TressFX Porting Guide

AMD Developer Relations
September 2013

Overview

TressFX is the name of the hair simulation and rendering technology from AMD. This guide will explain the TressFX code in the TressFX11 sample and how to port the TressFX code to a game. TressFX technology is already being used in a game. Crystal Dynamic's Tomb Raider, released in 2013 features realistic looking hair on the main character, Lara Croft. The TressFX11 sample code is based on the same code that was used in the game, so it has the same optimizations that were necessary to make this technology practical for use in a real game. Additionally, AMD has made further improvements to the code to make it faster, and more modular for easier porting to your game

Figure 1: TressFX hair rendering technology in Tomb Raider

AMD TressFX makes use of the processing power of high performance GPUs to do realistic rendering and utilizes DirectCompute to physically simulate each individual strand of hair. Rendering is based on an I3D 2012 Paper from AMD "A Framework for Rendering Complex Scattering Effects on Hair", Yu, et.al., with some modifications and optimizations in order to achieve a level of performance that enables TressFX to be used in games. The physics simulation is described in another paper by AMD which was presented at VRIPHYS 2012: "Real-time Hair Simulation with Efficient Hair Style Preservation", Han, et al.

Rendering

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

- Antialiasing
- Self-Shadowing
- Transparency

When used with a realistic shading model these features provide realism needed for natural looking hair. The sample uses the Kajiya-Kay illumination model which use anisotropic lighting to get the correct kind of highlights that hair has. Two specular highlights are rendered, similar to the Marschner model, using an algorithm first developed by Sheuermann at ATI in a presentation at GDC 2004 “Hair Rendering and Shading”. The slides for this presentation can be found at: http://developer.amd.com/wordpress/media/2012/10/Scheuermann_HairRendering.pdf. The hair is rendered as several thousand individual strands stored as thin chains of polygons. Since the width of a hair is considerably smaller than one pixel, antialiasing is required to achieve good looking results. In TressFX this is done by computing the percentage of pixel coverage for each pixel that is partially covered by a hair.

The second main feature, Self-Shadowing, is necessary for giving the hair a realistic looking texture. Without it, the hair tends to look artificial and more like plastic on a puppet than as opposed to real hair. Shadowing is done using a simplified deep shadow map. Typically a deep shadow map is several layers of depth values. To improve performance and memory usage, the implementation interpolates over a range of depth values to avoid having to keep a list of depth values for each pixel.

Transparency provides a softer look for the hair, similar to real hair. If transparency was not used the strands of hair would look too coarse, especially at the edges. In addition to that real hair is actually translucent, so rendering with transparency is consistent with simulating the lighting properties of real hair. Unfortunately transparent hair is difficult to render because there are thousands of hair strands that need to be sorted. To help with this, TressFX technology uses order independent transparency (OIT). The A-Buffer required for OIT is implemented as a per-pixel linked list (PPLL), so fragments can be sorted and rendered in the correct order. The PPLL is generated by writing into a DX11UAV from the hair pixel shader. Once the PPLL is filled the hair is rendered to the back buffer by applying a full screen quad.

Simulation

Physically accurate simulation of the hair is completely done on the GPU using DirectCompute. The key focus of simulation is to simulate styled hair such as curly or wavy efficiently. Of course, it is also possible to simulate straight hair and actually it is cheaper than styled hair in terms of computation time.

Since TressFX is for real-time video games, simulation uses constraints rather than springs to represents bending and twisting effects. Also it uses distance constraints to simulate inextensibility of hair. For simplicity and performance, bending and twisting effects are combined into one constraint called Local Shape Constraints (LSC). To help its convergence and improve the performance and stability, TressFX uses Global Shape Constraints (GSC). Lastly, Edge Length Constraints (ELC) is used to maintain the edge distances in hair strand.

To integrate gravity, a Verlet integration method is used. For wind, drag and lift forces are combined together and calculated per vertex basis. The important part of wind implementation is to randomize the direction and magnitude so that hair won't get clumped and can maintain its volume globally. Also it is necessary to modulate the wind magnitude using sine function to generate a nice wavy motion.

The TressFX hair simulation has many parameters that allow the programmer and artist to modify the look and behavior of the hair. For example the stiffness of the hair can be modified on the fly to make it look wet. The hair can also be made with varying levels of stiffness and damping, also adjustable during execution.

One thing to keep in mind is if the local/global shape constraints are modeled to look good with gravitation pointing downwards, strange behavior can occur if the gravitation direction is reversed (for example if a character using TressFX is hanging upside down). Should this become necessary it is easiest to create a special hair mesh with simulation parameters authored to look good while upside down.

