Integrating Shaders into Your Game Engine

Bryan Dudash

NVIDIA

Developer Technology

Agenda

- Why shaders?
- What are shaders exactly?
  - Evolution of graphics
- Using Shaders
  - High Level Shading Languages
  - C++ side API and semantics
  - Fallbacks
  - Shader Advice
- Tools

Why Shaders?

- Pixel Shaders are the #1 feature that will visually differentiate next-gen titles
- Distinct materials
  - Great way to show detail without geometry
  - Not everything matte or plastic
  - Moving away from just Blinn/Phong
  - Custom light types
    - Volumetric lights
    - Not limited to OpenGL fixed pipeline

No Shaders vs Shaders

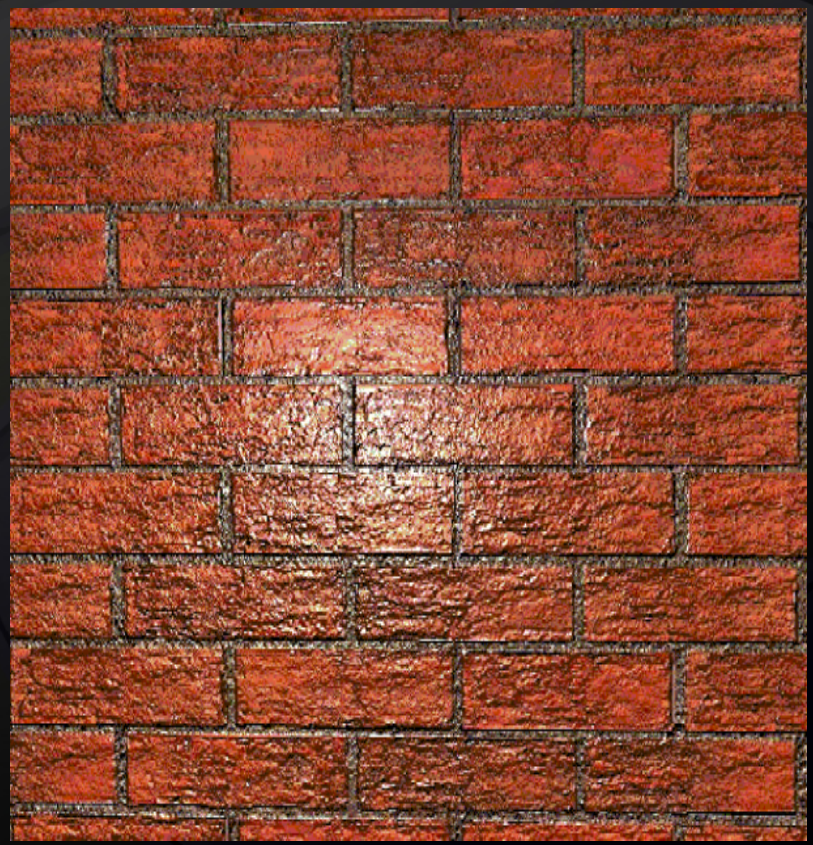
Flat texture, single texture,
vertex lighting, no shadow

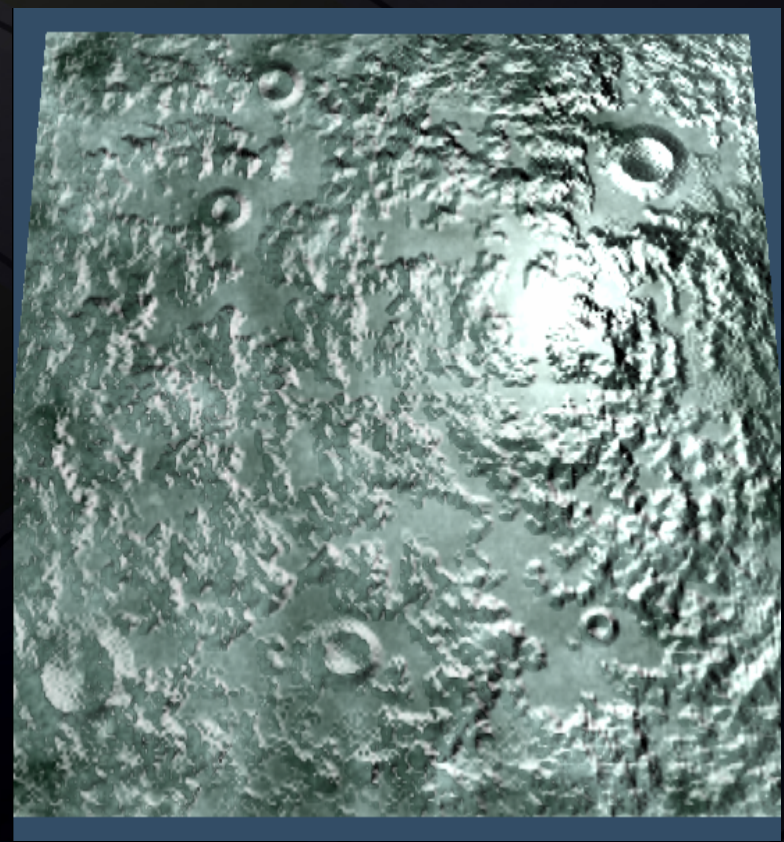
Bump mapped, multi texture,
per pixel lighting, soft shadow

