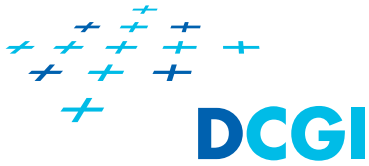


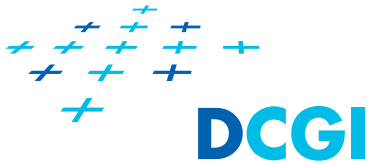
Základy programování shaderů v OpenGL
Část 2 - přenos dat

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



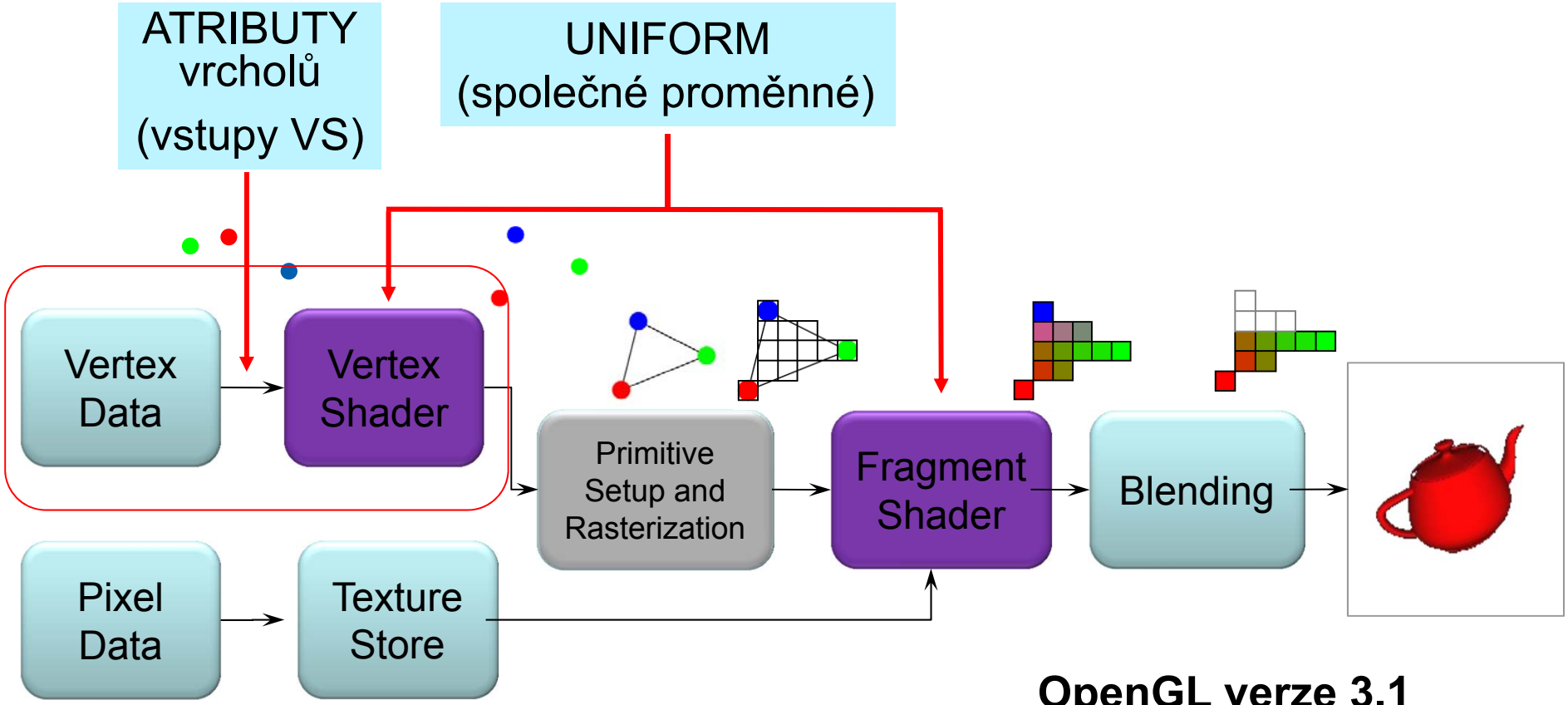
Minule

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

Dnes

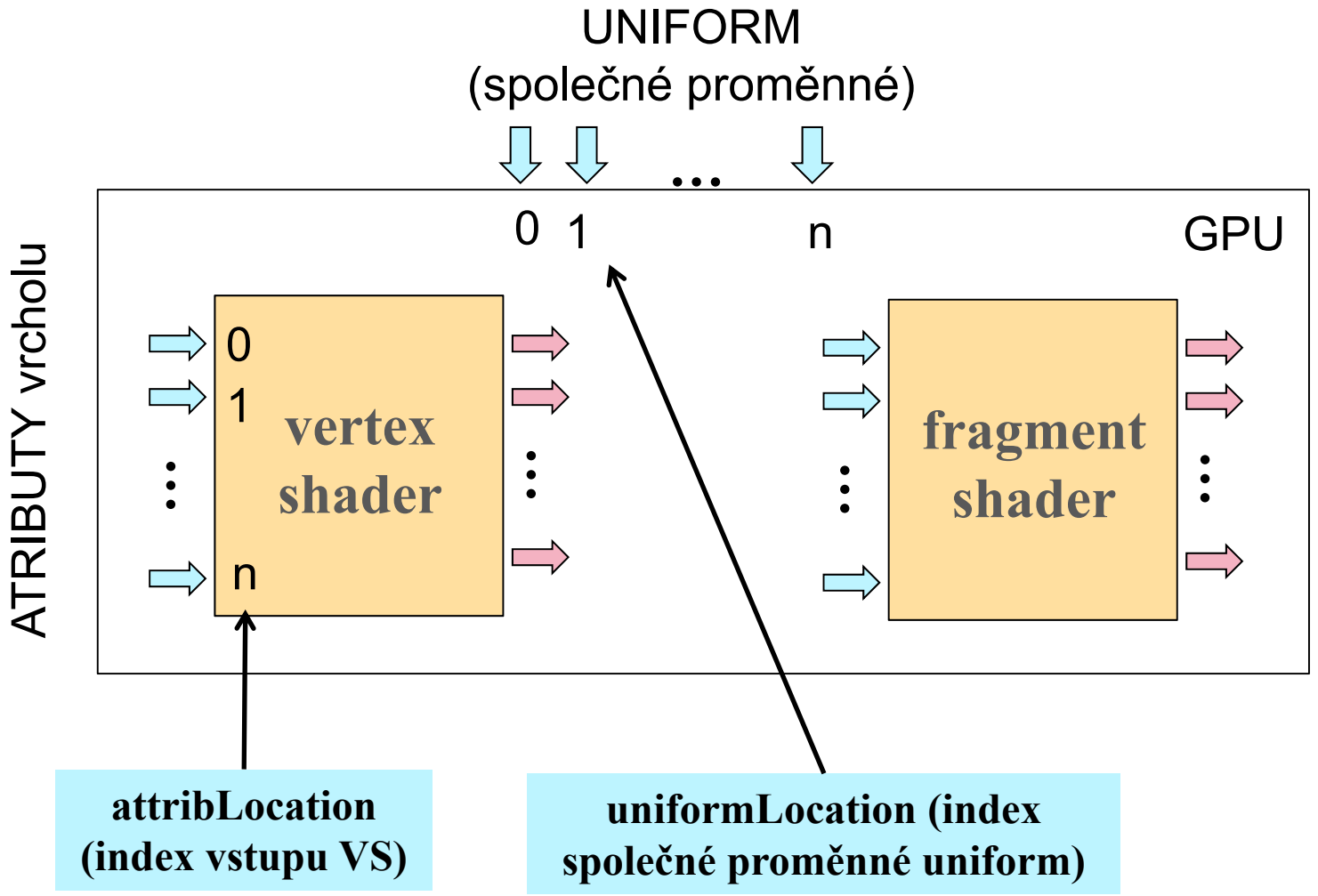
- Předávání dat a parametrů do programu typu shader
- Grafická primitiva
- Příklady zdrojových kódů pro shadery

Předávání parametrů shaderům



OpenGL verze 3.1

Atributy vrcholu a proměnné uniform



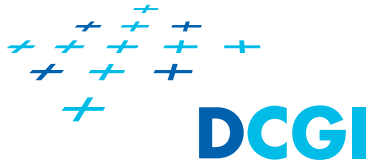
Propojení dat vrcholů na vstupy VS

Atributy vrcholů (hodnoty uložené pro každý vrchol)

- tvoří vstupy vertex shaderů (xyz, rgb, n,…)
- v shaderu označeny kvalifikátorem “**in**”
  in vec3 **VertexPosition**;
- pro každý vrchol se spustí jeden VS
- VS dostane data právě pro jeden vrchol

