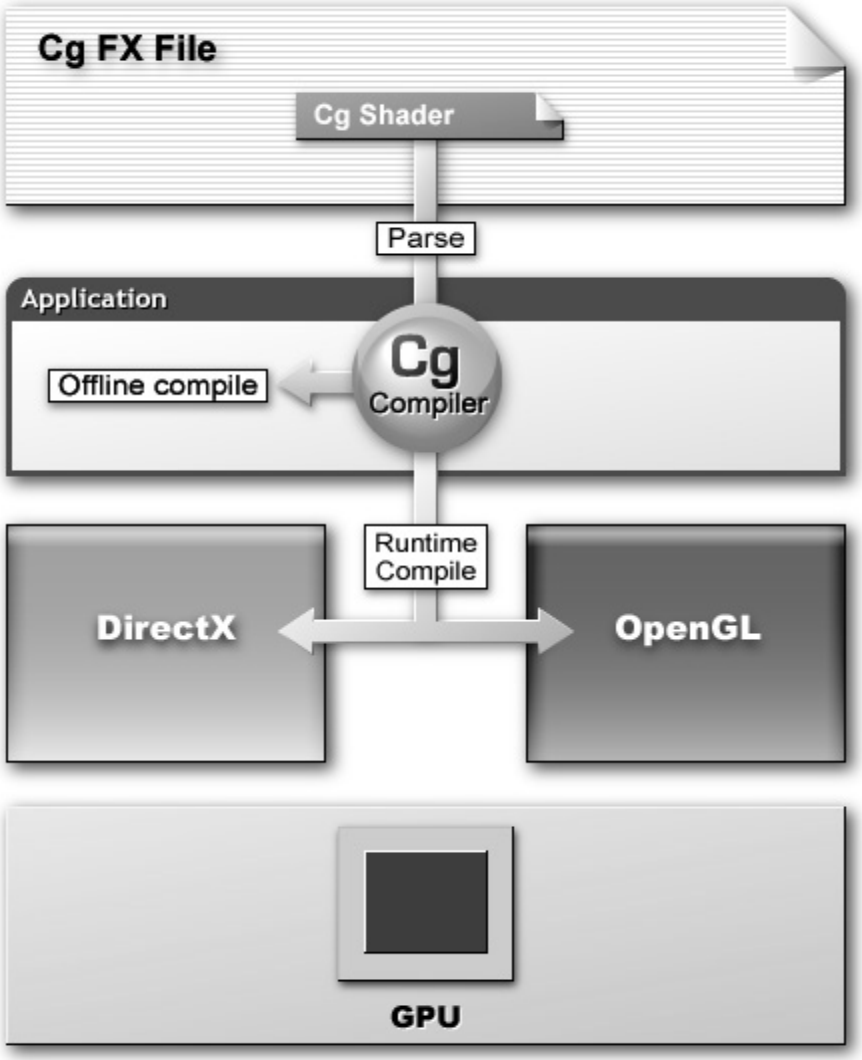
Cg

Kevin Bjorke

GDC 2003

A Whole New World with Cg



Graphics Program Written in Cg "C" for Graphics

Compiled & Optimized

Low Level, Graphics "Assembly Code"

Cg is a C-like language, modified for GPUs

- Syntax, operators, functions from C
- Conditionals and flow control
- Particularly suitable for GPUs:
  - Expresses data flow of the pipeline/stream architecture of GPUs (e.g. vertex-to-pixel)
  - Vector and matrix operations
  - Supports hardware data types for maximum performance
  - Exposes GPU functions for convenience and speed:
    - Intrinsic: (mul, dot, sqrt…)
    - Built-in: extremely useful and GPU optimized math, utility and geometric functions (noise, mix, reflect, sin…)
  - Language reserves keywords to support future hardware implementations (e.g. switch, case…)
  - Compiler uses hardware profiles to subset Cg as required for particular hardware feature sets

NVIDIA Cg Usage

- Three ways:
- Cg Runtime – a thin API:
  - ASCII .Cg shader compiled at runtime
  - Convenient interface for setting shader parameters and constants
- Command line compiler generates text file output:
  - DX / OpenGL – vertex and pixel shader files
  - Tweak the ASM yourself
  - Generates comments on program params & registers
- CgFX
  - Effect framework with render states