Per Pixel Lighting

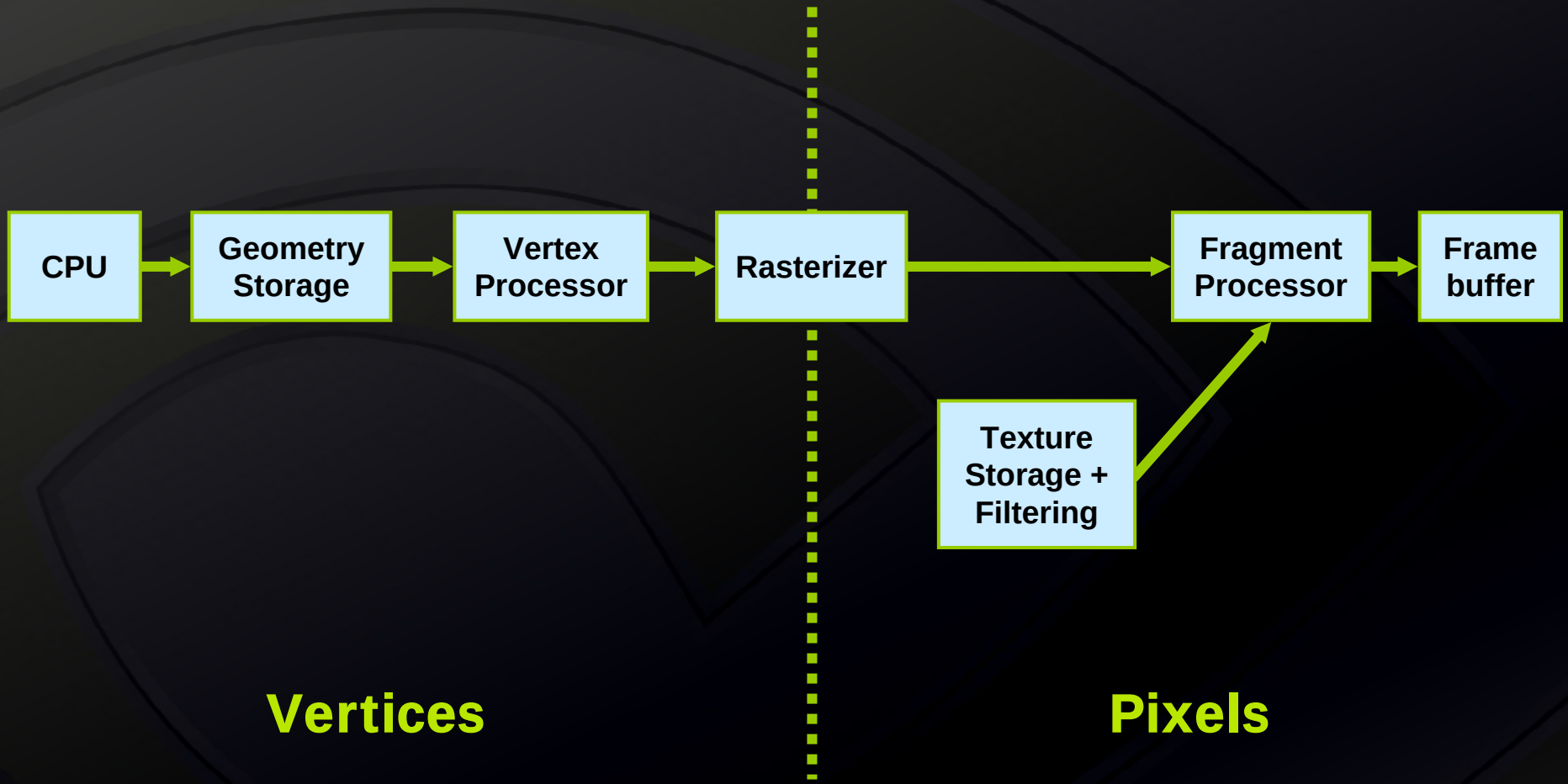
- Bump mapping / Normal Mapping / Per-Pixel Lighting are synonyms
  - Blinn Diffuse Specular lighting
  - With Tangent space Bump mapping
- Instead of calculating lighting on a per-vertex normal, use a per-pixel normal instead

Two quads lit per pixel





Pipelined Architecture



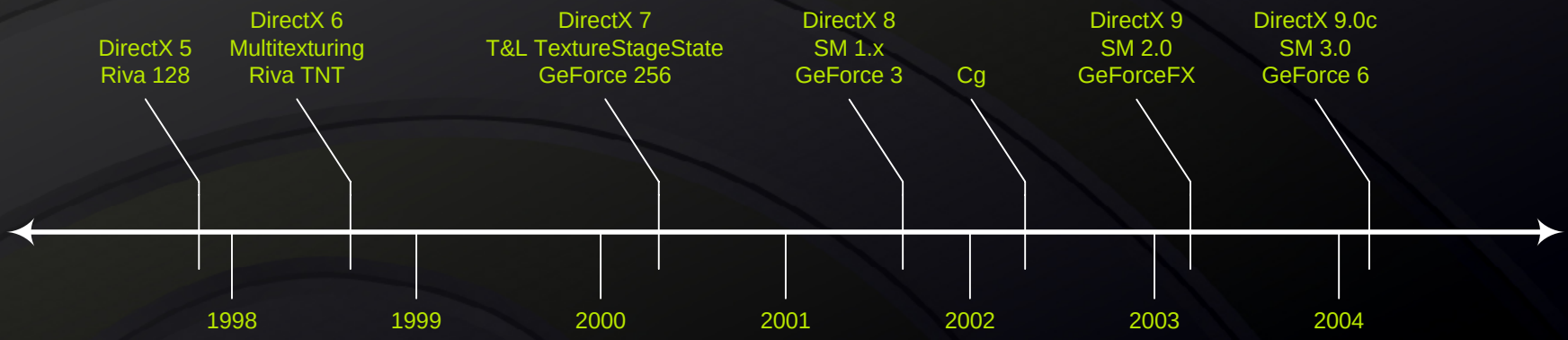
What are Shaders?