TressFX11 Sample Code Organization

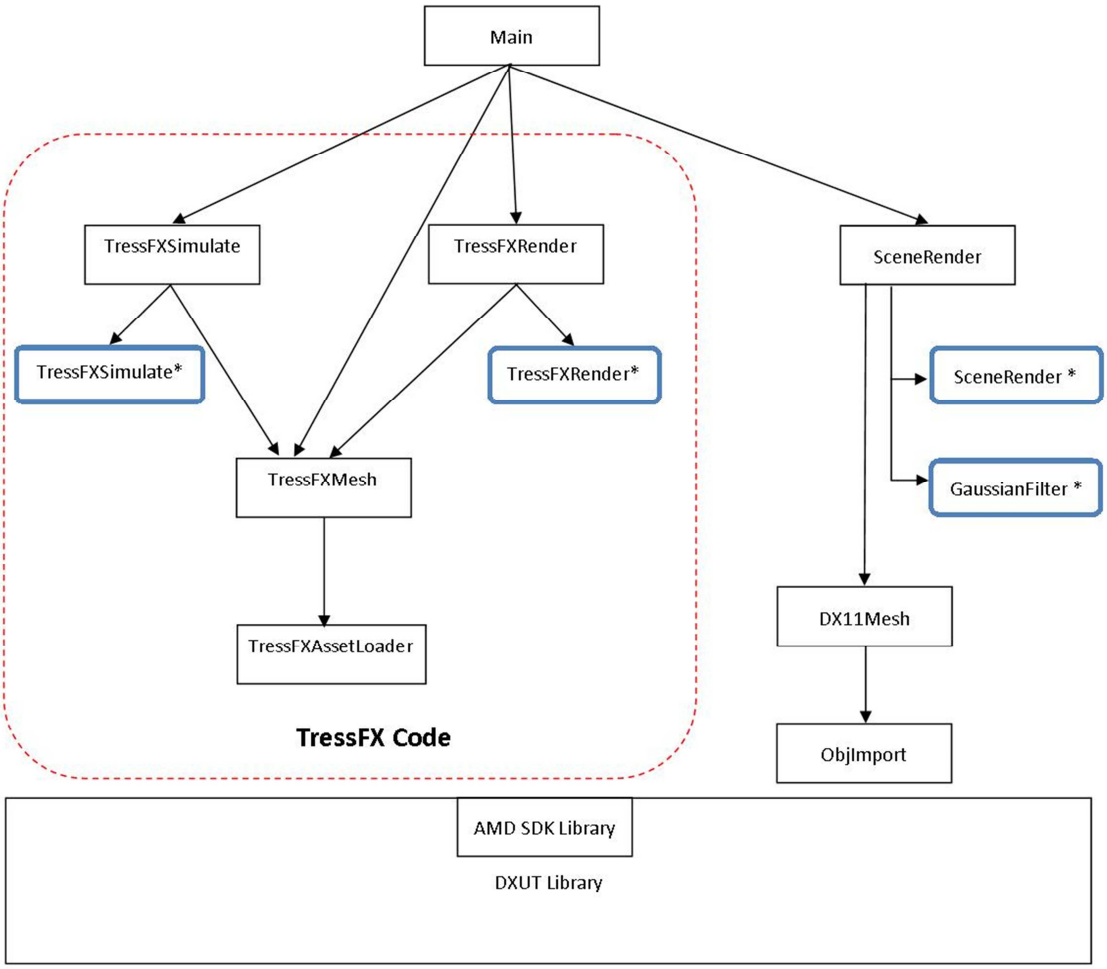

Figure 2: TressFX11 sample code dependency graph. The boxes marked with "*" indicate shader code

The latest version of the TressFX11 sample code is available for download at: http://developer.amd.com/tools-and-sdks/graphics-development/amd-radeon-sdk/ . Figure 2 shows the hierarchy of source code modules in the sample. Note that the entire TressFX implementation is in the modules surrounded by the red dashed line. This is the code that you will want to port to your game. In general all of the TressFX code is in files and classes with "TressFX" in their name. The other modules are either support libraries (AMD SDK, DXUT), or used for other rendering (such as scene geometry).

main.cpp

At the highest level, main.cpp contains the initialization code for the sample and the callback functions. Since the sample uses the DXUT framework (as is used in DirectX SDK Samples from Microsoft) it uses callbacks that are called from DXUT on various events. The two most important callbacks are OnD3D11CreateDevice(), and OnD3D11FrameRender(). The CreateDevice callback is called when DXUT creates the Direct3D device which happens shortly after execution begins. In the OnD3D11CreateDevice callback, the sample calls the corresponding OnCreateDevice functions for these classes: CTressFXMesh, CTressFXSimulation, CTressFXRender, and CSceneRender which are instantiated as global objects in main.cpp. Each of these classes uses the OnCreateDevice functions for initializing data and allocating Direct3D resources. The OnD3D11FrameRender callback is called from DXUT when the sample is ready to render the frame. In this callback, the hair simulation is executed by calling the CTressFXSimulation::Simulation() member function. This animates the hair vertices for rendering. The OnD3D11FrameRender callback then calls the CTressFXRender::GenerateShadowMap() function to create the shadow map of the hair which is used for self-shadowing and for shadows of the hair on the scene geometry. The CSceneRender::GenerateShadowMap() function is also called at this time to create a shadow map for casting shadows of the scene onto the hair. Finally, the OnD3D11FrameRenderer callback calls the actual rendering functions CSceneRender::RenderScene() for rendering the scene geometry (the head model) and CTressFXRender::RenderHair() for rendering the hair. It's important to note at this point that the RenderHair() member function takes a HAIR_PARAMS parameter which includes parameters for controlling the hair density (number of strands), thickness, maximum number of overlapping fragments, and transparency. If LODs are enabled, the OnD3D11FrameRender callback will adjust these parameters based on distance from the camera. In particular, the density adjustment is key to reducing the number of strands as the camera moves away from the hair which improves performance without much loss of quality. Also, because this is a continuous adjustment, popping is avoided.

