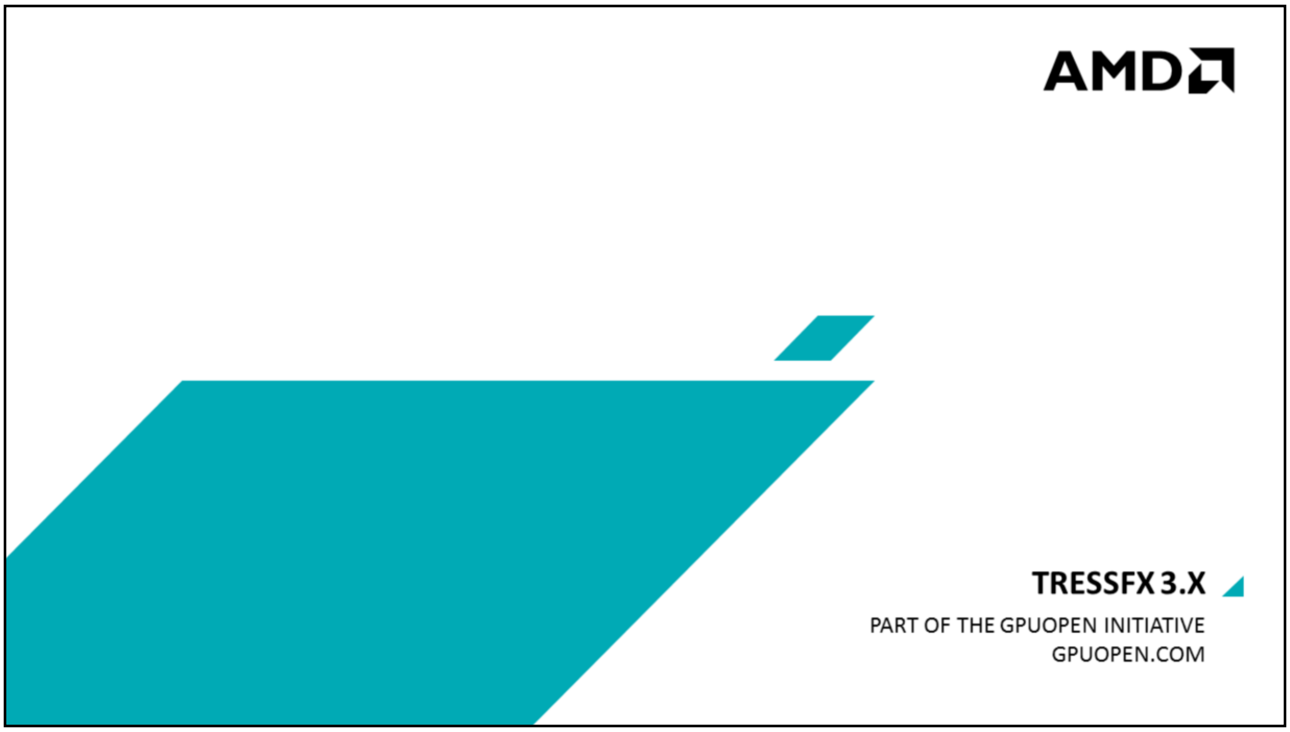


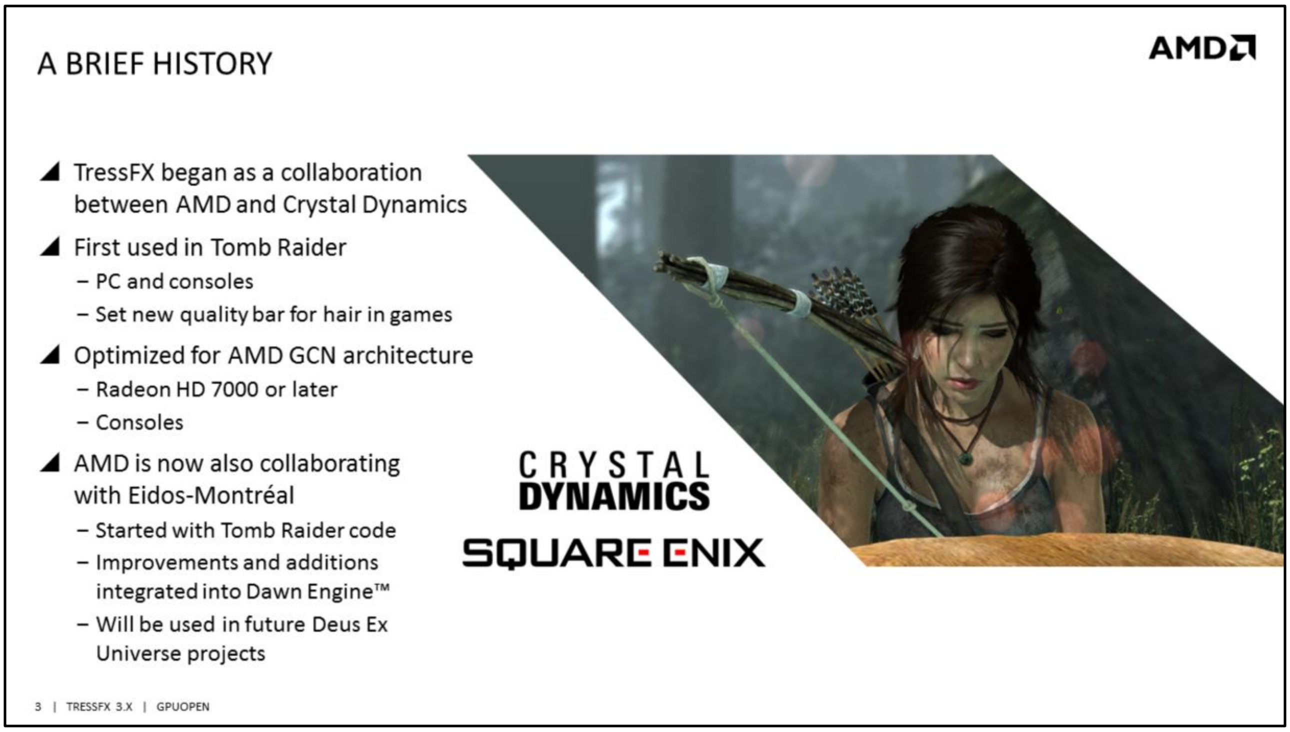
AMD

TRESSFX 3.X

PART OF THE GPUOPEN INITIATIVE
GPUOPEN.COM

HIGH-LEVEL CONTENTS

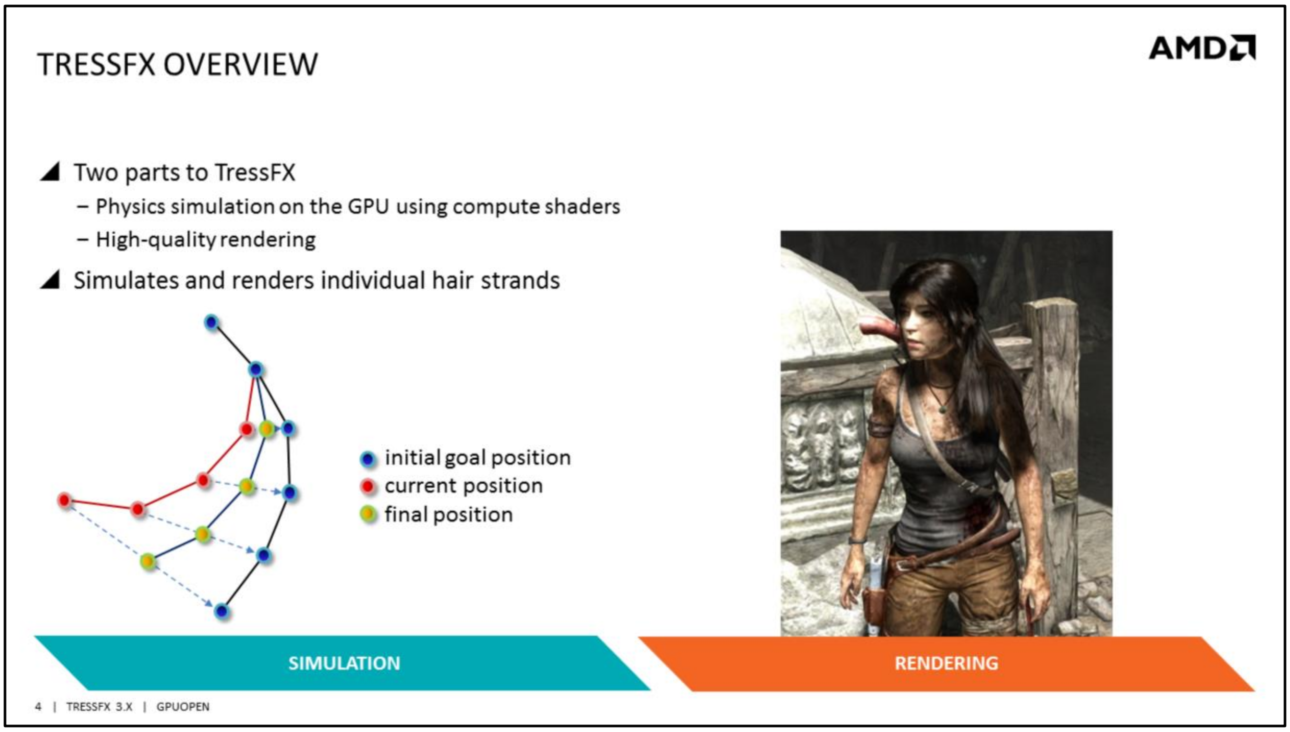
- Brief TressFX overview
- TressFX 3.x library
  - Update to AMD's TressFX example implementation
  - Maya plugin
  - Viewer and runtime library (with full source)
  - Fur support
  - Skinning
  - Future optimizations
  - New memory-friendly OIT method (ShortCut)



A BRIEF HISTORY

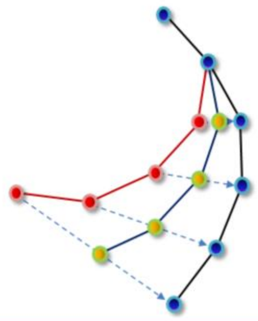
- TressFX began as a collaboration between AMD and Crystal Dynamics
- First used in Tomb Raider
  - PC and consoles
  - Set new quality bar for hair in games
- Optimized for AMD GCN architecture
  - Radeon HD 7000 or later
  - Consoles
- AMD is now also collaborating with Eidos-Montréal
  - Started with Tomb Raider code
  - Improvements and additions integrated into Dawn Engine™
  - Will be used in future Deus Ex Universe projects