Integrating Cg

- Options (not mutually exclusive):
  - CgFX
    - Manages whole rendering process
    - Handles render states – cross API support
    - Convenient exposure of tweakables & artist controls
  - Cg Shaders
    - semantics (was #pragma bind) directives to match your C++ and other custom hardware shaders
    - Bind textures/parameters to specific HW registers
  - Cg Runtime
    - Thin API to compile on demand at runtime
    - Optimizes & munges .Cg for range of target HW

Flexible Adoption Path

- Can use system for just fragment programs
  - Define a connector to specify how you'll supply data to these programs
- Can use system for just vertex programs
  - Define connectors to specify how you'll supply data to these programs; and what they have to output.
- Can use system with older OpenGL applications
  - Classical `glVertex()`, `glNormal()` can still be used
  - Use OpenGL matrix tracking to provide modelview matrix to shading program.
  - But, must load program & supply light state

Mix and Match Any Method

Vertex processing	Fragment processing
Fixed function	Fixed function
-or-	-or-
Hand-written ASM	Hand-written ASM
-or-	-or-
Compiled Cg	Compiled Cg
-or-	-or-
Hand-optimized Cg ASM	Hand-optimized Cg ASM

Using the Cg Compiler

Application Development

Cg program source code

```
//
// Diffuse lighting
//
float d = dot(normalize(frag.N),
normalize(frag.L));
if (d < 0)
  d = 0;
c = d*f4tex2D(t, frag.uv)*diffuse;
…
```

Cg Compiler

Shader program assembly code

```
…
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;
…
```

Shader Compiler (nvasm.exe, psa.exe)

Shader Binary

```
012b40  00 00 00 00 00 00 00 00
012b50  42 CD 09 84 51 3F 84 3C
012b60  93 AB D9 B1 87 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
```

Your Application

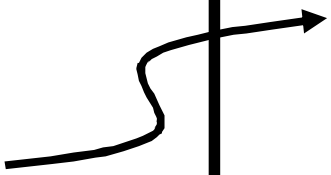
1) Load/bind program
2) Specify program parameters
3) Specify vertex inputs
4) Render

Using the Cg Runtime

Application Development

Cg program source code

```
//
// Diffuse lighting
//
float d = dot(normalize(frag.N),
normalize(frag.L));
if (d < 0)
  d = 0;
c = d*f4tex2D(t, frag.uv)*diffuse;
…
```



