NVIDIA®

Ogres and Fairies

Secrets of the NVIDIA Demo Team

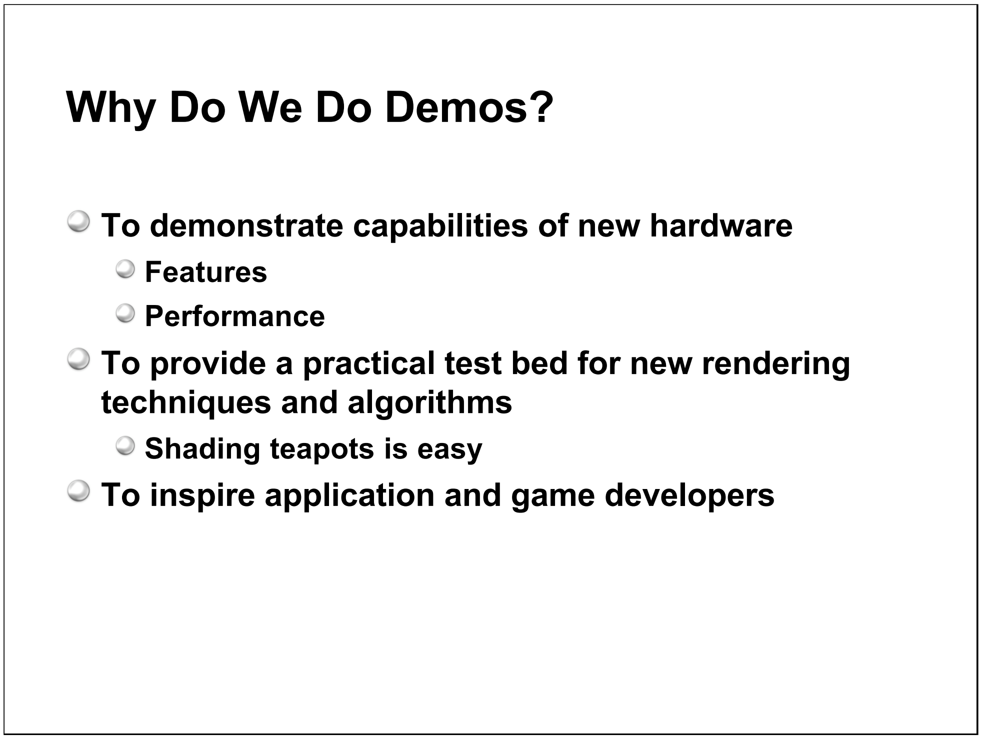
Overview

- Demo engine overview
- Procedural shading for aging effects in “Time Machine”
- Depth of field and post processing effects in “Toys”
- Subdivision surfaces and ambient occlusion shading in “Ogre”
- Advanced skin and hair rendering in “Dawn”
- Questions

The GeForce FX Demo Suite

- 4 demos for the launch of GeForce FX
  - “Dawn”
  - “Toys”
  - “Time Machine”
  - “Ogre” (Spellcraft Studio)

www.yeah the movie.de



Why Do We Do Demos?

- **To demonstrate capabilities of new hardware**
  - **Features**
  - **Performance**
- **To provide a practical test bed for new rendering techniques and algorithms**
  - **Shading teapots is easy**
- **To inspire application and game developers**

NVIDIA spends a lot of money on demos

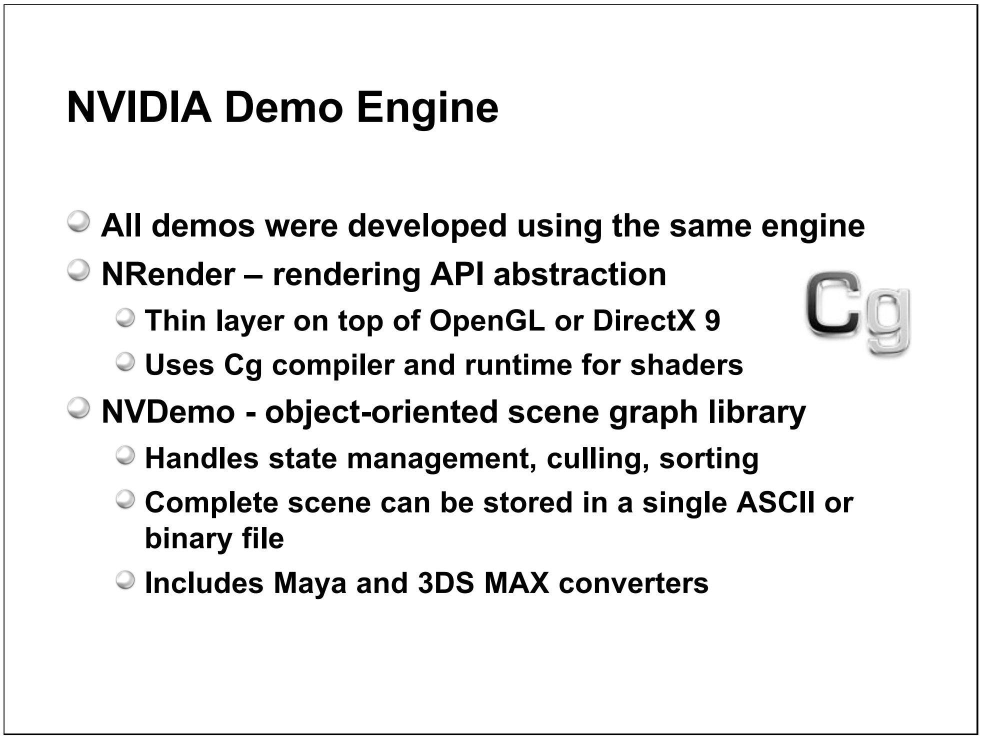
At launch there usually aren't many applications that take full advantage of the hardware

We are aware demos are not representative of games (often a single character, simple background).

Games have long development cycles, need to support a wide range of hardware

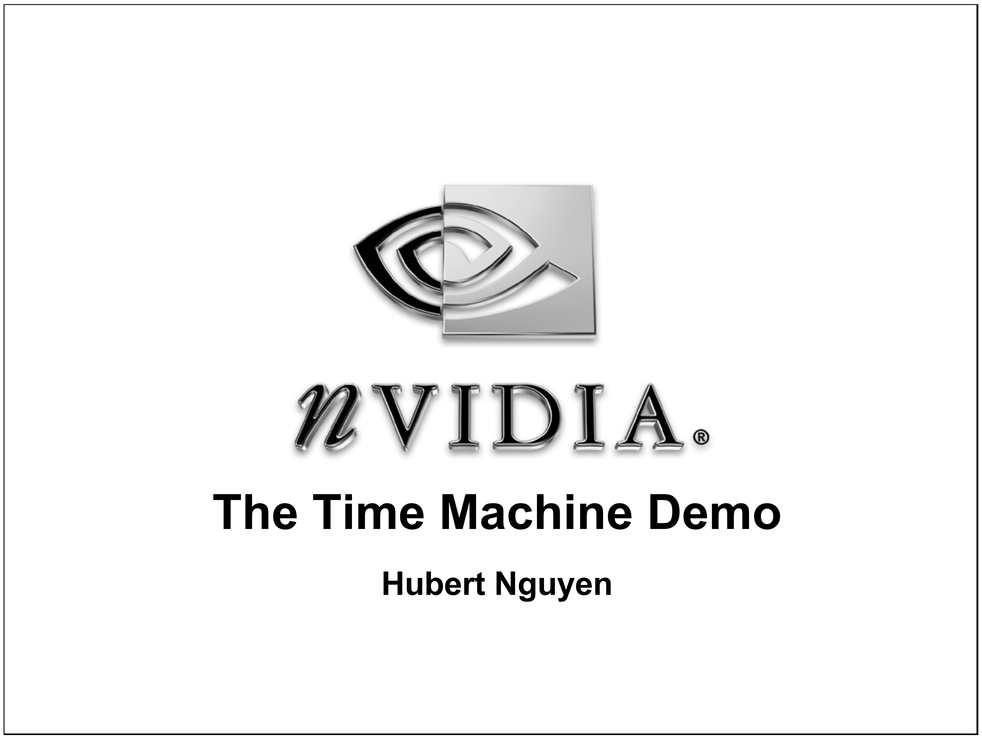
We have very early access to hardware

It's easy to do shaders on teapots, using real models is more complicated.



NVIDIA Demo Engine

- All demos were developed using the same engine
- NRender – rendering API abstraction
  - Thin layer on top of OpenGL or DirectX 9
  - Uses Cg compiler and runtime for shaders
- NVDemo - object-oriented scene graph library
  - Handles state management, culling, sorting
  - Complete scene can be stored in a single ASCII or binary file
  - Includes Maya and 3DS MAX converters



The Time Machine Demo

Hubert Nguyen

Goals of Time Machine

- Show the potential of a new architecture
  - More data
    - 16 texture inputs
    - 8 texture coordinate interpolators
  - Higher precision (128 bits)
  - More instructions (up to 1024)
    - Shading done in a single pass
  - Faster pixel processing
    - Higher clock speed

- Greater data access & faster processing

A truck ?

- Old pick-up trucks have a wide variety of surfaces.
  - Paint and rusting and oxidizing
  - Wood splintering and fading
  - Chromes being damaged and dirty
  - And more…

Live demo

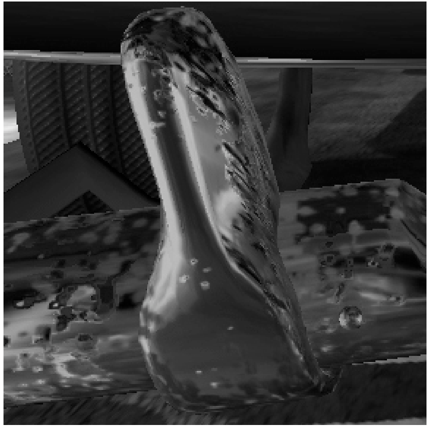
http://www.nvidia.com/object/demo_timemachine.html

A Simple “aging shader” : Chrome

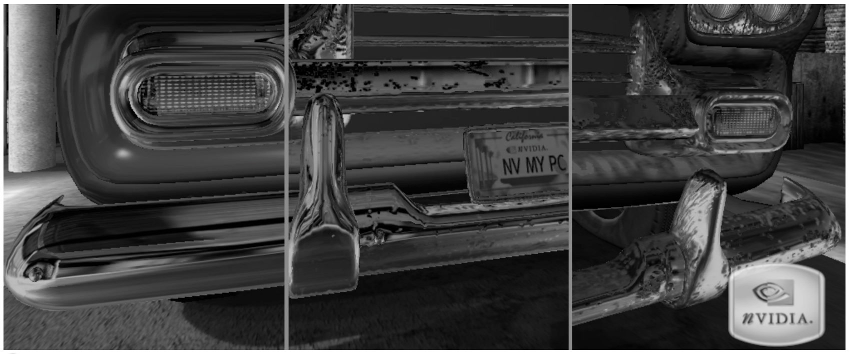
- Aging shaders are multi-layered shaders
  - Several stand-alone effects blended together by a function of time & space
- Case study : chrome
  - 2 layers :
    - Chrome (shiny) layer
    - Rust layer
  - Both are fully lit, bumped and shadowed
  - Each would barely fit on a DX8-class shader

Chrome : getting older

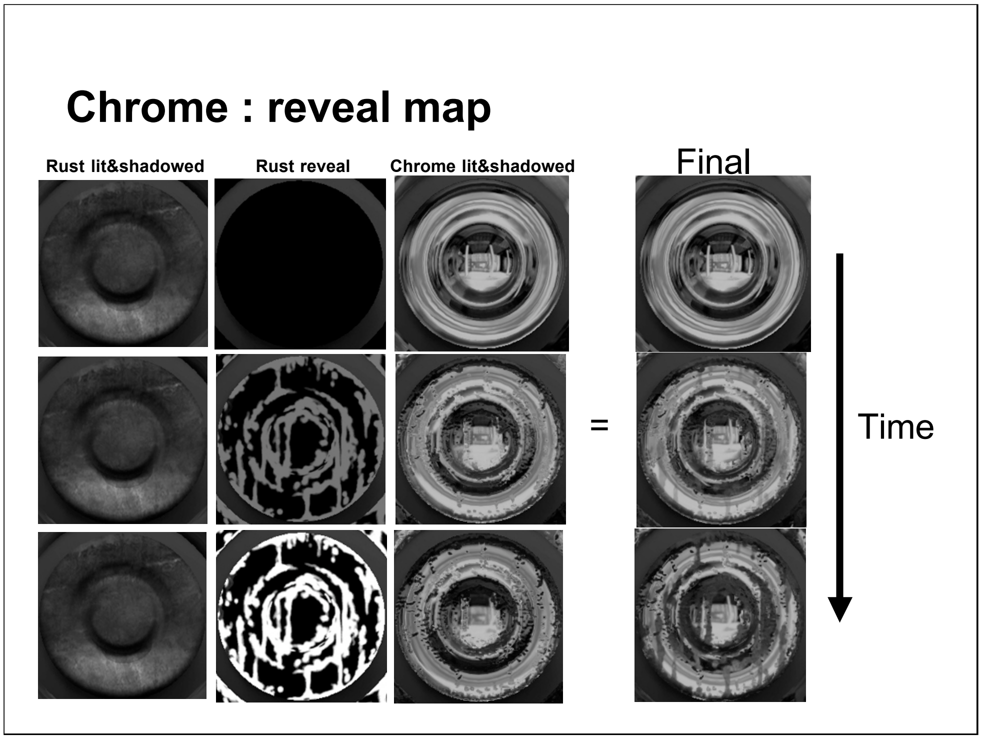
- Chrome still shines over the years
- Reflection fades slightly (dust, dirt, small damages)
- Bumps, scratches & rust shows up



Chrome: aging snapshots



- Full lighting, bump & shadows on all the layers
- Reflection blurred by blending two cube maps
- Bumpy reflection using EMBM, for performance
- “Reveal” texture pinpoints the rust location



Chrome : reveal map

Rust lit&shadowed

Rust reveal

Chrome lit&shadowed

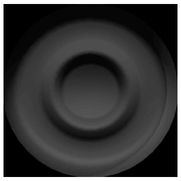
=

Final

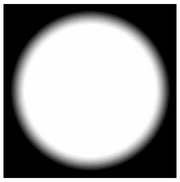
Time

Chrome : texture inputs

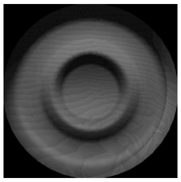
Lightmap



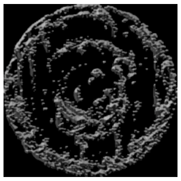
Spotmask



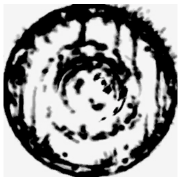
Shadow Map



Chrome bump

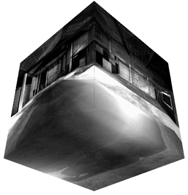


Rust Reveal

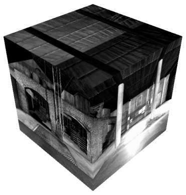


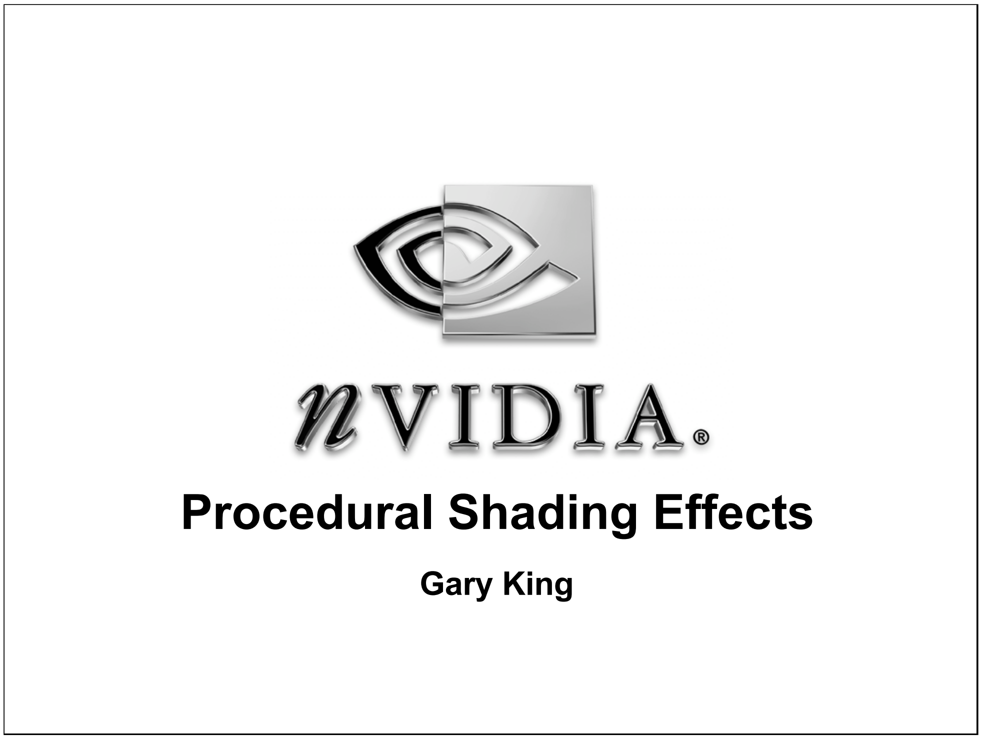
Rust Color

Cube map new



Cube map old

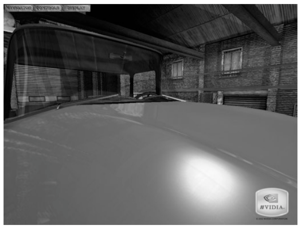




Procedural Shading Effects

Gary King

Time Machine Effects : Paint



Specular color shift

Oxidation

Bubbling

Rusting

Paint textures:

- •Paint Color
- •Rust LUT
- •Shadow map
- •Spotlight mask
- •Light Rust Color*
- •Deep Rust Color*
- •Ambient Light*
- •Bubble Height*
- •Reveal Time*
- •New Environment*
- •Old Environment*

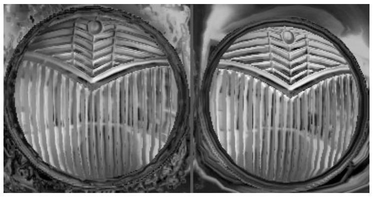
(* = artist created)