TressFX is very high-quality real-time hair for games
Eidos-Montréal is a Square Enix company, as is Crystal Dynamics.
Being a Square Enix company, they had access to the Tomb Raider code
Started with that and have been enhancing and improving it



TRESSFX OVERVIEW

- Two parts to TressFX
  - Physics simulation on the GPU using compute shaders
  - High-quality rendering
- Simulates and renders individual hair strands



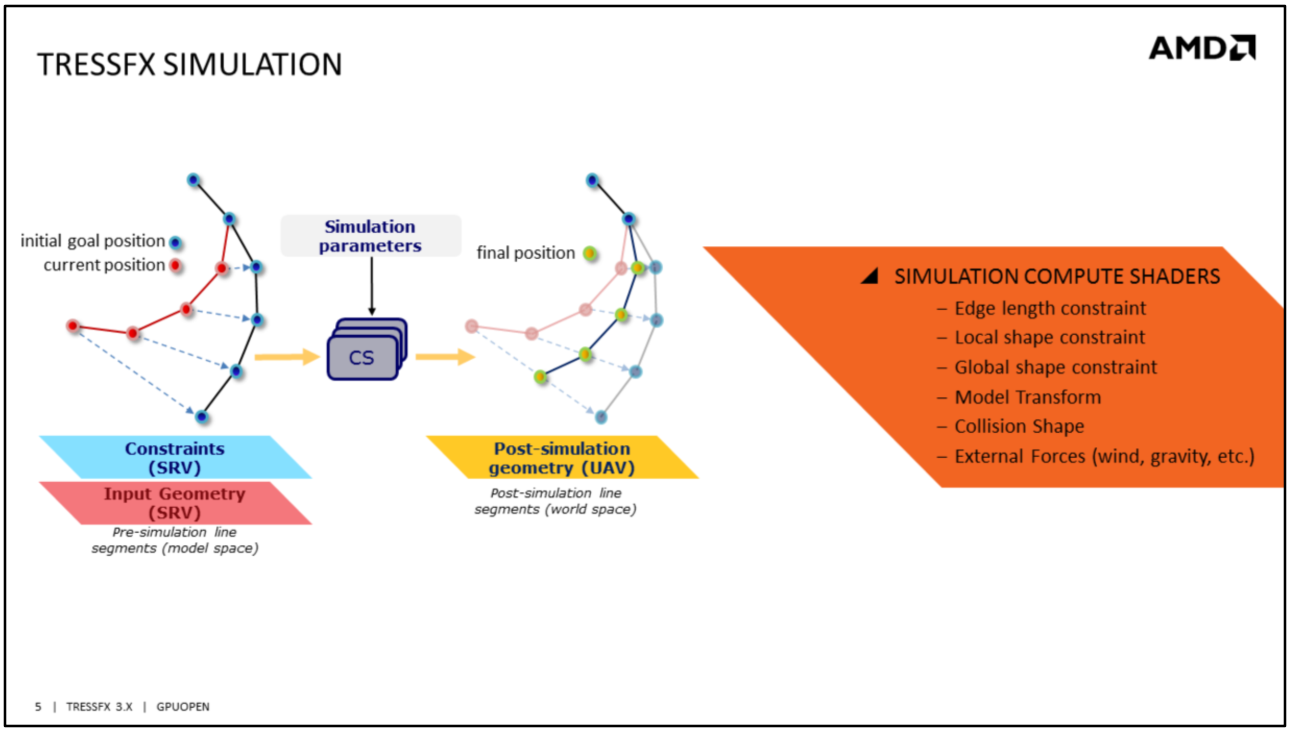
● initial goal position
● current position
● final position

SIMULATION

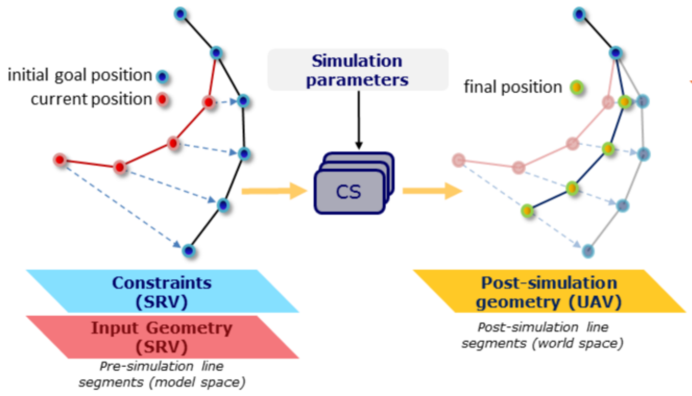
RENDERING

4 | TRESSFX 3.X | GPUOPEN

TressFX is very high-quality real-time hair for games
But how does it work?
At a high level, there are two parts

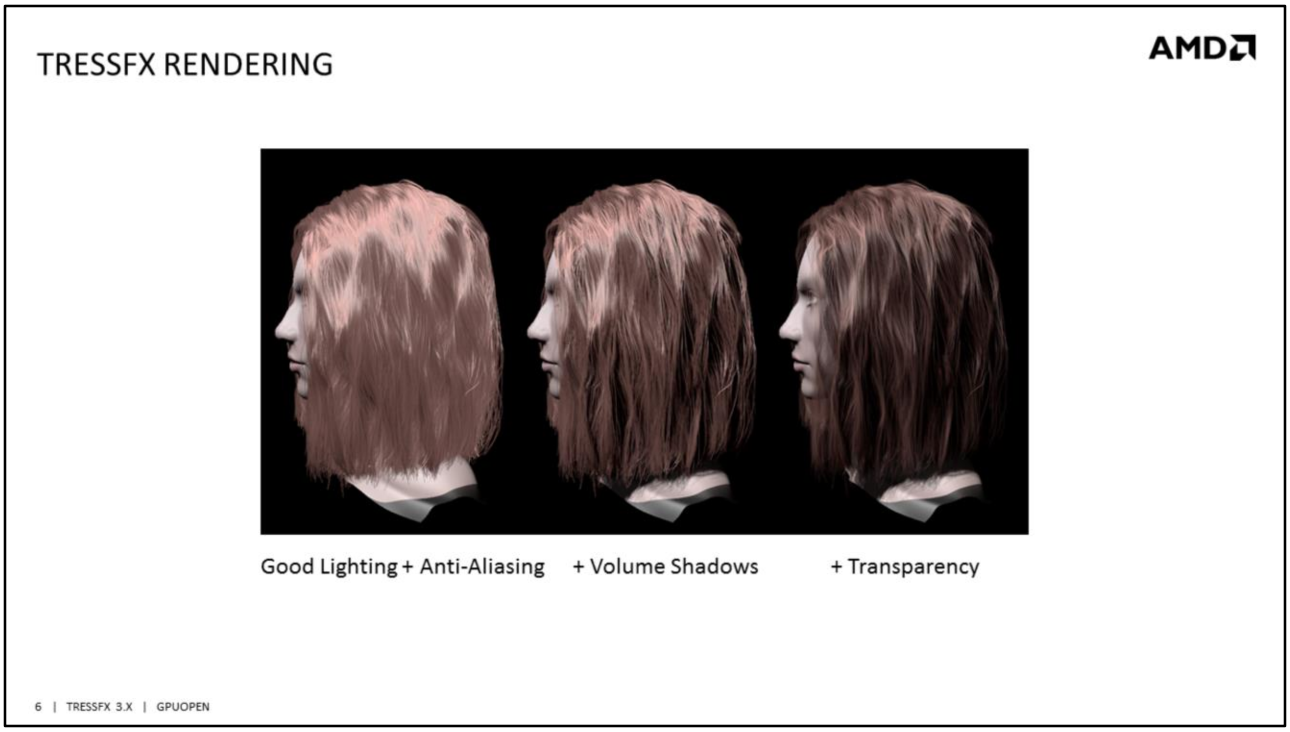


TRESSFX SIMULATION



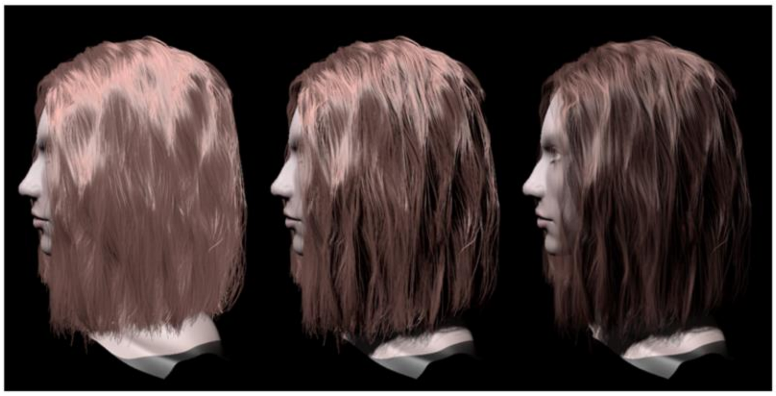
- SIMULATION COMPUTE SHADERS
  - Edge length constraint
  - Local shape constraint
  - Global shape constraint
  - Model Transform
  - Collision Shape
  - External Forces (wind, gravity, etc.)

