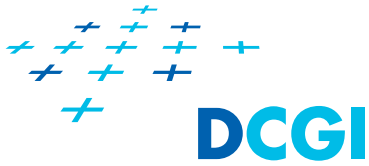


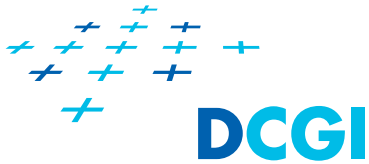
Základy programování shaderů v OpenGL část 1 – program

Petr Felkel, Jaroslav Sloup
Katedra počítačové grafiky a interakce, ČVUT FEL
místnost KN:E-413 (Karlovo náměstí, budova E)
E-mail: felkel@fel.cvut.cz

S použitím materiálů Vlastimila Havrana

Poslední změna: 20.2.2013

Obsah přednášky



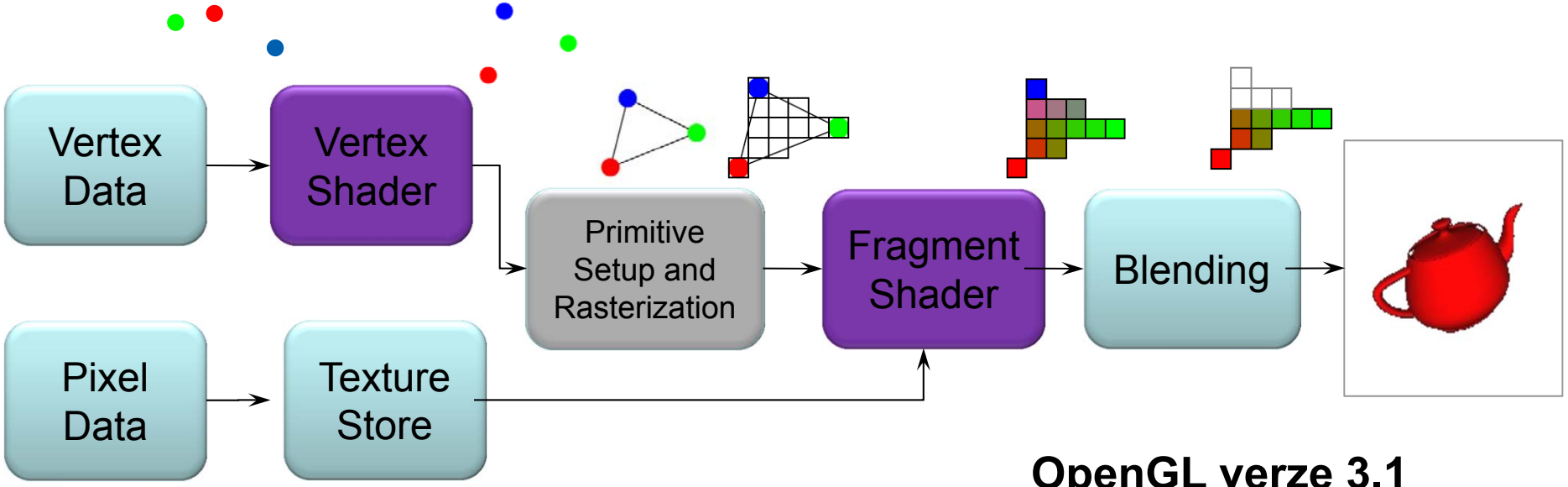
Dnes

- Tok dat v OpenGL se shadery
- Překlad a sestavení programu
- Struktura programu typu shader

Příště

- Předávání dat a parametrů do programu typu shader
- Příklady zdrojových kódů pro shadery

Hrubé schéma OpenGL se shadery

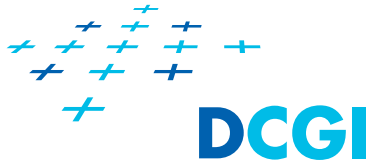


OpenGL verze 3.1

Každá aplikace definuje dva programy (tzv. shadery):

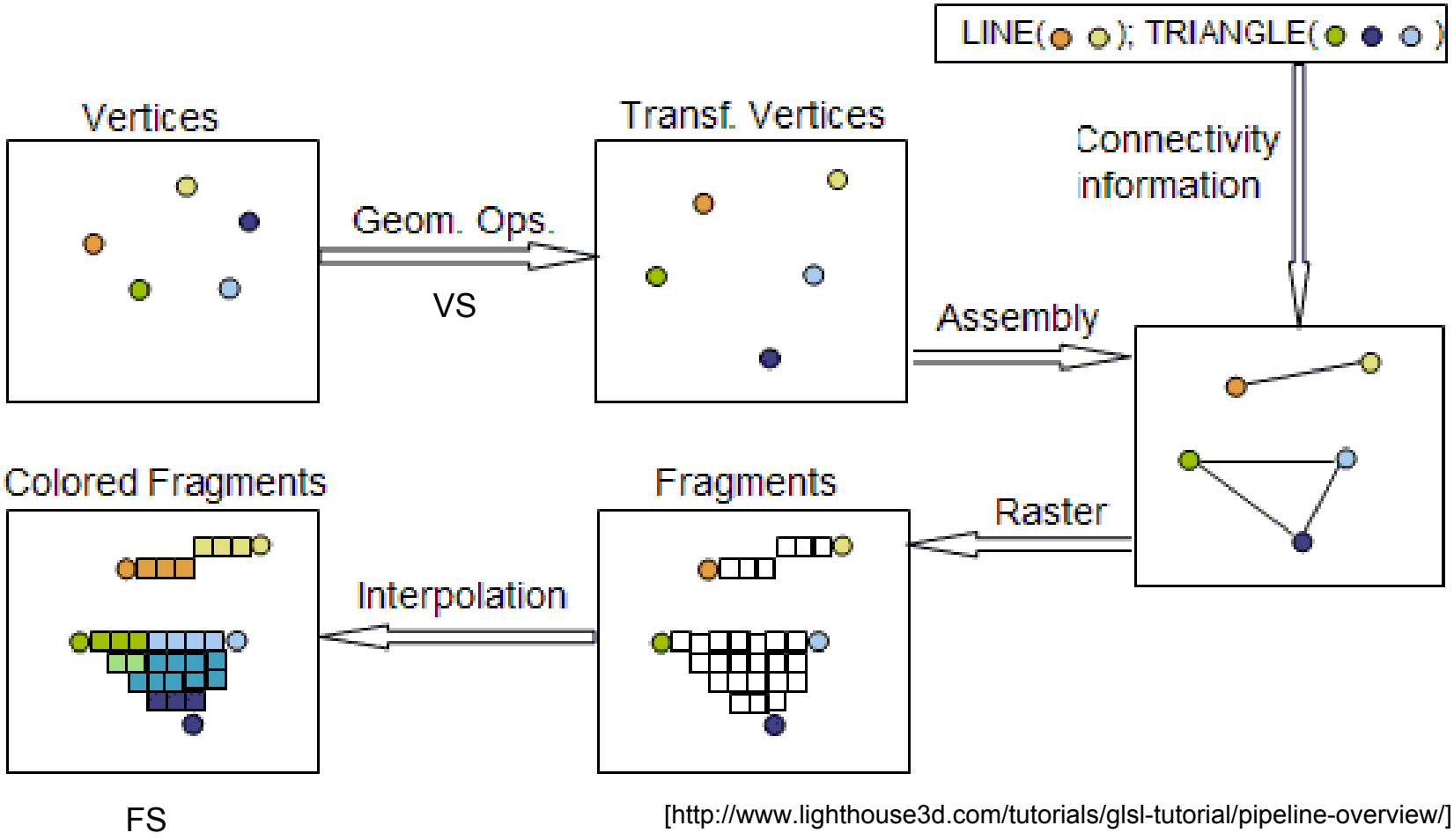
- **vertex shader (VS)**
  program zapsaný v jazyku GLSL zpracovává každý vrchol
- **fragment shader (FS)**
  program zapsaný v GLSL zpracovává tzv. fragment