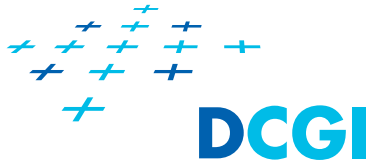
- aplikace musí předem připravit hodnoty pro tyto atributy:
  - uloží hodnoty pro všechny vrcholy do tzv. **buffer objektu (VBO)**
  - popíše, jak jsou v bufferu uloženy **array buffer (VAO)**
  - připojí obsah bufferu na vstupní proměnné **attribLocation**

Objekty v OpenGL



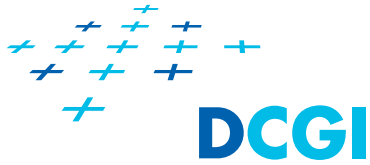
- OpenGL využívá princip **objektů**
- Všechny objekty mají **celočíselné jméno** (ID, handle, …)
- Pomocí něj aktivujeme příslušný objekt či jej použijeme pro připojení k jinému objektu.
- Z hlediska aplikace je struktura objektu neznámá
- Jméno slouží k předávání objektu funkcemi API
  - **buffer** jako pojmenovaný blok bytů v paměti
  - **vertex array** jako kolekce bufferů a stavu OpenGL spojených se vstupem vrcholů s atributy do Vertex Shaderu (jak jsou v bufferech uloženy souřadnice a atributy a jak je propojit na vstupy VS)
  - **textura**
  - Podobně se chovají shader a program, i když ty objekty nejsou

Práce s objekty v OpenGL



- Při vytváření objektu
  1. gl**Gen**XXX vytvoření jména objektu --> vždy typ GLuint
  2. gl**Bind**XXX target, object_name
  3. Copy data - např. glTexImage2D, glBufferData,,…,
  4. + nastavení stavu OpenGL
- Při použití
  - gl**Bind**XXX target and object_name
- Pokud objekt nepotřebujeme
  - glDelete

Předávání dat mezi GPU a CPU v OpenGL



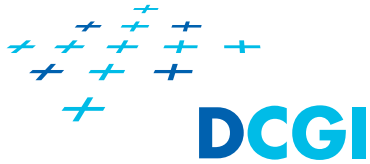
Probíhá prostřednictvím těchto objektů

- Buffer Objekty (pro vrcholy nazýváme VBO)
- Array Objekty (pro vrcholy nazýváme VAO)

VBO je principielně jen jednorozměrné pole dat, data asociovaná k vrcholům, ale nemusí to být jen souřadnice vrcholů

VAO obsahuje jeden čí více VBOs a dává intepretační strukturu datům v nich uložených

Další obsah přednášky



- Buffer objekty
- Vertex array objekty
- Použití a nastavení pro zobrazení dat

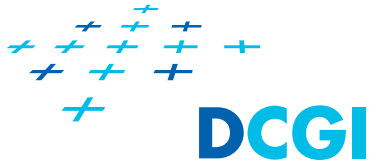
(Vertex) Buffer objekty

- Slouží k přístupu do paměti spravované OpenGL obvykle na GPU (CPU = klient, GPU = server)
- Je to blok bajtů bez dalších informací o jejich struktuře
- Typický postup vytvoření:

```
float buffer_data [] = {-0.8f, -0.8f, 0.0f, 0.8f, -0.8f, 0.0f};
GLuint buffer_name;    // často jen buffer
glGenBuffers(1, &buffer_name );                    // 1
glBindBuffer(target, buffer_name );                // 2
glBufferData(target, sizeof(buffer_data),
                buffer_data, GL_STATIC_DRAW); // 3
glBindBuffer(target, 0);
```

- `target` je způsob interpretace bufferu, např. GL_ARRAY_BUFFER, ale další jsou GL_ELEMENT_ARRAY_BUFFER pro indexy, GL_PIXEL_PACK_BUFFER, or GL_PIXEL_UNPACK_BUFFER pro pixely