Hair Rendering

CTressFXRender::OnCreateDevice() Function

TressFXRender.cpp contains the implementation of the CTressFXRender class which as the name suggests is responsible for rendering the hair. When CTressFXRender ::OnCreateDevice() is called it makes calls allocate and initialize the Direct3D resources. Among these is a call to CTressFXRender::OnCreateDevice () which uses the AMD_SDK ShaderCache class to create the necessary vertex and pixel shaders for rendering. This function will need to be changed for games, since they won't be using the AMD ShaderCache. This function can be modified to either make explicit Direct3D calls to load the shaders, or use the game engine's existing code for managing

shaders. CTressFXRender::CreateConstantBuffer() is also called from the OnCreateDevice() function. This is a simple function that uses D3D calls to allocate a constant buffer for hair rendering called m_pcbPerFrame (see table 1). Games may want to expose some of the elements of this buffer to the art pipeline since it contains parameters that change the appearance of the hair. In the sample, these parameters are exposed to the sample UI so you can see how these parameters can be used.

Constant Buffer Element	Definition
m_mWorld	world matrix
m_mViewProj	view projection matrix
m_mInvViewProj	inverse view projection matrix
m_mViewProjLight	view projection matrix from light source
m_vEye	eye position
m_fvFOV	field of view
m_AmbientLightColor	ambient light color
m_PointLightColor	point light color
m_MatBaseColor	base color of the hair
m_MatKValue	x = ambient material color y = diffuse material color z = primary specular highlight w = primary specular exponent
m_fHairKs2	secondary specular highlight
m_fHairEx2	secondary specular exponent
m_FiberAlpha	hair transparency value
m_HairSMAlpha	transparency of hair shadow map
m_bExpandPixels	If true, expand the hair width in the vertex shader by 0.71 pixels on either side.
FiberRadius	radius of each hair strand (1/2 width)
m_WinSize	x = screen width y = screen height z = 1/screen width w = 1/screen height
m_FiberSpacing	average spacing between fibers
m_bThinTip	If true, use the g_HairThicknessCoeffs buffer to adjust the thickness of the hair. This makes the hair thinner towards the tip.
m_fNearLight	distance from the light to the near side of the bounding sphere of the hair.
m_fFarLight	distance from the light to the far side of the bounding sphere of the hair
m_iTechSM	Shadow Map Technique (simplified deep shadow map technique, or no shadow)
m_bUseCoverage	Enable/Disable hair antialiasing
m_iStrandCopies	number of duplicate hair strands to render
m_iMaxFragments	When LODs are used, maximum number of fragments to search through the PPLL for the nearest k fragments.
m_alphaThreshold	When LODs are used, the minimum alpha value needed to include a fragment in the PPLL
m_mInvViewProjViewport	inverse view projection matrix

Table 1: Per-frame constant buffer (m_pcbPerFrame) elements.

Shadow Map Generation

Before rendering the hair, the shadow map is created for the hair by calling CTressFXRender::GenerateShadowMap(). This is straightforward shadowmap rendering where the hair is rendered from the point of view of the light. GenerateShadowMap() uses CTressFXRender::RenderHairGeometry() to render the actual hair geometry into the shadowmap. This function is also used later for rendering hair into the frame buffer. RenderHairGeometry() takes a parameter to indicate that line drawing should be used for the shadow map. The hair geometry is already stored as line segments, and rendering lines is sufficient for shadows. Rendering lines cuts down on the geometry and overdraw so it's faster than rendering triangles.

Render Pass 1: A-Buffer Fill Pass

The first pass of hair rendering is to fill the A-buffer with the overlapping transparent hair fragments in a per-pixel linked list. With the pixels stored as linked lists, they can be sorted in back to front order so correct transparent rendering can be done. In the main OnD3D11FrameRender() callback, CTressFXRender::RenderHair() is called, which in turn calls the same RenderHairGeometry() function used by the shadow map rendering. However in this case, RenderHairGeometry() is given a parameter that tells it to render triangles instead of lines. The line segments, tangents, and per-vertex hair thickness are stored in buffers that are passed as shader resource views to the vertex shader. For triangle rendering, an index buffer with the triangle indices is bound. The vertex shader uses this information to update the position of the current vertex. There are actually 4 ways that RenderHairGeometry() is called when rendering the hair, because it's called with a parameters to enable/disable antialiasing, and one for the number of copies for each strand (1 or multiple copies). If multiple copies are needed then RenderHairGeometry() will use DrawIndexedInstanced() for instanced rendering. There are 4 corresponding vertex shaders that get used in TressFXRender.hlsl: VS_RenderHair(),VS_RenderHair_StrandCopies(), VS_RenderHair_AA(), and VS_RenderHair_AA_StrandCopies().

