Transforming Production Workflows with the GPU

Kevin Bjorke, NVIDIA

SIGGRAPH 2004

Overview

- Games, Films, TV Production
- All are getting rapidly more complex and more expensive (and producers are getting increasingly risk-averse), but standard price is still $39.95
- All are intertwining: games of movies, movies from games, and audiences expect similar looks between them

"MRT" visualization
of texture coordinates

The Core Takeway Information:

- http://developer.nvidia.com/

Four Stages of Knowledge

- What
- About
- How
- Why

Realities of Production

- Cinematic Effects, via Programmable Shading, are the Most Powerful Artistic Tool Yet for Games
- But it's an Uphill Battle
  - Hard to implement and experiment
  - Hard to get into game engines
  - Even harder to debug

"Thad" from Animatrix – Character © Silver Pictures

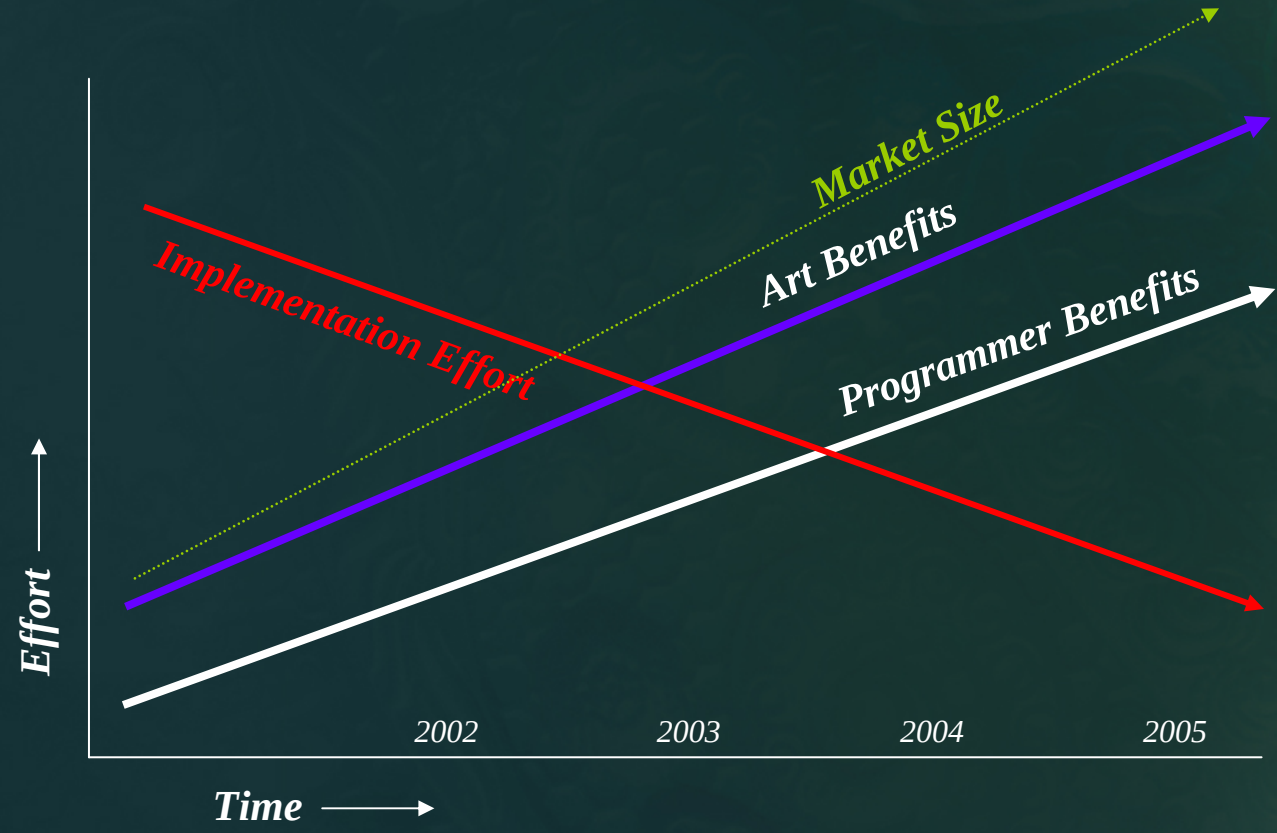
*Part I available at http://developer.nvidia.com/

Applying GPU Power

- Shader development is increasing in importance for anyone who works with pixels
  - Game Makers
  - Animators
  - Lighting TDs
  - Compositors
- **Shader Metaformats**: FX and CgFX
- Tools and Methods to Harness the GPU for everyone

Programmable Shading: When to Add to Your Production Pipeline?

- Every studio will have its own “break even” point



Look Development

- “Look Development” is when we decide what’s important (and what’s not), and lay down the elements of style for any project (or part of a project)
- The earlier in development that the “look” is determined, the better it is (and the cheaper it is to use)

Look Development in Games

- In gaming, look is often a byproduct of engine design
  - Hard for Artists to Guess at anything other than “Lowest Common Denominator,” or what they saw in the last demo
  - Design tends to be conservative and safe
  - Concerned with technical limitations
- In films, development is usually done initially without slavish attention to implementation “details” like budget
  - Artists completely free
  - Concerned with story

Test from *Animatrix* – © Silver Pictures

Developing Shaders: Programmers

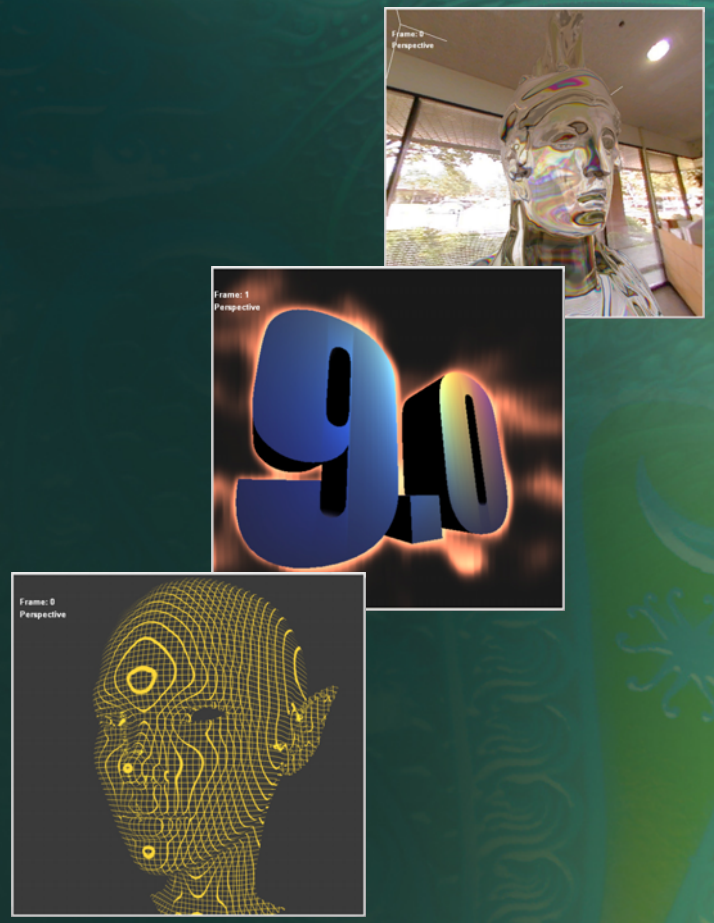
- Shading tools are important for both Programmers and Designers
- To be complete for modern game engines, tools have to support ideas like:
  - Render-To-Texture (RTT)
  - Multiple Render Targets (MRT)
  - Render States like stencil, alpha blend, etc.
  - Custom Texture Maps (e.g. Normalization cubes, noise)
  - SHIrradiance and PRT
  - Management details to make sure complex ordering matches any specific game engine's render loop
- How to get results in *and out* of your game engine or application, at every stage of production?

Developing Shaders: Artists

- Artists want to see what they design when they design it, not just guess at what it might look like later
- Not just seeing the correct models, but also the correct lighting environment, so that *the shaders viewed and/or developed can really be the ones used in-game.*
- Implementations are typically different in each different DCC application (Maya versus Max versus XSI versus….)
- We want to accommodate console-game designers, too: “What might my model look like on Playstation?”

HLSL FX Metaformat

- A Tool Built for the Task
- Easily swappable for artists and programmers
- Already a part of DirectX and **XNA** – no external SDK required
- Some viewing support already in XSI, 3DStudio Max, and Maya
- Already used in shipping, high-performance games

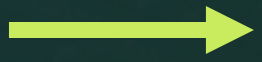


FX Composer

- The Gold Standard in FX editing tools, with more complete support than any other
- Combine & customize shaders
- Extensible
- Use any 3D model
- Performance tuning
- Works with RTZen etc
- **http://www.fxcomposer.com/**

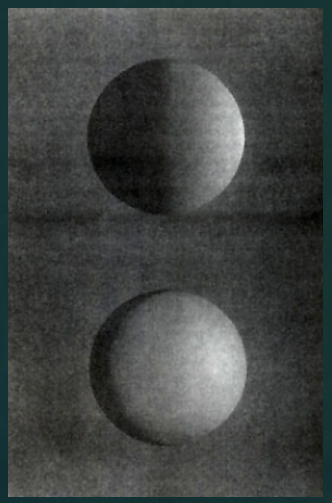
A Shading Sketchbook

- FX Composer gives artists and programmers an environment to play with complex ideas, without needing to write a whole C++ game engine to try them out!



Building a Library of Shaders

- FX Composer ships with lots of sample shaders
  - Any HLSL FX shader can be used, from other shader tools too (e.g., RTZen)
- Do experiments, save them and keep them around – even mistakes – you'll use them someday!
- Save, trade, and collect 'em

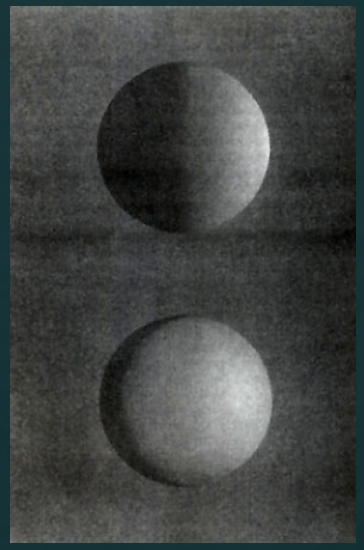


Ruskin's Shading Exercises, 1877

Bjorke's Dumb Mistake, 2003

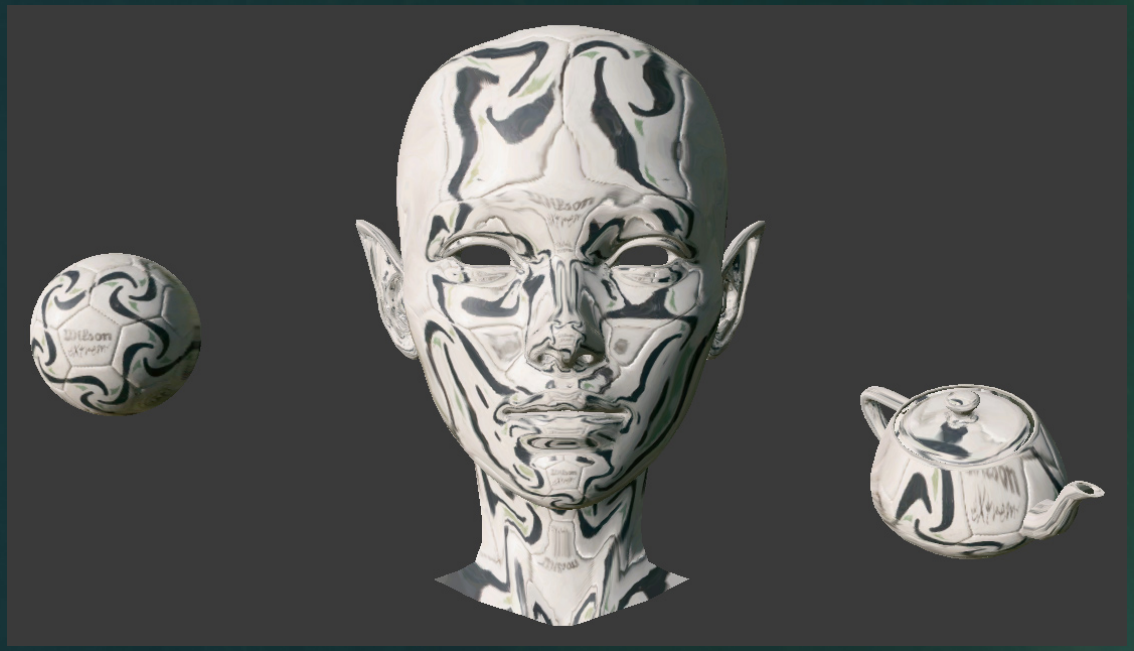
Turning Pencil Sketches into Shaders

- Side Note:
  - A shaded sphere is trivial to turn into a shader
    (See 2001 Siggraph: “The Lit Sphere”
    http://portal.acm.org/citation.cfm?id=781004&dl=ACM&coll=portal)
  - Useful as color reference
  - Beware tiny details (like JPEG noise), they smear



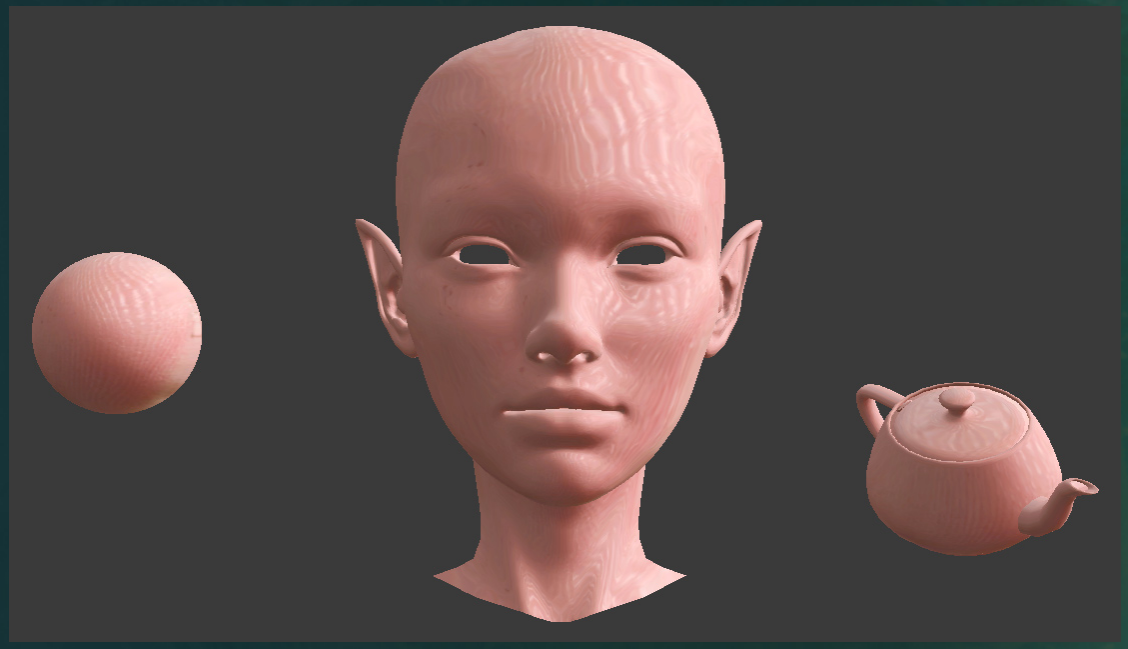
Fanciful examples

- Photos will distort
- Soccer balls are probably rarely useful, but *cheap* – only ONE cycle
- *Can* we do something generically useful with this?



