Computer Graphics

– Programmable Shading in HW –

Philipp Slusallek

Overview

- **So far:**
  - OpenGL
  - Clipping
  - Rasterization

- **Today:**
  - Programmable graphics hardware
  - Shading Language: Cg

Resources

- **Fernando, Kilgard, "The Cg Tutorial"**
  - Addison-Wesley, 2003
- **http://developer.nvidia.com/Cg**
  - Whitepapers
  - Presentations
  - Cg tutorials http://developer.nvidia.com/object/cg_toolkit.html
  - Cg User's Manual
  - Cg Language Specification
  - Cg Toolkit Downloads
  - Bug Reporting
- **www.CgShaders.org**
  - Forums
  - Shader Repository (Freeware)

Motivation

E 2666 C, Einzelpreis 2,60 € www.computer-zeitung.de 34. Jahrgang Nr. 8 / Montag, 17. Februar 2003

INDUSTRIE NET
Die Profi-Liga
• 15.000 Firmenporträts
• 100.000 Fachartikel
• Suchmaschinen
• Datenbanken
• Marktübersichten
www.industrienet.de

COMPUTER ZEITUNG

Besuchen Sie die COMPUTER ZEITUNG auf der CeBIT HANNOVER 12.–19. 3. 2003 Halle 1 Stand 5a9

Die Wochenzeitung für die Informationsgesellschaft

Im Fokus

Sparc und Intel unter einem Dach
Server-Blades auf Basis von Intel- und von Sparc-Prozessoren hat Sun vorgestellt. Die Besonderheit: Die unterschiedlichen Karten können das gleiche Rackeinschubgehäuse benutzen. **Seite 4**

Neue Software lähmt den Betrieb
Die Einführung unternehmensweiter Standardsoftware bringt oft eine Erstarrung der Geschäftsprozesse, warnen Analysten. **Seite 11**

Sicherheit wird nicht eingebaut
Die Protokolle für sichere Webservices sind jetzt vorhanden – an ihrer Implementierung hapert es aber noch, befürchtet Middleware-Experte Tim Eckardt. **Seite 12**

Isolation oder Integration?
Beim E-Commerce ist vor al-

Halbleiterkonferenz ISSCC: Leckströme in Leiterbahnen konterkarieren die Leistungssteigerung

Silizium-Chipdesign stößt an Grenzen

San Francisco (rr) – Vor den Chipherstellern türmen sich über die kommende Dekade zwei Blockaden: Die Kriechströme und der Stromverbrauch der Winzlinge verstärken sich mit steigender Miniaturisierung. Der Gigahertz-Schlag des E-Business droht auszusetzen.

„Die Halbleiterindustrie muss die Leckströme im Chipdesign innerhalb von zehn Jahren um zwei Größenordnungen reduzieren, oder die Steigerung der Chipkomplexität bricht ab", warnt Takayasu Sakurai von der University of Tokio auf der 50. International Solid-State Circuit Conference (ISSCC).

Ab 2010, wenn laut Ex-Intel-Chairman Gordon Moore die Strukturbreiten von heute 130 Nanometer auf 30 Milliardstelmeter geschrumpft sind, wechne Crux vor allem für die nachgefragten Mobilgeräte.

Die Chipschmieden arbeiten mit Hochdruck an neuen Transistortypen, einem optimierten Strommanagement und einer nagement. So hat Intel-Fellow Shekar Borkar auf der ISSCC den Schlaftransistor postuliert, der den Stromverbrauch um bis zu 44 Prozent senken kann. Neben Standby-Ansätzen dis- Energie aus Vibrationen und der menschlichen Abwärme.

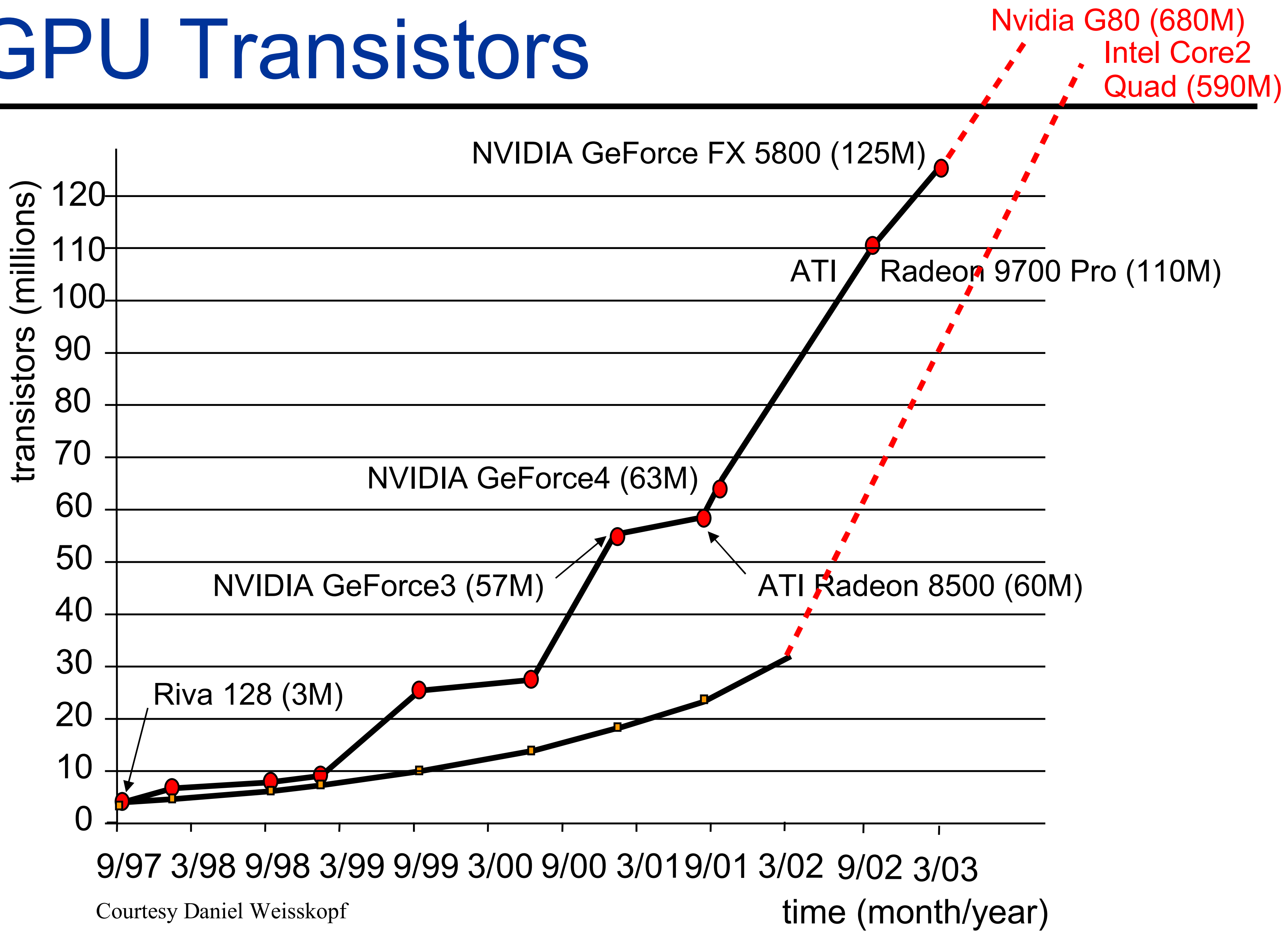
Die größte Hoffnung liegt auf der vertikalen Anordnung der Transistoren auf den Schaltkreisen, sodass sie mehrfach an- davon aus, dass die Konstruktionsvariante Strukturen von zehn Nanometern zulässt. Und Intel verfeinert den Entwurf für den Trigate-Transistor (*ISSCC-CPU-News auf Seite 2*).

Greifen die Ansätze, so lässt sich laut Moore die Verdoppelung der Transistordichte im 18- bis 24-Monate-Zyklus – das so genannte Moore'sche Gesetz – noch über zehn Jahre prolongieren. Miniaturisierungstechniken seien neue Belichtungsmethoden wie die höher auflösende extreme Ultraviolett-Lithografie oder Materialien wie gestrecktes Silizium mit höherer Leitfähigkeit.

Für Infineon-Technikchef Sönke Mehrgardt stellt das Jahr 2085 die harte Grenze dar: „Rechnerisch besteht dann ein Bauelement nur noch aus einem halben Atom." Spätestens dann

Für Furore hat Infineons Neurochip auf der Halbleiterkonferenz ISSCC gesorgt. 16 484 Sensoren (Durchmesser: 10 bis 50 Mikrometer) tasten die elektrischen Signale der Nervenzelle ab. Dagegen bangt die Chipszene um die Langfristperspektive bei Siliziumhalbleitern.
Foto: Infineon

