NVIDIA®

NVIDIA Developer Tools

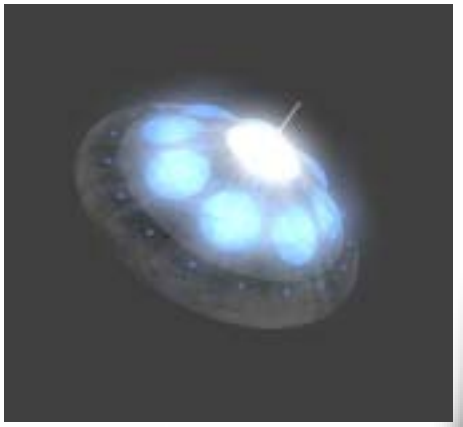
Koji Ashida

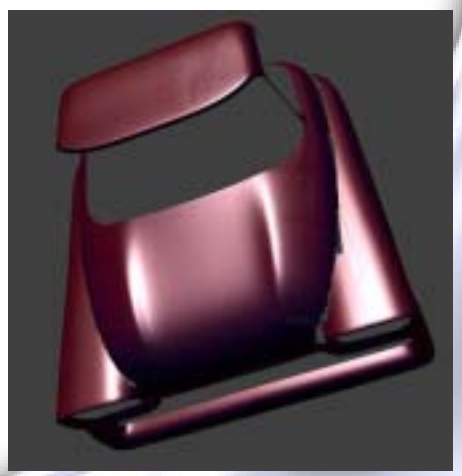
NVIDIA provides many tools

- NVSDK
- Performance tuning tools
- Content creation tools and plugins
  - Melody
  - NVTriStrip
- Creating special effects
  - Shader design and management
  - Shader plugins for DCC apps

NVSDK

- *The* real-time developers source
  - New shaders effects for GeForce FX:
    Skin, Gooch, Car Paint, Glow, Uber, Bicubic Filtering, and much more…
  - Hundreds of effects in DirectX & OpenGL
- Tons of source code:
- Wide and up-to-date distribution
  - 40,000 downloads per release
- Workflow focused: Artist to Coder
- developer.nvidia.com