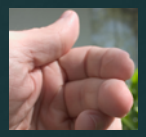
Does the shape have to be a sphere?

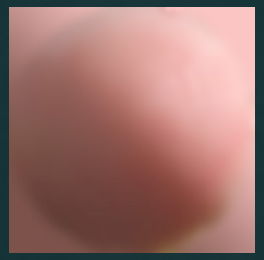
- Not if we're willing to do some work with Photoshop
  - I like "Liquify" and the Smudge/Stamp Tools

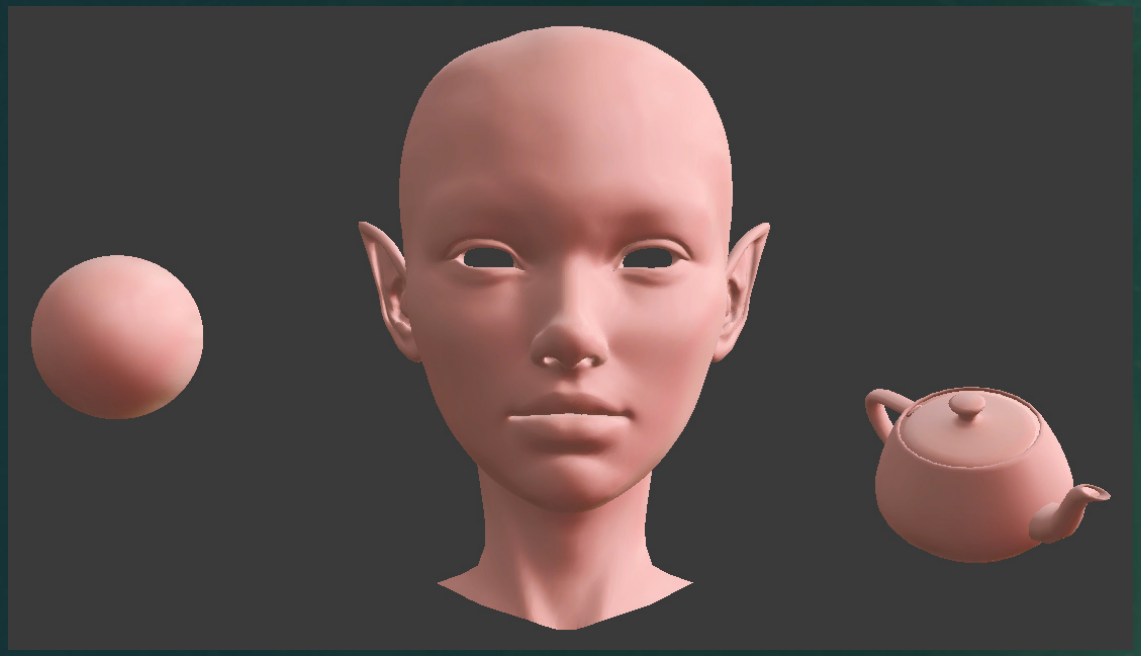


Refining the Color

- Gaussian Blur in the texture to isolate the color
- Great to mix with other shading models, this is *so* cheap








Render Pipelines
Identification and Feeding

Render Pipeline: The “A-B-C” Model

- Many programs still use the simplest possible Z-culled double-buffered pipeline:
  - Clear to BG Color
  - Render Object A
  - Render Object B
  - Render Object C
  - Render Object D…
  - Display complete scene!
- Examples include many older games and most all DCC and CAD applications.

Programmable Shading in A-B-C

- The method used by 3DStudio Max DirectX9 Materials and in the CgFX plugins for Maya etc
  - Clear to BG Color
  - Render Object A (potentially in multiple passes)
  - Render Object B (potentially in multiple passes)
  - Render Object C (potentially in multiple passes)
  - Render Object D…
  - Display complete scene!
- A great improvement in appearance, but still no render-to-texture effects etc. – so shaders may or may not be the “real” final shaders

Going Beyond A-B-C

- More complex rendering, like that in the Maya Hardware Render Device, or modern game engines, uses a rich mixture of 3D and 2D operations
  - Textures created on the fly
  - Special masks for compositing
  - Stencil
  - Glints and glows
  - Etc…

A Simple Complex Pipeline….

- (this is a *simplified* example):
  - Deep clear (floating point image)
  - Render object D & B into one shadow map
  - Deep clear (floating point image)
  - Render object C into another shadow map
  - Clear to bg texture
  - Render MRTs – object A and depth using previous shadow passes
  - Render objects D&B using shadow passes
  - Save to texture
  - Blur vertically
  - Blur again, horizontally
  - Restore unaltered BG image
  - Add part of blurred image to create bloom in brightest areas
  - Render object C using blurred bg texture as refraction
  - Add textured fog based on MRT depth samples
  - Display complete scene

Complex Pipelines

- Game pipelines are becoming as complex as film-production pipelines, which may have dozens or hundreds of composited elements, each broken into multiple passes for shadowing, glows, diffuse/specular separation and balancing, live-action BG plates, etc.

Complex Pipelines for Games

- While film pipelines use multiple rendering and compositing tools, interactive applications have to do rendering *and* compositing operations in the same app – and fast
- Game appearances may require different handling depending on the hardware – high end, low end, or console (potentially multiples)

Managing Complexity

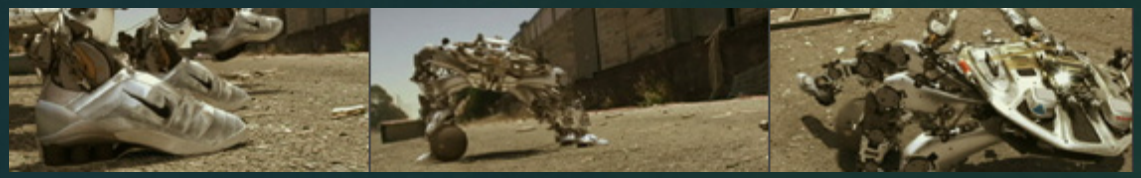
- Programmability lets us bundle ideas together and control them flexibly
- Programmability up and down the entire chain:
  - GPU shading units
  - Graphics API state
  - Tracking of basics like mouse, object transforms, etc
  - High-level scene building, control, data import and export

Managing Scale

- Films have Massive Scale
- Lots of Models
- Lots of Polygons
- *Lots of Shaders*
  - *Toy Story:* 1300
  - *Bugs:* Double
  - *Monsters Inc:* “thousands”
- Lots of Compositing Layers (sometimes hundreds)
- Long Schedules
  - Instant Rendering shaves off *some* schedule…

Long Schedules

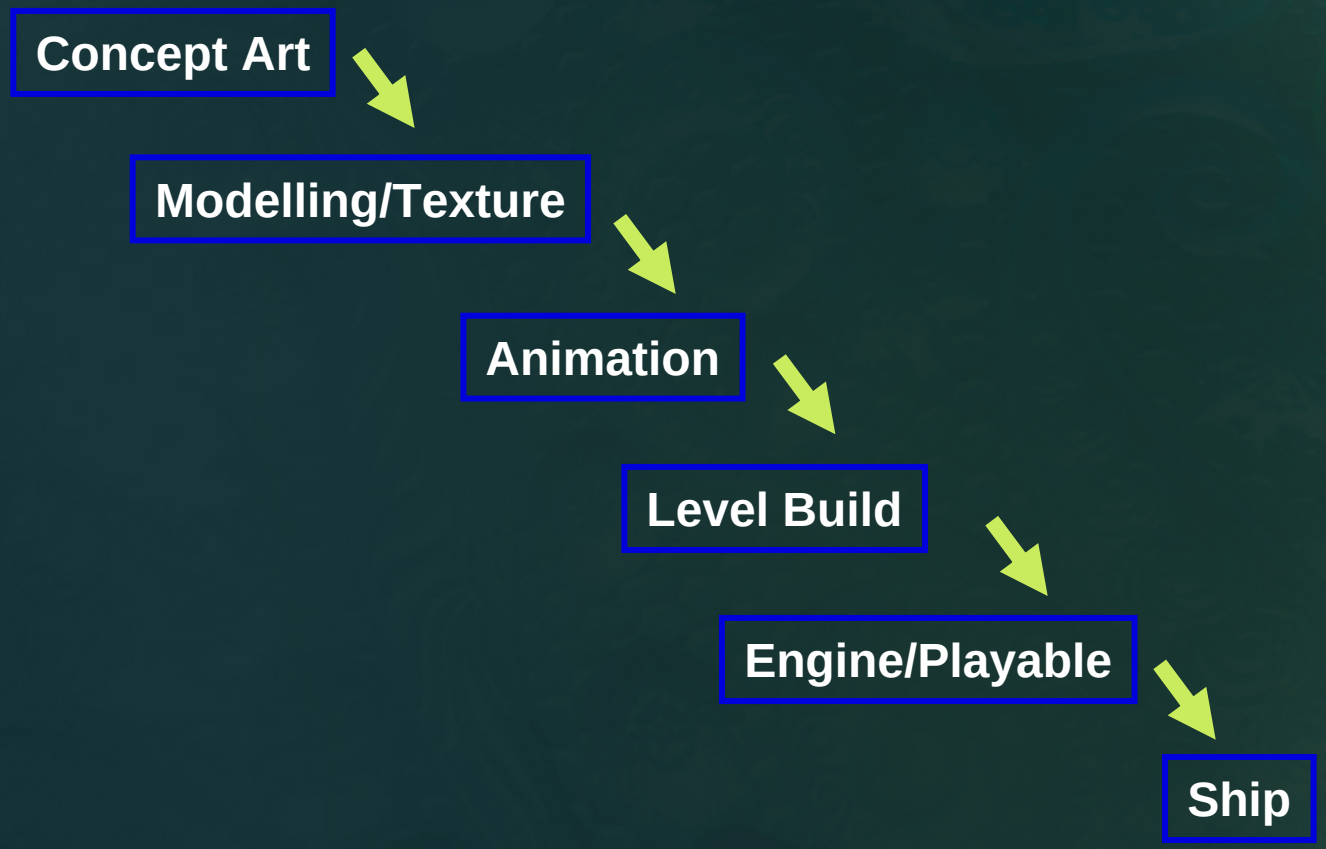
- Movies have lots of money and time, so they have the potential to develop cool technologies
- BUT: Those technologies need to be locked down early enough so that shots delivered on the last day of production look like they belong with shots delivered on the first day of production
- Fastest turnaround in innovation: TV Commercials



Nike.Com campaign, Weiden + Kennedy, Dir Neill Blomkamp http://www.theembassyvfx.com/

Not-Atypical Production

- Waterfall



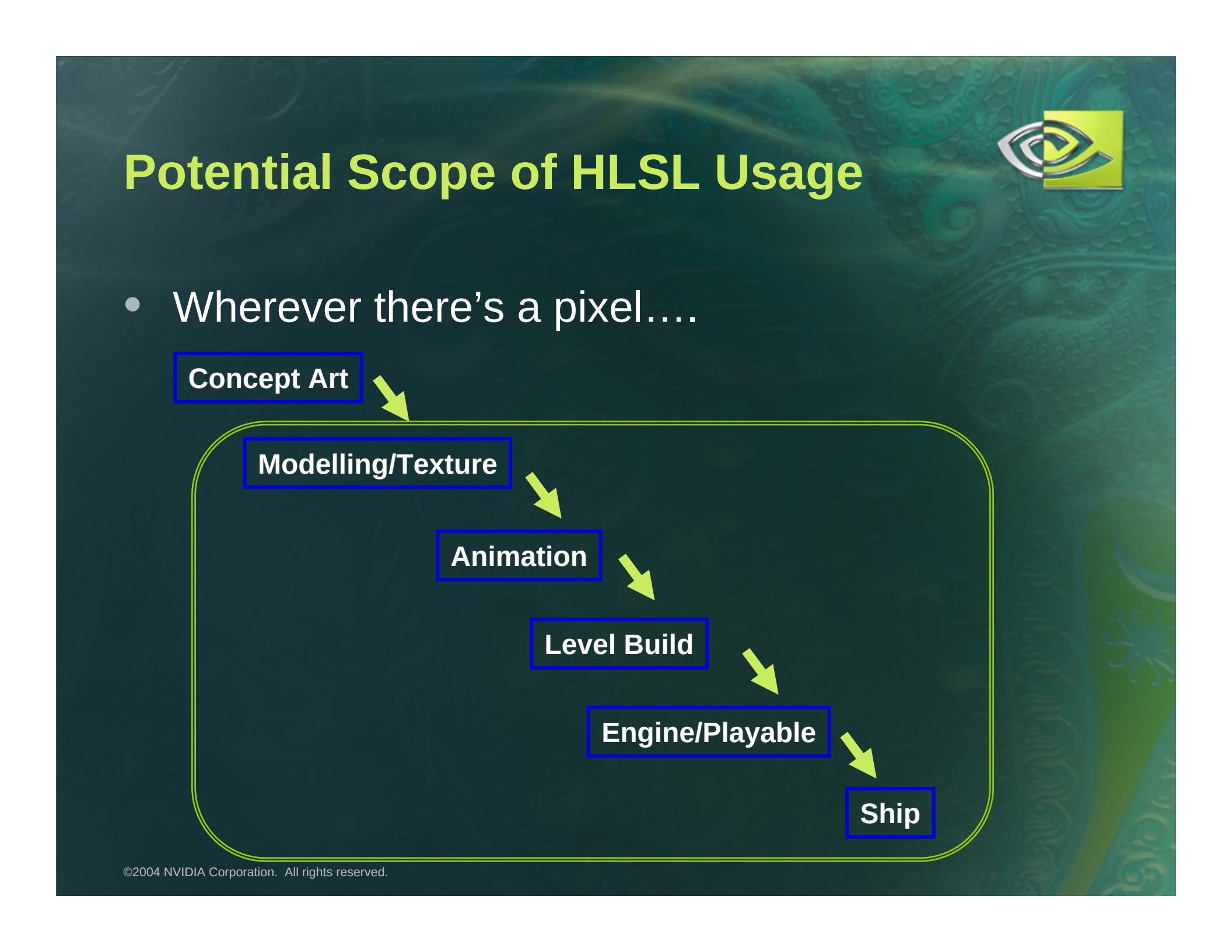
Artists Working Blind

- Often they can only guess at what the final playable engine will look like
- Can’t build/try ideas for rendering alternatives by themselves
- Safest bet: lowest-common-denominator assets