GPU Transistors



Courtesy Daniel Weisskopf

Graphics Hardware

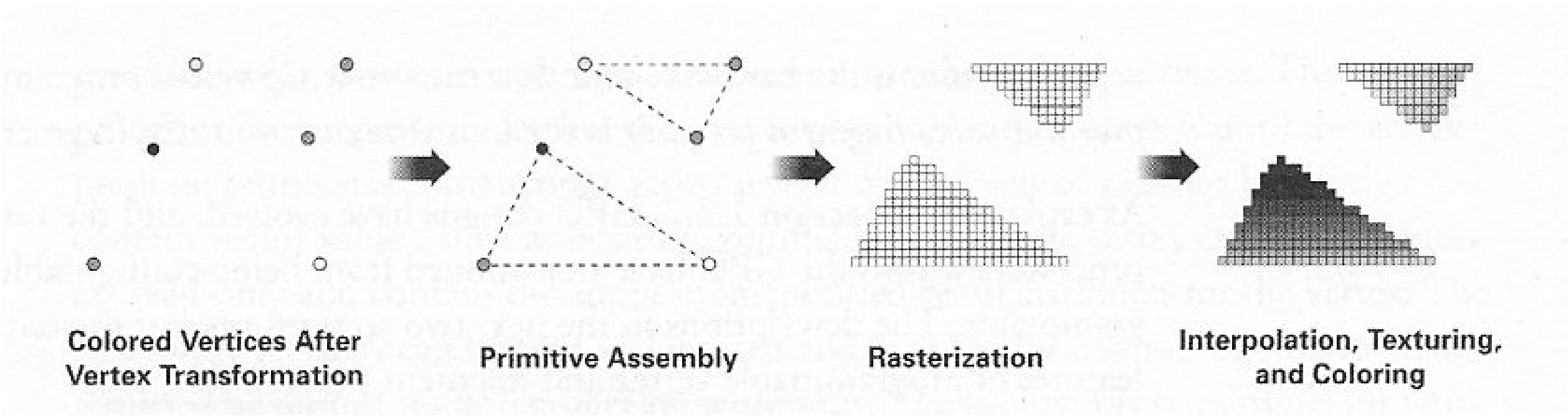
Generation	Year	Product Name	Process	Transistors	Antialiasing Fill Rate	Polygon Rate
First	Late 1998	RIVA TNT	0.25 µ	7 M	*50 M*	6 M
First	Early 1999	RIVA TNT2	0.22 µ	9 M	*75 M*	9 M
Second	Late 1999	GeForce 256	0.22 µ	23 M	*120 M*	15 M
Second	Early 2000	GeForce2	0.18 µ	25 M	*200 M*	25 M
Third	Early 2001	GeForce3	0.15 µ	57 M	800 M	30 M
Third	Early 2002	GeForce4 Ti	0.15 µ	63 M	1200 M	60 M
Fourth	Early 2003	GeForce FX	0.13 µ	125 M	2000 M	200 M
Eigth	Early 2007	GeForce 8800	0.090µ	681 M	13,800 M	10,800 M

History

- **Pre-GPU Graphics Acceleration**
  - SGI, Evans & Sutherland
  - Introduced concepts like vertex transformation and texture mapping.
- **First-Generation GPU (-1998)**
  - Nvidia TNT2, ATI Rage, Voodoo3
  - Vertex transformation on CPU, limited set of math operations.
- **Second-Generation GPU (1999-2000)**
  - GeForce 256, Geforce2, Radeon 7500, Savage3D
  - Transformation & Lighting. More configurable, still not programmable.
- **Third-Generation GPU (2001)**
  - Geforce3, Geforce4 Ti, Xbox, Radeon 8500
  - Vertex Programmability, pixel-level configurability.
- **Fourth-Generation GPU (2002)**
  - Geforce FX series, Radeon 9700 and on
  - Vertex-level and pixel-level programmability
- **Eigth-Generation GPU (2007)**
  - Geometry Programs, Unified Shaders, ...

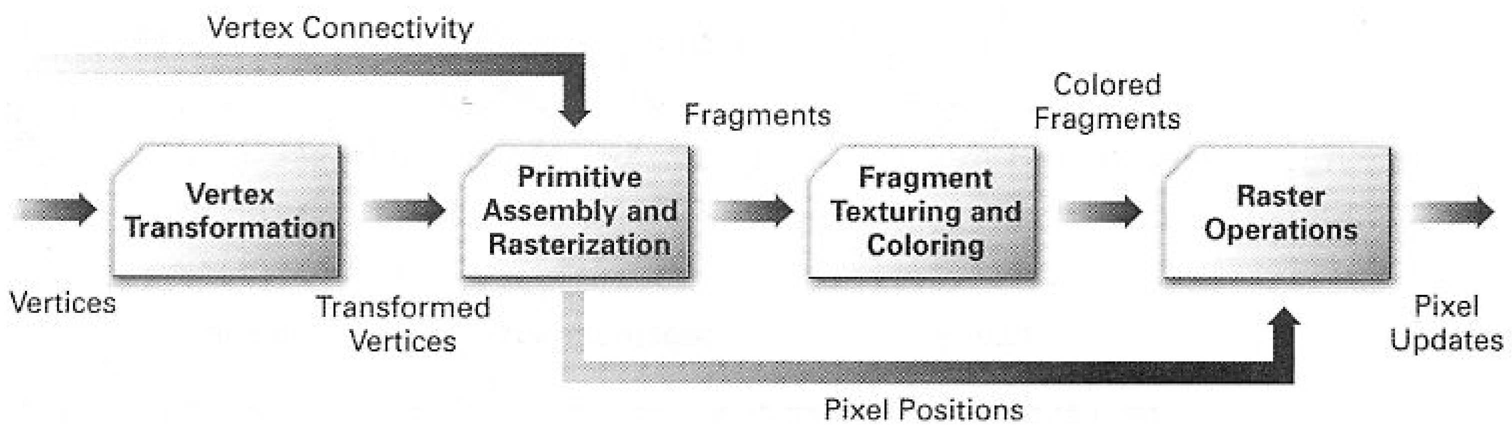
Before GPUs

- **All vertex transformations handled by CPU**
- **Limited the number of vertices in a scene**
- **Could still achieve many effects, but limited by CPU power**
- **Card "power" focused on fill rates**
- **Didn't allow much room for AI, physics**

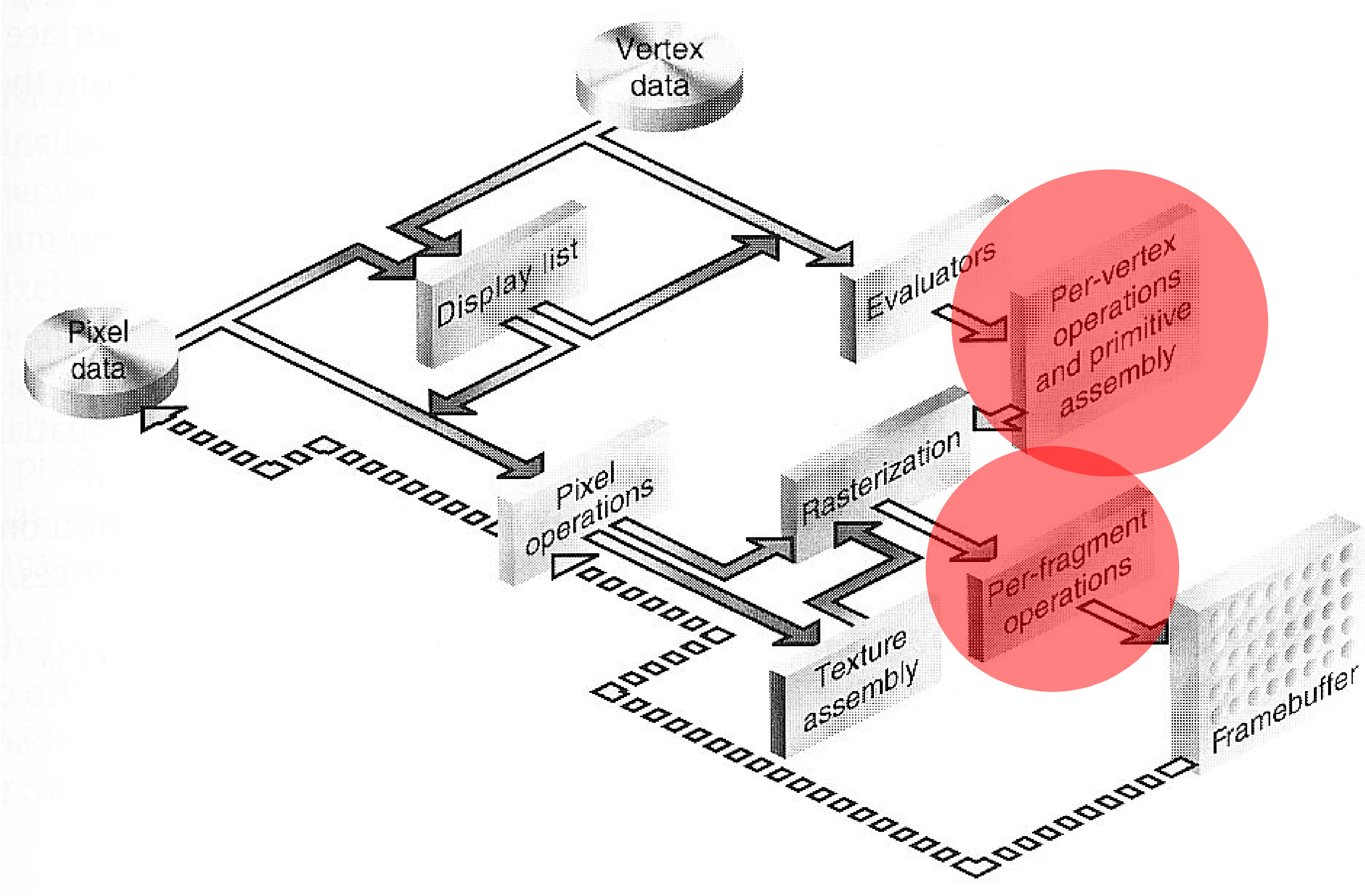


GeForce 256 – Hardware T&L

- **GPU now handled transformation of vertices**
- **Freed up CPU for AI and physics**
  - Allowed for other parts of the game to be made more realistic
- **Fixed function hardware**
  - Provides support for what OpenGL/DirectX did in the background
  - Could not be used to invent new techniques



Programmable Graphics Hardware



Geforce 3&4 – Vertex Shaders 1.0

- **Vertex Shaders**
  - Operate per-vertex
  - Allow customization of how vertices are transformed
  - Used in calculating per-vertex lighting
  - Support up to 128 instructions
    - No branching/conditional programming
    - 17 instructions available
    - All instructions operate on 4 float vectors (x,y,z,w)

