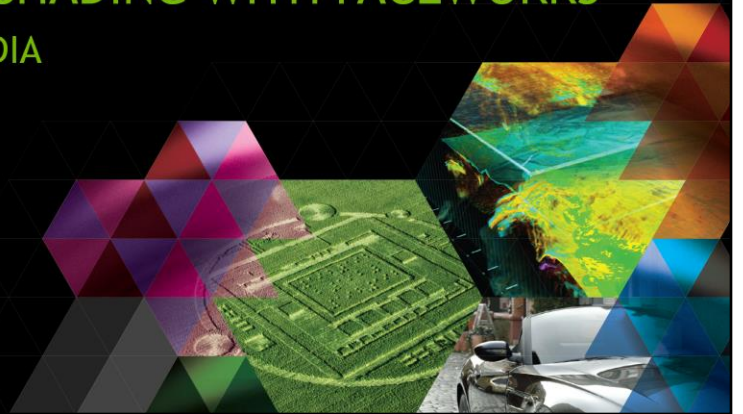



ADVANCED SKIN SHADING WITH FACEWORKS

Nathan Reed — NVIDIA

March 24, 2014






GPU TECHNOLOGY CONFERENCE

DIGITAL IRA

- Tech demo
- Collaboration with Dr. Paul Debevec at USC
- Lots of other inspiring work on skin & eyes
 - [Penner10], [Jimenez12], [Jimenez13]



First of all, what is FaceWorks?

Last year, NVIDIA showed off a tech demo called Digital Ira, which was a collaboration between NVIDIA and Dr. Paul Debevec's team at USC. In this project we took this extremely detailed head scan and performance capture, and basically threw every graphics technique we could think of at it, to render and animate this face as well as possible in real-time - and I think we had some good success.

Besides Digital Ira, there were also a lot of other great skin rendering papers and demos in the last few years, and all of these served as our inspiration for creating the FaceWorks API.

THE FACEWORKS API

- Middleware lib
- Integrate advanced skin shading in your game engine
- Goal: enable quality that matches Digital Ira
- This is a preview!

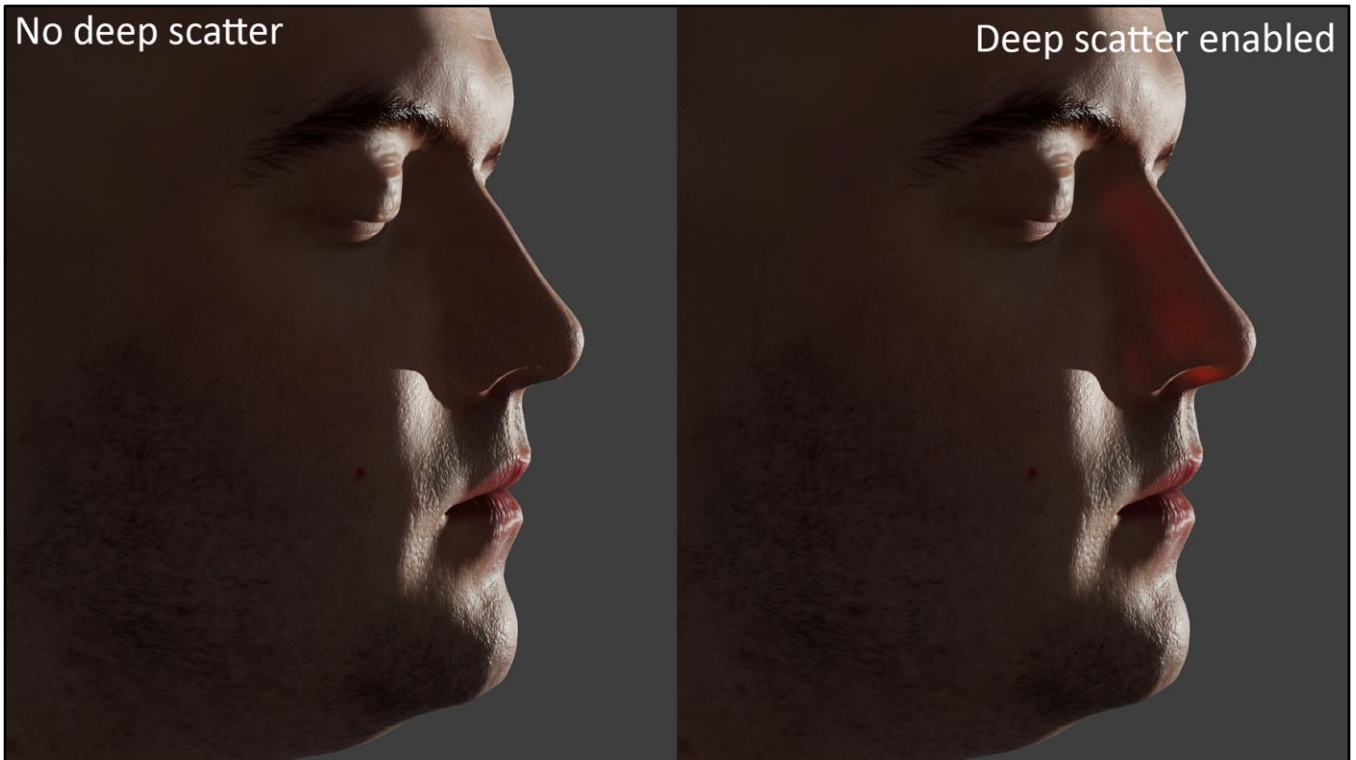
FaceWorks is a middleware library, under the GameWorks umbrella, that's designed to make it easy for game developers and engine developers to integrate high-quality skin shading into their games, with the goal of eventually enabling you to match the rendering quality of Digital Ira. FaceWorks is still under development, but we wanted to give you a preview today of some of the features it will enable you to add to your game engines, such as subsurface scattering. And more features are on their way in the future, such as a specular model for skin, and eye refraction and lighting.