- User-defined vertex and fragment processing
  - Custom animation, lighting, image processing, etc.
- Ubiquitous platform & API support
  - PCs, next-generation consoles, cellular phones
  - Direct3D, OpenGL, OpenGL-ES
- Programmed in C-like high level languages
  - HLSL (Direct3D)
  - GLSL (OpenGL)
  - GLSL-ES (OpenGL-ES)
  - Cg (OpenGL, OpenGL-ES)

Shader Taxonomy

- Hardware functionality often described relative to Direct3D shader models 1 – 3
- Newer shader models increase programmability
  - SM 1: Fixed-point color blending, static dependent texturing, <= 16 operations
  - SM 2: Floating-point arithmetic, programmable dependent texturing, <= 64 operations
  - SM 3: Branching & subroutines, 1000s of operations

PC/DirectX Shader Model Timeline



Quake 3

Giants

Halo

Far Cry

UE3

DirectX 5 / OpenGL 1.0 and Before

- Hardwired pipeline
  - Inputs are DIFFUSE, FOG, TEXTURE
  - Operations are SELECT, MUL, ADD, BLEND
  - Blended with FOG

    RESULT = (1.0-FOG)*COLOR + FOG*FOGCOLOR
- Example Hardware
  - RIVA 128, Voodoo 1, Reality Engine, Infinite Reality
- No “ops”, “stages”, or recirculation

DirectX 6, OpenGL 1.2

- Multi-texturing
- Configurable A op B op C pipeline
  - op can be SELECT, ADD, MUL, MAD, SUBTRACT, BLEND
  - Inputs can be CURRENT, DIFFUSE, TEXTURE, or CONST
  - Separate expressions for RGB and Alpha
- Example Hardware
  - Riva TNT, Rage 128, Voodoo 2, Matrox G500

DirectX 6-era Game: Quake 3

- Multi-texturing allows efficient lightmapping, enabling realistic (static) environment lighting
- Vastly improves upon DX5-era DIFFUSE lighting

DIFFUSE Lighting

Multitexture Lightmaps

DirectX 7 / OpenGL 1.3

- Improved Multi-texturing
  - New HW adds DOT3 and EMBM operations
  - New input: SPECULAR
  - Cube maps & projected textures
- Support for HW Transform & Lighting
  - Directional, point, and spot lights.
  - Vertex tweening & skinning
  - Texture coordinate transformation & generation
- Example HW
  - GeForce 256, ATI Radeon, Intel Extreme Graphics 2

DirectX 7-era Game: Giants

- T&L used to radically increase world complexity
- Vertex tweening & skinning improve animation
- DOT3 used to dynamically light everything per-pixel
- Environment mapping adds specular to objects
- Projected texturing used to cast shadow onto ground



DirectX 6 vs DirectX 7

DirectX 8, SM 1.x / OpenGL 1.4

- Programmable vertex shaders
  - Up to 128 floating-point instructions
- Programmable pixel shaders
  - Up to 16 fixed-point vector instructions and 4 textures
  - 3D texture support
  - Up to 1 level of dependent texturing
- Advanced Render-to-Texture support
- Example Hardware
  - GeForce 3, ATI Radeon 8500, XGI Volari V3, Matrox Parhelia

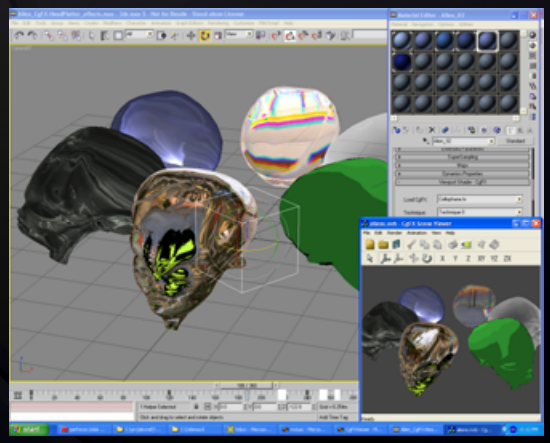
SM 1.x-era Game: Halo

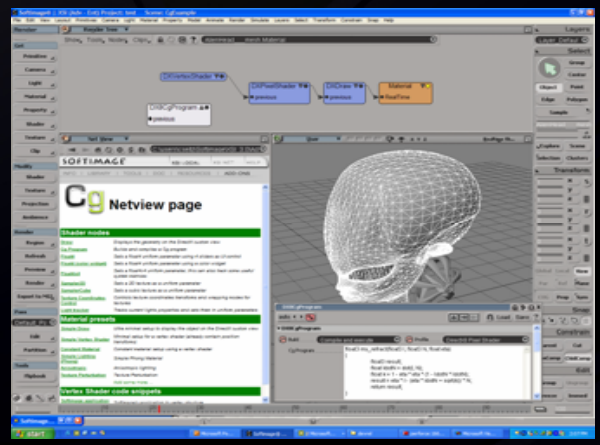
- Vertex shaders used to add fresnel reflection to ice
- Pixel shaders used to add glow to sun
- Render-to-texture used to distort pistol scope
- Dependent texturing used to animate & light water

