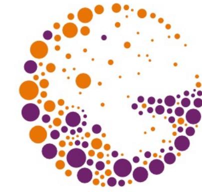

Jazyk Cg

© 2005-2012 Josef Pelikán, CGG MFF UK Praha
http://cgg.mff.cuni.cz/~pepca/
pepca@cgg.mff.cuni.cz

Obsah

- architektura Cg
  - Cg runtime
  - cílové profily, integrace s OpenGL a Direct3D
- jazyk Cg
  - datové typy, sémantika
  - příkazy, podprogramy
- efektivita shaderů
- příklady, ...

Vývojové prostředí Cg

- překlad programů pro **OpenGL** i pro **Direct3D**
  - Cg programy není v 99% případů nutno měnit
  - **integrace s 3D API** je ve dvou variantách (např. i komunikace „aplikace ↔ Cg shader“)
  - **překlad Cg programů** probíhá dynamicky, až v době běhu aplikace (optimalizace pro konkrétní GPU)
- **cílové profily** (≈ instrukční sady)
  - možnost přeložit Cg do různých profilů
  - výběr **optimálního profilu** pro aktuální GPU
  - **kompatibilita** Cg programů jedním směrem
  - zvlášť profily pro **vertex-** a pro **fragment-** programy

Prostředí Cg

Aplikace

OpenGL
Cg runtime

Direct3D
Cg runtime

Cg compiler

Core Cg runtime

3D API: OpenGL nebo Direct3D

GPU hardware

Jazyk Cg – základy

- **syntax Cg** je odvozena od ANSI C
  - inspirace v modernějších jazycích (C++, Java, C#)
  - Cg neobsahuje: `typedef`, `enum`, `union`, ukazatele, `goto`, `break`, `continue`, `do`, `switch`, `case`, `default`, ...
  - Cg má navíc: **sémantiku** parametrů, „**swizzle**” operátory, **selekce při zápisu** složek vektoru, **vektorové operace** („SIMD”), „**uniform**“ modifikátor
  - Cg má jinak: práce s poli (viz níže), deklarace proměnných (C++) a komentáře (C++)
  - datové typy: **float**, **half**, **fixed**, vektorové a maticové typy (**float3**, **float4x4**, ...), **sampler*** typy (texturové objekty)

Příklad vertex programu

```
struct VertOut
{
  float4 coord    : POSITION;
  float2 texCoord : TEXCOORD0;
  float4 diffuse  : COLOR0;
  float4 specular : COLOR1;
};

VertOut texLight ( float2 texCoord : TEXCOORD0,
                   float4 pos      : POSITION,
                   float4 normal   : NORMAL,
                   uniform float4x4 modelViewProj,
                   uniform float4x4 modelViewProjIT,
                   uniform float3 lightDir,
                   uniform float3 halfDir,
                   uniform float4 ambient,
                   uniform float4 specular )
{
  VertOut o;
  float3 N   = normalize( mul( modelViewProjIT, normal ).xyz );
  float cosa = dot( N, lightDir );
  float cosb = dot( N, halfDir );
  float4 light = lit( cosa, cosb, 32 );
  o.coord    = mul( modelViewProj, pos );
  o.texCoord = texCoord;
  o.diffuse  = ambient + light.yyyw;
  o.specular = specular * light.zzzw;
  return o;
}
```

Překlad do ARBvp1.0

```
!!ARBvp1.0
PARAM c[13] = { program.local[0..9],
                { 32 },
                program.local[11..12] };
TEMP R0;
  DP4  R0.z, vertex.normal, c[6];
  DP4  R0.y, vertex.normal, c[5];
  DP4  R0.x, vertex.normal, c[4];
  DP3  R0.w, R0, R0;
  DP4  result.position.w, vertex.position, c[3];
  DP4  result.position.z, vertex.position, c[2];
  RSQ  R0.w, R0.w;
  DP4  result.position.y, vertex.position, c[1];
  MUL  R0.yzw, R0.w, R0.xxyz;
  DP3  R0.x, R0.yzww, c[8];
  DP3  R0.y, R0.yzww, c[9];
  MOV  R0.z, c[10].x;
  DP4  result.position.x, vertex.position, c[0];
  LIT  R0.yzw, R0.xyzz;
  MOV  result.texcoord[0].xy, vertex.texcoord[0];
  ADD  result.color, R0.yyyw, c[11];
  MUL  result.color.secondary, R0.zzzw, c[12];
END
```