Geforce 3&4 – Vertex Shaders 1.0

- **Vertex Shader Assembly Language**
  - Five types of registers
    - Address Register – 0 (VS 1.0) 1 (VS 1.1+)
      - Write/Use (cannot be read)
    - Constant Registers – 96
      - Read Only to GPU – set by host application
    - Temporary Registers – 12
      - Read/Write – cannot be used between vertices
    - Input Registers – 16
      - Read Only to GPU – set by application or vertex stream
    - Output Registers – 7 vector, 2 scalar
      - Write Only – position, diffuse color component, specular color component, texture coordinates (4), fog value & sprite size(scalar)

GeForce 3&4 – Vertex Shaders 1.0

Non-standard lighting

Classic Blinn lighting

GeForce 3&4 – Pixel Shaders 1.0

- **Pixel Shaders**
  - Operate per-pixel
  - Used to combine textures & calculate lighting
  - Commonly used for per-pixel bump mapping
  - Limited to 32 instructions
    - No conditional/branching operations

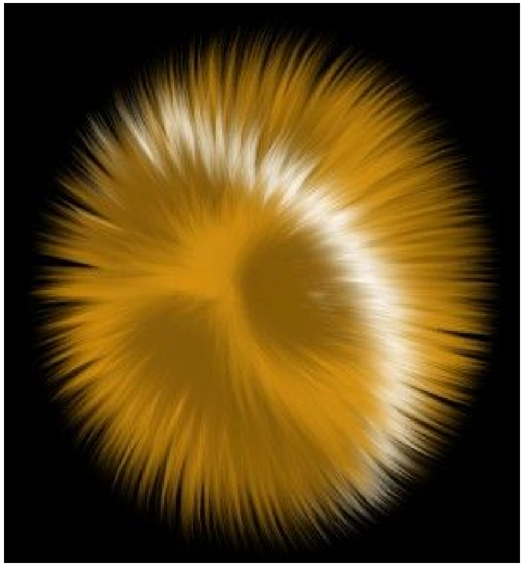
GeForce 3&4 – Pixel Shaders 1.0

- **Pixel Shader Assembly Language**
  - Four types of registers
    - Constants – 8
      - Read only (set by application)
    - Temporary – 2 (PS 1.4 has 8)
      - Read/Write (cannot be used by other pixels)
      - Output written to first temporary register (v0)
    - Textures – 4 (PS 1.4 has 6)
      - Read/Write (used to combine different texture)
    - Colors – 2
      - Read only
      - v0 for diffuse color
      - v1 for specular color

GeForce 3&4 – Pixel Shaders 1.0

Fresnel term

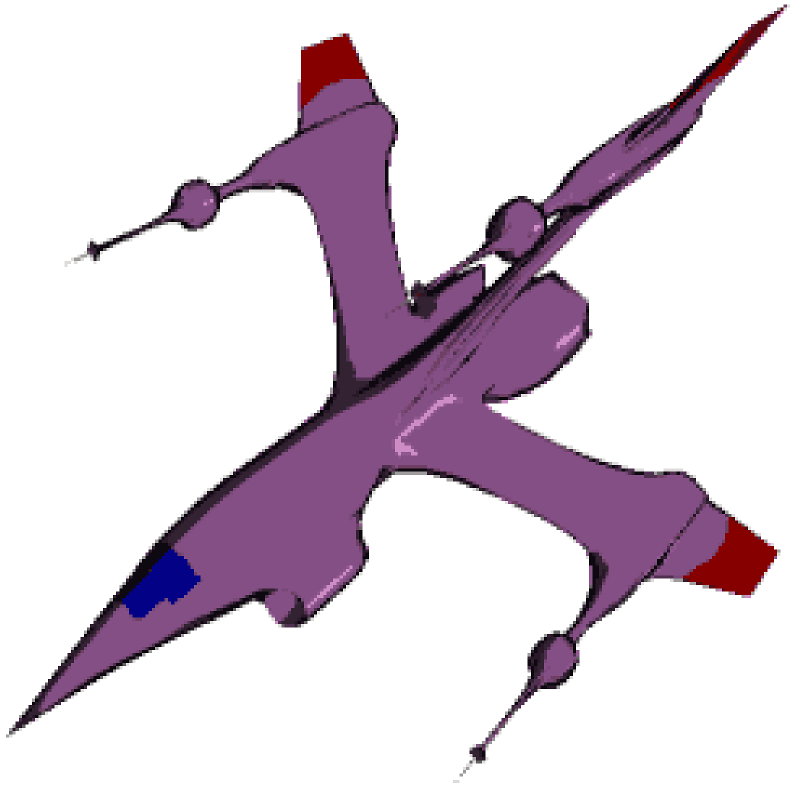
Fur/Hair



Refraction

GeForce 3&4 – Pixel Shaders 1.0

NPR Effects

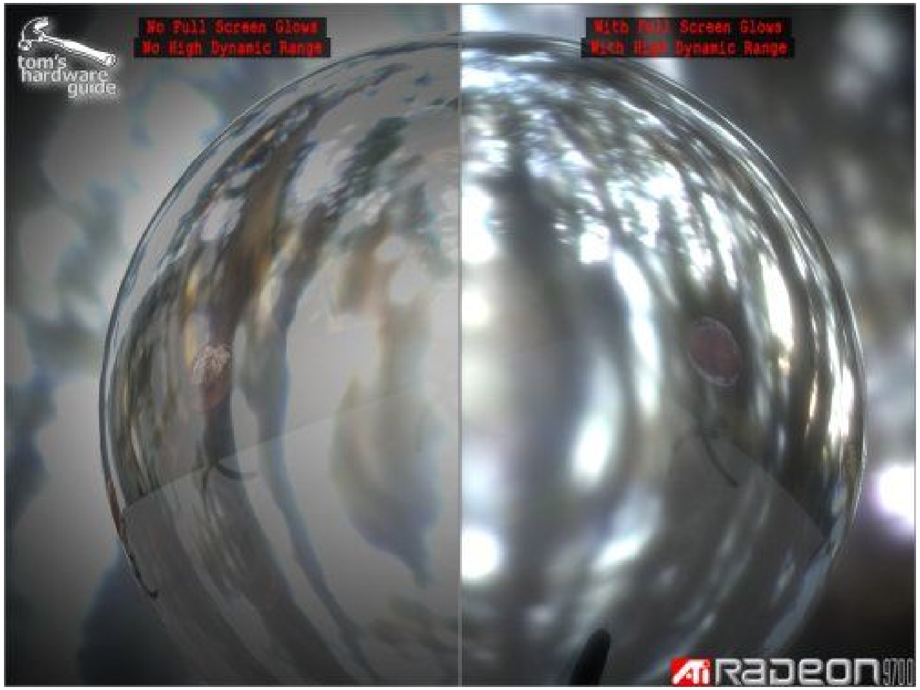


GeForce FX – Vertex Shaders 2.0

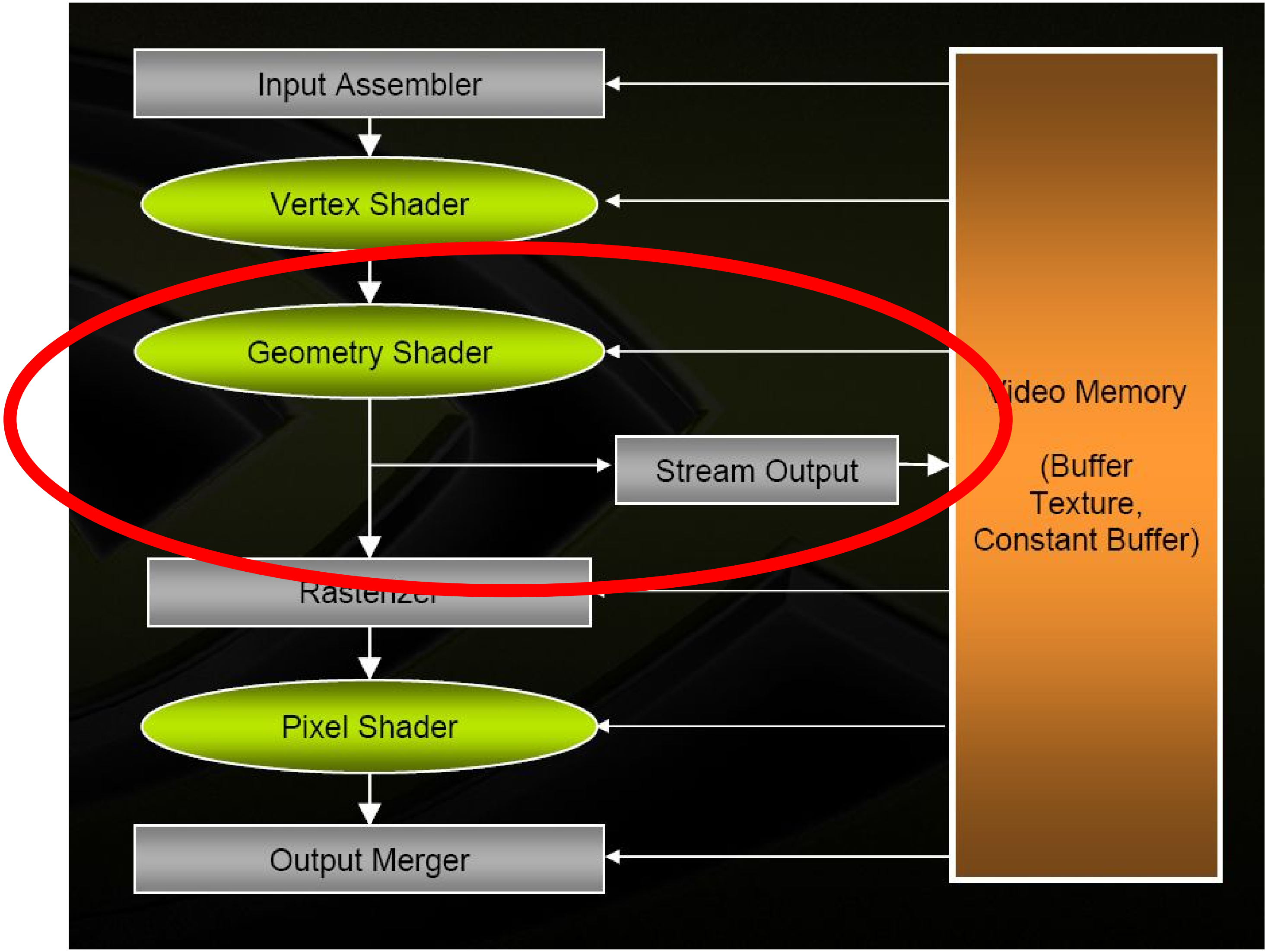
- **Now up to 1024 instructions**
  - (NVIDIA 2.0+ spec has up to 65536)
- **Supports Branching**
  - Conditional Jumps
  - Loops (up to 4 spec, NVIDIA up to 256)
  - Procedures
- **256 constants, 16 temporary registers**
- **128-bit floating point precision**
- **Supports N-patch ‘high order surface’ tessellation and ‘displacement mapped N-patch**