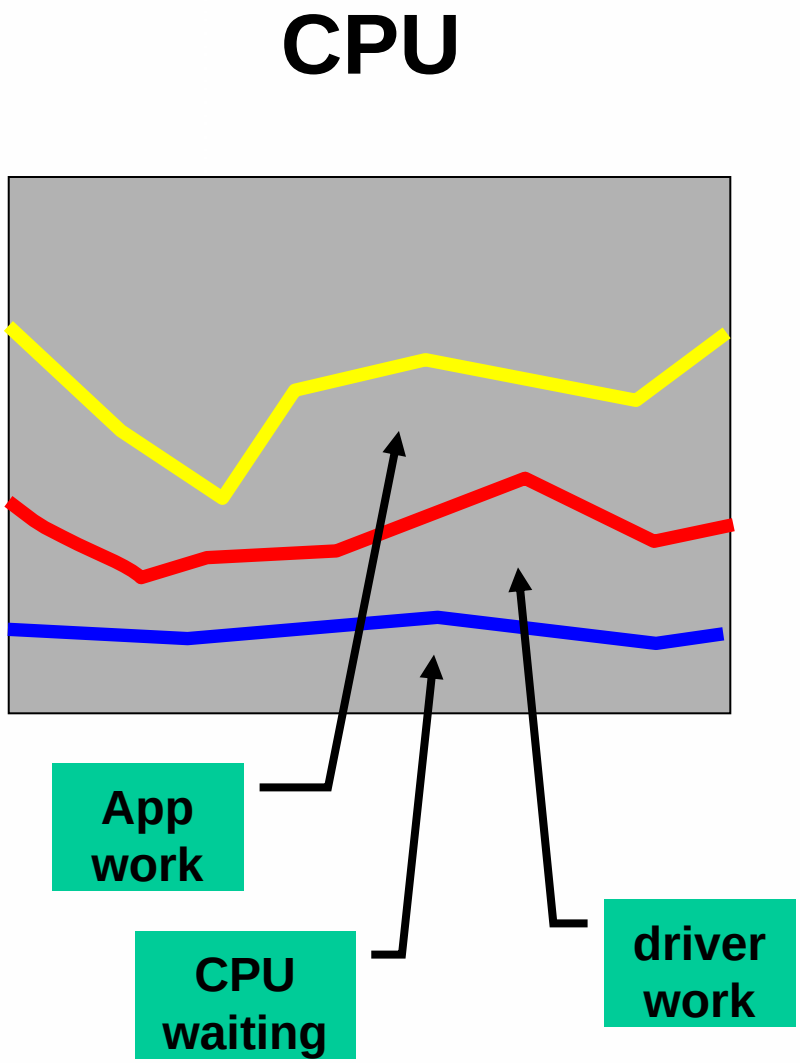
Demo: CgBrowser

Performance tuning: NVPerfHUD

- Drivers now support NVPerfHUD
- Overlay that shows vital various statistics as the application runs
- Top graph shows :
  - Number of API calls – Draw*Prim*, render states, texture states, shader states
  - Memory allocated – AGP and video
- Bottom graph shows :
  - GPU Idle – Graphics HW not processing anything
  - Driver Time – Driver doing work (states and resource management, shader compilation)
  - Driver Idle – Driver waiting for GPU to finish
  - Frame Time – Milliseconds for frame time