Mid-Production Changes

- Shader changes may come late – art staff might be gone (or certainly busy on something else) by the time new ideas come up or new hardware features are available
- Crossing those gaps is difficult and expensive (Tools like Melody are also aimed at this problem)
- We want formats that let us move forward *and backward* through the production chain, easily

Potential Scope of HLSL Usage



- Wherever there's a pixel….

Sharing FX Files Helps Close Gaps

- Having a single common shading representation lets artists see real shaders on their models
- Changes made late in production can still be art-previewed
- FX Programmability gives the capacity for it to genuinely run the *same* shader at all different levels of production

Layers of HLSL Programmability

- HLSL Pixel and Vertex shaders
- HLSL VM functions for the CPU
- HLSL VM Texture functions (“texture shaders”)
- HLSL DXSAS scripting for render-loop control
- HLSL DXSAS scripting for layering
- HLSL technique validation for varying GPUs
- In FX Composer:
  - Windows Common Language Runtime (CLR)
    - C#
    - VB.NET
  - FX Composer SDK

Only in DirectX FX:

- HLSL Pixel and Vertex shaders
- HLSL VM functions for the CPU
- HLSL VM Texture functions (“texture shaders”)
- HLSL DXSAS scripting for render-loop control
- HLSL DXSAS scripting for layering
- HLSL technique validation for varying GPUs
- In FX Composer:
  - Windows Common Language Runtime (CLR)
    - C#
    - VB.NET
  - FX Composer SDK

Only in FX Composer so far:

- HLSL Pixel and Vertex shaders
- HLSL VM functions for the CPU
- HLSL VM Texture functions (“texture shaders”)
- HLSL DXSAS scripting for render-loop control
- HLSL DXSAS scripting for layering
- HLSL technique validation for varying GPUs
- In FX Composer:
  - Windows Common Language Runtime (CLR)
    - C#
    - VB.NET
  - FX Composer SDK

Innermost Layer: HLSL Pixel and Vertex Shaders

- Vertex Shader: called for each vertex to define its screen position and potentially other data, which will be passed to the…
- Pixel Shader: called for each individual rendered pixel, it assigns output colors
- Both written in the same unified language: HLSL
- *Not teaching HLSL langauge in this talk: Microsoft is running Introductory and Advanced HLSL workshops in Room 402A. (If you know a little C/Java/Cg/etc, you're fine here)*

FX Files Unify Pixel & Vertex Shaders

- FX files let us store/define both pixel and vertex shaders together, in context with their tweakable controls and resource definitions (textures, lights, etc)
- FX Files define render state flags such as ZCull, AlphaBlend, texture address modes, etc.
- FX Files are the natural fundamental shading unit for HLSL applications

HLSL Virtual Machine Functions

- If the results of HLSL code are constant for an entire frame, we can separate that code and let it be run on the CPU once per frame.
- Written in the *same* HLSL language
- Examples:
  - matrix inversion and creation
  - Radian/degree conversions
  - Precalculated trig for spotlight cone angles

HLSL VM Texture Functions

- We can read a texture from a disk image, *or* generate the texture procedurally on the CPU
- Again, the generator function is written in the same HLSL language
- FXComposer-only variation:
  - We can generate textures on the CPU or GPU, and save them to disk

DXSAS

- DXSAS stands for “DirectX Standard Annotations and Semantics”
- DXSAS is part of DirectX 9.0c and XNA
- Semantics and Annotations can be assigned to global tweakables, global “untweakable” tracked values, to individual render-pass definitions, to render techniques, and to the FX file as a whole.
- DXSAS includes a script for defining complex render loops.

DXSAS Renderloop Scripting

- Scripts are stored in string annotations
- Scripts can be part of passes, techniques, and the global FX scope
- Technique scripts can call pass scripts
- Global scripts can call technique or pass scripts
- DXSAS Script is our window to:
  - Render to Texture (RTT)
  - Multiple Render Targets (MRT)
  - Conditional Rendering
  - Layer of FX effects

Technique Validation

- Happens behind the scenes
- Lets us create a hierarchy of techniques for different platforms
- Lets us create multiple techniques to emulate lower-powered platforms, consoles, etc
- Doesn’t validate texture formats
  - They typically fall back to 8-bit

FX Composer Scripting

- FX Composer is built as a .NET application
- Most controls are .COM assemblies
- Scriptable from *within* FX Composer – no additional program needed
  - C# and VB.NET supported via Common Language Runtime (CLR)
  - Other CLR languages, such as Managed C++ or Jscript, could be made available if someone really needed them
  - *Fast!*
- “Nv_sys” library can be browsed in the OLE Viewer
- Most operations can be done using this API
- Can talk to external DCC apps, SQL DBs, etc

FXComposer Plugin SDK

- Plugins are written externally in an environment like Visual Studio
- Compiled and then installed in the Plugin directory
- FX Composer's SDK is mainly designed for object import and material export (the CLR API is also good at export)

FX Files

What's Really Inside

FX File Anatomy Lesson

- FX files are text program files
- There must be at least one technique with at least one pass
- Everything else is optional (though usually needed)
- Order is flexible, like C++ etc

Global Variables

- **Constants**
- **User Tweakables**
- **“Untweakables”**
- **Texture Declarations**

Functions

- **CPU Functions**
- **Vertex Shaders**
- **Pixel Shaders**

Techniques

- **Passes**

FX Global Variables

- Controls for the shader(s) are declared as globals
- They will be either user controls in the parameters panel, or automatically assigned by FX Composer

Global Variables

- **Constants**
- **User Tweakables**
- **“Untweakables”**
- **Texture Declarations**

Functions

- **CPU Functions**
- **Vertex Shaders**
- **Pixel Shaders**

Techniques

- **Passes**

Semantics

- Variables often have an attached SEMANTIC:
  ```
  float3 SurfColor : DIFFUSE = {1,1,1};
  ```
- Semantics hint to the application that this variable maps to a standard part of the scene graph or application environment.
- DirectX scene values such as projection matrices, texture assignments, specularPower, light positions, etc can be given semantics.
- Some applications-specific semantics include TIME and MOUSE connections.
- Semantics can define GPU register assignments.

Annotations

- Annotations enhance semantics by giving more instance-specific info, such as “position of *which* light?”
- Annotations describe UI details, such as the range of scalars, the sorts of UI widgets to use (if any), and the displayed names in the Parameters panel.
- DXSAS scripts are also stored as annotations (more on these later).

Tweakables and Un-Tweakables

- User-accessible controls are tweakable:

```
float3 SurfColor : Diffuse <
    string UIName = "Surface";
    string UIWidget = "Color";
> = {1.0f, 0.7f, 0.3f};
```

- Automatically-assigned values are un-tweakable, so we hide their display:

```
float4x4 ViewProjXf : ViewProjection <
    string UIWidget="None";
>;
```

- Note that untweakables can ignore their default values.

FX Functions

- HLSL functions can be executed on vertex shader nit, pixel shader unit, or the CPU, depending on context

Global Variables

- Constants
- User Tweakables
- “Untweakables”
- Texture Declarations

Functions

- CPU Functions
- Vertex Shaders
- Pixel Shaders

Techniques

- Passes

Function Semantics

- Semantics attached to a function's returned values define the type of function
- If a function returns ": HPOSITION" it's a vertex shader
- If a function returns ": COLOR" it's a pixel shader (or texture function -- next slide)
- All other functions are general-use

Texture Functions

- Textures can be initialized by DirectX on the CPU at shader-load time by texture functions
- We write a function that returns a ": COLOR" value based on an input ": POSITION" value (1D, 2D, 3D)
- We assign the function to the texture declaration via a "function" annotation
- More on uses for this later….

FX Techniques

- A technique bundles-up a complete set of shading instructions, and may define one or many passes
- Passes can be executed in default order, or explicitly scripted

Global Variables

- Constants
- User Tweakables
- “Untweakables”
- Texture Declarations

Functions

- CPU Functions
- Vertex Shaders
- Pixel Shaders

Techniques

- Passes

Multiple Techniques

- Some FX files may have multiple techniques for the same appearance
- Reasons include:
  - HW sensitivity – an FX file might contain techniques that can only run on high-end hardware, so additional “fallback” techniques are included
  - Alternate emulations – an FX file might contain versions of an appearance for high-end PCs, low-end PCs, and multiple game consoles. All can be compared “live” and can be driven/tested from one set of art assets during development

FX Passes

- Each pass can define vertex and pixel shaders and their shading-model profiles; render states such as alpha blend; and can optionally aim at any render target(s) – textures or the framebuffer

Global Variables

- Constants
- User Tweakables
- “Untweakables”
- Texture Declarations

Functions

- CPU Functions
- Vertex Shaders
- Pixel Shaders

Techniques

- **Passes**

Technique and Pass Annotations

- Techniques and Passes can be annotated, mostly to allow the presence of DXSAS script commands

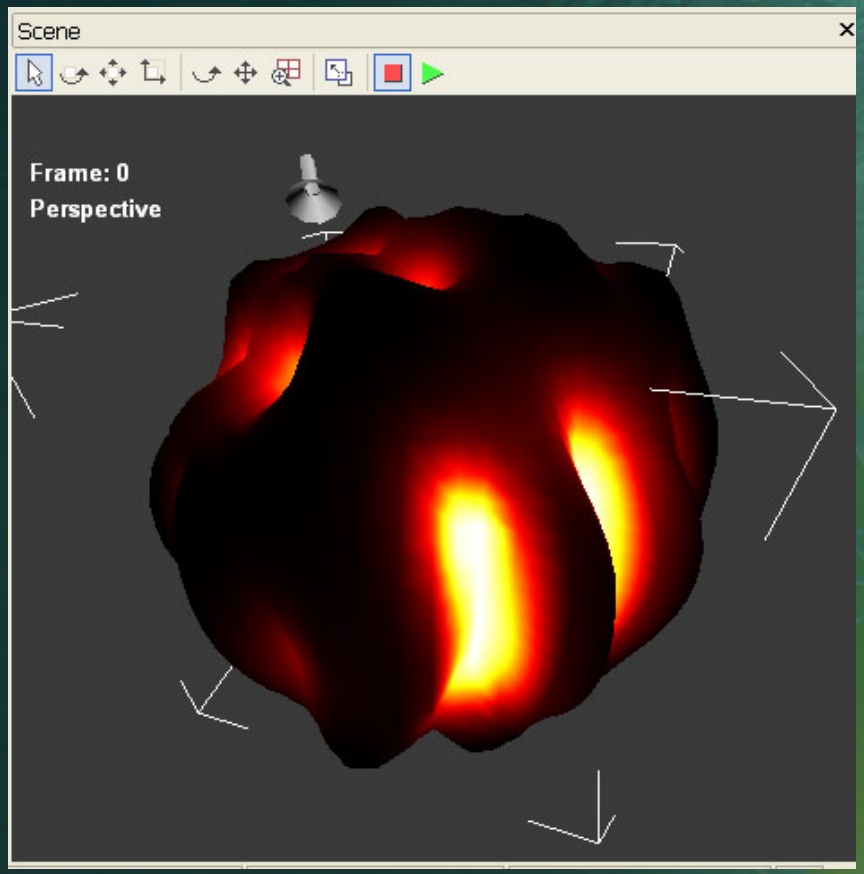
```
technique hotTech <
    string Script = "Pass=p0;";
> {
    pass p0 <
            string Script = "Draw=geometry;";
    > {
            VertexShader = compile vs_2_0 mainVS();
            ZEnable = true;
            ZWriteEnable = true;
            CullMode = None;
            AlphaBlendEnable = false;
            PixelShader = compile ps_2_a hotPS();
    }
}
```

- (In this case, this is the default – annotations *could* have been skipped)


Examples
Yeah!