GeForce FX – Vertex Shaders 2.0

- **Supports 64-bit and 128-bit FP precision**
- **Adds Loops, Conditionals, Functions**
- **Max instructions increased to 96 (1024 on NV)**



DX-10 and Shader Model 4.0

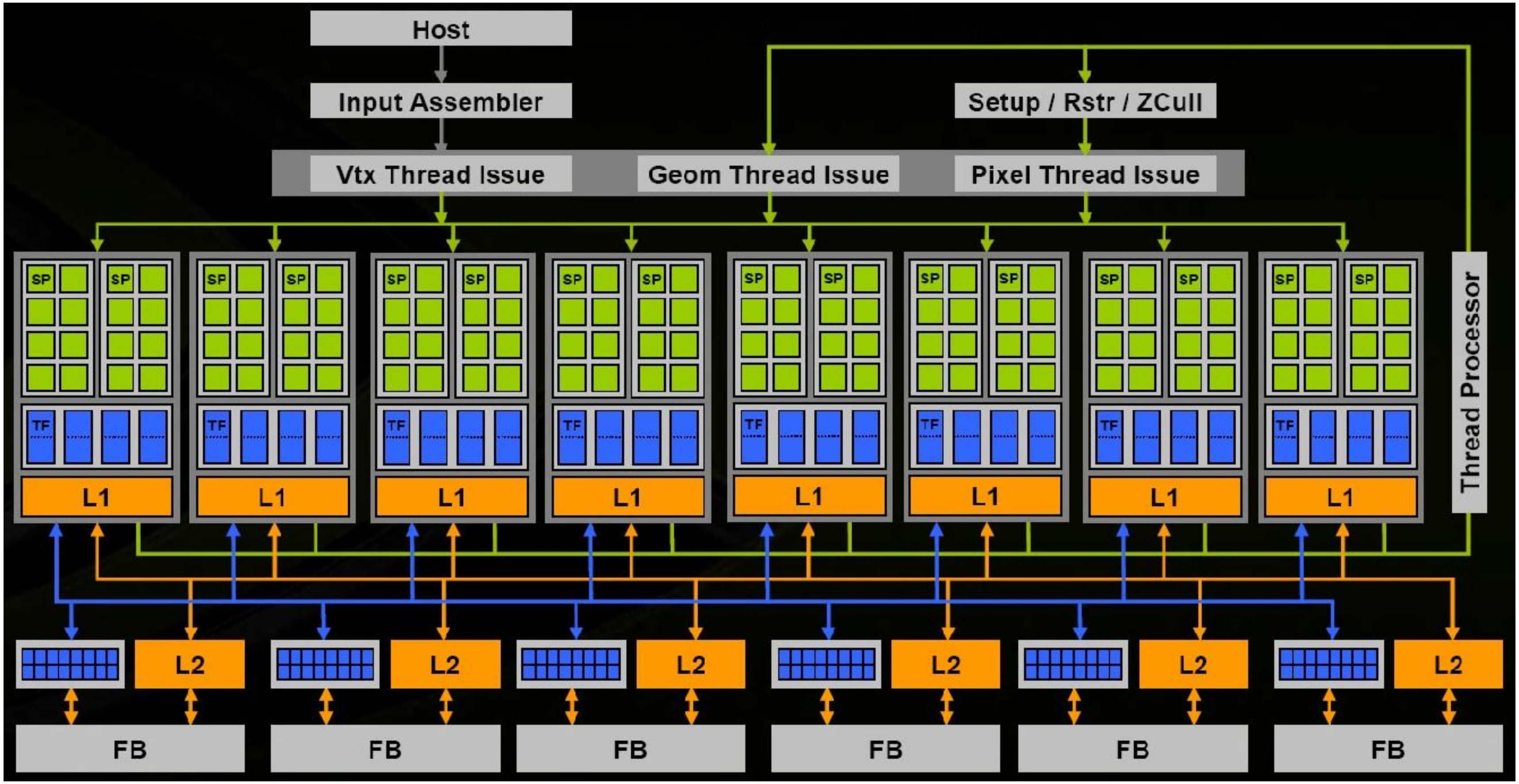


DX-10 and Shader Model 4.0

Feature	1.1 2001	2.0 2002	3.0 2004[†]	4.0 2006
instruction slots	128 4+8[‡]	256 32+64[‡]	≥512 ≥512	≥64K
constant registers	≥96 8	≥256 32	≥256 224	16x4096
tmp registers	12 2	12 12	32 32	4096
input registers	16 4+2[§]	16 8+2[§]	16 10	16 32
render targets	1	4	4	8
samplers	8	16	16	16
textures	 8	 16	4 16	128
2D tex size			2Kx2K	8Kx8K
integer ops				✓
load op				✓
sample offsets				✓
transcendental ops	✓ 	✓ ✓	✓ ✓	✓
derivative op			✓	✓
flow control	 	static 	stat/dyn stat/dyn	dynamic

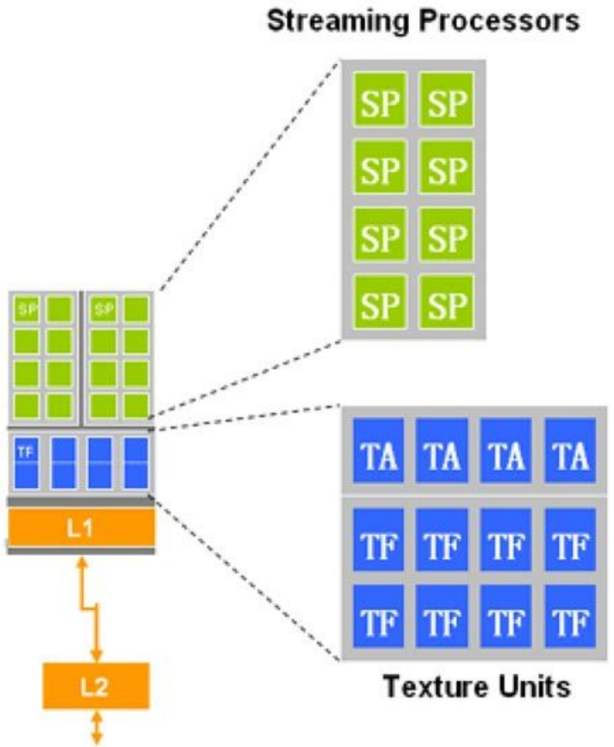
Table 1: Shader model feature comparison summary.