Příklad fragment programu

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

FragOut texColorGlass ( float2 texCoord : TEXCOORD0,
                        float4 diffuse  : COLOR0,
                        float4 specular : COLOR1,
                        uniform sampler2D tex,
                        uniform float4 colorGlass )
{
  FragOut o;
  float4 tmp = saturate( diffuse * tex2D( tex, texCoord )
                         + specular );
  o.color.rgb = lerp( tmp.rgb, colorGlass.rgb, colorGlass.a );
  o.color.a   = 1.0;
  return o;
}
```

Překlad do ARBfp1.0

```
!!ARBfp1.0
PARAM c[2] = { { 1 },
               program.local[1] };
TEMP R0;
TEMP R1;
  TEX      R0.xyz, fragment.texcoord[0], texture[0], 2D;
  MUL      R0.xyz, fragment.color.primary, R0;
  ADD_SAT  R0.xyz, R0, fragment.color.secondary;
  ADD      R1.xyz, -R0, c[1];
  MAD      result.color.xyz, R1, c[1].w, R0;
  MOV      result.color.w, c[0].x;
END
```

Profily překladu

- **výstupní instrukční sada**, abstraktní cílový procesor
  - aplikace spolu s Cg runtime se mohou až za běhu rozhodovat, pro jaký cílový kód se program přeloží
  - možnost lépe optimalizovat výsledek
  - profily se značně liší obecností, silou i svými limity (viz různá rozšíření OpenGL)
- oddělení **vertex programů** a **fragment programů**
  - zdrojový soubor může být jediný – s více vstupními body („entry points”)
- **přetížení funkcí** podle použitého profilu
  - varianty kódu – úplně pod kontrolou programátora

OpenGL profily

- pro **vertex** programy:
  - **arbvp1**: základní multivendor (`ARB_vertex_program`)
  - **vp20**: základní NVIDIA (`NV_vertex_program`)
  - **vp30**: pokročilý NVIDIA (`NV_vertex_program2`)
  - **vp40**: pokročilý NVIDIA (`NV_vertex_program3`)
- pro **fragment** programy:
  - **fp20**: základní NVIDIA (`NV_texture_shader`, `NV_register_combiners`)
  - **arbfp1**: pokročilý multivendor (`ARB_fragment_program`)
  - **fp30**: pokročilý NVIDIA (`NV_fragment_program`)
  - **fp40**: pokročilý NVIDIA (`NV_fragment_program2`)

DirectX profily

- pro **vertex** programy:
  - **vs_1_1**: základní multivendor (DirectX 8)
  - **vs_2_0**, **vs_2_x**: pokročilý miltivendor (DirectX 9)
- pro **fragment** programy:
  - **ps_1_1**, **ps_1_2**, **ps_1_3**: základní multivendor (DirectX 8)
  - **ps_2_0**, **ps_2_x**: pokročilý miltivendor (DirectX 9)

Přetížení podle profilu

- **profil** nebo jeho **maska** se předsazuje před funkci
  - „**vs**” nahrazuje „**vs_1_1**“, „**vs_2_0**“ nebo „**vs_2_x**”
  - „**vs_2**“ nahrazuje „**vs_2_0**“ nebo „**vs_2_x**”
- **příklad**:

```
arbvp1 VertOut main ( … )
{
  …
}

vp40 VertOut main ( … )
{
  …
  while ( i < MAX && p[i] > 0.0 ) {
    …
}
```

Číselné datové typy

- **int**
  - 32-bitové celé číslo, může být implementován jako „float"
- **float**
  - 32-bitů „single precision" (IEEE 754)
  - jediný povinný typ
- **half**
  - 16-bitový „low precision" float (IEEE 754)
  - může být implementován jako „float"
- **fixed**
  - málorozsahový „fixed-point" [-2,2], minimálně 10 desetinných bitů

Další datové typy

- **bool**
  - boolovská hodnota, „`false`“ nebo „`true`”
- **sampler***
  - handler na texturový objekt, implicitně „`const`”
  - `sampler` (generický), `sampler1D`, `sampler2D`, `sampler3D`, `samplerCUBE`, `samplerRECT`
- **array**, **packed array**
  - definice polí jako v C, modifikátor „`packed`”
  - povinně implementované vektory: `type1`, `type2`, `type3`, `type4` (`type` je libovolný číselný typ)
  - povinné matice: `type1x1` až `type4x4`

Literály, přetypování

- **číselné literály**: zapisují se jako v C
  - jednopísmenný suffix „`f`” (float), „`d`” (double), „`h`” (half), „`x`” (fixed)
- konstrukce vestavěných **vektorů**:
  - „`typ( x, y, … )`“
  - např. „`float4(0.0,1.0,0.0,1.0)`”
- **přetypování**:
  - explicitní přetypování jako v C „`(type)expression`”
  - sub-vektor: pomocí „swizzlingu” – např.:

```
float4 pos = { 0.5, 0.75, 0.2, 1.0 };
float3 vec = pos.xyz;
```

Kvalifikátory

- **const**
  - překladač může informaci využít k optimalizaci
- **in**, **out**, **inout**
  - parametry funkcí: „`in`” je default (volání hodnotou)
  - „`out`” – volání výsledkem, překladač kontroluje přiřazení (aspoň do jedné složky vektoru)
  - „`inout`” - kombinace „`in`“ a „`out`”
- **uniform**
  - globální proměnné nebo parametry, které jsou **inicializovány** externě (aplikace, dlouhodobější hodn.)
  - ostatní parametry jsou chápány jako „varying”

Preprocesor, komentáře

- klasická sada preprocesorových instrukcí z **jazyka C**
  - zejména „**#define**“, „**#if**“, „**#ifdef**“, „**#else**“ a „**#endif**”
  - není povinné implementovat „**#include**“ nebo makra v „**#define**”
- **komentáře**
  - jako v C i C++

```
/* short note */
// full-line comment
```

Sémantiky

- **identifikace dat** procházejících skrz „pipelines”
  - podobně jako v každém proudovém zpracování
  - musí se shodnout **FFP** (zbytek grafického adaptéru), **vertex program** a **fragment program**
  - zvláštní **identifikátory** v Cg
    - na každý program 2 „namespaces“ … pro vstup a výstup
- **syntax**: identifikátor oddělený dvojtečkou od parametru, globální proměnné nebo hlavičky funkce

```
in float4 norm : NORMAL;
in float4 pos  : POSITION;

out float psize : PSIZE = 1.0;
out float3 oPos : POSITION;
```

Standardní sémantiky

- některé sémantiky jsou **povinné** a přesně definované
  - např. „**POSITION**“ na výstupu vertex programu
- sémantiky se liší pro různé profily, **povinné sady**:
  - **vstup vertex programu**: **POSITION**, **NORMAL**, **TANGENT**, **BINORMAL**, **PSIZE**, **TEXCOORD0** až **7**, **BLENDINDICES**, **BLENDWEIGHT**
  - **výstup vertex programu**: **POSITION**, **PSIZE**, **COLOR0**, **COLOR1**, **TEXCOORD0** až **7**
  - **vstup fragment programu**: **PSIZE**, **COLOR0**, **COLOR1**, **TEXCOORD0** až **7**
  - **výstup fragment programu**: **COLOR**, option. **DEPTH**

Swizzling („permutace složek“)

- GPU umí zdarma **permutovat** složky vektorů
  - též lze složky **duplikovat** nebo **ignorovat**
    - moderní GPU: i negace nebo absolutní hodnota argumentů
- „**swizzle**“ operátor „`.`“
  - pro vektory: dvě rovnocenné sady znaků „**`xyzw`**“ a „**`rgba`**”
  - implicitní pořadí se uvádět nemusí
  - při zápisu („**write mask** operator”) se nesmí složky opakovat
- **maticový swizzling**
  - „**`_m<row><column>`**”, číslováno od **0**
  - „**`_<row><column>`**“, číslováno od **1** (kompat. s DirectX)

Swizzling – příklady

- **duplikace**