Shadery kontra OpenGL



- Shadery jsou základem moderního programování grafiky
- Jsou to programy, které plně běží na grafické kartě GPU
- OpenGL na CPU „jen organizuje data a shadery“
- Shadery pracují paralelně – SIMD (*single instruction multiple data*)
  - VS po vrcholech
  - FS po fragmentech
- Fragment = kandidát na zobrazený pixel
  - Zda bude nakonec zobrazen rozhodne další část zobrazovacího řetězce
  - Obecně nese více informací, než jen zobrazenou barvu (hloubku, průhlednost, …)

Princip „zpracování primitiv na pixely“

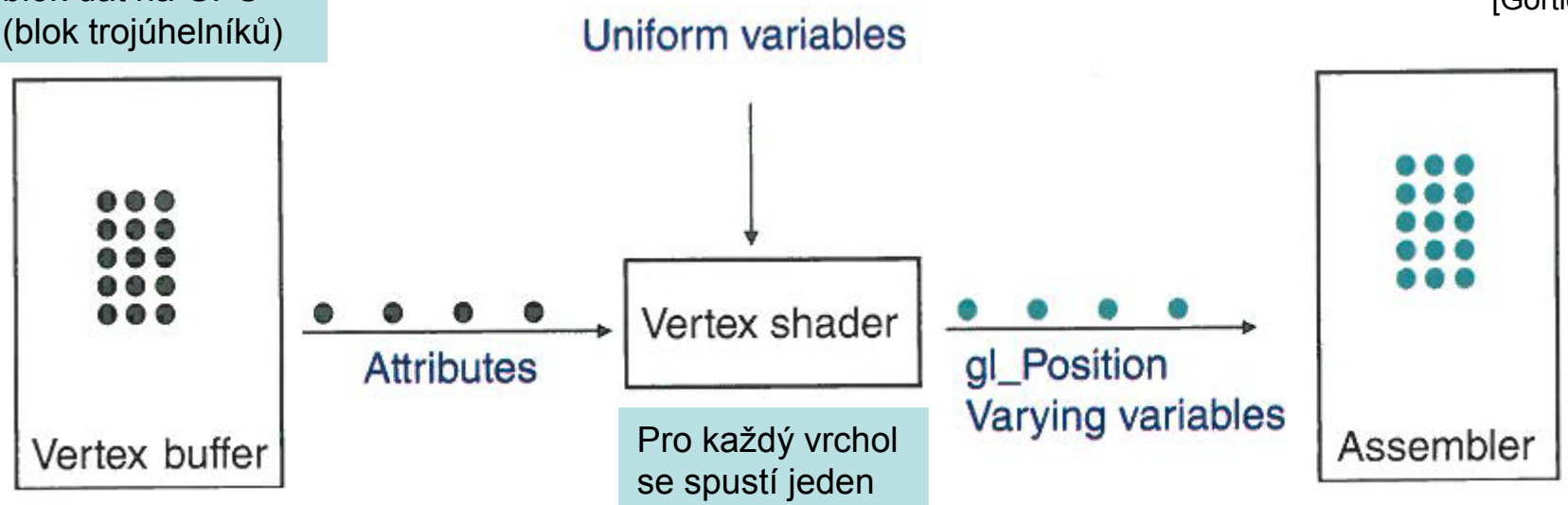


[http://www.lighthouse3d.com/tutorials/glsl-tutorial/pipeline-overview/]

Tok dat v OpenGL

OpenGL okopíruje blok dat na GPU (blok trojúhelníků)

[Gortler]



Pro každý vrchol se spustí jeden **Vertex shader**

Příkaz draw vykreslí trojice vrcholů (souřadnice a atributy)