G80 – Unified Shaders



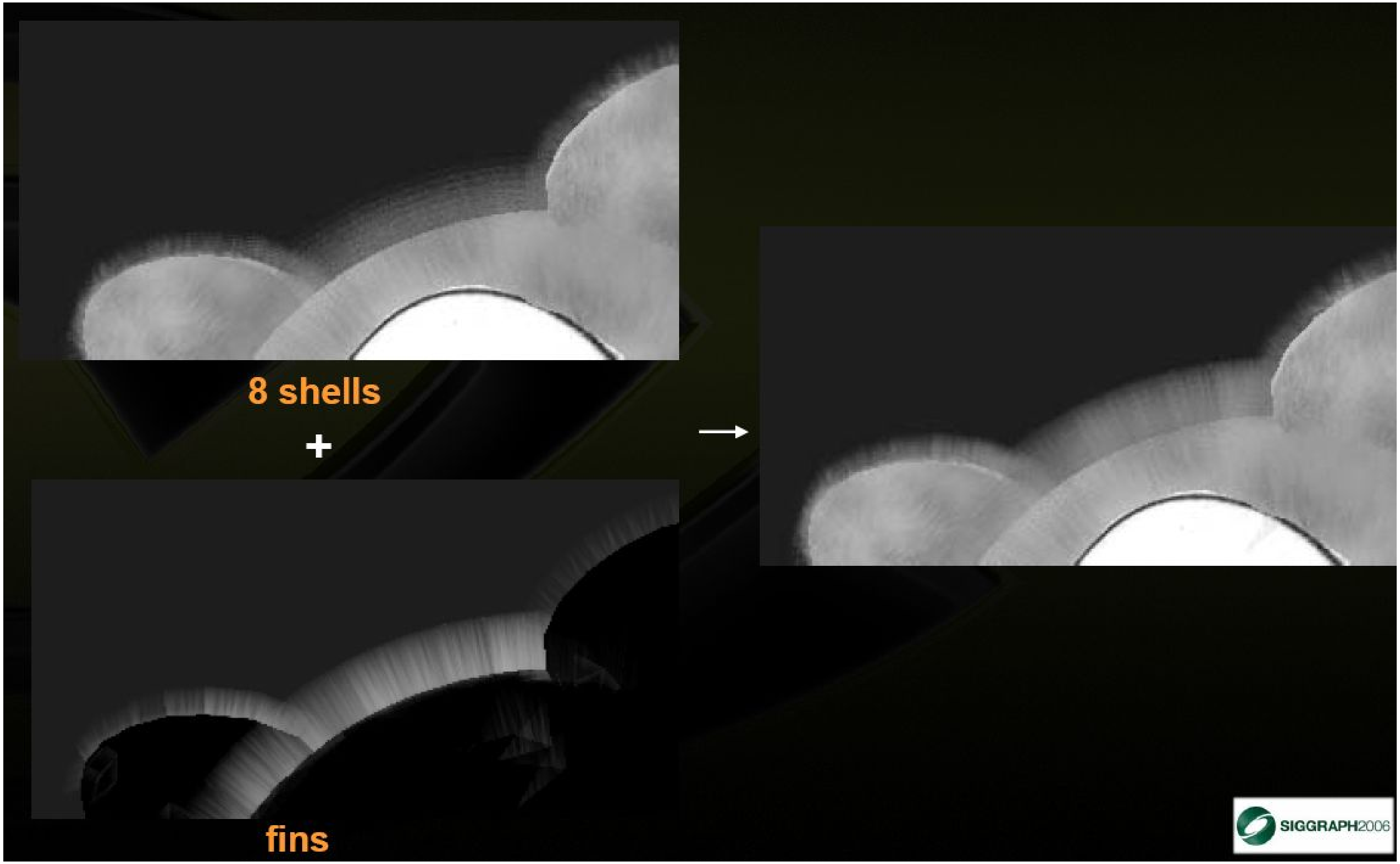
G80 – Unified Shaders

Streaming Processors, Texture Units, and On-chip Caches

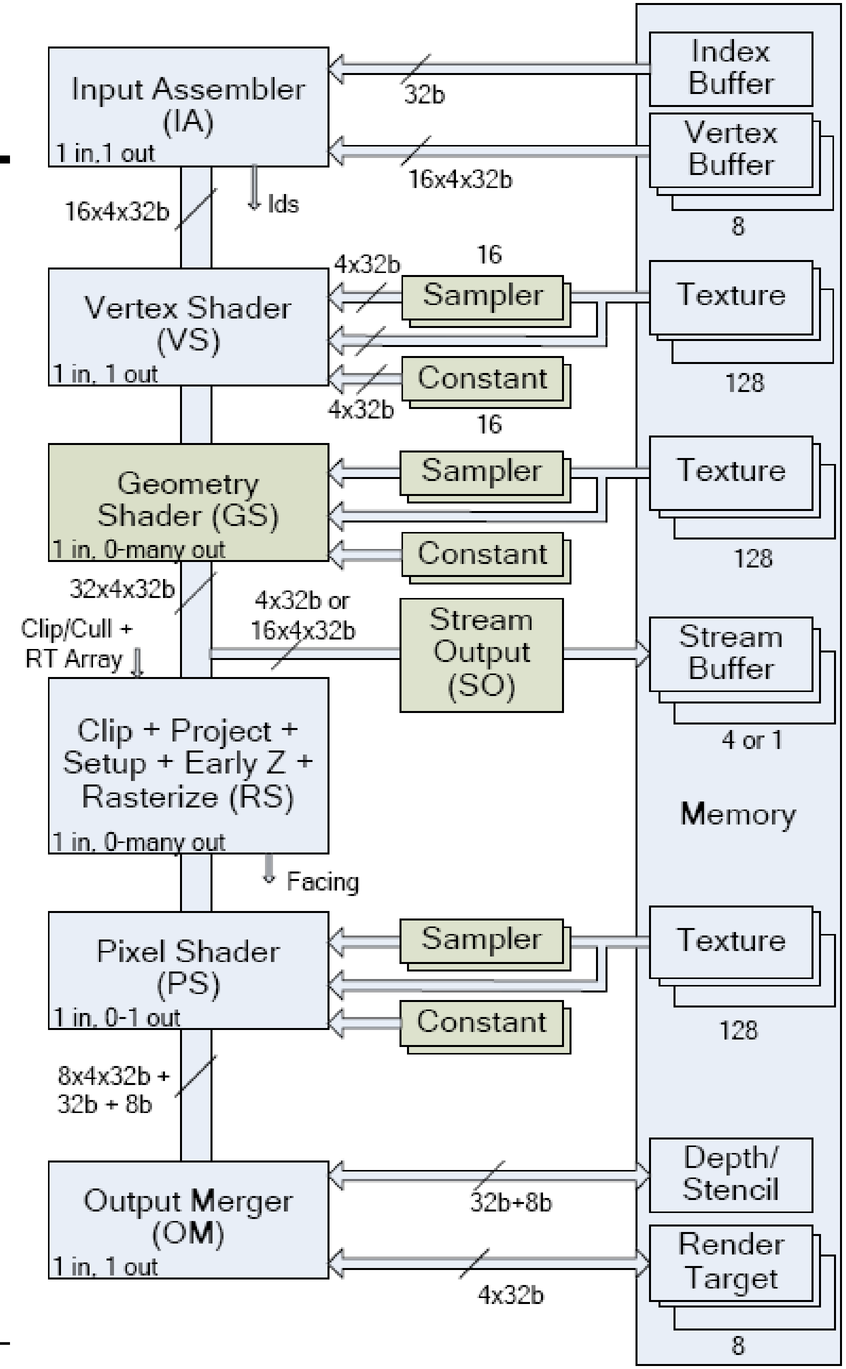


- **SP = Streaming Processors**
- **TF = Texture Filtering Unit**
- **TA = Texture Address Unit**
- **L1/L2 = Caches**

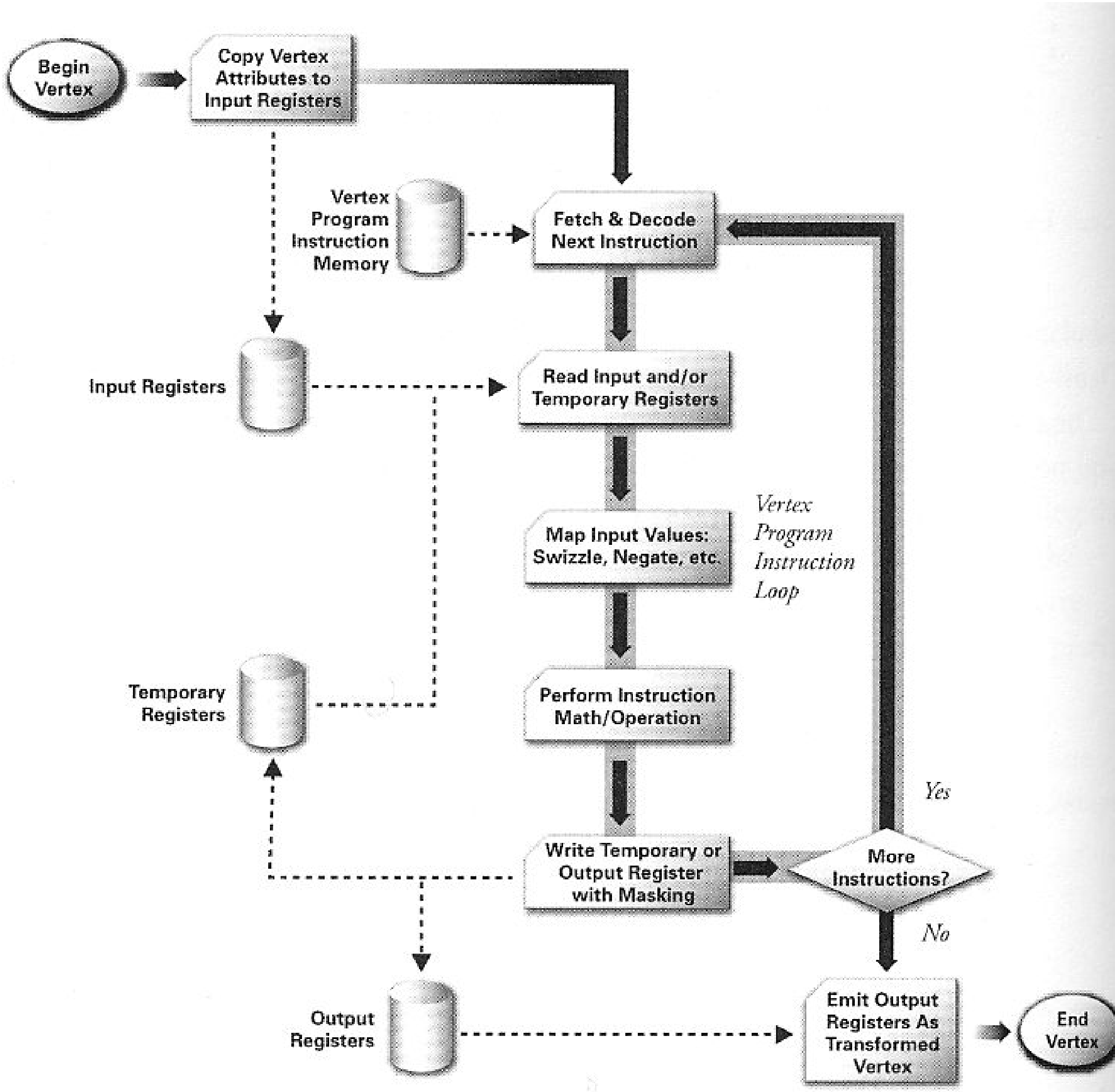
Figure 18. Streaming Processors and Texture Units


