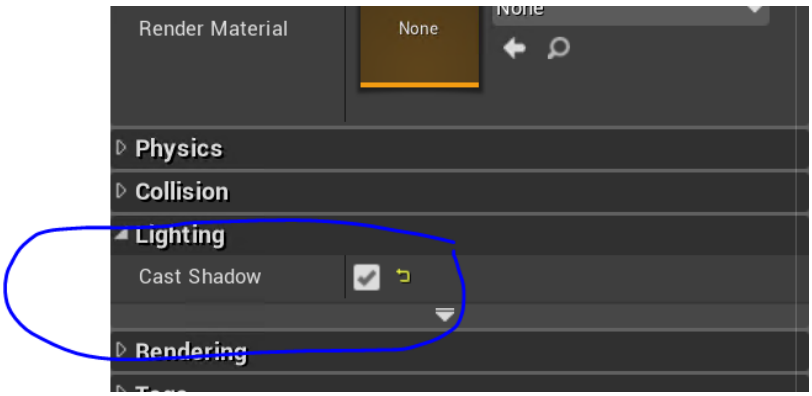
The **Scalp Albedo Base** takes a texture. TressFX will use the UV mapping of the original mesh to determine what texels to use from the texture. This is where you would have a UV mapping of your character, and where ever there is hair attached, that color will dictate the diffuse color of the hair in addition to effects from the other rendering features (Hair Parameter Blending, ScalpUV and the Rendering Material.

If you want, go ahead and drag in a texture to use for the hair, or finish adding a material to your skeletal mesh Actor, and use the same UV texture as on the Actor, also in **Scalp Albedo Base**.

For more information on all these settings, check the information elsewhere in this document.

Compile and **Save** the blueprint.

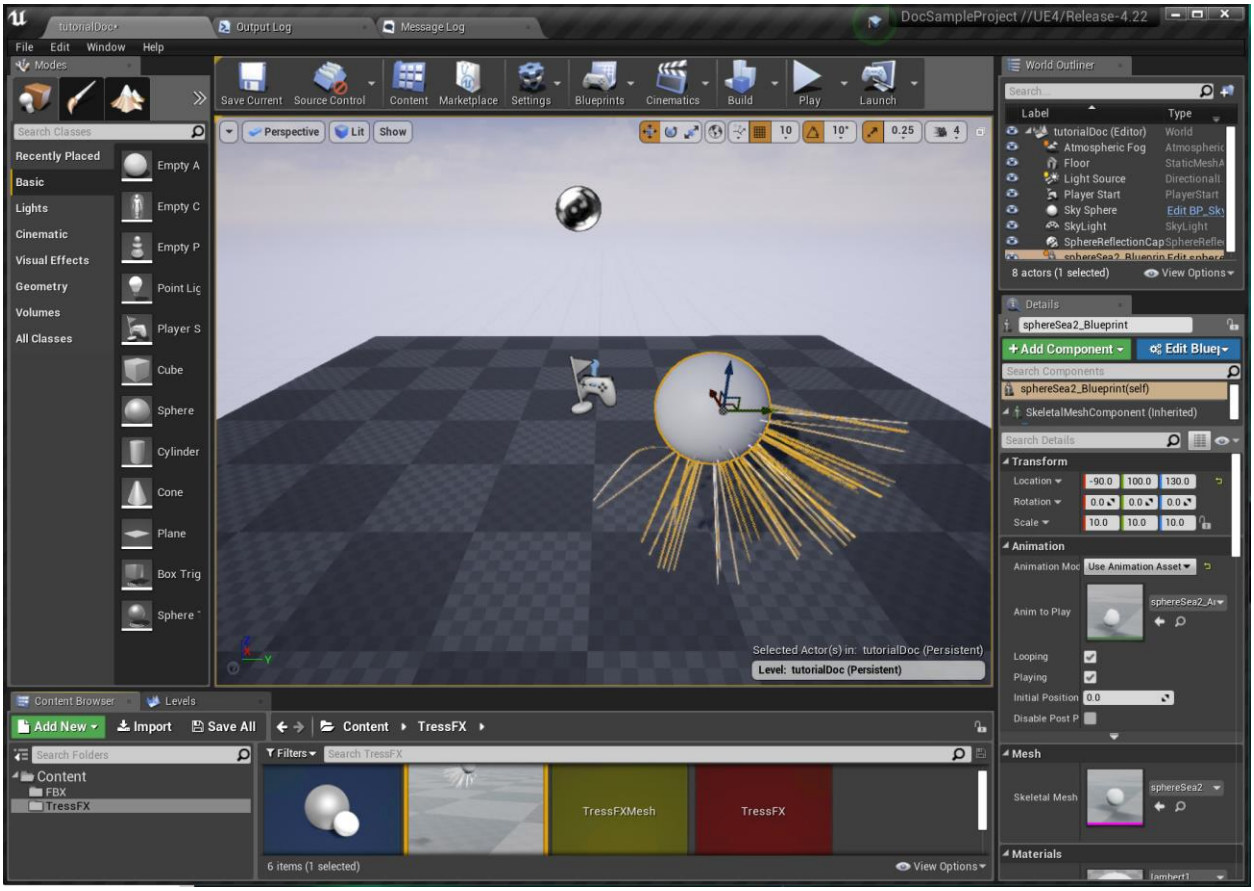
Note: To allow the hair to cast shadows, remember to turn on Cast Shadow from the TressFX Component Details Panel, under Lighting. This is **not** under TressFX subcategory, but in the same place you would typically use if an object casts shadows. For more information, look at the lighting sections in this document.