Your Application

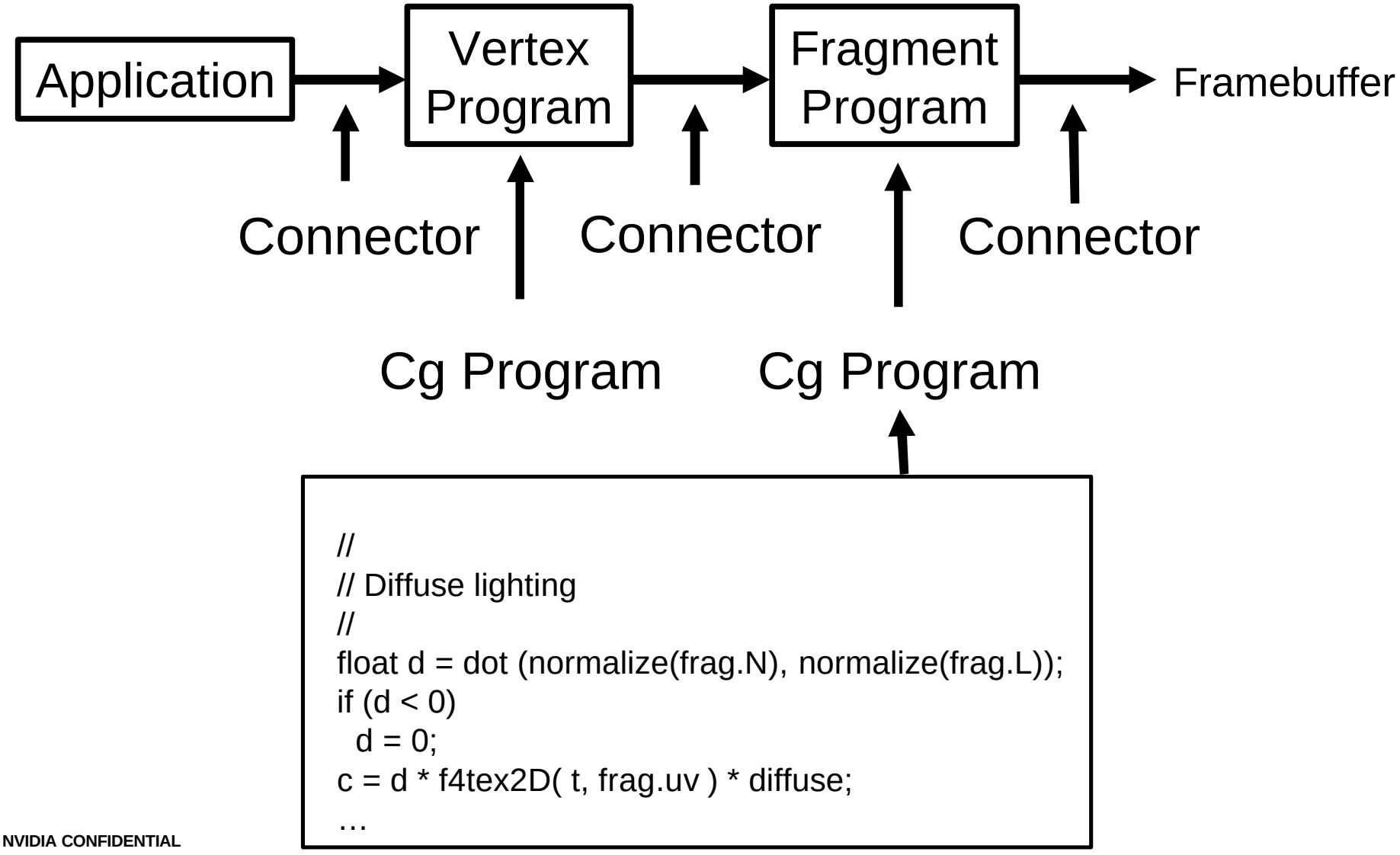
Cg Runtime

1) **Load/bind program**
2) **Specify parameters**
3) Specify vertex inputs
4) Render

Details

And now for the details...

Graphics Data Flow



Data types

- **float = 32-bit IEEE floating point**
- **half = 16-bit IEEE-like floating point**
- **fixed = 12-bit fixed [-2,2) clamping (*OpenGL only*)**
- **bool = Boolean**
- **sampler* = Handle to a texture sampler**

Array / vector / matrix declarations

- Declare vectors (up to length 4) and matrices (up to size 4x4) using built-in data types:

```
float4   mycolor;
float3x3 mymatrix;
```

- Declare more general arrays exactly as in C:

```
float lightpower[4];
```

- But, arrays are first-class types, not pointers
- Implementations may subset array capabilities to match HW restrictions

Extend standard arithmetic to vectors and matrices

- Component-wise + - * / for vectors
- Dot product
  - `dot(v1,v2);` // returns a scalar
- Matrix multiplications:
  - assuming `float4x4 M` and `float4 v`
  - matrix-vector: `mul(M, v);` // returns a vector
  - vector-matrix: `mul(v, M);` // returns a vector
  - matrix-matrix: `mul(M, N);` // returns a matrix

New vector operators

- Swizzle operator extracts elements from vector

```
a = b.xxyy;
```

- Vector constructor builds vector

```
a = float4(1.0, 0.0, 0.0, 1.0);
```

“Profiles” Define Specific HW Behavior

- Public NVIDIA Cg Compiler has three NV2X profiles:
  - DX8 Vertex Shader (vs1.1)
  - DX8 Pixel Shader (ps1.1)
  - OpenGL Vertex Program (currently based on NV_vertex_program, will move to ARB_vertex_program)
- Newest NVIDIA Cg Compiler currently has two NV30 profiles:
  - Vertex program (vp2.0)
  - Fragment Program (vp1.0)
- DX9 vertex/pixel shader profile support forthcoming
- Vertex profiles:
  - No “half” or “fixed” data type
  - No texture functions – It’s a vertex program!
- Fragment/pixel profiles:
  - No “for” or “while” loops (unless they’re unrollable)
  - etc.