8 shells
+
fins
SIGGRAPH2006

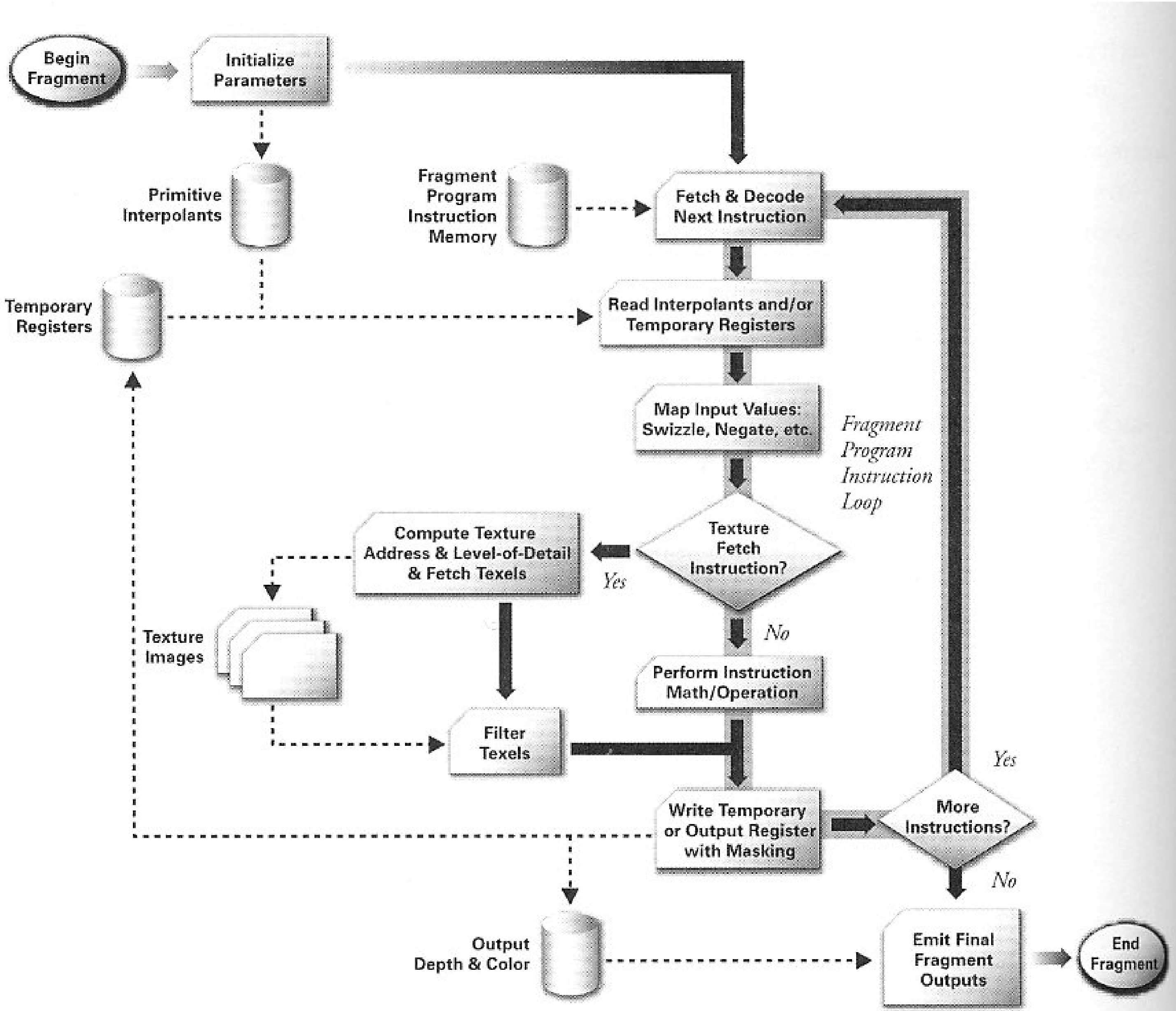
Shader Model 4



Vertex Processor Flow Chart



Fragment Processor Flow Chart



High Level Shader Languages

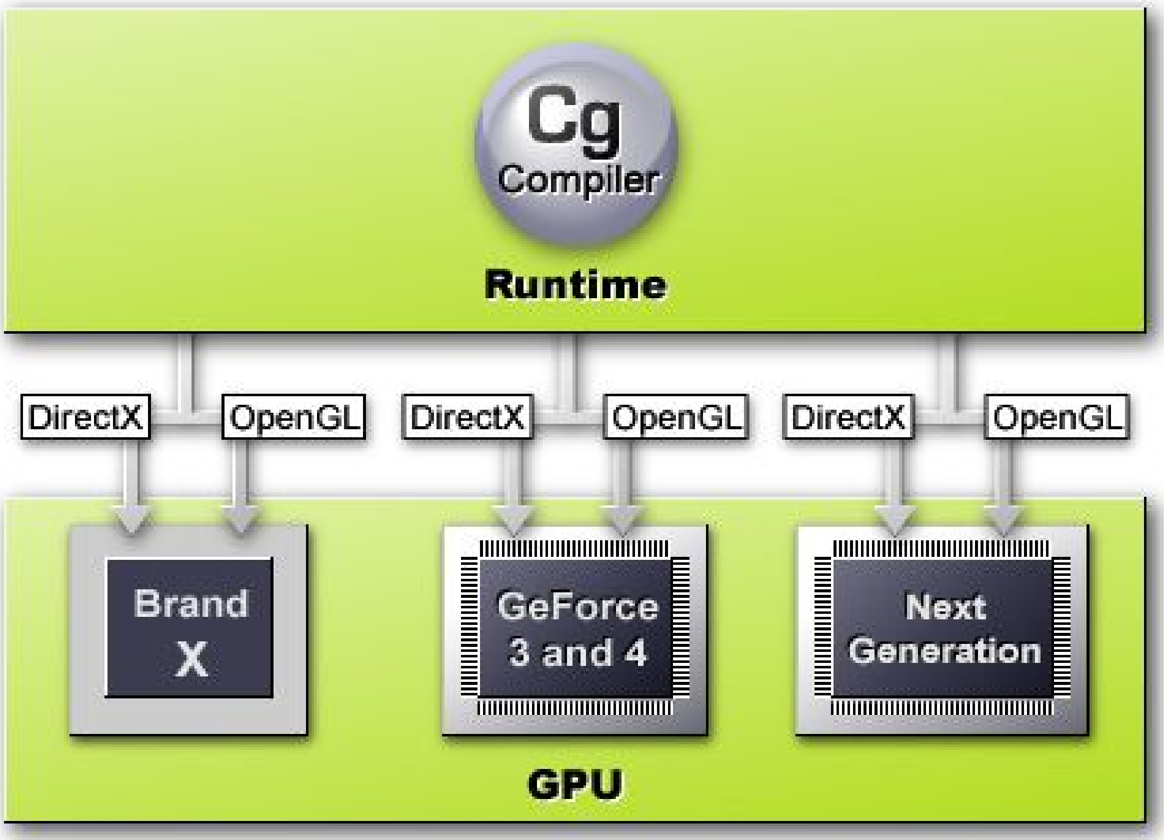
- **Programming shaders in machine code isn't easy**
- **DirectX 10 HLSL & NVIDIA Cg**
  - Writing shaders in C-like code
  - Supports C-like syntax
  - Open source compiler
    - New compilers can be written to compile the same code for different architectures
  - Optimization of code for different levels of hardware
  - Allows compiling for DirectX & OpenGL

Cg

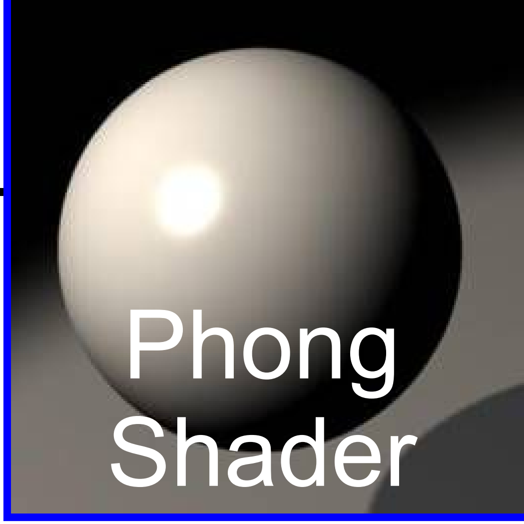
- "**C for Graphics**"
  - high-level, cross-platform language for graphics programming
- **C-like language**
  - Replaces tedious assembly coding
  - compiler generates assembler code
- **Cross-API, cross-platform language**
  - OpenGL and DirectX
  - Windows and Linux
  - NVIDIA, ATI, Matrox, any other programmable hardware that supports OpenGL or DirectX
- **Cg Runtime**
  - simplifies parameter passing from application to vertex and fragment programs

Cg

- **Forward compatibility**
- **Works with all programmable GPUs supporting DirectX 8/9/10 or OpenGL 1.5/2.0**



What does Cg look like ?