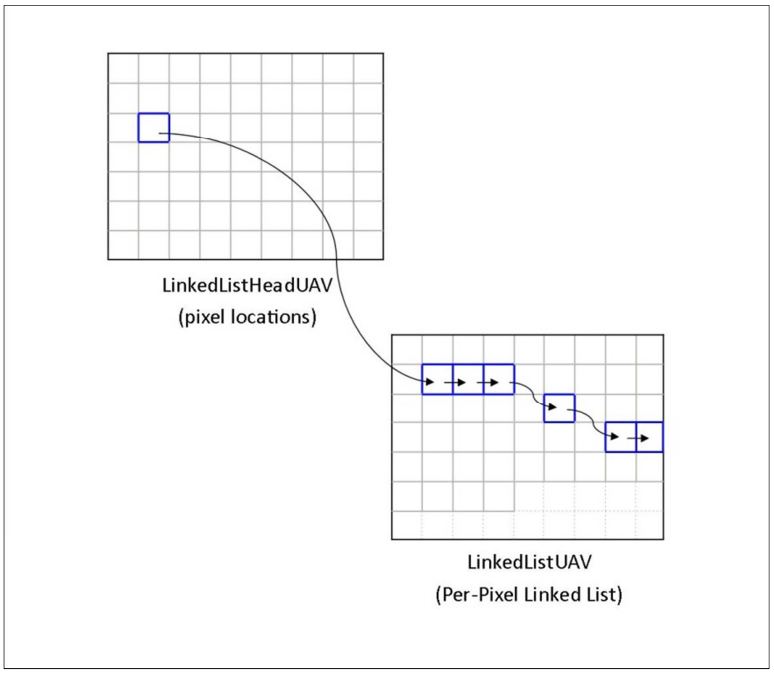

Figure 3: Per-Pixel Linked List A-Buffer.

The pixel shader used for the A-buffer pass is PS_ABuffer_Hair(). This shader stores the fragments in the per-pixel linked list (PPLL) which is implemented with two Unordered Access View (UAV) buffers (see figure 3). One UAV called LinkedListHeadUAV stores the start of the linked list for the pixel. Fragment data are stored in sequential order in the other UAV named LinkedListUAV which increments the UAV counter every time a fragment is entered. When a new fragment is entered, the previous fragment location for the pixel is read from LinkedListHeadUAV and exchanged with the new location in LinkedListUAV using the atomic InterlockedExchange function. This makes sure that the current LinkedListHeadUAV location isn't accidentally overwritten by another thread. The previous fragment location is then used as the "next" pointer for the linked list. The fragment data stored in the PPLL includes the tangent, pixel coverage, and depth. The tangent is stored with the fragment data because tangents are used instead of normals in the Kajiya-Kay lighting model used for the hair. Coverage is the amount of the pixel covered by the fragment which is used for antialiasing. Coverage is calculated in the ComputeCoverage() function which uses the screen space width of the hair strand to determine how much of the pixel is covered. The resulting coverage value is then used to modulate the alpha value of the fragment for correct antialiasing. Figure 4 shows the flow of the first pass from hair geometry to UAV output.