60 Pixel Shader instructions, 11 textures

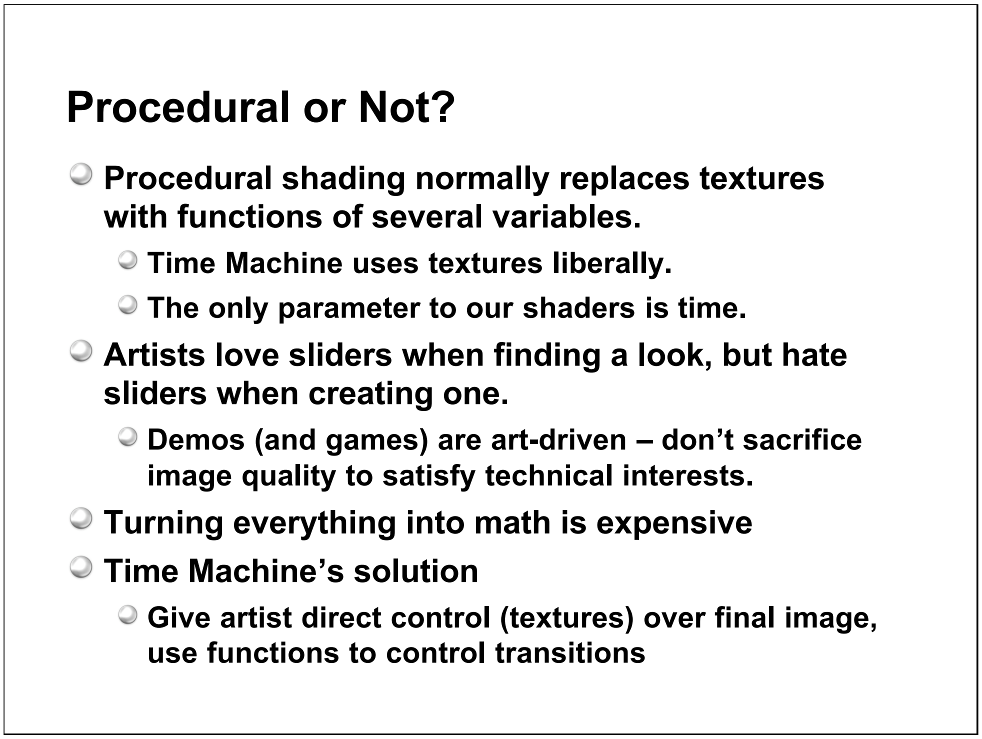
Effects (cont'd) : Wood, Chrome, Glass

Wood fades and cracks
31 instructions, 6 textures

Chrome welts and corrodes
23 instructions, 8 textures

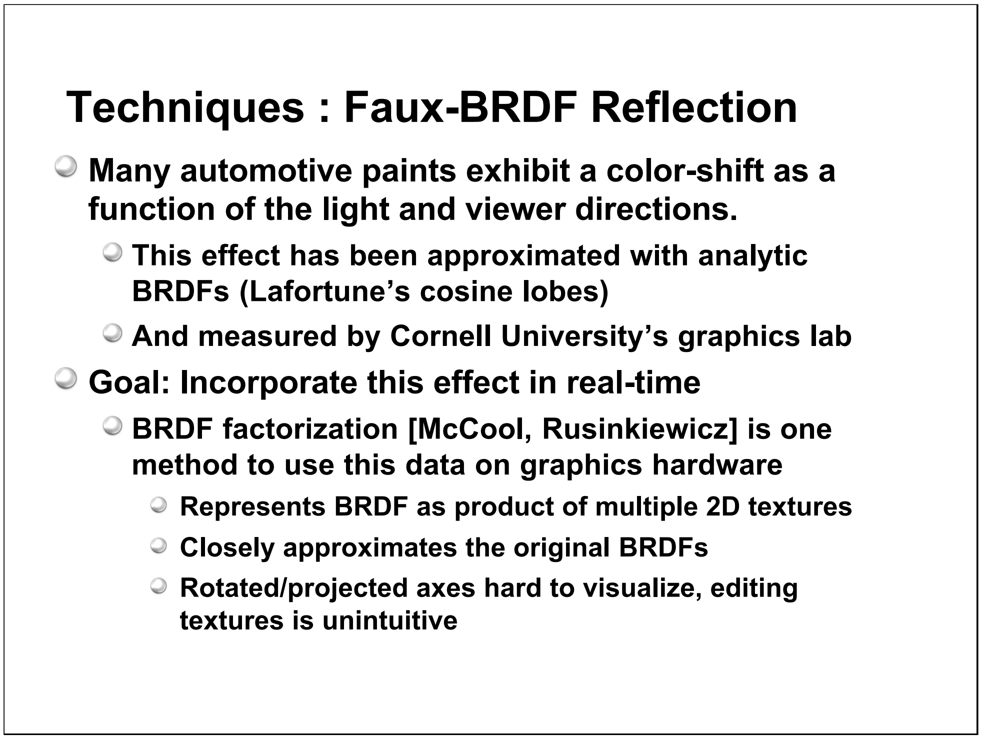


Headlights fog
24 instructions, 4 textures



Procedural or Not?

- Procedural shading normally replaces textures with functions of several variables.
  - Time Machine uses textures liberally.
  - The only parameter to our shaders is time.
- Artists love sliders when finding a look, but hate sliders when creating one.
  - Demos (and games) are art-driven – don’t sacrifice image quality to satisfy technical interests.
- Turning everything into math is expensive
- Time Machine’s solution
  - Give artist direct control (textures) over final image, use functions to control transitions

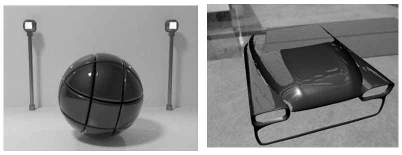


Techniques : Faux-BRDF Reflection

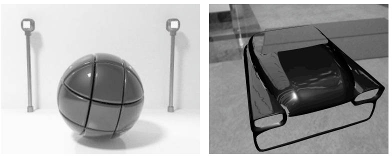
- Many automotive paints exhibit a color-shift as a function of the light and viewer directions.
  - This effect has been approximated with analytic BRDFs (Lafortune's cosine lobes)
  - And measured by Cornell University's graphics lab
- Goal: Incorporate this effect in real-time
  - BRDF factorization [McCool, Rusinkiewicz] is one method to use this data on graphics hardware
    - Represents BRDF as product of multiple 2D textures
    - Closely approximates the original BRDFs
    - Rotated/projected axes hard to visualize, editing textures is unintuitive

Techniques : Faux-BRDF Reflection 2

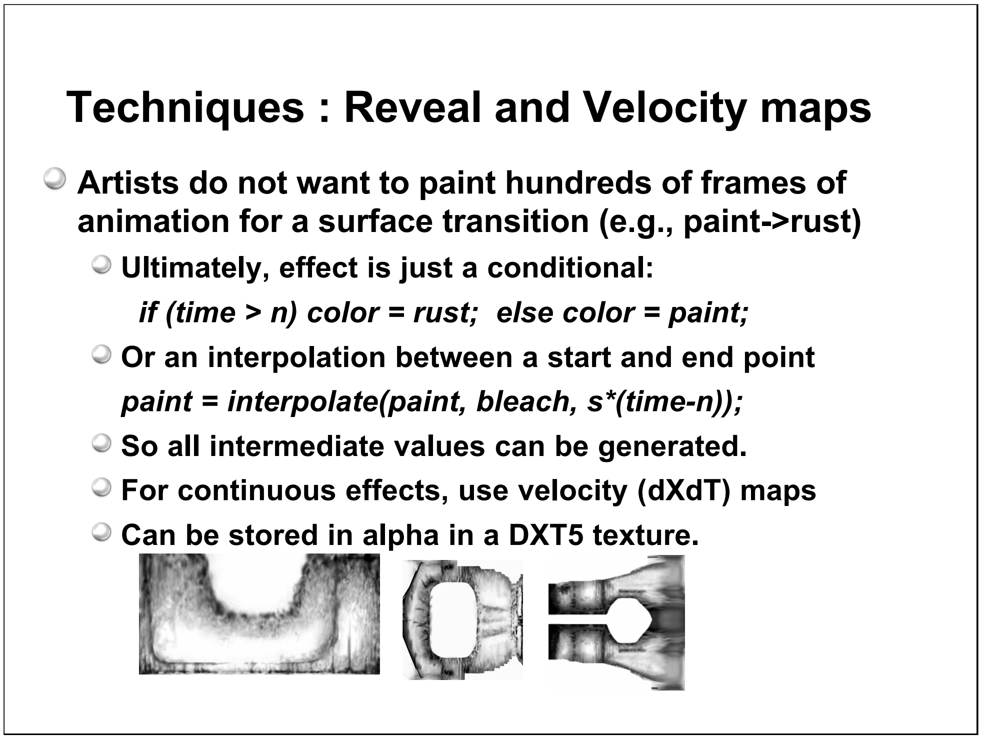
- Our solution: project BRDF values onto a single 2D texture, and factor out the intensity
  - Compute intensity in real-time, using $(N.H)^s$
  - Texture varies slowly, so it can be low-res (64x64).
  - Anti-aliasing texture fixes laser noise at grazing angles
  - For automotive paints, N.L and N.H work well for axes.
  - Not physically accurate, but fast and high-quality.
  - Easy for artists to tweak.



Dupont Cayman lacquer



Mystique lacquer



Techniques : Reveal and Velocity maps

- Artists do not want to paint hundreds of frames of animation for a surface transition (e.g., paint->rust)
  - Ultimately, effect is just a conditional:

    if (time > n) color = rust; else color = paint;
  - Or an interpolation between a start and end point

    paint = interpolate(paint, bleach, s(time-n));*
  - So all intermediate values can be generated.
  - For continuous effects, use velocity (dXdT) maps
  - Can be stored in alpha in a DXT5 texture.

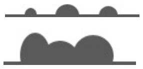
Techniques : Dynamic Bump mapping

- Scaling a normal map by a constant doesn't change surface topology.

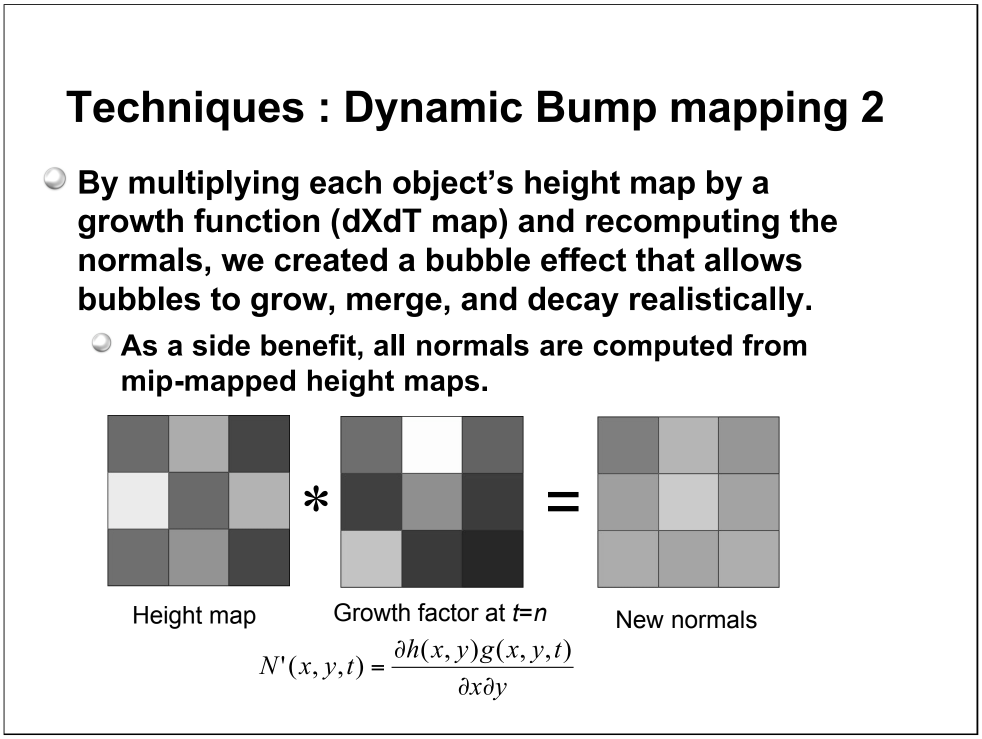
$\iint N(x,y)\partial x \partial y = h(x,y)$ $\iint cN(x,y)\partial x \partial y = ch(x,y)$

- To change surface topology, the height map needs to be updated every frame, and the normals recomputed from that (chain rule).

$$N'(x,y) = \frac{\partial h'(x,y)}{\partial x \partial y}$$



- This is analogous to techniques that use the GPU to solve partial differential equations.

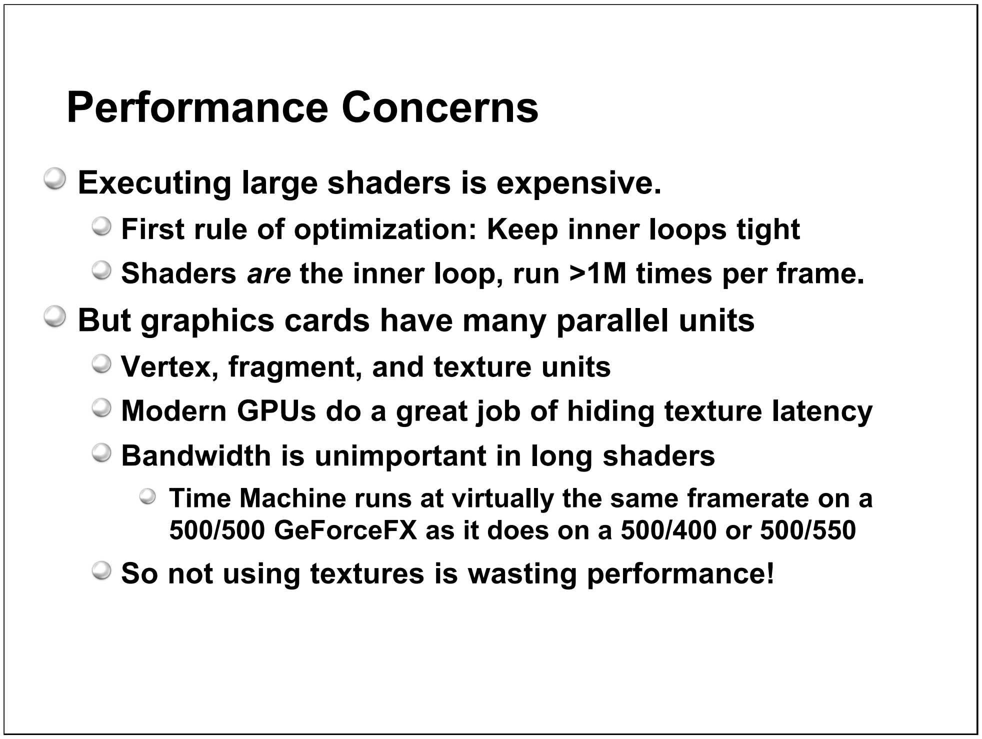


Techniques : Dynamic Bump mapping 2

- **By multiplying each object’s height map by a growth function (dXdT map) and recomputing the normals, we created a bubble effect that allows bubbles to grow, merge, and decay realistically.**
  - **As a side benefit, all normals are computed from mip-mapped height maps.**

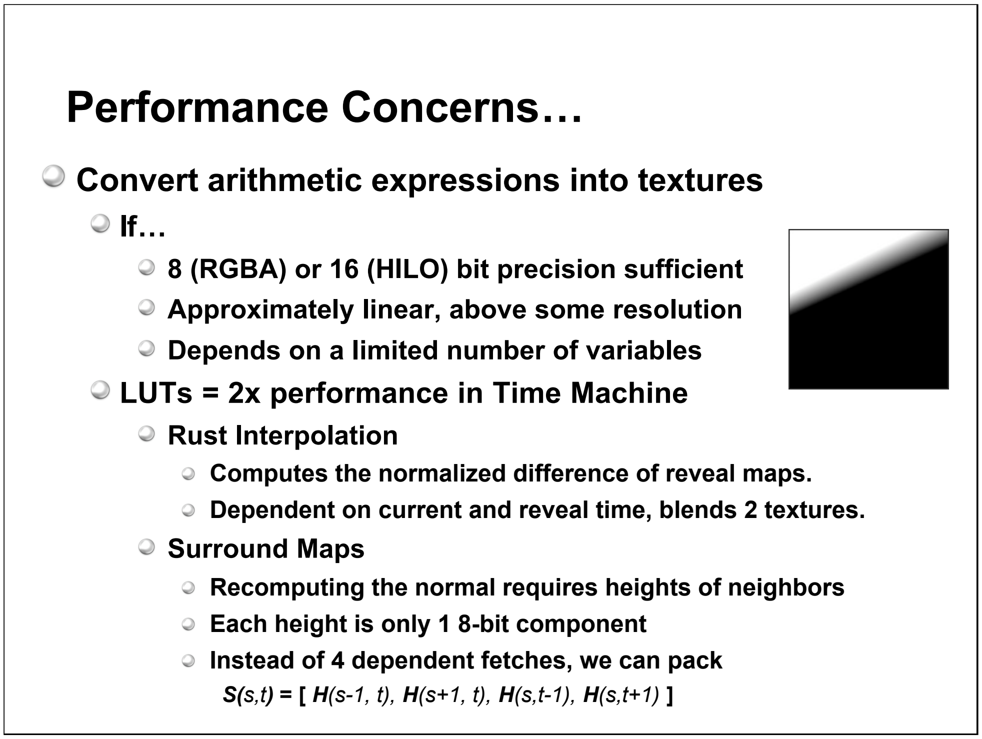
*
=
Height map
Growth factor at *t*=*n*
New normals

$$N'(x,y,t) = \frac{\partial h(x,y)g(x,y,t)}{\partial x \partial y}$$



Performance Concerns

- Executing large shaders is expensive.
  - First rule of optimization: Keep inner loops tight
  - Shaders *are* the inner loop, run >1M times per frame.
- But graphics cards have many parallel units
  - Vertex, fragment, and texture units
  - Modern GPUs do a great job of hiding texture latency
  - Bandwidth is unimportant in long shaders
    - Time Machine runs at virtually the same framerate on a 500/500 GeForceFX as it does on a 500/400 or 500/550
  - So not using textures is wasting performance!