Assembly

```
…
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```

Cg

```
COLOR cPlastic = Ca + Cd * dot(Nf, L) +
      Cs * pow(max(0, dot(Nf, H)), phongExp);
```

Why Cg ?

- **Simplifies developing OpenGL and DirectX applications with programmable shading**
  - Easier than assembly
  - Simplified parameter management
  - Abstraction from hardware and graphics API
- **Flexible—use as little or as much of it as you want**
  - Cg language only
  - API-independent libraries
  - API-dependent libraries
- **Productivity increase for graphics development**
  - Game developers
  - DCC (Digital Content Creation) artists
  - Artists & shader writers
  - CAD and visualization application developers

Compiling Cg at Runtime

At Development Time

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

Cg program source code

At Runtime

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

Cg Runtime Compilation

- **Pros**:
  - Future compatibility: The application does not need to change to benefit from future compilers (future optimizations, future hardware)
  - Easy parameter management

- **Cons**:
  - Loading takes more time because of compilation
  - Cannot tweak the result of the compilation

OpenGL Cg Runtime

- **Makes the necessary OpenGL calls for you**
- **Allows you to:**
  - Load a program into OpenGL: `cgGLLoadProgram()`
  - Enable a profile: `cgGLEnableProfile()`
  - Tell OpenGL to render with it: `cgGLBindProgram()`
  - Set parameter values: `cgGLSetParameter{1234}{fd}{v}()`, `cgGLSetParameterArray{1234}{fd}()`, `cgGLSetTextureParameter()`, etc...

Cg and C

- **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., pointers, switch, case…)
  - Compiler uses *hardware profiles* to subset Cg as required for particular hardware capabilities (or lack thereof)

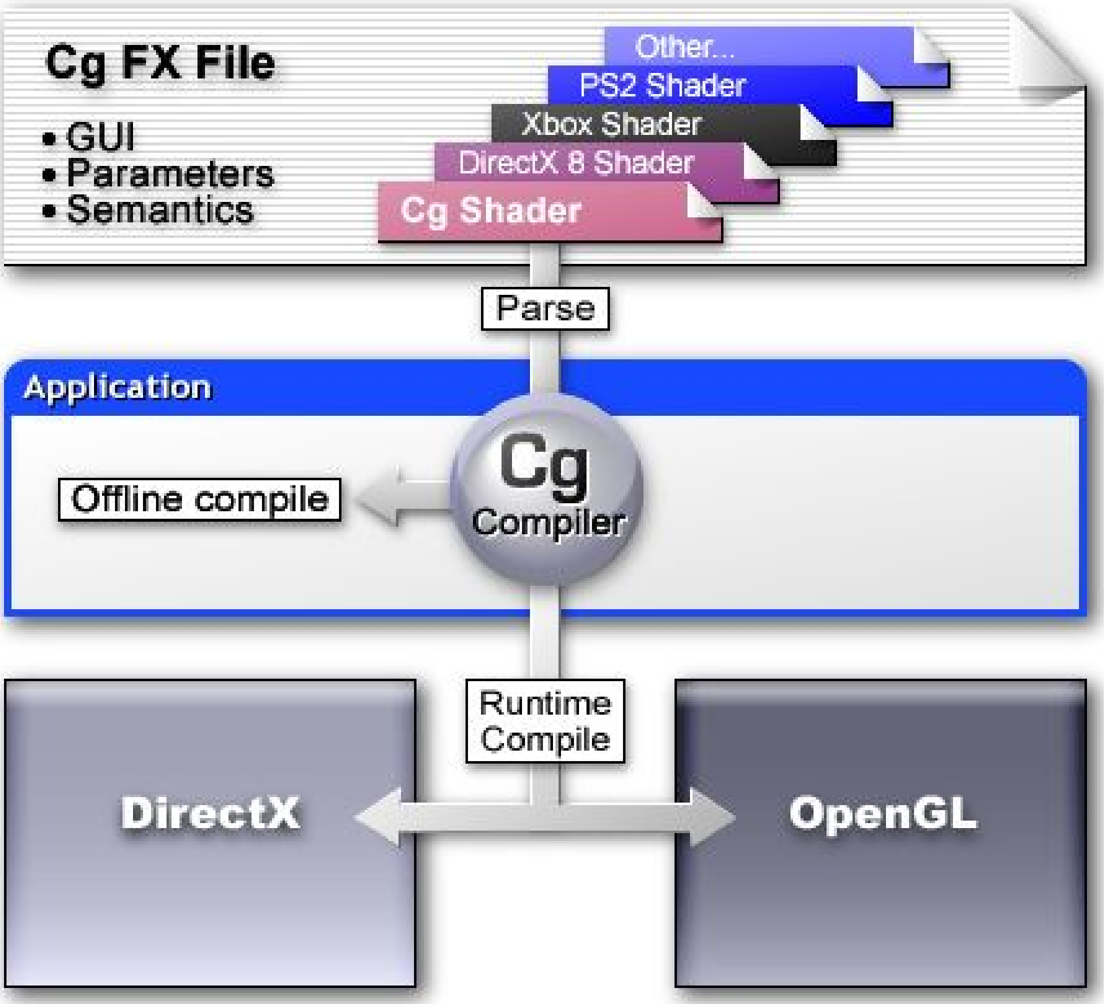
Cg and CgFX

- **One Cg vertex & pixel program together describe a single rendering pass**

- **CgFX shaders can describe multiple passes**
  - Although CineFX architecture supports 1024 pixel instructions in a single pass!

  - CgFX also contains multiple implementations
    - For different APIs
    - For various HW
    - For Shader LOD

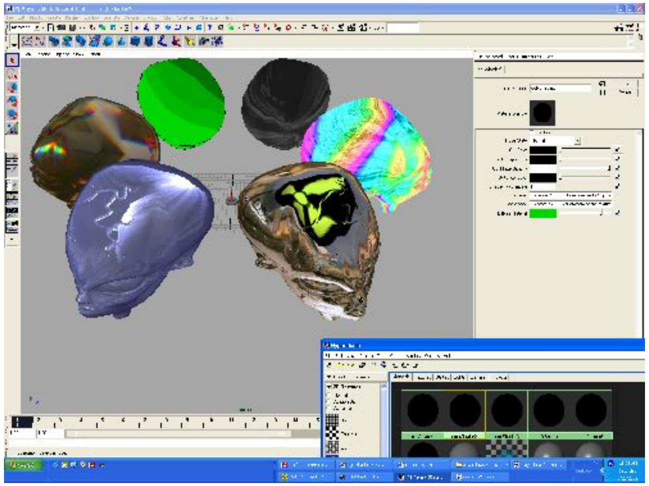
CgFX



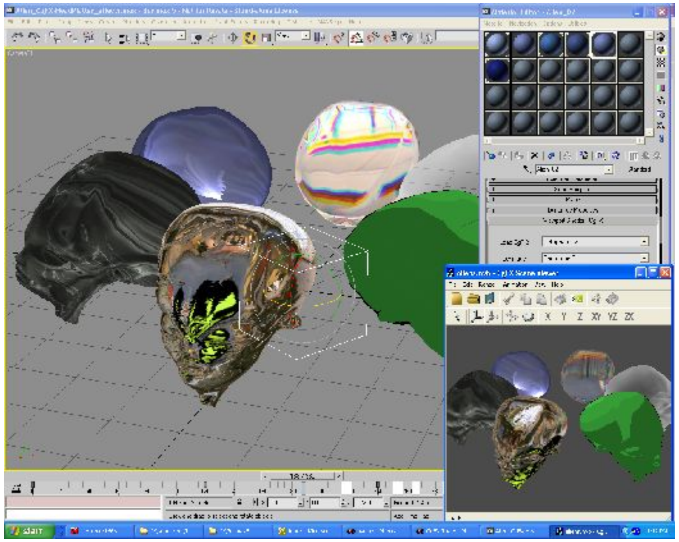
- **CgFX files contain shaders and the supplementary data required to use them**
- **Unlimited multiple implementations**
  - API (D3D vs. OpenGL)
  - Platform (Xbox, PC, …)
  - Shader Level of Detail
- **NVIDIA's Compiler includes a CgFX parser for easy integration**

Cg in Professional Graphics Software

Maya®



3ds max™



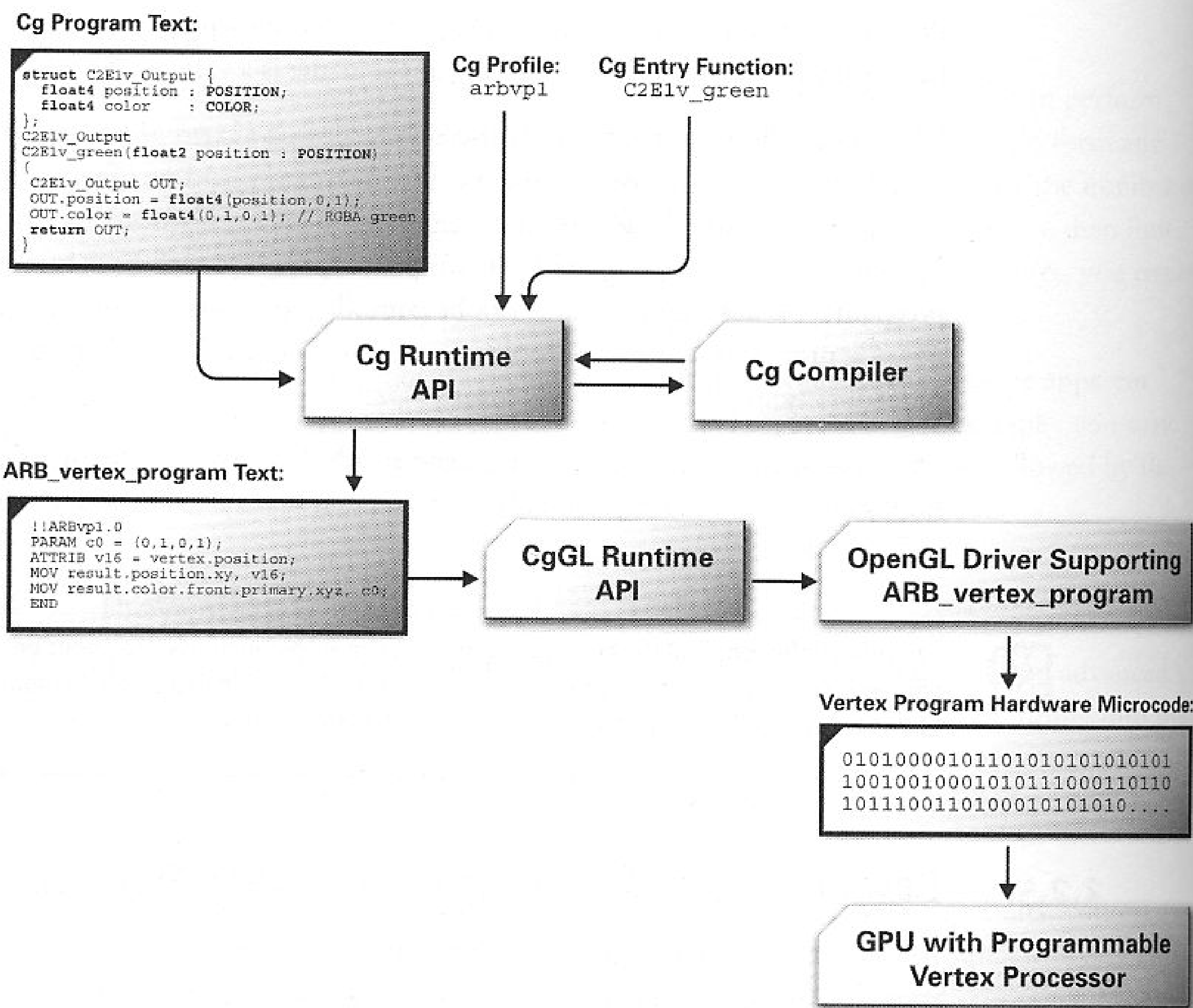
SOFTIMAGE® | XSI™

Cg Compiler Profiles

- **Different graphics cards have different capabilities**
  - Exploit individual hardware

- **Programs must be compiled to a certain profile**
  - Input: Cg program + profile to compile to
  - Output: Assembly language for the specified hardware

Compiling & Loading Cg Program



A First Cg Example

Vertex Program