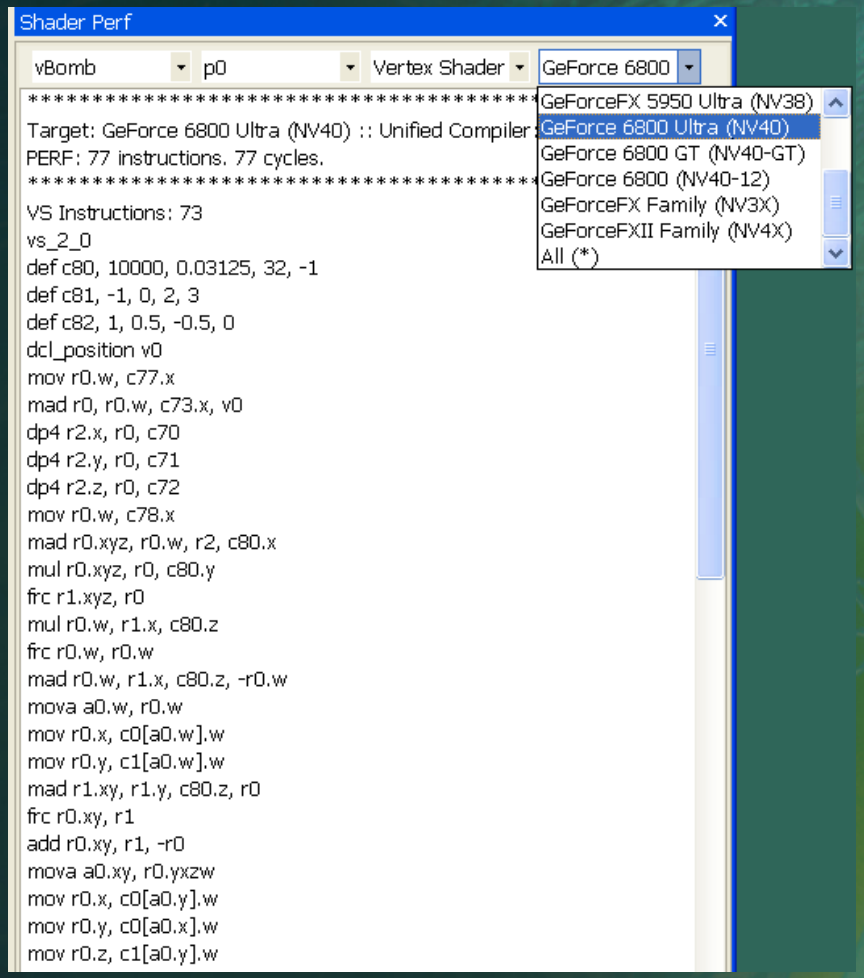
Shader Demo: VBomb

- Shows:
- General HLSL functions
- Vertex Shader
- Texture Declaration
- TIME semantic and vertex animation
- Vertex Perf Panel
- Pixel Perf Panel
- vbomb.fxcomposer



VBomb: Looking at the Perf Window

- FX Composer's Shader Perf window lets us see the exact running ASM code as compiled by DirectX
- Selectable by specific pass: vertex or pixel shaders
- Selectable by GPU model



Vertex Animation and TIME

- The simple TIME sematic on a global variable lets us update continuously:

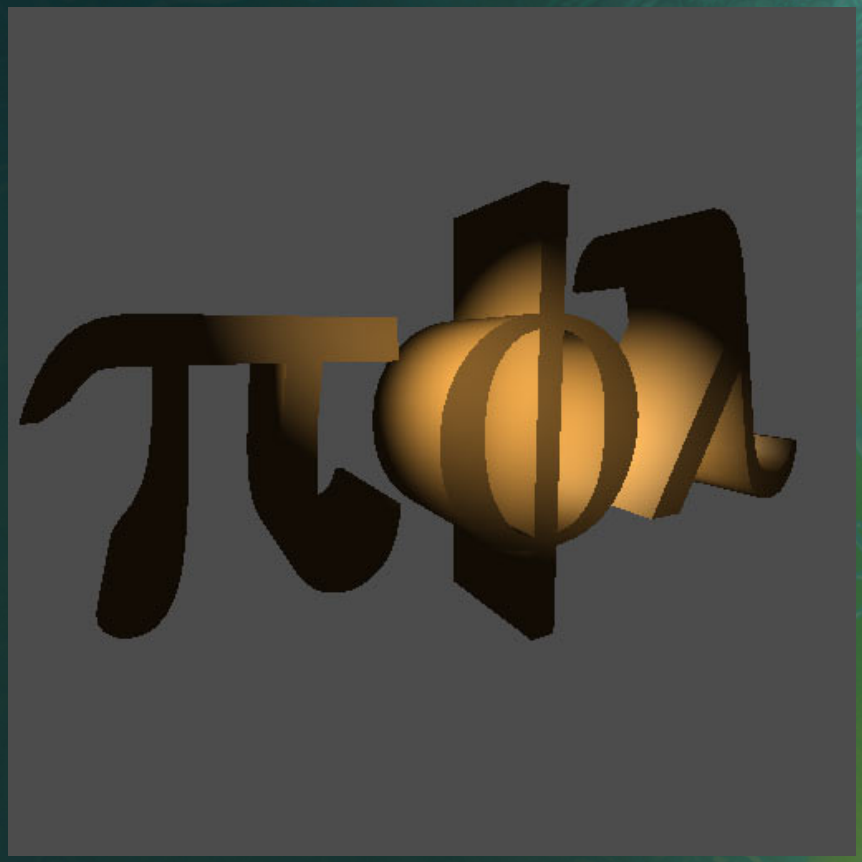
```
float Timer : TIME < string UIWidget="none";>;
```

- We can then use it in the vertex or pixel shaders (or anywhere) just like any other variable

- *TIP: The existence of a TIME variable causes FX Composer to be redrawing constantly – be aware of it!*

Shader Demo: Spot-Early

- Shows:
- Shader Model Profiles
- Technique Selection
- VM functions for CPU



Spot-Early: Profile Selection

```
technique PS3 <
    string Script = "Pass=drawPass;";
> {
    pass drawPass < string Script = "Draw=geometry;"; > {
        VertexShader = compile vs_3_0 mainCamVS();
        ZEnable = true;
        ZWriteEnable = true;
        CullMode = None;
        PixelShader = compile ps_3_0 spotLightPS(
            cos(SpotLightCone*(float)(3.141592/180.0)));
    }
}

technique PS2 <
    string Script = "Pass=drawPass;";
> {
    pass drawPass < string Script = "Draw=geometry;"; > {
        VertexShader = compile vs_2_0 mainCamVS();
        ZEnable = true;
        ZWriteEnable = true;
        CullMode = None;
        PixelShader = compile ps_2_a spotLightPS(
            cos(SpotLightCone*(float)(3.141592/180.0)));
    }
}
```

Spot-Early: Technique Selection

- Technique selection is via DXSAS script
- ": STANDARDSGLOBAL" variable should appear before techniques begin

```
float Script : STANDARDSGLOBAL <
   string UIWidget = "none";
   string ScriptClass = "object";
   string ScriptOrder = "standard";
   string ScriptOutput = "color";
   string Script = "Technique=Technique?PS3:PS2;";
> = 0.8; // DXSAS version #
```

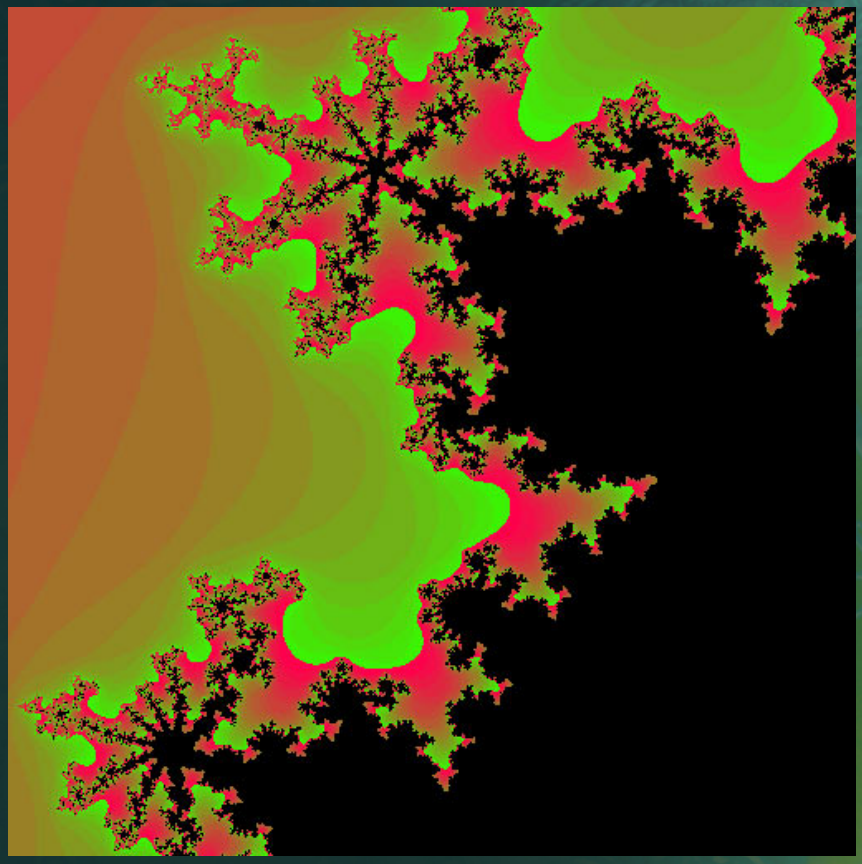
- In the simplest object-shader cases, we can sometimes skip this step – FX Composer will figure it out and "fill in the blanks."

Spot-Early: CPU-Side (VM) Math

```
technique PS3 <
   string Script = "Pass=drawPass;";
> {
   pass drawPass < string Script = "Draw=geometry;";
   > {
      VertexShader = compile vs_3_0 mainCamVS();
      ZEnable = true;
      ZWriteEnable = true;
      CullMode = None;
      PixelShader = compile ps_3_0 spotLightPS(
                           cos(radians(SpotLightCone));
   }
}
```

Shader Demo: Mandelbrot PS_3

- Shows:
- Full-Screen Effect
- Pixel-Shader Looping



Mandelbrot: Full-Screen Drawing

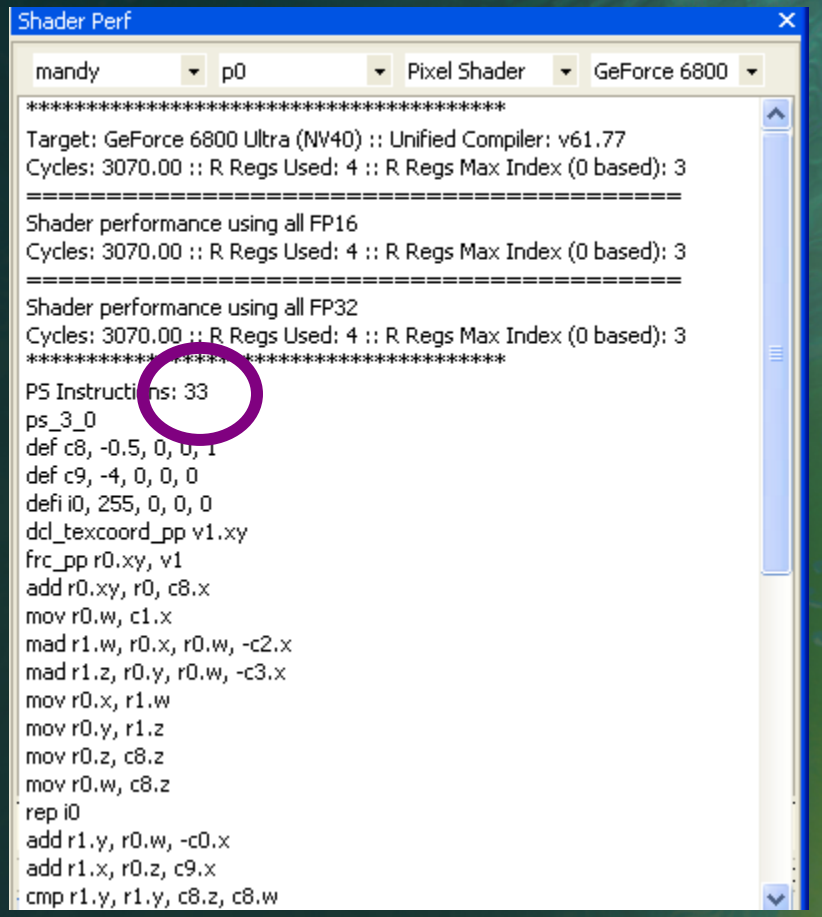
- Built-in full-screen Quad

```
#include "Quad.fxh“

technique mandy <
   string Script = "Pass=p0;";
> {
    pass p0 <
            string Script = "Draw=Buffer;";
    > {
            VertexShader = compile vs_3_0 ScreenQuadVS();
            cullmode = none;
            ZEnable = false;
            ZWriteEnable = false;
            AlphaBlendEnable = false;
            PixelShader = compile ps_3_0 mandyPS();
     }
}
```

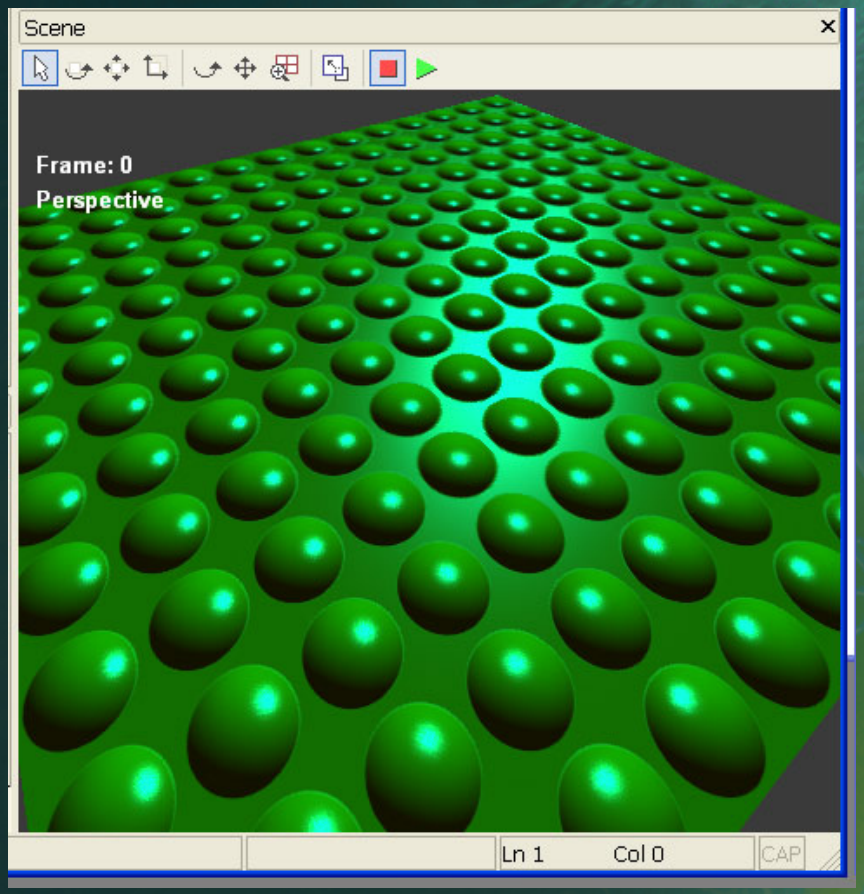
Mandelbrot: Pixel Looping

- Big difference in SM3 versus SM2 program size – SM3 version has dynamic # of loops
- SM2 version (with constant # of loops) was hundreds of PS instructions…
- More in the session: ***Shader Model 3.0 Unleashed***



