Introduction

- Demos for GeForce FX & Quadro FX
- “Dawn” & “Dusk”
- “Toys”
- “Time Machine”
- “Last Chance Gas”
- “Ogre” (Spellcraft Studio)
- “Vulcan”
NVIDIA Demo Engine

- All demos were developed using the same engine
- Object-oriented scene graph library
- Handles state management, culling, sorting
- Includes Maya and MAX converters
- Uses Cg compiler and runtime for shaders
Hair Rendering & Vertex Shaders in “Dawn”
Characters Look Better With Hair
Rendering Hair

Two options:
- 1) Volumetric (texture)
- 2) Geometric (lines)

We have used volumetric approximations (shells and fins) in the past (e.g. Wolfman demo)

Doesn’t work well for long hair

We considered using textured ribbons (popular in Japanese video games). Alpha sorting is a pain.

Performance of GeForce FX finally lets us render hair as geometry
Rendering Hair as Lines

- Each hair strand is rendered as a line strip (2-20 vertices, depending on curvature)
- Problem: lines are a minimum of 1 pixel thick, regardless of distance from camera
- Not possible to change line width per vertex
- Can use camera-facing triangle strips, but these require twice the number of vertices, and have aliasing problems
Anti-Aliasing

Two methods of anti-aliasing lines in OpenGL

- **GL_LINE_SMOOTH**
  - High quality, but requires blending, sorting geometry
- **GL_MULTISAMPLE**
  - Usually lower quality, but order independent

We used multisample anti-aliasing with "alpha to coverage" mode

By fading alpha to zero at the ends of hairs, coverage and apparent thickness decreases

"SAMPLE_ALPHA_TO_COVERAGE_ARB" is part of the ARB_multisample extension
Hair Without Antialiasing
Hair With Multisample Antialiasing
Hair Shading

- Hair is lit with simple anisotropic shader (Heidrich and Seidel model)
- Low specular exponent, dim highlight looks best
- Black hair = no shadows!
- Top of head is painted black to avoid skin showing through
- We also had a very short hair style, which helps
Vertex shaders in “Dawn”

Vertex shaders in Dawn are all generated procedurally using our 'NVVertexShaderGenerator'.

This work is done at export time:
- Looks at the values requested by the fragment shader (like worldspace normal, eyespace reflection, etc)
- Tries to generate those quantities given some information about the mesh (number of bones, number of bones per vertex, number of blendshapes, etc)
- It even uses workaround rules for taking cross products or just re ortho-normalizing if we couldn’t pass those quantities

This job is possible because we concatenate various pieces of Cg code: connectors and function arguments are very convenient.

The Shader generator writes some combinations of Cg-generated code and choose the best solutions.
Blending shapes in “Dawn” (1)

- done in Maya using blendshapes
- Dawn has a total of 50 blendshapes
  - some are complex emotion faces (happy, sad, angry)
  - some are just modifiers (ears back, eyebrow up, etc)
- each blendshape is the difference between the blendshape and the original
  - you can apply any number of blendshapes at once,
  - weight > 1.0 to exaggerate the look.
Blendshapes in “Dawn” (2)

For each blendshape, we store
- an 'object space coord diff'
- an 'object space normal diff' vector

actual CG code for applying 5 blendshapes to both coordinates and normals is:

```c
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;
objectCoord.xyz = objectCoord.xyz + morphWeight3 * a2v.coordMorph3;
objectCoord.xyz = objectCoord.xyz + morphWeight4 * a2v.coordMorph4;

float4 objectNormal = a2v.normal;
objectNormal = objectNormal + morphWeight0 * a2v.normalMorph0;
objectNormal = objectNormal + morphWeight1 * a2v.normalMorph1;
objectNormal = objectNormal + morphWeight2 * a2v.normalMorph2;
objectNormal = objectNormal + morphWeight3 * a2v.normalMorph3;
objectNormal = objectNormal + morphWeight4 * a2v.normalMorph4;
objectNormal.xyz = normalize(objectNormal.xyz);
```
Blendshapes in “Dawn” (3)

You see that only 5 on 50 blendshapes are available
- Enough for Artists
- To solve the limited resource: Took 10 vertex registers
- We used 3 kind of emotions at a time + 2 modifiers for various parts of the face
- ‘diff’ tangent & binormal are computed
  - Take a perpendicular vector for tangent
  - Do a cross product for binormal
Skinning in “Dawn”

- Indexed skinning in the vertex shaders across the entire model: 98 bones for this skeleton
- But we broke the mesh up by shaders:
  - 22 separate meshes
  - Each mesh using ~20-30 bones
- No use of for() loop: our 'NVVertexShaderGenerator' procedurally generated the series of bones.
Dawn Fragment Shaders (1)