Note: Like any other Unreal Actors using blueprints, you can have your *default* settings in the blueprint, then *override* them for individual actors in the main editor (under the selected Actor's Details panel in the main editor).

Step 7

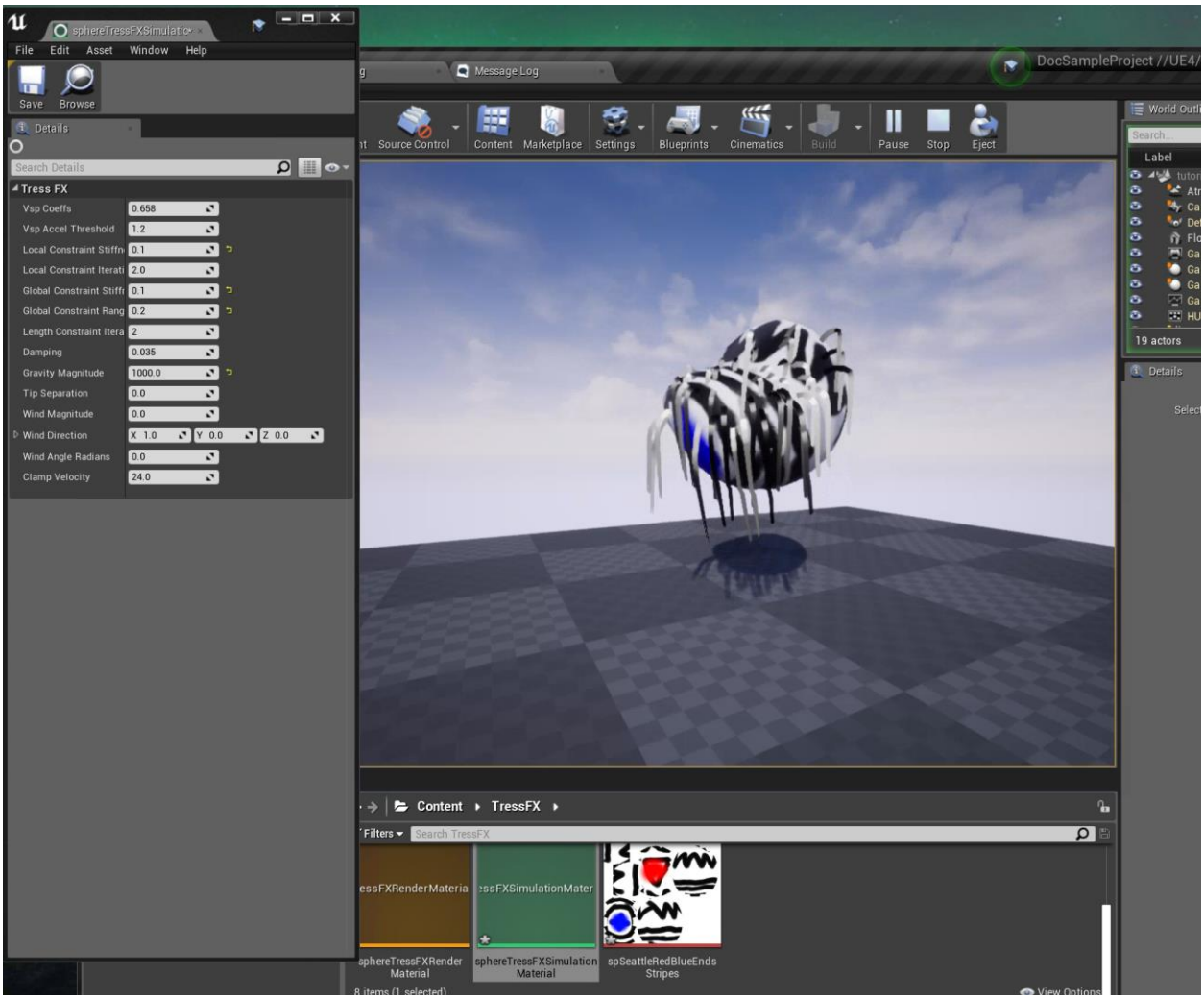
Close or move your blueprint editor to the side so you can see the main editor, and your skeletal mesh. And hit the Play button in the blueprint editor. You can see your blueprint activate. And you can observe what happens in the main editor. Your mesh, if everything was set up correctly (in Maya and in Unreal), should show hair now.



