Creating Real Shaders in FX Composer

Kevin Bjorke

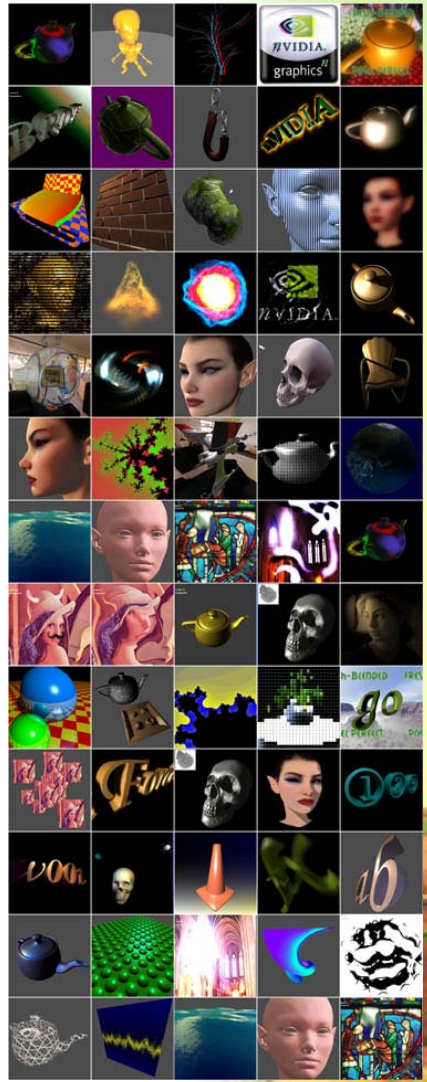
NVIDIA Corporation

HLSL for both Artists and Programmers

- High-Level talk here at GDC
- Examples of what you can do in FX Composer
  - Code details in these slides, available at http://developer.nvidia.com along with complete source code
- Your Models, Your Game Engine...
- Using FX Composer with DCC tools
  - Alias Maya
  - 3DS Max 7
  - RTZen Ginza

Touring the SDK Effects Examples

- **What's in there: more than we can show in the next few minutes!**
- **Start projects from the NVIDIA SDK Browser, or look in these directories:**
  - **MEDIA\project:** FX Composer project files
  - **MEDIA\HLSL:** individual .fx files
  - **MEDIA\scripts:** VB & C# script samples
- **Names identify general classes of .fx files: regular materials, "scene_xxx," "post_xxx," "pre_xxx," and "paint_xxx"**
- **Projects show shaders set-up, and sometimes show shaders interacting**



Some SDK Projects

Programmers: HLSL Beyond the Manual

- **This talk will include examples that show how to:**
  - **Use the CPU to generate textures etc**
  - **Use DirectX/XNA's DXSAS scripting**
  - **Write shaders for both DCC apps and FX Composer**
  - **Call on macros and functions from the NVIDIA #include files (.fxh) with FX Composer:**
    - `Quad.fxh, shadowMaps.fxh, Noise_3d.fxh, noise_2d.fxh, Spot_tex.fxh, nvMatrix.fxh`
- **Get at new NV4x Features**

DXSAS Scripting

- **These examples include techniques for:**
  - **MRTs**
  - **Loops of Passes**
  - **Looping on Booleans**
  - **FXCOMPOSER_RESET**
  - **Re-Using Texture Samplers**
  - **Using the GPU for Texture Creation**

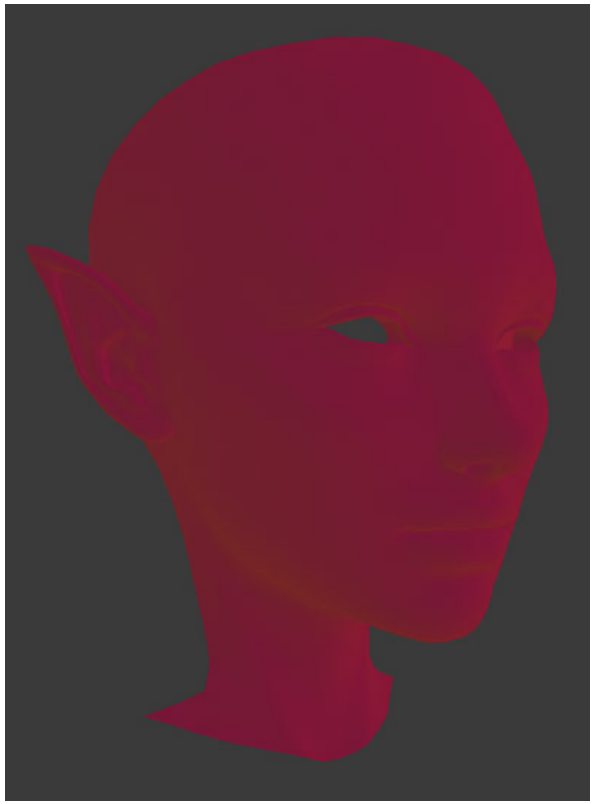
Example Shader: scene_lineDraw.fx

- Uses #include
- Uses MRT
- Uses “half” data
- Uses DXSAS scene commands
- Uses static data

Sample from scene_lineDraw.fx

Edge Detect Based on Normals

- **Potential, but Has Artifacts**

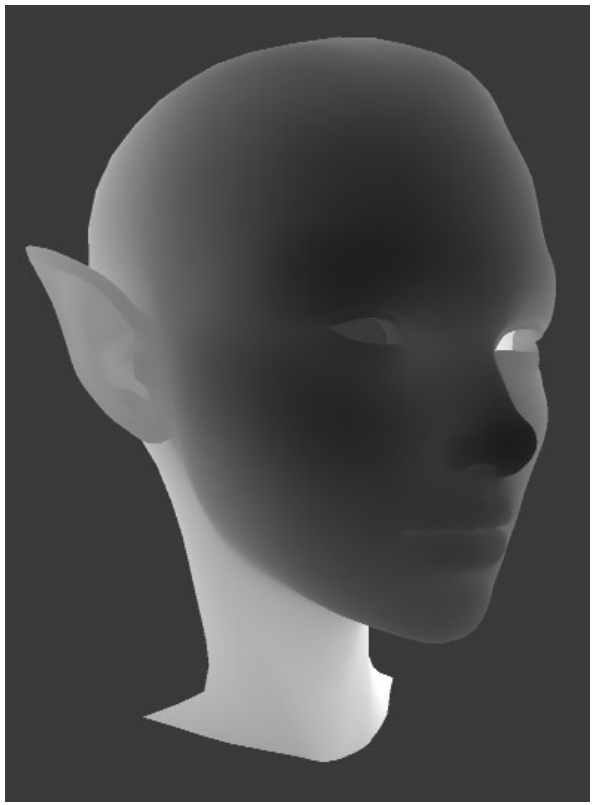


Worldspace Normals

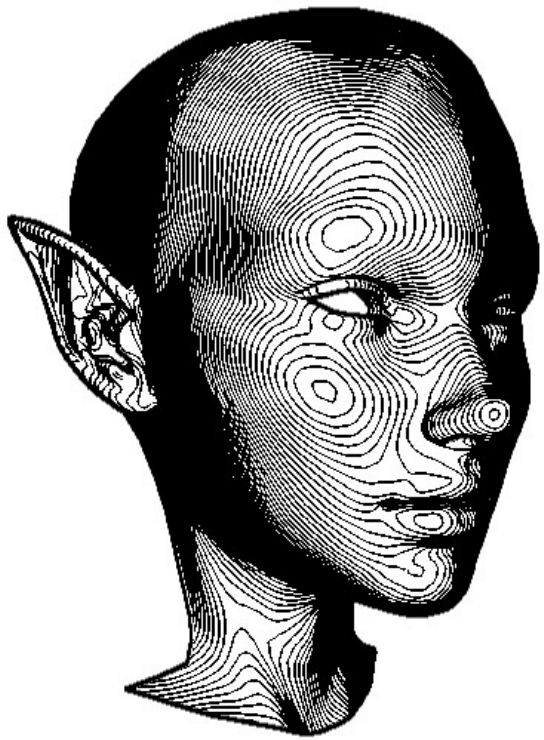
Edges

Edge Detect Based on Depth

- **Has Different Artifacts**



Depth



Edges

Combining Results

- **Much Smoother, Artifacts tend to cancel even in bad cases**

Artist-tweaked

Intersection of (Poor) Edges

Parameters We Will Need

- **The parameters we borrow from the original shaders:**
  - **Two edge-detect threshholds**
  - **Hither/Far values for depth image**
- **For scene commands:**
  - **Color for screen-clear**
  - **Value for depth-clear (hidden)**

lineDraw - beginning

- We include “Quad.fxh” for macros, types, and shader functions
- QUAD_REAL defaults to “half”
  - We can override it by #defining QUAD_FLOAT before #including Quad.fxh
- We will use Quad.fxh Render-to-Texture Declaration Macros
- Quad.fxh also provides vertex and pixel shader functions for simplest screen-aligned-quad cases: writing “straight” textures.

```
#include "Quad.fxh"
```

lineDraw – starting DXSAS

- **This shader is a “scene” effect**
- **We provide mutliple techniques, for different HW profiles**
- **Two extra techniques for artist tuning**

```
float Script : STANDARDSGLOBAL <
    string UIWidget = "none";
    string ScriptClass = "scene";
    string ScriptOrder = "standard";
    string ScriptOutput = "color";
    string Script =
    "Technique=Technique?NV3X:NV4X:NormsOnly:DepthOnly;";
> = 0.8; // version #
```