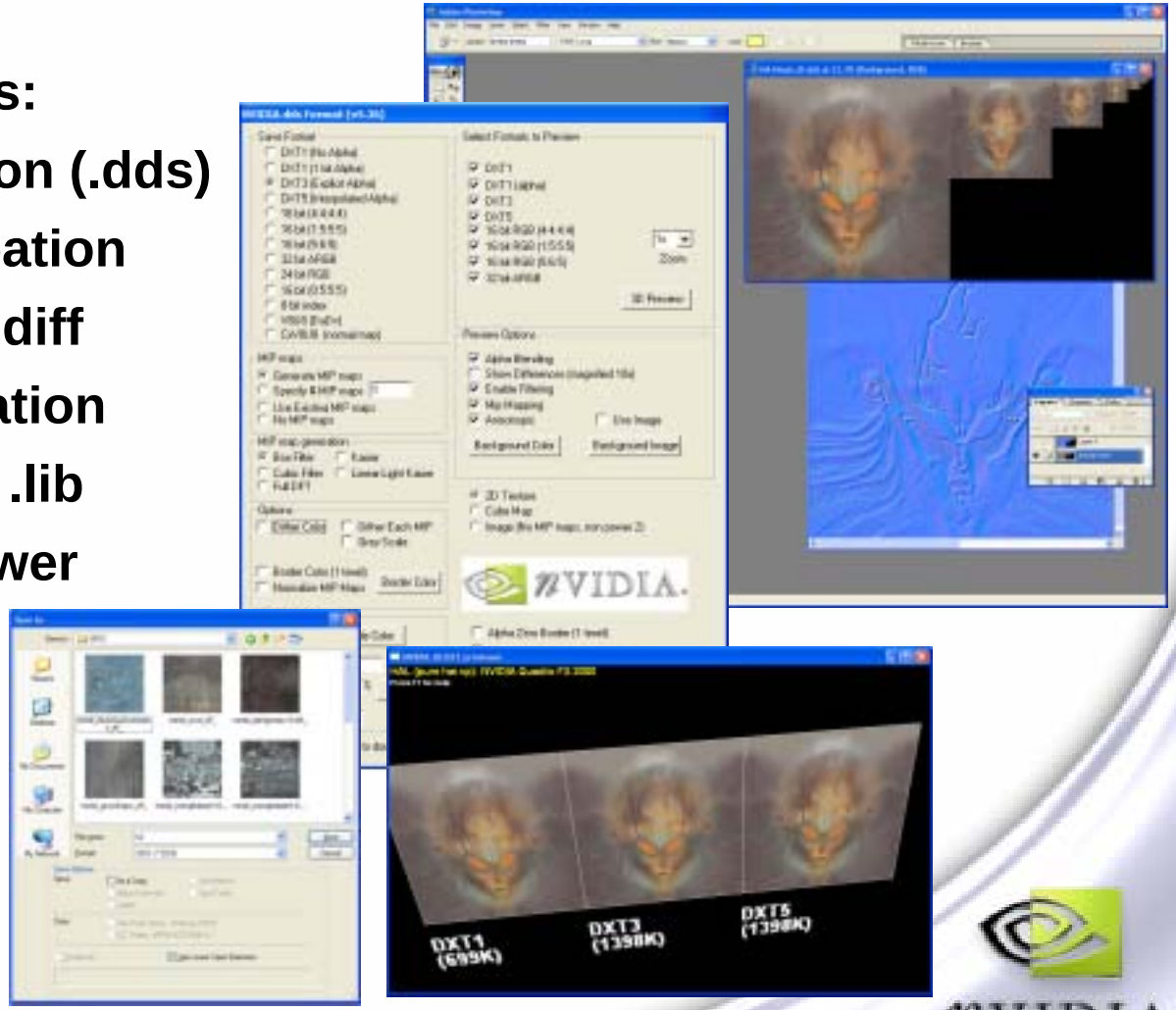

fps: 014
tris/frame: 69215
AQUAMARK
3
2000
1500
1000
500
0
Number of batches
100 ms
75 ms
50 ms
25 ms
256 Mb
192 Mb
128 Mb
64 Mb
0 Mb
Ms per frame
Driver Time
CPU waits for GPU
GPU Idle
VID Mem
AGP Mem
Complex Multimaterial Shader

NVPerfHUD: CPU and GPU usage



Texture tools & plugins

- Photoshop Plug-ins:
  - DXT compression (.dds)
  - Normal Map creation
  - 3d preview and diff
  - MIP map generation
- Command line and .lib
- DDS thumbnail viewer



NVIDIA.