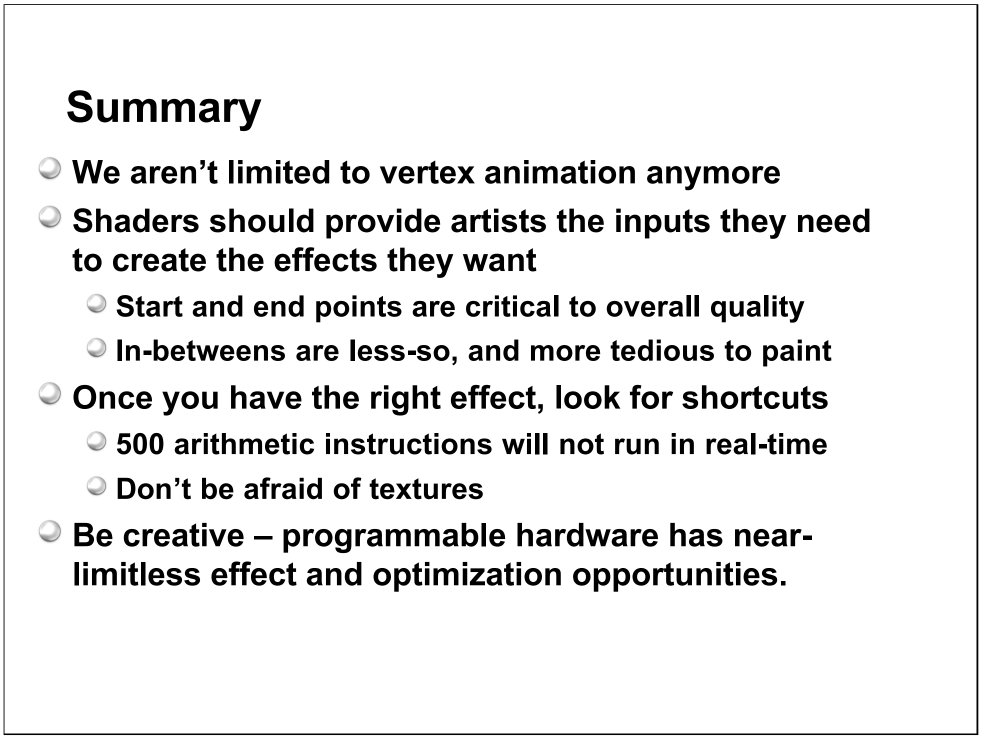
Performance Concerns…

- **Convert arithmetic expressions into textures**
  - **If…**
    - **8 (RGBA) or 16 (HILO) bit precision sufficient**
    - **Approximately linear, above some resolution**
    - **Depends on a limited number of variables**
  - **LUTs = 2x performance in Time Machine**
    - **Rust Interpolation**
      - **Computes the normalized difference of reveal maps.**
      - **Dependent on current and reveal time, blends 2 textures.**
    - **Surround Maps**
      - **Recomputing the normal requires heights of neighbors**
      - **Each height is only 1 8-bit component**
      - **Instead of 4 dependent fetches, we can pack**

$S(s,t) = [\ H(s-1, t),\ H(s+1, t),\ H(s,t-1),\ H(s,t+1)\ ]$

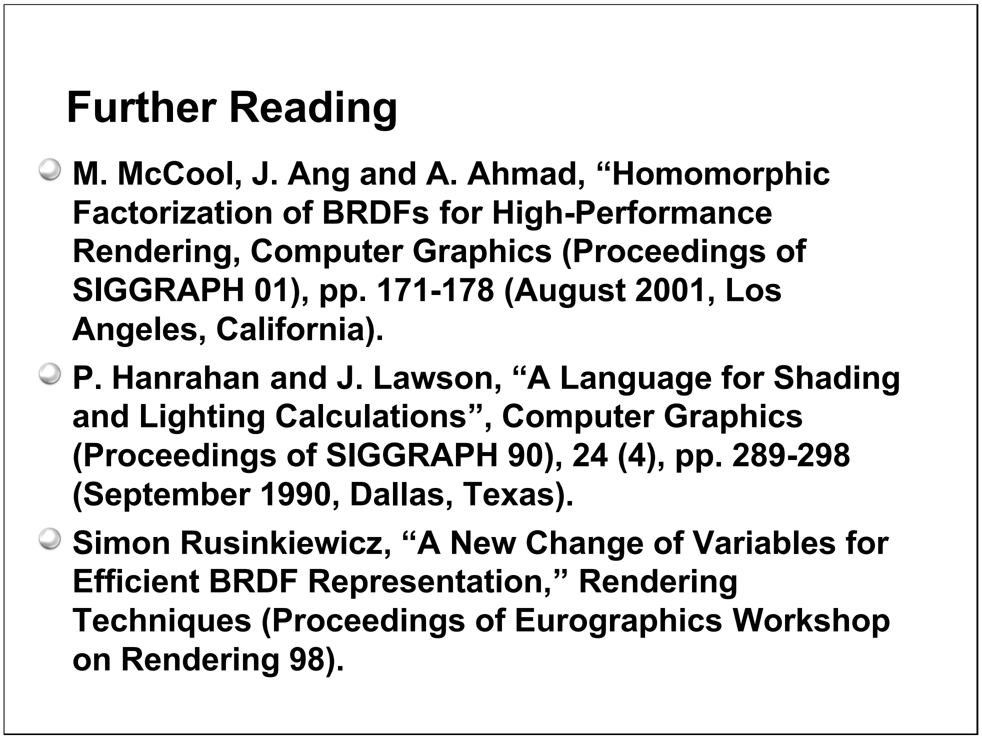
Performance Concerns…

- Defer common operations
  - Lighting for each effect layer is $(K_s*(N.H)^b + K_d*(N.L))*v$
    - Compute normal, select K_s, b, and K_d based on the per-pixel layer, and light once (don’t call pow() more times than absolutely necessary!).
- Invisible results don’t need to be correct.
  - Example: The texture coordinates for the specular color-shift don’t matter once the paint has rusted



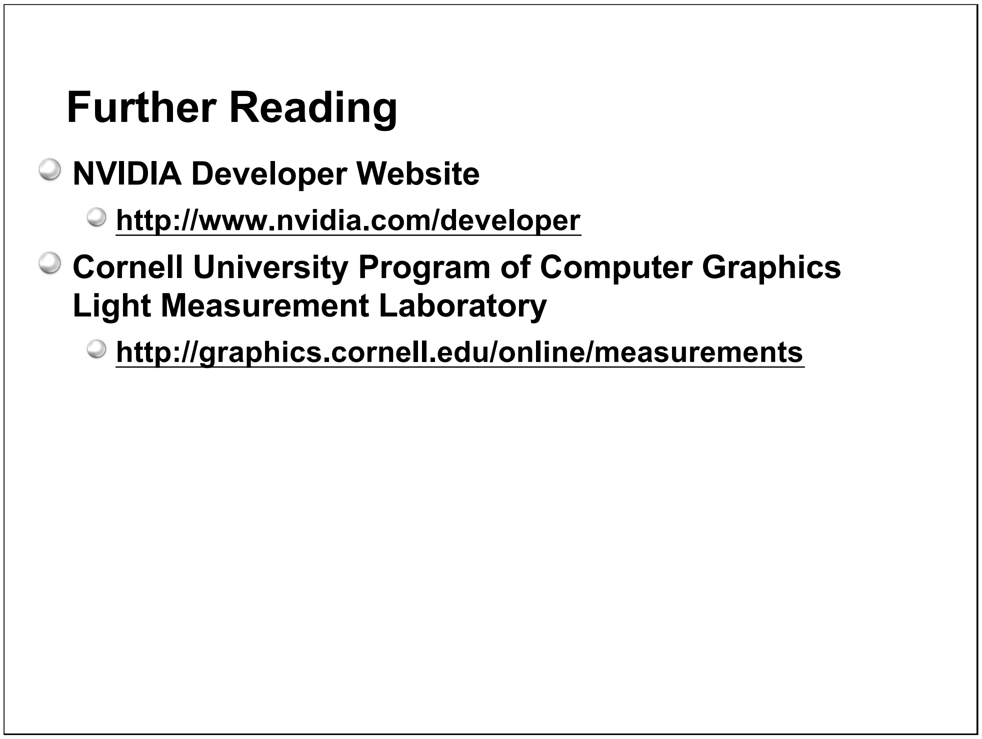
Summary

- We aren’t limited to vertex animation anymore
- Shaders should provide artists the inputs they need to create the effects they want
  - Start and end points are critical to overall quality
  - In-betweens are less-so, and more tedious to paint
- Once you have the right effect, look for shortcuts
  - 500 arithmetic instructions will not run in real-time
  - Don’t be afraid of textures
- Be creative – programmable hardware has near-limitless effect and optimization opportunities.



Further Reading

- M. McCool, J. Ang and A. Ahmad, “Homomorphic Factorization of BRDFs for High-Performance Rendering, Computer Graphics (Proceedings of SIGGRAPH 01), pp. 171-178 (August 2001, Los Angeles, California).
- P. Hanrahan and J. Lawson, “A Language for Shading and Lighting Calculations”, Computer Graphics (Proceedings of SIGGRAPH 90), 24 (4), pp. 289-298 (September 1990, Dallas, Texas).
- Simon Rusinkiewicz, “A New Change of Variables for Efficient BRDF Representation,” Rendering Techniques (Proceedings of Eurographics Workshop on Rendering 98).

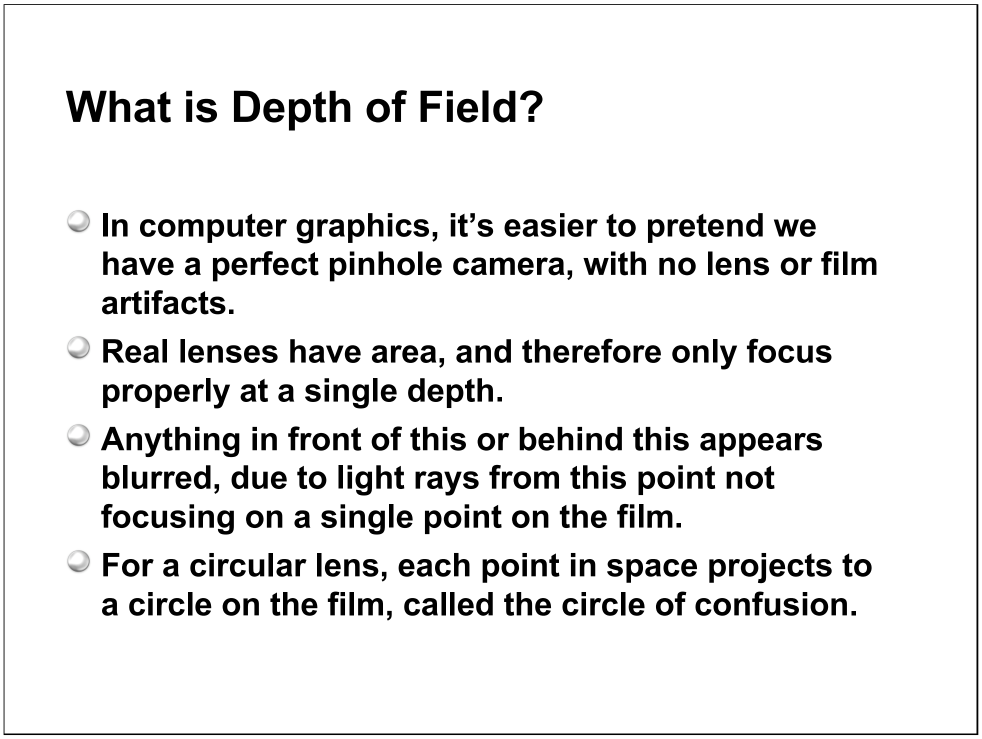


Further Reading

- NVIDIA Developer Website
  - http://www.nvidia.com/developer
- Cornell University Program of Computer Graphics Light Measurement Laboratory
  - http://graphics.cornell.edu/online/measurements

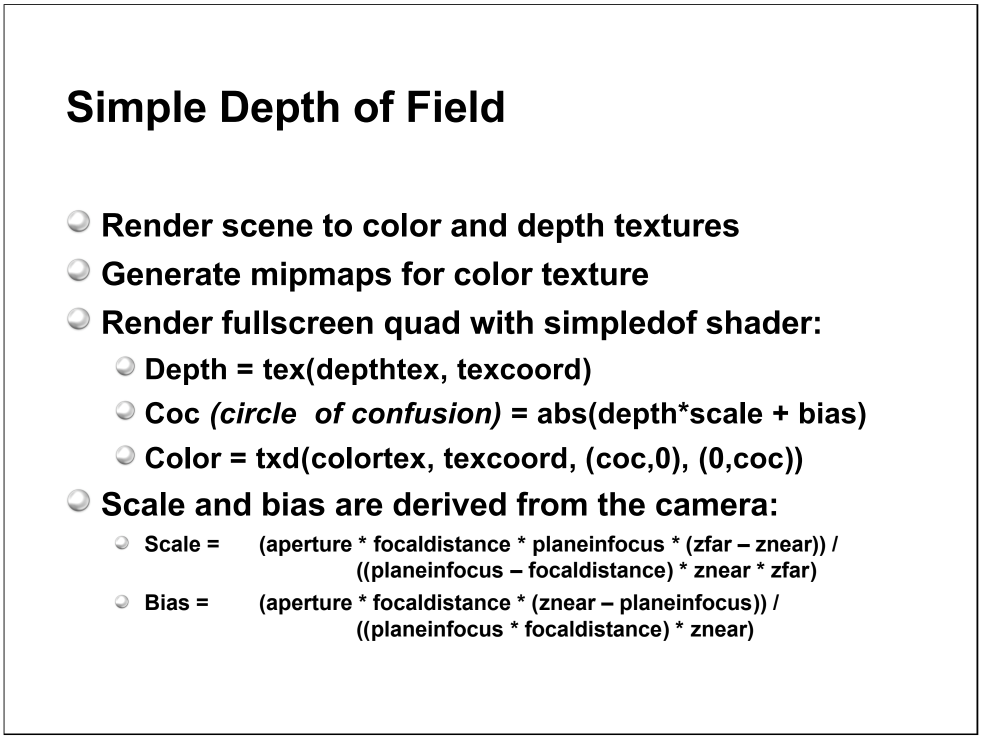
Depth of Field in the Toys Demo

Fun with Realtime Post-Processing



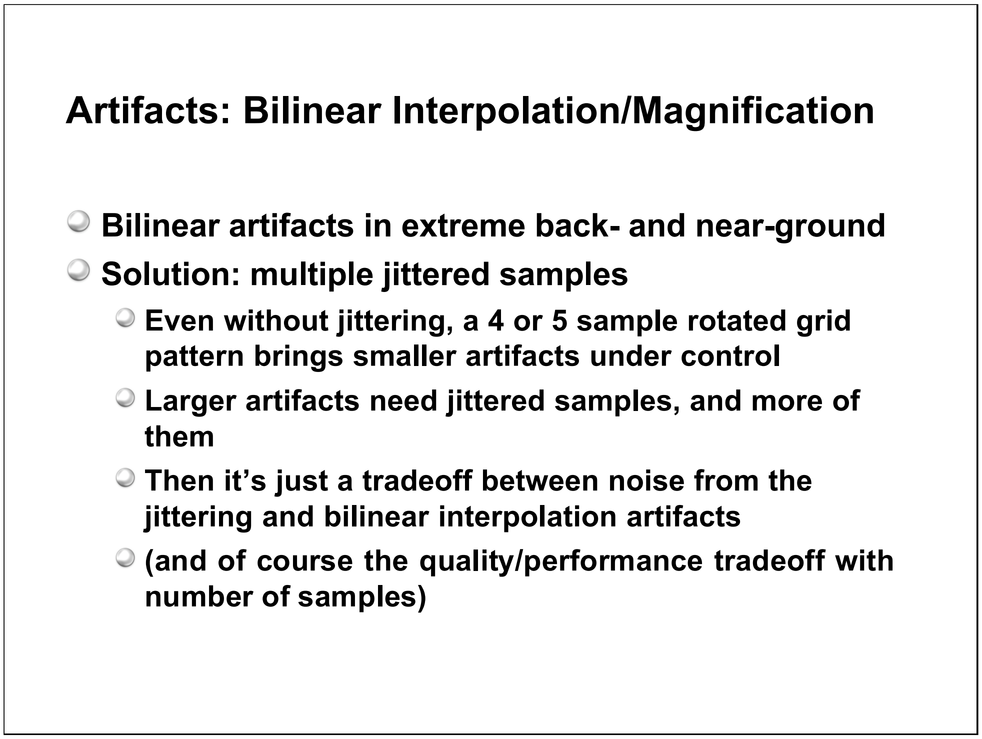
What is Depth of Field?

- In computer graphics, it’s easier to pretend we have a perfect pinhole camera, with no lens or film artifacts.
- Real lenses have area, and therefore only focus properly at a single depth.
- Anything in front of this or behind this appears blurred, due to light rays from this point not focusing on a single point on the film.
- For a circular lens, each point in space projects to a circle on the film, called the circle of confusion.



Simple Depth of Field

- Render scene to color and depth textures
- Generate mipmaps for color texture
- Render fullscreen quad with simpledof shader:
  - Depth = tex(depthtex, texcoord)
  - Coc *(circle of confusion)* = abs(depth*scale + bias)
  - Color = txd(colortex, texcoord, (coc,0), (0,coc))
- Scale and bias are derived from the camera:
  - Scale = (aperture * focaldistance * planeinfocus * (zfar – znear)) / ((planeinfocus – focaldistance) * znear * zfar)
  - Bias = (aperture * focaldistance * (znear – planeinfocus)) / ((planeinfocus * focaldistance) * znear)

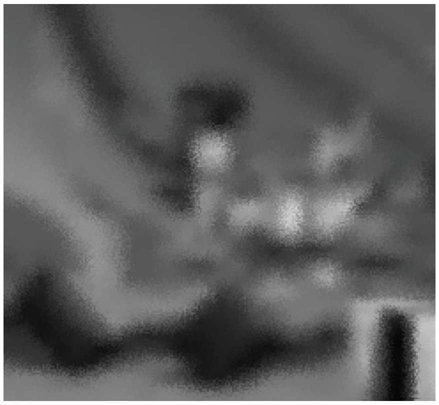


Artifacts: Bilinear Interpolation/Magnification

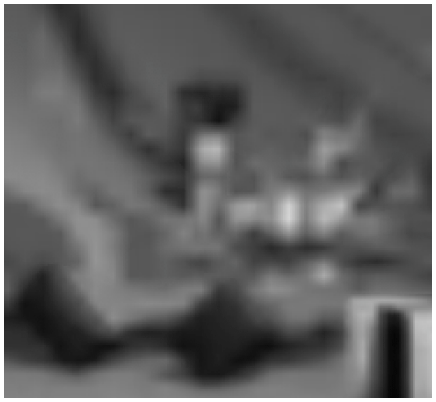
- Bilinear artifacts in extreme back- and near-ground
- Solution: multiple jittered samples
  - Even without jittering, a 4 or 5 sample rotated grid pattern brings smaller artifacts under control
  - Larger artifacts need jittered samples, and more of them
  - Then it’s just a tradeoff between noise from the jittering and bilinear interpolation artifacts
  - (and of course the quality/performance tradeoff with number of samples)