Before we dive in, let's see some screenshots of what the FaceWorks library can do for your characters. I'm using Lee Perry-Smith's head scan here, which is a very high-quality model he kindly released as Creative Commons.

The first thing anyone ever talks about when skin shading comes up is subsurface scattering, and this talk is no exception! Subsurface scattering is a physical process in which light penetrates into the surface layers of the skin, bounces around and comes back out at a different point. Without it, skin looks very hard and unnatural, almost like a statue carved out of stone, as you see on the left here.

FaceWorks provides an implementation of subsurface scattering based on a SIGGRAPH paper from a couple years ago by Eric Penner, called "Pre-Integrated Skin Shading". This technique has the advantage of requiring only a single rendering pass, as opposed to expensive multi-pass solutions like texture-space diffusion; this makes the preintegrated technique quite efficient and relatively easy to integrate into a game engine, yet it still does a great job of softening the appearance of the skin and reproducing its characteristic red glow in the shadows, as you see on the right.



Another thing that happens with subsurface scattering is that light can diffuse all the way through thin areas of a face, such as the nose and ears. In FaceWorks, we refer to this feature as “deep scatter”, and we simulate it for direct light by using the shadow map to estimate the thickness of an object, and scaling the deep scatter effect based on the distance the light has to travel through the model. This produces the red glow you can see in areas like the nose and ears when they’re backlit. Deep scatter can work with your existing shadow maps, so it doesn’t require any additional rendering passes.



In this screenshot you can see both the subsurface scattering and the deep scatter effects working together. This also highlights how FaceWorks enables subsurface scattering for ambient lighting as well. The front of the face in both of these shots is lit entirely by image-based lighting, using a pre-convolved cubemap, and as you can see, subsurface scattering makes a big difference in the feeling of softness for that area as well. Meanwhile, on the right side of the face, there's strong direct lighting from behind, and deep scatter produces the red glow in the ear.

FEATURES

- Subsurface scattering (SSS)
 - One-pass solution based on [Penner10]
 - Both direct and ambient light
- Deep scatter
 - Based on thickness from shadow map [Green07]
 - Direct light only, so far
- D3D11 only, so far

These are the two main features that FaceWorks currently offers: an efficient, one-pass subsurface scattering solution that handles both direct light and ambient light, as well as a deep scatter solution for direct light based on estimating thickness from a shadow map. Currently we don't have a solution in place for ambient-light deep scatter, but we're still working on that.

Also, FaceWorks currently supports Windows and D3D11 only, although in the future we'd like to support other platforms and graphics APIs as well.

I'll talk about how these features work at the high level, then a bit later, I'll show a bit of what it would look like to actually integrate FaceWorks into your game engine.