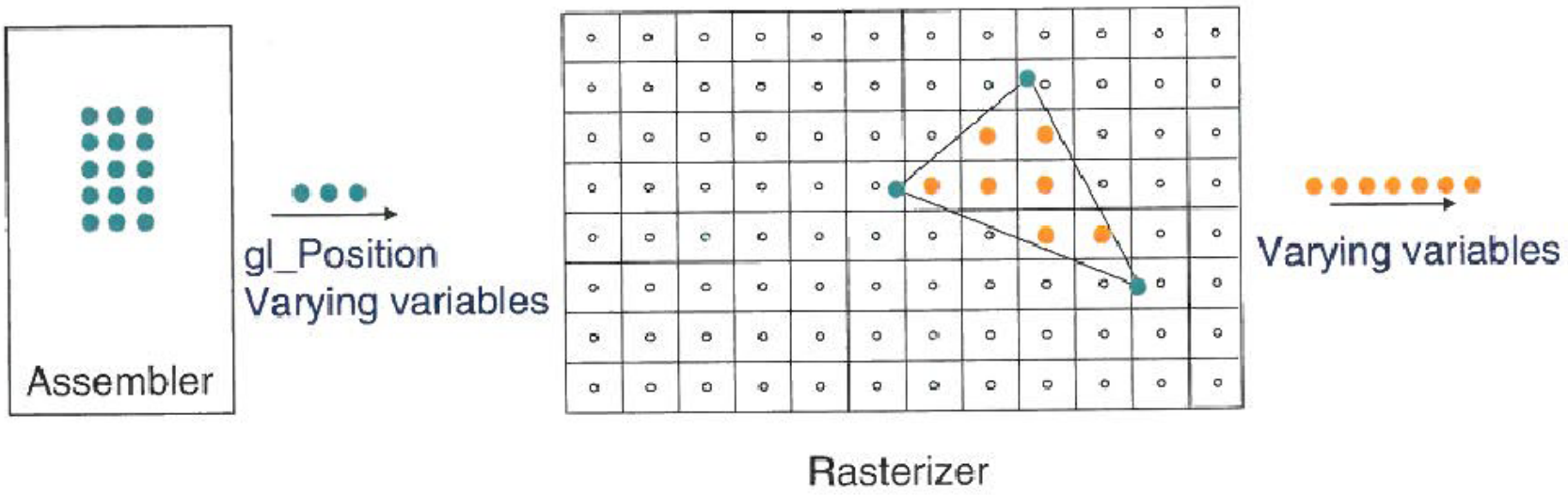
VS obdrží souřadnice a atributy vrcholu
+
Uniformní proměnné (společné více primitivům)

VS vypočítá normalizovanou 2D pozici vrcholu v okně (-1,-1) – (1,1)
+
další proměnné měnící se v ploše trojúhelníka

Blok **primitive assembler** poskládá zpět trojice vrcholů (trojúhelníky)

Tok dat v OpenGL

[Gortler]

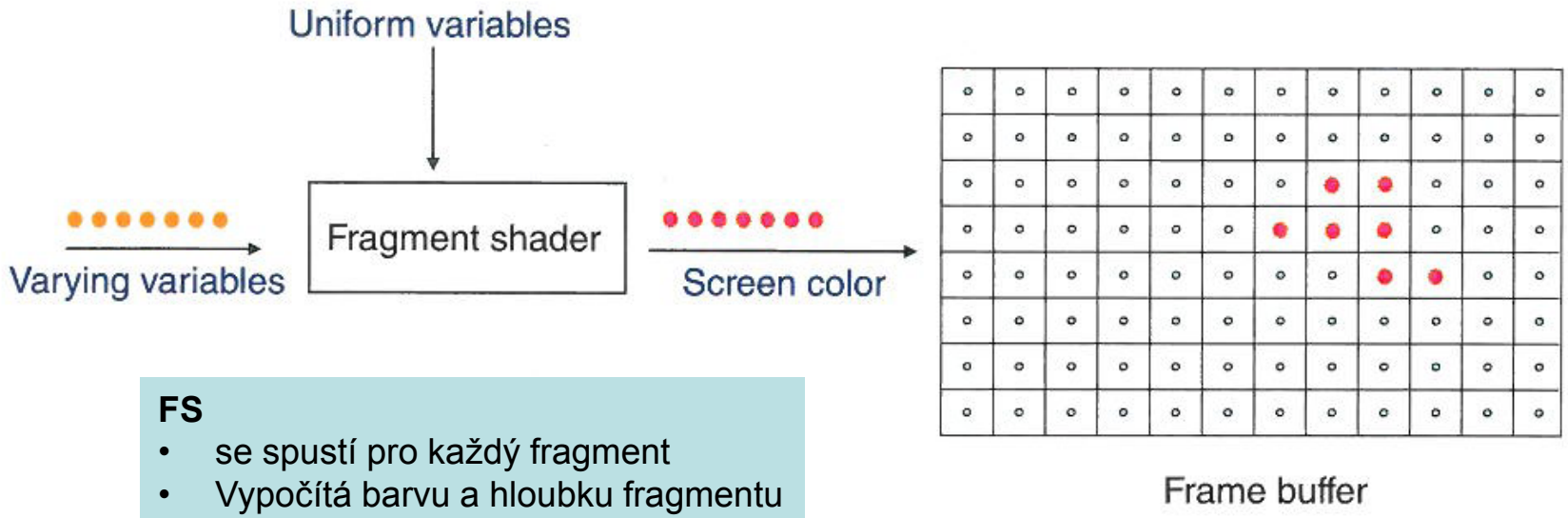


Blok **primitive assembler** předá trojice vrcholů (trojúhelníky) rasterizátoru

Rasterizer
- Vygeneruje fragmenty trojúhelníka
- Dále odešle pozici fragmentu a
- Interpolované hodnoty proměnných (varying) ve vrcholech

Tok dat v OpenGL

[Gortler]