Demo: Melody

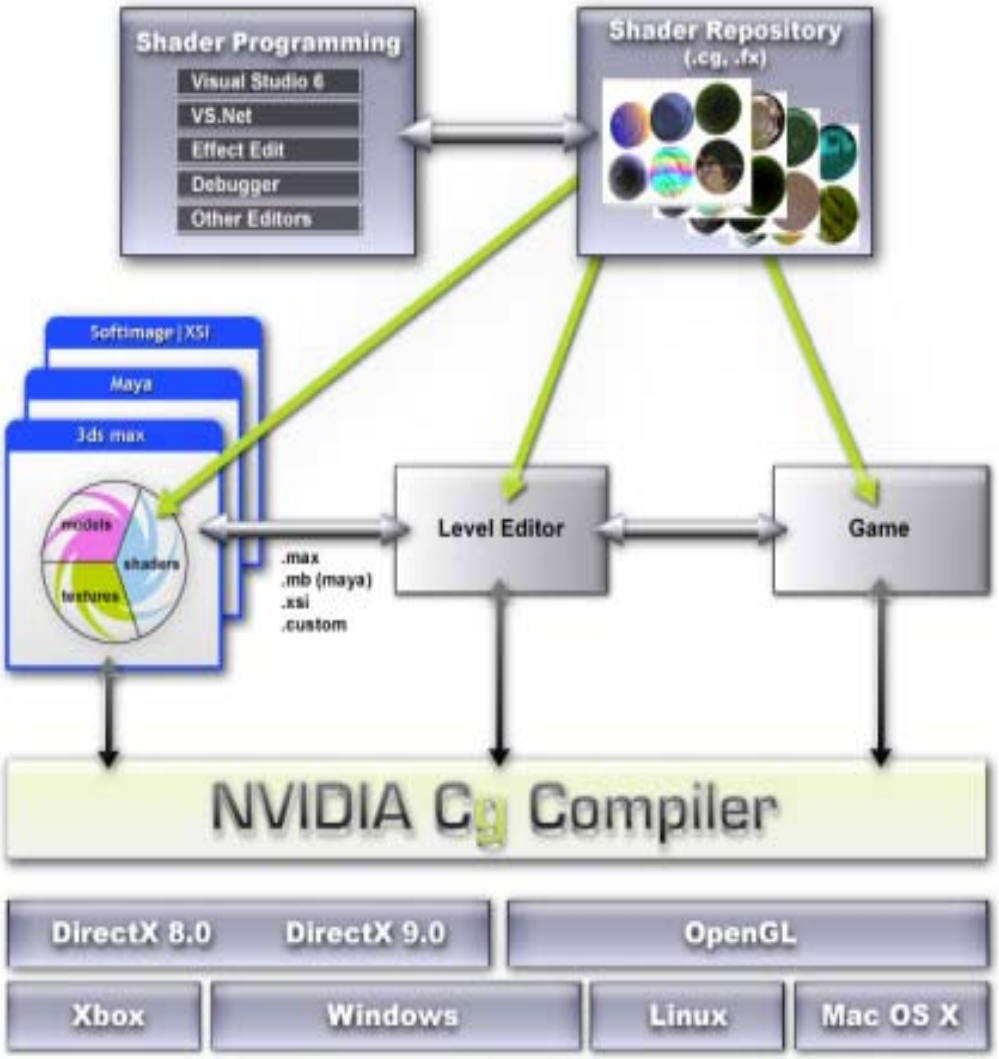
Shader development: FX Composer

- Integrated IDE for HLSL FX development
- Simulated shader scheduling on nv3x family
- Disassembly of vertex and pixel shaders
- Bakes textures from HLSL code
- Allows render to texture effects
- HLSL Intellisense
- Allows scene import from .x and .nvb files
- Supports animation, lights, skinned meshes, etc...
- Allows pluggable geometry modifiers (fins, ...)
- Project files .fxcomposer
- Fxmapping.xml – custom semantic/annotation mapping

Demo: FX Composer UI components

How FX (effect) works

- **Shaders** are small programs designed to control how GPUs transform geometry in space, and how the pixels of that rasterized geometry are colored.
- FX is applicable to every stage in the creation and use of digital pictures:
  - Modeling
  - Texturing
  - Animation
  - Level Design
  - Game Engines
  - Rendering



FX: a complete shading language

- A simple way to unify vertex/fragment shaders into complete appearances
- Can be used at all stages of production
  - Already integrated into the most-popular DCC apps
- Multiple render passes supported
- Techniques accommodate different hardware devices with distinct rendering capabilities
- Simple text file for easy process management

FX file structure

- FX files look similar to programs
- Each FX contains:
  - User "Tweakables"/Tracking Declarations: User variables & UI hints
  - Other global declarations
  - Vertex & Pixel Shaders, declared as functions
  - Techniques to encapsulate Shaders, Tweakables, Render Passes & Graphics-State Settings.

Demo: a skeleton FX file

FX example – tweakables

```
float4 lightPos : Position
<
    string Object = "PointLight";
    string Space = "World";
> = {100.0f, 100.0f, 100.0f, 1.0f};

float lightIntensity
<
    string gui = "slider";
    float uimin = 1.0;
    float uimax = 10000.0;
    float uistep = 1.0;
    string Desc = "lamp power";
    float min = 0.0;
    float max = 10000.0;
> = 10.0;
```

- **: Semantics give applications hints for automatic binding**
- **<Annotations> give additional application-specific UI hints**
- **Both Semantics and Annotations are optional**

FX example – “un-tweakables”

- Some universal “tweakables” can be automatically tracked by FX
- We don’t let the user tweak them, so they’re “untweakables”

```
float4x4 worldIT : WorldIT;
float4x4 wvp : WorldViewProjection;
float4x4 world : World;
float4x4 viewIT : ViewIT;
```

FX example – application-only globals

- Unused by the shaders themselves, these values can be used by applications for cataloging, defining UI's, etc.

```
string description = "Shader Template";
string Category = "Template";
string keywords =
  "bumpmap,texture,glossmap,fresnel";
```