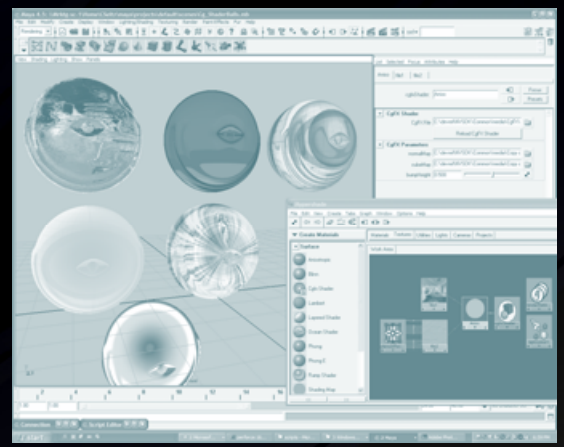
DirectX 7 vs DirectX 8

Cg – C for Graphics

- High-level language designed for real-time shaders
- Supported in major DCC apps (Maya, Max, XSI)
  - What artists see in tool chain matches in-game result





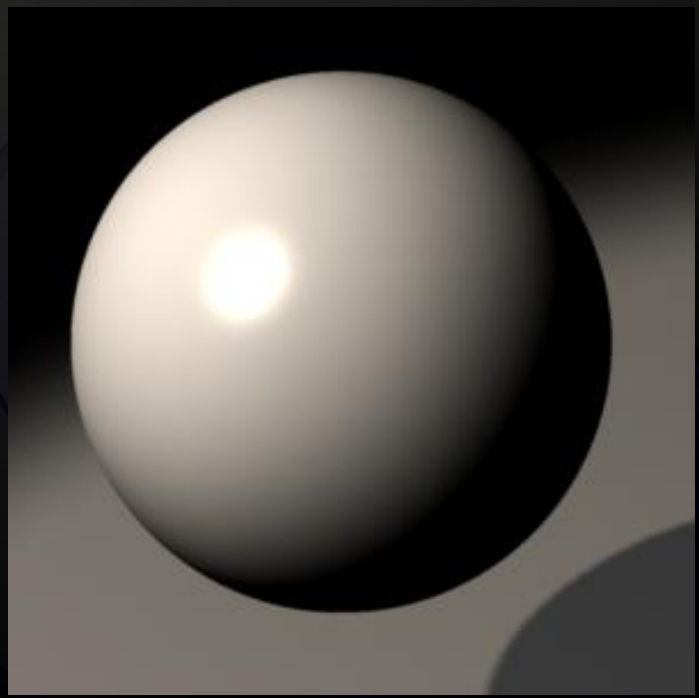


HLL vs Assembly

High-level source code

```
float3 L = normalize(lightPosition – position.xyz);
float3 H = normalize(L + normalize(eyePosition –
                                   position.xyz));

color.xyz = Ke + (Ka * globalAmbient) +
   Kd * lightColor * max(dot(L, N), 0) +
   Ks * lightColor * pow(max(dot(H, N), 0), shininess);
color.w = 1;
```

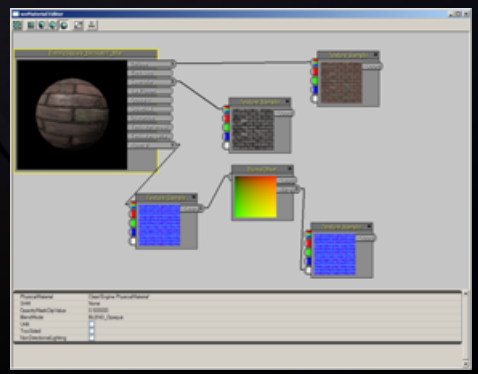


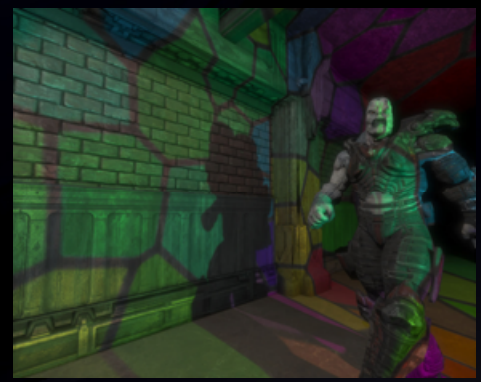
Assembly

```
ADDR R0.xyz, eyePosition.xyzx, -f[TEX0].xyzx;
DP3R R0.w, R0.xyzx, R0.xyzx;
RSQR R0.w, R0.w;
MULR R0.xyz, R0.w, R0.xyzx;
ADDR R1.xyz, lightPosition.xyzx, -f[TEX0].xyzx;
DP3R R0.w, R1.xyzx, R1.xyzx;
RSQR R0.w, R0.w;
MADR R0.xyz, R0.w, R1.xyzx, R0.xyzx;
MULR R1.xyz, R0.w, R1.xyzx;
DP3R R0.w, R1.xyzx, f[TEX1].xyzx;
MAXR R0.w, R0.w, {0}.x;
SLER H0.x, R0.w, {0}.x;
DP3R R1.x, R0.xyzx, R0.xyzx;
RSQR R1.x, R1.x;
MULR R0.xyz, R1.x, R0.xyzx;
DP3R R0.x, R0.xyzx, f[TEX1].xyzx;
MAXR R0.x, R0.x, {0}.x;
POWR R0.x, R0.x, shininess.x;
MOVXC HC.x, H0.x;
MOVR R0.x(GT.x), {0}.x;
MOVR R1.xyz, lightColor.xyzx;
MULR R1.xyz, Kd.xyzx, R1.xyzx;
MOVR R2.xyz, globalAmbient.xyzx;
MOVR R3.xyz, Ke.xyzx;
MADR R3.xyz, Ka.xyzx, R2.xyzx, R3.xyzx;
MADR R3.xyz, R1.xyzx, R0.w, R3.xyzx;
MOVR R1.xyz, lightColor.xyzx;
MULR R1.xyz, Ks.xyzx, R1.xyzx;
MADR R3.xyz, R1.xyzx, R0.x, R3.xyzx;
MOVR o[COLR].xyz, R3.xyzx;
MOVR o[COLR].w, {1}.x;
```