SSS COMPONENTS

- Geometric curvature
- Normal maps
- Shadow edges
- Ambient

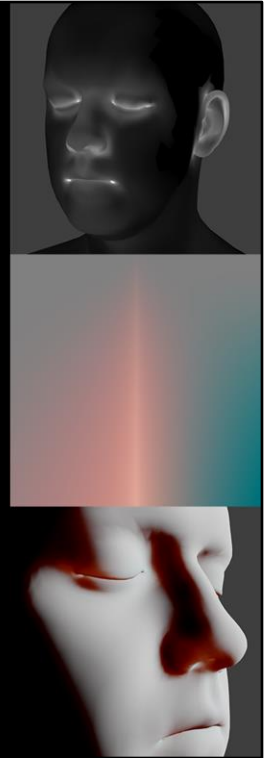
At the high level, FaceWorks breaks down subsurface scattering into four components, and handles each one individually.

First, there's the scattering due to the geometric curvature of the surface, where parts of the model with a small radius of curvature have more apparent scattering. Second, there's scattering through the small bumps and features in the normal map. Third, there's scattering across shadow edges, from lit into unlit areas. Those first three things take care of direct light, but finally we also need to handle ambient light.

Each of these components requires some data to be generated and marshalled by the game engine into the pixel shaders where FaceWorks is being used.

SSS 1/4: GEOMETRIC CURVATURE

- Precompute curvature per vertex (1 float)
 - Use bind pose of mesh
- Precompute a LUT (lookup texture)
 - For a given range of curvatures
 - Based on diffusion profile [d'Eon07]
- In PS, sample LUT based on curvature & N·L



For the curvature component, first we have to know what the curvature is at each point on the mesh. In FaceWorks, we do this by precomputing the curvature per vertex; we have a helper function that takes the mesh positions and normals, and calculates the curvature as a single float component per vertex, which you'll store in your vertex buffer. Currently we just do this for the bind pose of the mesh; in principle, as a mesh animates or deforms at runtime, its curvature values should change, but in practice it's hard to notice the effects of this, so we don't worry about it.

The other piece of data we need is a precomputed lookup texture, which encodes the scattering result for a range of curvatures and orientations to the light source, using the diffusion profile for human skin. FaceWorks provides a function to generate this texture as well.

Finally, at runtime, the curvature value should be interpolated from vertices down to pixels, and then in the pixel shader we sample that lookup texture using the curvature and $N \cdot L$.

SSS 2/4: NORMAL MAPS

- In PS, sample the normal map twice
 - Once as usual
 - Once with higher mip level
- Precompute mesh UV scale
- Combine lighting from both normals & curvature



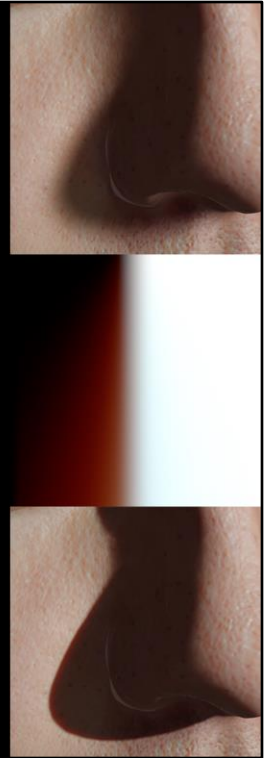
For the normal mapping component of SSS, in the pixel shader, we sample the normal map twice—once as usual, and once with a higher mip level, to get a normal that's blurred based on the SSS radius.

In order to work out the right mip level to sample at, we also have to know the UV scale of the mesh, i.e. how large the UV unit square is in world space. FaceWorks has a routine to compute the average UV scale over the whole mesh, which you can call from your art tools, and then store that UV scale alongside the mesh somewhere.

FaceWorks then combines both of these normals, as well as the curvature calculation from the previous step, to arrive at the total diffuse lighting.

SSS 3/4: SHADOWS

- Wide shadow filter
 - PCF, VSM, ESM, etc.
 - Not contact-hardening – want consistent filter radius
- Precompute a LUT
 - Sharpens shadows
 - Adds red glow in shadowed area
- In PS, sample LUT based on shadow result



The last component of direct light is shadows, or more precisely, shadow edges. With subsurface scattering, light bleeds across these edges from the lit into the unlit area. The trick here is to use a wide shadow filter; then we'll sharpen up the actual shadows, and use the remaining range of the wide filter to provide that red glow in the shadowed areas.

You can do this using the soft shadow technology of your choice, such as PCF, VSM, ESM or something else—you just don't want contact-hardening shadows for this, but rather a consistent filter radius everywhere. The details of the shadow filtering are up to you.

There's also another lookup texture that stores more precomputed scattering results for a range of shadow configurations, and in the pixel shader we sample this texture using the wide shadow filter result to arrive at the final shadow color.