Fragment shaders is a mix of effects that we were playing with (Kevin Bjorke, Gary King, Curtis Beeson)
elements selected more out of artistic preference than any logical necessity

VertexToFragment Connector Inputs:
1. WorldSpaceEyeDirection
2. TangentToWorldspaceMatrix ([tan, binorm, norm]^T)
3. Blood Transmission Vector (1, v.n, (1-v.n))
4. Color-Per-Vertex occlusion (could have been a texture but our tools calculated occlusion on a per-vertex basis, so we just stored in the .w of normal).
Dawn Fragment Shaders (2)

Fragment Shader Texture Inputs:
1. Colormap + Front Spec (rgb = color, a = "front specular map")
2. Bumpmap + Side Spec (rgb = bump, a = "side specular map")
3. Blood Texture (rgb = color of blood under skin surface)
4. Blood Transmission Texture: Describes how to let the underlying blood texture to be revealed
   - r = constant minimum amount of blood to show
   - g = how much to reveal based on v.n
   - b = how much to reveal based on 1-v.n
5. Hilight Cubemap: Blurry image of bright lights in cubemap used like lens flare
6. Diffuse Lighting Cubemap: Incoming diffuse light indexed by world normal
7. Specular Lighting Cubemap: Incoming specular light indexed by world normal
8. Normalization Cubemap: for Per-pixel normalization
The basic algorithm that we employ looks something like:

1. `worldNormal = TangentToWorldMatrix * BumpMap;`

2. `diffuseLight = DiffuseLightCubemap (worldNormal);`

3. `specularLight = SpecularLightCubemap(ComputeReflection(worldNormal))`

4. `..../..`
Dawn Fragment Shaders (4)

4. `passThruLight = HilightCubemap(worldEyeDirection);`
   This is doing the glow effect: a kind of back-lighted ‘duvet’ on the skin

5. `bloodAmount = dot (BloodTransmissionVector, BloodTransmissionTerms);`
   Brings a scalar value depending on how we’re looking at the skin

6. `diffuseColor = lerp(ColorMap, BloodTexture, bloodAmount);`
   Is a mix of the skin surface and what’s under the skin: the blood

7. `specularColor = lerp(front specular map, side specular map, BloodTransmissionVector.z);`
   A simple approximation of fresnel: (1-v.n) is BloodTransmissionVector.z

8. `output = occlusion * (diffuseLight * diffuseColor + specularLight * specularColor + passThruLight);`
Dawn’s “Evil Twin” – *Dusk*

- Dusk’s hair, animation, &c largely similar to Dawn
- High performance of GeForce FX 5900/QuadroFX 3000 permit additional shadow passes and spotlighting
The Time Machine demo
Aging Materials: 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
Crucial Material: Automotive Paint

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
Techniques: Faux-BRDF Reflection

- Our solution: project BRDF values onto a single 2D texture, and factor out the intensity
  - Compute intensity in real-time, using \((N \cdot H)^s\)
  - Texture varies slowly, so it can be low-res (64x64).
  - 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 dXdT 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 dXdT (velocity) maps
  - Can be stored in alpha in a DXT5 texture.
“Chrome” examples

Rust lit & shadowed  Rust reveal  Chrome lit & shadowed  Final

Time
Techniques: Dynamic Bump mapping

Scaling a normal map by a constant doesn’t change surface topology

Normals are just derivatives of height map

\[
\iint N(x, y) \, dx \, dy = h(x, y) \quad \iint cN(x, y) \, dx \, dy = ch(x, y)
\]

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

\[
N'(x, y) = \frac{\partial h'(x, y)}{\partial x \, dy}
\]

This is analogous to techniques that use the GPU to solve partial differential equations. We used this to create dynamic bubbles.
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.

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

Height map  \*  Growth factor at \( t=n \)  =  New normals
Performance Concerns…

Textures Used, continued…

Surround Maps

- Recomputing the normal requires knowing the heights of 4 texels \((s-1,t), (s+1,t), (s,t+1)\) and \((s,t-1)\)
- Each height is only 1 8-bit component
- Instead of 4 dependent fetches, we can pack all into 1
  
  \[
  S(s,t) = [H(s-1, t), H(s+1, t), H(s,t-1), H(s,t+1)]
  \]

- Saved 4 math ops and 3 texture fetches + shuffle logic
Real-Time Depth of Field
Fun With Real-Time Post-Processing
Simple Depth of Field

- Render scene to color and depth textures
- Generate mipmaps for color texture
- Render full screen 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: jittered samples
  - We only took one sample, i.e. we are fetching the color with a small translation (jitter), depending on texcoord