For fun, either in the blueprint editor or on the instance in the scene, click the skeletal mesh component and under Animation in the Details panel, choose **Use Animation Asset**, then choose one of your animations for your skeletal mesh.

You'll likely soon find, as you play with TressFX parameters, that it is easy to get a wide variety of effects, but often difficult to get exactly the effect you want. The TressFX team realizes that the hair needs to be easier to tweak and more realistic. The focus of TressFX 4.1 was on the integration/implementation with Unreal and Cauldron.

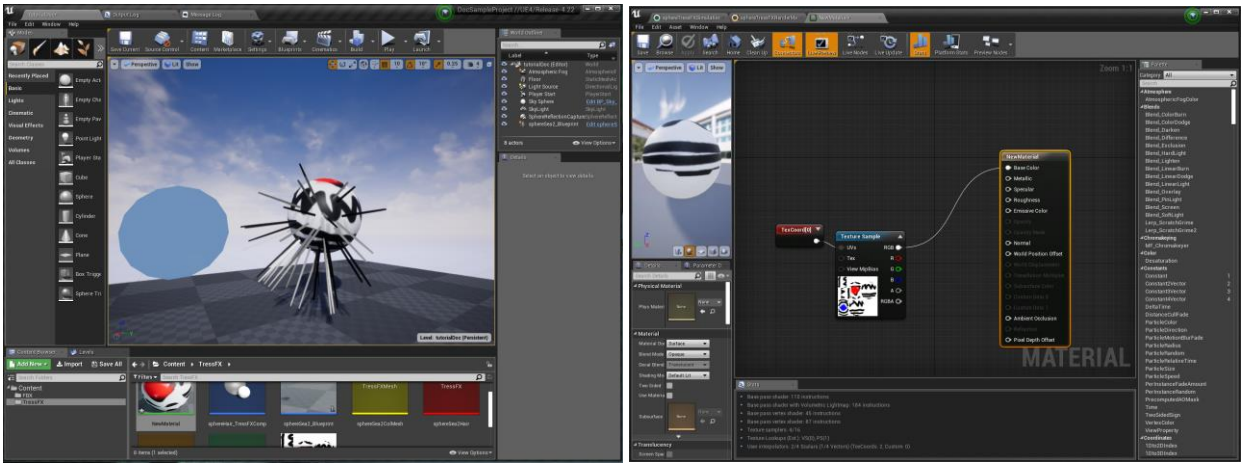
In the following image, the sphere has been UV textured, with the same UV texture used for the hair (Scalp Albedo Base). The fibers are wider to show the colors, which correspond to the UV coordinates. Some in the image may appear gray (versus white) but that is the lighting (and shadows in some cases). Notice that shadow casting is turned on, and you can see the strands' shadows on the ground. You can also see the hair tracking as the sphere animates – twisting and moving. In this case, gravity was turned up to 1000, while local constraint stiffness was set low (.1) as was global constraint stiffness and range (0.1, 0.2). If gravity were at 10, the hair would appear instead to pull tighter toward the body (shorten) as the mesh animated. Hair tuning is not straightforward with TressFX 4.x. Hair rendering and simulation is a complex area at best, and making it real time is even more demanding (as is integrating into a commercial engine which has competing demands for processing time).