Other profile limitations for NV30

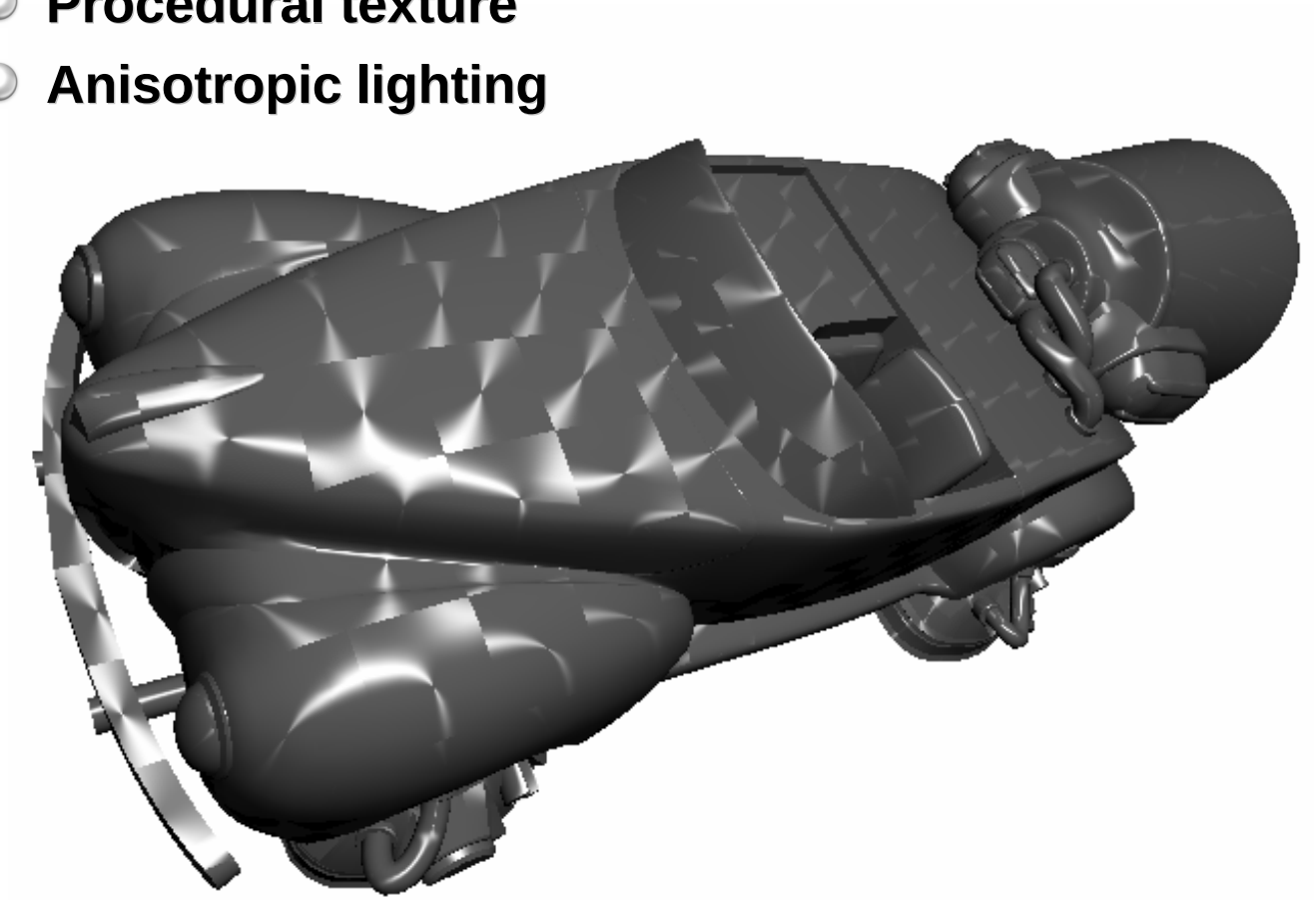
- No pointers – not supported by HW
- Function parameters are passed by value/result
  - not by reference as in C++
  - use `out` or `inout` to declare output parameter
  - aliased parameters are written in order
- No unions or bit-fields
- No `int` data type