ScriptClass

Dedicated “Artist” Techniques

lineDraw “untweakables”

- **Tracked automatically by app - no user override**
- **UIWidget = “none” improves performance**

No Widget Display

```
float4x4 WorldITXf : WorldInverseTranspose <
                              string UIWidget="None"; >;
float4x4 WorldViewProjectionXf : WorldViewProjection <
                              string UIWidget="None"; >;
float4x4 WorldViewXf : WorldView <
                              string UIWidget="None"; >;
float4x4 WorldXf : World <
                              string UIWidget="None"; >;
float4x4 ViewIXf : ViewInverse <
                              string UIWidget="None"; >;
```

lineDraw static parameters

- Static values are “invisible” to the UI
- Calculated by the CPU
- Can call most HLSL functions, intrinsic or user-defined
- QUAD_REAL type declared by Quad.fxh
- QuadTexOffset and QuadScreenSize are hidden parameters declared by Quad.fxh

static

```
static float EdgeT2 = (Threshhold * Threshhold);
static float DeepT2 = (ThreshholdD * ThreshholdD);

static QUAD_REAL2 TexelCornerOffset =
       QUAD_REAL2(QuadTexOffset/(QuadScreenSize.x),
                  QuadTexOffset/(QuadScreenSize.y));
```

lineDraw Texture Declarations

- **Macros from “Quad.fxh” for common RTT texturing**
- **Standard declarations (like these) match screen size exactly (so resizing the window will re-allocate them)**

```
DECLARE_QUAD_TEX(NormTexture,NormSampler,"X8R8G8B8")
DECLARE_QUAD_TEX(DeepTexture,DeepSampler,"X8R8G8B8")
DECLARE_QUAD_DEPTH_BUFFER(DepthBuffer, "D24S8")
```

lineDraw Template

- **QUAD_REAL data**
- **We perform both edge detects and multiply the results**
- **:COLOR semantic on function itself**

Function Output Semantic

```
QUAD_REAL4 edgeDetect2PS(EdgeVertexOutput IN) : COLOR {
   QUAD_REAL n = edgeDetectGray(IN,NormSampler,EdgeT2);
   QUAD_REAL d = edgeDetectR(IN,DeepSampler,DeepT2);
   QUAD_REAL line = 1 - (n*d);
   return line.xxxx;
}
```

Complete Technique

- **Looks Complex but Just 4 (or 3) Chunks:**
  - **Script; Normal, Depth, & Edge Passes**

```
technique NV3X <
          string Script = "Pass=Norms;"
                          "Pass=Depth;"
                          "Pass=ImageProc;";
> {
          pass Norms <
               string Script = "RenderColorTarget0=NormTexture;"
                          "RenderDepthStencilTarget=DepthBuffer;"
                          "ClearSetColor=BlackColor;"
                          "ClearSetDepth=ClearDepth;"
                          "Clear=Color;"
                          "Clear=Depth;"
                          "Draw=Geometry;";
          > {
                          VertexShader = compile vs_2_0 simpleVS();
                          ZEnable = true;
                          ZWriteEnable = true;
                          CullMode = None;
                          AlphaBlendEnable = false;
                          PixelShader = compile ps_2_a normPS();
          }
          pass Depth <
               string Script = "RenderColorTarget0=DeepTexture;"
                          "RenderDepthStencilTarget=DepthBuffer;"
                          "ClearSetColor=BlackColor;"
                          "ClearSetDepth=ClearDepth;"
                          "Clear=Color;"
                          "Clear=Depth;"
                          "Draw=Geometry;";
          > {
                          VertexShader = compile vs_2_0 simpleVS();
                          ZEnable = true;
                          ZWriteEnable = true;
                          CullMode = None;
                          AlphaBlendEnable = false;
                          PixelShader = compile ps_2_a deepPS();
          }
          pass ImageProc <
              string Script = "RenderColorTarget0=;"       // re-use
                          "RenderDepthStencilTarget=;"
                          "Draw=Buffer;";
          > {
                          cullmode = none;
                          ZEnable = false;
                          ZWriteEnable = false;
                          AlphaBlendEnable = false;
                          VertexShader = compile vs_1_1 edgeVS();
                          PixelShader = compile ps_2_0 edgeDetect2PS();
          }
}
```

Technique: Chunk 1 of 4

- DXSAS scripts at each step
- The “Technique” script is optional for this case (one pass after another)

```
technique NV3X <
   string Script = "Pass=Norms;"
                   "Pass=Depth;"
                   "Pass=ImageProc;";
>  {
   // . . .
```

Technique: Chunk 2 of 4

- **We redirect color output to “NormTexture” & Draw the Model Geometry**

```
pass Norms <
    string Script = "RenderColorTarget0=NormTexture;"
                    "RenderDepthStencilTarget=DepthBuffer;"
                    "ClearSetColor=BlackColor;"
                    "ClearSetDepth=ClearDepth;"
                    "Clear=Color;"
                    "Clear=Depth;"
                    "Draw=Geometry;";
> {
    VertexShader = compile vs_2_0 simpleVS();
    ZEnable = true;
    ZWriteEnable = true;
    CullMode = None;
    AlphaBlendEnable = false;
    PixelShader = compile ps_2_a normPS();
}
```

Render to Texture

Offscreen Depth Buffer

All Current Models

Technique: Chunk 3 of 4

- **Redirect Color Output to "DeepTexture" & Draw Model Again**

```
pass Depth <
    string Script = "RenderColorTarget0=DeepTexture;"
                    "RenderDepthStencilTarget=DepthBuffer;"
                    "ClearSetColor=BlackColor;"
                    "ClearSetDepth=ClearDepth;"
                    "Clear=Color;"
                    "Clear=Depth;"
                    "Draw=Geometry;";
> {
    VertexShader = compile vs_2_0 simpleVS();
    ZEnable = true;
    ZWriteEnable = true;
    CullMode = None;
    AlphaBlendEnable = false;
    PixelShader = compile ps_2_a deepPS();
}
```

New Render Target

Re-use Depth Buffer

All Current Models

Technique: Chunk 4 of 4

- **Combine, Edge Detect, write result to Frame Buffer**
- *Ignore scene geometry*

```
pass ImageProc <
    string Script = "RenderColorTarget0=;"
                    "RenderDepthStencilTarget=;"
                    "Draw=Buffer;";
> {
    cullmode = none;
    ZEnable = false;
    ZWriteEnable = false;
    AlphaBlendEnable = false;
    VertexShader = compile vs_1_1 edgeVS();
    PixelShader = compile ps_2_0 edgeDetect2PS();
}
```

Reset Render Target

Reset Depth Target

Screen-Aligned Quad

lineDraw MRT Technique

- **We can collapse the first two passes**
- **Remember to reset *all* outputs!**

```
pass NormsAndDepth <
    string Script = "RenderColorTarget0=NormTexture;"
                    "RenderColorTarget1=DeepTexture;"
                    "RenderDepthStencilTarget=DepthBuffer;"
                    "ClearSetColor=BlackColor;"
                    "ClearSetDepth=ClearDepth;"
                    "Clear=Color;"
                    "Clear=Depth;"
                    "Draw=Geometry;";
> {
    VertexShader = compile vs_2_0 simpleVS();
    ZEnable = true;
    ZWriteEnable = true;
    CullMode = None;
    AlphaBlendEnable = false;
    PixelShader = compile ps_2_a geomMRT_PS();
}
```

Target 0

Target 1

Offscreen Depth Buffer

All Current Models

lineDraw MRT shader

- Use “out” to specify multiple return values
- Func can be “void” or return a value via function semantic

```
QUAD_REAL4 vecColorN(QUAD_REAL3 V) {
    QUAD_REAL3 Nc = 0.5*(normalize(V)+((1.0).xxx));
    return QUAD_REAL4(Nc,1);
}

void geomMRT_PS(
          vertexOutput IN,
          out QUAD_REAL4 normColor : COLOR0,
          out QUAD_REAL4 deepColor : COLOR1
) {
    normColor = vecColorN(IN.WorldNormal);
    QUAD_REAL d = (IN.EyePos.z-Near)/(Far-Near);
    deepColor = QUAD_REAL4(d.xxx,1);
}
```

Target 0

Target 1

MRT shader alternative form

- Shader function can be “void” or return a value via function semantic
- :COLOR0 is the same as :COLOR

Function Output Semantic

```
QUAD_REAL4 geomMRT_PS(
        vertexOutput IN,
        out QUAD_REAL4 deepColor : COLOR1) : COLOR0
{
   QUAD_REAL d = (IN.EyePos.z-Near)/(Far-Near);
   deepColor = QUAD_REAL4(d.xxx,1);
   return vecColorN(IN.WorldNormal);
}
```

lineDraw Tuning Technique 1

- **Provide a visualization for artists to tune params for edgeNorms**

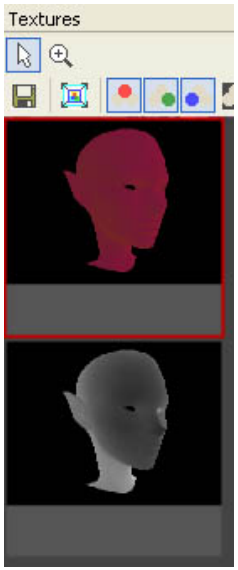
```
technique NormsOnly {
   pass Norms <
       // . . .
```