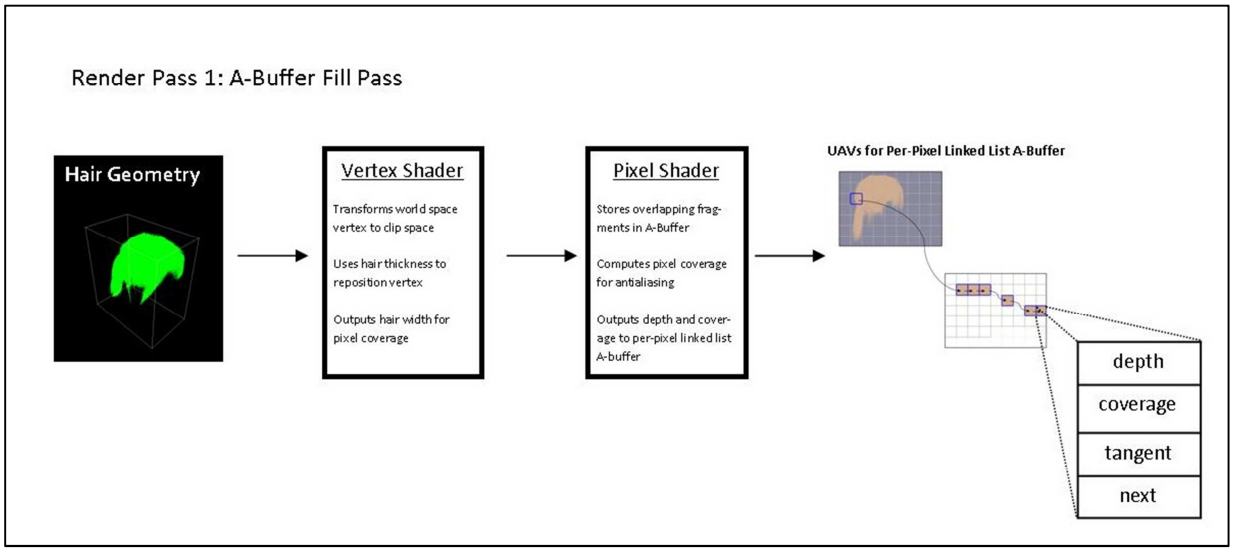

Figure 4. Hair Rendering Pass 1 - A-Buffer Fill Pass

Render Pass 2: K-Buffer Sort and Draw Pass

The second pass of hair rendering goes through the A-Buffer and for each pixel, finds the nearest k fragments. In the TressFX11 sample k is set to 8 by default. High quality results can be obtained by properly sorting and rendering just the nearest k fragments. For the remaining overlapping fragments it's acceptable to use lower quality shading with an out of order blend to improve performance, since these fragments are below k or more layers of hair fragments and don't contribute much to the final pixel color. To sort and draw the final hair pixels, a full screen quad is rendered. The pixel shader, PS_KBuffer_Hair() will use the pixel's location to lookup the head of the linked list in the LinkedListHeadUAV(). Note that the geometry rendering in the first pass initializes the stencil buffer so that only pixels that contain hair fragments are rendered on the second pass. PS_KBuffer_Hair() then goes through the linked list finding the nearest k fragments and storing them into the kBuffer[] array. Fragments that don't make it into the k-buffer array use the

ComputeSimpleShadow() and SimpleHairShading() functions to light the pixel, then blended with the other fragments that aren't in the nearest k array. ComputeSimpleShadow is a faster version of the shadow calculation that doesn't use shader filtering. Similarly, SimpleHairShading() uses a faster but lower quality shading. After this, PS_KBuffer_Hair() iterates over the kBuffer[] array, blending the fragments in back to front order and uses the higher quality ComputeShadow() and ComputeHairShading() to light the pixel. Since the lighting is done in the second pass, the A-buffer needs to store depth and tangent, similar to a G-Buffer used in deferred shading. The 3D position is calculated using the screen space pixel position and an inverse perspective transformation. Figure 5 shows the flow of the second render pass and how it uses the UAV per-pixel linked list from the first pass.