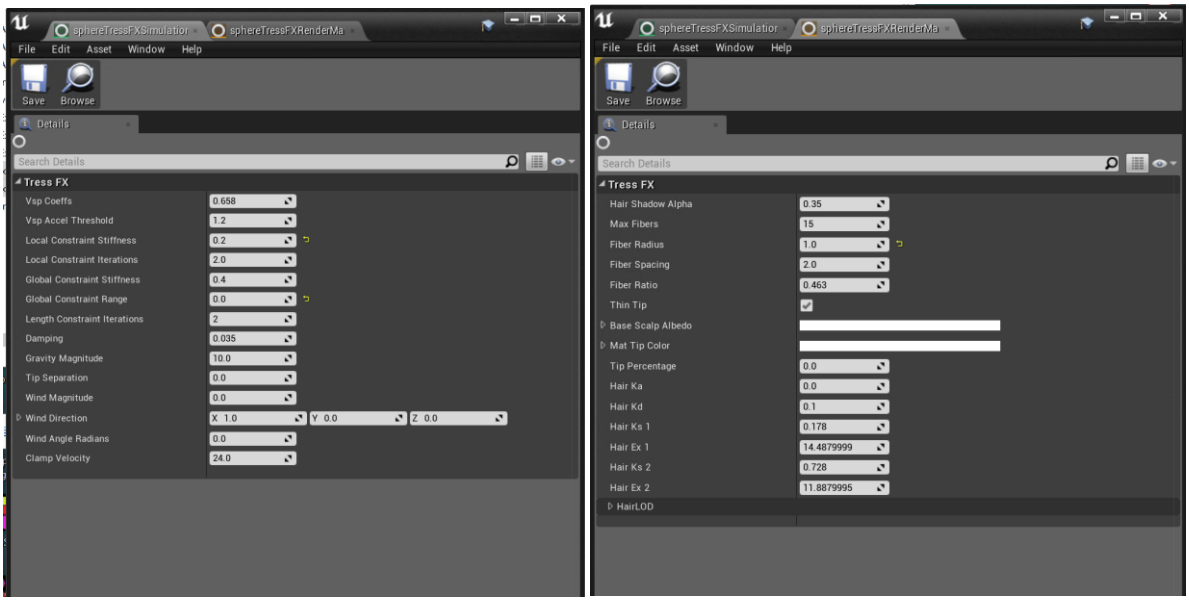
Just for Fun (step)

As a final task, add a material to your imported skeletal mesh Actor, use the same texture for both the body material and for the hair asset (**Scalp Albedo Base**), and change the Simulation Material settings for global and local stiffness around to see the effect, as well as the effects of Fiber Radius on the hair strands (width).

You might even try setting gravity to 1000 (versus 10) to see how that might create better hair length consistency (especially on a model with as few hairs as this tutorial showed). You'll find that the amount of hair, and therefore the demand on the simulation, can have a dramatic effect on how the hair reacts to gravity values, among others.



It's good to let yourself experiment with the settings, even using parameter values that seem impossible in some way, and then try these and other values on other machines with more or less processing power.



