Unity TressFX Documentation

Index

IMPORTANT: Unity TressFX is still in Beta!

1 Overview

TressFX is a hair simulation and rending technology developed by AMD.
This documentation contains information about the TressFX technology and the Unity implementation of it.

1.1 Minimum Requirements

TressFX has various requirements, which will be explained in this section.
First of all, it needs a DirectX >= 11 graphics card (so it's also bound to >= Windows 7).

Due to the high processing power needed by TressFX mid-range gaming cards are a minimum requirement (>= gtx 650). Also, a large amount of VRam is needed which is highly depending on your configuration. For example, the demo scene with AMDs tomb raider demo hair needs about 1gb of VRam to simulate and render the hair.

2 Unity TressFX

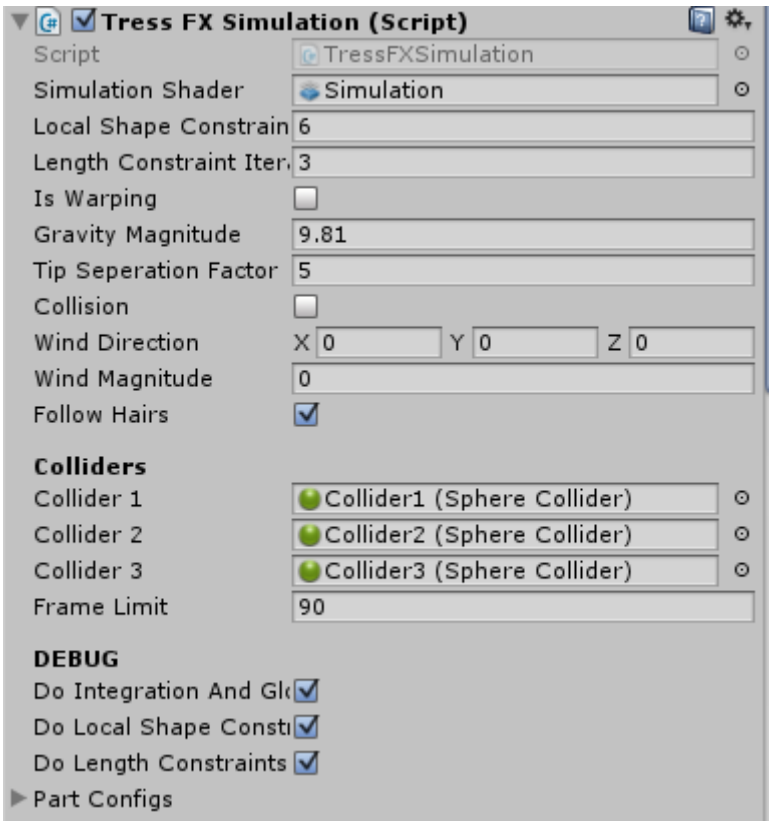
2.1 TressFX Components

Unity TressFX is split into several components. This was done in order to separate simulation code, rendering and datamanagement (Similar to the MVC-Pattern).

The datamanagement component of Unity TressFX is named “TressFX”:

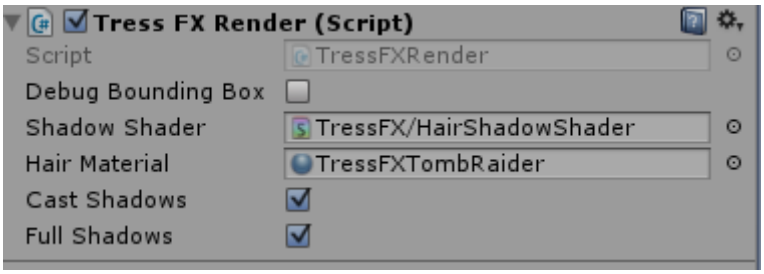
You can assign a hair data asset to it which it will use to load data.

The second component is the “TressFXSimulation” component:



This component runs the hair simulation on the GPU. It uses DirectCompute (ComputeShaders in unity) to simulate the hair data based on the data it reads from the “TressFX” component.

The last component is the “TressFXRenderer”:



This is the standard “non-fancy” Unity TressFX renderer. It is seamlessly integrated into the unity shading system and supports Shadow casting aswell as Shadow Receiving.
It is mainly implemented to get new renderer implementations started on an already working base.

2.2 Setting up TressFX Components

2.2.1 Simulation

The TressFXSimulation component is pretty easy to configure. Most settings will never need to be changed. Here is an explanation about the parameters:

Simulation Shader: This shader is located in Assets/TressFX/Shaders/Simulation.compute, just Drag’n’Drop it.

Local Shape Constraint Iterations: Iteration count for the simulation shader pass that enforces local shape constraints. Higher numbers improve stability, but also need more time to execute. 4-6 should fit for every hair just fine.

Length Constraint Iterations: Iteration count for the simulation shader pass that enforces length constraints. Higher numbers improve stability, but also need more time to execute. 2-4 should fit for every hair just fine.