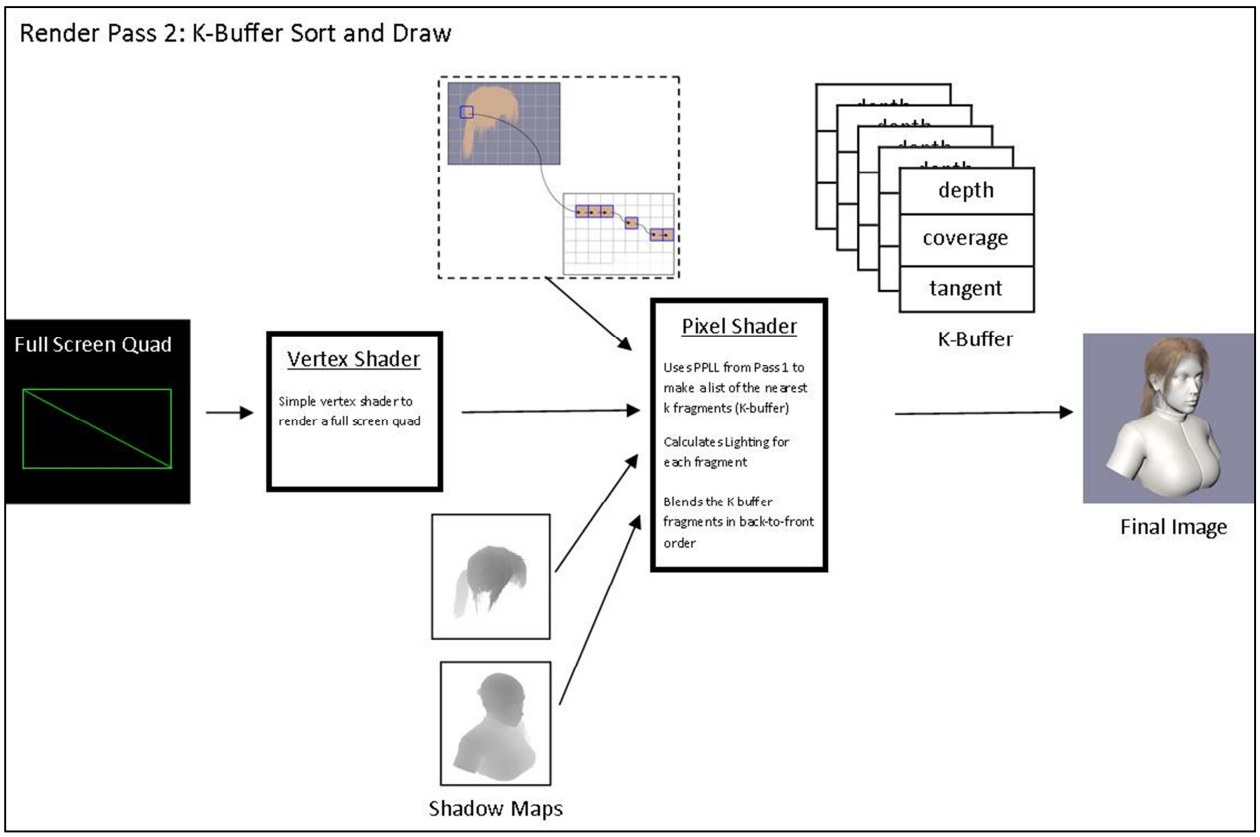

Figure 5. Hair Rendering Pass 2 – K-Buffer Sort and Draw Pass

Hair Simulation

CTressFXSimulation::OnCreateDevice() function

As with the render OnCreateDevice() function, the simulation OnCreateDevice() creates the shaders that are needed by the simulation code. It uses the same AMD_SDK ShaderCache class as the render code to create the compute shaders, so you may want to replace this code with an existing implementation in your game for managing shaders. Additionally this function calls CreateComputeShaderConstantBuffers() to create the m_pCBCSPerFrame constant buffer which gets updated every frame (see table 2). This constant buffer contains the simulation parameters,

some of which can be changed in the TressFX11 sample from the simulation menu. These are also parameters you may want to expose to artists for setting the default values, and possibly for saving different values to represent the hair under different conditions, such as wet hair.

Constant Buffer Element	Definition
ModelTransformForHead	The transformation on the head which is also used for transforming the hair vertices before simulation.
ModelPrevInvTransformForHead	Previous inverse transformation on the head for calculating the previous location of the hair vertex.
ModelRotateForHead	Rotation on head used in calculating rotation of hair vertices.
Wind, Wind1,Wind2, Wind3	Wind force vectors which are each slightly offset from a main wind direction. The simulation interpolates between the wind vectors based on strand index.
NumLengthConstraintIterations	Number of iterations to do length constraints. A higher number is more accurate, but slower.
bCollision	If true, hair collision with head is simulated.
GravityMagnitude	Magnitude of gravity used in the integration.
Damping*N*	Damping coefficient for hair section *N*, where *N* = 0..3 which is used in the Verlet integration
StiffnessForLocalShapeMatching*N*	Stiffness value used in local shape matching for hair section *N*, where *N* = 0…3. Values range from 0 to 1 and the closer to 1, the greater the local shape constraint is applied.
StiffnessForGlobalShapeMatching*N*	Stiffness value used in global shape matching for hair section *N*, where *N* = 0…3. Values range from 0 to 1 and the closer to 1, the greater the global shape constraint is applied.
GlobalShapeMatchingEffectiveRange*N*	The range of vertices in a strand that are affected by the global shape matching for hair section *N*, where *N* = 0…3. Values range from 0 to 1, with 1 meaning all vertices in the strand are affected.

Table 2: Per-frame simulation constant buffer (m_ pCBCSPerFrame) elements.

Note that there are currently 4 sets of shape matching and damping parameters which correspond to the maximum of 4 sections of hair. During loading, the hair can be read from multiple files, each file representing a different section of hair (front, back, sides, etc.) when appending the hair data to the list of strands, each vertex is given a “strand type” value which is based on the file it was read from. That way, different parts of the hair can have different shape constraints and damping. Currently the maximum is 4 files and 4 sets of parameters, but it wouldn’t be difficult to increase this to more if for some reason the hair needed more parts.

CTressFXSimulation::Simulate() Function

In the TressFX sample, the Simulate() function is called from the OnD3D11FrameRender() function in main.cpp prior to calling the TressFX render function. The Simulate() function simulates the motion of the hair due to external forces caused by gravity, head movement, and wind. The updated hair vertex positions are then passed to the hair render code for rendering. Simulate() mainly does the setup for the 4 dispatch calls which run the compute shaders for doing the actual physics calculations. It initializes the m_pCBCSPerFrame constant buffer, sets 4 shader resource

views (see table 3) and 6 UAV buffers (see table 4) for the compute shaders. The SRVs and UAVs are part of the TressFXMesh class because they are initialized during the hair asset loading, and because it also makes it easier to share the buffers with the hair rendering code.