Note: If your settings don't seem to take effect immediately in the main editor, try the **Hide TressFX** checkbox (TressFX Details panel) to hide/show the hair, in order to force the main editor to redraw the hair.

Adding Collision to TressFX Hair Components (UE4 Blueprint)

Overview

TressFX Hair Components support collision through Signed Distance Field (SDF). In each frame, the TressFX plugin builds signed distance fields (SDFs) from user-supplied collision meshes and collides each SDF with each connected TressFX Hair Component. This tutorial will show how to add SDF and TressFX Collision components to a blueprint with TressFX Hair Components and how to connect SDF and Collision components together, as well as with Hair components.

Related Links

Creating and Exporting TressFX 4 Hair from Maya
A Quick Tutorial on Creating a Basic Skeletal Mesh in Maya
The TressFX Collision Mesh Component
The TressFX SDF Component

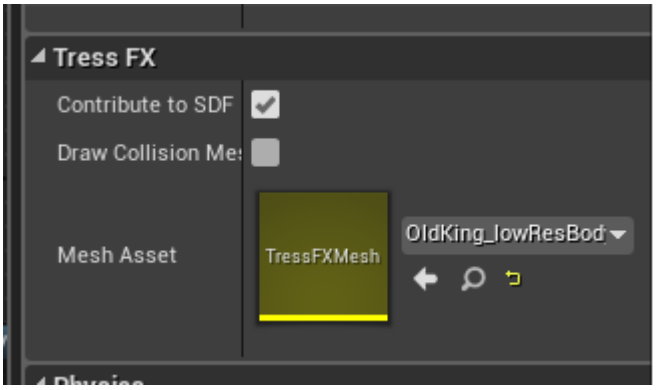
Adding the Needed Components

The first step in adding collision to TressFX Hair Components is to add new components to the actor, as children of the skeletal mesh: BP Collision Mesh Component and BP SDF Component. The BP Collision Mesh Component wraps around a single TressFXMesh asset and represents a single collision-contributing component. Multiple Collision Mesh components can contribute to a single SDF Component. This component builds an SDF of the specified dimension and resolution, adds each associated Collision Mesh to the SDF, and then collides the SDF with each connected hair asset.

Once the components are added, the collision is set up as follows.

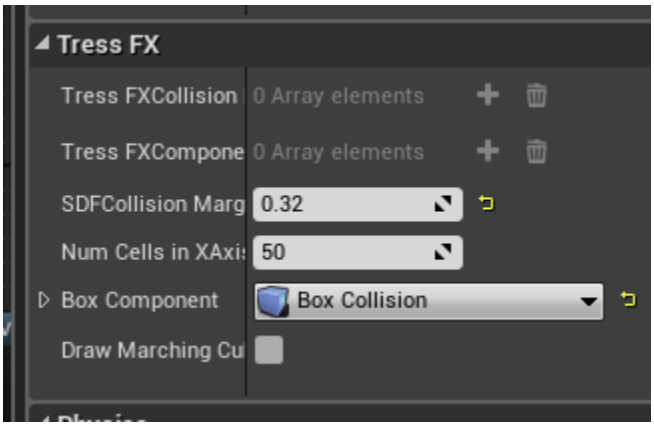
Step 1

In the details of the BP Collision Mesh Component you'll find the **Mesh Asset** property under the **Tress FX** category. This needs to be set to the appropriate TressFXMesh asset for your character. One Collision Mesh component takes a single TressFXMesh.