```
struct C2E1v_Output {
  float4 position : POSITION;
  float4 color    : COLOR;
};

C2E1v_Output C2E1v_green(
    float2 position : POSITION)
{
  C2E1v_Output OUT;

  OUT.position = float4(position, 0, 1);
  OUT.color    = float4(0, 1, 0, 1);

  return OUT;
}
```

Fragment Program

```
struct C2E2f_Output {
  float4 color : COLOR;
};

C2E2f_Output C2E2f_passthrough(
    float4 color : COLOR)
{
  C2E2f_Output OUT;

  OUT.color = color;
  return OUT;
}
```

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**

Arrays, Matrices, Vectors

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

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

- **Not the same as arrays :**

```
float mycolor[4];
float mymatrix[3][3];
```

- **Arrays are first-class types, not pointers**

Function Overloading

- **Examples:**

```
float myfuncA(float3 x);
float myfuncA(half3 x);

float myfuncB(float2 a, float2 b);
float myfuncB(float3 a, float3 b);
float myfuncB(float4 a, float4 b);
```

- **Very useful with all the different Cg data types**

Vector and Matrix Arithmetics

- **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;
```

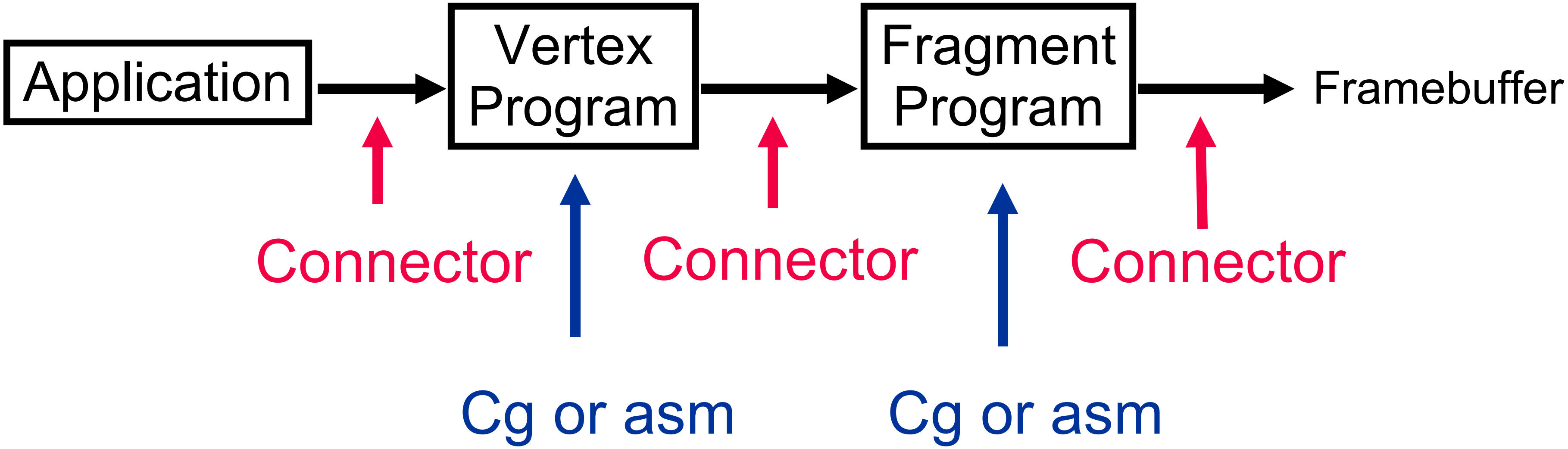
- **Vector constructor builds vector**

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

- **Masking operator specifies which elements are overwritten**

```
a.wx = b.xy;
```

The Big Picture



What is a Connector ?

- **Describes shader inputs and outputs**
- **Defines an interface that allows mixing-and-matching of programs**
- **It’s a struct, but with optional extra information**

Connecting Cg to C++

- **#include <Cg/cgGL.h>**

- **Create context**
  cgContext* context = cgCreateContext();

- **Add program to context**
  cgAddProgramFromFile(context, "Skinning.cg", cgVertex30Profile, 0);

- **Get pointer to program**
  cgProgramIter* vertexProg = cgProgramByName(context, "main");

- **Get pointer to parameter**
  cgBindIter* ModelViewProjBind =
  cgGetBindByName(vertexProg, "modelViewProj");

Enabling Cg Programs

- **Bind program**

  cgGLBindProgram(vertexProg);

- **Enable program**

  cgGLEnableProgramType(cgVertex30Profile);

- **Disable program**

  cgGLDisableProgramType(cgVertex30Profile);

➢ **Just like OpenGL states**

Cg Input

- **Two types of input**
  - Varying
  - Uniform
- **Can be defined in either a struct or just as input parameter to the function**
- **No predefined names**
  - Input, program name and output can be named freely
  - Program must have specified input and output
- **Example**

```
struct app2vert {
        float4 position         : POSITION;
        float4 normal           : NORMAL;
        float4 TexCoord0        : TEXCOORD0;
};
```

Sending Data to Cg Vertex Program

- **Uniform**

```
cgGLBindUniformStateMatrix(vertexProg, ModelViewProjBind,
                cgGLModelViewProjectionMatrix, cgGLMatrixIdentity);

cgGLBindUniform4fv(vertexProg, lightBind, float [4]);
```

- **Varying**

```
glVertex3f, glNormal3f, glTexCoord2f, glColor3f, …
glVertexAttrib4fNV(x, float[4]);
```

Defining a Vertex Program

output connector

Input connector

```
v2f skinning(myappdata vin,
             uniform float4x4 m1,
             uniform float4x4 m2)
{
    v2f vout;
    // skinning
    vout.pos   =  vin.w1*(mul(m1,vin.pos)) +
                  vin.w2*(mul(m2,vin.pos));
    vout.color = vin.color;
    vout.uv    = vin.uv;
    return vout;
}
```

Note that “uniform” variables don’t come in via a connector.

Connecting Vertex Program to Fragment Program

- **Returned from vertex program**

name of connector

```
struct v2f
{
  float4 pos : POSITION;
  float4 color;
  float2 uv;
};
```

Must have POSITION

Defining a Fragment Program

predefined output connector

input connector

```
f2fb do_texture(v2f fragin,
                uniform sampler2D tex)
{
    f2fb fragout;
    fragout.COL = fragin.color * f4tex2D(tex, fragin.uv);
    return fragout;
}
```

- **Predefined output connector**

```
struct f2fb {
    half4 col       : COLOR;
    float depth     : DEPTH;
}
```

Optional: Specify Connector Registers

```
struct v2f
{
  float4 pos      : POSITION;
  float4 color    : TEXCOORD0;
  float2 uv       : TEXCOORD1;
};
```

These **semantics** allow mix & match with manually-written assembly code.

Vertex-Fragment Programs: Differences

- **Specific hardware limitations**
  - Removed in latest versions !!!
- **Vertex profiles**
  - No “half” or “fixed” data type
  - No texture-mapping functions
- **Fragment/pixel profiles**
  - No “for” or “while” loops (unless they’re unrollable)
  -

Features

- **Static function calls and static loops in all profiles**
- **Lots of library functions (e.g. cos, sin, sqrt)**
- **Non static loops and function calls in NV30**

Limitations

- **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**

Wrap-Up

- **Cg: C-like language**
  - succeeds assembly code
- **Vertex & fragment programs**
- **Different profiles for different HW**
- **Compiler optimizes usage of parallel pipelines**
- **Various HW-supported data types**
- **Supports vector and matrix operations**
- **Uniform vs. varying parameters**