Impact of HLLs

- Dramatic increase in shader adoption
  - Tens of games per year to hundreds
- Shift in game development
  - Shaders become content requirement, not tech feature
  - “What do I want?”, not “what can I do?”
  - Gives control of the look of the game to artists





DirectX 9, SM 2.0 / OpenGL 1.5

- Floating point pixel processing
  - 16/32-bit floating point shaders, render targets & textures
  - Up to 64 vector instructions and 16 textures
  - Arbitrary dependent texturing
- Longer vertex processing – 256 instructions
- Multiple Render Targets – up to 16 outputs per pixel
- Example Hardware
  - GeForce FX 5900, ATI Radeon 9700, S3 DeltaChrome

DirectX 9.0c, SM 3.0 / OpenGL 2.0

- Unified shader programming model
  - Pixel & vertex shader flow control
  - Infinite length vertex & pixel shaders
  - Vertex shader texture lookups
- Floating-point filtering & blending
- Geometry instancing
- Example Hardware
  - GeForce 6800, GeForce 7800 GTX

SM 3.0-era Game: Unreal Engine 3

- 16-bit FP blending for high dynamic range lighting
- 16-bit FP filtering accelerates glow and exposure FX
- Long shaders & flow control for virtual displacement mapping, soft shadows, iridescence, fog, etc.

GPGPU?

- General-purpose Programming Graphics Processing Unit
  - Non-graphics operations on the GPU
- The GPU has is an extremely flexible and powerful processor
  - Programmability
  - Precision
  - Performance
- Example applications (from GPGPU.org)
  - Advanced Rendering: Global Illumination, Image-based Modeling
  - Computational Geometry
  - Monte Carlo Methods
- Many GPGPU Examples in NVIDIA SDK
  - http://developer.nvidia.com/object/sdk_home.html
- GPU Gems 2 Book
  - http://developer.nvidia.com/GPUGems2/

Example: Fluid Simulation

- Navier-Stokes fluid simulation on the GPU
- Interior obstacles
  - Without branching
  - Zcull optimization
- Fast on latest GPUs
  - ~8ms/frame at 256x256 on GeForce 6800 Ultra(includes render
- Available in NVIDIA SDK 8.5

id Dynamics Simulation on the
PU", Mark Harris. In *GPU Gems*.

Using Shaders

“Effects”

- Direct3D FX and CgFX
  - *ID3DXEffect* or *CGeffect*
- Wrapper around pixel and vertex shaders
- Can Configure
  - Target shader version
  - Common case variables
- Can reference a library of shader functions
- Define multi-pass techniques

Semantics

- Define any variable naming you want
  - Semantics make sure constants get set

```
float4x4 wvp      : WorldViewProjection;
```

- D3D SAS is standardized
  - Supported by many applications
    - FX Composer
    - 3D Studio Max
- OpenGL semantics standardized for CgFX in 1.4