Start with standard Verlet position update to account for forces such as gravity, with some damping
Then various constraints, to tune the behavior of the hair and to make the simulation behave
The example shows the global shape constraint, which tries to pull the hair verts back to their initial positions



TRESSFX RENDERING

AMD



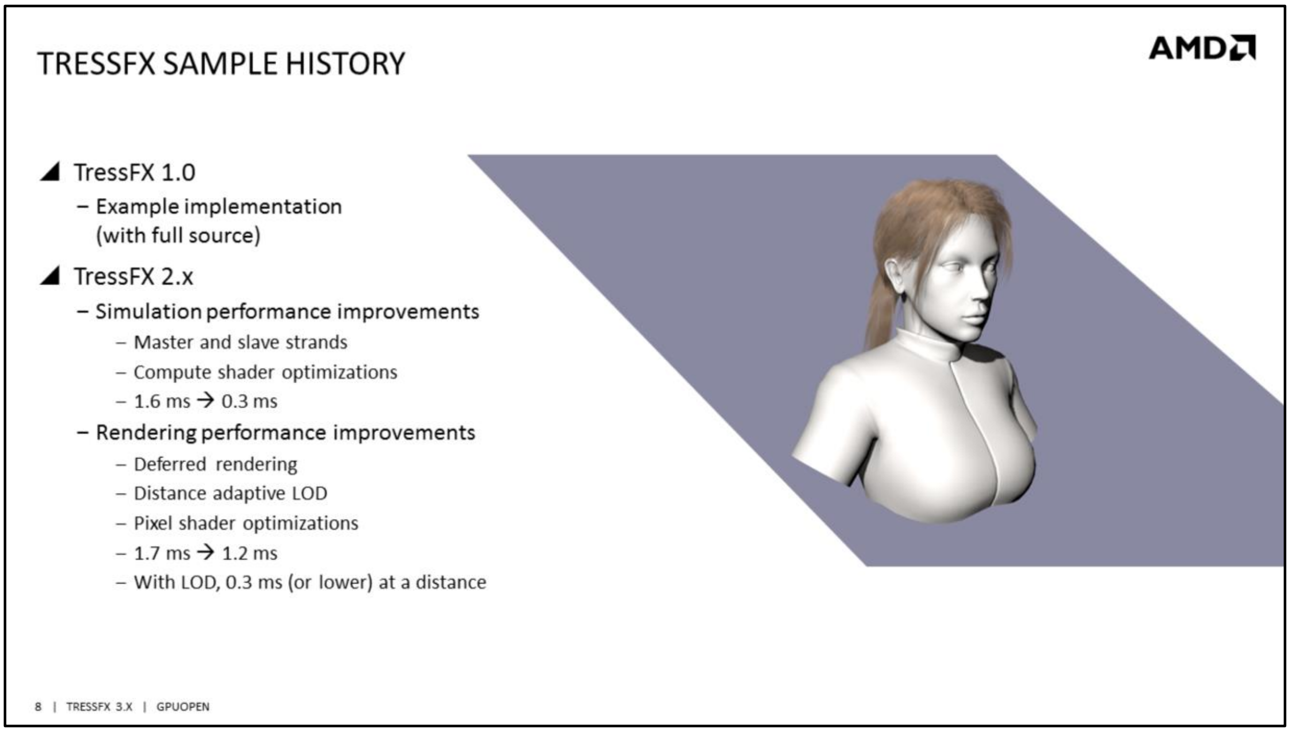
Good Lighting + Anti-Aliasing + Volume Shadows + Transparency

6 | TRESSFX 3.X | GPUOPEN

Hair shading (good lighting) using Marschner dual highlight approach and Kajiya-Kay lighting model
AA is not hardware MSAA. Calculate coverage in pixel shader and convert to alpha value
Simplified Deep Shadow Map technique
Order-independent transparency (OIT) using a per-pixel linked list (PPLL)

AMD

TRESSFX 3.X LIBRARY

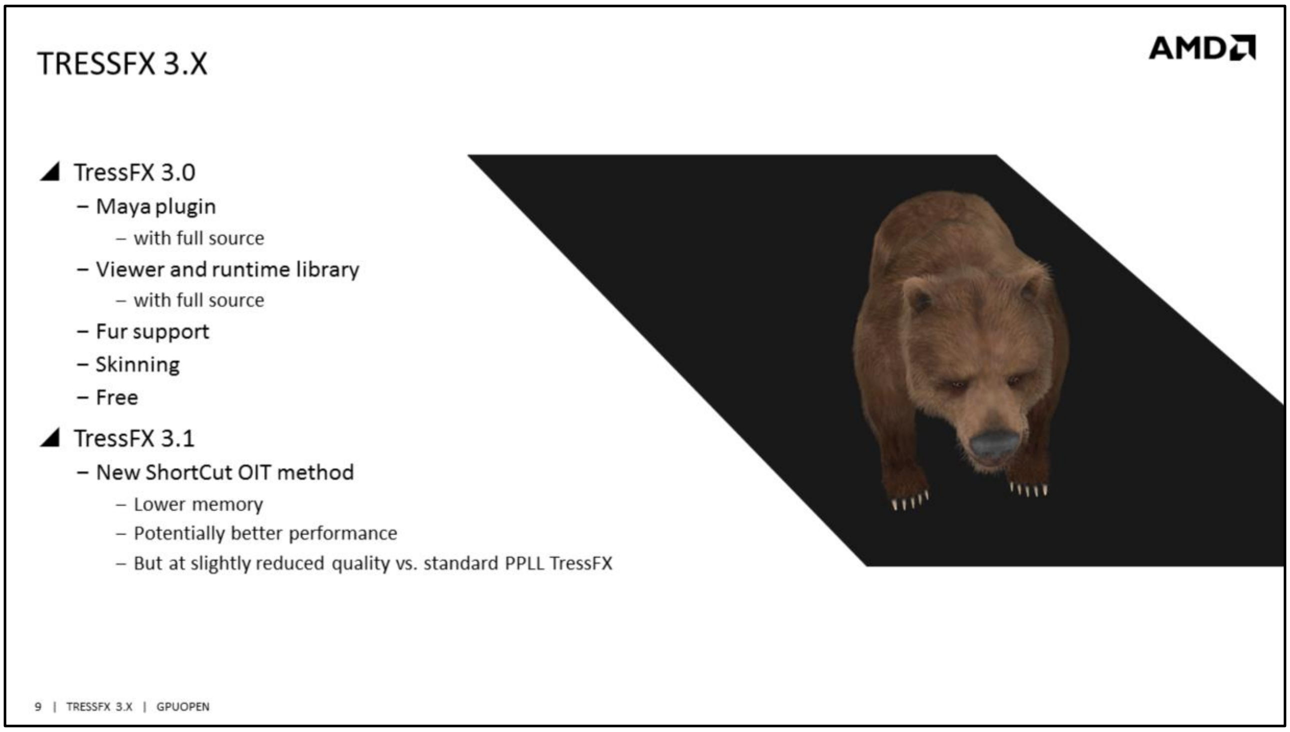


TRESSFX SAMPLE HISTORY

- TressFX 1.0
  - Example implementation (with full source)
- TressFX 2.x
  - Simulation performance improvements
    - Master and slave strands
    - Compute shader optimizations
    - 1.6 ms → 0.3 ms
  - Rendering performance improvements
    - Deferred rendering
    - Distance adaptive LOD
    - Pixel shader optimizations
    - 1.7 ms → 1.2 ms
    - With LOD, 0.3 ms (or lower) at a distance

8 | TRESSFX 3.X | GPUOPEN

~22k hair strands in the example on the slide
Updates to this point have been mostly focused on perf improvements



TRESSFX 3.X

- TressFX 3.0
  - Maya plugin
    - with full source
  - Viewer and runtime library
    - with full source
  - Fur support
  - Skinning
  - Free
- TressFX 3.1
  - New ShortCut OIT method
    - Lower memory
    - Potentially better performance
    - But at slightly reduced quality vs. standard PPLL TressFX

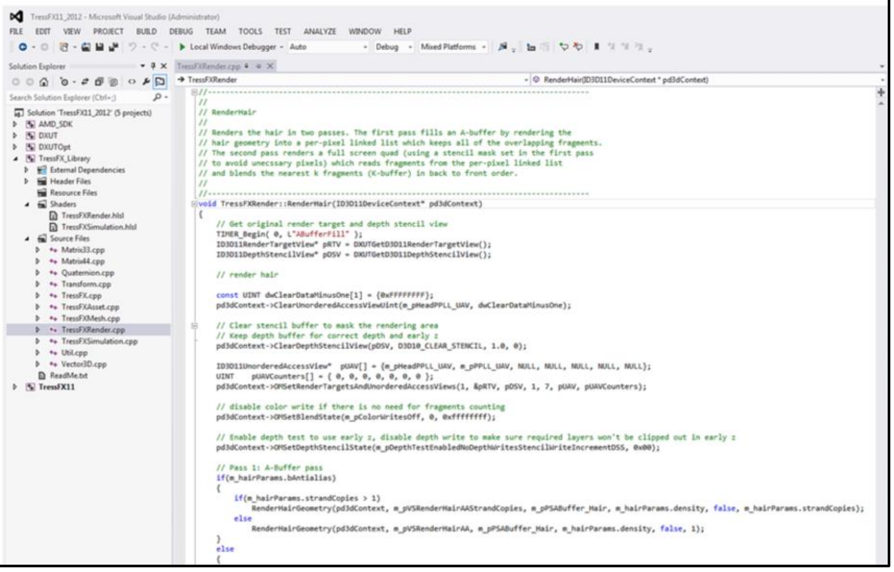
Whereas previous updates have been mostly focuses on perf, TressFX 3.0 is a larger update
It's a big update that we hope will be exciting and useful to game developers

TressFX 3.1 adds a new order-independent transparency (OIT) method called ShortCut