Cg Summary

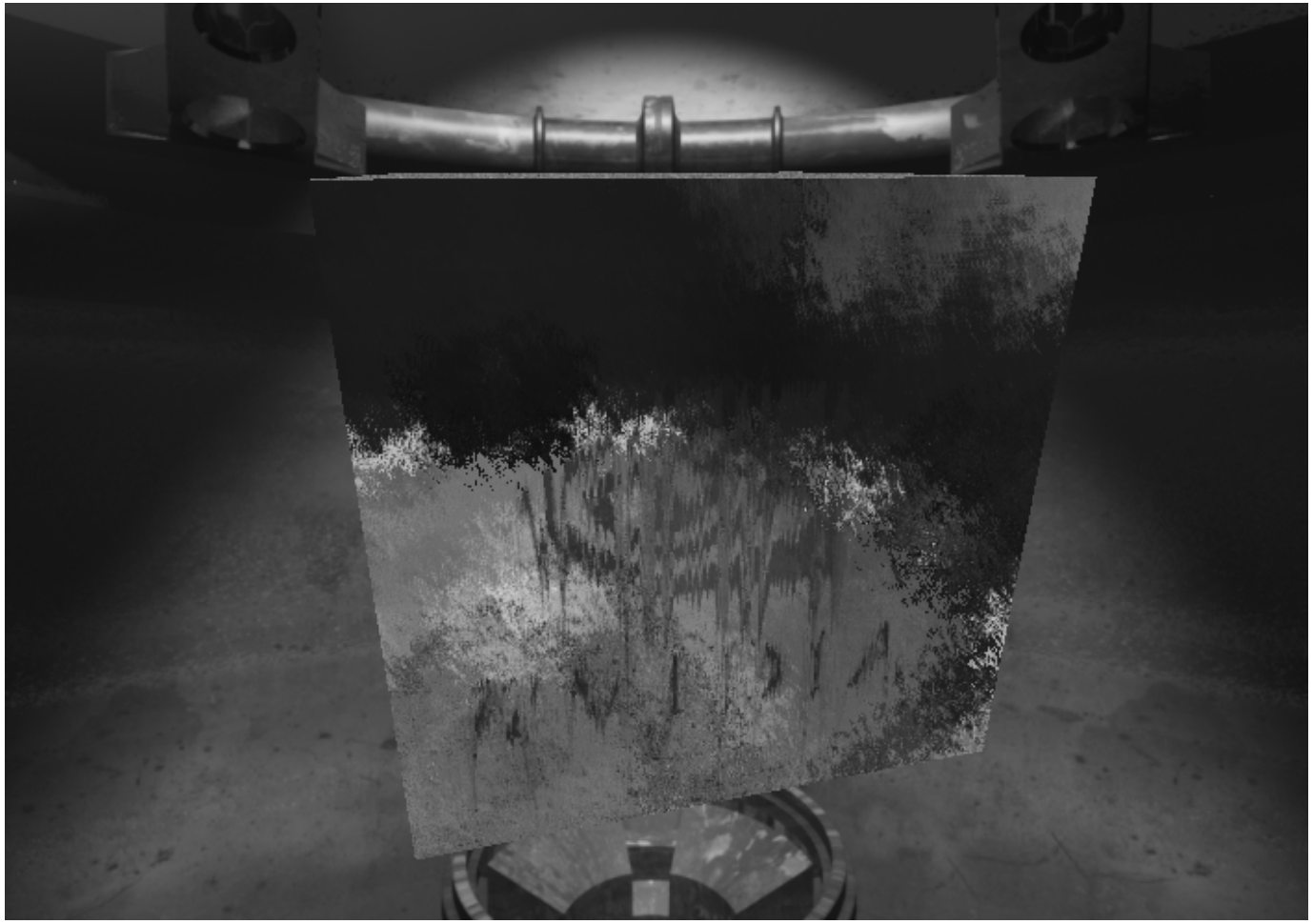
- C-like language – expressive and efficient
- HW data types
- Vector and matrix operations
- Write separate vertex and fragment programs
- Connectors enable mix & match of programs by defining data flows
- Will be supported on any DX9 hardware
- Will support future HW (beyond NV30/DX9)