Is Warping: If this flag is set, no simulation is run, when you move your character from scripts, always enable this to prevent simulation instability due to very fast movement.

Collision: Toggle collision with the hair colliders.

Tip Seperation Factor: The maximum offset for follow hair tips in relation to their guidance hair. This is highly depending on your hair scale.

Follow Hairs: Toggles follow hair updating. If your hair asset uses follow hairs, you want this enabled always.

Colliders: TressFX uses a special collision system. The collision volume which TressFX hair can collide against is described by 3 spheres. Those spheres are treated as “Connected Capsules”. The algorithm will create capsule colliders from those Spheres:

- Sphere 0 -> Sphere 1
- Sphere 1 -> Sphere 3
- Sphere 3 -> Sphere 0

2.2.2 Rendering

The TressFXRenderer can render the hair using the given material. The material must use a special shader that can read the hair spline data.

Shadow Shader: This shader is used to render shadow casters for your hair.

Hair Material: This material is used to render the hair geometry. The shader this material uses **MUST** be compatible with the unity tressfx rendering system. You cannot use standard shaders here and you can also not use the tressfx shader for normal geometry!

The shadow toggles control shadow casting / receiving.

2.2.3 Skinning

Skinning is not yet supported by Unity TressFX! Sorry ☹

2.3 Troubleshooting / FAQ

2.3.1 Hair simulation just fails and hair style is ignored / Importing hair on realistic scale

This problem mostly happens if the asset import didn't work well.
This mostly happens on hair that is modeled in "realistic" scale.
In this case the asset importer runs into rounding issues because of 32bit float limitations.

There is a very simple workaround for this:
- Reimport your hair with scale set to 100 on import (so it is imported 100x bigger)
- Set the hairs scaling to 0.01|0.01|0.01. This scaling will only affect the rendering!

This will make the hair simulation work in a 100x more precise / scaled up coordinate system while the renderer transforms those coordinate back into the view coordinates.

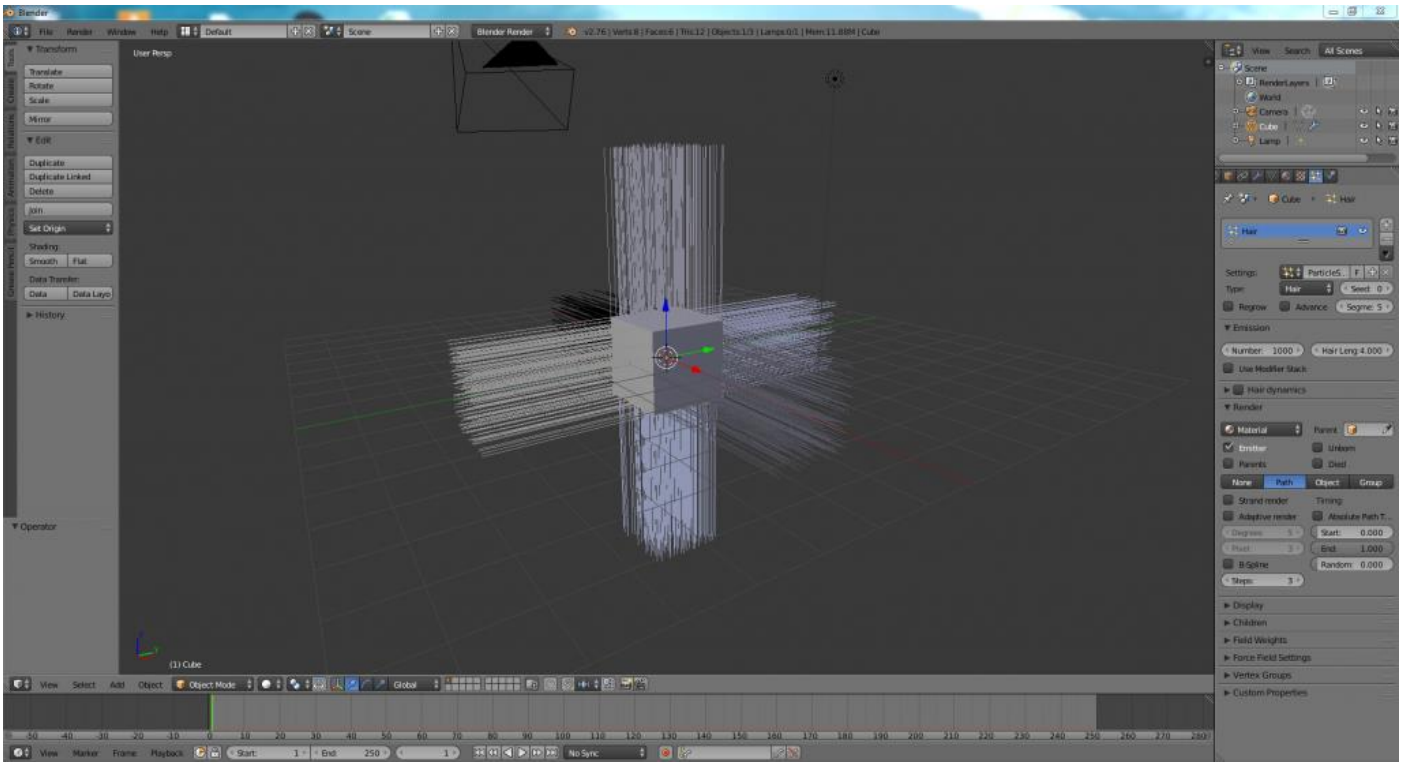
2.3.2 Further support

Post an issue on github: https://github.com/kennux/TressFXUnity
Support is available via E-Mail: kennux@virtual-illusions.com