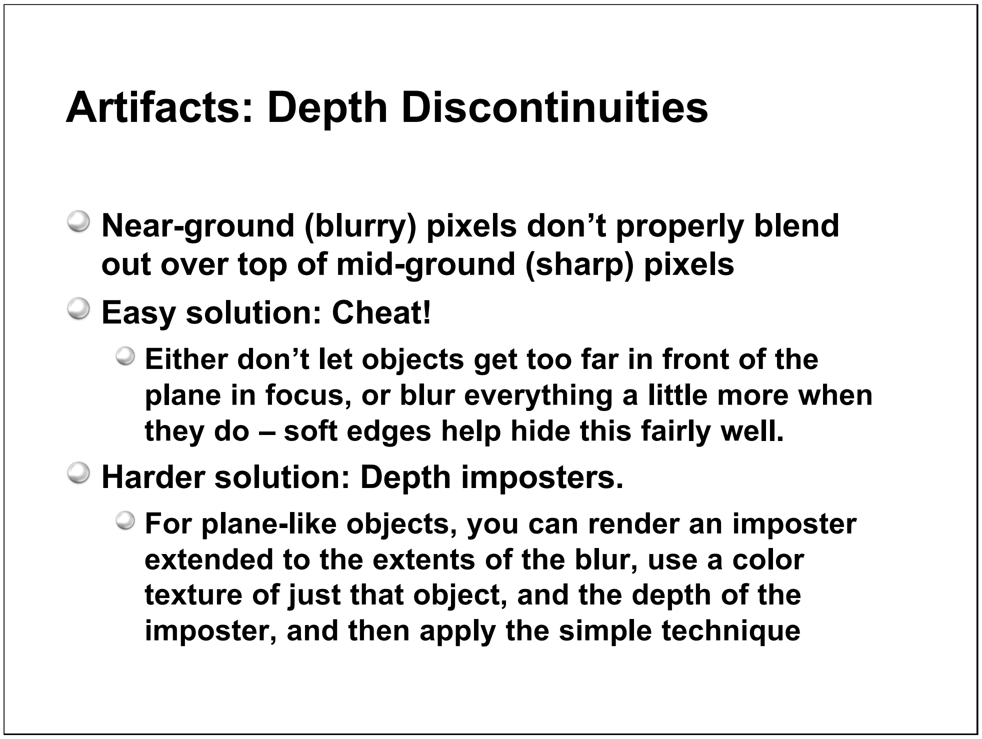
Noise vs. Interpolation Artifacts

With Noise



Without Noise

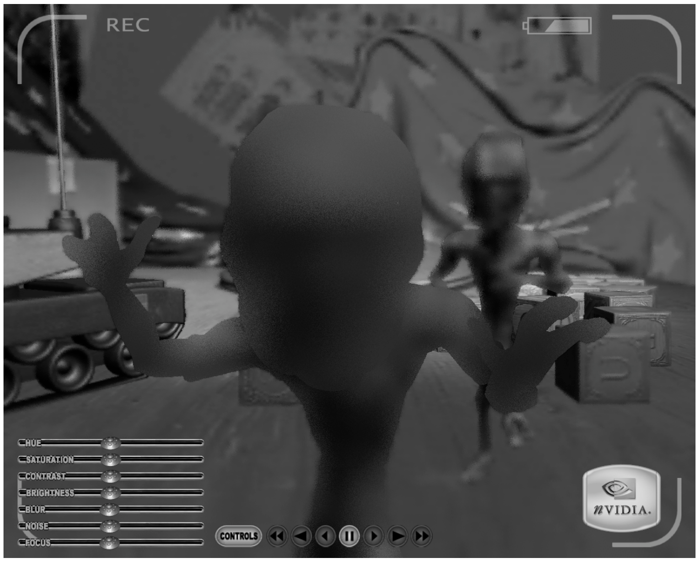


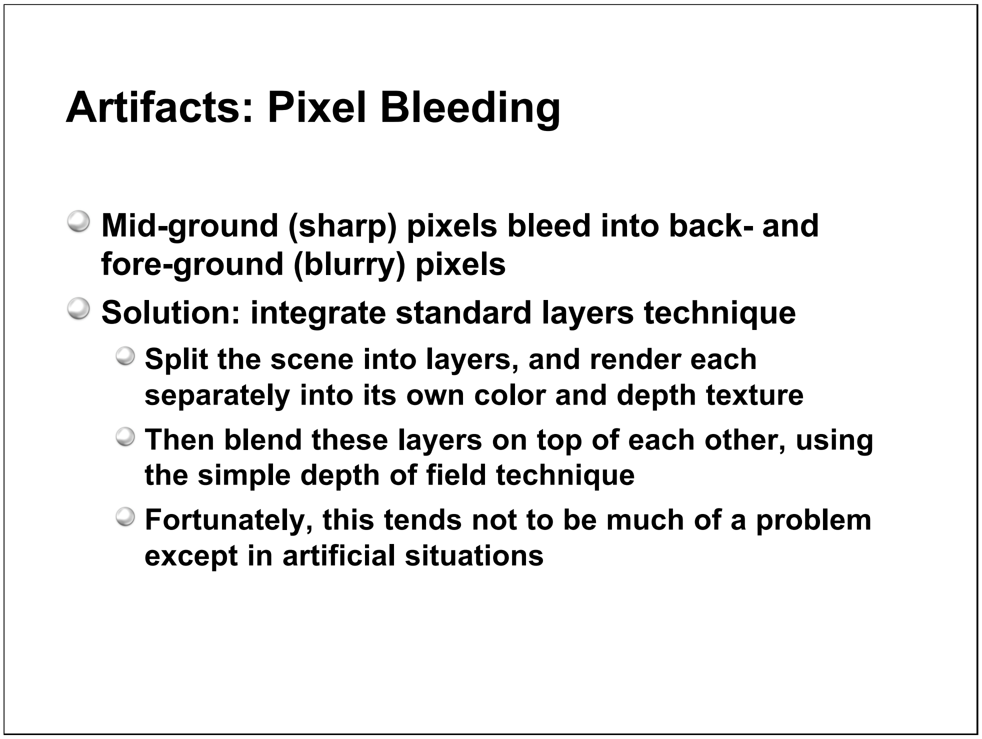


Artifacts: Depth Discontinuities

- Near-ground (blurry) pixels don’t properly blend out over top of mid-ground (sharp) pixels
- Easy solution: Cheat!
  - Either don’t let objects get too far in front of the plane in focus, or blur everything a little more when they do – soft edges help hide this fairly well.
- Harder solution: Depth imposters.
  - For plane-like objects, you can render an imposter extended to the extents of the blur, use a color texture of just that object, and the depth of the imposter, and then apply the simple technique

Depth Discontinuities



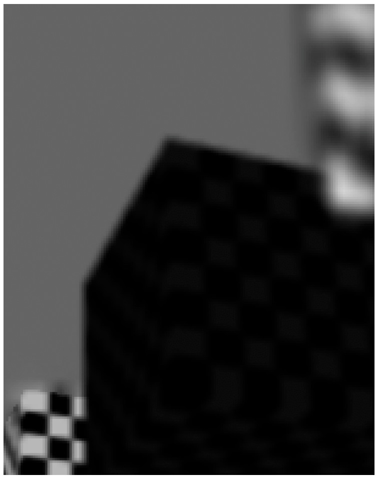


Artifacts: Pixel Bleeding

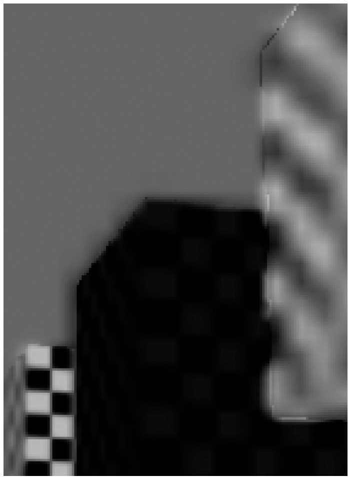
- Mid-ground (sharp) pixels bleed into back- and fore-ground (blurry) pixels
- Solution: integrate standard layers technique
  - Split the scene into layers, and render each separately into its own color and depth texture
  - Then blend these layers on top of each other, using the simple depth of field technique
  - Fortunately, this tends not to be much of a problem except in artificial situations

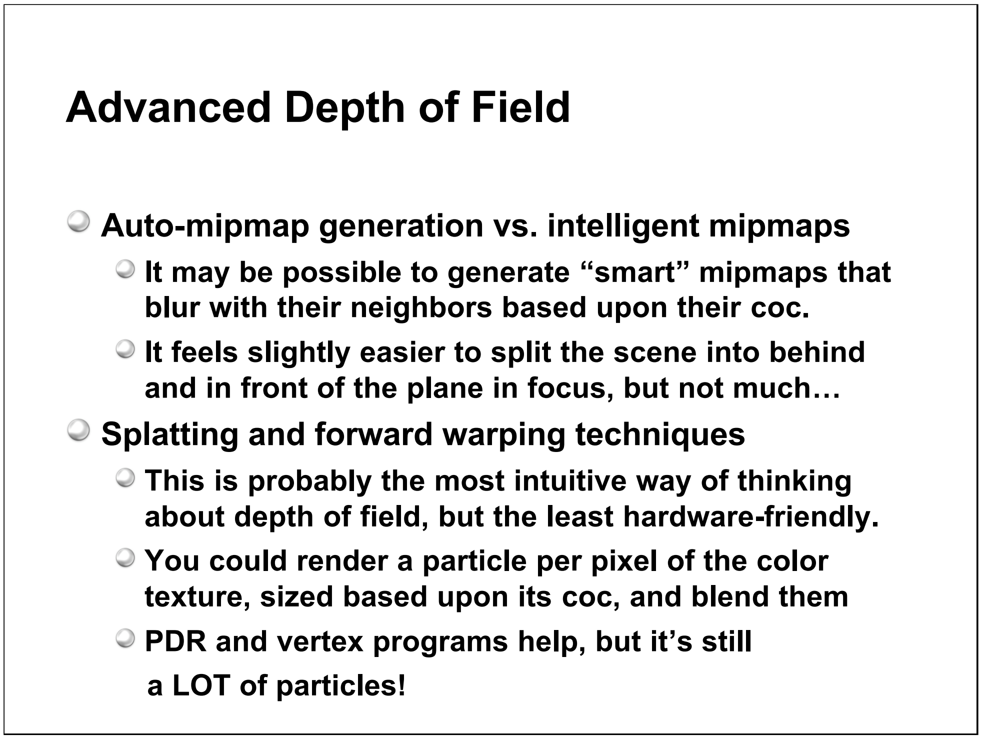
Simple DOF Vs. Layered DOF

Layered DOF



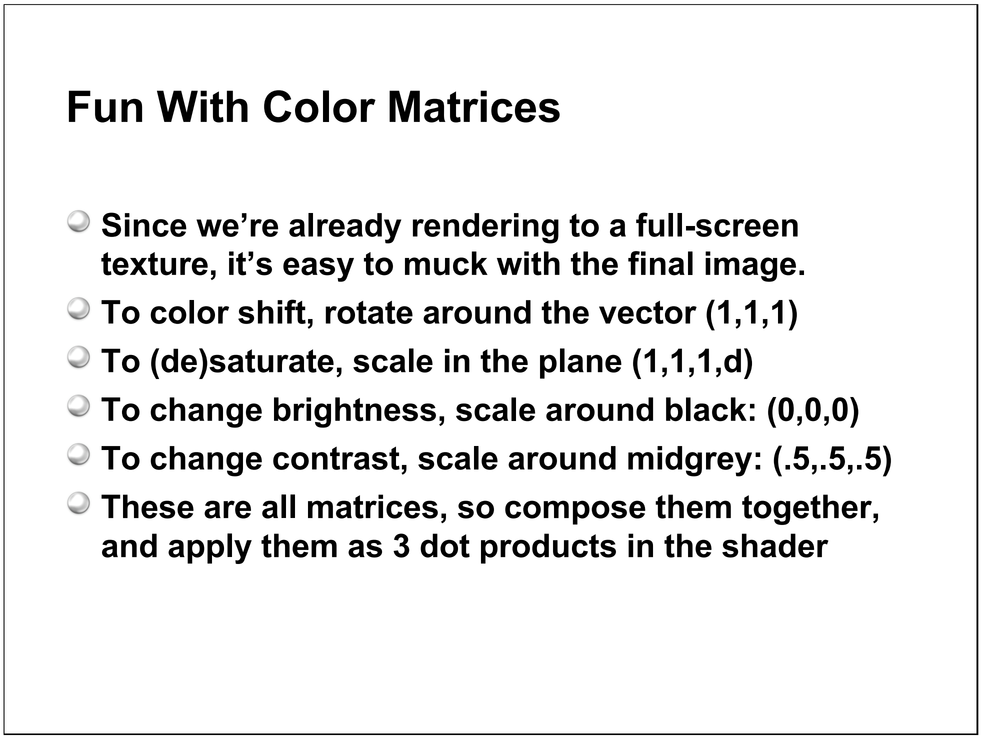
Simple DOF





Advanced Depth of Field

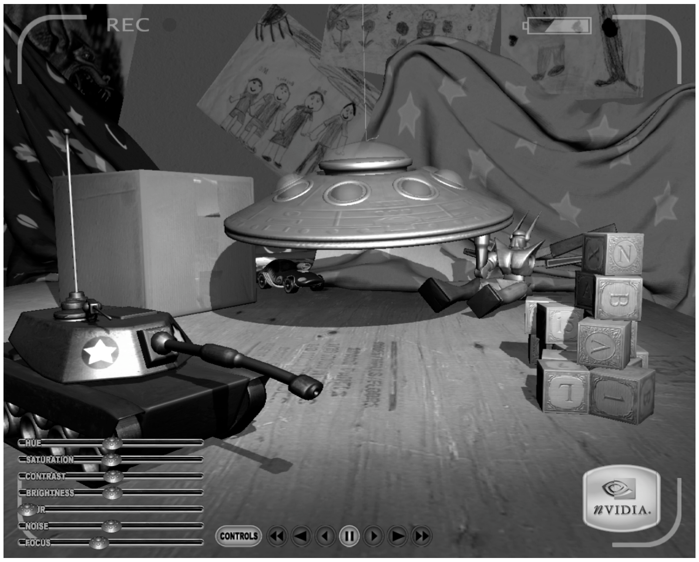
- Auto-mipmap generation vs. intelligent mipmaps
  - It may be possible to generate “smart” mipmaps that blur with their neighbors based upon their coc.
  - It feels slightly easier to split the scene into behind and in front of the plane in focus, but not much…
- Splatting and forward warping techniques
  - This is probably the most intuitive way of thinking about depth of field, but the least hardware-friendly.
  - You could render a particle per pixel of the color texture, sized based upon its coc, and blend them
  - PDR and vertex programs help, but it’s still a LOT of particles!



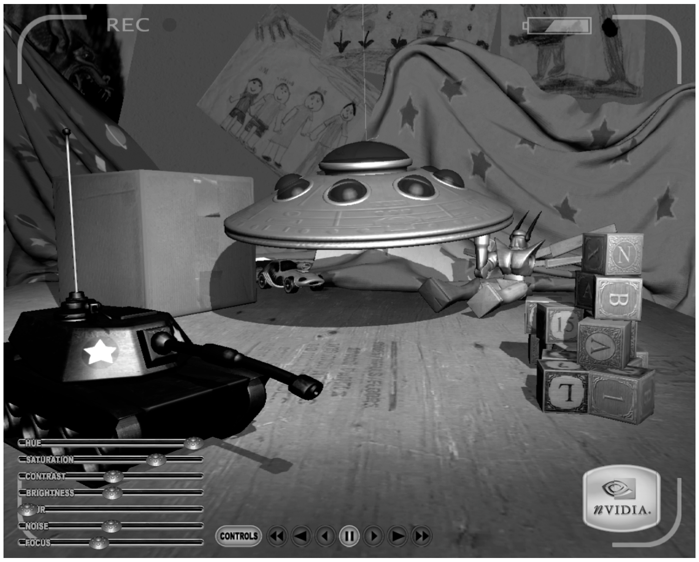
Fun With Color Matrices

- Since we’re already rendering to a full-screen texture, it’s easy to muck with the final image.
- To color shift, rotate around the vector (1,1,1)
- To (de)saturate, scale in the plane (1,1,1,d)
- To change brightness, scale around black: (0,0,0)
- To change contrast, scale around midgrey: (.5,.5,.5)
- These are all matrices, so compose them together, and apply them as 3 dot products in the shader