Shader Resource View	Definition
m_HairVerticesOffsetsSRV	Offset in the vertex array to the start of each strand
m_HairRestLengthSRV	Rest lengths of hair strand line segments
m_HairStrandTypeSRV	Integer value to indicate which hair asset file the strand was loaded from.
m_HairRefVecsInLocalFrameSRV	Initial reference vector of edges in their local frame. Calculated once during hair asset loading by creating a translation vector between the previous vertex and the current one in the array.

Table 3: Shader resource views for simulation compute shaders.

UAV Buffer	Definition
m_HairVertexPositionsUAV	Current positions of the hair strand vertices
m_HairVertexPositionsPrevUAV	Previous positions of the hair strand vertices
m_HairVertexTangentsUAV	Current tangent vector for each vertex
m_InitialHairPositionsUAV	Initial positions of the hair vertices
m_GlobalRotationsUAV	Global rotations for each line segment in the hair strands.
m_LocalRotationsUAV	Local rotations for each line segment in the hair strands.

Table 4: Unordered access views for simulation compute shaders.

The m_HairVertexPositionsUAV and m_HairVertexPositionsPrevUAV buffers are used as both input and output to the compute shaders called by Simulate(). Each compute shader call uses the vertex positions from the previous call and passes the modified vertex data to the next compute shader. The following are the compute shaders that are dispatched from the Simulate() function:

IntegrationAndGlobalShapeConstraints

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, x_{i+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().

LocalShapeConstraints

LocalShapeConstraints is the next compute shader to be dispatched from the Simulate function. 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_GuideHairRefVecsInLocalFrame[globalVertexIndex+1];