Tuned Normals Edges

lineDraw Tuning Technique 2

- **Likewise for Depth and edge parameters**

```
technique DepthOnly {
   pass Depth <
       // . . .
```

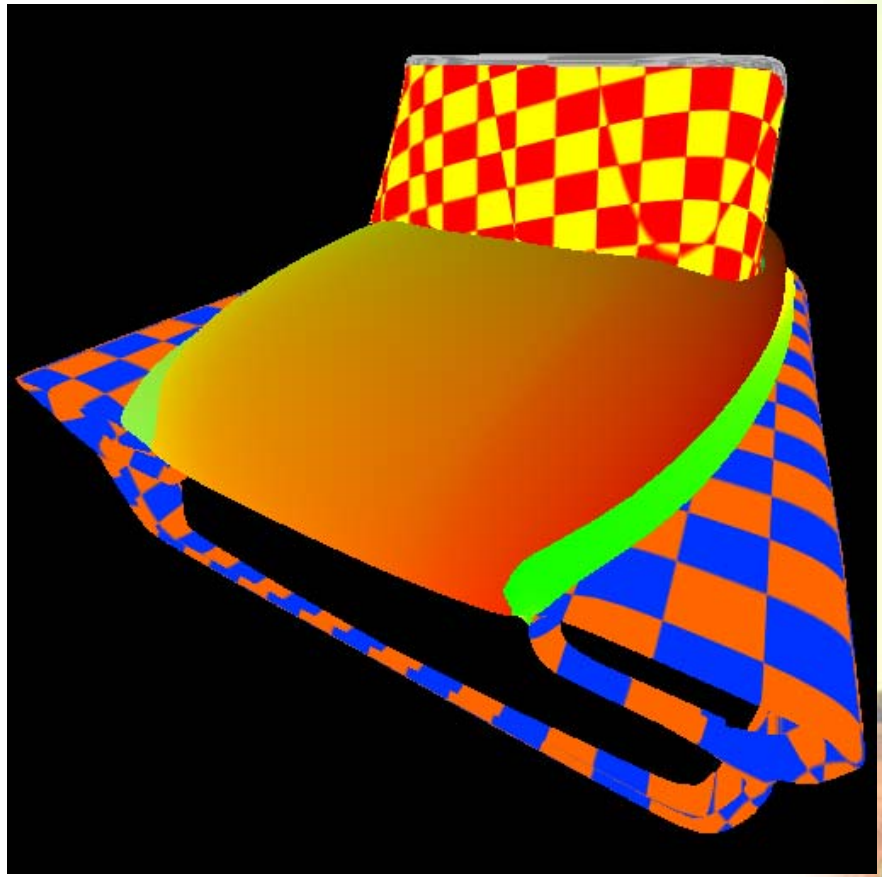


Live Texture Display in FX Composer

Tuned Depth Edges

Example Shader: SeeSpaces.fx

- Artist Visualization
- Uses texture generation and texture derivatives on CPU for fast AA
- Debugging



Sample from "DebugCab.fxproj"

Generating procedural textures

- :COLOR sematic like a pixel shader
- :PSIZE input semantic gives texel size as function is called for *each* MIP level
- This is the *only* way to get at the HLSL noise() intrinsic

```
float4 MakeStripe(float2 Pos : POSITION,float ps : PSIZE) : COLOR
{
   float v = 0;
   float nx = Pos.x+ps; // keep the last column full-on, always
   v = nx > Pos.y;
   return float4(v.xxxx);
}

#define TEX_SIZE 128
texture stripeTex <
    string function = "MakeStripe";
    string UIWidget = "None";
    float2 Dimensions = { TEX_SIZE, TEX_SIZE };
>;
sampler2D StripeSampler = sampler_state {
    Texture = <stripeTex>;
    MinFilter = LINEAR; MagFilter = LINEAR; MipFilter = LINEAR;
    AddressU = WRAP;
    AddressV = CLAMP;
};
```

Output Semantic

Input Semantic

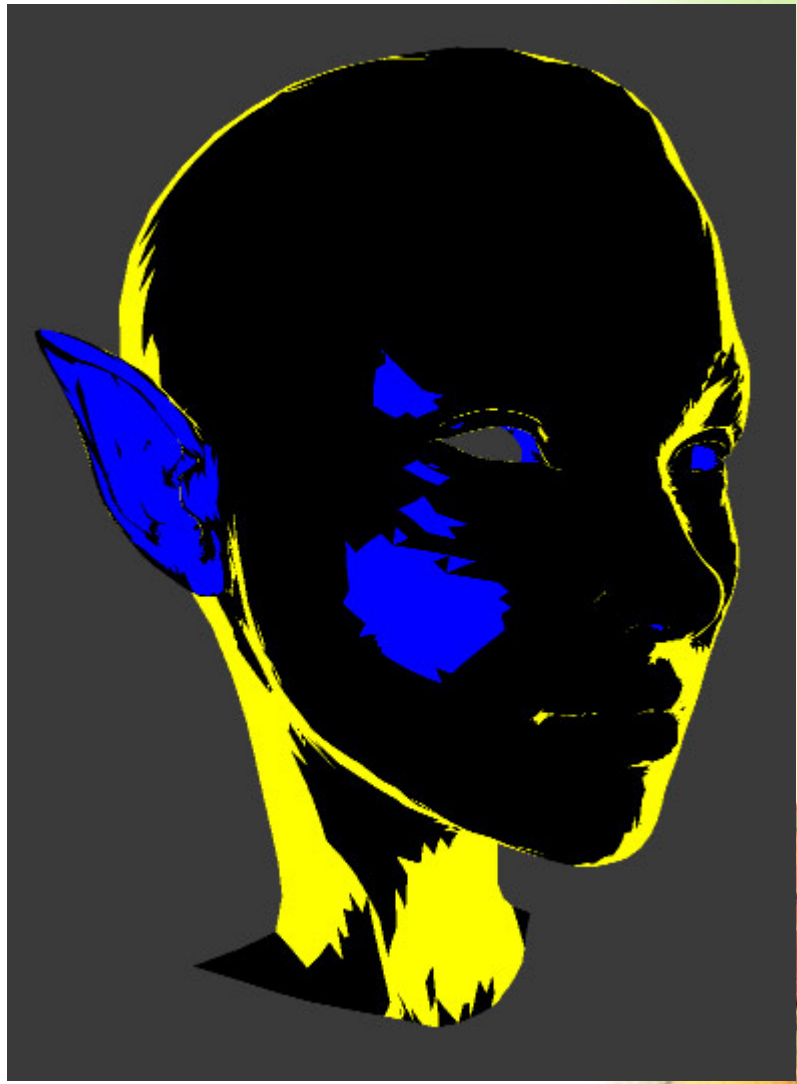
Call generator function

No user interface needed

Be sure to set address modes appropriate for individual texture and algorithm

Example Shader: uvDetective

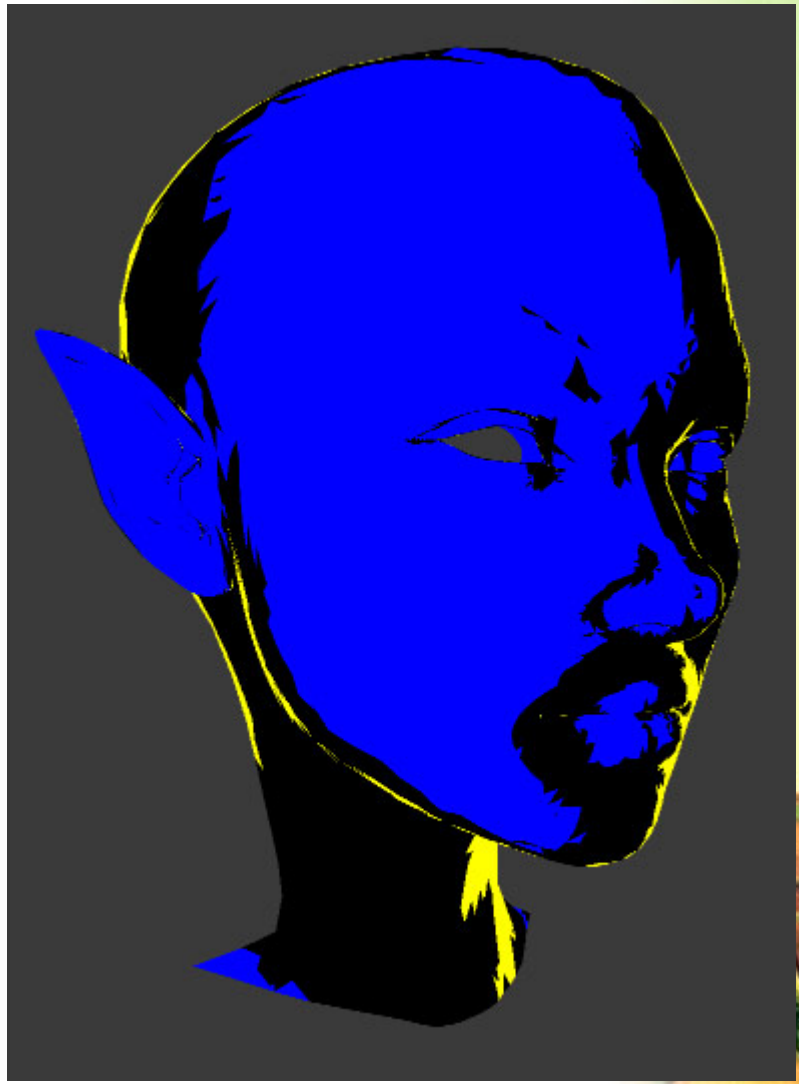
- Visualization for Artists Tuning Models
- Black – texture should be around 512x512 for close-to-texel-sized pixels