3 Tutorials

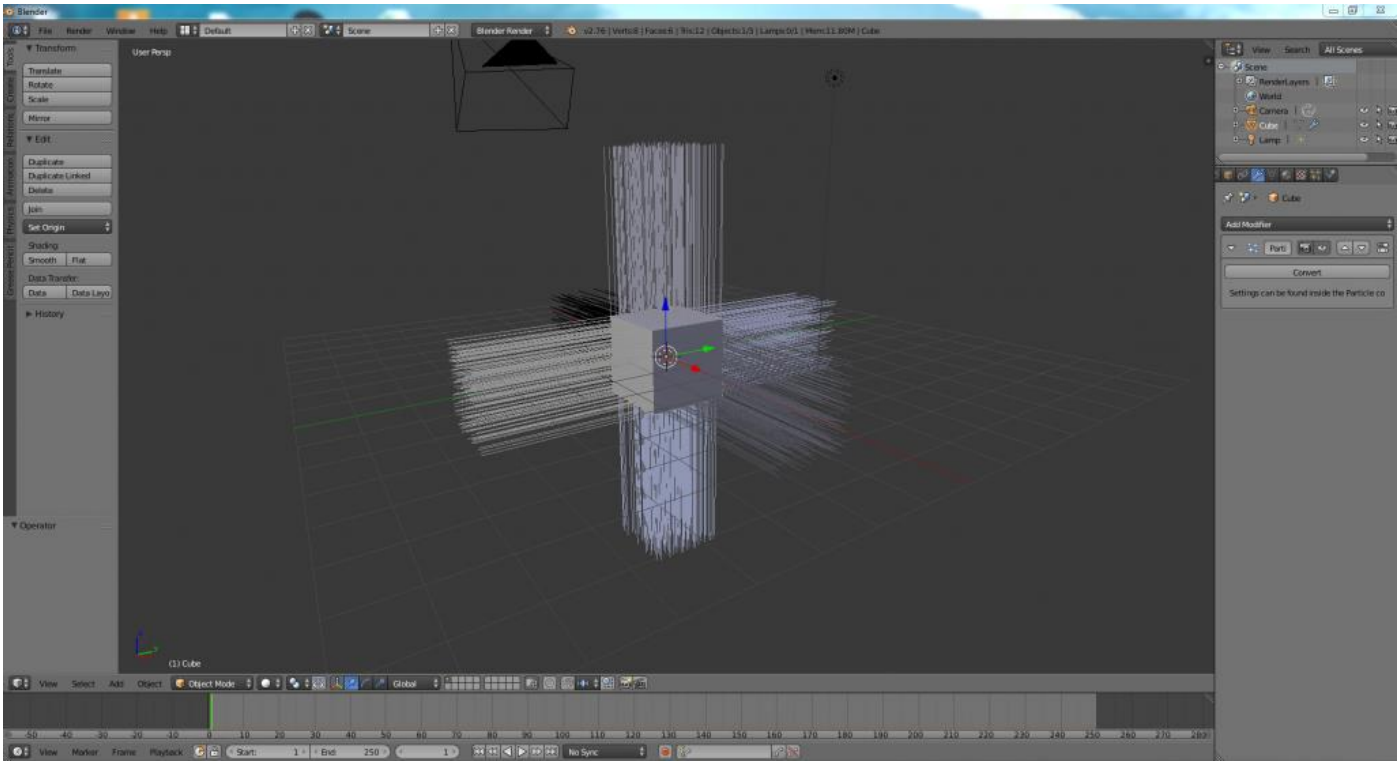
3.1 Blender to Unity hair import

First, open up your blender (I'm using version 2.76).
Add a new particle system to any object in the scene (I'm using the cube that's always added on a new scene).
Your scene should now look like this:

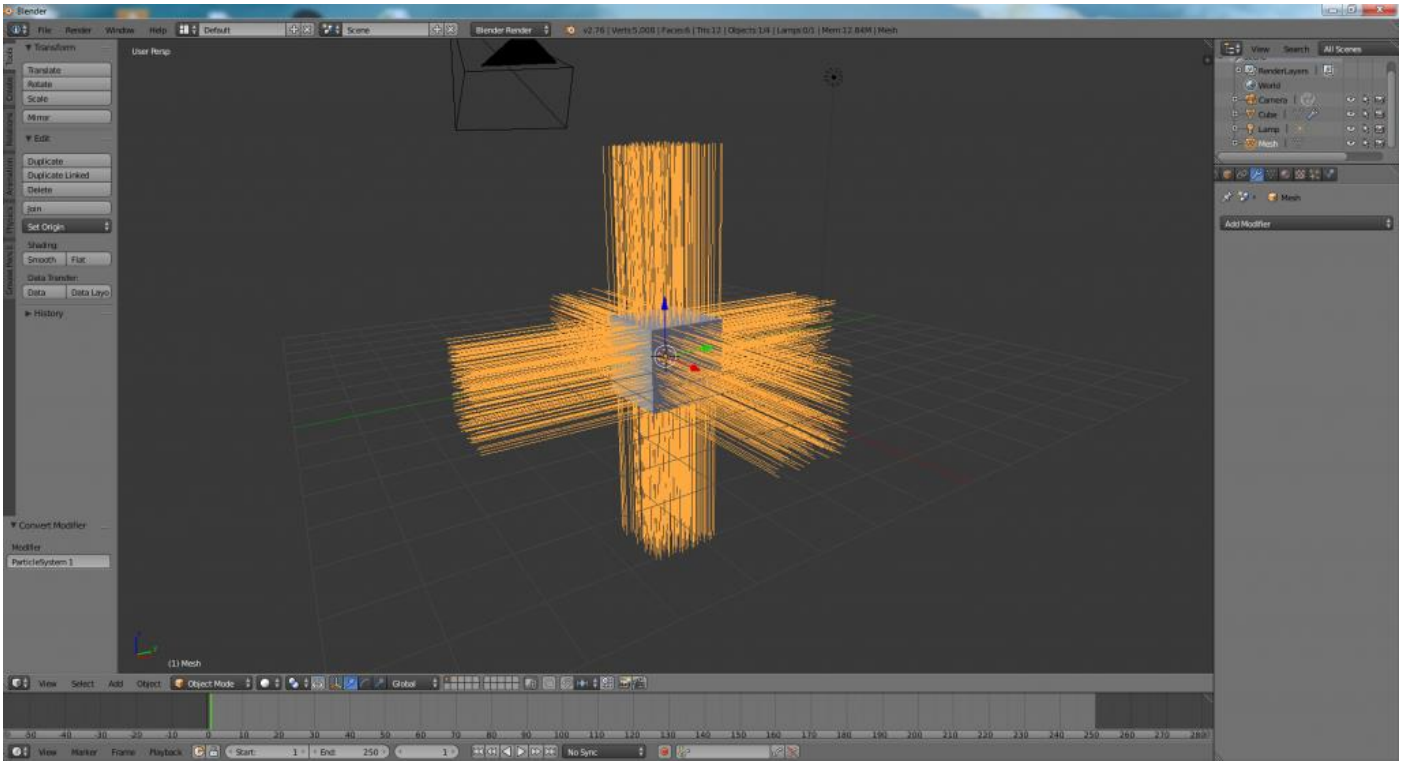


I will leave the hair now just as it is, you can freely shape it to what you like with the particle hair editors.

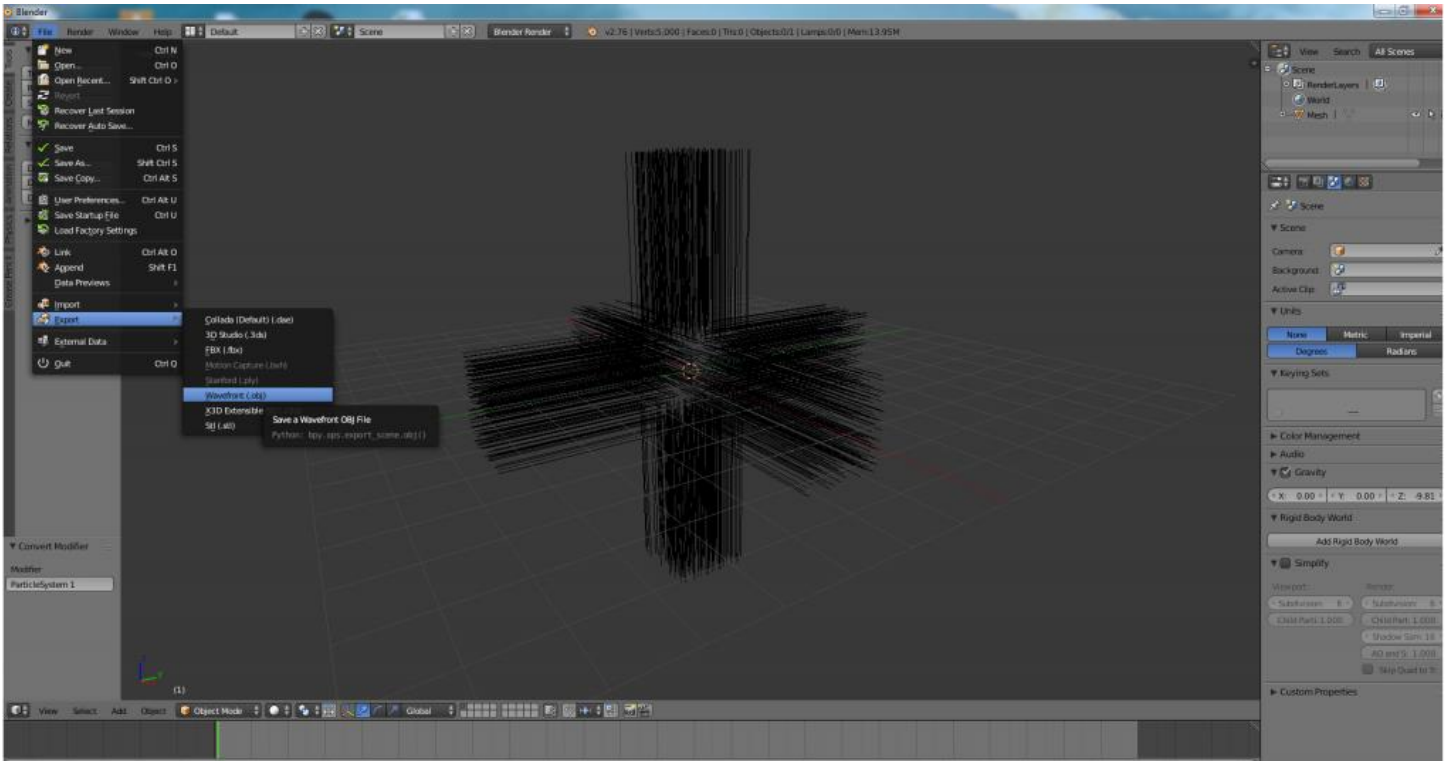
The next step is converting the hairs into an actual mesh we can export, in order to do so we need to add a modifier, which can be done at the modifier tab:



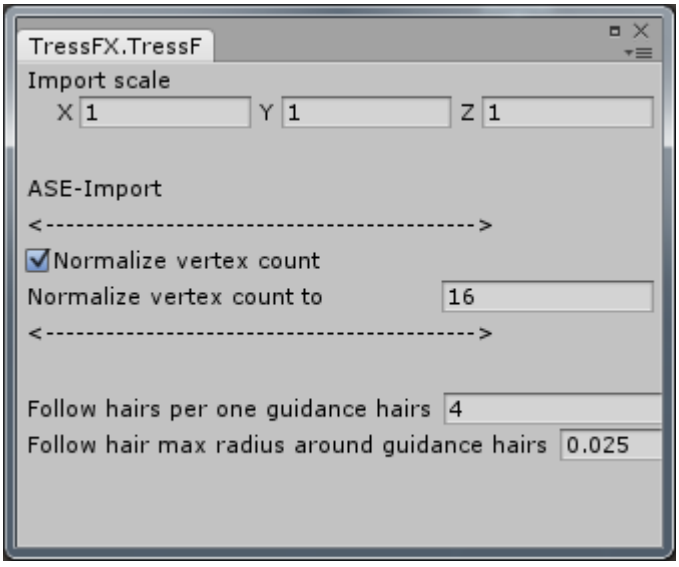
Press the “Convert” button to convert your particle system into a mesh. Now your scene should contain a new object:



Now, remove all unnecessary things from your scene (the light, camera and the cube) and export the hair mesh as Wavefront OBJ file:

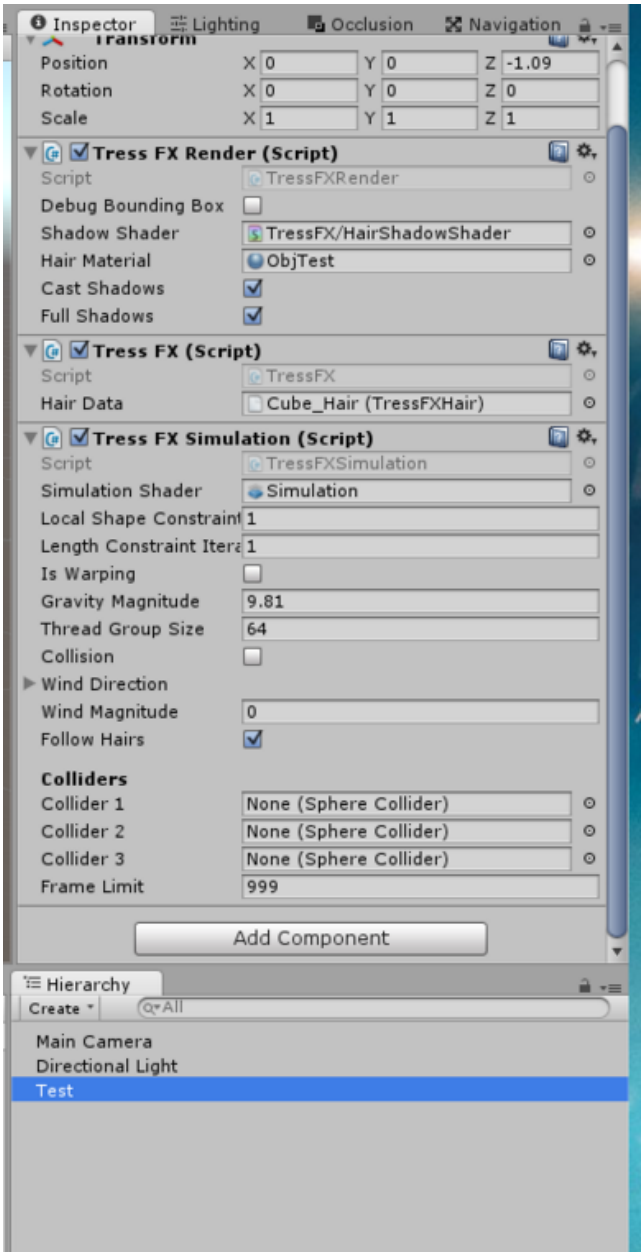


Now, we are almost done.
In order to import this hair into unity now we need to customize the TressFX import settings a bit. To do so, open up the TressFX Window (Window > TressFX).
First, you need to tick normalize vertex count in order to normalize the hair strands to 16 vertices per strand (this is only needed in order to use simulation).
The follow hair max radius around guidance hairs is ways too high for this kind of hair aswell. It controls the radius around the hairs you import where additional hairs are added to make the hair more dense. I used 0.025 here:
Important: Remember to scale up by 100 if your units are 1 = 1 meter and you want to use Simulation. This is further explained in Section 2.3.1.



Now right click in the project explorer, select “Create > TressFX > Hair from OBJ” and let unity import the obj file.
After that you are done importing the hair mesh into unity.

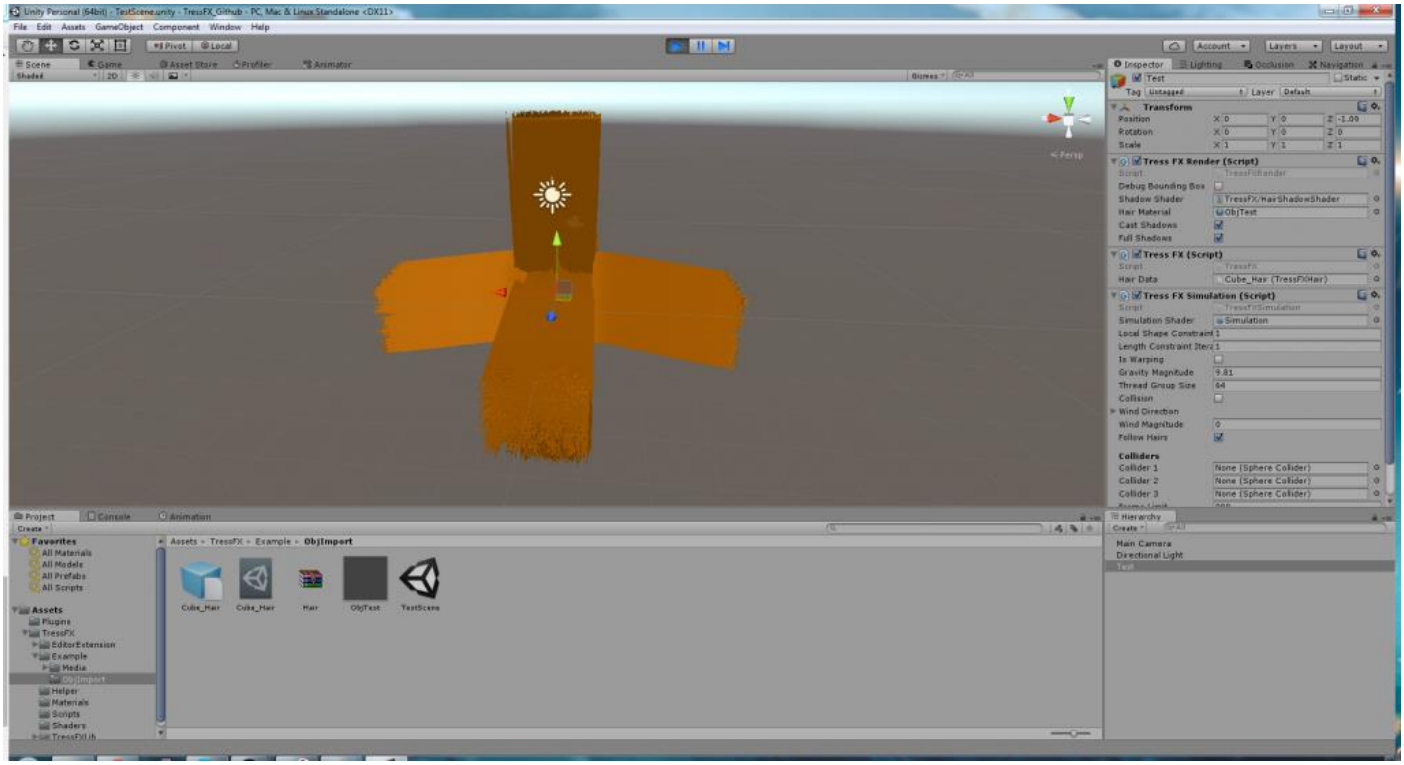
Now, creating an example scene is pretty straightforward.
Create a new gameobject and add the TressFXRender and TressFXSimulation script to the gameobject.
Assign the “Shadow Shader” and “Hair Material” to the TressFXRender (The shader is located in Assets/TressFX/Shaders/TressFXShadow.shader, the material must use the shader “TressFX/Surface”).
Now, assign the “Simulation Shader” of the TressFXSimulation component and the “Hair Data” of the TressFX Behaviour.
Finally it should look like this:



The last step now is setting up the simulation parameters for the hair, click the hair asset and configure them as you like.
Here is my example config:

Press “Save”, and you are done

If you hit the start button, it should get rendered like this now:

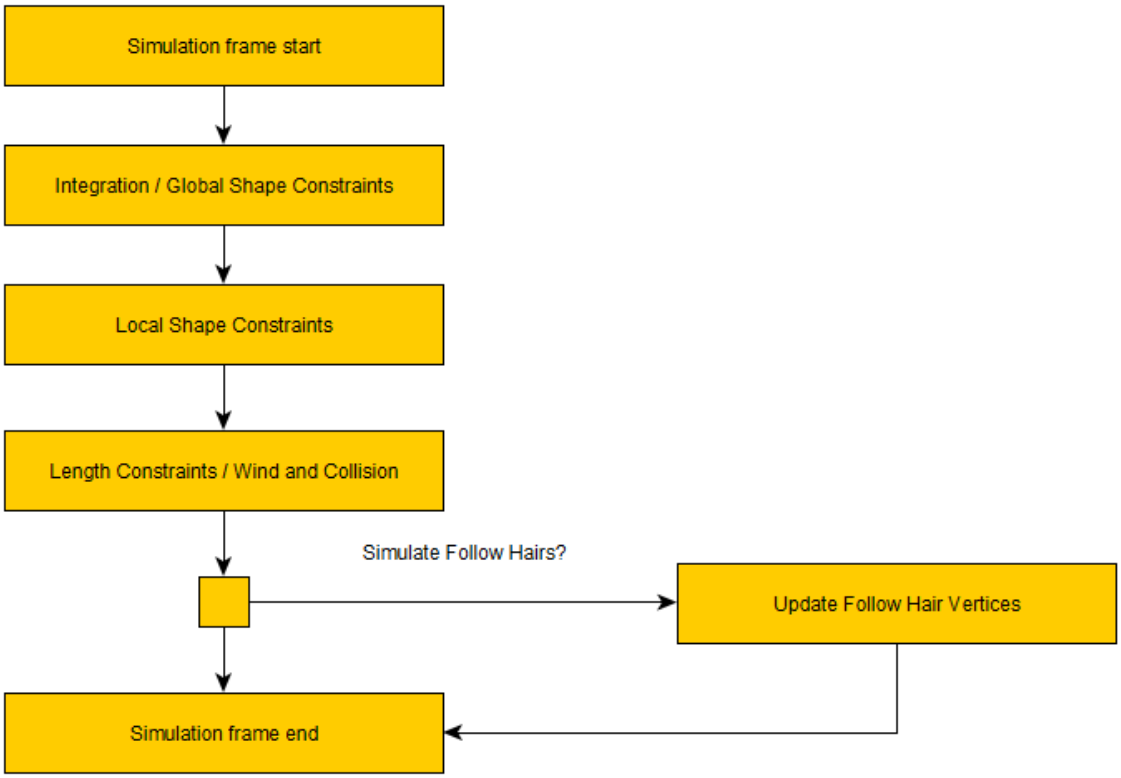


4 Technical Documentation

The following section will describe several technical parts of TressFX and Unity TressFX.

4.1 Simulation

The simulation system runs in several stages:



Each step in this figure runs in an own compute kernel. The following descriptions were copied from AMD's documentation for the simulation part of TressFX 2.

4.1.1 Integration / Global Shape Constraints

This shader does the Verlet integration to solve the equation for vertex motion and applies the global shape constraints. Each thread in the thread group calculates the position of one vertex, so Dispatch uses a thread group count = (NumHairStrands * MaxNumOfVerticesInStrand) / THREAD_GROUP_SIZE. Currently THREAD_GROUP_SIZE is defined at 64, which is the size of a wavefront on AMD GPUs. The shader copies the current position, rest length and local rotation for each vertex in thread group shared memory before calling GroupMemoryBarrierWithGroupSync(). That way the shaders has faster access to these variables than if they were stored in video memory. The new position is calculated with the Integrate function which uses Verlet integration with the velocity modulated by the damping coefficient:

$$x_{i+1} = x_i + (1 - \text{damping}) * (x_i - x_{i-1}) + \text{gravity} * dt^2$$

After the new position, xi+1, is calculated, the vertex needs to start moving back to its initial position in order for the hair to maintain its shape. This is done by applying the global shape constraint:

$$x = x_{i+1} + \text{stiffnessForGlobalShapeMatching} * (x_{initial} - x_{i+1})$$

Therefore, the higher the stiffness factor, the faster the vertex returns to its initial position. The value of the stiffness factor depends on the value of strandType which is stored with the vertex data. The strandType value is set when the hair asset file is read, so hair that was read from the same file have the same strandType value. This way, different sections of hair can use different values for stiffness and damping parameters. Finally, the old position and the updated new position are stored in the g_HairVertexPositionsPrev and g_HairVertexPositions UAVs with the function UpdateFinalVertexPositions().