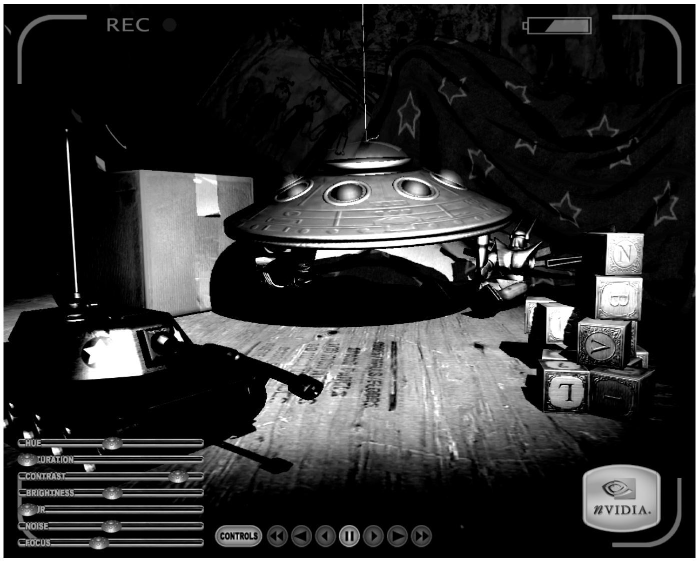
Original Image

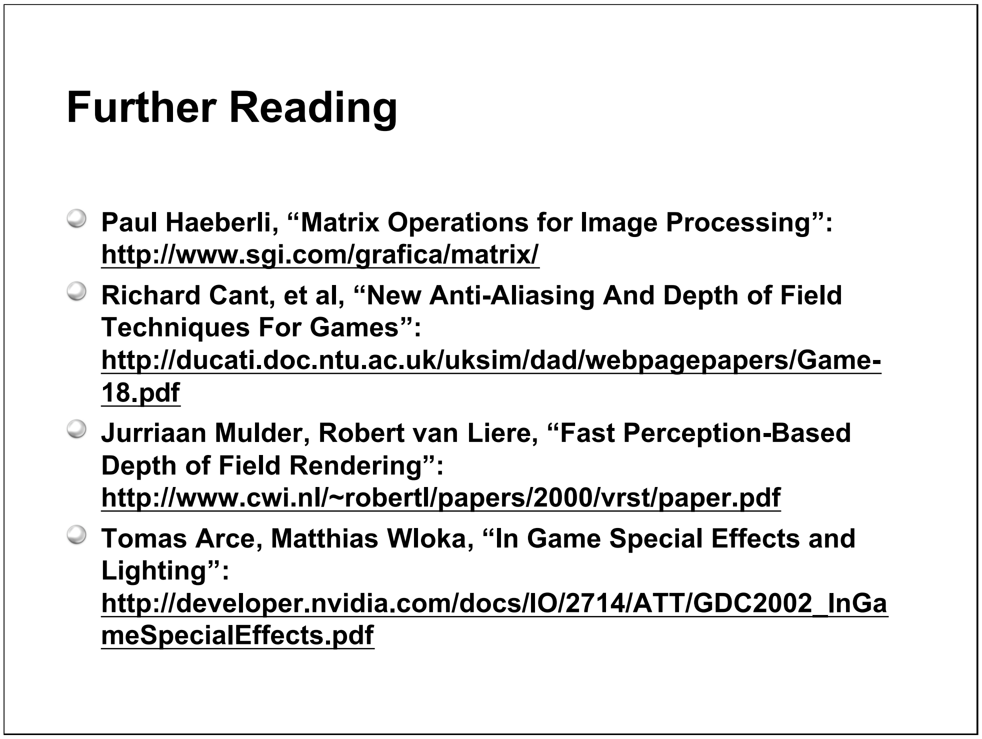


Colorshifted Image



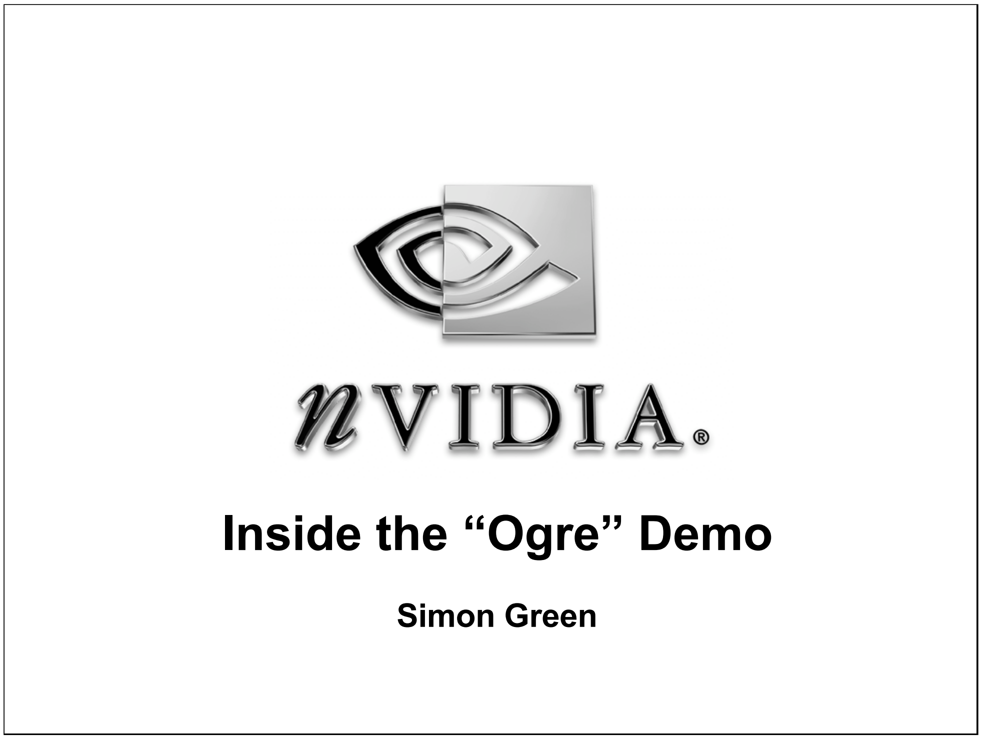
Black and White Image





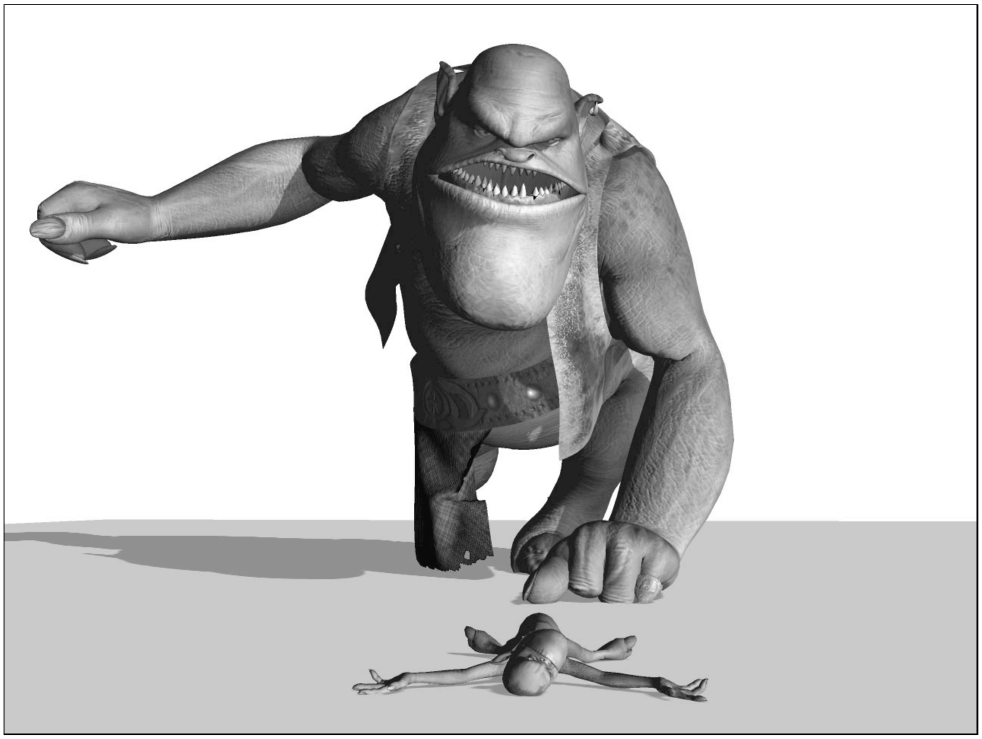
Further Reading

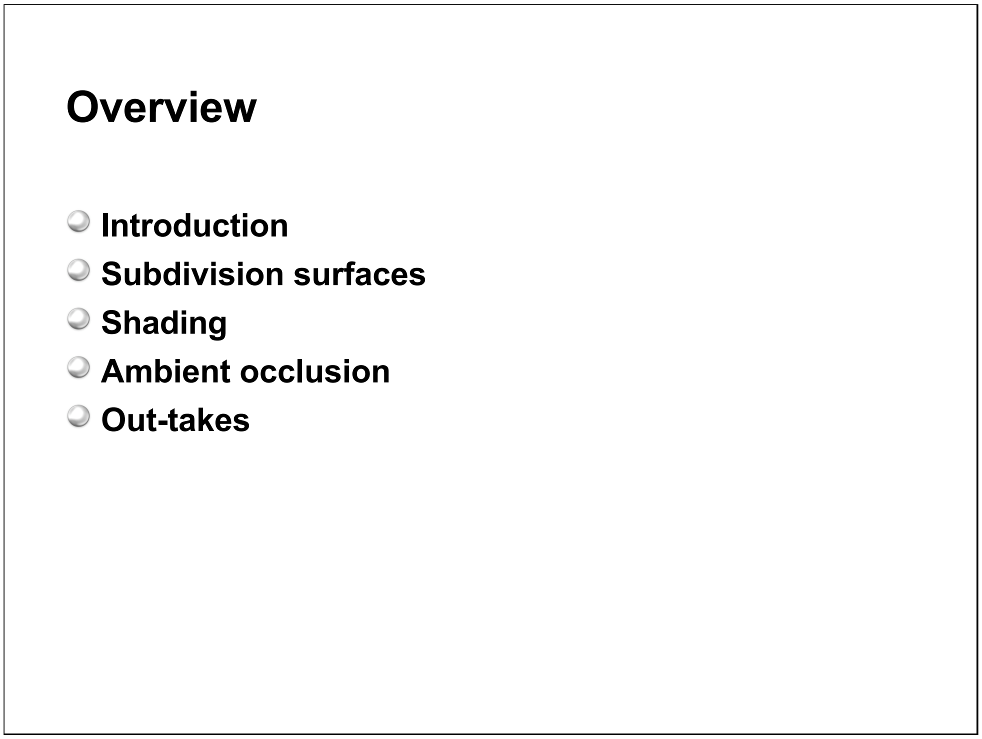
- Paul Haeberli, “Matrix Operations for Image Processing”: http://www.sgi.com/grafica/matrix/
- Richard Cant, et al, “New Anti-Aliasing And Depth of Field Techniques For Games”: http://ducati.doc.ntu.ac.uk/uksim/dad/webpagepapers/Game-18.pdf
- Jurriaan Mulder, Robert van Liere, “Fast Perception-Based Depth of Field Rendering”: http://www.cwi.nl/~robertl/papers/2000/vrst/paper.pdf
- Tomas Arce, Matthias Wloka, “In Game Special Effects and Lighting”: http://developer.nvidia.com/docs/IO/2714/ATT/GDC2002_InGameSpecialEffects.pdf



Inside the “Ogre” Demo

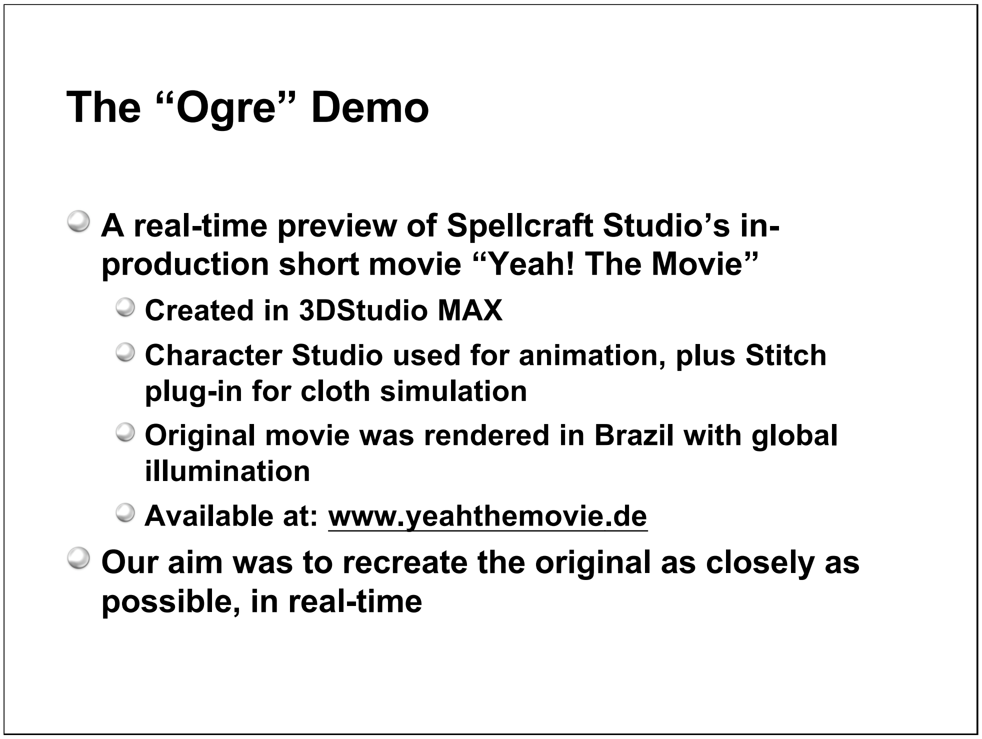
Simon Green





Overview

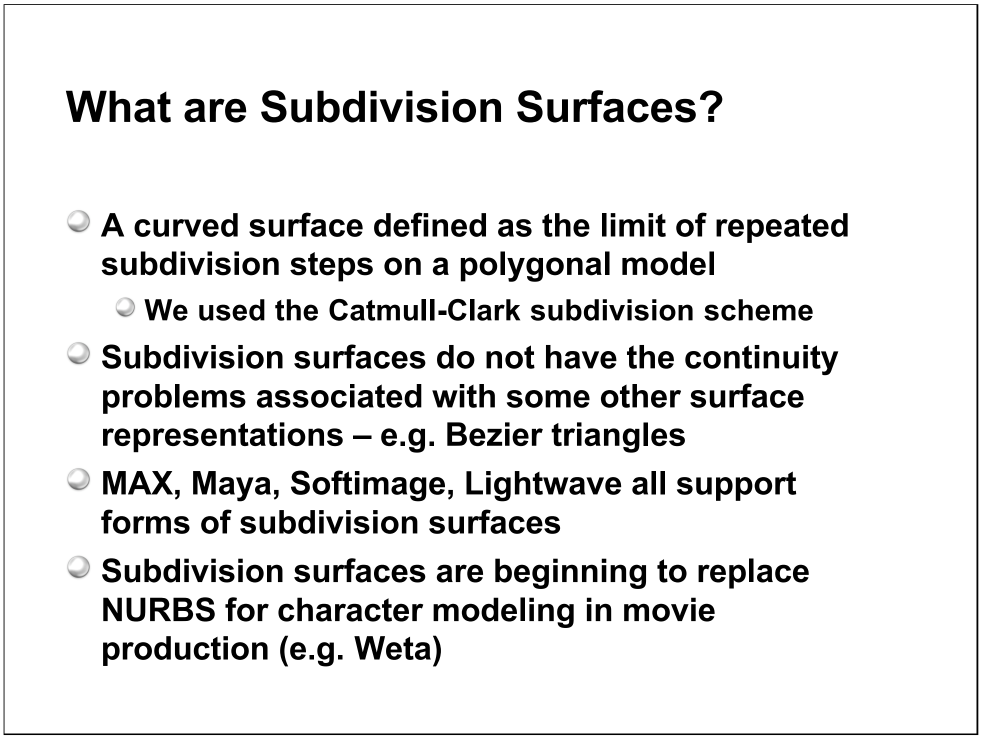
- Introduction
- Subdivision surfaces
- Shading
- Ambient occlusion
- Out-takes



The “Ogre” Demo

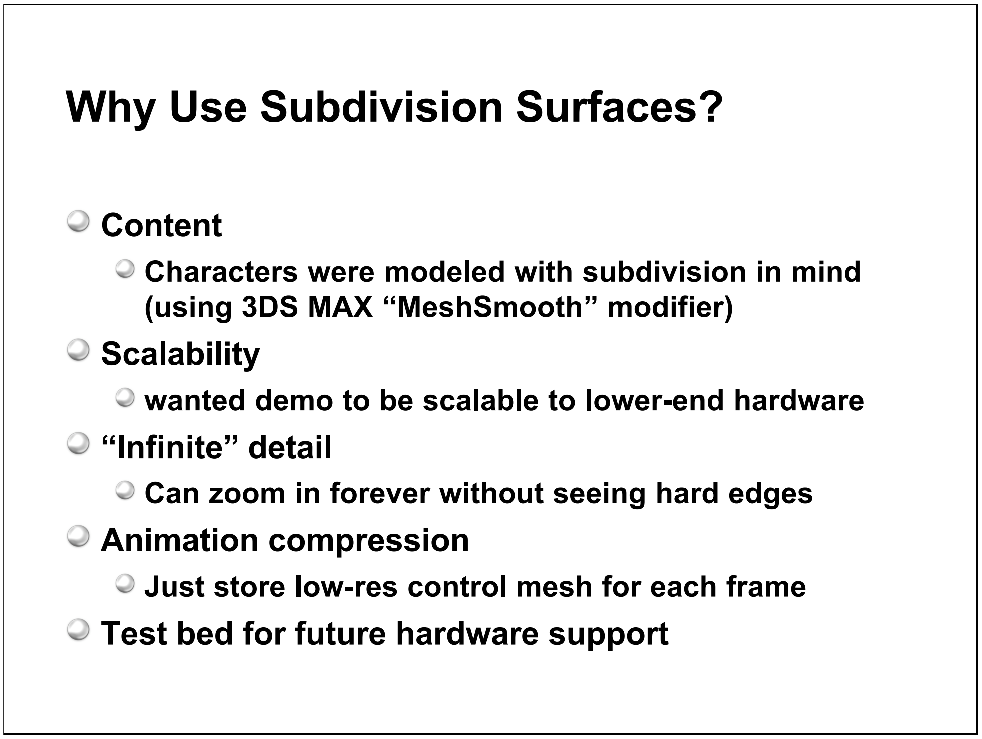
- A real-time preview of Spellcraft Studio’s in-production short movie “Yeah! The Movie”
  - Created in 3DStudio MAX
  - Character Studio used for animation, plus Stitch plug-in for cloth simulation
  - Original movie was rendered in Brazil with global illumination
  - Available at: www.yeahthemovie.de
- Our aim was to recreate the original as closely as possible, in real-time

The Original Short Movie



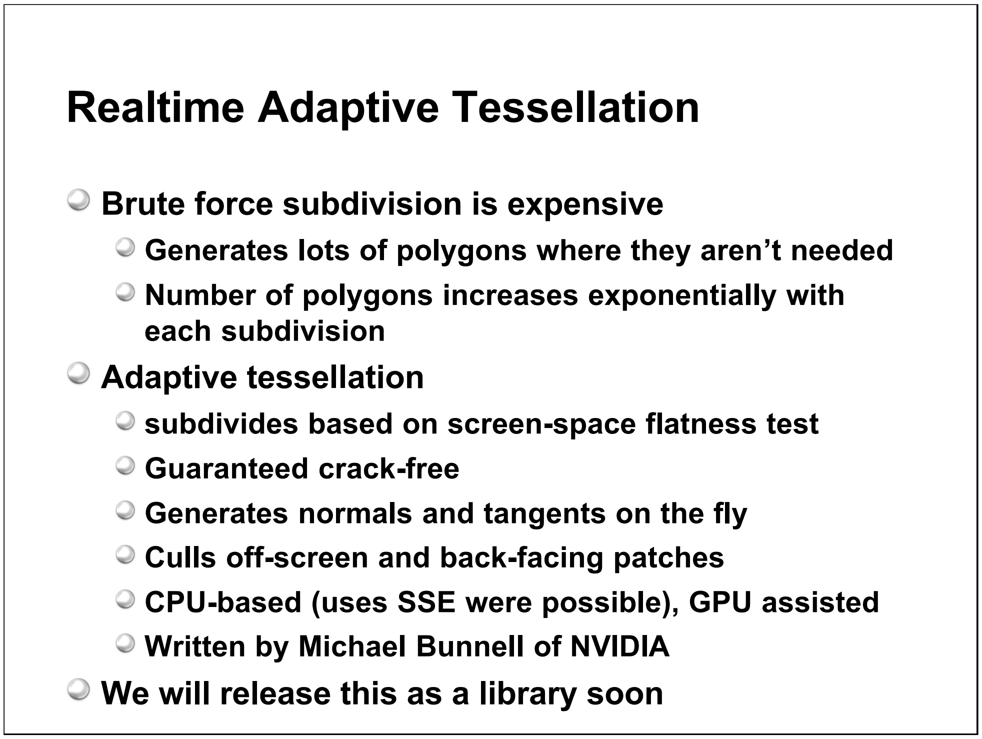
What are Subdivision Surfaces?

- A curved surface defined as the limit of repeated subdivision steps on a polygonal model
  - We used the Catmull-Clark subdivision scheme
- Subdivision surfaces do not have the continuity problems associated with some other surface representations – e.g. Bezier triangles
- MAX, Maya, Softimage, Lightwave all support forms of subdivision surfaces
- Subdivision surfaces are beginning to replace NURBS for character modeling in movie production (e.g. Weta)



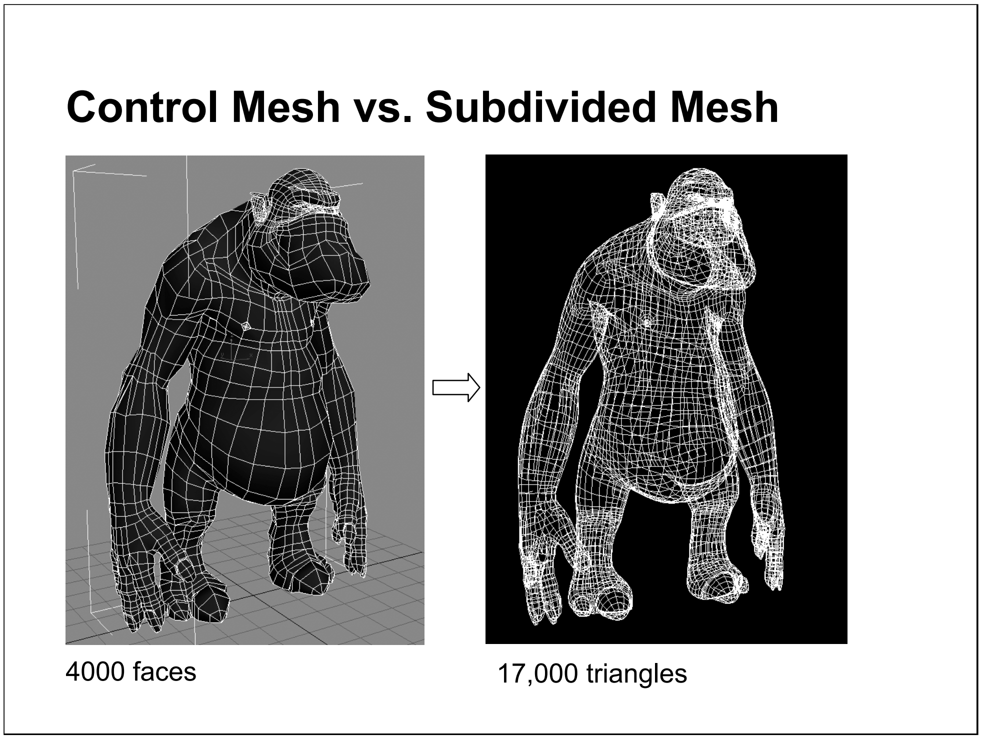
Why Use Subdivision Surfaces?