Black areas for 512x512 texture

Can be set to any size

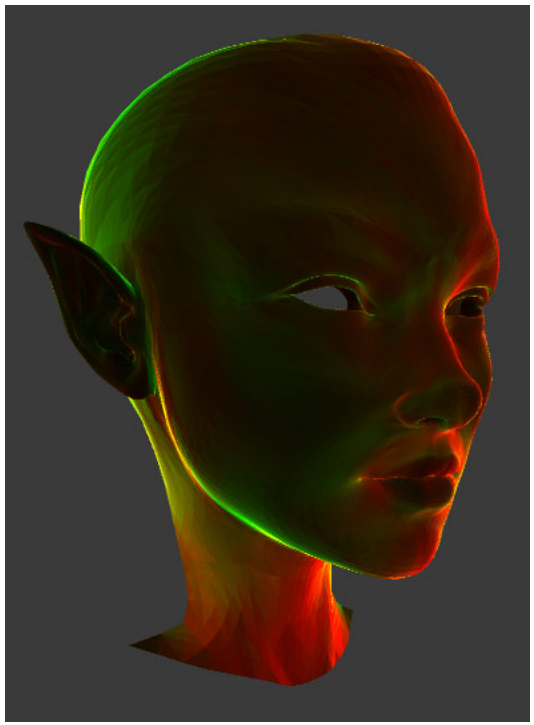
- Now black is for 256 res
- Blue shows area where a higher-res texture *could* be useful



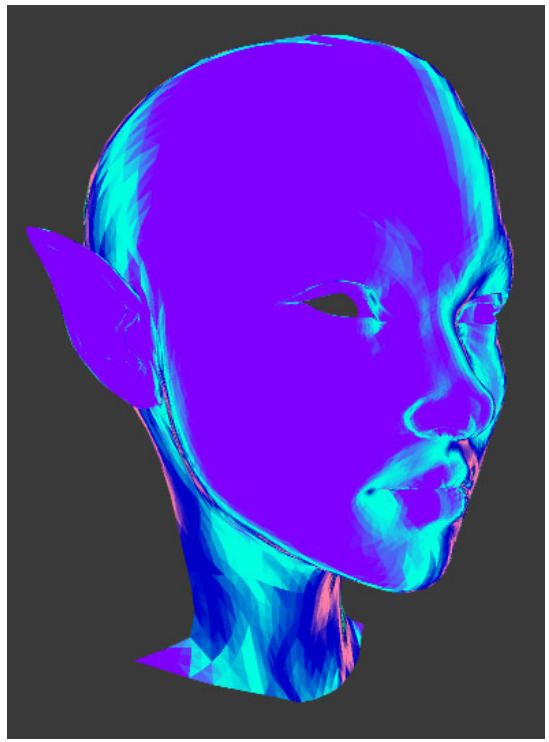
Black areas for 256x256 texture

Show Related Visualizations Too

- **Direct Derivatives and (CPU-generated) false MIP coloring**



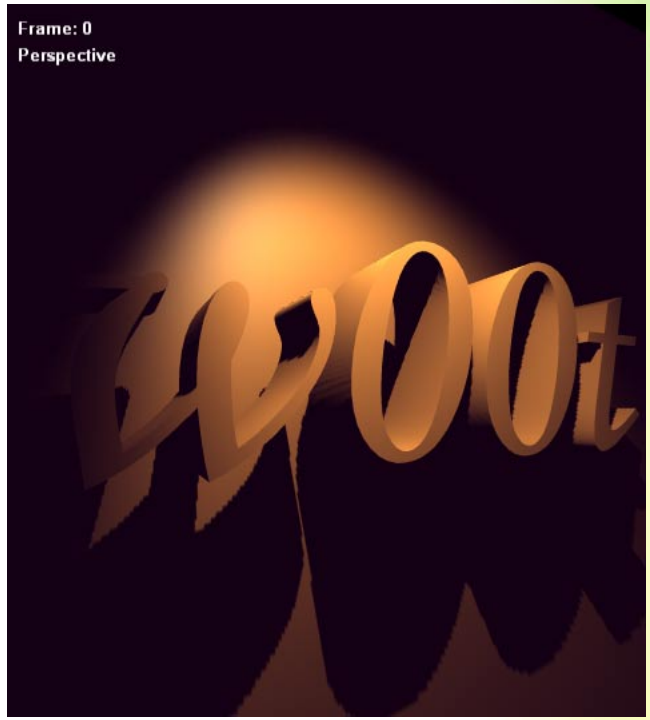
Direct Visualization of Texture Derivatives
(Amount of texture stretching)



"False Color MIP Texture" Display
(texture generated by uvDetective.fx)

Example Shader: shadowSpot2.fx

- Special shadow format
- DXSAS:
  - "sceneorobject" ScriptClass
  - Script/No Script
- Uses RenderPort
- Uses CPU intrinsics
- Include files:
  - shadowMap.fxh
  - spot_tex.fxh



HW shadow mapping

shadowSpot2 – shadow texture

- **Shadow texture format**
- **We throw away color portion**
- **Vertex shader declared for us**

Found in …\MEDIA\HLSL

```
#include “shadowMap.fxh”

DECLARE_SHADOW_XFORMS("light0",LampViewXf,
        LampProjXf,ShadowViewProjXf)
DECLARE_SHADOW_BIAS
DECLARE_SHADOW_MAPS(ColorShadMap,ColorShadSampler,
        ShadDepthTarget,ShadDepthSampler)
```

Inside shadowMap.fxh - Maps

- DECLARE_SHADOW_MAPS will set up two map and sampler pairs
- Default Size is 512
- We can override by pre-#defining SHADOW_SIZE
- Uses format "D24S8_SHADOWMAP“ which will provide HW-accelerated multisample PCF filtering

```
DECLARE_SHADOW_MAPS(ColorShadMap,ColorShadSampler,
        ShadDepthTarget,ShadDepthSampler)
```

Inside shadowMap.fxh - Transforms

- **DECLARE_SHADOW_XFORMS declares attachable transforms using special "frustum" annotation and an additional "static" declaration:**

```
// DECLARE_SHADOW_XFORMS("light0",LampViewXf,
//                LampProjXf,ShadowViewProjXf)
// expands to:

float4x4 LampViewXf : View < string frustum = "light0"; >;
float4x4 LampProjXf : Projection < string frustum = "light0"; >;
static float4x4 ShadowViewProj = mul(LampViewXf,LampProjXf);
```

"frustum" annotation

"static" declaration executes HLSL code on CPU each frame

Inside shadowMap.fxh - Bias

- DECLARE_SHADOW_BIAS will set up a user parameter “ShadBias”
- We can override range for small or large models by pre-#defining MAX_SHADOW_BIAS

```
DECLARE_SHADOW_BIAS
```

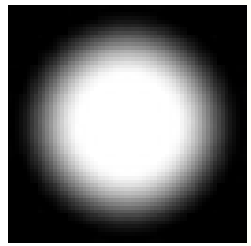
Inside shadowMap.fxh - Shaders

- Vertex shader for creating shadow maps: “shadCamVS”
- No pixel shader needed for shadow-creation passes
- Vertex shader for using shadow maps: “shadowUseVS”
  - Shadow projection TexCoords (UVs) passed in “LProj”
- Code sample in .fxh for usage in Pixel shaders

shadowSpot2 – spotlight pattern

- **“SpotSamp” sampler will be declared for you and filled**
- **Compile-time shaping options**

```
#include “spot_tex.fxh”
```