TRESSFX 3.X VIEWER AND RUNTIME LIBRARY

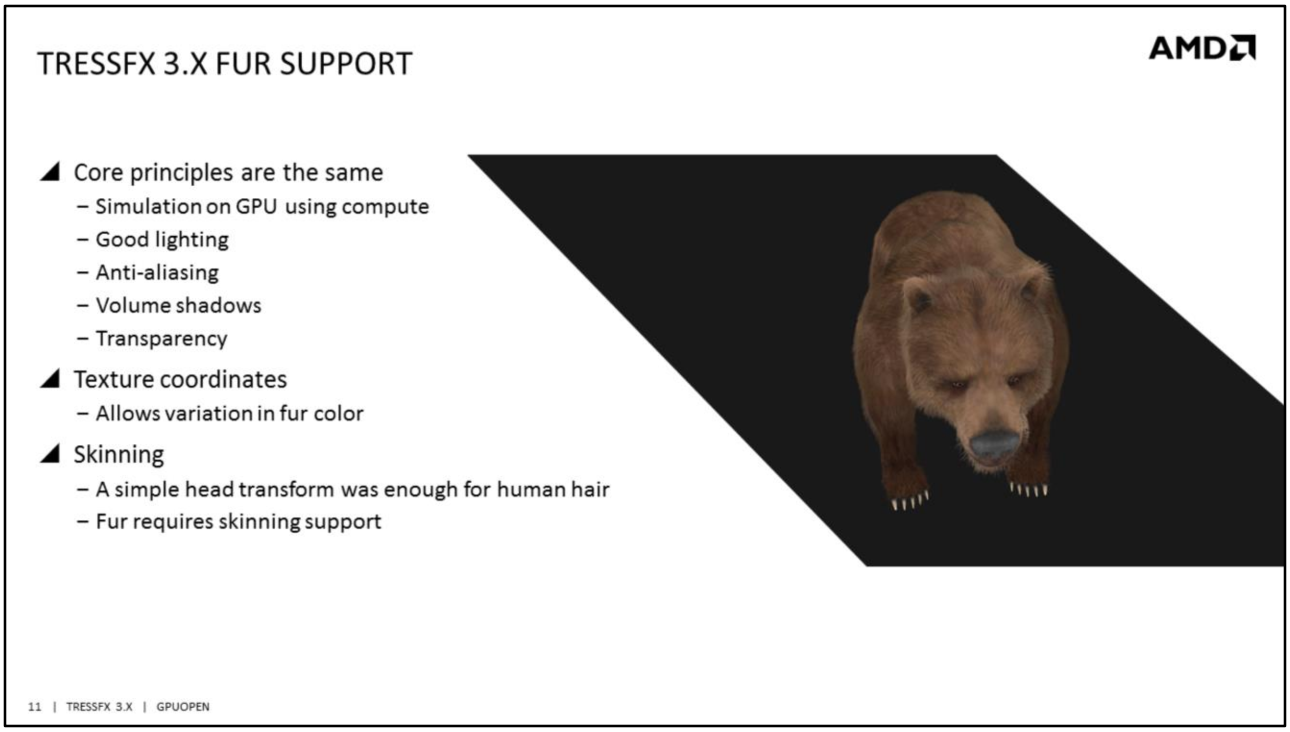
AMD

- Viewer provided
  - Preview TressFX assets
- Runtime reorganized into a library
  - Easier integration
- Full source
  - Free
  - No black boxes



```
//-----------------------------------------------------------------------------------
//
// RenderHair
//
// Renders the hair in two passes. The first pass fills an A-buffer by rendering the
// hair geometry into a per-pixel linked list which keeps all of the overlapping fragments.
// The second pass renders a full screen quad (using a stencil mask set in the first pass
// to avoid unecssary pixels) which reads fragments from the per-pixel linked list
// and blends the nearest k fragments (K-buffer) in back to front order.
//
//-----------------------------------------------------------------------------------
void TressFXRender::RenderHair(ID3D11DeviceContext* pd3dContext)
{
    // Get original render target and depth stencil view
    TIMER_Begin( 0, L"ABufferFill" );
    ID3D11RenderTargetView* pRTV = DXUTGetD3D11RenderTargetView();
    ID3D11DepthStencilView* pDSV = DXUTGetD3D11DepthStencilView();

    // render hair

    const UINT dwClearDataMinusOne[1] = {0xFFFFFFFF};
    pd3dContext->ClearUnorderedAccessViewUint(m_pHeadPPLL_UAV, dwClearDataMinusOne);

    // Clear stencil buffer to mask the rendering area
    // Keep depth buffer for correct depth and early z
    pd3dContext->ClearDepthStencilView(pDSV, D3D10_CLEAR_STENCIL, 1.0, 0);

    ID3D11UnorderedAccessView*  pUAV[] = {m_pHeadPPLL_UAV, m_pPPLL_UAV, NULL, NULL, NULL, NULL, NULL};
    UINT    pUAVCounters[] = { 0, 0, 0, 0, 0, 0, 0 };
    pd3dContext->OMSetRenderTargetsAndUnorderedAccessViews(1, &pRTV, pDSV, 1, 7, pUAV, pUAVCounters);

    // disable color write if there is no need for fragments counting
    pd3dContext->OMSetBlendState(m_pColorWritesOff, 0, 0xffffffff);

    // Enable depth test to use early z, disable depth write to make sure required layers won't be clipped out in early z
    pd3dContext->OMSetDepthStencilState(m_pDepthTestEnabledNoDepthWritesStencilWriteIncrementDSS, 0x00);

    // Pass 1: A-Buffer pass
    if(m_hairParams.bAntialias)
    {
        if(m_hairParams.strandCopies > 1)
            RenderHairGeometry(pd3dContext, m_pVSRenderHairAAStrandCopies, m_pPSABuffer_Hair, m_hairParams.density, false, m_hairParams.strandCopies);
        else
            RenderHairGeometry(pd3dContext, m_pVSRenderHairAA, m_pPSABuffer_Hair, m_hairParams.density, false, 1);
    }
    else
    {
```

Viewer provides an example of how to hook into the runtime
And then also lets you preview your exported results from Maya



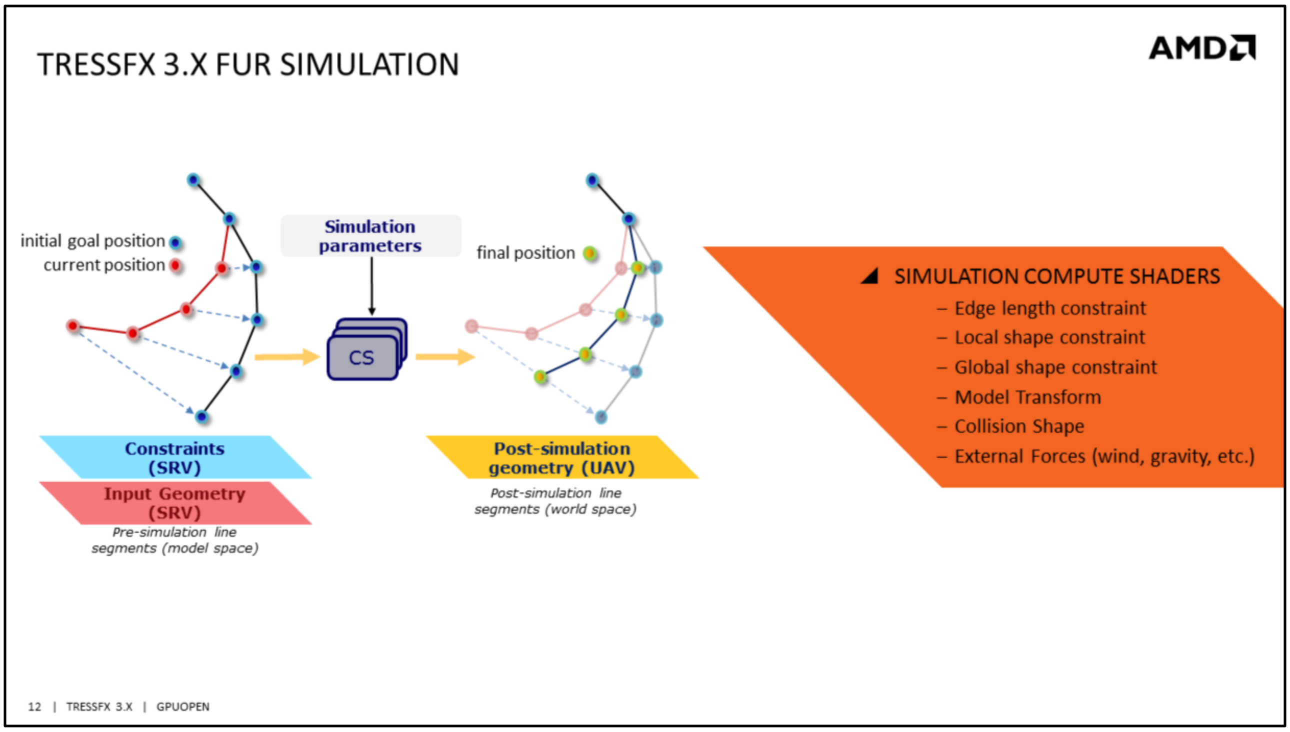
TRESSFX 3.X FUR SUPPORT

AMD

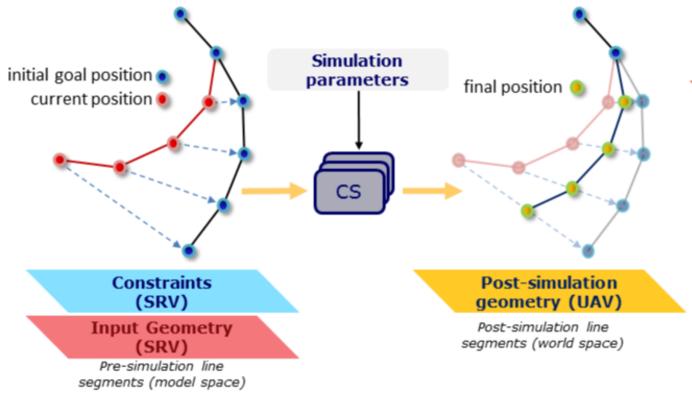
- Core principles are the same
  - Simulation on GPU using compute
  - Good lighting
  - Anti-aliasing
  - Volume shadows
  - Transparency
- Texture coordinates
  - Allows variation in fur color
- Skinning
  - A simple head transform was enough for human hair
  - Fur requires skinning support

11 | TRESSFX 3.X | GPUOPEN

Core principles are the same as for hair
But does require some additional features
In existing implementations of TressFX, the hair is one color.
Fur color varies across the animal’s body
And need animation support