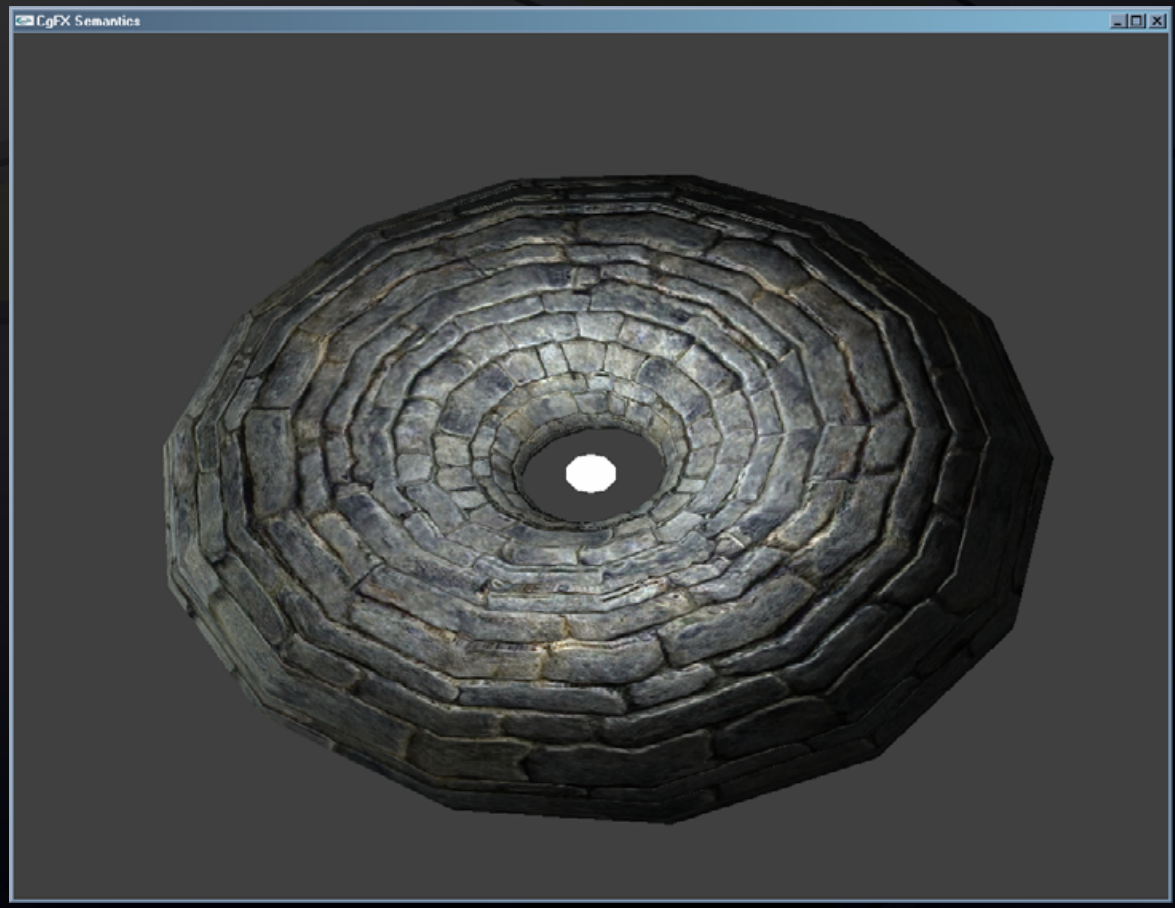
Annotations

- **Custom data associated with any element of your HLSL or CgFX effect**

```
sampler2D anisoTextureSampler <
    string file = "Art/stone-color.png";
> = sampler_state {
        generateMipMap = true;
        minFilter = LinearMipMapLinear;
        magFilter = Linear;
        WrapS = Repeat;
        WrapT = Repeat;
        MaxAnisotropy = 8;
};
```

- **Allows you to provide hooks to set per object data**
  - **E.g. Used extensively by shader tools for UI controls**

CgFX Semantics Demo



Demo Important Bits

- Tangent basis interpolated from vertex shader
- Single fragment shader for lighting
- An unsized array of light structures that is dynamically resized by the C++ side
- A handful of different light types that implement the light interface
  - Point light
  - Spot Light
  - Etc…
- Optional Bump mapping based on a constant

Light Interface

- **Define a CgFX interface to represent a light object**

```
interface Light
{
    LIGHTRES compute(const LIGHTINFOS infos, LIGHTRES prevres);
};
```

- **Then can have the C++ side set whatever, or however many lights it wants to.**
  - **CgFX runtime automatically recompiles**
  - **Alternatively, you can precompile all variations and use directly**

Single Lighting Function

- Sample albedo map for base color
- Normalize interpolated vectors
  - Tangent space basis vectors
- Optionally perturb our normal based on a normal map
- Iterate over our lights and accumulate diffuse and specular
- Combine color and lighting values to produce final result

Lighting Shader

```
float4 color = tex2D(AlbedoMapSampler,inUV);

// Up front normalize to correct interpolated error, skip tangent basis, as the bumped normal gets
normalized
float3 toView = normalize(inToView);
float3 lightDir = normalize(inLightDir.xyz);

// get our bumped normal, or jsut use the original based on the configuration (compiled out)
float3 normal = bumpNormal(inNormal,inTangent,inBiNormal,NormalMapSampler,inUV.xy);;

LIGHTINFOS lightInfo;
lightInfo.Pw = inWorldPos; lightInfo.Vn = toView;
lightInfo.Nb = normal; lightInfo.SpecExpon = sExp;

// Go through our lights
LIGHTRES blinnValues;
for(int i=0;i<aLights.length;i++)
{
        blinnValues = aLights[i].compute(lightInfo,blinnValues);
}

return float4(blinnCombine(color.xyz,ambient.xyz,blinnValues),1);
```

C++ Side

```
CGparameter lArray;
lArray = cgGetNamedEffectParameter(effect,"aLights");
assert(lArray);
cgSetArraySize(lArray,numLights);
for(int i=0;i<numLights;i++)
{
 CGparameter p = cgGetArrayParameter(lArray,i);
 if(p)
 {
        cgDisconnectParameter(p);
        cgConnectParameter(lights[i].handle,p);
 }
}
```

C++ Side

- Assign the light position through the effect given a handle to the variable
- Sets number of lights and light info based on program code dynamically
- Can also pick whether or not to use normal maps
  - When not using it, shader gets faster
- Any dynamic configuration can be represented as a uniform parameter or global constant

Shader Fallbacks

- A Fallback is just a way for your application to control the speed quality tradeoff
- Makes porting much easier!
- Max lights in the scene is a great fallback param
- Can add in and out complex full screen effects
- Regular shadows vs Soft Shadows
- Shader Model differences
  - Maybe same effect, but faster on SM3.0 due to branching.