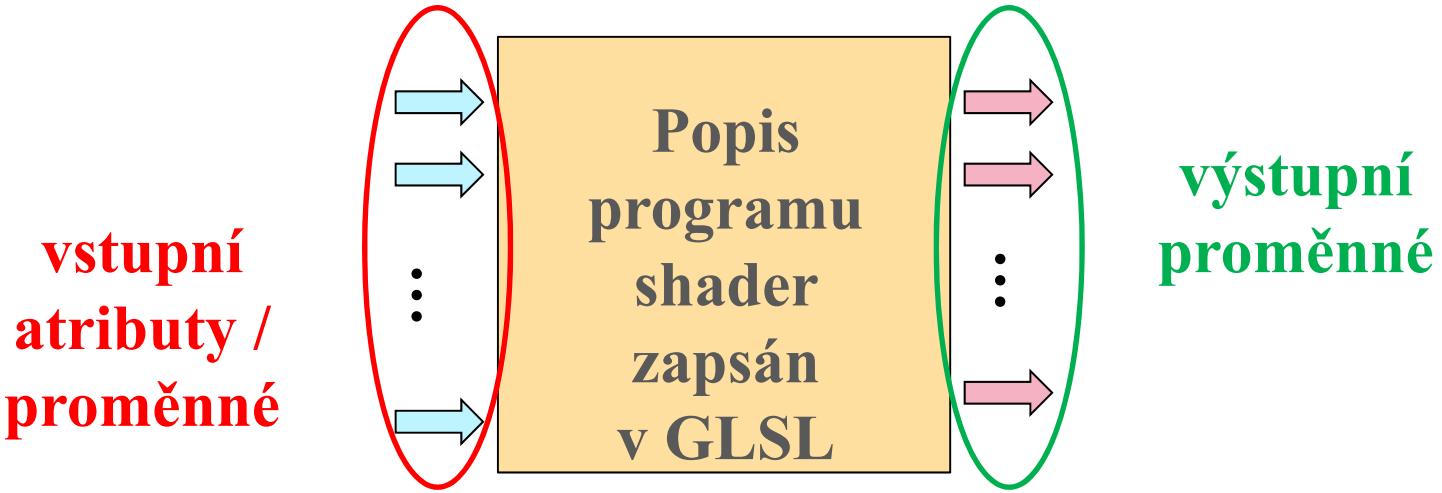


FS
- se spustí pro každý fragment
- Vypočítá barvu a hloubku fragmentu
- Odešle je do grafické paměti

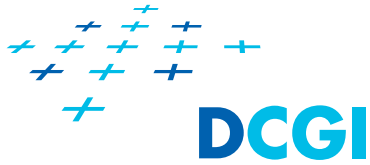
Fragment se stane **pixelem** na obrazovce, pokud projde testy
(výstřižek, šablona, alfa, hloubka)

Shadery

- **shader** je program, který běží na grafické kartě (GPU) jako součást zobrazovacího řetězce OpenGL
- OpenGL používá jazyk podobný jazyku C, který se jmenuje angl. **OpenGL Shading Language** (**GLSL**)



Shadery (typické úlohy)



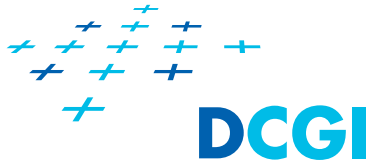
vertex shader (VS) – pracuje na úrovni **vrcholů**

- transformace vrcholu
- transformace a normalizace normály
- transformace souřadnic pro textury případně i jejich generování
- stínování metodou per-vertex
  (výpočet barvy ve vrcholu, která se následně interpoluje - Gouraud)
- změna atributů asociovaných s vrcholy

fragment shader (fs)– pracuje na úrovni **fragmentů**

- přístup k textuře a další zpracování textury
- mlha, míchání barev, sčítání barev, stínování metodou per-pixel
  (výpočet barvy v každém fragmentu)
- operace na interpolovanými hodnotami

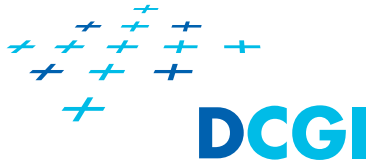
Moderní programování v OpenGL



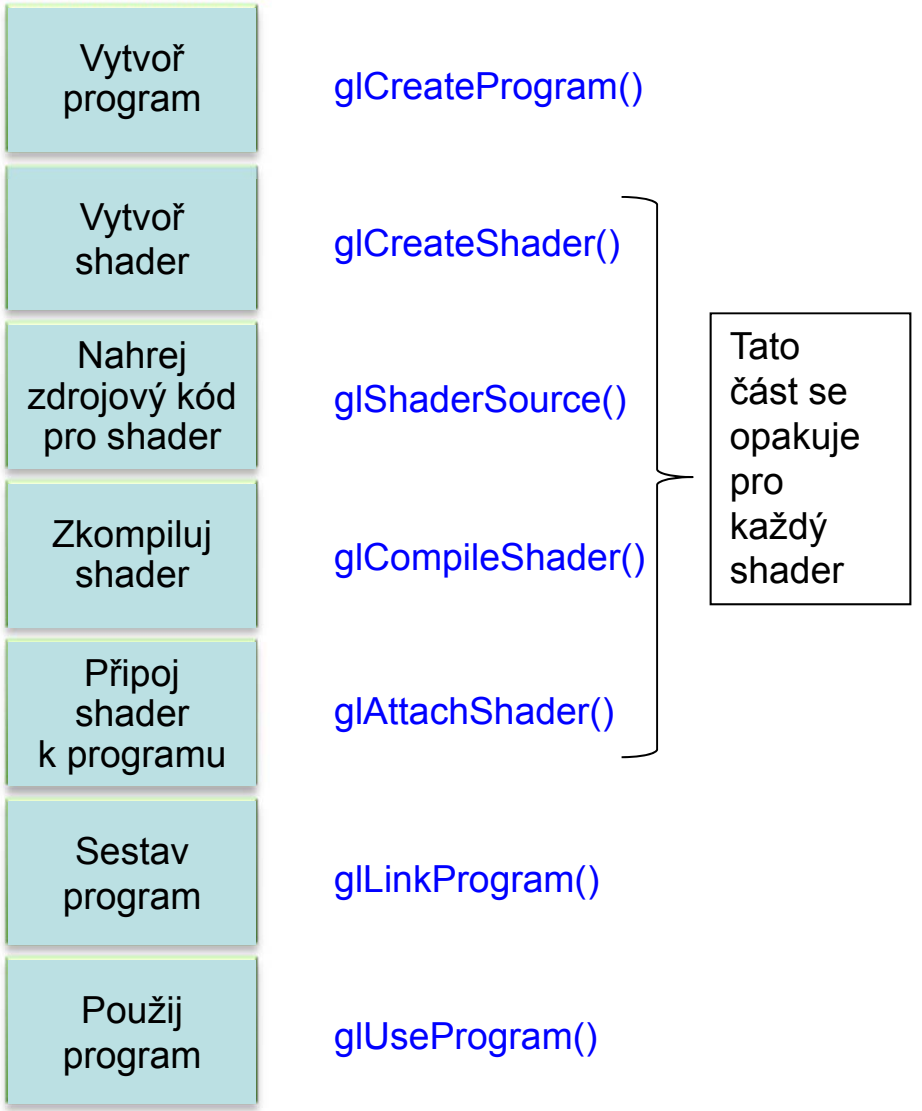
Moderní programy v OpenGL v zásadě provádějí pouze tyto čtyři kroky:

1. Vytvoř všechny programy typu „shader“ (ty běží na GPU)
2. Vytvoř „buffer objects“ a nahraj do nich data
3. Propoj data a proměnné v programech typu „shader”
4. Spusť vykreslování, obrázek je vypočítán a zobrazen.