FX example – shaders as functions

```
vertexOutput basicVS(appdata IN,
    uniform float4x4 WorldViewProj,
    uniform float4x4 WorldIT,
    uniform float4x4 World,
    uniform float4x4 ViewIT,
    uniform float3 LightPos
) {
    vertexOutput OUT;
    OUT.WorldNormal = mul(WorldIT, IN.Normal).xyz;
    float4 Po = float4(IN.Position.xyz,1.0);
    float3 Pw = mul(World, Po).xyz;
    OUT.PtLightVec = LightPos - Pw;
    OUT.TexCoord = IN.UV.xy;
    OUT.WorldView = normalize(ViewIT[3].xyz - Pw);
    OUT.HPosition = mul(WorldViewProj, Po);
    return OUT;
}
```

FX example – technique

```
technique Main
{
   pass p0
   {
        VertexShader = compile arbvp1 basicVS(wvp,
               worldIT,world,viewIT,lightPos);
        ZEnable = true;
        ZWriteEnable = true;
        CullMode = None;
        PixelShader = compile arbfp1 basicPS(colorSampler,
               envSampler,diffStrength,
               specStrength,specExpon,metalness,
               reflStrength,reflMin,fresExp,
               ambiLightColor,surfColor,
               lightColor,lightIntensity);
   }
}
```

- Profiles
- Can also use asm{…}
- Render State

Demo: blue circle

- Simply projection transform the vertices
  - Position = {x, y, z, w} * Winv * V * P
- Make pixel color blue
  - Color = {0.0, 0.0, 1.0}
- Sphere looks circle

Demo: per pixel phong shading

- Same computation for vertices
- Phong color model
  - Color = C * (L . n + (H . n)^S)
  - C – surface color
  - L – light direction
  - n – surface normal
  - H – half angle (E + L) / 2 where E is eye direction
  - S – specular constant
- This is usually done per vertex for efficiency