CPU Side.. How to Deal with Fallbacks

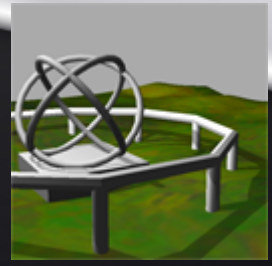
- Need to handle multiple levels of hardware support for different effects
  - Turn on and off effects
- Provide a convenient mechanism to add cooler effects as the game gets close to shipping
- Use the right technique for the right card
  - Just because a card says it supports SM2.0 doesn't mean you want to use the full scale SM2.0 effect,
    - Maybe use SM1 path on low end SM2 cards
  - Use QA testing to define exceptions
- NVIDIA has a CSR test lab to help with this!

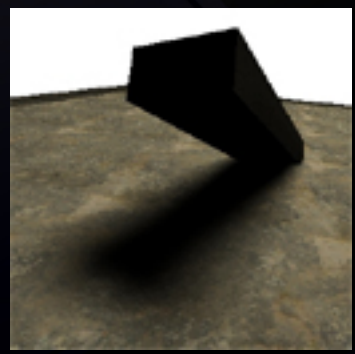
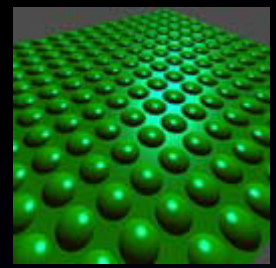
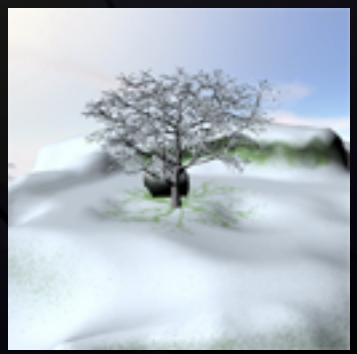
Fallbacks Final Thoughts

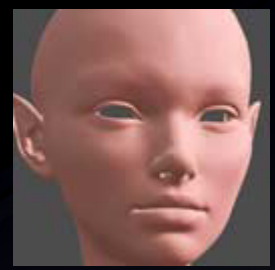
- This can be great to put in early
  - Can easily add a new fallback
    - Reassign cards to a new bucket
  - Have each shader level get the best experience with a decent frame rate
- Use Device Caps!
  - Only override for a driver/hardware bug or user settings
- Create a few common paths
  - Based on Shader model at the coarsest level
  - Possibly just high end and compatibility paths
  - A few tweakables
    - # of lights, samples, etc

Shader Library

- Rather than writing each shader separately
- Code re-use is good!!
- Establish common interpolated values
  - Vertex to Fragment/Pixel program
  - e.g. At a base, COLOR0, TEXCOORD0 off limits
- Create a library of useful functions
  - Break everything out
  - Only costs compile time (can be preprocessed!)