Brushed Metal

- Procedural texture
- Anisotropic lighting

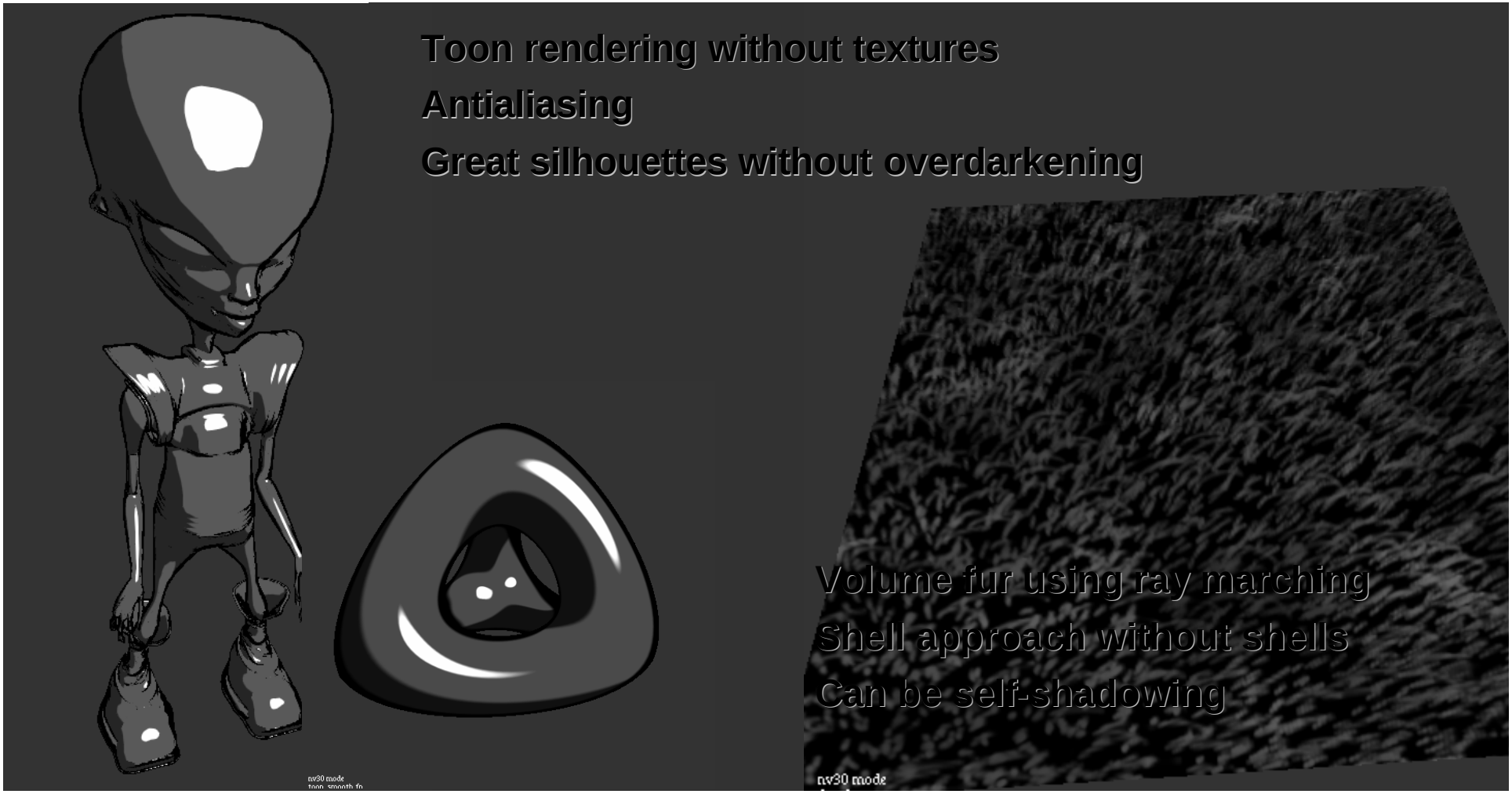


Melting Ice



- Procedural, animating texture
- Bumped environment map

Toon & Fur



Vegetation & Thin Film

Translucence

Backlighting

Example of custom lighting

Simulates iridescence

Questions?

- http://www.nvidia.com/Cg/
- http://www.cgshaders.org/
- kbjorke@nvidia.com