- Content
  - Characters were modeled with subdivision in mind (using 3DS MAX “MeshSmooth” modifier)
- Scalability
  - wanted demo to be scalable to lower-end hardware
- “Infinite” detail
  - Can zoom in forever without seeing hard edges
- Animation compression
  - Just store low-res control mesh for each frame
- Test bed for future hardware support



Realtime Adaptive Tessellation

- Brute force subdivision is expensive
  - Generates lots of polygons where they aren’t needed
  - Number of polygons increases exponentially with each subdivision
- Adaptive tessellation
  - subdivides based on screen-space flatness test
  - Guaranteed crack-free
  - Generates normals and tangents on the fly
  - Culls off-screen and back-facing patches
  - CPU-based (uses SSE were possible), GPU assisted
  - Written by Michael Bunnell of NVIDIA
- We will release this as a library soon



Control Mesh vs. Subdivided Mesh

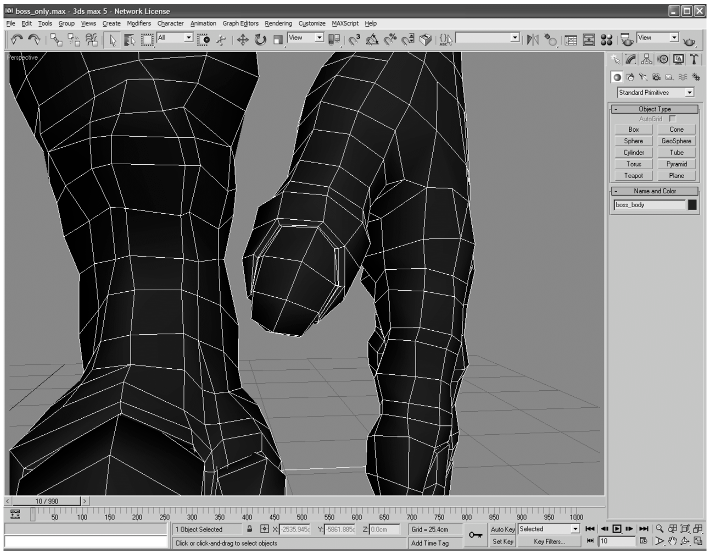
4000 faces

17,000 triangles

Control mesh is mainly four-sided faces, with some five and three sided.

Output is quads

Control Mesh Detail (3DS MAX)



boss_only.max - 3ds max 5 - Network License

Standard Primitives

Name and Color: boss_body

10 / 990

1 Object Selected

X: -2535.945c Y: -5861.885c Z: 0.0cm

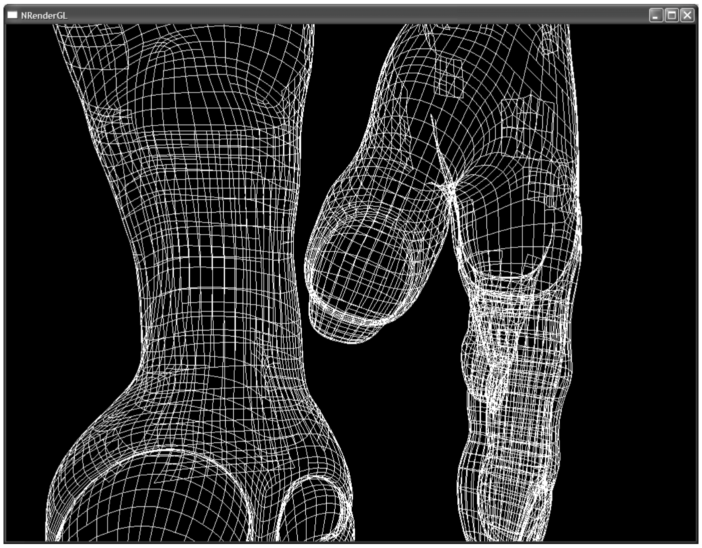
Grid = 25.4cm

Selected

10

Click or click-and-drag to select objects

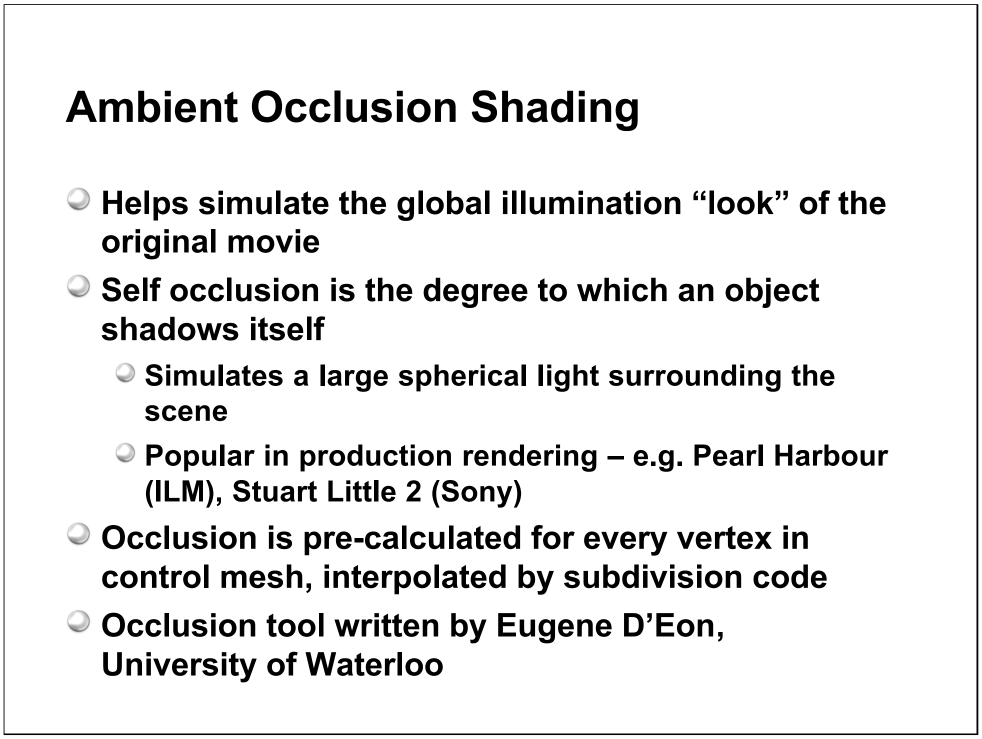
Subdivided Mesh Detail (Realtime)



Shading

- Skin shader
  - Uses 4 textures:
    - Color map, bump map, specular map, shadow map
  - Uses Blinn-style bump mapping (not tangent space)
    - `float3 bump = f3tex2D(bumpTex, v2f.texcoord)`

    -
      ```
      float3 bumpedNormal = normalize(normal +
        bumpScale * (bump.x*v2f.tangent + bump.y*v2f.binormal)));
      ```
  - Ambient term comes from pre-calculated occlusion
- Shadows
  - Uses hardware shadow map support
  - 2k x 2k resolution
  - Uses 8 jittered samples on floor to soften edges

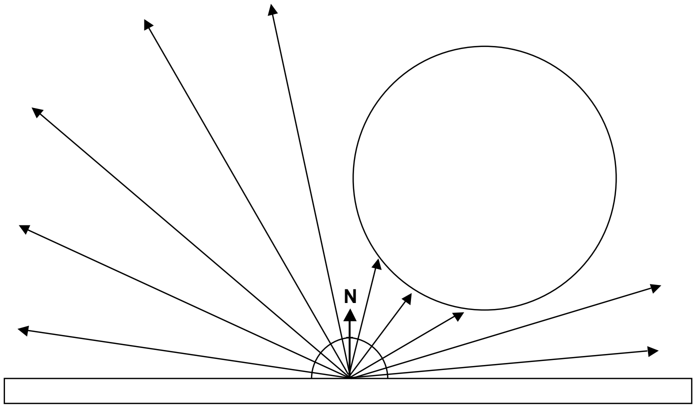


Ambient Occlusion Shading

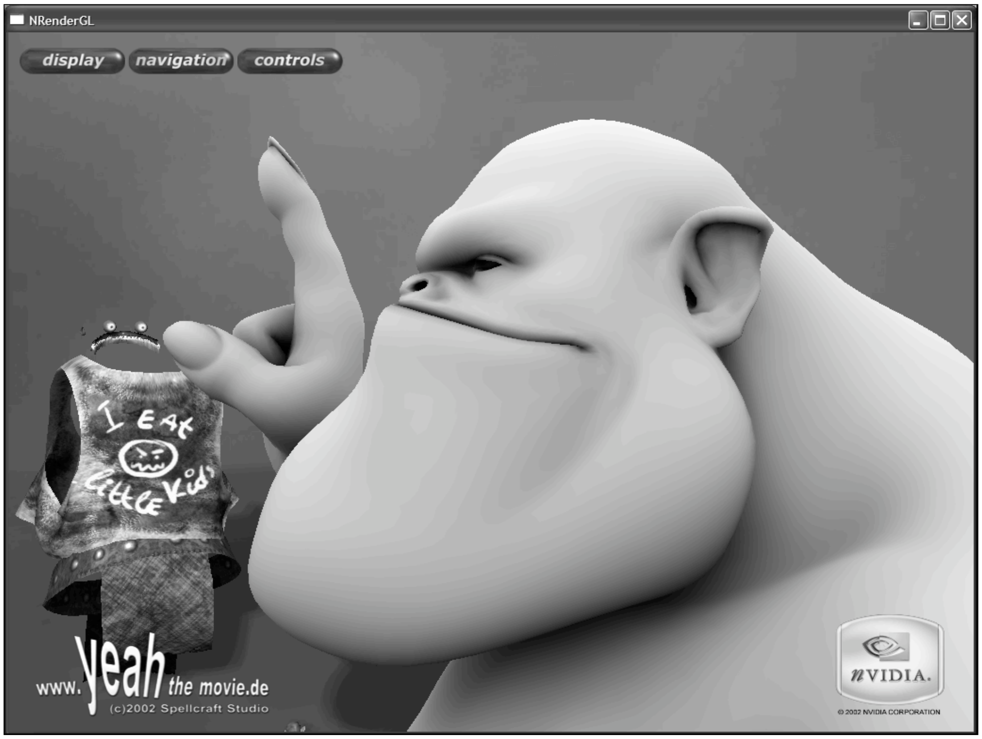
- Helps simulate the global illumination “look” of the original movie
- Self occlusion is the degree to which an object shadows itself
  - Simulates a large spherical light surrounding the scene
  - Popular in production rendering – e.g. Pearl Harbour (ILM), Stuart Little 2 (Sony)
- Occlusion is pre-calculated for every vertex in control mesh, interpolated by subdivision code
- Occlusion tool written by Eugene D’Eon, University of Waterloo

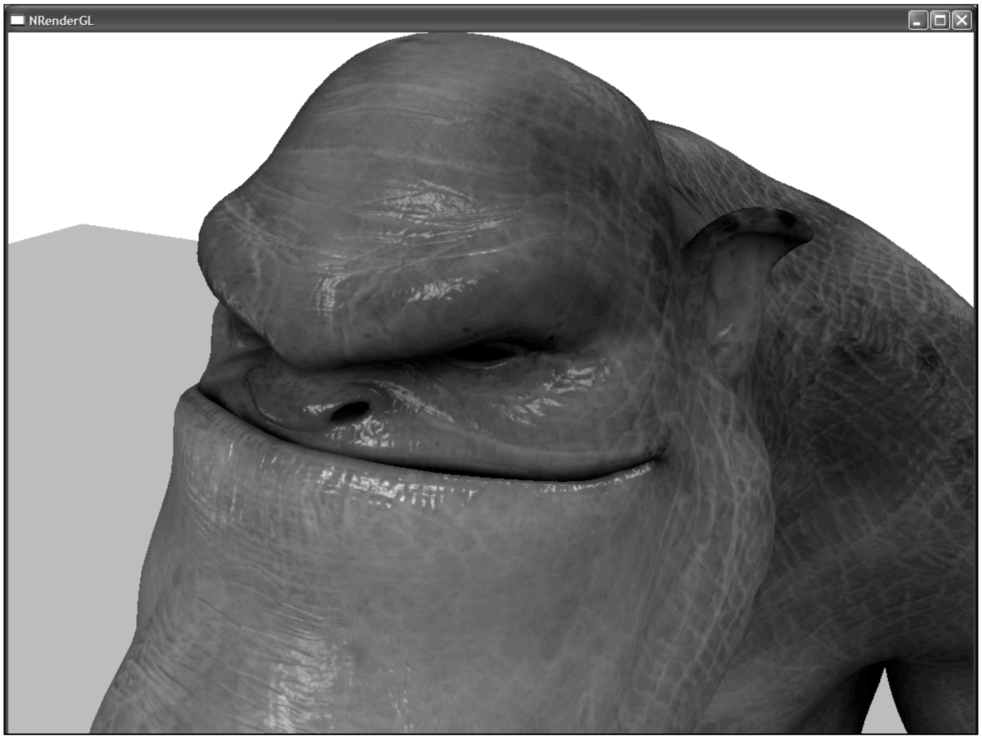
Self occlusion is the main reason why hard to reach areas, such as the corners of rooms, tend to be darker.

Occlusion

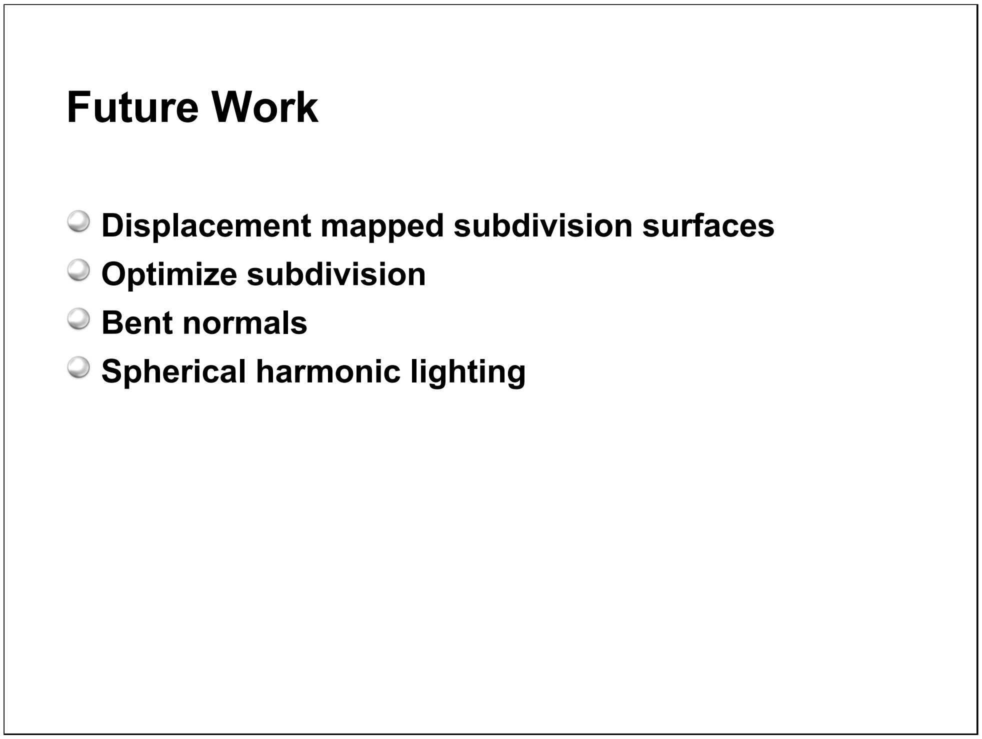


NRenderGL
display
navigation
controls
www.yeah the movie.de
(c)2002 Spellcraft Studio
nVIDIA.
© 2002 NVIDIA CORPORATION


NRenderGL
display
navigation
controls
I Eat Little Kids
www.yeah the movie.de
(c)2002 Spellcraft Studio
nVIDIA.
© 2002 NVIDIA CORPORATION

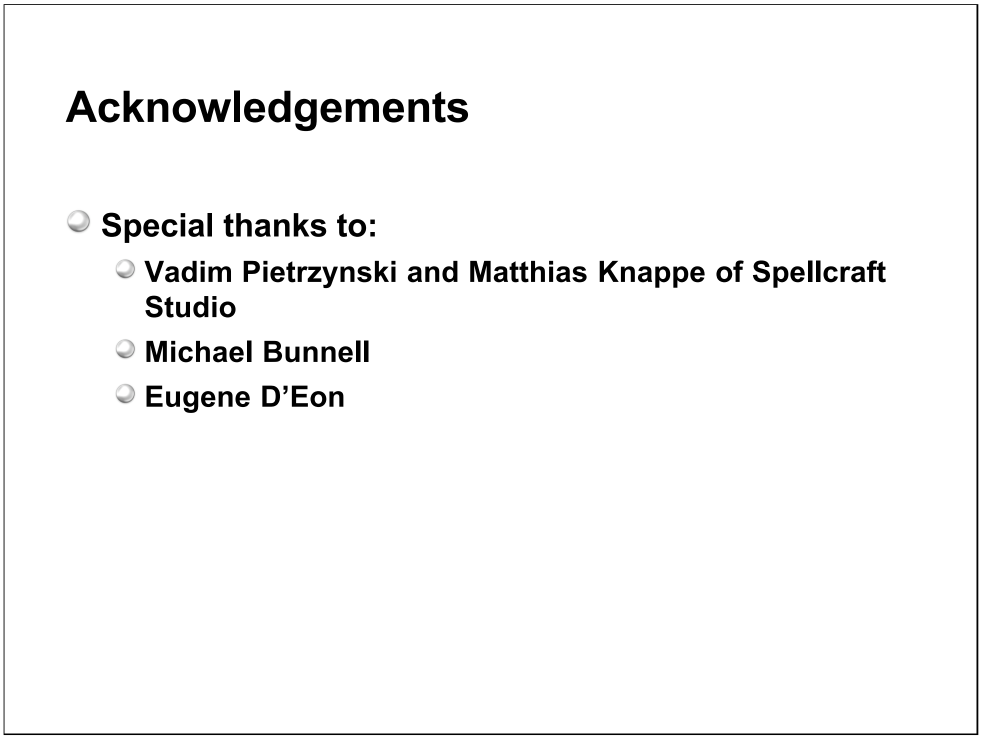


Porcelain shader



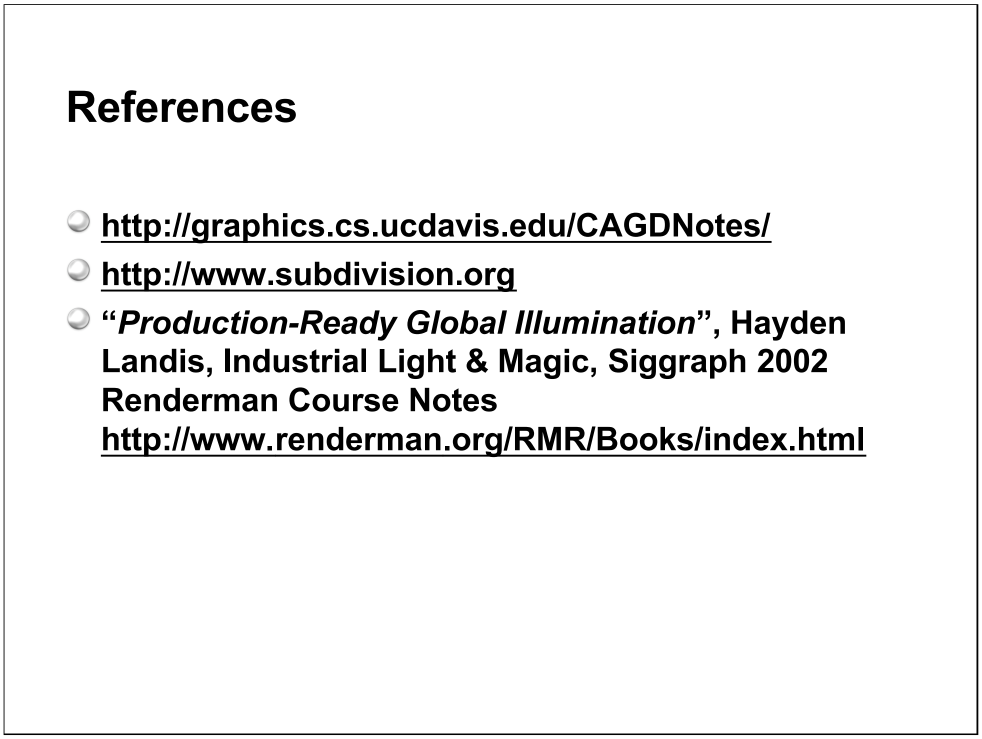
Future Work

- Displacement mapped subdivision surfaces
- Optimize subdivision
- Bent normals
- Spherical harmonic lighting