Shader Demo: Toksvig-split

- Shows:
- VM Texture Generation
- Using pixel shading to get “look” right, and texture for speedy AA on final result
- *This technique described in depth on NVIDIA.COM (http://developer.nvidia.com/object/mipmapping_normal_maps.html) “Mipmapping Normal Maps”*



Toksvig: VM Texture Function

```
float spec_func(
    float s,
    float NaH,
    float NaNa
) {
     float toksvig = sqrt(NaNa)/(sqrt(NaNa)+
                                 s*(1-sqrt(NaNa)));
     return (1.0+toksvig*s)/(1.0+s) *
                 pow(NaH/sqrt(NaNa), toksvig*s);
}

float4 make_specular_tex(
    float2 Pos : POSITION
) : COLOR {
    float f  = spec_func(SPEC_EXPON,Pos.x,Pos.y);
    return float4(f.xxx,0.0);
}
```

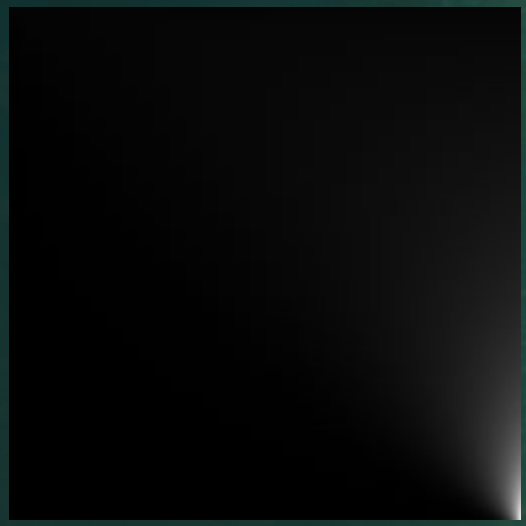
Toksvig: Assigning Texture Function

```
texture SpecTex <
    string function = "make_specular_tex";
    int width = TOX_TABLE_SIZE;
    int height = TOX_TABLE_SIZE;
   string UIWidget = "None";
   string format = “g16r16”;
>;

sampler SpecSampler = sampler_state
{
    texture = <SpecTex>;
    AddressU  = CLAMP;
    AddressV  = CLAMP;
    MIPFILTER = NONE;
    MINFILTER = ANISOTROPIC;
    MAGFILTER = ANISOTROPIC;
};
```

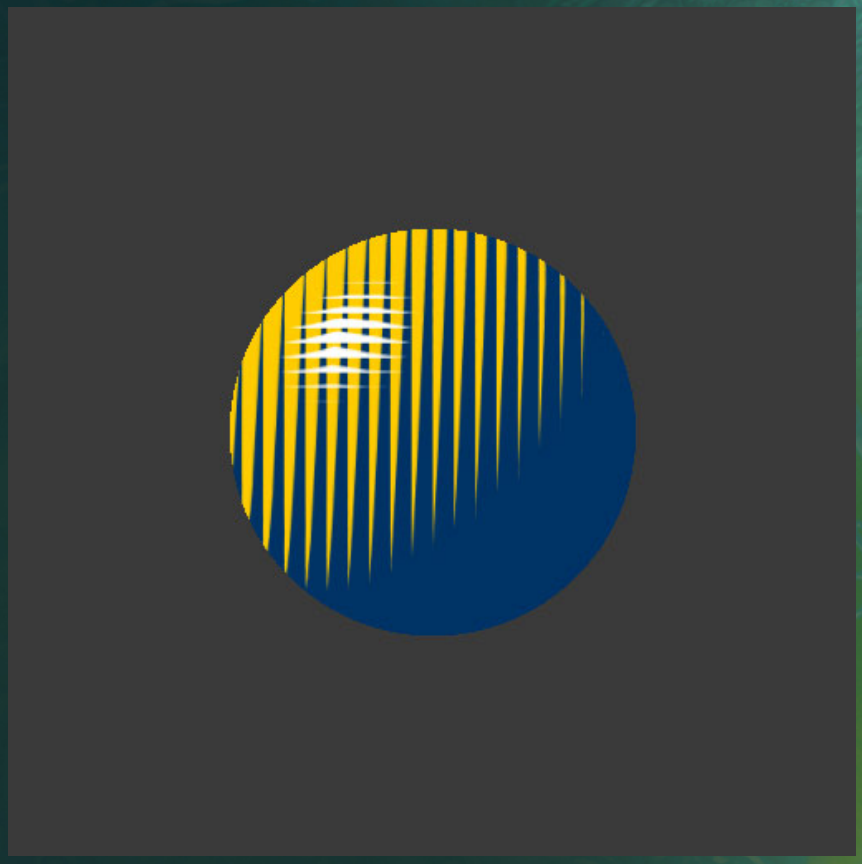
Toksvig: Saving Textures to Disk

- Any loaded texture can be saved to disk
- More-efficient if multiple shaders will be generating the same texture
- We can also save GPU render targets



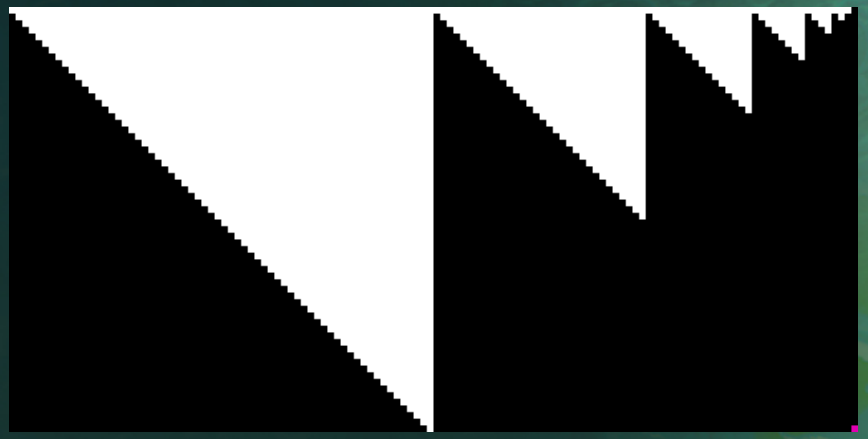
Shader Demo: Durer

- Shows:
- Texture Generation with texture derivatives
- Fast anti-aliasing via the texture engine (follow-on to Toksvig)
- NPR render technique describe in *GPU Gems*



Durer: Getting Texel Derivatives

- The previous example let the system generate MIP maps
- “Durer” makes custom MIP levels:



```
half4 stripe_function(
        half2 Pos : POSITION,
        half2 ps : PSIZE
) : COLOR
{
   half v = 0;
   half nx = Pos.x+ps.x; // keep the last column full-on, always
   v = nx > Pos.y;
   return half4(v.xxxx);
}
```

Shader Demo: Paint_Brush

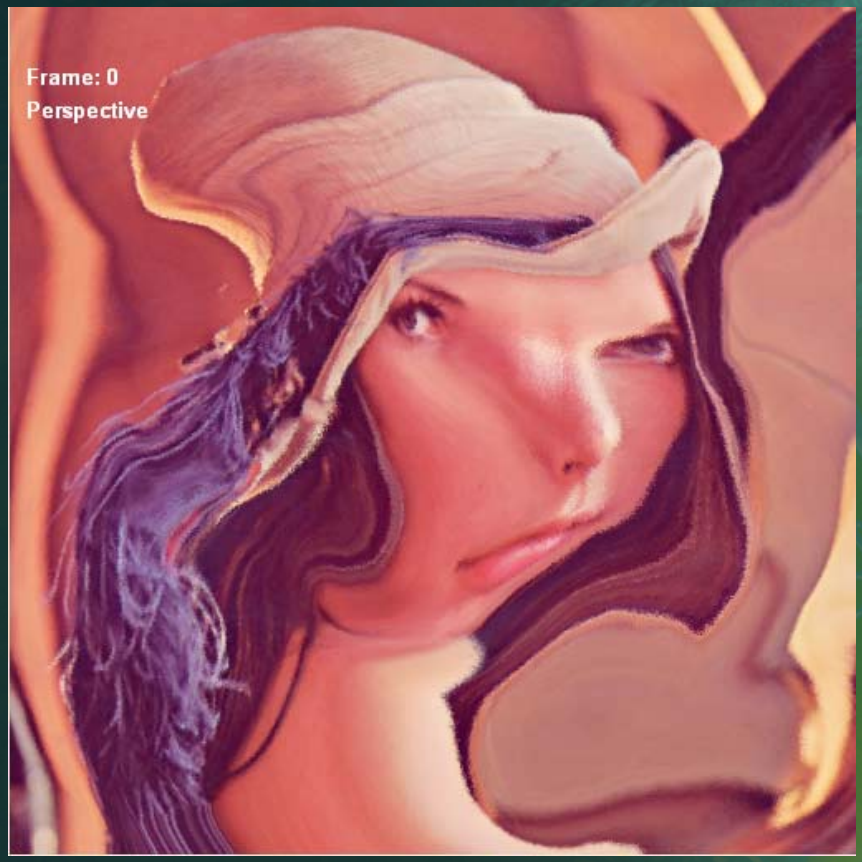
- Shows:
- DXSAS bool/loops and ScriptClass
- Interactivity:
- MOUSEPOSITION
- LEFTMOUSEDOWN
- TIMER
- FXCOMPOSER_RESET PULSE
- Using a shader instead of “clear”

Paint_brush: Pixel-shader clear

```
technique paint <
     string Script = "RenderColorTarget0=;“
                     "RenderDepthStencilTarget=;"
                     "LoopByCount=bReset;"
                              "Pass=revert;"
                     "LoopEnd=;"
                     "Pass=splat;";
> {
     pass revert <
            string Script = "Draw=Buffer;";
     > {
            VertexShader = compile vs_1_1 ScreenQuadVS();
            PixelShader  = compile ps_2_0 revertPS(true);
            AlphaBlendEnable = false;
            ZEnable = false;
     }
     pass splat <
            string Script = "Draw=Buffer;";
     > {
            VertexShader = compile vs_1_1 ScreenQuadVS();
            PixelShader  = compile ps_2_0 strokePS(fadeout(),lerpsize());
            AlphaBlendEnable = true;
            SrcBlend = SrcAlpha;
            DestBlend = InvSrcAlpha;
            ZEnable = false;
     }
}
```

Shader Demo: Liquid

- Shows:
- DXSAS RenderTarget assignments
- FP16 Blending
- MOUSE events
- Render to texture
- VM math for efficiency



Paint_brush: Reset Events

- We can set a bool value to be triggered by “reset” events (explicitly toggled in the Parameters pane, or from reload/resize events)
- We can use this to conditional invoke CLEAR events or special passes in DXSAS scripts

```
bool bReset : FXCOMPOSER_RESETPULSE
<
   string UIName="Clear Canvas";
>;
```

Liquid: Defining a Render Target

- We use a “Quad.fxh” macro:
  `DECLARE_QUAD_TEX(PaintTex,PaintSamp,"A16B16G16R16F")`
- Which expands to:

```
texture PaintTex : RENDERCOLORTARGET <
    float2 ViewPortRatio = {1.0,1.0}; // screen-sized
    int MipLevels = 1;
    string Format = "A16B16G16R16F";
   string UIWidget = "None";
>;
sampler PaintSamp = sampler_state {
    texture = <PaintTex>;
    AddressU  = CLAMP;
    AddressV = CLAMP;
    MipFilter = POINT;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
};
```

- This viewport-sized, OpenEXR-style floating-point image is blendable and filterable on GeForce 6 and Quadro 4K+