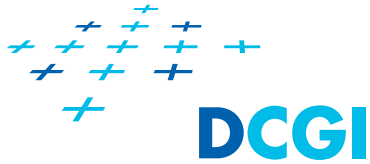
Programy s použitím shaderu v OpenGL



- Před použitím musí být shadery přeloženy a sestaveny
- Překládá se až za běhu aplikace na konkrétním HW
- OpenGL má proto příkazy pro kompilátor a linker programu
- Každý program musí obsahovat:
  - vertex shader
  - fragment shader
  - (případně další shader)



Zápis programu typu shader – dvě možnosti

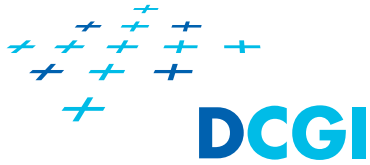


Tělo shaderu

- je typicky uloženo ve zvláštních souborech
  - vertex shader přípona *.vert*, .vs, …
  - fragment shader *.frag, .fs, …*

- lze jej vytvářet i přímo v kódu jazyka C z textových řetězců.

Práce se shadery v kódu aplikace

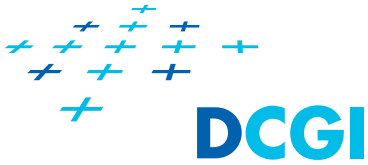


I. Kroky při překladu shaderů:
   1. Vytvoř objekt typu shader *(shader object)*
   2. Předej mu zdrojový kód programu
      (formou textového řetězce nebo pole řetězců)
   3. Přelož zdrojový kód shaderu uvnitř objektu typu shader
      (*compile*)
   4. Zkontroluj výsledek (*status*) překladu

II. Kroky při sestavení programu (*vertex a fragment shader)*
   5. Vytvoř objekt programu
   6. Připoj objekty shaderů k objektu programu
   7. Sestav program (*link*)
   8. Zkontroluj status sestaveného programu

III. Použití programu v aplikaci

I. Překlad programu typu shader

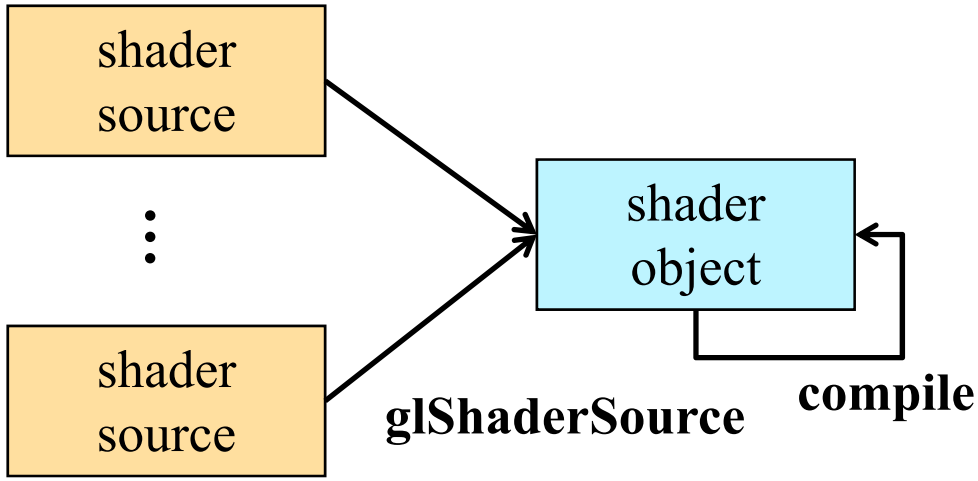


I. Kroky při překladu shaderů:

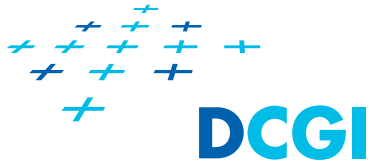
1. Vytvoř objekt typu shader *(shader object)*
2. Předej mu zdrojový kód programu
   (formou textového řetězce nebo pole řetězců)
3. Přelož zdrojový kód shaderu uvnitř objektu typu shader
4. Zkontroluj výsledek (*status*) překladu

Příklad zdrojového programu pro vertex shader

```
#version 400
in vec3 VertexPosition;
in vec3 VertexColor;
out vec3 Color;
void main() {
  Color = VertexColor;
  gl_Position = vec4(VertexPosition, 1.0);
}
```



1. Vytvoření objekt typu shader



Jméno shaderu = kladná hodnota odkazující na vytvořený *shader objekt*

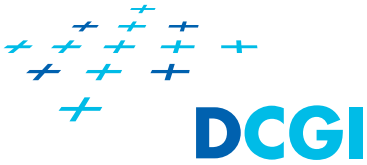
Typ vytvářeného shaderu **GL_VERTEX_SHADER** nebo **GL_FRAGMENT_SHADER**

```
GLuint vertShader = glCreateShader( GL_VERTEX_SHADER );

if  (vertShader == 0) {    // check for error
    fprintf(stderr, "Error creating vertex shader.\n");
    exit(1);
}
```

Objekt shaderu udržuje zdrojový kód a přeložený shader.