Acknowledgements

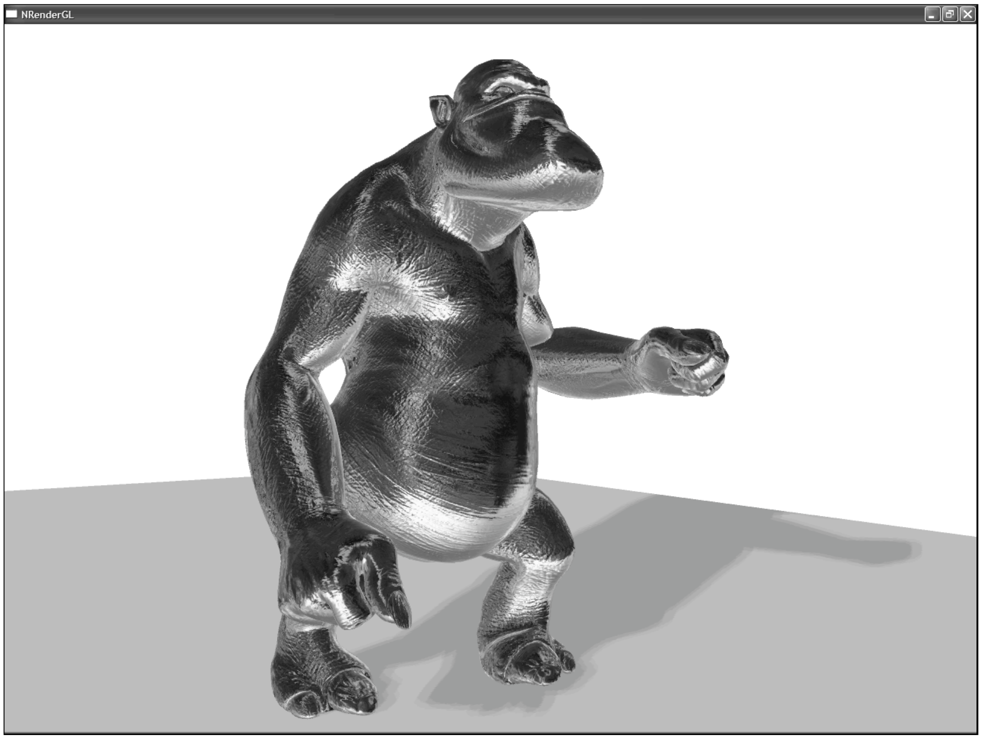
- Special thanks to:
  - Vadim Pietrzynski and Matthias Knappe of Spellcraft Studio
  - Michael Bunnell
  - Eugene D’Eon



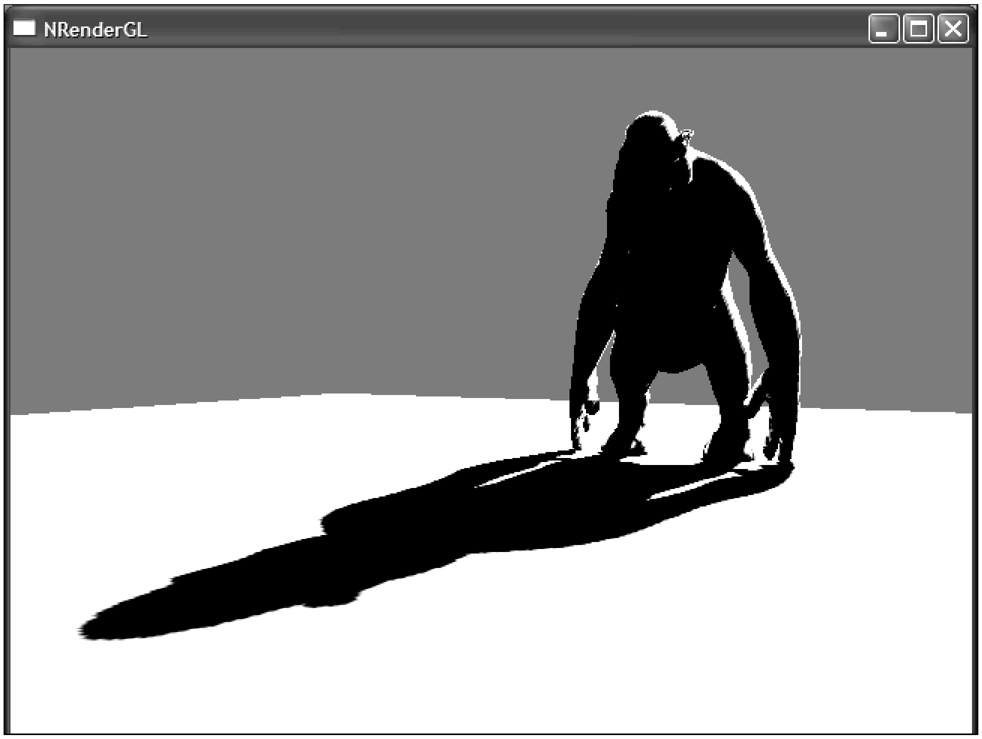
References

- http://graphics.cs.ucdavis.edu/CAGDNotes/
- http://www.subdivision.org
- "*Production-Ready Global Illumination*", Hayden Landis, Industrial Light & Magic, Siggraph 2002 Renderman Course Notes http://www.renderman.org/RMR/Books/index.html

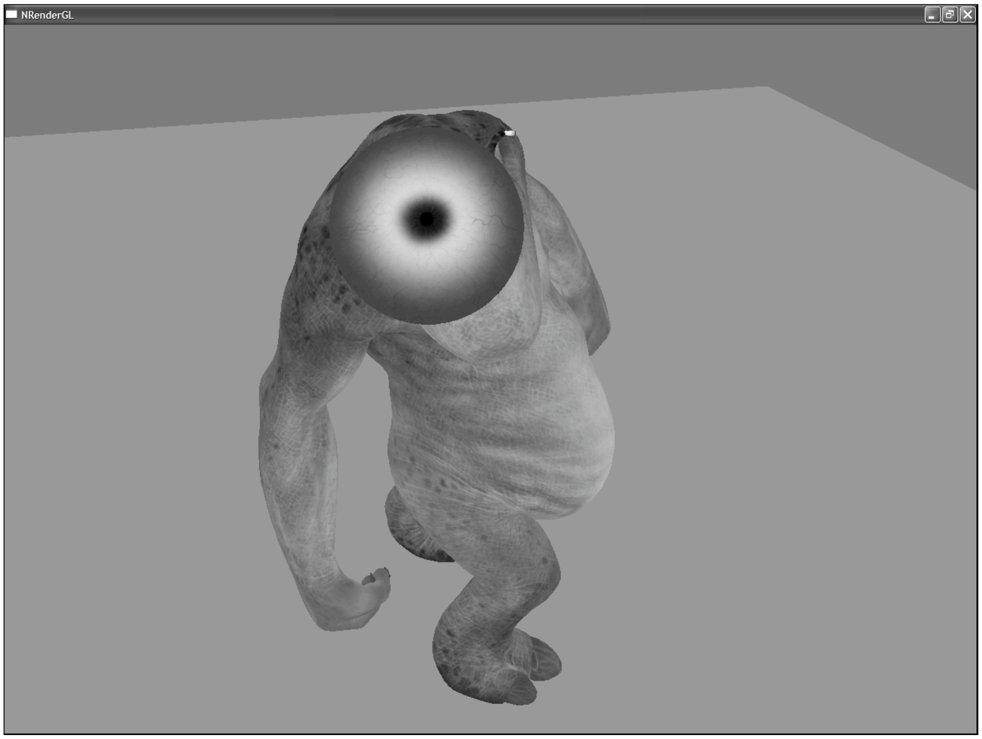
Outtakes



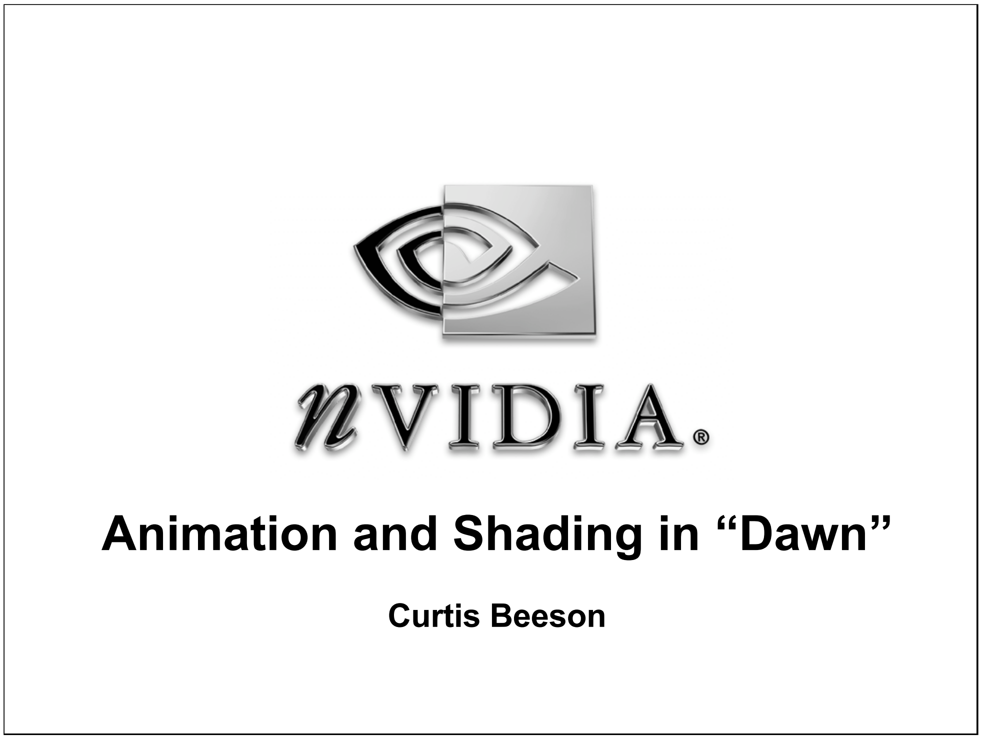
Bumpy shiny test



Shadow test (high noon)



Transform bug



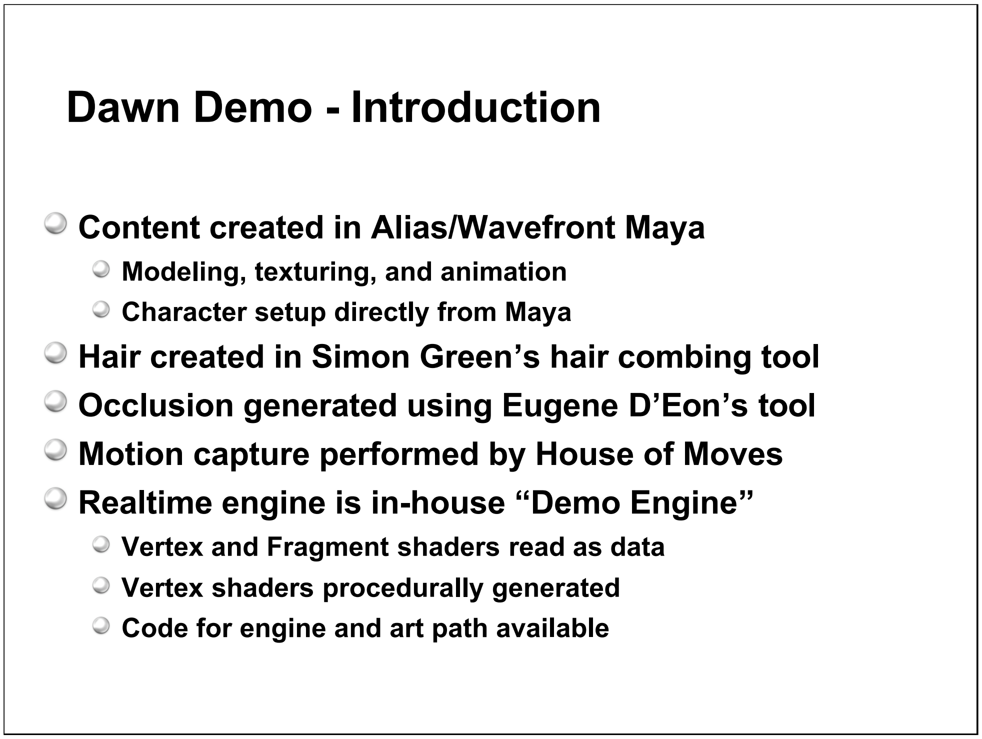
Animation and Shading in “Dawn”

Curtis Beeson

nVIDIA.

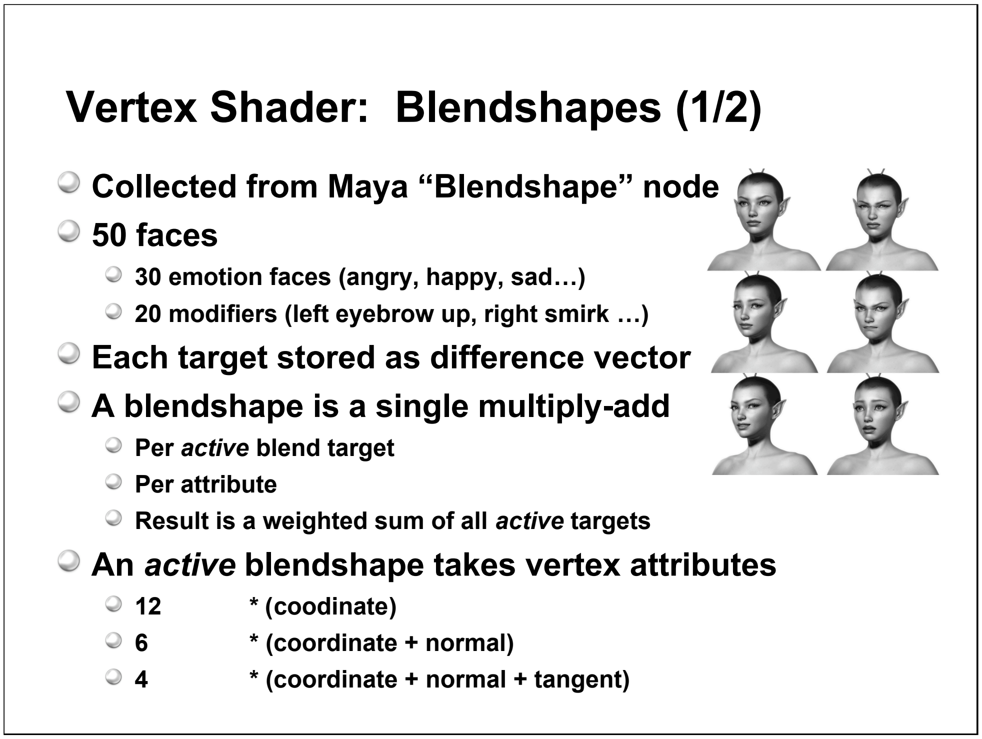
Overview – The Devil is in the Details

- Introduction
- Vertex Shaders
  - Blendshapes
  - Indexed Skinning
  - Fragment Shader Setup
- Fragment Shaders
  - Skin Shader Inputs
  - Skin Shader Algorithm
  - Simplification and Generalization
- Summary



Dawn Demo - Introduction

- Content created in Alias/Wavefront Maya
  - Modeling, texturing, and animation
  - Character setup directly from Maya
- Hair created in Simon Green’s hair combing tool
- Occlusion generated using Eugene D’Eon’s tool
- Motion capture performed by House of Moves
- Realtime engine is in-house “Demo Engine”
  - Vertex and Fragment shaders read as data
  - Vertex shaders procedurally generated
  - Code for engine and art path available



Vertex Shader: Blendshapes (1/2)

- Collected from Maya “Blendshape” node
- 50 faces
  - 30 emotion faces (angry, happy, sad…)
  - 20 modifiers (left eyebrow up, right smirk …)
- Each target stored as difference vector
- A blendshape is a single multiply-add
  - Per *active* blend target
  - Per attribute
  - Result is a weighted sum of all *active* targets
- An *active* blendshape takes vertex attributes
  - 12 * (coodinate)
  - 6 * (coordinate + normal)
  - 4 * (coordinate + normal + tangent)

Vertex Shader: Blendshapes (2/2)

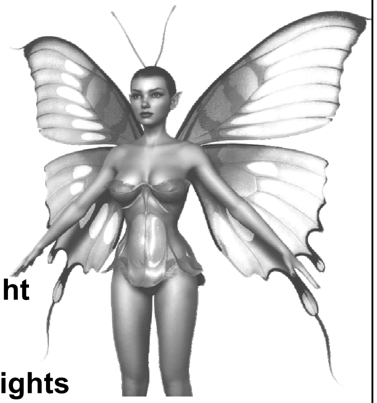
- **In the ApplicationToVertex connector:**

```
// normals & normal targets are float4(normal.x, normal.y, normal.z, occlusion)
struct a2vConnector : application2vertex {
          float4 coord;                float4 normal;
          float3 coordMorph0;      float4 normalMorph0;
          float3 coordMorph1;      float4 normalMorph1;
          float3 coordMorph2;      float4 normalMorph2;
          …
}
```