SSS 4/4: AMBIENT

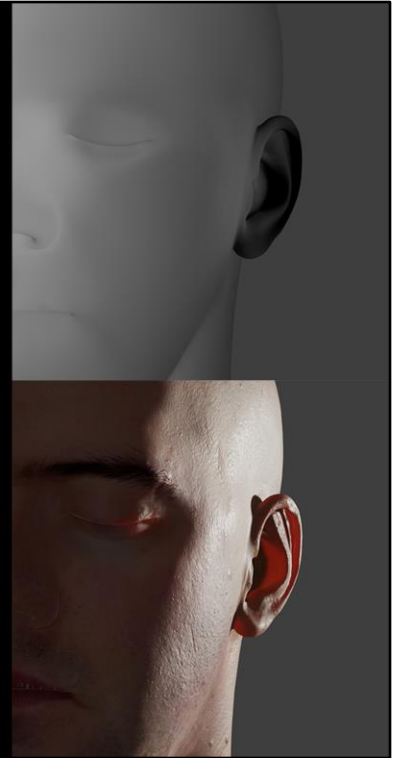
- In PS, generate 3 normal vectors
 - Combine blurred & unblurred normals
- Eval ambient light for each normal
- Combine 3 lighting values



Finally, we come to ambient light. In FaceWorks, we do this by evaluating the ambient light several times, for different normal vectors. We take the two normal map samples seen previously, and also combine them to generate a third, intermediate normal vector. Your pixel shader evaluates ambient light for each of those three normals—you could be using spherical harmonics, preconvolved cubemaps or some other system for ambient; it doesn't matter, as long as you can evaluate it for different normal vectors. Then FaceWorks combines the three lighting values together into one, consistent with the diffusion profile.

DEEP SCATTER

- Estimate thickness from shadow map
 - Helper functions for standard depth maps
 - Inward normal offset to fix silhouette edge issues
 - Poisson disk filter
- Apply falloff function using thickness & $N \cdot L$
- Multiply by texture for veins, bones etc.



Then there's deep scatter. While subsurface scattering models the diffusion of light locally along the surface, deep scatter lets us compute light diffusing through the bulk of a model, such as through the ears and nose.

Fortunately, this is a good deal simpler: the only hard part here is to get a good estimate of an object's thickness from a shadow map. FaceWorks provides some helper functions for the case of standard depth maps with either parallel or perspective projections, but you can also write your own code for this part, and you'll need to do so if you're using some other type of shadow map, such as VSM or ESM, or cascaded shadow maps. Unfortunately, we can't handle every possible case. :)

We found it's necessary to apply an inward normal offset to the shadow sampling position to fix silhouette edge issues. It's also worthwhile to use a wide filter, such as a Poisson disk filter, to soften up the thickness values; the deep scatter result can look unnaturally sharp otherwise.

Once you have the thickness, FaceWorks will apply a falloff based on thickness and $N \cdot L$, and return you the result. It's useful to also multiply the result by a texture map that represents veins, bones, and suchlike under the surface of the skin, as well as the overall color of the deep scatter effect.

RECAP

Precomputed

- Per-vertex curvature
- Per-mesh UV scale
- Curvature LUT
- Shadow LUT

Pixel Shader

- Interpolate curvature
- Sample normal twice
- Wide shadow filter
- Eval ambient 3 times
- Estimate thickness

That's the high-level of the advanced skin shading features that FaceWorks provides. Let me just recap quickly, since there are a bunch of moving parts here. First you'll need to generate some precomputed data in your art pipeline: for each mesh that will have FaceWorks applied, you'll need to generate the per-vertex curvature and per-mesh UV scale data. You'll also need to generate the two lookup textures, which can probably just be done once and then checked in as regular textures.

Then, in the pixel shader to actually render the model, you'll interpolate the per-vertex curvature, sample the normal map twice, evaluate shadows using a wide filter, evaluate the ambient light three times using three different normals, and estimate the thickness from the shadow map. Hand all that data off to FaceWorks and it'll compute the resultant diffuse lighting and deep scatter lighting, which you'll multiply by the diffuse color and light color as usual, and add in a specular model of your choice.

INTEGRATION

- C API & .dll, link into your engine & tools
 - Precompute data
 - Set graphics state per draw
- HLSL API, #include into your shaders
 - Evaluate lighting in PS

Now let's look briefly at what it would take to actually integrate FaceWorks into your game engine.