Demo: using tweakables

- Make a lighting parameter tweakable

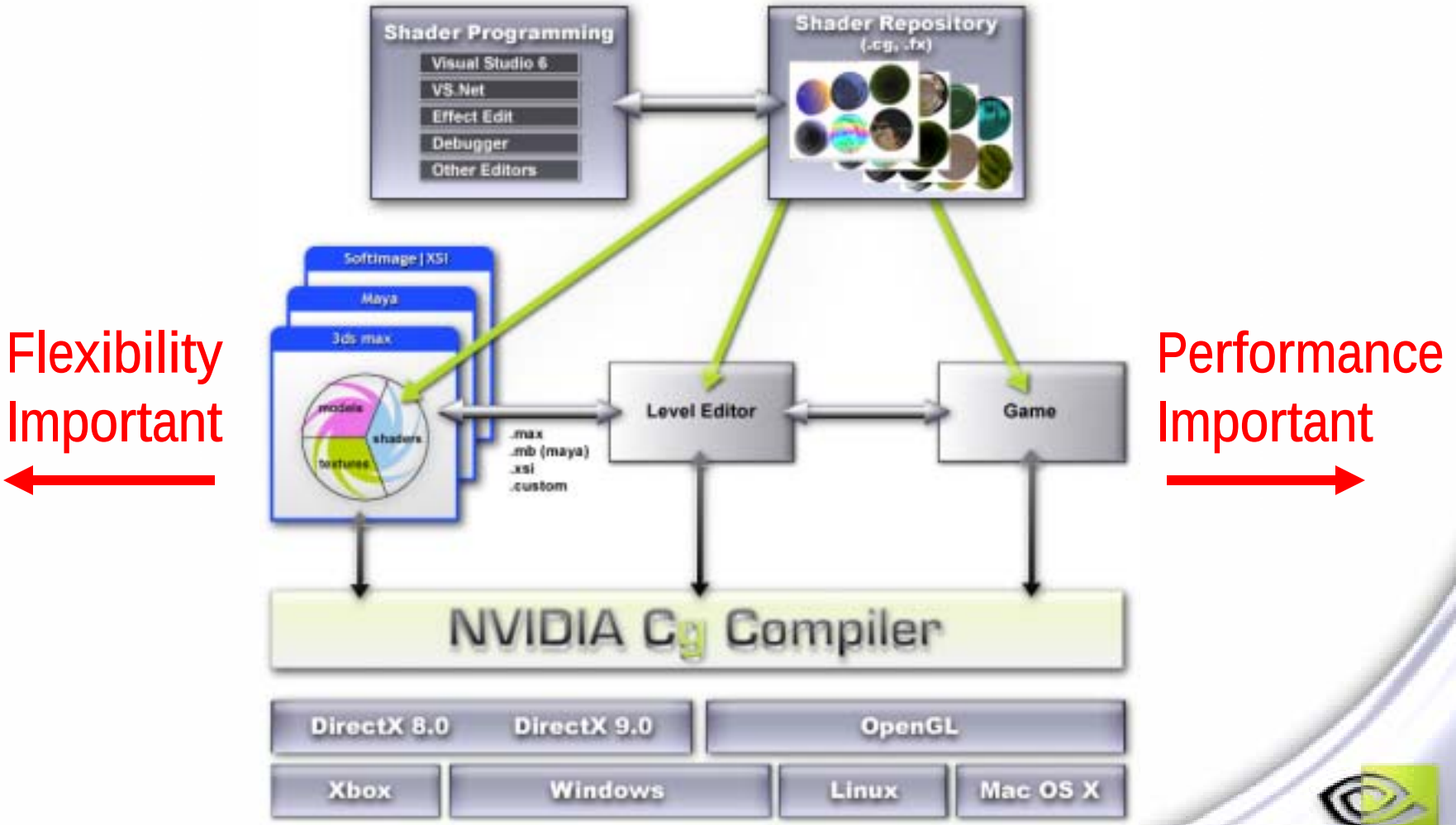
Demo: multi-pass rendering

- Pass 2 with blending enabled

Demo: moving object

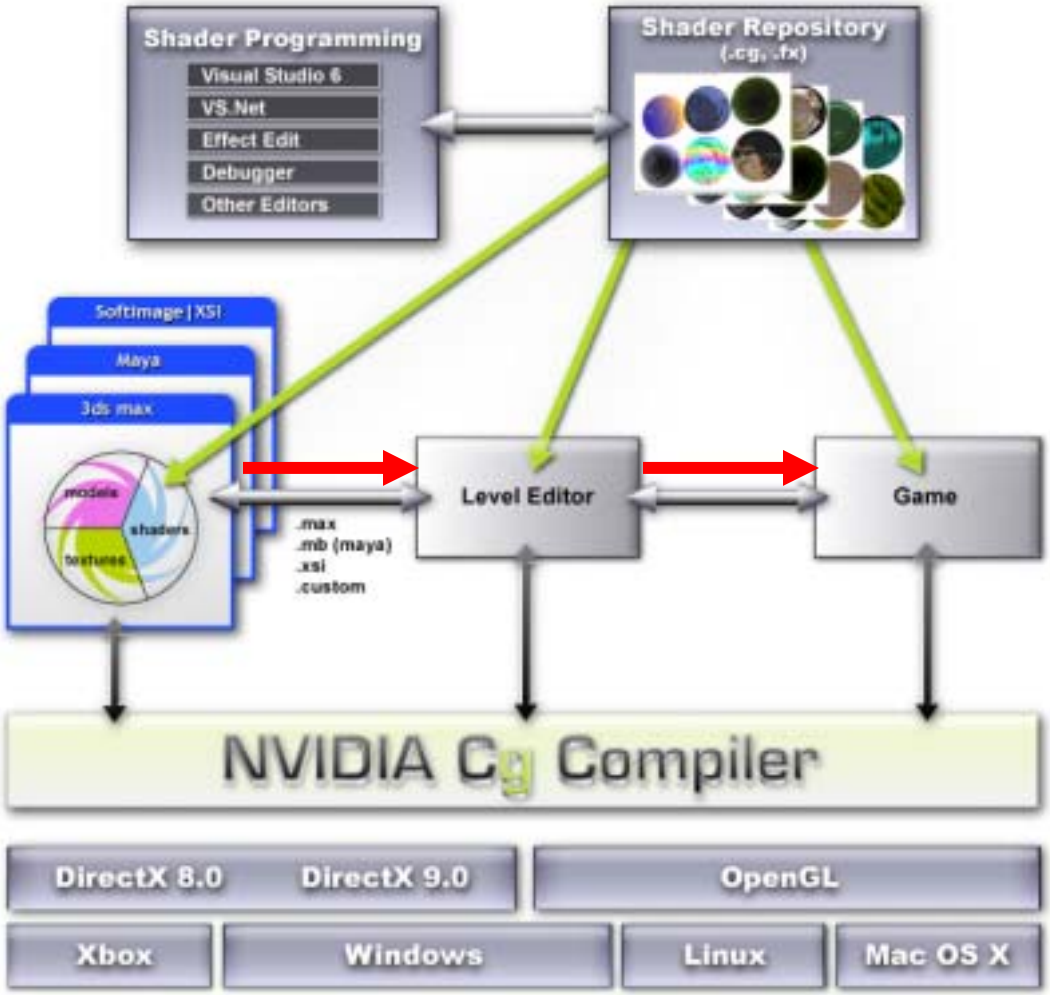
- Using ‘Time’ to control vertex positions