Paint_Liquid: Target Assignment

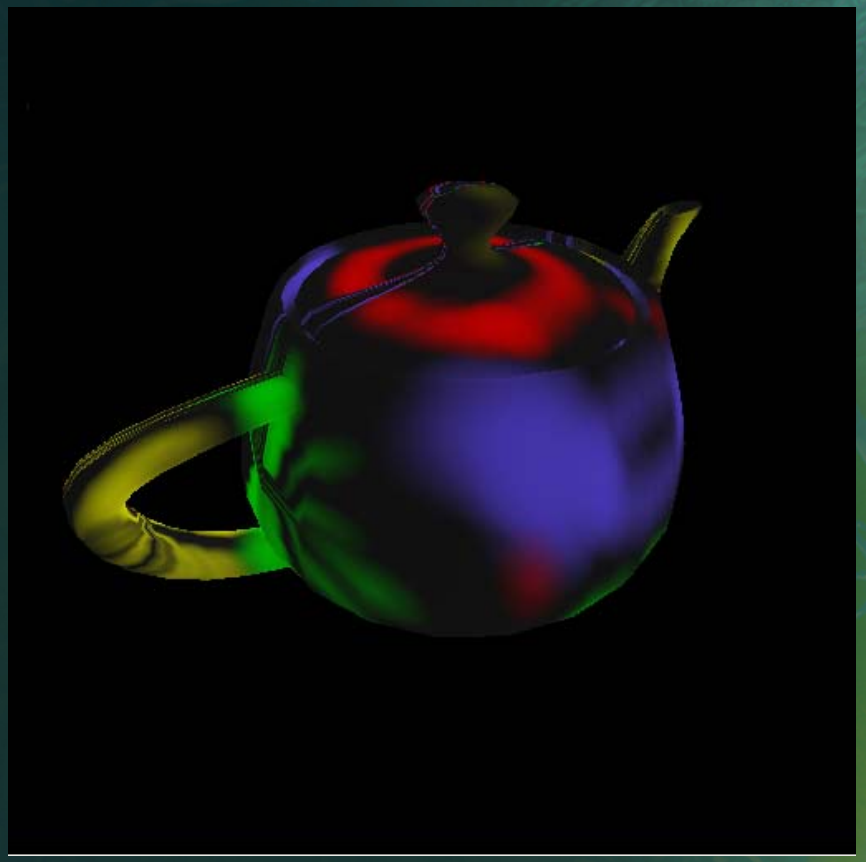
```
technique liquid <
      string Script =
              "Pass=paint;"
              "Pass=display;";
> {
      pass paint <
              string Script = "RenderColorTarget0=PaintTex;"
                              "RenderDepthStencilTarget=;"
                              "LoopByCount=bReset;"
                                            "ClearSetColor=ClearColor;"
                                            "Clear=Color0;"
                              "LoopEnd=;"
                              "Draw=Buffer;";
      > {
              VertexShader = compile vs_1_1 ScreenQuadVS();
              PixelShader  = compile ps_2_0 strokePS(dir_color(),fadeout(),
                                                     fadein(),lerpsize());
              AlphaBlendEnable = true;
              SrcBlend = SrcAlpha;
              DestBlend = InvSrcAlpha;
              ZEnable = false;
      }
      pass display <
              string Script = "RenderColorTarget0=;“ // no target
                              "RenderDepthStencilTarget=;"
                              "Draw=Buffer;";
      > {
              VertexShader = compile vs_1_1 ScreenQuadVS();
              PixelShader  = compile ps_2_0 liquidPS();
              AlphaBlendEnable = false;
              ZEnable = false;
      }
}
```

Paint_Liquid: CPU VM Math

```
technique liquid <
      string Script =
              "Pass=paint;"
              "Pass=display;";
> {
      pass paint <
              string Script = "RenderColorTarget0=PaintTex;“
                              "RenderDepthStencilTarget=;"
                              "LoopByCount=bReset;"
                                              "ClearSetColor=ClearColor;"
                                              "Clear=Color0;"
                              "LoopEnd=;"
                              "Draw=Buffer;";
      > {
              VertexShader = compile vs_1_1 ScreenQuadVS();
              PixelShader  = compile ps_2_0 strokePS(dir_color(), fadeout(),
                                                     fadein(),lerpsize());
              AlphaBlendEnable = true;
              SrcBlend = SrcAlpha;
              DestBlend = InvSrcAlpha;
              ZEnable = false;
      }
      pass display <
              string Script = "RenderColorTarget0=;"
                              "RenderDepthStencilTarget=;"
                              "Draw=Buffer;";
      > {
              VertexShader = compile vs_1_1 ScreenQuadVS();
              PixelShader  = compile ps_2_0 liquidPS();
              AlphaBlendEnable = false;
              ZEnable = false;
      }
}
```

Shader Demo: 3D Paint

- Shows:
- Multipass
- FP blending
- “dependant read”

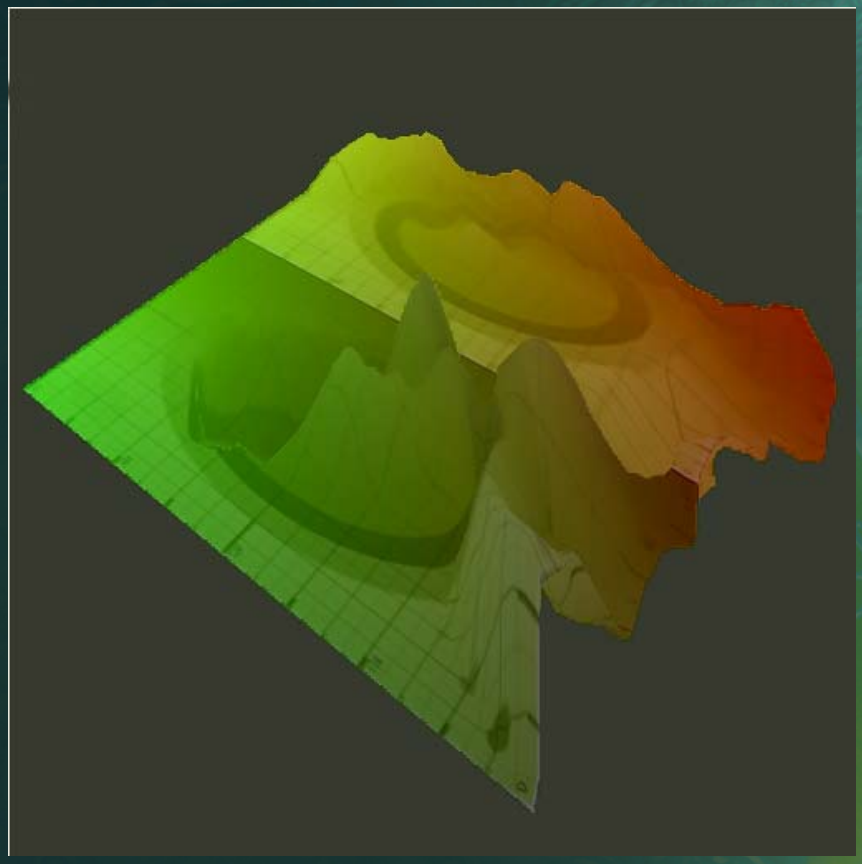


3D Paint – Mixing 3D and 2D Passes

```
pass ShowUV <
      string Script = "RenderCoLorTarget0=UVMap;"
                      "RenderDepthStencilTarget=DepthBuffer;"
                      "ClearSetColor=ClearColor;"
                      "ClearSetDepth=ClearDepth;"
                      "Clear=Color;"
                      "Clear=Depth;"
                      "Draw=Scene;";
> {
      VertexShader = compile vs_2_0 minVS();
      ZEnable = true;
      ZWriteEnable = true;
      CullMode = None;
      PixelShader = compile ps_2_0 uvPS();
}
pass restorePaint <
      string Script ="RenderColorTarget0=BufMap;"
                      "Draw=Buffer;";
> {
      VertexShader = compile vs_1_1 ScreenQuadVS();
      PixelShader  = compile ps_2_0 restorePaintPS();
      AlphaBlendEnable = false;
      ZEnable = false;
}
```

Shader Demo: Sculpt

- Shows
- FP16 blend
- FP16/FP32 pingpong
- Vertex Texture Fetch
- More painting fun

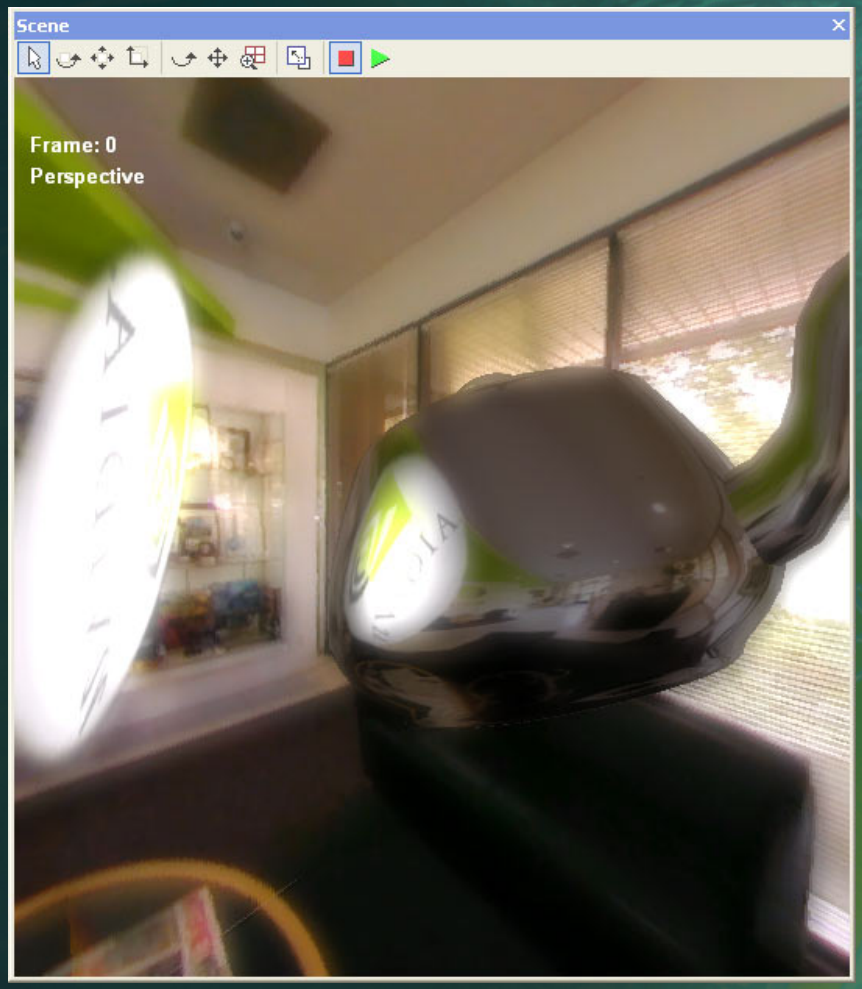


Sculpt: Mixing FP16 and FP32

- FP16 is blendable for painting
- FP32 is needed for Vertex Texture Fetch (VTF)
- So we use utility shaders in Quad.fxh to copy the FP16 PaintSampler to FP32 Displacement Map in a special pass:

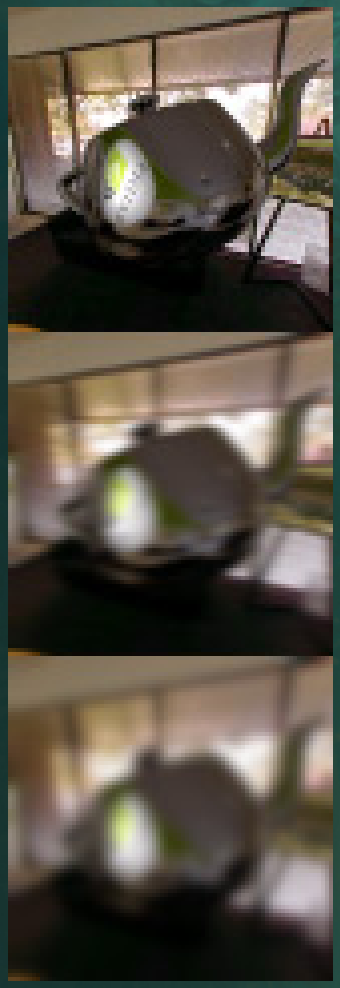
```
    pass boost <
            string Script =
                    "RenderColorTarget0=DisplaceMap;"
                    "Draw=Buffer;";
    > {
            VertexShader = compile vs_3_0
                           ScreenQuadVS();
            ZEnable = false;
            ZWriteEnable = false;
            CullMode = None;
            PixelShader = compile ps_3_0
                           TexQuadPS(PaintStrokeSampler);
    }
```

Shader Demo: Bloom & Reflection



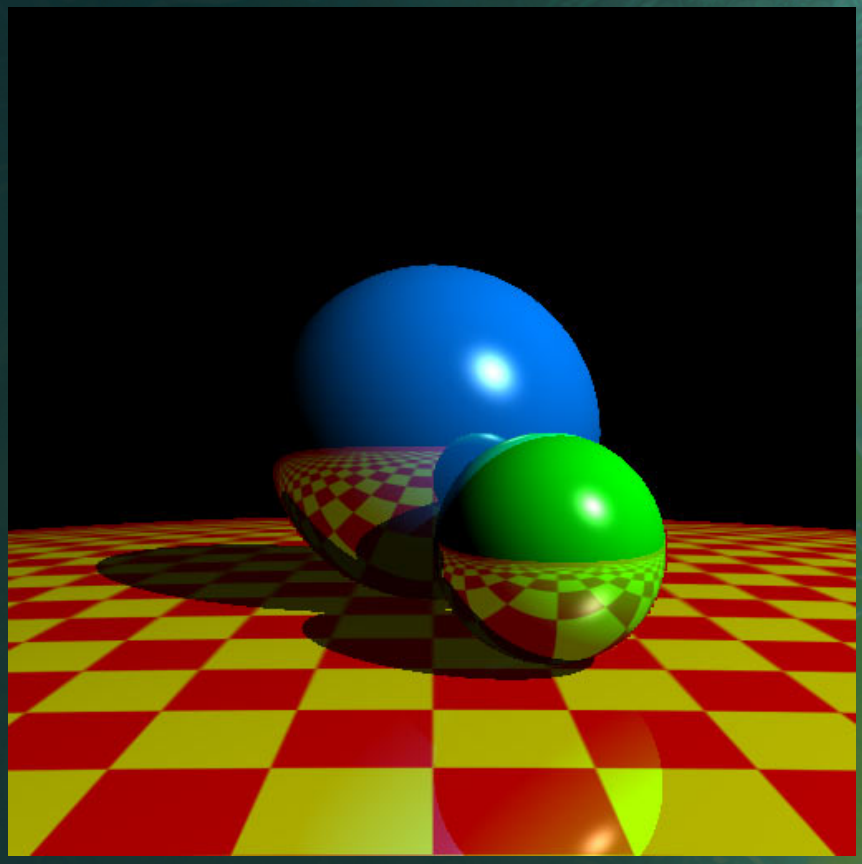
- Shows:
- DXSAS post-process
- VM matrix math
- FP16 imaging
- Separable multipass filtering
- “Fake reflection” technique – restricted one-bounce raytracer

Separable Image Filters



- Done in two passes – blur once in the horizontal direction, then blur *that* vertically
- FP16 pixels let us avoid clamping in the brightest areas

Shader Demo: Raytracer



- Shows:
- Non-poly-intensive use of GPU with “real” raytracer writing to a full-screen buffer
- FX is appropriate for numeic uses other than games