Step 2

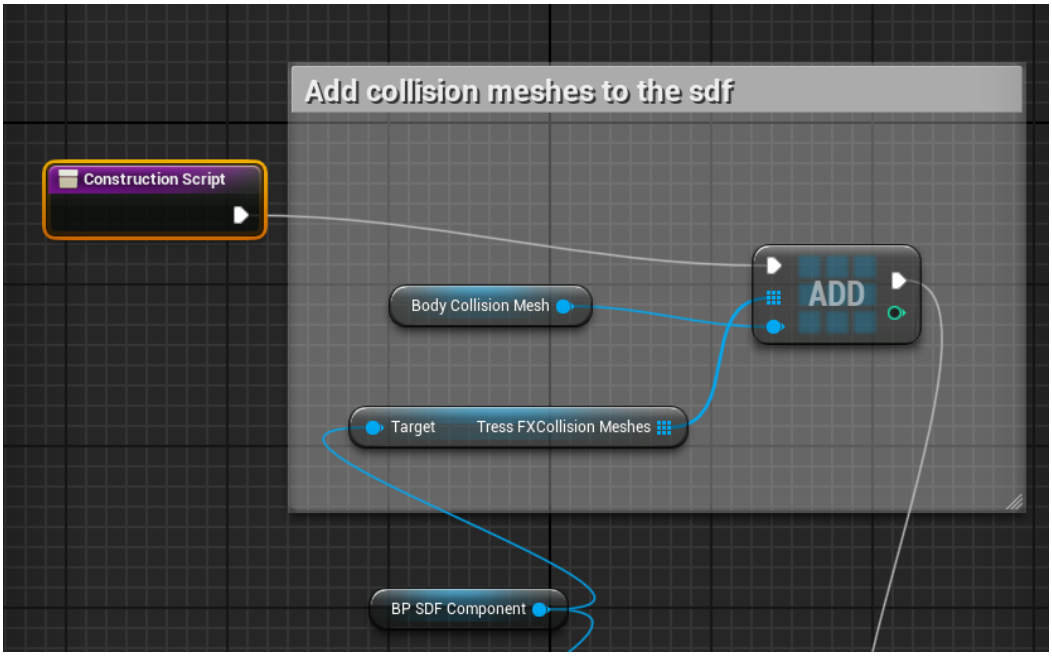
Look over the default settings on the SDF component. The SDF Collision Margin controls the amount of padding that is added around the default bounds. The Num Cells in X Axis controls the resolution of the SDF. The default values are usually fine.