- Po sestavení programu (*link*) lze objekt shaderu smazat
- Odkazujeme na něj jménem (kladná celočíselná hodnota)

2. Nahrání zdrojového kódu



Funkce sloužící k nahrání textů zdrojového kódu shaderu z externího souboru

```
const GLchar * shaderCode = loadShaderAsString("basic.vert");
const GLchar* codeArray[] = {shaderCode};

glShaderSource( vertShader, 1, codeArray, NULL );
```

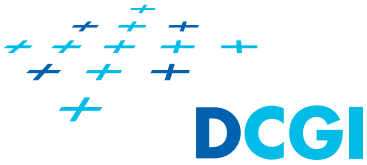
Jméno objektu shader

Počet řetězců zdrojových kódů

Pole ukazatelů na řetězce → zdrojový text shaderu lze poskládat z více částí

Pole hodnot Glint → délky jednotlivých řetězců zdrojových kódů (NULL indikuje, že řetězce jsou ukončeny nulovým znakem 0x00)

2. Nahrání zdrojového kódu



Funkce sloužící k nahrání textů zdrojového kódu shaderu z externího souboru

```
const GLchar * shaderCode = loadShaderAsString("basic.vert");

glShaderSource( vertShader, 1, &shaderCode, NULL );
```

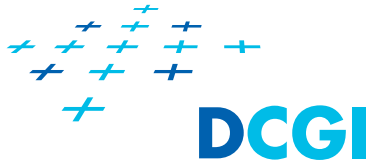
Jméno objektu shader

Počet řetězců zdrojových kódů

Reference na odkaz na řetězec (celý program je v jediném řetězci)

NULL indikuje, že jediný řetězec je ukončen nulovým znakem 0x00

3. – 4. Překlad a kontrola správnosti




```
glCompileShader(vertShader);
GLint result;
glGetShaderiv(vertShader, GL_COMPILE_STATUS, &result);
if (result == GL_FALSE) {
   fprintf( stderr, "Vertex shader compilation failed!\n" );
   GLint logLen;
   glGetShaderiv(vertShader, GL_INFO_LOG_LENGTH, &logLen );
   if ( logLen > 0 ) {
      char * log = (char *)malloc(logLen);
      GLsizei written;
      glGetShaderInfoLog(vertShader, logLen, &written, log);
      fprintf(stderr, "Shader log:\n%s", log);
      free(log);
   }
}
```

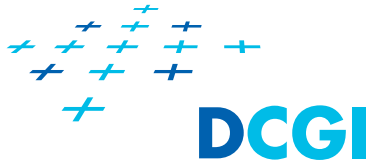
Překlad zdrojového kódu v objektu shaderu

Dotaz na výsledek překladu

Dotaz na délku chybové zprávy o překladu

Získání textové zprávy o chybě

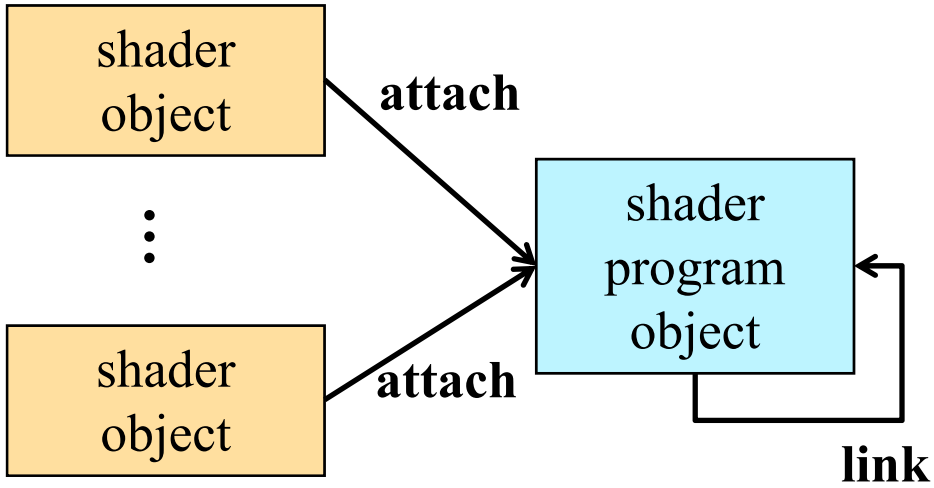
II. Sestavení objektu typu program se shadery



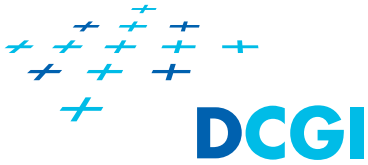
- Přeložené shadery musí být sestaveny do **shader programu** (*vertex* + *fragment shader*)
- Kroky k sestavení shader programu:
  5. Vytvoř objekt programu
  6. Připoj shadery k objektu typu program (attach)
  7. Sestav program (link)
  8. Zkontroluj výsledek sestavování programu

Příklad zdrojového programu fragment shader

```
#version 400
in vec3 Color;
out vec4 FragColor;
void main() {
  FragColor = vec4(Color, 1.0);
}
```



5. Vytvoř objekt typu program



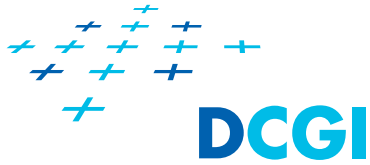
Nenulová hodnota k identifikaci vytvořeného program objektu

Vytvoř prázdný objekt programu

```
GLuint programHandle = glCreateProgram();

if (programHandle == 0)  {// check for error
  fprintf(stderr, "Error creating program object.\n");
  exit(1);
}
```

Objekt typu program v OpenGL obsahuje kód všech shaderů použitých v programu (*alespoň vertex + fragment shader*).