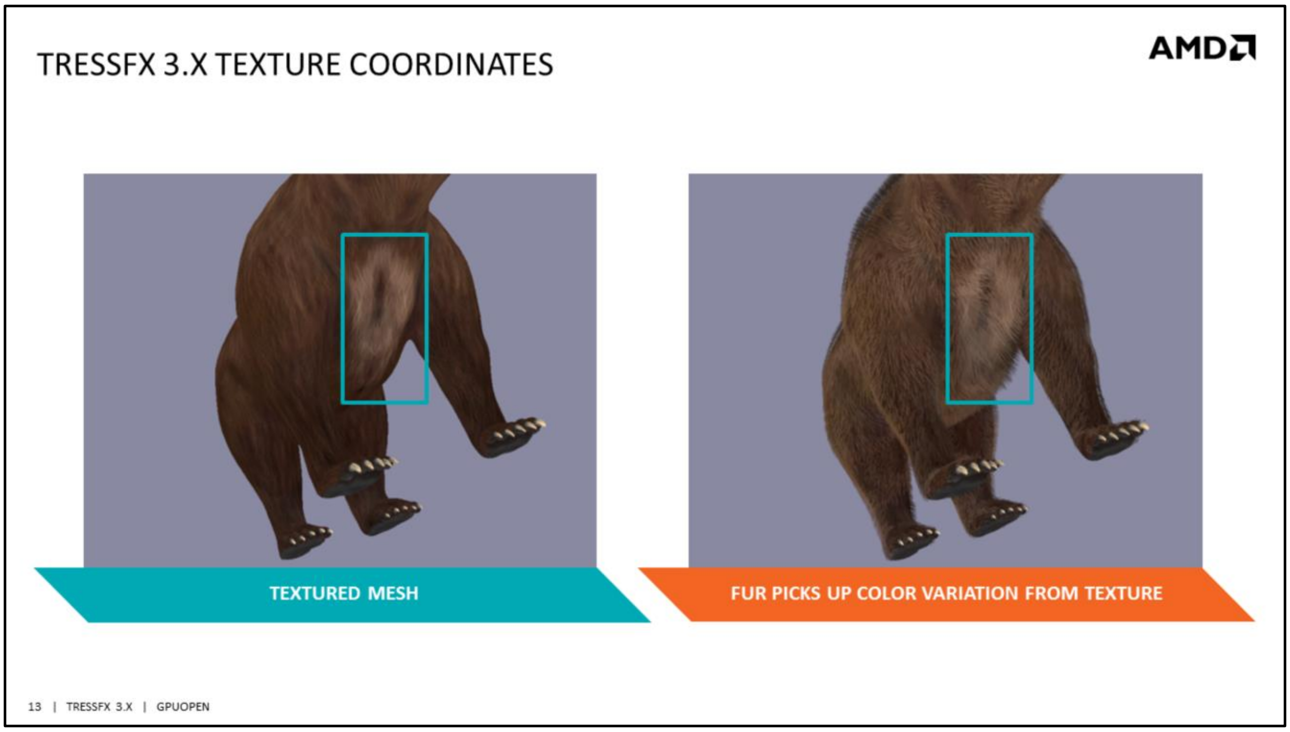
TRESSFX 3.X FUR SIMULATION



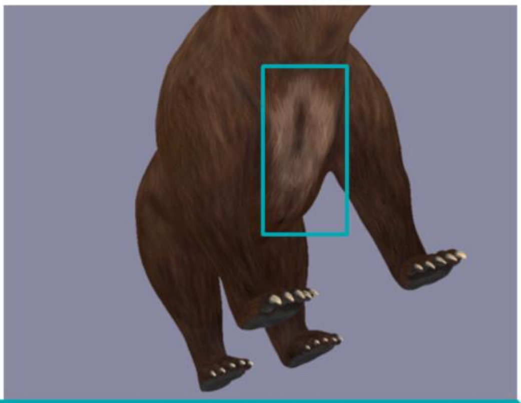
- SIMULATION COMPUTE SHADERS
  - Edge length constraint
  - Local shape constraint
  - Global shape constraint
  - Model Transform
  - Collision Shape
  - External Forces (wind, gravity, etc.)

12 | TRESSFX 3.X | GPUOPEN

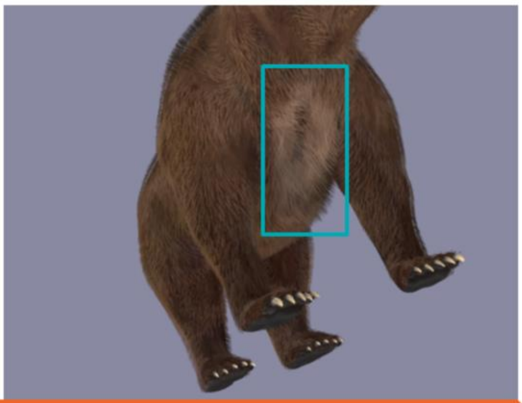
Global shape constraint and collision shape might not be needed for fur



TRESSFX 3.X TEXTURE COORDINATES

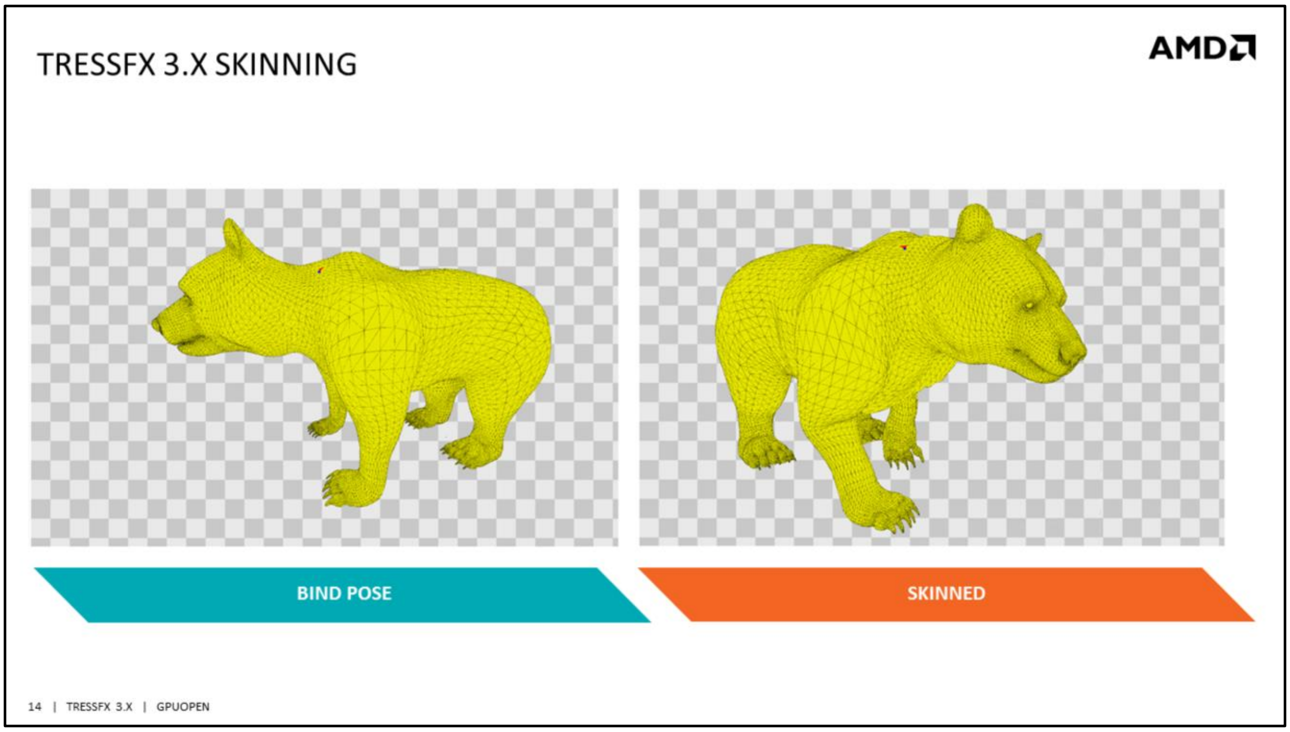


TEXTURED MESH



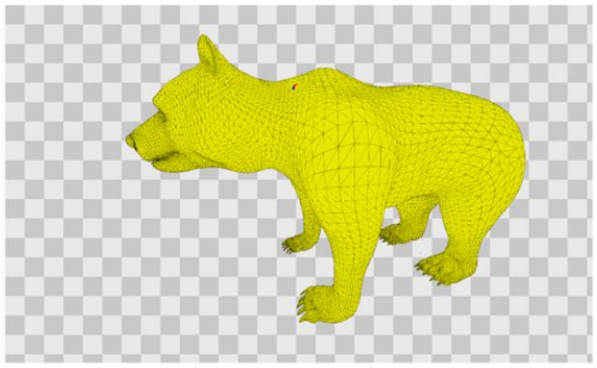
FUR PICKS UP COLOR VARIATION FROM TEXTURE

Fur picks up color from the underlying texture on the model



TRESSFX 3.X SKINNING

AMD

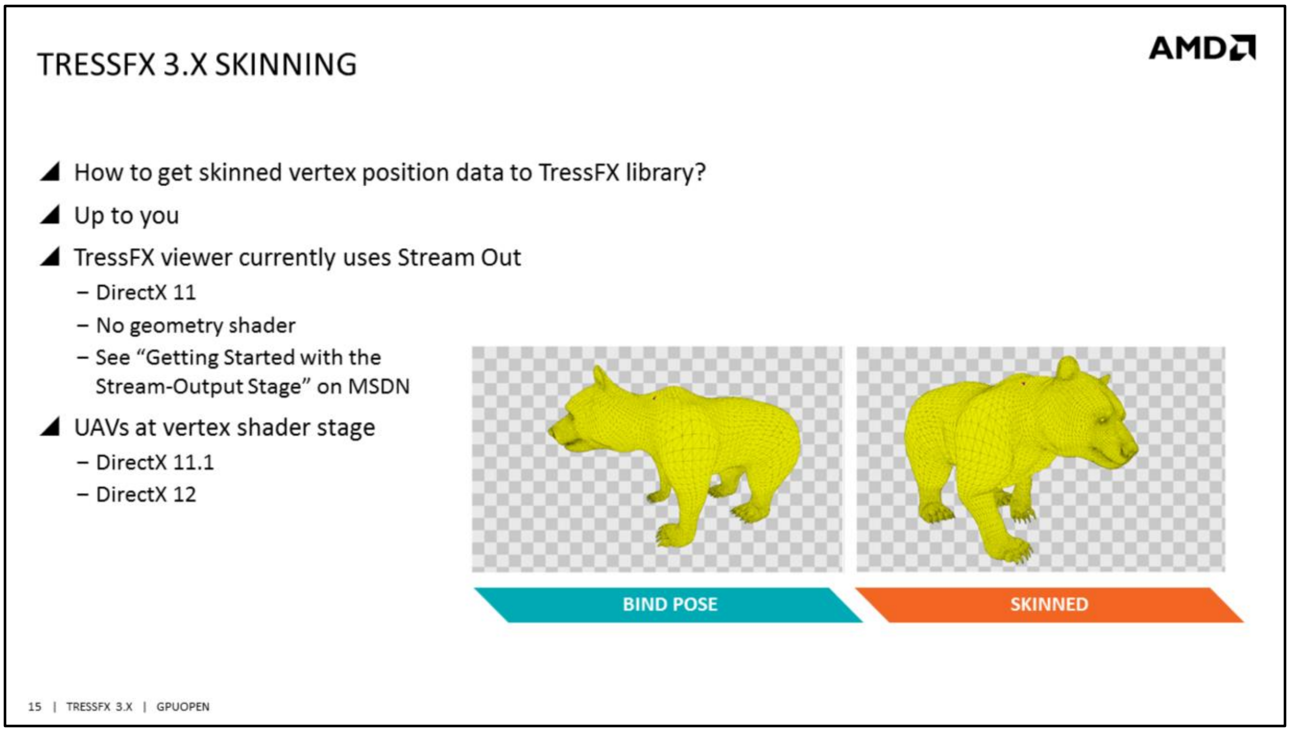


BIND POSE

SKINNED

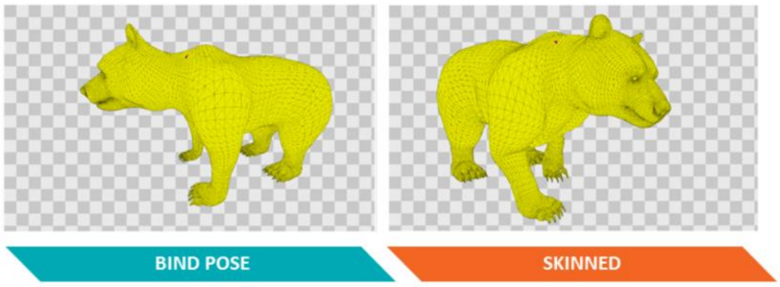
14 | TRESSFX 3.X | GPUOPEN

Game engine does animation update and vertex skinning as usual
Then, bind pose vertex positions and current skinned vertex positions are fed into the TressFX runtime