```
float4 a = b.xxyy;
float4 c = 1.0f.xxxx;   // same as float4(1.,1.,1.,1.)
a = c.wwww;
b = sin( a.w ).xxxx;
```

- **permutace**

```
a = b.wzyx;
b = (a + c).xzyw;
float4x4 M;
a = M._m00_m11_m22_m33; // same as M._11_22_33_44
float3 vecProd = a.yzx * b.zxy - a.zxy * b.yzx;
```

- **maska zápisu**

```
a.xyz = float3(2.0,2.0,2.0)
a.w = 1.0;
```

Řídící příkazy Cg

- **logické operátory**: `&& || ! < <= != == >= >`
  - fungují i s **vektorovými operandy** (po složkách)
  - **nepoužívá** se zkrácené vyhodnocování !
- **ternární operátor** `?:`
  - při vektorových operandech musí mít všechny vektory stejnou délkou, vše probíhá nezávisle po složkách
- `if else`
  - povinné pro všechny profily, ale starší HW ho neumí
- `for`, `while`
  - povinné, jestli lze počet opakování zjistit **v době překladu**, některé profily umějí i obecné cykly (vp40)

Funkce, příkaz discard

- **deklarace** jako v jazyce C
  - nepovinná návratová hodnota („`void`”)
  - nemusí mít žádné argumenty (používá globál. proměnné)
  - návratová hodnota je podobná „`out`” parametru – mohou se vracet i vektory nebo matice (formálně je to „rvalue“)
- „`static`“ funkce
  - nemůže být kompilovaná jako hlavní program
  - není přístupná z jiných souborů
- „`discard`“ příkaz – ruší zpracování fragmentu/vrcholu
  - DirectX profily používají místo něj funkci „`clip()`”

Vestavěné funkce

- vysoce **optimalizované**
  - pokud nejsou dnes realizovány jediným assemblerským příkazem, může to tak být v budoucích profilech
  - skoro vždy se **vkládají** do kódu („inlining”)
- funkce nemusí být podporovaná ve všech **profilech**
  - viz detailní referenci
- **okruhy** vestavěných funkcí:
  - aritmetika
  - geometrie
  - čtení z texturových objektů
  - diferenciály

Aritmetické funkce

- **„omezovače”**
  - abs, ceil, clamp, fmod, frac, frexp, max, min, modf, round, saturate
- **čistá aritmetika**
  - acos, asin, atan, atan2, cos, cosh, cross, degrees, determinant, dot, exp, exp2, ldexp, lerp, lit, log, log2, log10, mul(*,*), noise, pow, radians, rsqrt, sign, sin, sincos, sinh, smoothstep, step, sqrt, tan, tanh, transpose
- **predikáty**
  - all, any, isfinite, isinf, isnan

Texturové funkce

- první m argumentem je vždy objekt typu „**sampler***”
  - použitá funkce musí vždy souhlasit s **dimenzí** textury
  - jediné použití „**sampler***” objektů v Cg programech !
- **projektivní varianty** funkcí
  - před adresováním se provede převod z homogen. souř.
- další **pomocné argumenty**:
  - parciální **derivace**, **hloubka** „z” („depth compare”)
- **texturovací funkce** (počet [přetížených] variant):
  - tex1D(4), tex1Dproj(2), tex2D(4), tex2Dproj(2), texRECT(4), texRECTproj(2), tex3D(2), tex3Dproj(1), texCUBE(2), texCUBEproj(1)

Geometrické a ostatní funkce

- **geometrie**
  - distance, faceforward, length, normalize, reflect, refract
- **diferenciály**
  - ddx, ddy
- **ladění**
  - debug

Cg runtime a Cg API

- **Cg API**:
  - **překládá programy** z jazyka Cg do formy použitelné ve zvoleném 3D API (OpenGL nebo DirectX)
  - **spojuje** GPU programy s aplikací
    - aplikace může posílat „uniform“ i „varying“ data do GPU
    - GPU i zavedené GPU progr. správně interpretují tato data
- **dvě možnosti**:
  1. „**statický překlad**“ Cg programů (slinkuje se předem celá aplikace)
  2. použití „**Cg runtime**“ – překlad Cg při každém spuštění aplikace (pružnější a efektivnější řešení)

Cg runtime

Aplikační program

OpenGL Cg runtime

Direct3D Cg runtime

Cg compiler

Core Cg runtime

3D API: OpenGL nebo Direct3D

Cg runtime

- **výhody**:
  - **kompatibilita** do budoucna
    - i v dosud neexistujících profilech
    - lepší optimalizace v budoucích profilech
  - aplikace **není závislá** na profilu a konkrétním přeloženém kódu (může využít speciální schopnosti GPU)
  - **správa parametrů** – snadné předávání „uniform“ parametrů z aplikace do GPU
- **společné jádro** („Core Cg runtime“) + **varianty pro obě API** („cgGL“ a „cgD3D“)

Cg runtime pro OpenGL

- knihovny „**cg.dll**“ a „**cgGL.dll**” (podobně v Linuxu)