Default “spot_tex” texture

- **Call “SpotSamp” using light projection UVs like so:**

```
float cone = tex2Dproj(SpotSamp,IN.LProj);
```

shadowSpot2 – pixel shader

- **Just shadow portion**
- **“LProj” provided by vertex shader “shadowUseVS”**

```
float4 useShadowPS(ShadowingVertexOutput IN) : COLOR
{
   float3 litPart, ambiPart;
   lightingCalc(IN,litPart,ambiPart);
   float4 shadowed =  tex2Dproj(ShadDepthSampler,IN.LProj);
   return float4((shadowed.x*litPart)+ambiPart,1);
}
```

shadowSpot2 - pixel shader

- **Compare to a completely unshadowed version:**
  - **We supply an *unshadowed* version for apps with limited DXSAS scripting, like 3DStudio Max**
    - **And declare ScriptClass = "sceneorobject";**

```
float4 unshadowedPS(ShadowingVertexOutput IN) : COLOR
{
   float3 litPart, ambiPart;
   lightingCalc(IN,litPart,ambiPart);
   return float4(litPart+ambiPart,1);
}
```

shadowSpot2 - shadow technique

- **Vertex shader from .fxh file:**
- **Note assign of "RenderPort"**

```
technique Shadowed <
    string Script = "Pass=MakeShadow;"
           "Pass=UseShadow;";
> {
 pass MakeShadow <
     string Script = "RenderColorTarget0=ColorShadMap;"
          "RenderDepthStencilTarget=ShadDepthTarget;"
          "RenderPort=light0;"
          "ClearSetColor=ShadowClearColor;"
          "ClearSetDepth=ClearDepth;"
          "Clear=Color;"
          "Clear=Depth;"
          "Draw=geometry;";
 > {
    VertexShader = compile vs_2_0 shadowGenVS(WorldXf,WorldITXf,ShadowViewProjXf);
    ZEnable = true;
    ZWriteEnable = true;
    ZFunc = LessEqual;
    CullMode = None;
    // no pixel shader!
 }
 // . . . Continued . . .
```

"RenderPort" sets clipping etc correctly for this viw

Provided by shadowMap.fxh

shadowSpot2 – technique (cont'd)

- **Vertex Shader provided from .fxh**
- **Remember, Reset "RenderPort"**

```
        // . . .
        pass UseShadow <
        string Script = "RenderColorTarget0=;"
                        "RenderDepthStencilTarget=;"
                        "RenderPort=;"
                        "ClearSetColor=ClearColor;"
                        "ClearSetDepth=ClearDepth;"
                        "Clear=Color;"
                        "Clear=Depth;"
                        "Draw=geometry;";
      > {
        VertexShader = compile vs_2_0 shadowUseVS(WorldXf,WorldITXf,
                        WorldViewProjXf,ShadowViewProjXf,
                        ViewIXf,ShadBiasXf, SpotLightPos);
        ZEnable = true;
        ZWriteEnable = true;
        ZFunc = LessEqual;
        CullMode = None;
        PixelShader = compile ps_2_a useShadowPS();
        }
}
```

Reset Renderport to scene camera

Provided by shadowMap.fxh

shadowSpot2 – unshadowed technique

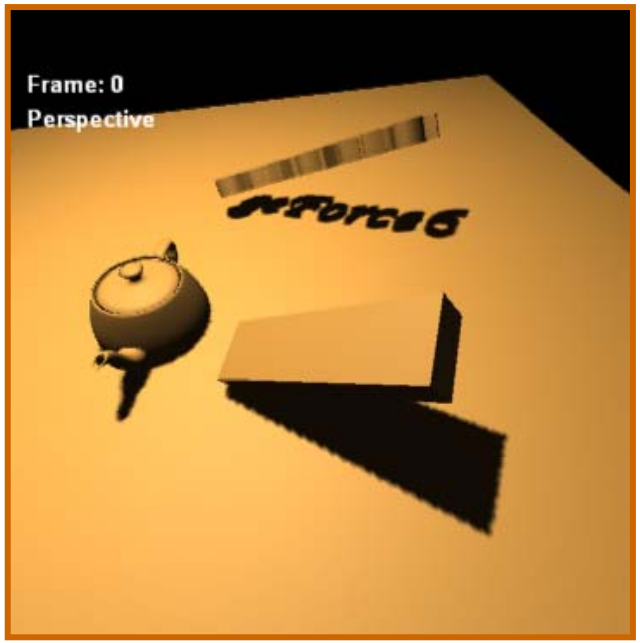
- **Provided for apps like 3DS Max**
- **Just one pass, shared code**
- **DXSAS Script optional**
- **Declare ScriptClass “sceneorobject”**

Scene w/o shadow

Provided by shadowMap.fxh

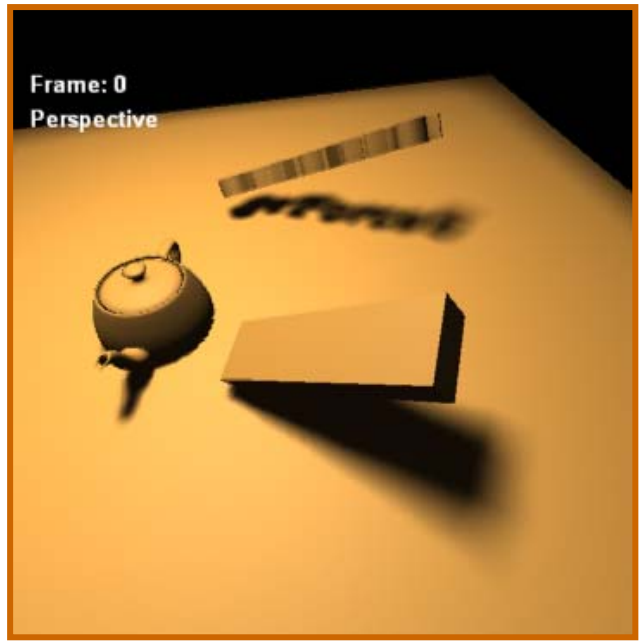
```
technique Unshadowed {
  pass NoShadow {
    VertexShader = compile vs_2_0 shadowUseVS(WorldXf, WorldITXf, WorldViewProjXf,
                                              ShadowViewProjXf, ViewIXf,
                                              ShadBiasXf, SpotLightPos);

    ZEnable = true;
    ZWriteEnable = true;
    ZFunc = LessEqual;
    CullMode = None;
    PixelShader = compile ps_2_a unshadowedPS();
  }
}
```

Differing Shadow Formats & Algorithms



D24S8 Shadow Maps

- Fast, good quality
- Antialiased on NVIDIA hardware
- Sharp edges
- Trivial to use

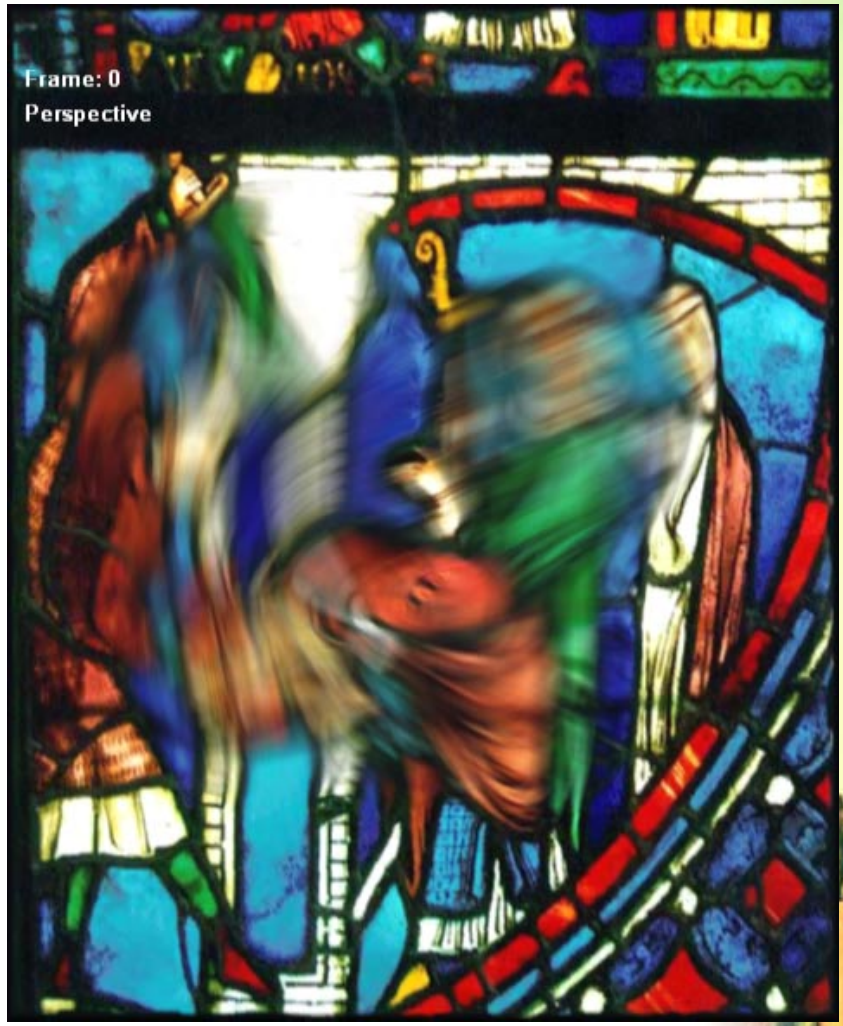


Floating Point

- Most flexible
- AA calculated in shader, so anything is possible
- Can be mixed with RGB in one texture

Example Shader: paint_blur

- Uses FP16 Blending
- Uses DXSAS accumulation loops
- Uses “bool loops”
- Uses CPU funcs and static vars for mouse tracking



Painted Accumulation-Buffer Motion Blur

Paint_blur - Three key params

- **Loop counter & limit**
- **RESET pulse boolean**
  - **Can also be toggled manually**

Hidden loop counter

```
float passnumber <string UIWidget = "none";>;
float npasses <
   float UIStep = 1.0;
   string UIName = "# of blur passes";
> = 8.0f;
bool bReset : FXCOMPOSER_RESETPULSE
<
   string UIName="Clear Canvas?";
>;
```