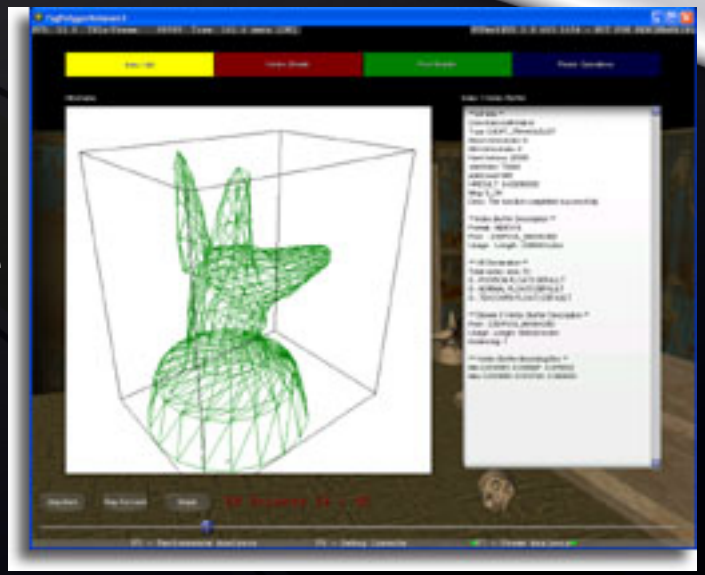



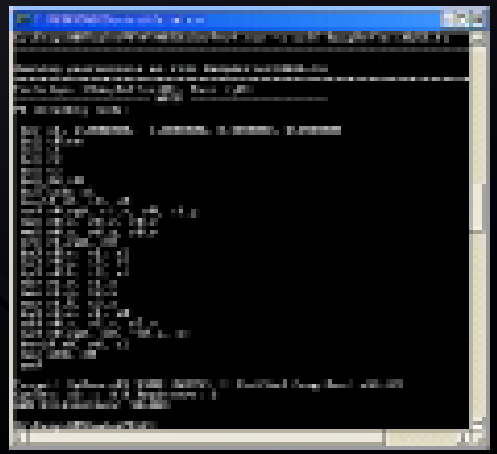

Write with extensibility in Mind

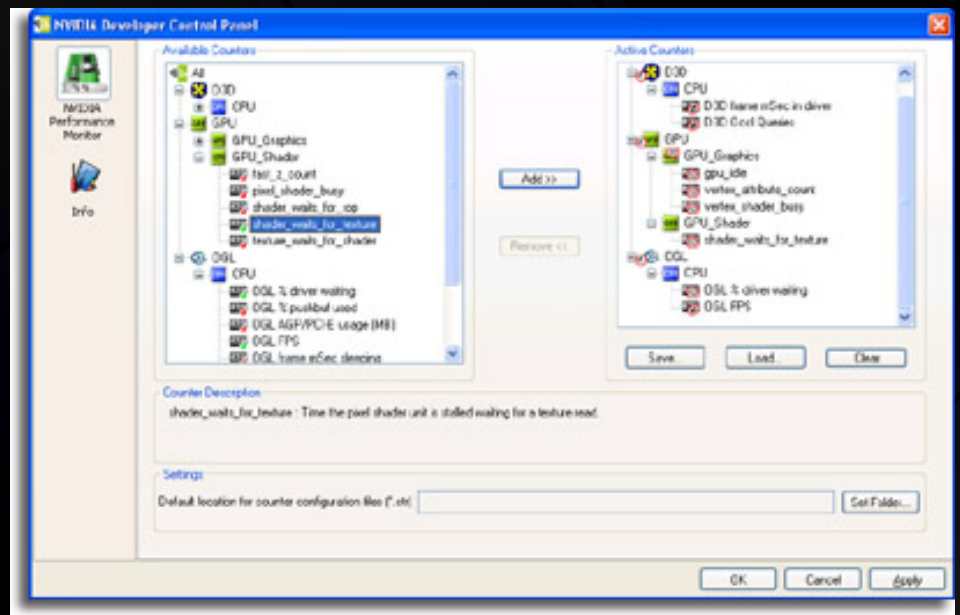
- Quick hacks are for prototyping
- Same as regular code
- Establish guidelines for style
- Full preprocessor support
  - #ifdef #define etc
- Naming convention for techniques
- No Assembly!

Performance

- CPU bound, or Pixel Shader
- NVIDIA's GPU Programming Guide
- NVIDIA provides a number of handy performance analysis tools
  - NVShaderPerf
  - NVPerfHUD
  - NVPerfKit





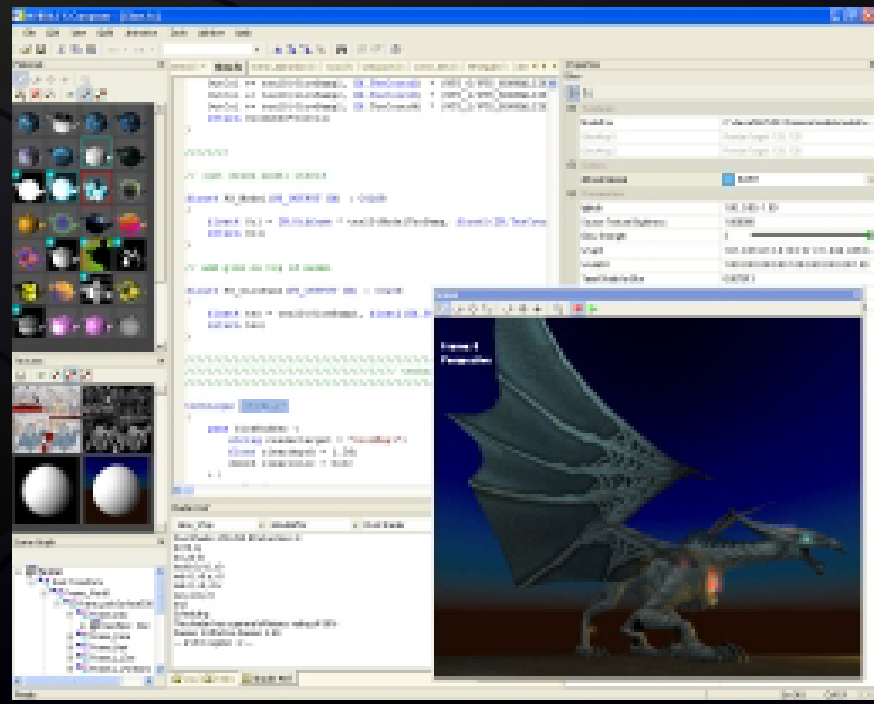


Shader creation pipeline

- Goal
  - Easy creation and integration of shaders
- Requirements
  - Artist tweakable parameters?
  - Technical artist shader creation?
    - Export from DCC tool?

FX Composer

- FX Composer 1.8
  - D3D .fx and SAS
  - Prototype Shaders,
  - Develop the Shader Library
  - Create Fallbacks
  - Export custom parameters
- FX Composer 2.0
  - CgFX support!
- See Tools Session later on today!
- http://developer.nvidia.com



Collada

- XML Interchange format
- Provides a standard way to read and associate shaders and data
- Can even handle model data
  - Vertices, etc
- Sony is working to ensure compatibility with its tools
- FX Composer 2.0 to support

DCC Tools

- Discreet's 3ds Max 5
  - HLSL FX with SAS
- Alias|Wavefront's Maya 4.5
  - CgFx
- XSI's Softimage
  - HLSL FX and CgFX
- Can load a FX files and tweak customizable parameters within the DCC app
- Take those tweaked values and use them to set the standards for the shaders
- Make a model export that associates the shader and customized parameters so that an artist can control how a model looks in game

Conclusion

- Use high-level shading languages
- Use FX files
  - Either CgFX or D3D FX
- Use Semantics
  - Very easy way for you to drop in effects
- Define the art pipeline in advance
  - Understand how artists will interact with and preview shaders
- Treat Shaders like C++ code
  - Good design can save tons of time in making your game look amazing!

Questions

- http://developer.nvidia.com
- http://developer.nvidia.com/CgTutorial
- **Email:** bdudash@nvidia.com