Alternate FX workflows



FX workflow 1: artist control

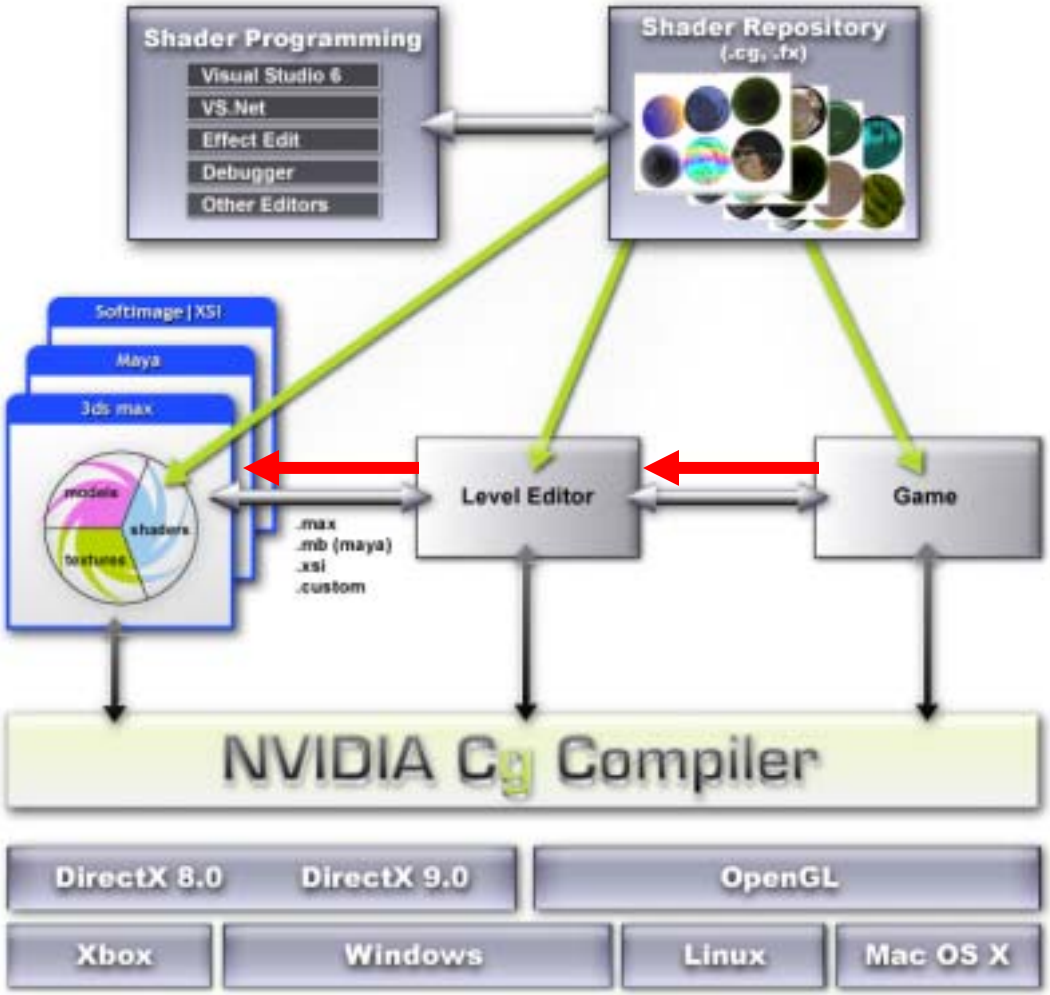
Artist-Defined Shaders



Artist-controlled workflow

- Artists can choose or create shaders appropriate for their art assets
- Shaders can then be used “downstream” (as FX, integrated HLSL, or compiled assembly code)
- Final product looks just as the artists intended