Dedicated Semantic

Declaring Floating Point Textures

- Just like any other texture
- Our paint strokes are added using Alpha Blending – works fine on FP16 formats
- Caution: FXC will still compile if a format is not available – it will switch to 8bit int

```
DECLARE_QUAD_TEX(PaintTex,PaintSamp,"A16B16G16R16F")
```

A sample "live" displacement texture

Paint_blur - DXSAS looping

- **Loop value from parameter in technique script**
  - **Change value to change blur quality**

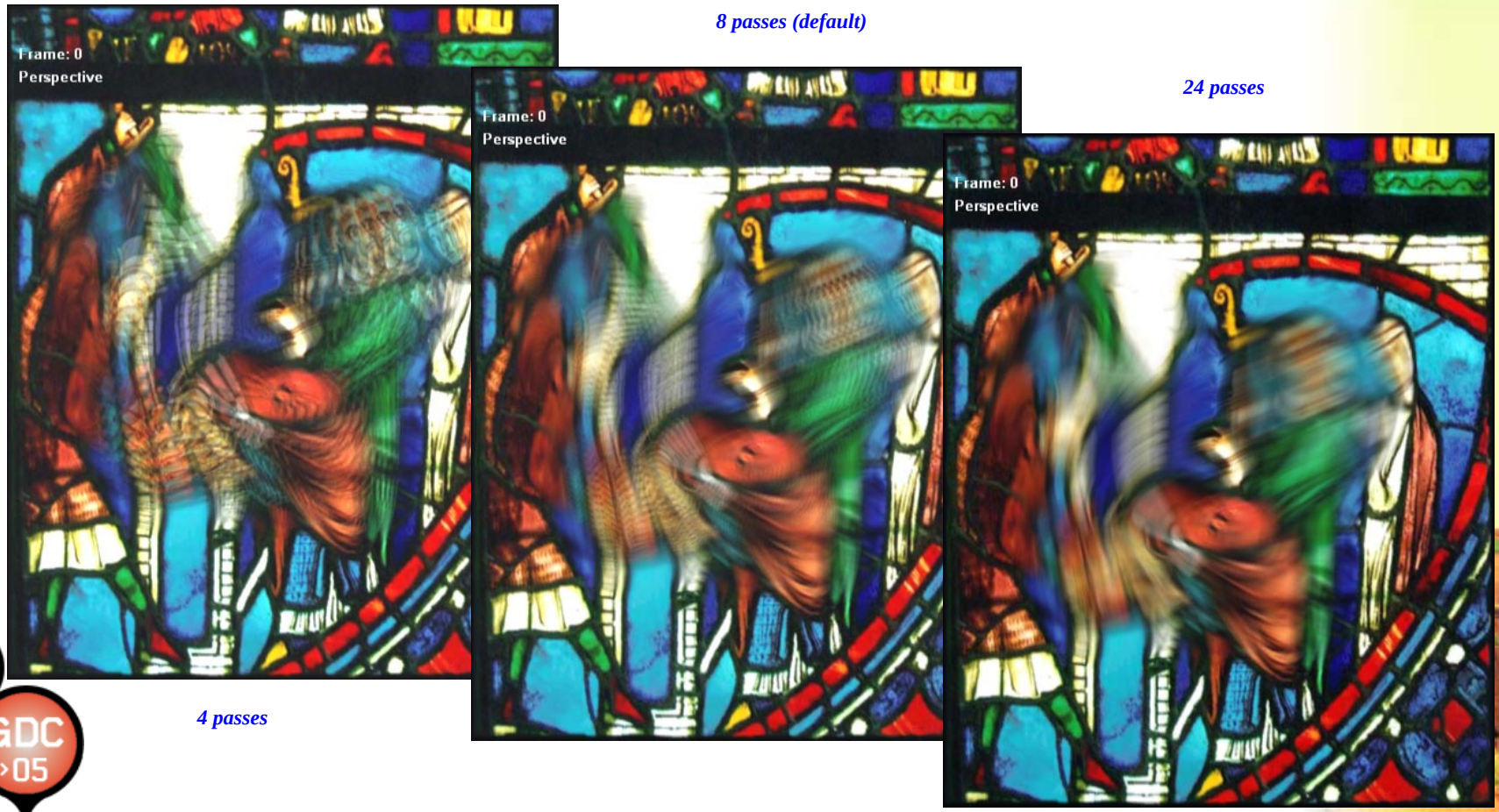
```
string Script =
   // Clear Accum Buffer
   "RenderColorTarget0=AccumBuffer;"
   "ClearSetColor=ClearColor;"
   "Clear=Color;"
   // paint into blur-dir buffer...
   "Pass=paint;"
   // accumulate
   "LoopByCount=npasses;"
       "LoopGetIndex=passnumber;"
       "Pass=Accumulate;"
   "LoopEnd;"
   // draw accum buffer to framebuffer
   "Pass=FinalPass;";
```

User-defined loop limit

Script counter assignment

Effects of Changing Pass Count

- Tune for Quality versus Performance

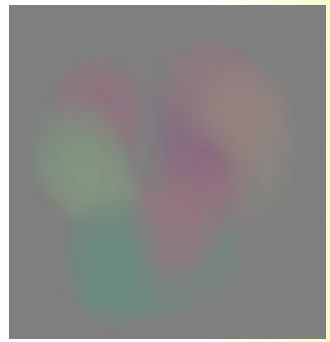


Paint_blur - DXSAS "bool" looping

- **Loop value from RESET, inside script for "Paint" pass**
  - **Painting clears itself as needed**
  - **Otherwise "PaintTex" persists from frame to frame**

```
string Script =
   "RenderColorTarget0=PaintTex;"
   "RenderDepthStencilTarget=;"
   "LoopByCount=bReset;"
              "ClearSetColor=ClearColor;"
              "Clear=Color0;"
              "LoopEnd=;"
   "Draw=Buffer;";
```

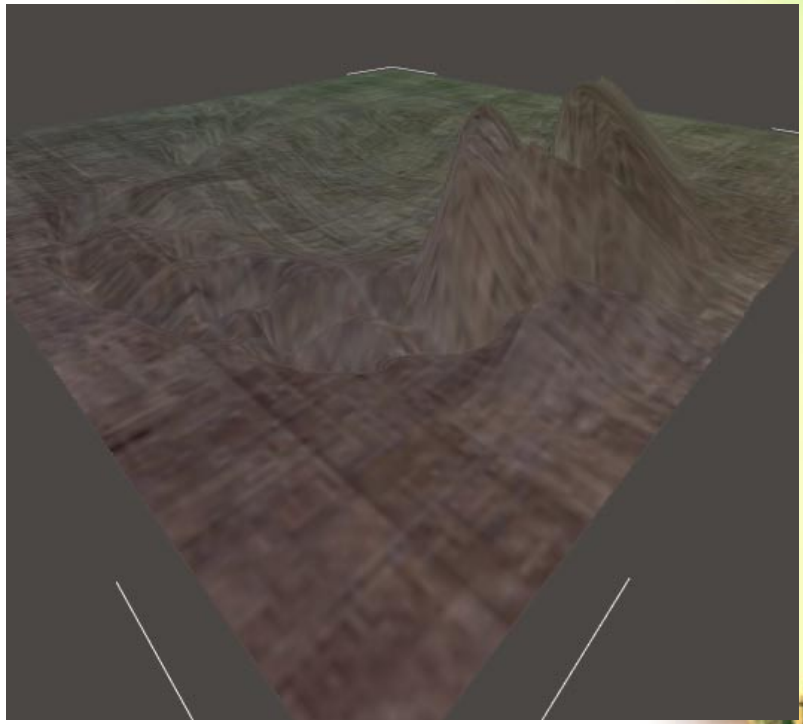
With "bool" value, acts like "if()"



"PaintTex" display

Example shader: paint_sculpt

- Uses FP blending
- Converts to FP32
- Uses FP32 VTF



Live texture sculpting on static plane

Paint_sculpt – mixing data

- **FP16 blending for paint, as before**
- **Extra copy pass for VTF FP32**
- **Use Quad.fxh utility shaders**