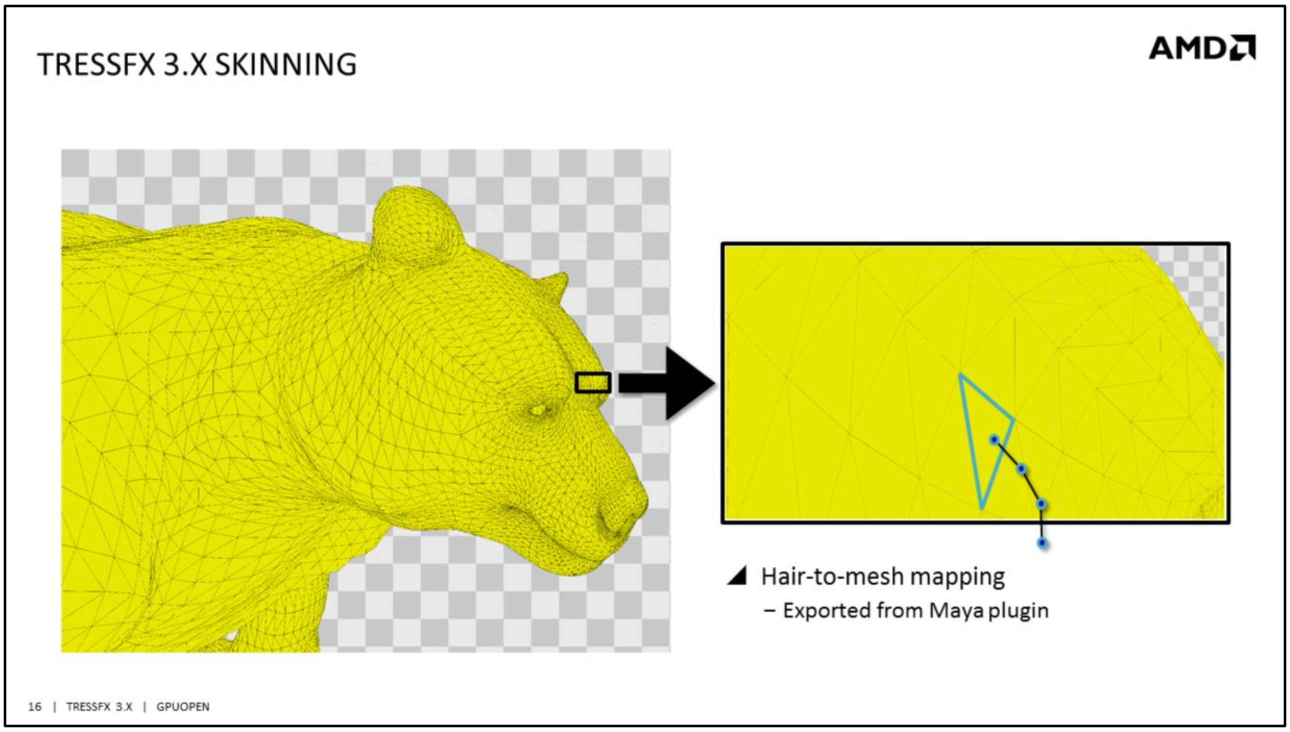
TRESSFX 3.X SKINNING

- How to get skinned vertex position data to TressFX library?
- Up to you
- TressFX viewer currently uses Stream Out
  - DirectX 11
  - No geometry shader
  - See “Getting Started with the Stream-Output Stage” on MSDN
- UAVs at vertex shader stage
  - DirectX 11.1
  - DirectX 12



15 | TRESSFX 3.X | GPUOPEN

To use the SO stage without using a geometry shader, call ID3D11Device::CreateGeometryShaderWithStreamOutput and pass a pointer to a vertex shader to the pShaderBytecode parameter.



TRESSFX 3.X SKINNING

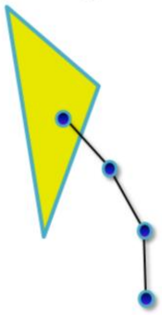
- Hair-to-mesh mapping
  - Exported from Maya plugin

Okay, so the engine does its animation and skinning update as usual. Then what?
The TressFX data from the exporter has hair-to-mesh mapping. Specifies the mesh triangle to which a hair strand belongs.