4.1.2 Local Shape Constraints

LocalShapeConstraints is the next compute shader to be dispatched from the Simulation. As the name suggests, it applies a local shape constraint to handle bending and twisting of individual strands. Instead of computing a vertex per thread, this shader computes a full strand of vertices per thread. This shader is dispatched multiple times to improve the accuracy of the constraint. Curly hair needs more iterations than straight hair. The shader iterates over all of the vertices in the strand and applies the StiffnessForLocalShapeMatching associated with the strandValue for the vertex. This function calculates a delta value to determine how much to move the current and next vertex in the hair strand to return it to its original shape. For example a hair strand that has a curl in it needs to return to the original curl shape after the vertices have moved. It does this by using the reference vectors in the local frame stored in g_HairRefVecsInLocalFrame. LocalShapeConstraints rotates the

reference vector according to the target rotation, then adds this vector to the current vertex (stored in the g_HairVertexPositions from the IntegrationAndGlobalShapeContraints shader) to get the target goal position of the next vertex in the hair strand. The delta vector is therefore the difference between this edge and the actual edge between the current and next vertex in the strand. Additionally the delta is scaled by the StiffnessForLocalShapeMatching to control the rate at which the vertices returns to their target goal positions.

```
float4 rotGlobalWorld = MultQuaternionAndQuaternion(g_ModelRotateForHead, rotGlobal);

float3 orgPos_i_plus_1_InLocalFrame_i = g_HairRefVecsInLocalFrame[globalVertexIndex+1].xyz;

float3 orgPos_i_plus_1_InGlobalFrame = MultQuaternionAndVector(rotGlobalWorld,
orgPos_i_plus_1_InLocalFrame_i) + pos.xyz;

float3 del = stiffnessForLocalShapeMatching * (orgPos_i_plus_1_InGlobalFrame -
pos_plus_one.xyz).xyz;
```

This code listing shows how the delta is calculated from the reference vectors and the current position. Note that the 0.5 is a constant factor in the equation because half of the distance is subtracted from the current vertex, and the other half added to the next vertex. This causes both vertices to move an equal distance to the target goal together.

4.1.3 Length Constraints, Wind and Collision

The next compute shader called from the Simluation is LengthConstraintsAndWind which is used for keeping the lengths of the hair strands constant and for adding motion caused by a wind vector. Each thread in the compute shader calculates a single vertex position. Vertices are loaded into thread group shared memory for faster access. First the effect of wind is calculated using integration. The force vector is calculated with a cross product between the line segment formed by the current vertex and the next one in the list and the wind vector:

$$v = x_{current} - x_{next}$$
$$force = -(v \times wind) \times v$$
$$x_{i+1} = x_i + force * dt^2$$

A group memory barrier is needed to make sure all of the wind calculations are done before enforcing the length constraint. The length constraint is applied by using the rest lengths g_HairRestLengthSRV buffer. This buffer has the target rest lengths for each segment of all of the hair strands. Similar to local shape constraints, the length of the segment formed by the current vertex and the next one in the list is calculated. Then the difference between the current length and the rest length for the segment is used to determine how much to move the two endpoints of the segment to return to the target rest length.

After that, the CapsuleCollision () function is called to check if the hair vertex collides with the top sphere, bottom sphere or middle cylinder of the collision capsule. Any collision will cause the position of the vertex to be moved to the edge of the collision geometry. After a group memory barrier, to make sure all of the collisions have been calculated, the new tangent values for each vertex are calculated. Tangents are simply the normalized vector from the current vertex position to the next one in the hair strand. They are needed for the Kajiya hair lighting model which uses a tangent instead of a normal for the light calculations.

4.1.4 Update Follow Hair Vertices

This is the last shader called updates the follow hair vertices based on their guidance hair references. First all vertices of a threadgroup are copied into groupshared memory, then a group memory barrier is enforced to synchronize all thread group threads.

Then, the follow root offset is multiplied with the follow hair follow offset factor (calculated using tip separation factor). The final position is written back to the position buffer.

This shader operates on one vertex per thread level only for guidance hairs.

4.1.5 Simulation data

The following table contains information about the data buffers the simulation system use and their format:

The following table contains information about the simulation shader parameters / it's constant buffer:

4.2 Rendering system

5 License

5.1 Unity TressFX