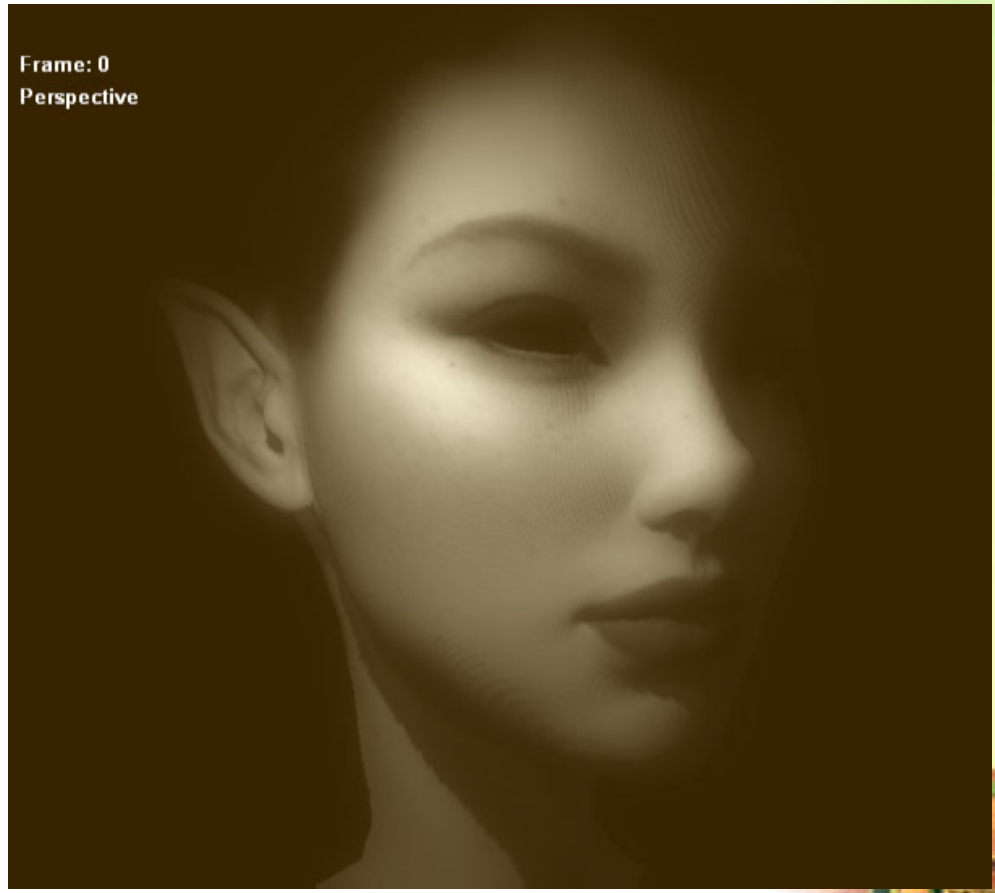
```
pass boost <
   string Script =     "RenderColorTarget0=DisplaceMap;"
                       "Draw=Buffer;";
> {
   VertexShader = compile vs_3_0 ScreenQuadVS();
   ZEnable = false;
   ZWriteEnable = false;
   CullMode = None;
   PixelShader = compile ps_3_0 TexQuadPS(PaintStrokeSampler);
}
```

Provided by Quad.fxh

Provided by Quad.fxh

Example shader(s): post_holga & friends

- Uses noise_2d, spot_tex, Quad.fxh,
- FP16 if you have it
- DXSAS Effect stacking



Dusk's 1935 Debut

Post_holga - noise textures

- **Textures are still the fastest way to get noise in pixel shading**
  - **This noise, at low scales, will also be pretty continuous at a variety of visible sizes**
- **Emulate Optical Distortion by Offseting screen U,V with 2D Noise**
- **Default NOISE2D_SCALE was 500 - we want *much* smoother noise for this application**

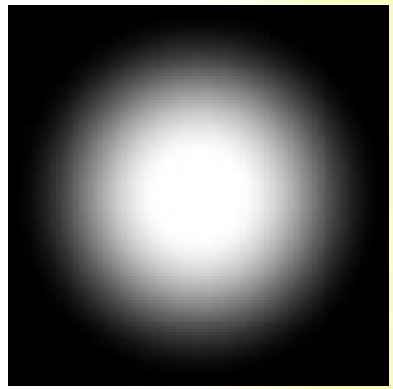
```
#define NOISE2D_SCALE 1
#define NOISE2D_FORMAT "A16B16G16R16F"
#include "noise_2d.fxh"
```

2D Noise

Post_holga - spot_tex

- Using this texture for a different purpose – to isolate distortion to the edges of the frame, and to control the vignetting effect
- Change a couple of defaults to get a different shape

```
#define SPOT_TEX_SIZE 128
#define SPOT_TEX_INSIDE 0.2
#include "spot_tex.fxh"
```



Tweaked spot_tex image

Post_holga - buffering the scene

- **Post_holga (and other postprocess effects) are declared ScriptOrder="postprocess"**
- **We use "ScriptExternal=" to hand-off scene rendering to FX Composer, while using our own texture ("SceneMap") as the scene render target, rather than the framebuffer**

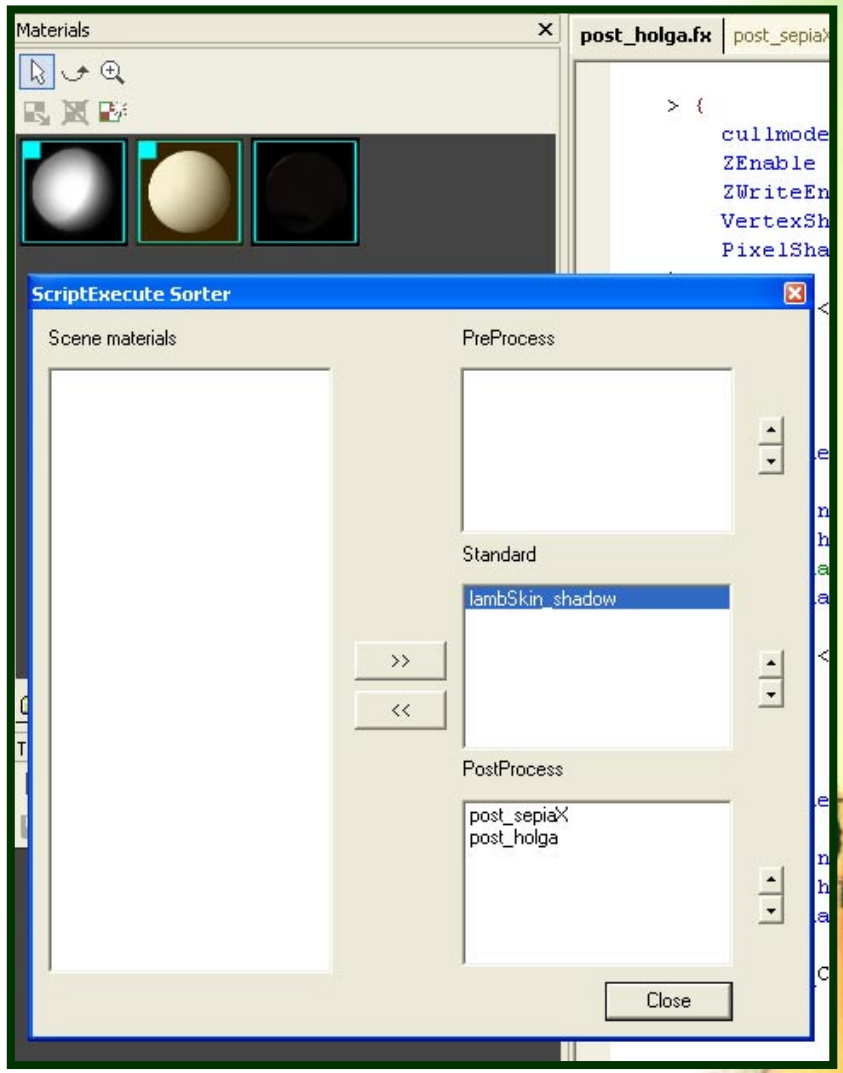
```
string Script = "ClearSetDepth=ClearDepth;"
        "RenderColorTarget=SceneMap;"
        "RenderDepthStencilTarget=DepthMap;"
        "ClearSetColor=ClearColor;"
        "ClearSetDepth=ClearDepth;"
                "Clear=Color;"
        "Clear=Depth;"
        "ScriptSignature=color;"
        "ScriptExternal=;"
        "Pass=DownSample;"
        "Pass=GlowH;"
        "Pass=GlowV;"
        // . . .
```

What do I output?

Render all "below" me

Adding More Shaders to the Scene

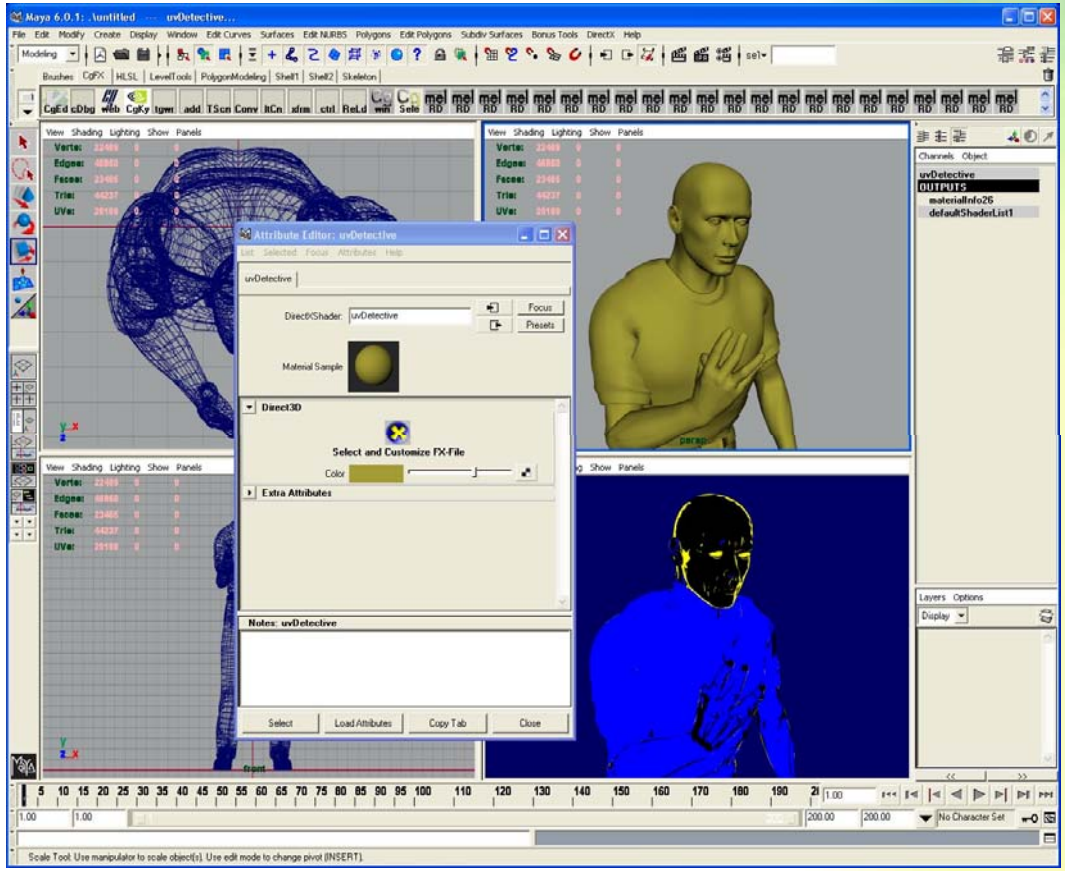
- Use the ScriptExecute Sorter, found in the menu of the Materials Pane
- Build up the look you like
- Maybe reduce to one shader later (maybe not)