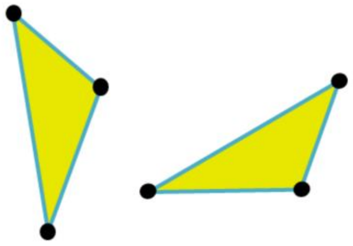
TRESSFX 3.X SKINNING

AMD

- Use hair strand index to get triangle index

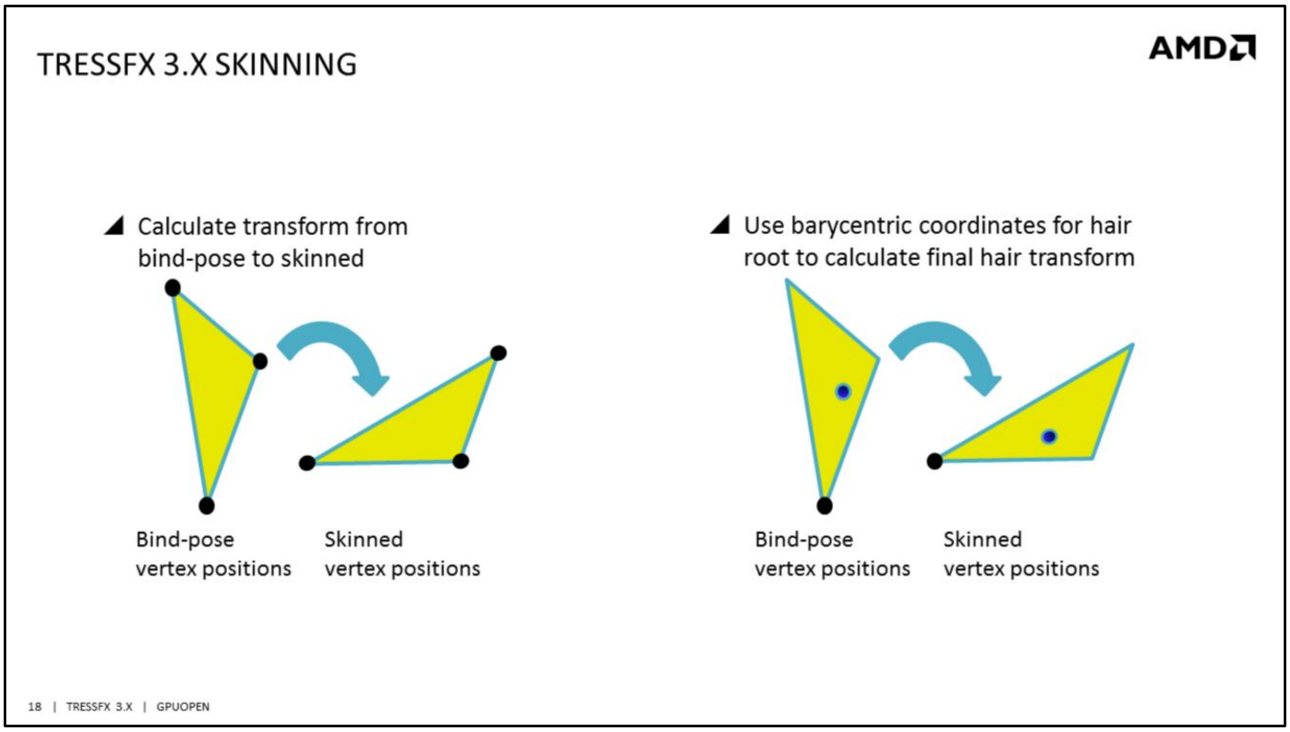


- Use triangle index to get bind-pose verts and skinned verts



Bind-pose vertex positions

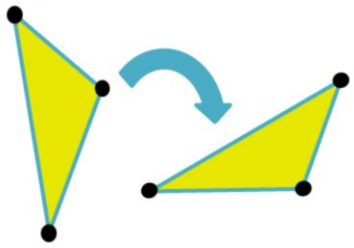
Skinned vertex positions



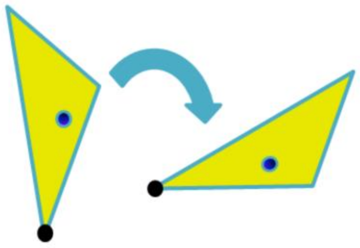
TRESSFX 3.X SKINNING

AMD

- Calculate transform from bind-pose to skinned

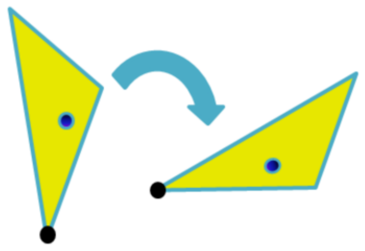


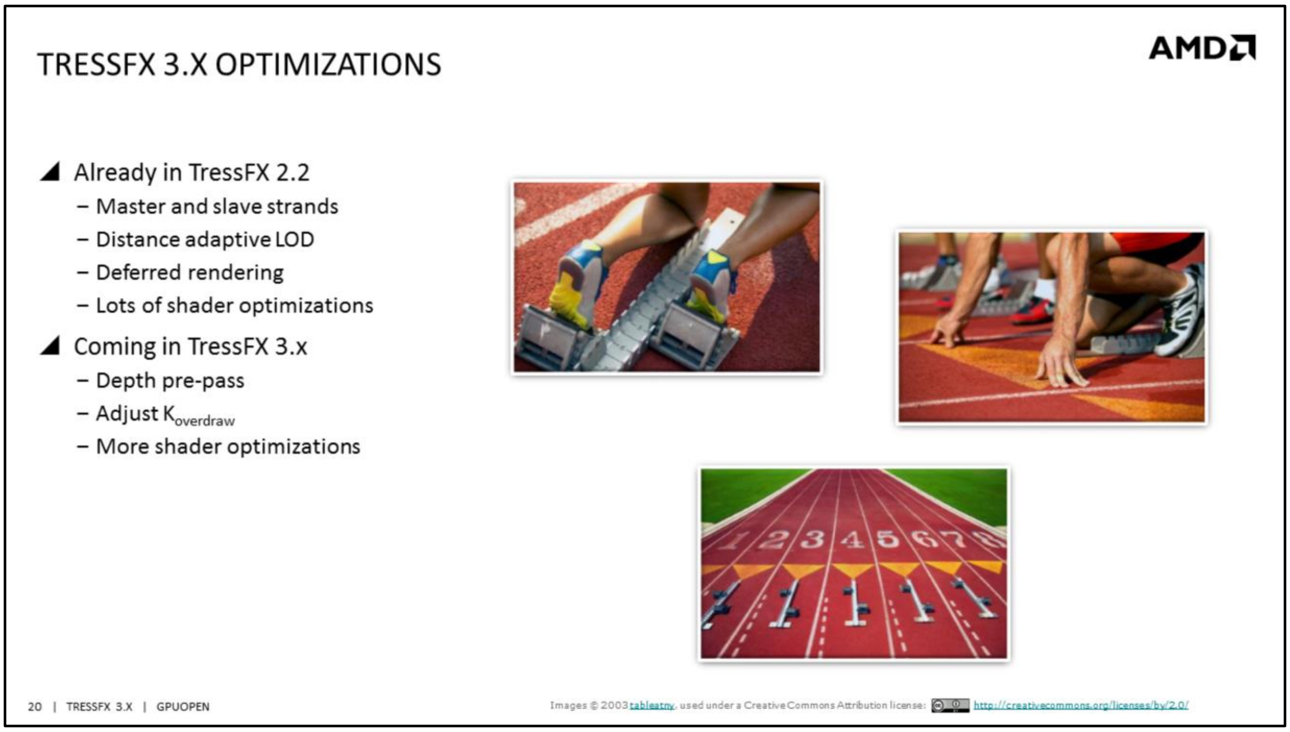
- Use barycentric coordinates for hair root to calculate final hair transform



TRESSFX 3.X SKINNING

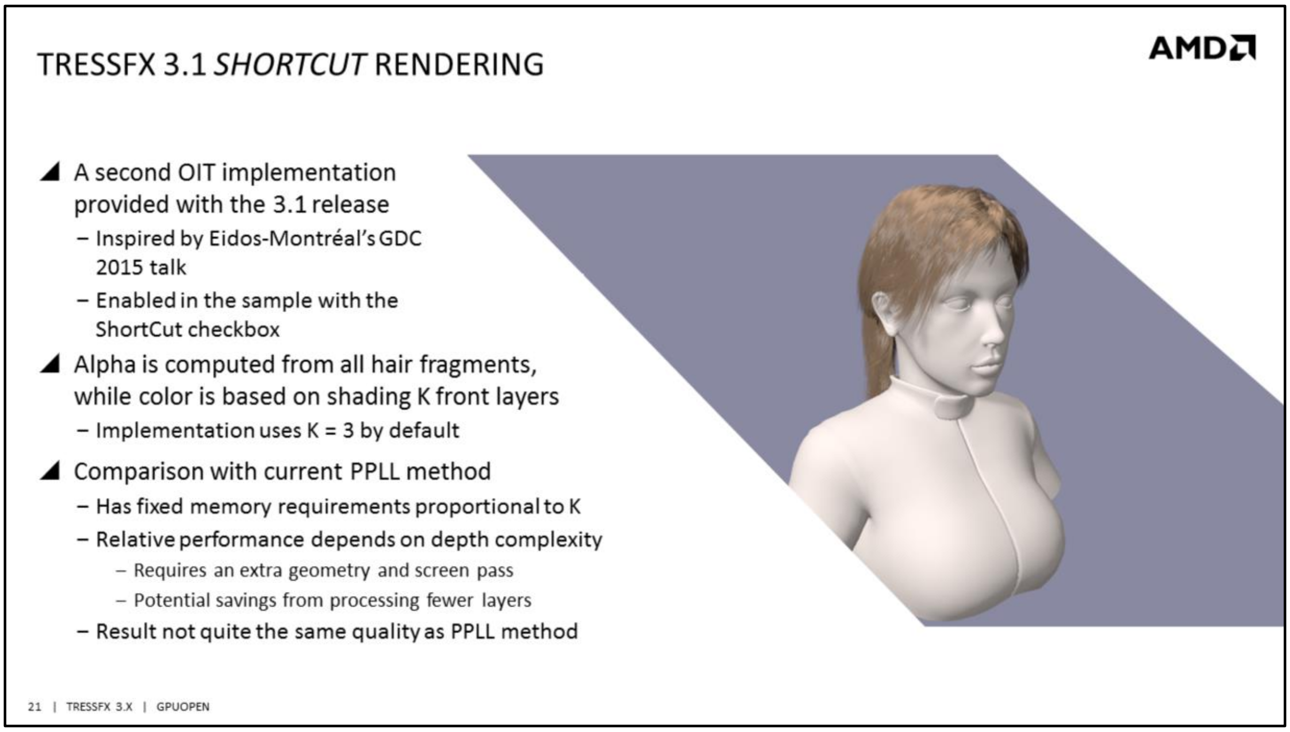
- Yeah, okay, but why not just skin the hair directly?
- This way doesn't impose any requirements on how the game engine does the animation update
  - Morph targets/blend shapes
  - Whatever, we just need the updated vertex positions
- But may code a fast path for ordinary skinning with max 4 bones





TRESSFX 3.X OPTIMIZATIONS

- Already in TressFX 2.2
  - Master and slave strands
  - Distance adaptive LOD
  - Deferred rendering
  - Lots of shader optimizations
- Coming in TressFX 3.x
  - Depth pre-pass
  - Adjust $K_{overdraw}$
  - More shader optimizations



TRESSFX 3.1 *SHORTCUT* RENDERING

AMD

- A second OIT implementation provided with the 3.1 release
  - Inspired by Eidos-Montréal's GDC 2015 talk
  - Enabled in the sample with the ShortCut checkbox
- Alpha is computed from all hair fragments, while color is based on shading K front layers
  - Implementation uses K = 3 by default
- Comparison with current PPLL method
  - Has fixed memory requirements proportional to K
  - Relative performance depends on depth complexity
    - Requires an extra geometry and screen pass
    - Potential savings from processing fewer layers
  - Result not quite the same quality as PPLL method

21 | TRESSFX 3.X | GPUOPEN

- Eidos-Montréal talk available at http://developer.amd.com/resources/documentation-articles/conference-presentations/
- Four main steps of ShortCut:
  - Render hair geometry, using a sequence of InterlockedMin calls to update the list of nearest fragments, while computing an overall alpha.
  - Screen space pass that puts the *k*th nearest depth in the depth buffer for early z culling in the next step.
  - Render hair geometry again. Shade the fragment and write or blend the color (depending on variant). [earlydepthstencil] focuses shading cost on the front *k*.
  - Screen space pass that does the final blending.
- Implementation supports *k* = 2 or 3

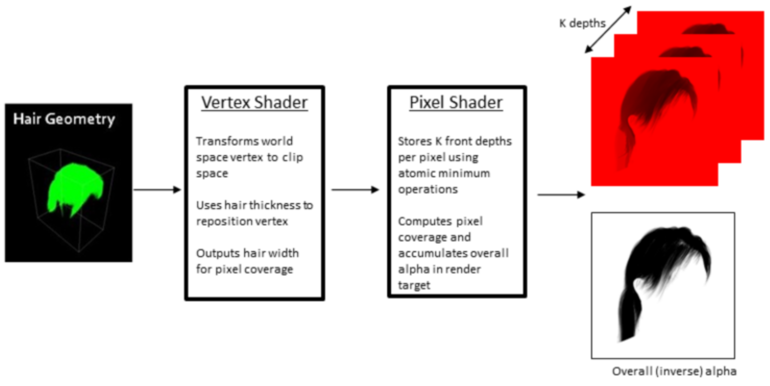
SHORTCUT DETAILS: MEMORY REQUIREMENTS

- Resources required
  - `FragmentDepthsTexture`: Screen-sized texture array that stores K FP32 values per pixel for the front layer depth values
  - `AccumInvAlpha`: Screen-sized R16F texture that accumulates total alpha for hair pixels
  - `FragmentColorsTexture`: Screen-sized FP16 RGBA texture containing hair color and alpha sums

- Total added memory cost is width x height x (K x 4 + 10) bytes
  - Given K = 3, added memory is 22 bytes per pixel