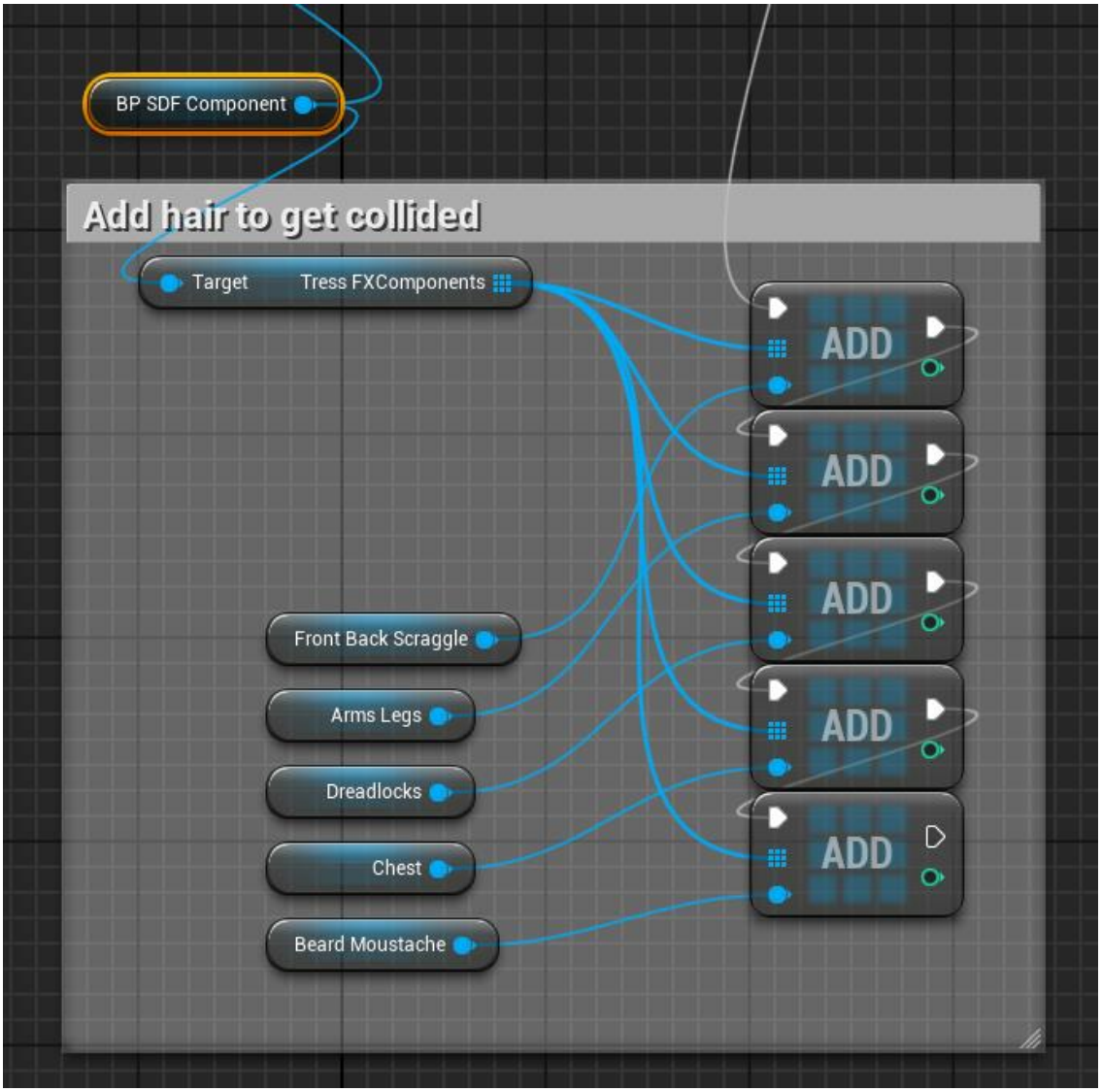
Step 3

Most of the setup work is done within the blueprint's Construction Script. Two array properties on the SDF Component need to be populated with their appropriate components, and in no particular order: the Tress FX Collision Meshes and the Tress FX Components. To get references to these arrays, create a reference to the SDF Component (you can drag the component from the Components Window into the Construction Script Window), drag off the reference's right connector, and let go to open the dialogue to place a new node. Search for both "Get Tress FXCollision Meshes" and "Get Tress FXComponents" individually to get a reference to the arrays. From the array reference, drag off the right connector and release again, this time add an Add node. This is used to add a single component to the array, so we will need an Add for each item that needs to be added to each array. The Tress FX Collision Meshes array is filled with references to Collision Mesh components, the Tress FX Components array is filled with references to Tress FX Components (the hair objects).

The set up will end up looking similar to this:



SDF Component connecting to Collision Mesh



Connecting the SDF with the Hair Assets (TressFX Components)

Each Collision Mesh will contribute to the SDF, and each hair component will collide with the SDF.

Visualizing the Components

Each Collision and SDF component have an option to visualize the component. On the Collision component, the Draw Collision Meshes checkbox will enable the debug draw of the collision mesh in the Viewport and in any level the blueprint is placed. The **Draw Marching Cubes** checkbox on the SDF component enables the marching cubes visualization of the SDF, again in the viewport and in any level the blueprint is placed.

For more information on these settings, see The TressFX Collision Mesh Component and The TressFX SDF Component.

SDF Dimensions

The size of the SDF created by the SDF Component is decided by the bounds of the component's parent, in this case, the skeletal mesh. If the SDF is too large, make sure the SDF Collision Margin isn't too large, and then look into ways to minimize the bounds of the skeletal mesh.

For more information on this setting, see The TressFX SDF Component.

A Quick Tutorial on Creating Collision Meshes for TressFX (using Houdini)

This page gives an overview of good practices for authoring collision meshes for use in TressFX and a simple tutorial of creating a collision mesh from a render mesh using SideFX Houdini.