The ScriptExecute Sorter

FX Composer & Maya

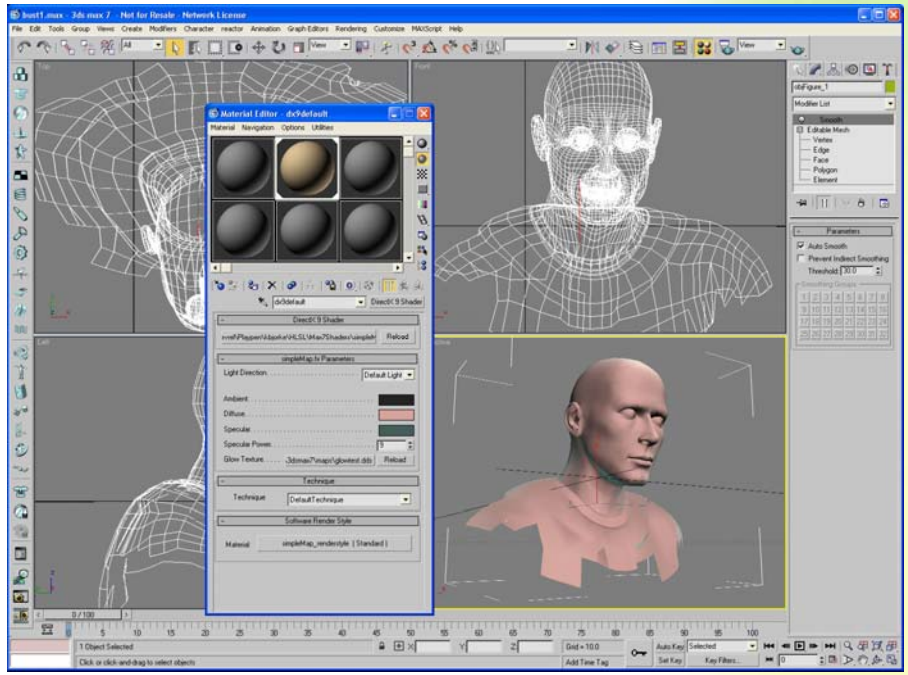
- **Microsoft DX9 Viewer**
  - **Newest in February 2005 DirectX SDK Update**
  - **Special sub-dialog from Attribute Editor**
  - **Maya 6 or Maya 5**
  - **DirectX in Maya window or “floater”**
  - **Integrates .X exporter**



Maya 6.0.1 Model showing “uvDetective”

FX Composer & 3DS Max 7

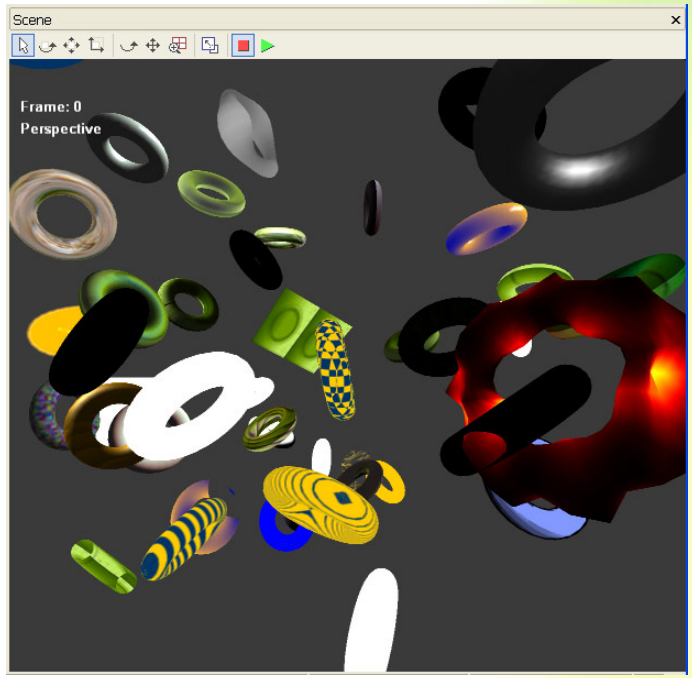
- **3DStudio Max support for DX9 built-in**
  - **Define shaders in Max Materials Pane**
  - **Limited DXSAS support so far**
  - **Which is why we make shadow scripts “smart”**
  - **New NVB exporter from 3DS Max will carry all FX Composer attributes too.**



HLSL Shader in 3DS Max 7

C# Scripting

- Can use C# or Visual Basic, with full text-edit intellisense etc
  - Works off .NET “CLR” so others could work too
- Setting Animation Keys
  - From Programs or External Files
- Creating Objects
  - From Primitives or External Files
  - Can call C++ plugins or work directly
- Cycling Through Shaders and Projects
  - Preview examples like “Scatter_scene.cs”
- Exporting
  - See example “export_material_keys.cs” to access and export all properties of the current scene to XML
- Most FX Composer Internals Are Exposed
  - Use the OLE Viewer in Visual Studio, expand library “nvsys”
    - Data types, structures, and methods are all there



Sample Animated Display from “scatter_scene.cs”

Sample C# Script: “rtzImport.cs”

- Translates app-specific semantics from RTZen Ginza (http://www.rtzen.com/) FX export files into forms most-friendly to FX Composer.
- Creates a tweaked copy of your Ginza shader, then opens it.
- Be sure to include the RTZen path “...\RTShaderGinza\media\images\” in your FX Composer Settings... dialog

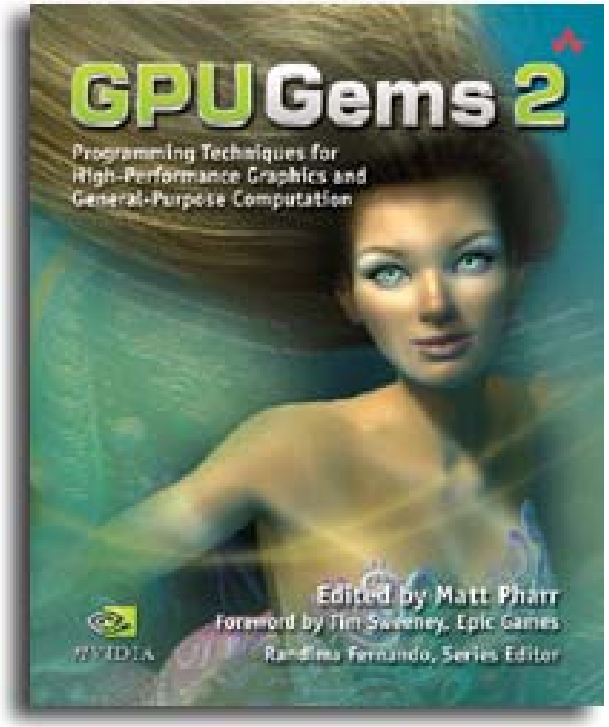
Connecting Outside of FX Composer

- **User-defined annotations and semantics: "…\data\fxmapping.xml"**
- **Geometry Importers & C++ SDK**
- **More!**
  - **Details coming up from Chris Maughan…**
- *Thanks!*

Sepia + Holga + lambSkin_shadow

GPU Gems 2

Programming Techniques for High-Performance Graphi and General-Purpose Computation



- **880 full-color pages, 330 figures, hard cover**
- **$59.99**
- **Experts from universities and industry**

"The topics covered in *GPU Gems 2* are critical to the next generation of game engines."

— Gary McTaggart, Software Engineer at Valve, Creators of Half-Life and Counter-Strike

"*GPU Gems 2* isn't meant to simply adorn your bookshelf—it's required reading for anyone trying to keep pace with the rapid evolution of programmable graphics. If you're serious about graphics, this book will take you to the edge of what the GPU can do."

—Rémi Arnaud, Graphics Architect at Sony Computer Entertainment