Shader Demo: shadRPort

- Shows:
- Frustum annotation
- Renderport assignment
- FP texture shadow maps
- VM used to pre-calculate spotlight cone values

Assigning a Frustum and RenderPort

- We can re-assign :VIEW and :PROJECTION matrices to a specific spotlamp:

```
float4x4 LampProjXf : Projection <
    //string UIWidget="None";
    string frustum = "light0";
>;
```

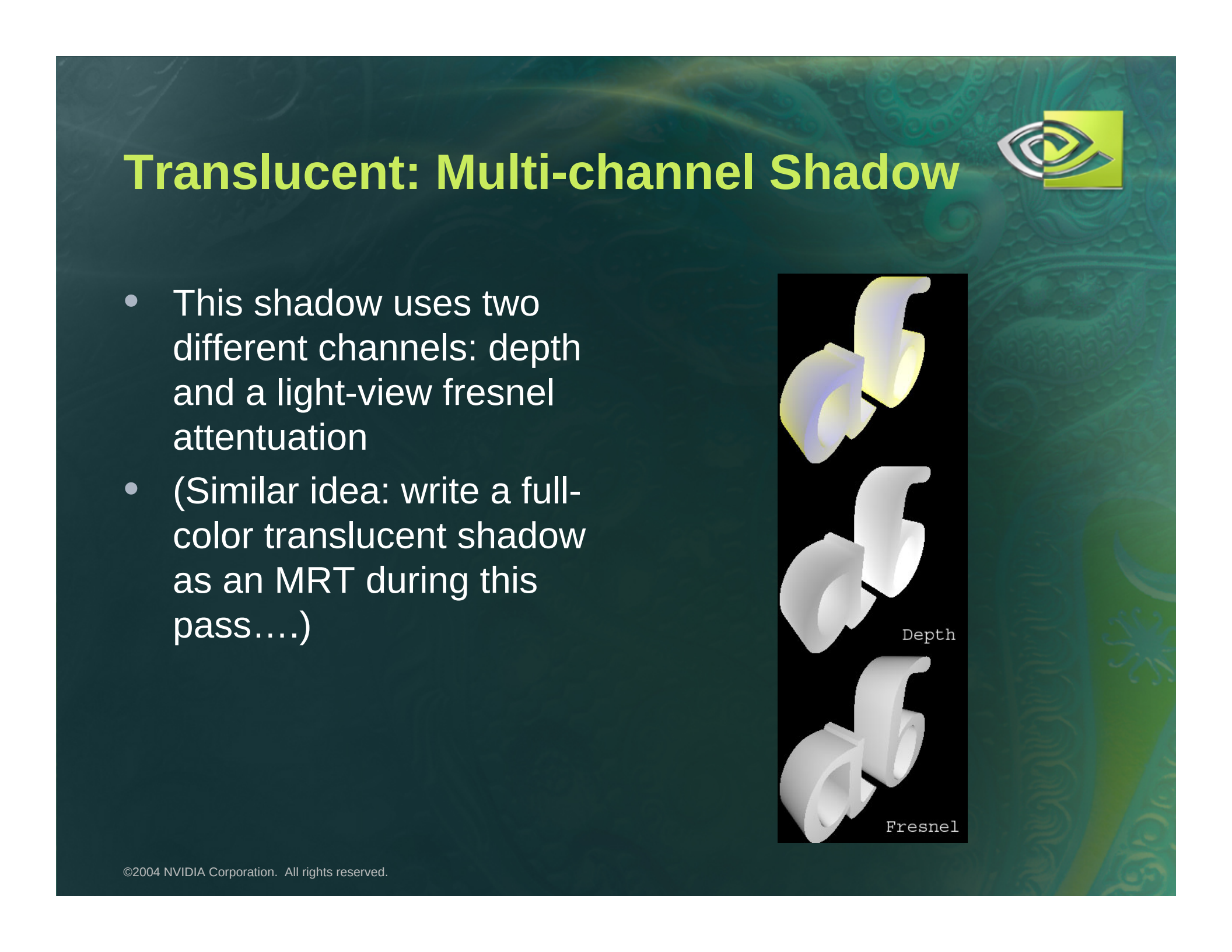
- To adjust for screen aspect, we also define a RenderPort in the DXSAS script for the shadow-generation pass, as if it was a Render Target:

```
"RenderPort=light0;"
```

Shader Demo: Translucent

- Shows:
- Frustum annotation
- Renderport assignment
- FP texture shadow maps
- Simple VM math
- Specialized usage of maps

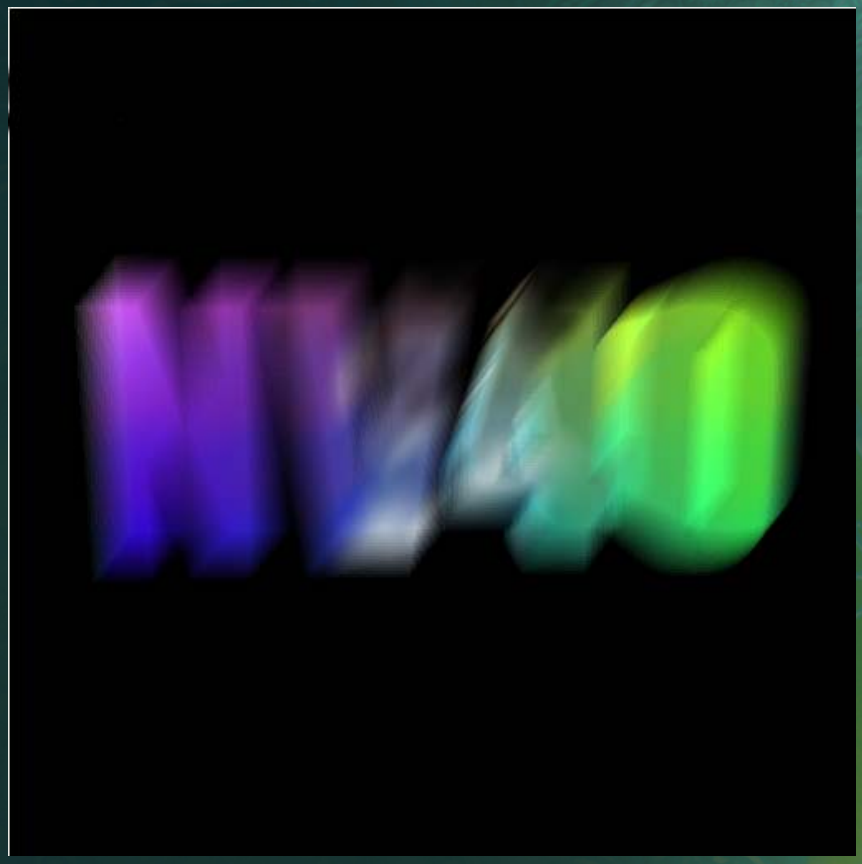
Translucent: Multi-channel Shadow



- This shadow uses two different channels: depth and a light-view fresnel attentuation
- (Similar idea: write a full-color translucent shadow as an MRT during this pass….)

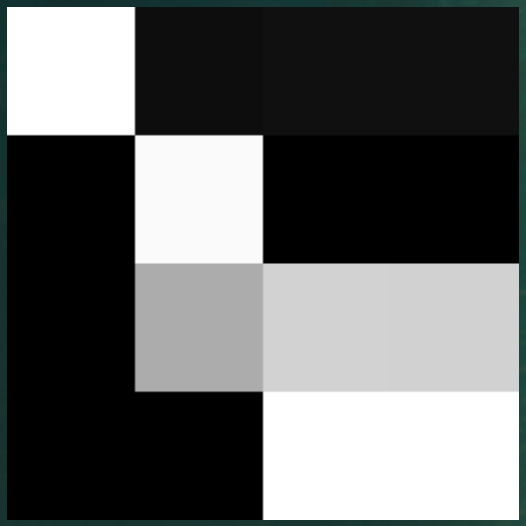
Shader Demo: Motion Blur

- Shows:
- Mouse interception
- Unusual GPGPU-ish use of textures
- FP16 blending provides accumulation-buffer motion blur
- DXSAS lets us dynamically select the # of accumulations



Saving Persistent Data in Texture

- “MoBlur” has a funny little 4x4 1-channel fp 32 texture….
- It’s the previous frame’s projection matrix, saved by a “saveXf” pass
- Shader uses it to interpolate against current-frame projection for motion blur
- (Alternative crazy idea: use blending to mix them…)



Motion Blur: DXSAS Looping

- Given a global tweakable scalar “npasses,” and a (hidden) untweakable scalar called “passnumber”:

```
technique Blur <
    string Script =
            "LoopByCount=bReset;"
                    "Pass=resetXf;"
            "LoopEnd=;"
            // Clear Accum Buffer
            "RenderColorTarget0=AccumBuffer;"
            "ClearSetColor=ClearColor;"
            "Clear=Color;"
            "LoopByCount=npasses;"
                    "LoopGetIndex=passnumber;"
                    // Render Object(s)
                    "Pass=drawObj;"
                    // Blend Results into accum buffer
                    "Pass=Accumulate;"
          "LoopEnd;"
          // draw accum buffer to framebuffer
          "Pass=saveXf;"
          "Pass=FinalPass;";
```

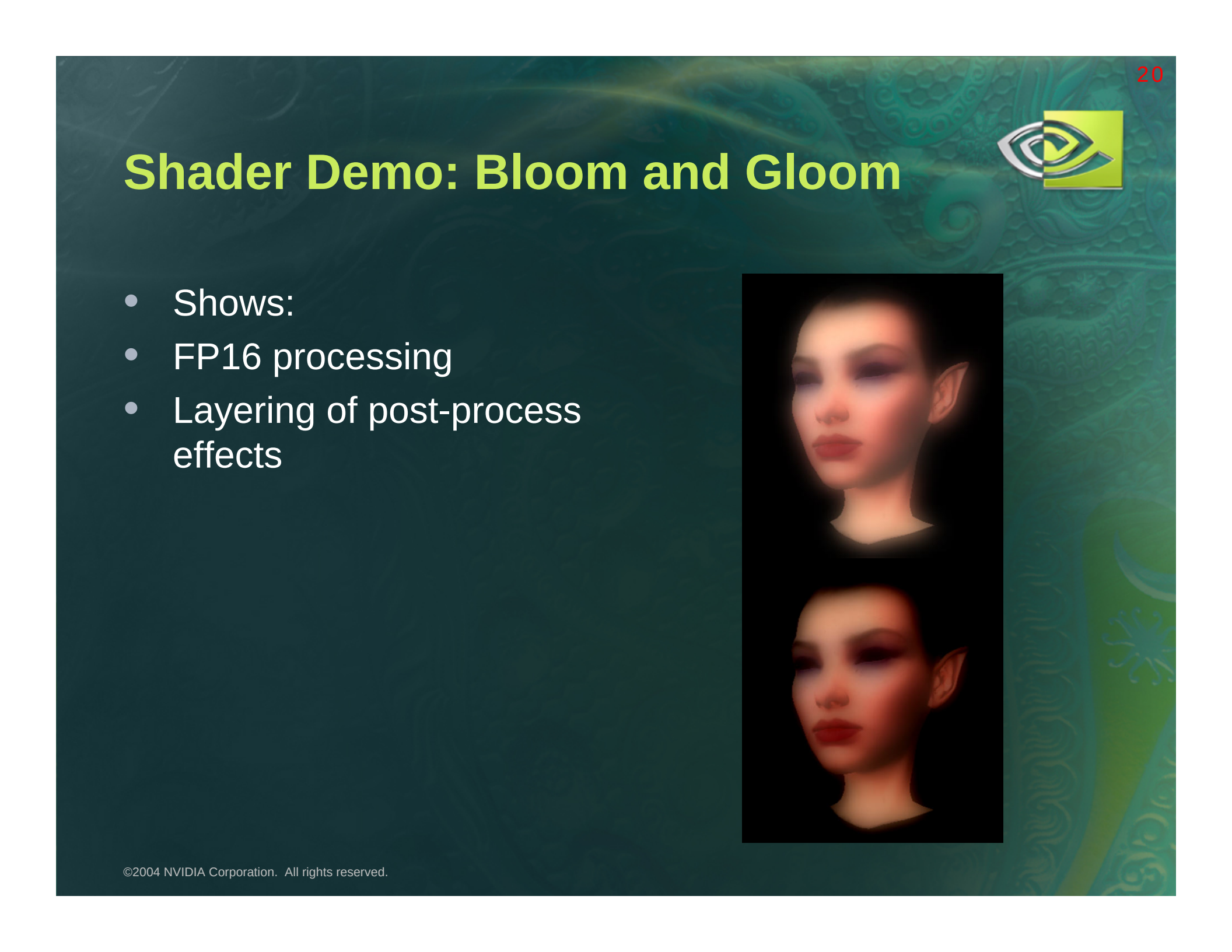
Shader Demo: Parallax Bump

- Shows:
- It’s simple and fast!