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

float3 del = stiffnessForLocalShapeMatching * 0.5f * (orgPos_i_plus_1_InGlobalFrame - pos_plus_one.xyz).xyz;
```

Code Listing 1: Calculation of delta in LocalShapeConstraints compute shader

Code Listing 1 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.

LengthConstraintsAndWind

The next compute shader called from the Simluate function 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 X wind) X 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.

CollisionAndTangents

CollisionAndTangents is the last compute shader to be dispatched form the Simulate() function. The vertex positions are copied into the thread group shared memory, and each thread calculates one new vertex position. For collision detection, the head and body are represented by 3 capsules. 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.

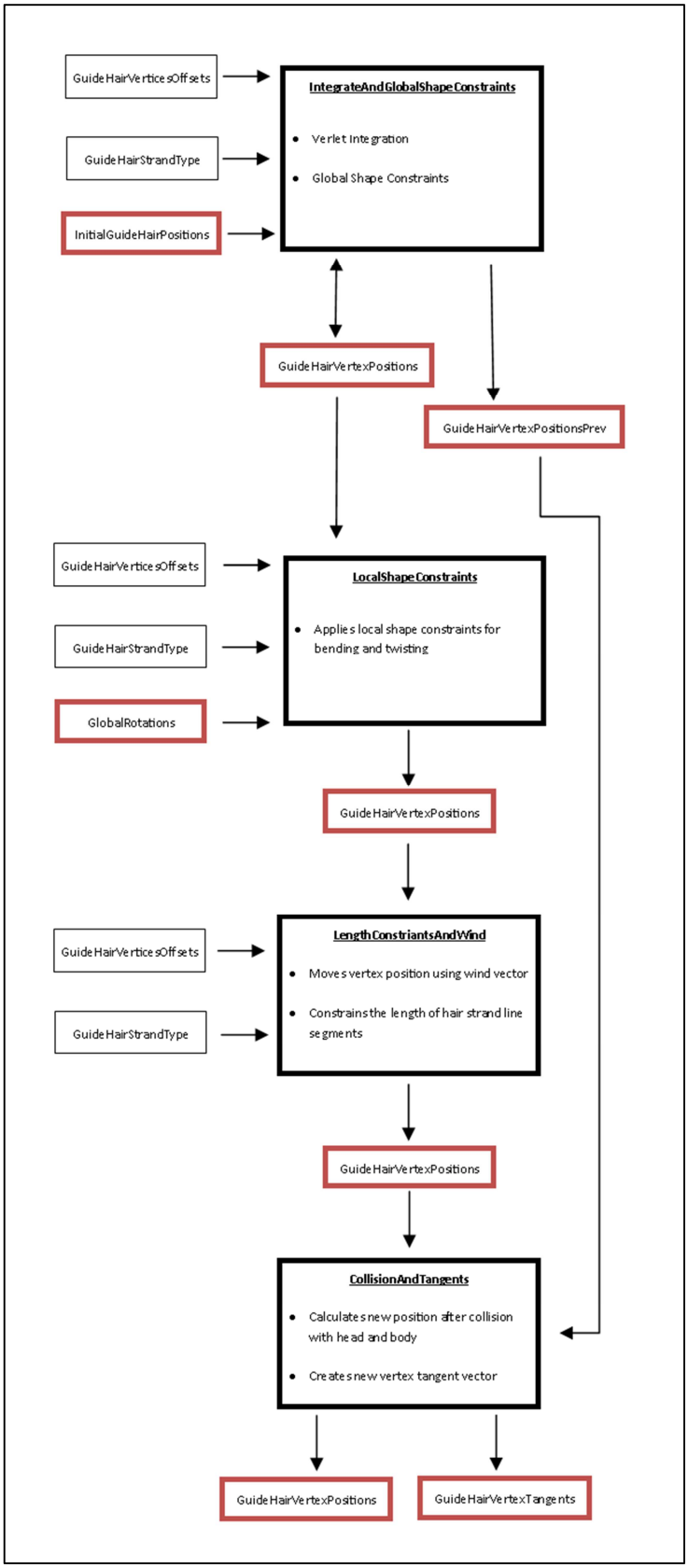

Figure 6: Data and execution flow of simulation. Red boxes are UAVs

Hair Asset Management

The CTressFXMesh class is responsible for storing the hair data in a form that can be used by the CTressFXRender and CTressFXSimulation classes for rendering and simulation. Both of these classes contain pointers to a CTressFXMesh object which gets initialized during the OnD3D11CreateDevice() callback. When CTressFXMesh::OnCreateDevice() is called it takes parameters which include an array of hair file names, the number of files, and affine transformations for the hair vertices. OnCreateDevice goes through the array of files names and loads each one and appends each set of hair vertices into one big array of hair strands. The CTressFXMesh contains a CTressFXAssetLoader class which does the work of reading the hair vertex data from a file and storing it in a vector template class. Hair vertices are stored in groups of strands. The CTressFXMesh class sets the maximum number of vertices per strand (m_MaxNumOfVerticesInStrand) to 32 by default. If you change the maximum number of vertices per strand, it should be set to 4, 8, 16, or 64, if the thread group size is 64. The more vertices in the strand, the longer the simulation takes, so you may want to adjust how many vertices per strand the assets contain for performance. The number of strands is trimmed so that it's always a multiple of the thread group size. The thread group size is set in the simulation compute shader code, and is based on the number of wavefronts (groups of threads) the graphics hardware is designed to use. OnCreateDevice() also calls CTressFXAssetLoader::ProcessVertices() to process the vertex data so that it's ready to be used by the rendering and simulation. This function computes the tangents for each vertex, the parametric distance to the root for each vertex, the length of each strand, transforms the vertex data and stores the final results into arrays. The TressFX11 sample does this all during the device creation callback, but since this only needs to be done once, it would be possible to do the hair loading as an offline process. You could create a stand-alone program that uses the CTressFXAssetLoader class to load and preprocess all of the hair files. That way, the processing of the hair vertices wouldn't impact the startup time of the game. The vertex data arrays created by CTressFXAssetLoader::ProcessVertices() could be saved to a file. At level load time, the game could read this file and store vertex data in the CTressFXMesh class.

The final thing that CTressFXMesh::OnCreateDevice() does is allocate all of the Direct3D resources that contain the hair data. These resources are public member variables that can then be accessed by the CTressFXRender and CTressFXSimulation classes.

Hair Asset File Format

TressFX hair used in the sample and in games has been authored with off the shelf content creation tools. For example, the "Shave and a Haircut" Maya plugin is good for designing hair. The hair can be exported by writing a simple exporter to .tfx format which is used by the TressFX technology. Tfx files are a text files which start with the number of strands and sorting information:

numStrands 1595
is sorted 1

This is followed by a line which gives the strand number and the number of vertices along with texture coordinates (which can be zero)

strand 0 numVerts 11 texcoord 0.000000 000000

Immediately following this is x,y,z vertex positions of the strands:

-4.80447 103.333 25.4651
-8.97057 105.761 27.8268
-14.103 106.443 29.189
-18.9103 104.199 28.8422
-22.2072 100.107 28.0818
-23.2806 94.8577 27.7387
-23.6553 89.5007 27.697
-24.0455 84.1447 27.6378
-24.3112 78.8325 26.9485
-24.4854 73.8321 25.0157
-24.8791 69.4098 21.9887

Since the number of vertices in the strand is 11, the number of coordinates is 11. The specification of strand followed by vertex list continues until all of the strands of hair are specified.

Summary of Porting Tips

- For hair that needs to be simulated upside down, it's best to create a separate mesh for upside down rendering with its own set of simulation parameters.
- Use the camera distance to set the hair density and other parameters when rendering the hair (as done in main.cpp) for continuous LODs.
- Expose the hair rendering and simulation parameters to the art pipeline so artists can control the look and animation of the hair.
- Replace the AMD shader cache with the game's shader management code to load shaders
- For improved simulation performance try reducing the maximum number of vertices per strand. The trade-off will be that long hairs may look too polygonal.
- Consider moving the CTressFXAssetLoader class into a separate tool for offline generation of the processed hair vertex data.