FX workflow 2: programmer control



Programmer-Defined Shaders

Programmer-controlled workflow

- Shaders are defined for performance, specific lighting models, game-console compatibility, simulation of real-world materials, etc.
- HLSL or assembler code is incorporated into FX and passed “upstream” to artists
- Artists can then design specifically to the hardware/performance demands, and see the final result accurately in their design applications
- Artists can also design art assets for multiple game delivery target simultaneously, via techniques: Xbox, Playstation, fast or slow PCs

Flexible runtime compilation (FX)

At Development Time

shader program source code

```
//
// Diffuse lighting
//
float d = dot(normalize(frag.N),
normalize(frag.L));
if (d < 0)
  d = 0;
c = d*tex2D(t, frag.uv)*diffuse;
…
```

At Runtime

- **At initialization:**
  - **Compile and load HLSL program**
- **For every frame:**
  - **Load program parameters with the Shader Runtime API**
  - **Set rendering state**
  - **Load geometry**
  - **Render**

Compiling offline

At Development Time

HLSL or FX source code

Shader Compiler

Shader program assembly code

Shader Compiler (nvasm.exe, psa.exe)

Shader program binary code

```
//
// Diffuse lighting
//
float d = dot(normalize(frag.N),
normalize(frag.L));
if (d < 0)
  d = 0;
c = d*tex2D(t, frag.uv)*diffuse;
…
```

```
…
DP3 r0.x, f[TEX0], f[TEX0];
RSQ r0.x, r0.x;
MUL r0, r0.x, f[TEX0];
DP3 r1.x, f[TEX1], f[TEX1];
RSQ r1.x, r1.x;
MUL r1, r1.x, f[TEX1];
DP3 r0, r0, r1;
MAX r0.x, r0.x, 1.0;
MUL r0, r0.x, DIFFUSE;
TEX r1, f[TEX1], 0, 2D;
MUL r0, r0, r1;
…
```

```
012b40  00 00 00 00 00 00 00 00
012b50  42 CD 09 84 51 3F 84 3C
012b60  93 AB D9 B1 07 87 70 B2
012b70  5C A5 D1 1C 58 65 58 F4
012b80  1F 27 1F 22 22 1F 1F 22
012b90  12 22 22 12 22 22 22 22
012ba0  1F 2F 2F 2F 2F 2F 23 FF
012bb0  37 37 37 37 37 37 2C 2C
012bc0  30 BE 47 04 4A BE A8 E6
012bd0  50 78 92 DD 90 B9 72 CE
012be0  03 69 8D EE 46 73 85 F9
```

At Runtime

- **At initialization:**
  - **Load**

    assembly or binary program
- **For every frame:**
  - **Load program parameters to hardware registers**
  - **Set rendering state**
  - **Load geometry**
  - **Render**

NVIDIA.

Pros and cons of runtime compilation

- **Pros:**
  - **Future compatibility: The application does not need to change to benefit from future compilers (future optimizations, future hardware)**
  - **Easy parameter management**
  - **Game players can “mod” the results**
- **Cons:**
  - **Loading takes more time because of compilation**
  - **Cannot hand-tune the result of the compilation**
  - **Text files may be insecure**
  - **API State is fixed (for FX files)**

Summary

- NVIDIA provides various tools for game production
- Using FX Composer, you can:
  - Quickly prototype and experiment with shader ideas
  - Divide shader development work between programmer and artists using tweakables
- FX files help you manage your effects as art assets

Questions, comments, feedback?

- Koji Ashida <kashida@nvidia.com>
- http://developer.nvidia.com