Collision Meshes

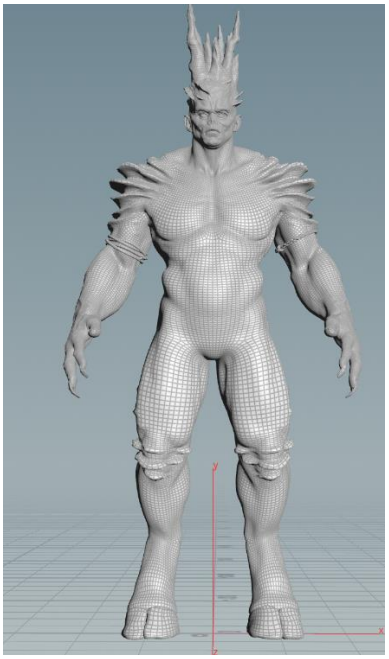
The TressFX collision model uses signed distance fields. A SDF is constructed for each collision mesh every frame. The method for constructing the SDF is: for each voxel in the volume, find the distance to each triangle in the collision mesh and store the shortest as that voxel's value. This means that this operation can be slowed down significantly by a dense volume with a lot of voxels and by a dense collision mesh with a lot of triangles. In order to attempt to improve SDF generation time, you can try reducing the number of voxels in the SDF or authoring a lower resolution collision mesh. One method for procedurally generating a collision mesh from a render mesh is presented below.

Houdini Collision Mesh from Render Mesh

The basic algorithm overview is to convert the render mesh to a volume, and back into triangle mesh which gives more control over mesh topology and vertex count. The mesh can be further reduced afterwards. Houdini can use VDBs to simplify this process.

Step 1

Start with a high-resolution render mesh:



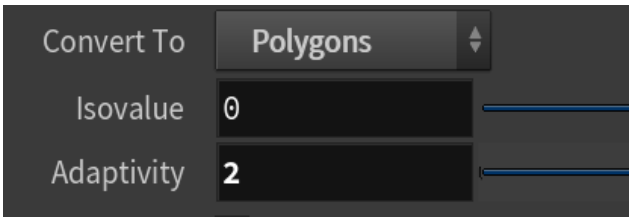
This mesh has 417,504 vertices.

Step 2

Use the **VDB from Polygons** node to create a Distance VDB, i.e. a SDF that is displayed in Houdini as an isosurface:

Step 3

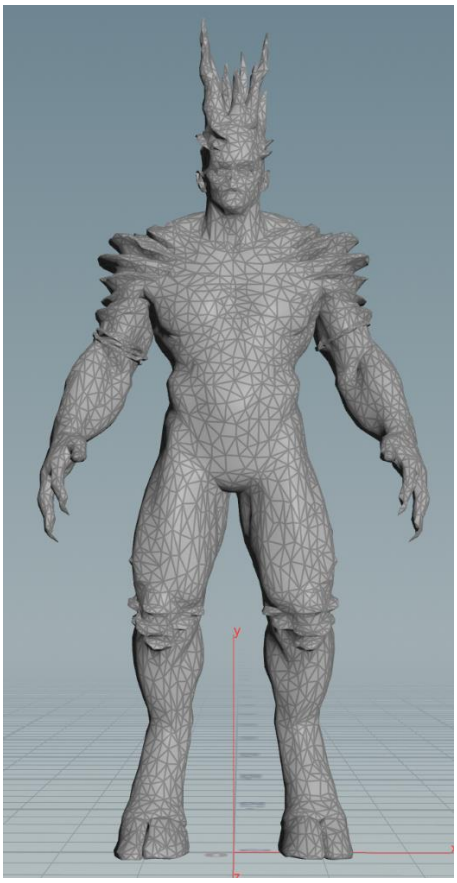
Then, use the **Convert VDB** node to convert the volume back into a triangle mesh. The following two parameters in this node are important in controlling the output geometry.



The **Isovalue** parameter can be used to expand or contract the mesh, useful for creating a collision mesh that is slightly larger than the render mesh. The **Adaptivity** parameter can be used to control the resolution of the output mesh.

Step 4

Often this isn't enough of a reduction, so the **PolyReduce** node can be used to further reduce the mesh. Simply change the **Percent to Keep** number to reduce the mesh.



This mesh has 34,606 vertices.

The mesh is now ready to be written out as a .fbx or .obj file, which can be imported into Maya, bound to the appropriate skeleton, and exported as a .tfxmesh file.