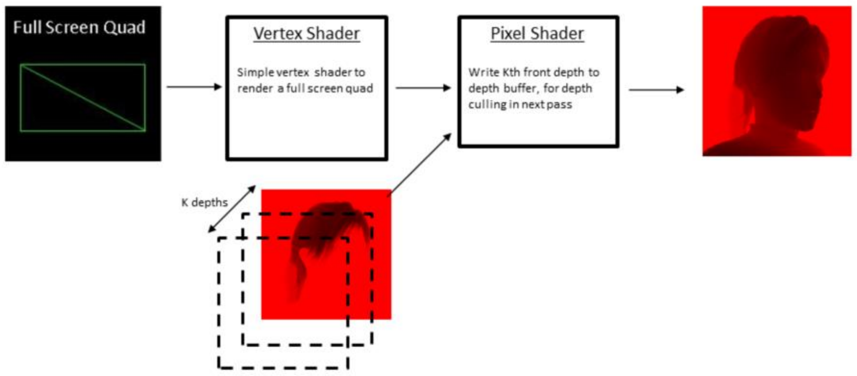
SHORTCUT DETAILS: RENDER PASSES

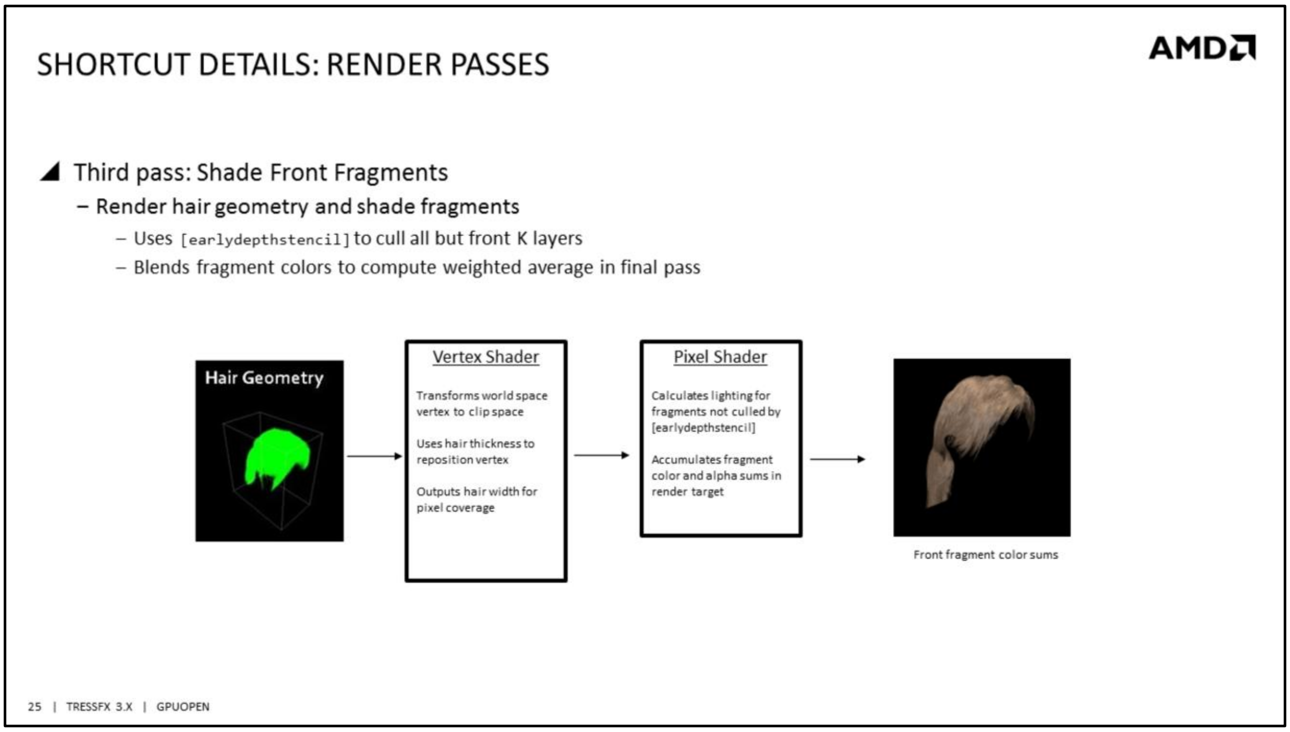
- First pass: Alpha and K Front Depths
  - Render hair geometry to find K front depth values and accumulate overall alpha
    - K minimum depth values found using sequence of `InterlockedMin` operations
    - Multiplicative blend of (1 – alpha) into render texture


Hair Geometry
Vertex Shader
Transforms world space vertex to clip space
Uses hair thickness to reposition vertex
Outputs hair width for pixel coverage
Pixel Shader
Stores K front depths per pixel using atomic minimum operations
Computes pixel coverage and accumulates overall alpha in render target
K depths
Overall (inverse) alpha

SHORTCUT DETAILS: RENDER PASSES

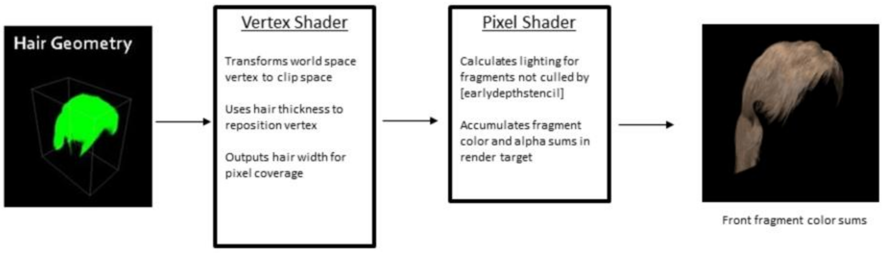
- Second pass: Write Depth Buffer
  - Render full-screen quad to write Kth depth value into depth buffer for culling in the next pass


Full Screen Quad
Vertex Shader
Simple vertex shader to render a full screen quad
Pixel Shader
Write Kth front depth to depth buffer, for depth culling in next pass
K depths



SHORTCUT DETAILS: RENDER PASSES

- Third pass: Shade Front Fragments
  - Render hair geometry and shade fragments
    - Uses `[earlydepthstencil]` to cull all but front K layers
    - Blends fragment colors to compute weighted average in final pass



Vertex Shader

Transforms world space vertex to clip space

Uses hair thickness to reposition vertex

Outputs hair width for pixel coverage

Pixel Shader

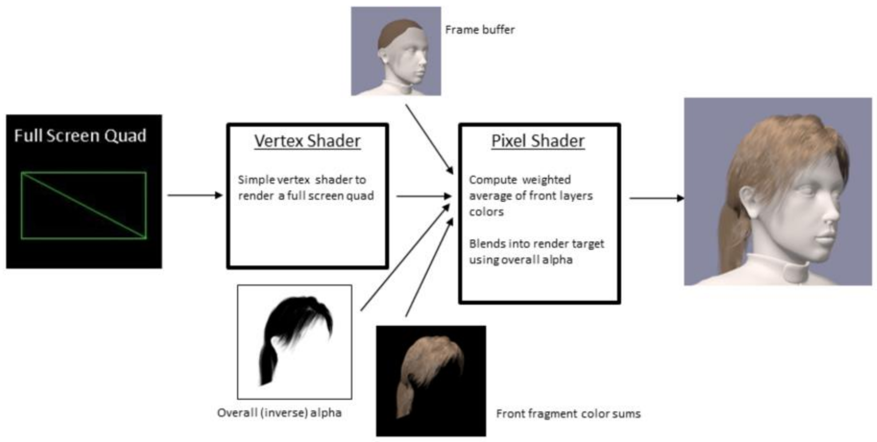
Calculates lighting for fragments not culled by [earlydepthstencil]

Accumulates fragment color and alpha sums in render target

Front fragment color sums

SHORTCUT DETAILS: RENDER PASSES

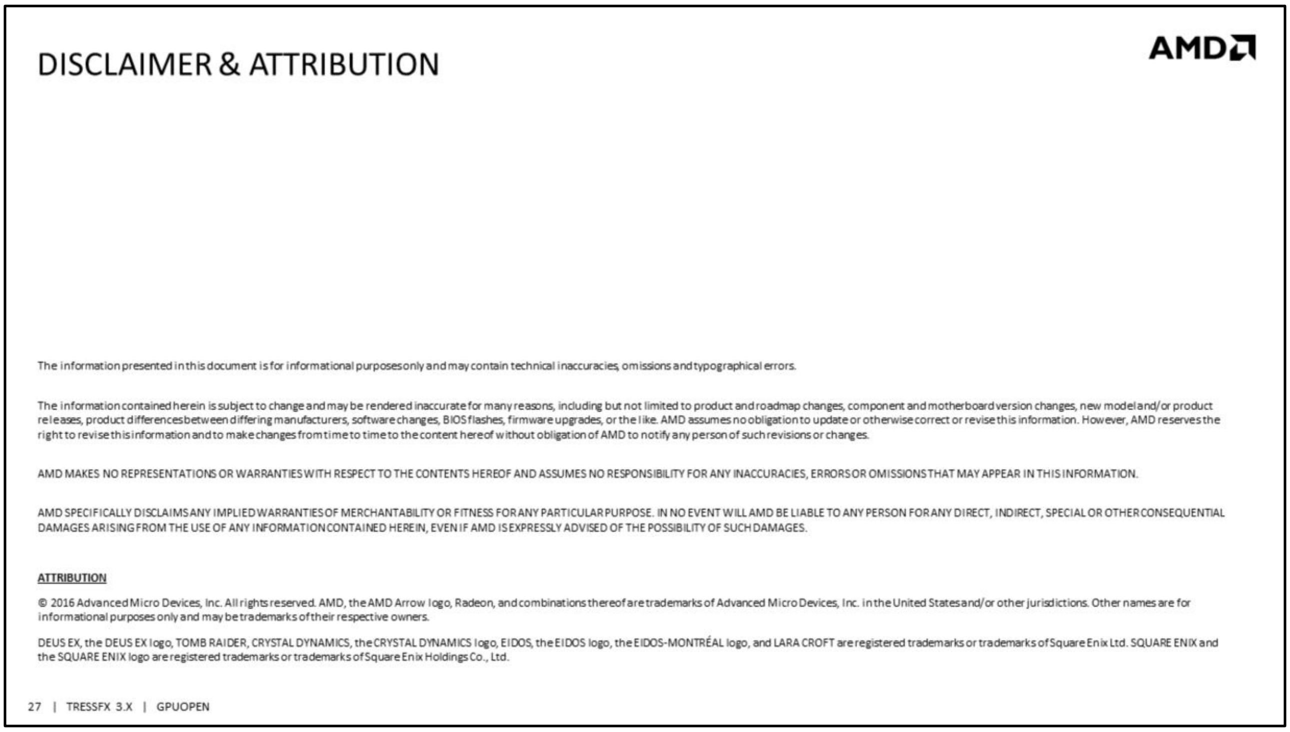
- Fourth pass: Blend with Frame Buffer
  - Render full-screen quad to composite hair into scene



Simple vertex shader to render a full screen quad

Compute weighted average of front layers colors

Blends into render target using overall alpha



DISCLAIMER & ATTRIBUTION

AMD

AMD