The FaceWorks API ships in two pieces. First of all, there's a standard C API and a .dll that you can link into your engine and tools, and redistribute with your game. This has all the routines to precompute the various data that we saw in the previous slides, and also a runtime API to set up the graphics state needed by the shaders.

The other piece is an HLSL API, which is a blob of code that you can #include into your shaders and will be compiled into them. This contains the routines you'll call from your pixel shader, to access all the data we've set up and actually evaluate the lighting equations.

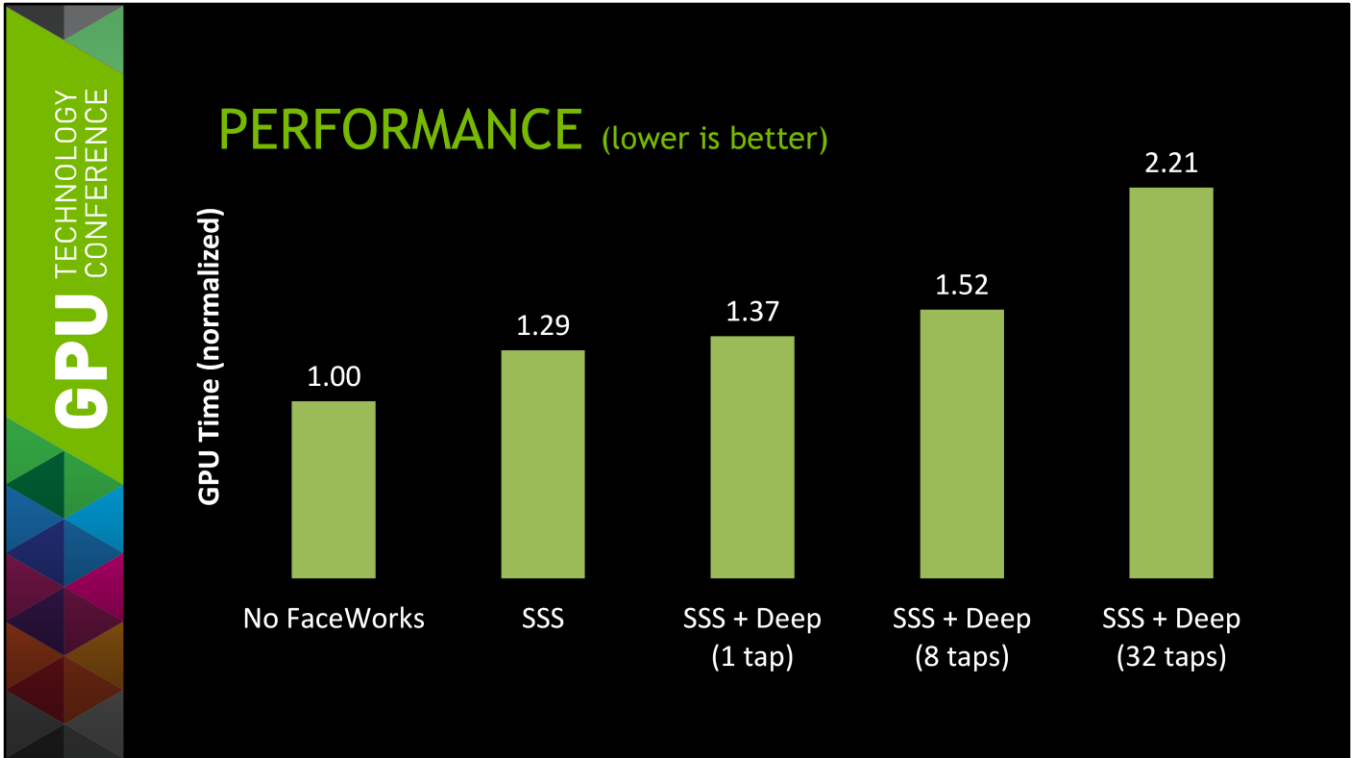
INTEGRATION

- Look up bind points using shader reflection
- Create context object
- Per draw: fill out config struct, set state

There's some glue code you'll need to write to marshall all the data and get it into the pixel shader. First of all, the FaceWorks shader code uses various internal resources such as constant buffers, textures, and samplers. When compiling a pixel shader that uses FaceWorks, you'll need to use the D3D11 shader reflection interface to look up the bind points for those resources, and store those alongside the compiled shader code somewhere.

Then, in your game engine when you initialize your renderer, you'll need to create a FaceWorks context object, which will create and manage the graphics resources using the D3D11 device.