```
#include <Cg/cg.h>
#include <Cg/cgGL.h>
```

1. **Cg kontext** (OpenGL už musí běžet!)

```
CGcontext ctx = cgCreateContext();
```

2. **překlad Cg programu** (z paměti nebo souboru)

```
CGprogram vp = cgCreateProgram( ctx, CG_SOURCE,
                  programText, CG_PROFILE_VP20,
                  “main”, args );
```

nebo ze souboru

```
        cgCreateProgramFromFile( ctx, CG_SOURCE,
          "vertexProg.cg“, CG_PROFILE_VP20,
          "main", args );
```

Použití Cg runtime II

3. **upload přeloženého programu** do GPU

```
cgGLLoadProgram(vp);
```

4. **handle parametrů** („uniforms“)

```
CGparameter light =
   cgGetNamedParameter( vp, “lightVec“ );
```

nebo např.

```
   cgGetNamedParameter( vp, "main.lightVec“ );
```

5. **nastavení** „uniform“ **parametrů**

```
cgGLSetParameter4fv( light, value );
```

nebo třeba

```
cgGLSetParameter4f( light, 1.f, 0.f, 0.f, 1.f );
```

Použití Cg runtime III

6. **povolení** příslušného **profilu**

```
cgGLEnableProfile( CG_PROFILE_VP20 );
```

7. **přepnutí vertex programu** (mohu jich mít více)

```
cgGLBindProgram( vp );
```

8. **nyní všechny vykreslovací procedury používají daný vertex program !**

9. **zákaz** příslušného **profilu**