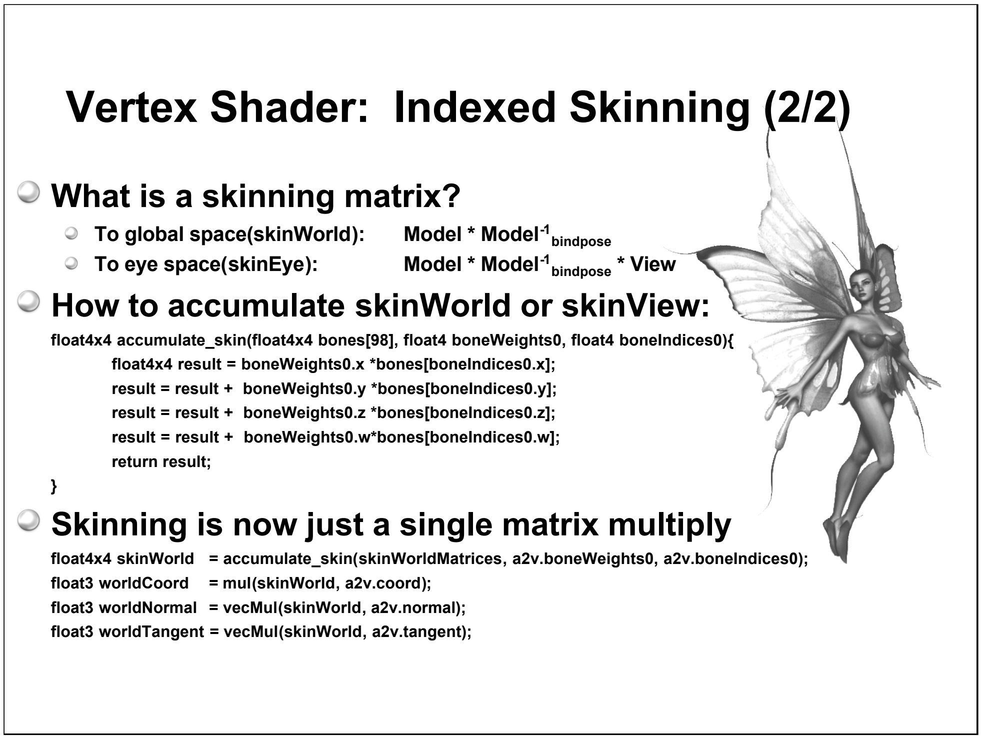
- **In the vertex shader body:**

```
float4 objectCoord = a2v.coord;
objectCoord.xyz = objectCoord.xyz + morphWeight0 * a2v.coordMorph0;
objectCoord.xyz = objectCoord.xyz + morphWeight1 * a2v.coordMorph1;
objectCoord.xyz = objectCoord.xyz + morphWeight2 * a2v.coordMorph2;
…
float4 objectNormal = a2v.normal;
objectNormal = objectNormal + morphWeight0 * a2v.normalMorph0;
objectNormal = objectNormal + morphWeight1 * a2v.normalMorph1;
objectNormal = objectNormal + morphWeight2 * a2v.normalMorph2;
…
```

Vertex Shader: Indexed Skinning (1/2)



- Mesh exported in “Bind Pose”
- Skinning Vertex Data
  - Float4 channel(s) for indices
  - Float4 channel(s) for weights
  - Sort from strongest to weakest weight
- “Accumulated Matrix” Skinning
  - Accumulates all used bones and weights
  - Faster when doing >2 vertex quantities and >2 bones
  - Not intuitive, but the math works out



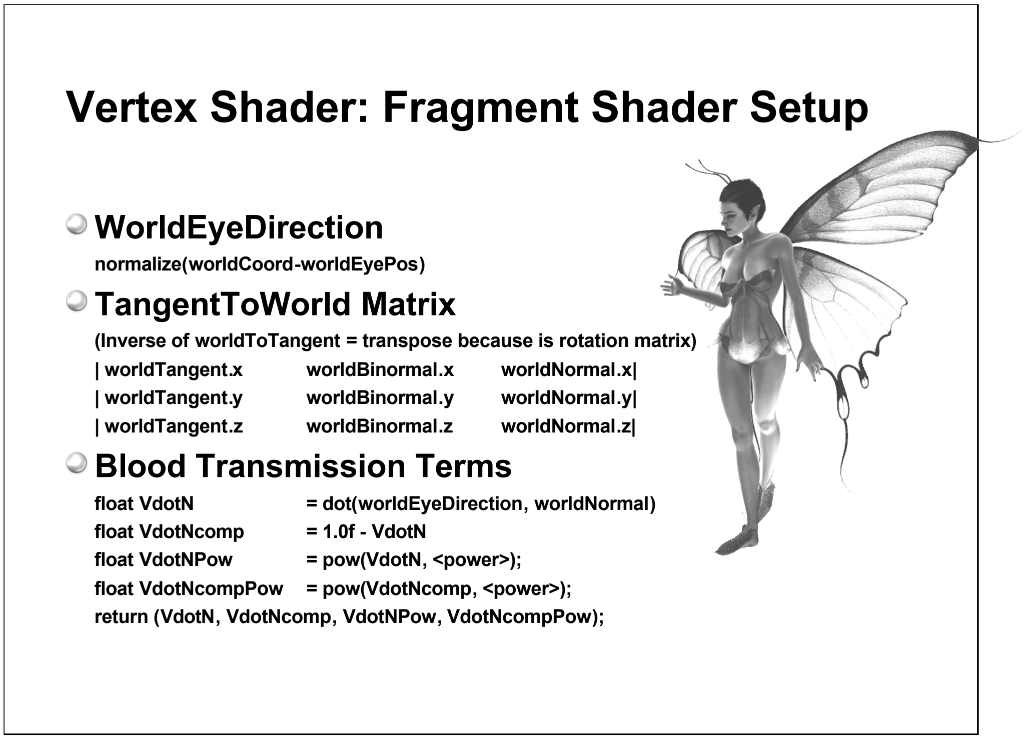
Vertex Shader: Indexed Skinning (2/2)

- What is a skinning matrix?
  - To global space(skinWorld): Model * $\text{Model}^{-1}_{\text{bindpose}}$
  - To eye space(skinEye): Model * $\text{Model}^{-1}_{\text{bindpose}}$ * View
- How to accumulate skinWorld or skinView:

```
float4x4 accumulate_skin(float4x4 bones[98], float4 boneWeights0, float4 boneIndices0){
        float4x4 result = boneWeights0.x *bones[boneIndices0.x];
        result = result +  boneWeights0.y *bones[boneIndices0.y];
        result = result +  boneWeights0.z *bones[boneIndices0.z];
        result = result +  boneWeights0.w*bones[boneIndices0.w];
        return result;
}
```

- Skinning is now just a single matrix multiply

```
float4x4 skinWorld   = accumulate_skin(skinWorldMatrices, a2v.boneWeights0, a2v.boneIndices0);
float3 worldCoord    = mul(skinWorld, a2v.coord);
float3 worldNormal   = vecMul(skinWorld, a2v.normal);
float3 worldTangent = vecMul(skinWorld, a2v.tangent);
```



Vertex Shader: Fragment Shader Setup

- **WorldEyeDirection**

  normalize(worldCoord-worldEyePos)

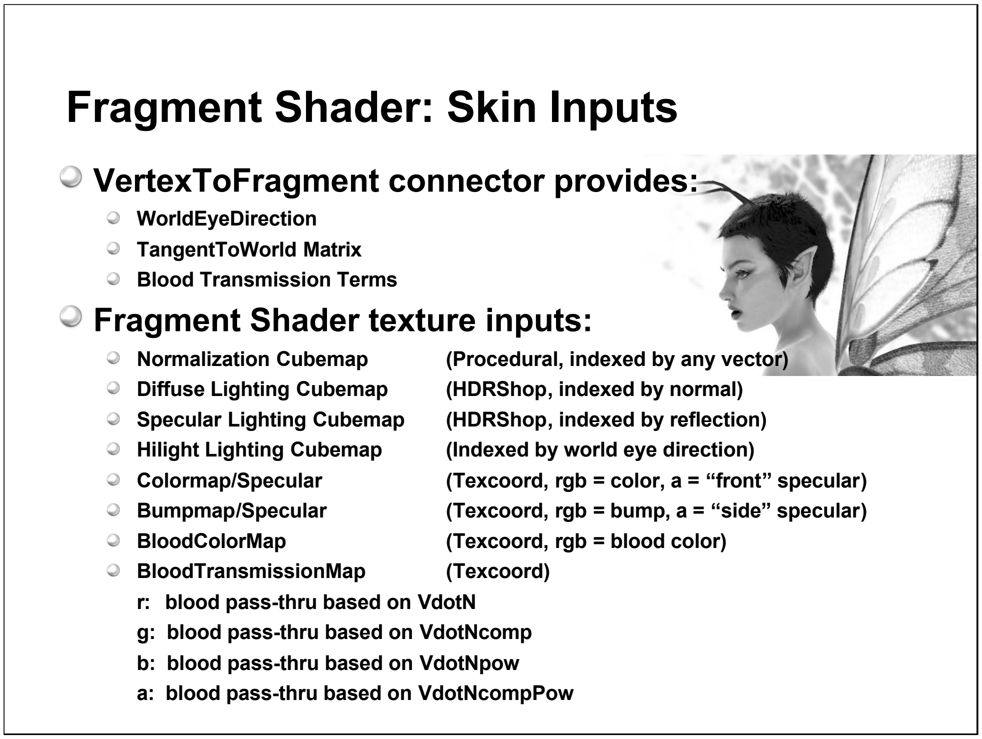
- **TangentToWorld Matrix**

  (Inverse of worldToTangent = transpose because is rotation matrix)

```
| worldTangent.x     worldBinormal.x     worldNormal.x|
| worldTangent.y     worldBinormal.y     worldNormal.y|
| worldTangent.z     worldBinormal.z     worldNormal.z|
```

- **Blood Transmission Terms**

```
float VdotN           = dot(worldEyeDirection, worldNormal)
float VdotNcomp       = 1.0f - VdotN
float VdotNPow        = pow(VdotN, <power>);
float VdotNcompPow    = pow(VdotNcomp, <power>);
return (VdotN, VdotNcomp, VdotNPow, VdotNcompPow);
```



Fragment Shader: Skin Inputs

- VertexToFragment connector provides:
  - WorldEyeDirection
  - TangentToWorld Matrix
  - Blood Transmission Terms
- Fragment Shader texture inputs:

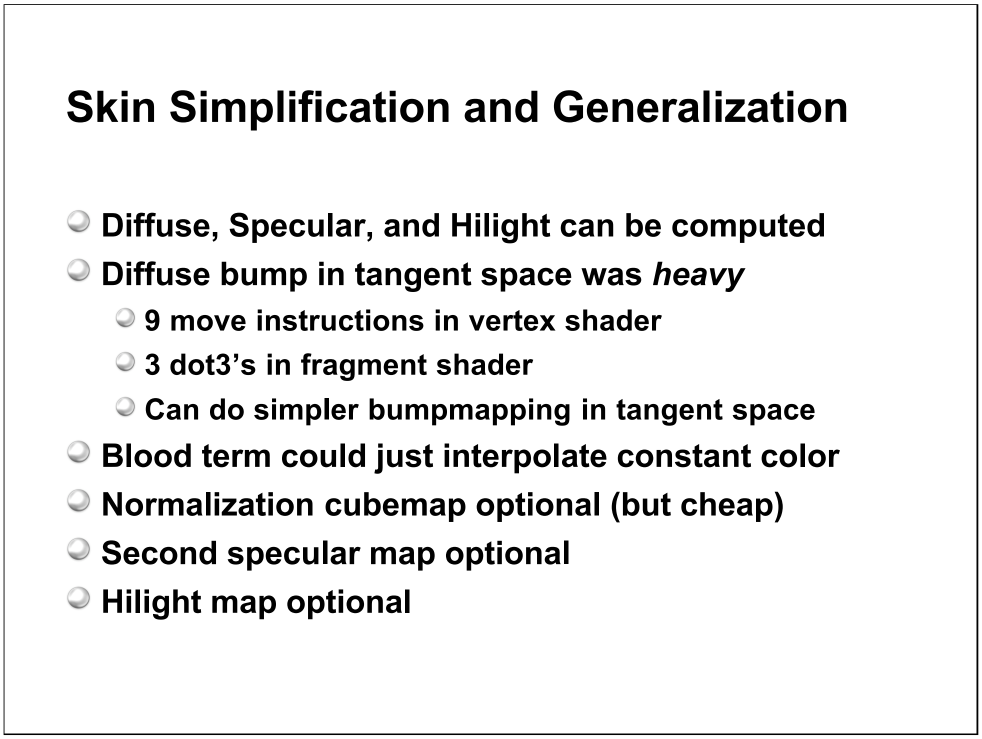
Normalization Cubemap	(Procedural, indexed by any vector)
Diffuse Lighting Cubemap	(HDRShop, indexed by normal)
Specular Lighting Cubemap	(HDRShop, indexed by reflection)
Hilight Lighting Cubemap	(Indexed by world eye direction)
Colormap/Specular	(Texcoord, rgb = color, a = “front” specular)
Bumpmap/Specular	(Texcoord, rgb = bump, a = “side” specular)
BloodColorMap	(Texcoord, rgb = blood color)
BloodTransmissionMap	(Texcoord)

r: blood pass-thru based on VdotN
g: blood pass-thru based on VdotNcomp
b: blood pass-thru based on VdotNpow
a: blood pass-thru based on VdotNcompPow

Fragment Shader: Skin Algorithm

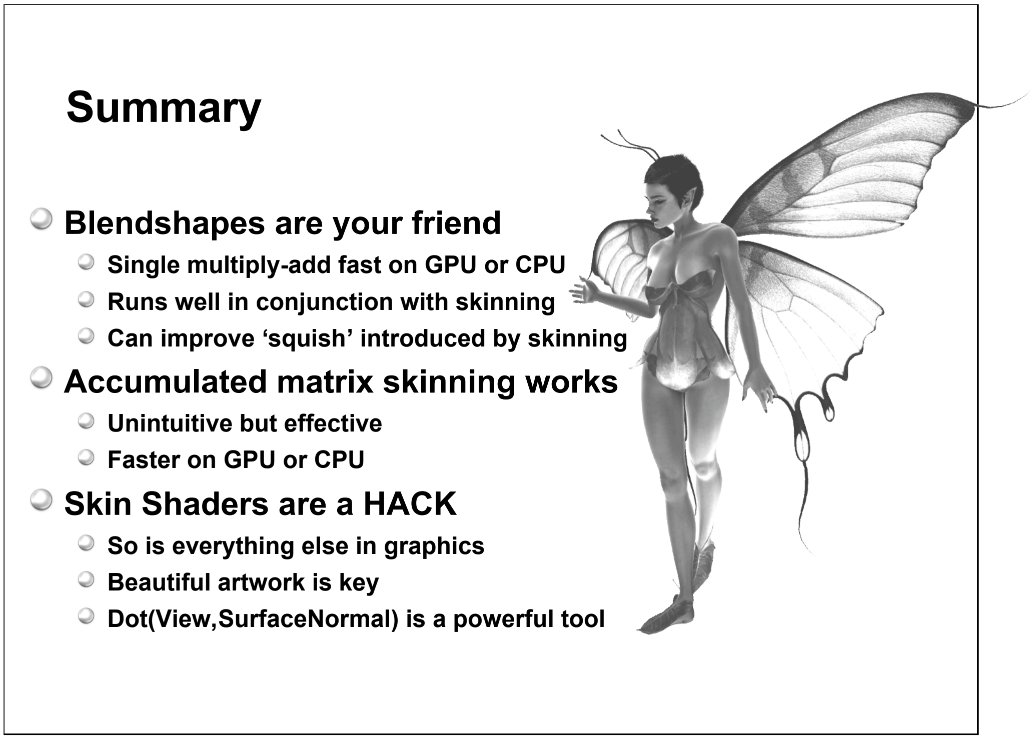
- Like anything, diddle the knobs until it's pretty…
- Our fairy shader ended up as:

```
worldNormal      = TangentToWorldMatrix * BumpMap

diffuseLight     = DiffuseLightCube(worldNormal)
specularLight    = SpecularLightCube(ComputeReflection(worldEyeDir, worldNormal))
passThruLight    = HilightCube(worldEyeDir)

bloodAmount      = dot (BloodTransmissionMap, BloodTransmissionTerms)

diffuseColor     = lerp(ColorMap, BloodColorMap, bloodAmount)
specularColor    = lerp(frontSpecularMap, sideSpecularMap, BloodTransmissionVector.z)

return (occlusion*(diffuseLight *diffuseColor  + specularLight *specularColor + passThruLight))
```



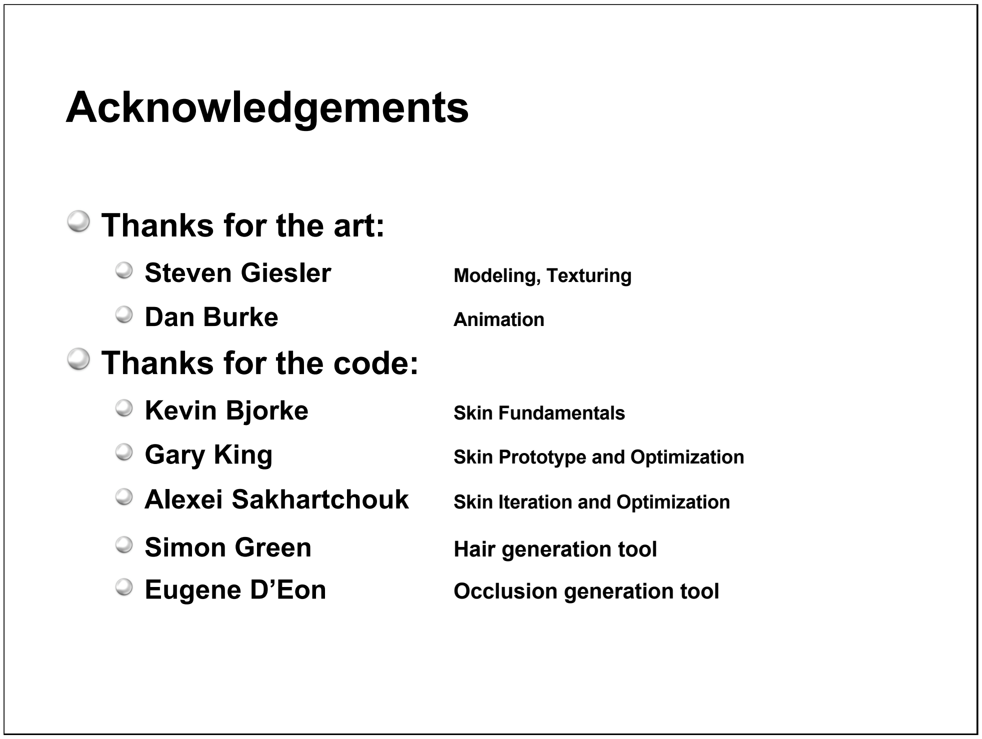
Skin Simplification and Generalization

- Diffuse, Specular, and Hilight can be computed
- Diffuse bump in tangent space was *heavy*
  - 9 move instructions in vertex shader
  - 3 dot3's in fragment shader
  - Can do simpler bumpmapping in tangent space
- Blood term could just interpolate constant color
- Normalization cubemap optional (but cheap)
- Second specular map optional
- Hilight map optional



Summary

- Blendshapes are your friend
  - Single multiply-add fast on GPU or CPU
  - Runs well in conjunction with skinning
  - Can improve ‘squish’ introduced by skinning
- Accumulated matrix skinning works
  - Unintuitive but effective
  - Faster on GPU or CPU
- Skin Shaders are a HACK
  - So is everything else in graphics
  - Beautiful artwork is key
  - Dot(View,SurfaceNormal) is a powerful tool



Acknowledgements

- Thanks for the art:

Steven Giesler	Modeling, Texturing
Dan Burke	Animation

- Thanks for the code:

Kevin Bjorke	Skin Fundamentals
Gary King	Skin Prototype and Optimization
Alexei Sakhartchouk	Skin Iteration and Optimization
Simon Green	Hair generation tool
Eugene D’Eon	Occlusion generation tool

nVIDIA.
© 2002 NVIDIA CORPORATION

Credits

- Art Team
  - Dan Burke, Bonnie O’Claire, Steven Gielser, Daniel Hornick
- Programming Team
  - Curtis Beeson, Joe Demers, Simon Green, Gary King, Hubert Nguyen, Thant Tessman
- Interns
  - Eugene D’Eon, Denis Dmitriev, Dean Lupini, Jonathan McGee, Alex Sakhartchouk
- Management
  - Mark Daly

Questions…

?