The pseudo-code looks like this:

```plaintext
depth = tex(depthtex, texcoord)
coc = abs(depth * COC_scale + COC_bias);
jitter = tex(jittertex, texcoord)
color = txd(colortex, texcoord+jitter)
```
Noise vs. Interpolation Artifacts

With Noise

Without Noise
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. Hybrid Layered DOF

Hybrid Layered DOF

Simple DOF
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 around the vector (1,1,1)
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
Inside the “Ogre” Demo
The Ogre Demo

A real-time preview of Spellcraft Studio’s in-production short movie “Yeah!”
- Created in 3DStudio MAX
- Used Character Studio for animation, plus Stitch plug-in for cloth simulation
- Available at: www.yeahthemovie.de

Our aim was to recreate the original as closely as possible, in real-time
Subdivision Surfaces

Content
- Characters were modeled with subdivision in mind (using 3DSMax “MeshSmooth/NURMS” 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
- May be accelerated on future GPUs
Control Mesh vs. Subdivided Mesh

4000 faces

17,000 triangles
Disadvantages of Realtime Subdivision

- CPU intensive
  - But we might as well use the CPU for something!
- View dependent
  - Requires re-tessellation for shadow map passes
- Mesh topology changes from frame to frame
  - Makes motion blur difficult
Ambient Occlusion Shading

- Helps simulate the global illumination “look” of the original movie
- Self occlusion is the degree to which an object shadows itself
  - “How much of the sky can I see from this point?”
- Related to: geometric exposure, accessibility shading, dirt mapping
- Simulates a large spherical light surrounding the scene
- Popular in production rendering – Pearl Harbour (ILM), Stuart Little 2 (Sony)
Occlusion
How To Calculate Occlusion

- Shoot rays from surface in random directions over the hemisphere (centered around the normal)
- The percentage of rays that hit something is the occlusion amount
- Can also keep track of average of un-occluded directions – “bent normal”
- Some Renderman compliant renders (e.g. Entropy) have a built-in occlusion() function that will do this
- We can’t trace rays using graphics hardware (yet)
- So we pre-calculate it!
Occlusion Baking Tool

Written by Eugene D’Eon, University of Waterloo

Uses ray-tracing engine to calculate occlusion values for each vertex in control mesh

We used 128 rays / vertex

Stored as floating point scalar for each vertex and each frame of the animation

Calculation took around 5 hours for 1000 frames

Subdivision code interpolates occlusion values using cubic interpolation

Used as ambient term in shader
Last Chance Gas
Creating Soft Shadows

- Precompute solar visibility for each texel in the scene for 256 timesteps

- Store compressed visibility function as ranges of time when texel enters and leaves shadow

- We used up to 5 textures per surface, one quarter the size of the diffuse maps (half x half), to get up to 20 shadow transitions per texel over time
Rendering Soft Shadows

- Uncompress visibility function and integrate over time in fragment shader to get soft lighting
- This is just convolving a box filter with a series of more box filters
- We can even do these separately, and 4 at a time with vector min & max:
  \[ \text{min}(t+dt, t_{\text{shadow}}) - \text{max}(t_{\text{light}}, t-dt) \]
Global Illumination

- Precompute radiosity for scene and bake into textures
- Use radiosity texture as an ambient map
- Modulate ambient with cool sky color and diffuse with warm sun color
High Dynamic Range Lighting

- Render the scene, storing luminance/10 in alpha
- Create quarter size color texture scaled by alpha
- Iteratively create multiple downsampled versions
- Add the average of all of them on top of the main rendered screen to get softer edges and a nice glow effect
Analytic Sky Model

- Use established functions to match sky color for various values parameters (latitude, haziness, etc.)
- Precalculating constant subexpressions on the CPU allows vertex shader to calculate sky color from these parameters
- Or, for constant haziness values, you can create a 3D texture parameterized by time of day, azimuth and altitude
- Preetham et al., “A Practical Analytic Model for Daylight”
A Creature Made of Fire

- How to create fire in realtime?
  - Particles
  - Procedural Flow
  - Video-Textured Sprites
- Sprites work in realtime and are great even for film effects, e.g. Lord of the Rings
- Live-Action fire sprites, mixed with particle-rendered smoke sprites
Animating Fire

- Multiple emitters on body
- Particles drift away with age
- Sprites are randomly oriented up/down/left/right
- Sorted and Alpha-Blended
- Several hundred particles per frame
- Sprites are rendered to a smaller pbuffer than then full screen to optimize fill rate
Heat and Glows

- Heat effects are added as “displacement particles” and used in compositing.
- Final images are distorted, composited, blurred.

Cloud of “heat particles”
Questions?
devrelfeedback@nvidia.com