```
cgGLDisableProfile( CG_PROFILE_VP20 );
```

10. **úklid prostředků**

```
cgDestroyProgram( vp );
cgDestroyContext( ctx );
```

Efektivní programování v Cg

- počítat s **vektorovým počítáním** (SIMD)
  - překladač se snaží automaticky vektorizovat každý Cg kód, ale pokud na to programátor sám myslí, je to lepší..
- efektivně používat „**swizzling**“ – pomáhá vektorizaci
  - někdy jsou výhodné netradiční a nezvyklé postupy..
- používat **standardní funkce**
  - lepší kód i v budoucích profilech
- pro **velmi složité funkce** používat **předpočítané textury**
  - moderní GPU mají **velmi rychlé** čtení textur a dost texturovacích jednotek

Efektivní programování v Cg II

- používat datové typy s **minimální přesností** (která ještě dostačuje naší aplikaci)
  - „**fixed**“ je rychlejší než „**half**“, ten je rychlejší než „**float**”
- používat efektivní vestavěné funkce pro **stínování**
  - používat „**dot()**”
  - „**saturate()**” je ve fragment programu zdarma
  - používat „**lit()**”
- **konstantní hodnoty** přesunout do VP nebo na CPU
  - lineárně nebo téměř lineárně se měnící veličiny počítat ve VP a nechat si je interpolovat do fragmentů
  - zkoumat vždy možnost méně přesné, rychlejší varianty

Efektivní programování v Cg III

- **netransponovat** matici jen kvůli násobení!
  - `mul( transpose(m), v ) == mul( v, m )`
- minimalizovat délku **podmíněného kódu** ve fragmentových programech
  - profily **< fp40** neumějí opravdové podmíněné skoky
  - program se vykonává tak, že se spočítají **všechny větve** a až na konci se provede podmíněné přiřazení

CgFX formát

- textový formát pro výměnu shaderů i celých algoritmů
  - syntakticky identický s **.fx** (Microsoft)
  - místo HSLS jazyka je použitý **Cg**
- shadery ve **vyšším jazyce** i v **assembleru**
  - možnost popsat FFP algoritmus (FFP fázi)
- napojení na **GUI** („tweakable parameters“)
- propojení s **texturami** a **datovými soubory** (3D)
- CgFX obsahuje kompletní popis vizuálního „**efektu**”
  - víceprůchodové algoritmy, nastavení GPU stavu, apod.
  - starší terminologie: „effect file” (Microsoft)

CgFX „techniky“

- popis (konfigurace) efektu pro jeden **cílový profil**
  - fragment shader verze / FFP verze
  - jemné detaily / hrubý LoD
- syntax:

```
technique ShaderVersion
{ … };
```

- každá technika může obsahovat více **průchodů**
- každý průchod definuje použití **shaderů** na vrcholech, fragmentech – nebo **konfiguraci FFP**
  - typicky se v jedné technice nemíchá FFP se shadery („Z-fighting“)

Popis jednoho průchodu

- nastavení **zobrazovacího řetězce** (zápis do depth-bufferu, použití stencil-bufferu, apod.)
  - použitý vertex shader a fragment shader
- příklad:

```
pass firstPass
{
  DepthWriteEnable = true;
  AlphaBlendEnable = true;
  MinFilter[0] = Linear;
  MagFilter[0] = Linear;
  …
  PixelShader = asm
  {
    ps.1.1
    tex t0;
    mov r0, t0;
  };
};
```

Globální proměnné CgFX

- popis globálních proměnných (sémantika)
  - „proměnné“ CgFX souboru, propojení s aplikací přes „sémantiku”

```
float4x4 myViewMatrix : ViewMatrix;
texture2D texture     : DiffuseMap;
```

- použití v shaderech (na pozici „**uniform**“ parametrů):

```
pass firstPass
{
  ZEnable      = true;
  ZWriteEnable = true;
  CullMode     = None;
  …
  VertexShader = compile vp30 vsMy ( myViewMatrix, texture, … );
};
```

Anotace globálních proměnných

- jak má být hodnota proměnné („tweakable“) nastavována v **GUI aplikaci** (FX Composer, …)
  - píše se do špičatých závorek:

```
float4x4 bumpHeight
<
  string gui   = “slider“;
  float uimin  = 0.0f;
  float uimax  = 1.0f;
  float uistep = 0.1f;
> = 0.5f;
```

Literatura

- Randima Fernando, Mark J. Kilgard: ***The Cg Tutorial***, Addison-Wesley, 2003, ISBN: 0321194969

- **http://developer.nvidia.com/Cg/**