Finally, at some point before you do a draw call, you'll fill out a configuration struct that contains the shader bind points as well as runtime parameters like the desired subsurface radius, and call the FaceWorks API to set up the graphics state for that draw. This will update constant buffers, bind things to the pipeline and so on.



To give you an idea of the relative performance of FaceWorks, here's the results of benchmarking a few scenarios. The graph shows GPU time, so lower is better, and the numbers have been normalized relative to the control test (on the far left) of simply rendering the mesh with standard diffuse and specular lighting, with no FaceWorks involved.

Turning on FaceWorks subsurface scattering raises the cost by about 30%, which is quite cheap considering how much of a visual benefit you get. It's also much cheaper than multi-pass texture-space techniques would be.

Adding deep scatter can increase the cost significantly, depending on how many taps you use for the shadow thickness estimate. The sweet spot for performance versus quality is probably somewhere around 8 taps. At that point the cost is still pretty reasonable—1.5x the base cost—but if you have GPU time to burn, you can even go to something like 32 taps, which is 2.2x the base cost but looks quite nice.

FUTURE WORK

- Ambient-light deep scatter
- Specular model for skin
 - Including occlusion
- Eye rendering
- Customizable diffusion profiles
- Support more APIs/platforms
 - OpenGL, Mobile, Consoles

As mentioned before, the FaceWorks API is still a work in progress, and there's still much more to be done.

We'd like to extend deep scatter to support ambient light; our current implementation only works for direct light.

Of course, specular lighting, including proper specular occlusion, is also a very important feature of the appearance of skin, and our library doesn't address that right now, but we'd like to do so in the future. Eyes are also a very important feature of faces and we're planning to add features for rendering realistic eyes.

The diffusion profile for human skin is hard-coded right now, and we'd like to make the library more versatile by enabling you to customize the diffusion profile, so you can simulate, say, creature skin that looks very different from human skin, or potentially even other translucent materials like cloth, paper, wax, or marble.

Finally, while today FaceWorks only runs on Windows and D3D11, we hope to add support for other APIs and platforms in the future, including OpenGL, mobile, and consoles.

FEEDING THE RENDERER

- Need high-quality models and textures for good results!
 - LPS head: 18K tris (before tess), 4K textures
- Normal map should not be softened
 - Need strong bumps for specular
 - The shader will soften the diffuse
- Head scans are great, if within your means

I also want to say a few words about the art requirements. Having great rendering technology is all well and good, but it won't achieve its potential unless it's fed with high-quality art. For FaceWorks to look really good, it needs very high-quality models and textures. The Lee Perry-Smith head scan I've been using for the screenshots has about 18K triangles before tessellation, and the diffuse and normal maps are both at 4K resolution.

The normal map for a shader like this should not be softened at all. It's tempting for artists to soften the normals if they don't have a good subsurface shader to work with, since the skin looks way too crunchy otherwise. However, a strong normal map is needed for specular to look right on skin, and the subsurface scattering will soften the diffuse component.

If it's within your means, high-quality head scans are a great resource; even if you don't use a scanned head directly in your game, it's still great for reference material, or as a starting point for artists to modify.

EXTRA GOODIES IN THE SAMPLE APP

- Physically-based shading
 - Blinn-Phong NDF, Schlick-Smith visibility, Schlick Fresnel
 - 2-lobe specular [Jimenez13]
- IBL with preconvolved cubemaps [Lagarde11]
- Filmic tonemapping [Hable10]
- Adaptive tessellation
 - More polygons where curvature is high

The FaceWorks sample app, in addition to simply demonstrating the FaceWorks API, has a few other goodies that you might be interested in. It has a pretty advanced physically-based shading implementation, including a 2-lobe specular model and preconvolved diffuse and specular cubemaps for IBL. It also uses filmic tonemapping, which just makes everything look better, and a form of tessellation based on screen-space error that adapts to the curvature of the mesh, adding more polygons where the curvature is high.

REFERENCES

- d'Eon & Luebke, "Advanced Techniques for Realistic Real-Time Skin Rendering", GPU Gems 3 (2007)
- Green, "Real-Time Approximations to Subsurface Scattering", GPU Gems 3 (2007)
- Hable, "Uncharted 2 HDR Lighting", GDC 2010
- Jimenez et al, "Separable Subsurface Scattering and Photorealistic Eyes Rendering", SIGGRAPH 2012
- Jimenez & von der Pahlen, "Next-Generation Character Rendering", GDC 2013
- Lagarde, "Adopting a physically based shading model", blog post (2011)
- Penner, "Pre-Integrated Skin Shading", SIGGRAPH 2010