6-7. Připoj shader k objektu typu program



Připoj vertex shader k programu

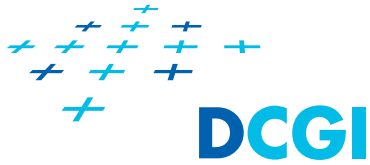
Připoj fragment shader k programu

```
glAttachShader(programHandle, vertShader);
glAttachShader(programHandle, fragShader);

glLinkProgram(programHandle);
```

Sestav spustitelný program „typu .gpuexe“, který běží na dvou komponentách GPU:

- programovatelný vertex processor (*je-li shader objekt typu GL_VERTEX_SHADER připojen k programu*)
- programovatelný fragment processor (*je-li shader objekt typu GL_FRAGMENT_SHADER připojen k programu*)

8. Zkontroluj stav sestaveného programu



Příklad části programu v C++

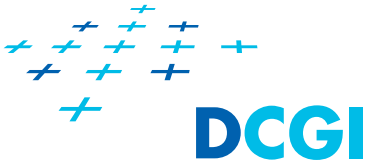
```
GLint status;
glGetProgramiv(programHandle, GL_LINK_STATUS, &status);
if (status == GL_FALSE) {
   fprintf( stderr, "Failed to link shader program!\n" );
   GLint logLen;
   glGetProgramiv(programHandle, GL_INFO_LOG_LENGTH, &logLen);
   if ( logLen > 0 ) {
      char * log = (char *)malloc(logLen);
      GLsizei written;
      glGetProgramInfoLog(programHandle, logLen, &written, log);
      fprintf(stderr, "Program log: \n%s", log);
      free(log);
   }
}
```

Zkontroluj zda sestavení proběhlo v pořádku

Dotaz na délku chyb. zprávy o sestavení

Získej textové zprávy o chybě sestavení programu a vypiš je

III. Nahraj program do zobrazovacího řetězce OpenGL




```
if (status == GL_FALSE) {
}
else {
  glUseProgram(programHandle);
}
```

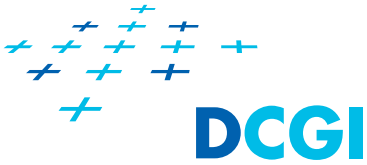
Instaluj program s připojenými shadery do zobrazovacího řetězce na GPU

Příklad části programu v C++

Jméno objektu programu, který se nyní bude provádět v zobrazovacím řetězci → program je připraven pro spuštění a zobrazovaní obrazu na výstup

Objektů typu program lze vytvořit několik a poté mezi nimi přepínat s použitím příkazu glUseProgram() v rámci běhu jednoho programu.
Programy přitom mohou mít připojené stejné shadery

Struktura zdrojového kódu shaderu



Verze GLSL (zde 4.0)

vstupní atributy/proměnné shaderu

výstupní proměnné shaderu

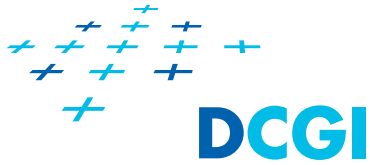
Funkce main(), která je spouštěna pro každý vrchol nebo fragment

Příklad zdrojového programu pro vertex shader:

```
#version 400

in vec3 VertexPosition;
in vec3 VertexColor;

out vec3 Color;

void main() {
    Color = VertexColor;
    gl_Position = vec4(VertexPosition, 1.0);
}
```

Vestavěná výstupní proměnná (prefix gl_)

Kvalifikátory proměnných



Kvalifikátor uložení v paměti	Význam
in	Vstup atributu (VS), nebo propojení na výstup předchozího bloku (FS - výstup rasterizátoru)
out	Výstupní hodnota shaderu – propojí se s dalším blokem (VS s rasterizátorem, FS s blokem testů)
uniform	Hodnota společná více primitivům. Je přístupná ve všech shaderech (VS i FS). Nastavuje ji OpenGL (v C++) před příkazem draw.

- Způsob interpolace hodnot v rasterizátoru se nastavuje interpolačním kvalifikátorem (pro out ve VS a pro in FS):

Interpolační kvalifikátor	význam
smooth	perspektivně správná interpolace hodnot
flat	bez interpolace
noperspective	lineární interpolace

Propojení shaderů navzájem

- Propojení VS a FS (FS dostávají interpolované hodnoty)

Příklad zdroj. kódu vertex shader

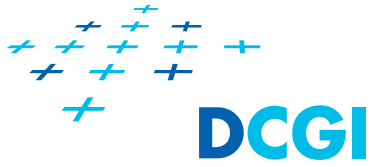
```
#version 400
in vec3 VertexPosition;
in vec3 VertexColor;
out vec3 Color;
void main() {
  Color = VertexColor;
  gl_Position = vec4(VertexPosition, 1.0);
}
```

Jméno a datový typ musí být stejný

Příklad zdroj. kódu fragment shader

```
#version 400
in vec3 Color;
out vec4 FragColor;
void main() {
  FragColor = vec4(Color, 1.0);
}
```

Připojení atributů = vstupů vertex shaderu



Každá vstupní proměnná vertex shaderu **má** vlastní celočíselný **index (*location*)**. Ten lze nastavit:

- V OpenGL aplikaci před sestavením shader programu příkazem glBindAttribLocation()

```
glBindAttribLocation(programHandle, 0, "VertexPosition");
```

Index (location) atributu

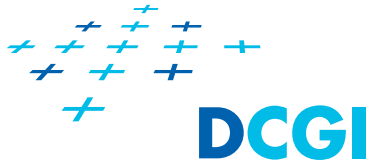
Jméno atributu

- V shaderu pomocí kvalifikátoru layout:

```
layout (location = 0) in vec3 VertexPosition;
```

- Pokud hodnotu indexu (*location*) explicitně nenastavíme, je index vygenerován při sestavení programu automaticky

Připojení atributů = vstupů vertex shaderu



- Pokud je index atributu přiřazen automaticky linkerem při sestavení programu, lze se na jeho hodnotu dotázat příkazem:

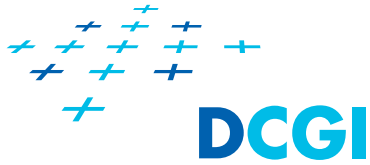
```
glGetAttribLocation(programHandle, "VertexPosition");
```

vrací index atributu (=location, typ GLint)

jméno atributu

- Index atributu (*location*) nabývá hodnot
  - 0,1,2,,…, pokud byl atribut ve VS „aktivně“ použit
  - –1 pokud nebyl v textu VS nalezen, nebo v nebyl v shaderu použit a byl proto vypuštěn

Typ datových proměnných



základní datové typy:

- **float**, **double**, **bool**, **int**, **uint** *stejně jako v jazyce C*

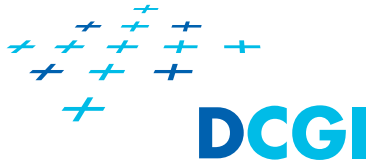
vektory s 2, 3 nebo 4 složkami:

- **vec**{2,3,4} *{2,3,4}-složkový vektor s jednoduchou přesností (float)*
- **dvec**{2,3,4} *{2,3,4}-složkový vektor s dvojitou přesností (double)*
- **bvec**{2,3,4} *{2,3,4}-složkový vektor s hodnotami bool*
- **ivec**{2,3,4} *{2,3,4}-složkový vektor s celočíselnými hodnotami*

čtvercové matrice:

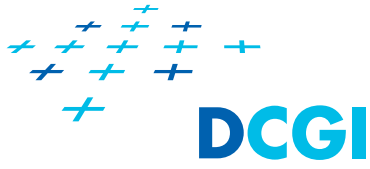
- **mat2, dmat2** *matice 2×2 s jednoduchou či dvojnásobnou přesností*
- **mat3, dmat3** *matice 3×3 dtto v pohyblivé řádové čárce*
- **mat4, dmat4** *matice 4×4 dtto*

Deklarace a inicializace proměnných



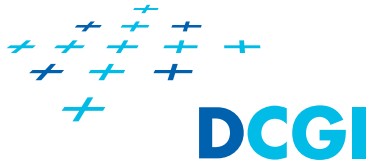

```
int       a = 2;              // c is initialized with 2
bool      d = true;           // d is true

float     b = 2;              // incorrect, no automatic type casting supported
float c = float(a);           // correct. c is 2.0

vec3 f;                       // declaring f as a vec3
vec3 g = vec3(1.0, 2.0, 3.0);
mat4 m = mat4(1.0)            // initializing the diagonal of the matrix with 1.0
mat2 k = mat2(1.0,0.0,1.0,0.0);                 // all elements are specified

float frequencies[3] = float[](3.4, 4.2, 5.0);   // initialize array of floats

struct dirLight {                               // type definition
    vec3 direction;
    vec3 color;
};
dirlight d1;
dirlight d2 = dirlight(vec3(1.0,1.0,0.0),vec3(0.8,0.8,0.4)); // structure initialization
```

Přístup ke složkám vektorových proměnných




```
vec3 pos = vec3(1.0, 2.0, 3.0);
float f = 1.2;     double c = 2.0LF;  // float and double constants
pos.x              // is legal, the same as pos.r
pos.xy             // is legal, the same as pos.rg
pos.w              // is illegal
f.x                // is legal, the same as f.r or f.s
f.y                // is illegal
const int L = pos.length();          // number of components in vector
// the order of the components can be different to swizzle them, or replicated
vec3 swiz= pos.zyx;                  // swizzled  = (3.0, 2.0, 1.0)
vec4 dup = pos.xxyy;                 // duplicated = (1.0, 1.0, 2.0, 2.0)
vec4 dup = f.xxxx;                   // duplicated = (1.2, 1.2, 1.2, 1.2)
// accessing matrix components
mat4 m;
m[1] = vec4(2.0);          // sets the second column to all 2.0
m[2][3] = 2.0;             // sets the 4th element of the third column to 2.0
```

Vestavěné funkce v GLSL



trigonometrické funkce

- radians(), degrees(), sin(), cos(), atan(), …

exponenciální funkce

- pow(), exp(), log(), sqrt(), …

geometrické funkce pro vektory

- **dot**(), **cross**(), length(), **normalize**(), reflect() …

maticové funkce

- **transpose**(), determinant(), inverse()

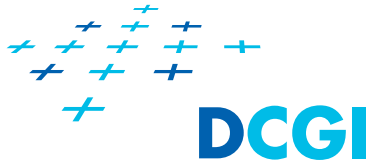
funkce pracující s vektory relačně po složkách:

- bvec equal(vec x, vec y), bvec greaterThan(vec x, vec y)

běžné algebraické funkce

- abs(), floor(), min(), max(), **mix**(), …

Uživatelské funkce v rámci shaderu



- Definice funkce může být přetížena pokud jsou parametry funkce rozdílné
- Chování rekurzivních funkcí není definováno

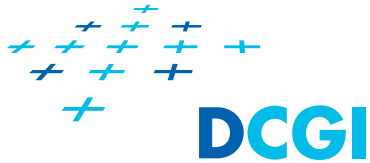
Uživatelsky definovaná funkce s identifikátorem funkce12

Vestavěná výstupní proměnná (prefix gl_)

```
in vec3 VertexPosition;
in float VertexStartTime;
out float Transp;
uniform float ParticleLifetime;

void funkce12(float Time) {
    float age = Time - VertexStartTime;
    Transp = 0.0;
    if(Time >= VertexStartTime)
        Transp = 1.0 - age / ParticleLifetime;
    gl_Position = MVP * vec4(VertexPosition, 1.0);
}
```

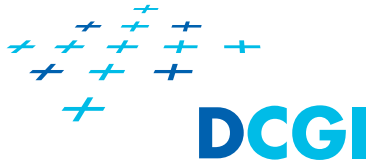
PGR

Příště



- Předávání dat z aplikace a mezi shadery

Zajímavé odkazy



- David Wolff: ***OpenGL 4.0 Shading Language Cookbook***. Packt Publishing, 2011, ISBN 978-1-849514-76-7.
- Richard S. Wright, Nicholas Haemel, Graham Sellers, Benjamin Lipchak: ***OpenGL SuperBible: Comprehensive Tutorial and Reference***. 5th ed., Addison-Wesley Professional, 2010, ISBN 0-321-71261-7.
- Ed Angel, Dave Shreiner: *An Introduction to Modern OpenGL Programming*, SIGGRAPH 2011 tutorial, http://www.daveshreiner.com/SIGGRAPH/s11/Modern-OpenGL.pptx
- Joe Groff. *An intro to modern OpenGL*. Updated July 14, 2010 http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Table-of-Contents.html
- Vertex Array Object na OpenGL Wiki: http://www.opengl.org/wiki/Vao
- Vertex Specification na OpenGL Wiki: http://www.opengl.org/wiki/Vertex_Specification