“Shading Sketchbook” in Action…

- New demo shader, literally written during the break between the first two classes here at Siggraph on Monday afternoon

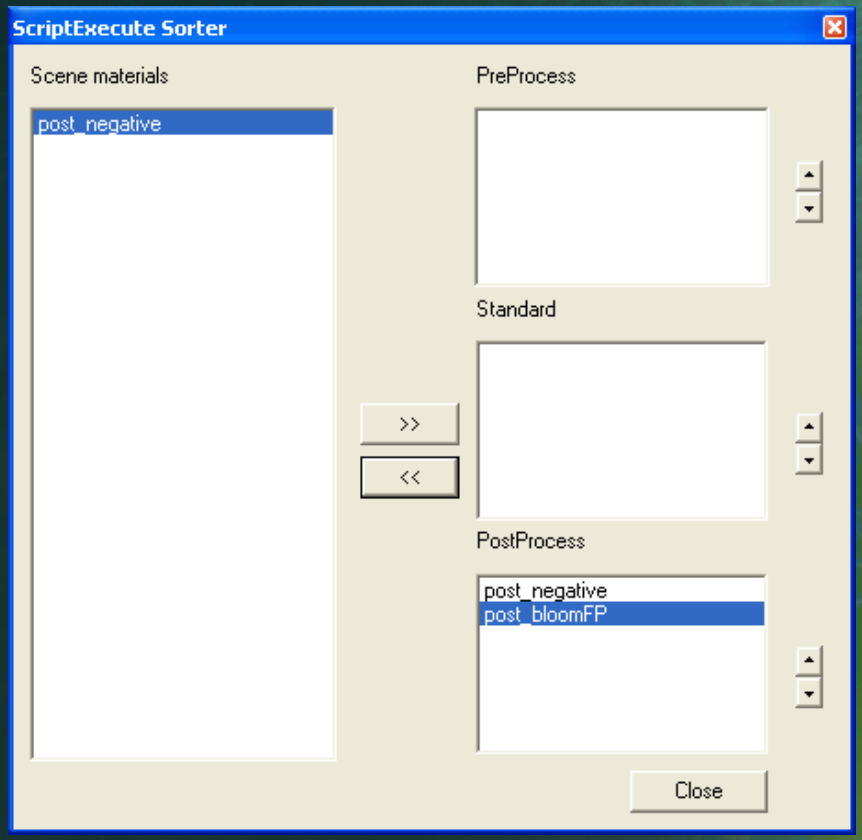
Shader Demo: Bloom and Gloom



- Shows:
- FP16 processing
- Layering of post-process effects

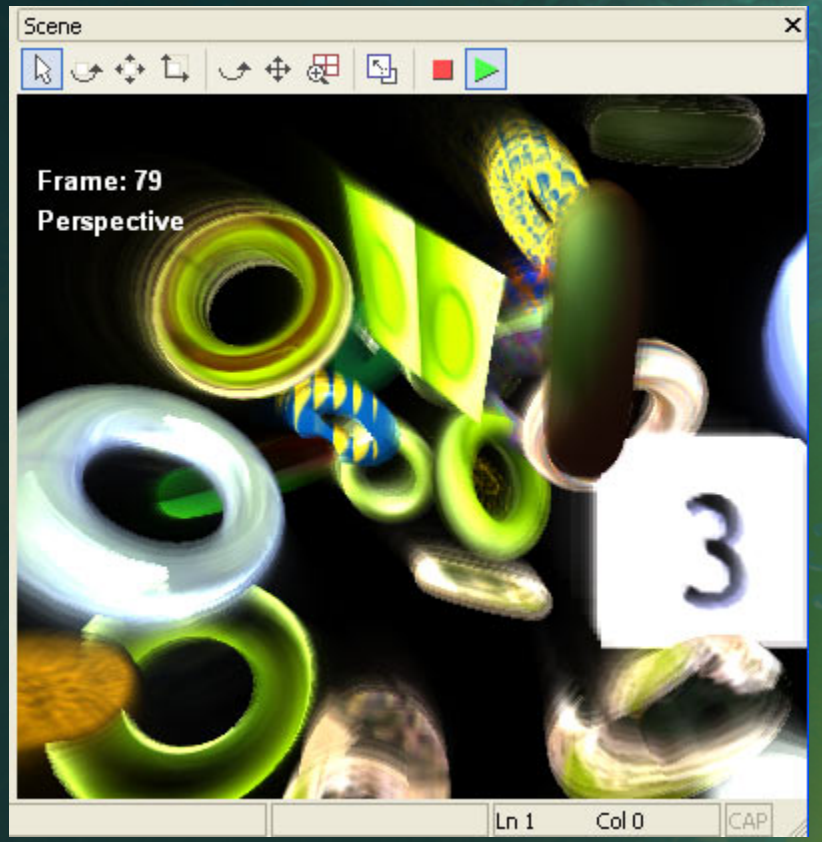
Bloom and Gloom: Shading Stack

- We can assign by hand in Material Editor, or edit in the ScriptExecute Sorter Window



Script Demo: Shader Donuts

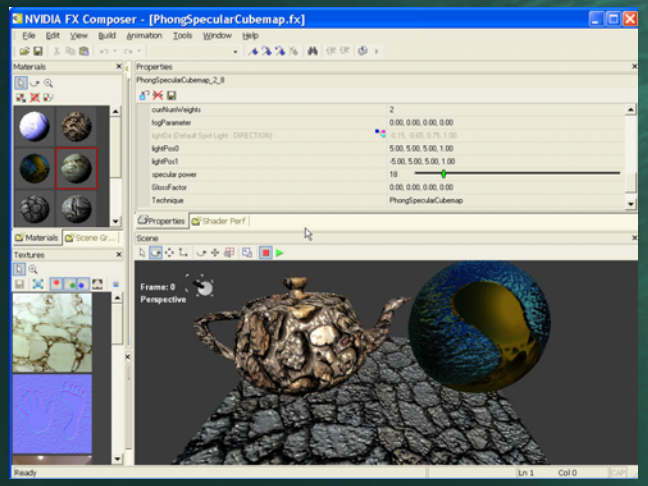
- Shows:
- C# Integration and highlighting
- OLE Browser for “nv_sys”
- Reading files from C#
- Creating Nodes
- Creating Keyframes
- Reading Annotations

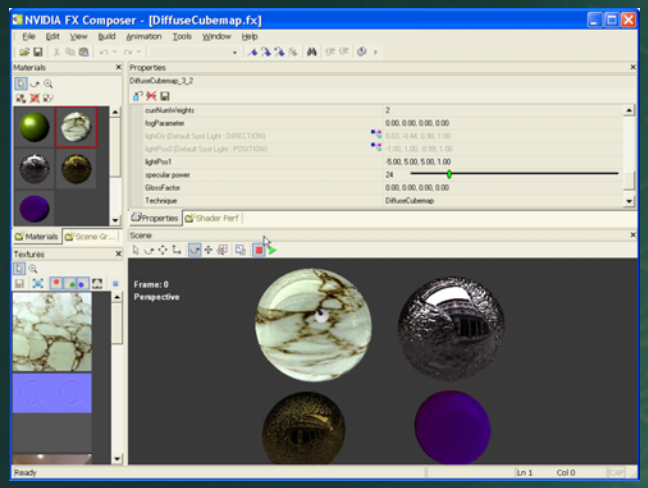


C# Animation

- C# Scripts are subclass of “INVUtility”
- The entry point is always
  ```
  public int Run(interop.nv_sys.NVSystem Sys)
  ```
- Use OLE browser to explore nv_sys namespace
- We can access almost every part of the FX Composer data structure, create objects, save images…
- …as well as communicate with other processes, files, etc.
- Samples supplied for both C# and VisualBasic.NET

SDK Demo: Import/Export





- Not a Demo *per se* – Plugins are created in Visual Studio etc
- Some functionality overlaps C#
- Can be used for Import of custom geometry formats, material export (mtl exporter shown)
- Pix courtesy Mike Chow at Rainbow Studios


Bonus Round
Some Shader Ideas

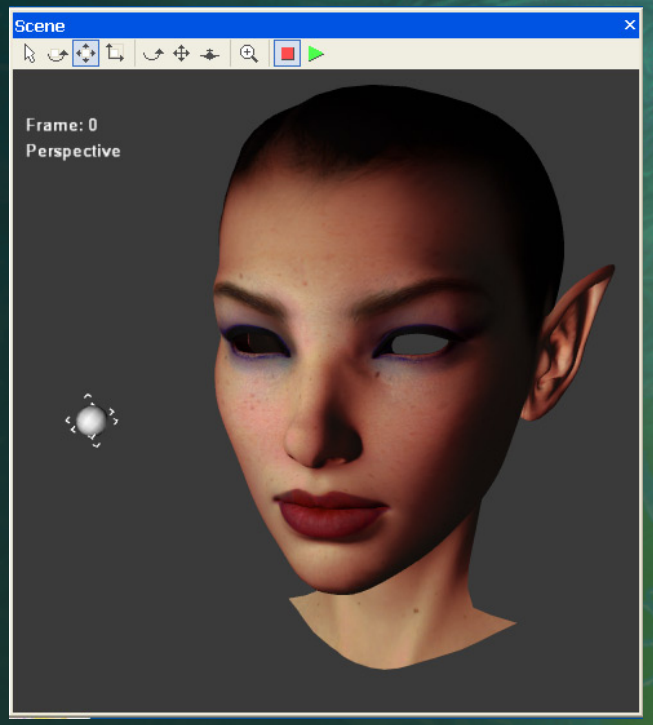
Coming Soon: Son of GLM

- DXSAS spec has looping constructs for lights and objects
  - Not in the current release, but soon!
- These will permit building shaders that can handle an arbitrary # of lights, and of varying types!
- Syntax is the same as current DXSAS looping
- Global vars can be assigned value-per light:

```
string frustum = “light#”;
```

Skin and Shading

- Diffuse Subsurface Scattering on the Cheap:
  - By remapping “(N·L)” in our diffuse-shading calulations to “((N·L)+w)/(1+w)” we can “wrap” light around the contours of an object
  - (Don’t worry about the math details – an example awaits!)
  - Since this is all in the diffuse lighting, it’s sometimes okay to do the job in the vertex shader



Skin and Direct Reflectance

- The younger you are, the less dead skin
- Live skin cells reflect like little cat's eye reflectors
- Therefore, a flat skin tone = youthful appearance
- Oren-Nayar Shading (expensive) and "grisaille" shading (cheap!)
- Combining ideas

One Modern Variation

Traditional Grisaille Relief

Reflections

- Can replace all specular in some circumstances
- Can use VM to generate CUBE maps
- Can have finite radius (see *GPU Gems*)
- Can have distance with quadratic falloff

Environment-mapped background, reflected card-shaped light source, 16-bit blending with overbright bloom

CGI, Films, and Painting

- Film borrows lighting and composition from media like painting
- Lighting leads attention
- Lighting sets emotional tone

Scott's *Blade Runner*

Rafael's *Transfiguration*

Smart Light Placement

- Magy Seif El-Nasr's "ELE": The Expressive Lighting Engine
- http://ist.psu.edu/SeifElNasr/
- Uses robotics load-balancing equations to maximize visibility and "mood" for a limited set of lights

Mirage, El-Nasr et al, CIRA

Compositing & 2D Effects

Halftoning Patterns

- FP buffers make things more powerful than ever
- Lots of fun…
- Color controls
- Final “sweetening”
- Blend modes
- Mix 2D/3D sprites
- Floating-point pixels

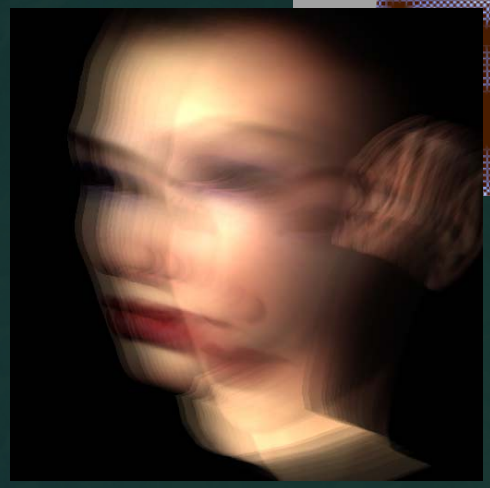


Image Trails


Conclusion

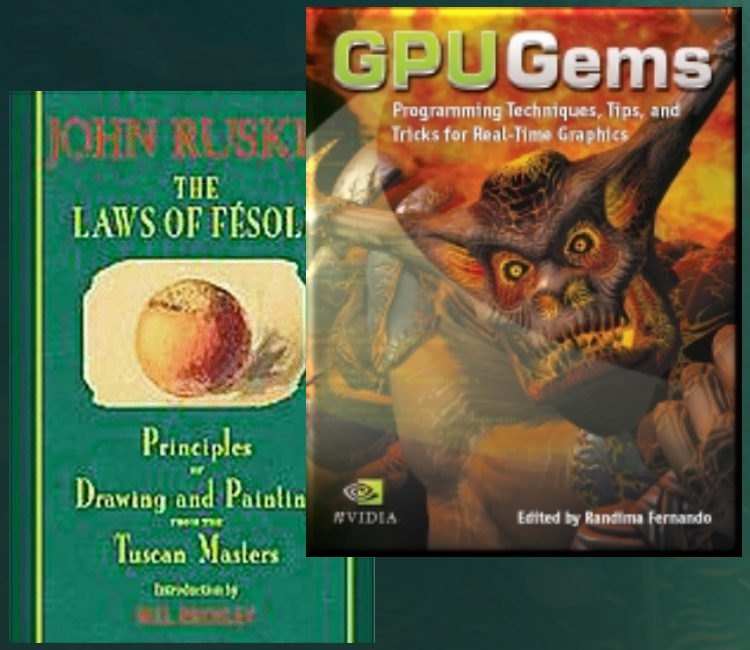
That’s a Wrap!

The End

- Games now have the capacity to match film shading, in character if not pixel-to-pixel
  - Get used to *lots* of shaders
  - Get tools that let you play
  - http://www.fxcomposer.com/
  - Play with shaders, try everything, keep a “sketchbook” of useful ideas

Some Recommended Books

- Jon Ruskin: *The Laws of Fésole, Priciples of Drawing and Painting from the Tuscan Masters*
- John Alton: *Painting with Light*
- Randima Fernando: *GPU Gems*



More On These Topics

- http://developer.nvidia.com/
- http://www.fxcomposer.com/
- kbjorke@nvidia.com

More At Siggraph

- NVIDIA Sponsored Sessions – Here in Room 401
  - *Shader Model 3.0 Unleashed*
    Tuesday 1:45
    Wednesday 3:45
    Thursday 10:30
  - *GPU Performance Tools and Analysis Techniques*
    Monday 3:45
    Wednesday 10:30
    Thursday 1:45
  - *Image Processing with the GPU*
    Monday 10:30
    Tuesday 3:45
    Wednesday 1:45
- Microsoft Sponsored Sessions – Room 402A
  - *HLSL Shader Workshop: Introductory*
    Monday & Wednesday, 10:30 and 3:45
    Tuesday and Thursday, 1:45
  - *HLSL Shader Workshop: Introductory*
    Monday and Wednesday, 1:45
    Tuesday and Thursday, 10:30 and 345