(Vertex) buffer objekty – funkce na vytvoření

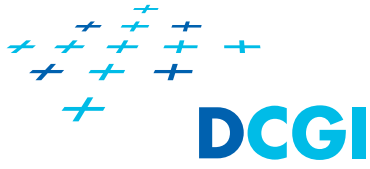


- Pro vytváření objektů bufferů se může hodit tato funkce, která vrací jméno vytvořeného bufferu

```
static GLuint make_buffer(
    GLenum target,      // typ bufferu
    GLsizei buffer_size
    const void *buffer_data,
  )
{
    GLuint buffer;
    glGenBuffers(1, &buffer);          // 1
    glBindBuffer(target, buffer);      // 2a
    glBufferData(target, buffer_size,// 3
                 buffer_data, GL_STATIC_DRAW);
    glBindBuffer(target, 0);           // 2b
    return buffer;   // vrací jméno bufferu - ID
}
```

[Groff]

Buffer Objekty – 1. generování jmen



void **glGenBuffers**(GLsize *n*, GLuint * *buffers*); `// 1`

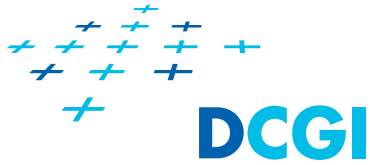
- Vygeneruje *n* jmen bufferů, jméno je číslo typu Gluint
- Uloží je do pole *buffers*
- `glGenBuffers(1,&buffer); // 1`

  Vytvoří jedno jméno bufferu

void **glDeleteBuffers**(GLsize *n*, const GLuint * *buffers*);

- Uvolní *n* jmen bufferů v poli *buffers* a smaže je

Buffer Objekty – 2. propojení jména s cílem



void **glBindBuffer**(enum target, GLuint buffer); // 2

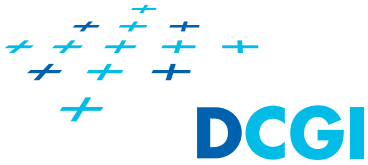
- Propojí jméno objektu buffer s OpenGL cílem target

target	Účel
ARRAY_BUFFER	Atributy vrcholů (Vertex array)
ELEMENT_ARRAY_BUFFER	Indexy do pole vrcholů
…	

- target se používá i jako parametr příkazů OpenGL
- Propojení ovlivňuje i další příkazy, které target jako parametr nemají, ale využívají jím nastaveného stavu (např. glDrawArrays pracuje s navázanými vrcholy, glVertexAttribPointer nastavuje připojený buffer ve VAO)
- Volá se podruhé při vykreslování – před použitím bufferu
- Propojení se zruší parametrem 0

```
glBindBuffer(target,0);
```
PGR `// 2b`

Buffer Objekty – 3. předání dat



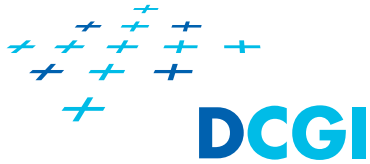

```
void glBufferData(enum target, sizeiptr size,
                  const void * data, enum usage);   // 3
```

- Okopíruje data z klientské paměti do paměti buffer objektu

usage – rada, kam data dát	**Účel**
STATIC_...	Data nehodláme měnit (rigidní objekty)
DYNAMIC_...	Data budeme měnit často
STREAM_…	Data budeme měnit pravidelně (animace)
…_DRAW	CPU -> GPU
…_READ	GPU -> CPU
…_COPY	Propojka oběma směry

- Pro vertex a element arrays tedy typicky `GL_STATIC_DRAW`
-
```
glBufferData(target, buffer_size,
             buffer_data, GL_STATIC_DRAW);   // 3
```

Vertex Array Object (VAO)

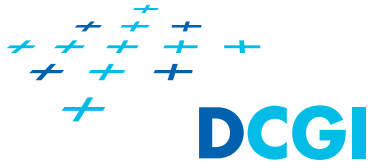


- Sdružuje kompletní informace o atributech vrcholů potřebné pro předávání dat vertex shaderům
  - Připojené buffery se souřadnicemi a dalšími atributy
  - Informace o struktuře dat v bufferech (jak jsou data v bufferech poskládána)
  - Navázání na vstupy vertex shaderů (*location*)
- V nových verzích OpenGL je nutno VAO explicitně definovat (gen, bind,…) a před vykreslením se provede jen propojení (bind)

Poznámka: v OpenGL ES a WebGL VAO není

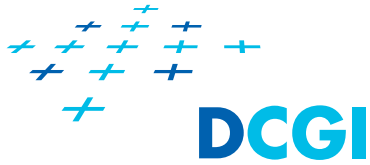
- ve starší verzi OpenGL existoval implicitní VAO s číslem 0
- propojení se vstupy (`glBindBuffer` a `glVertexAttribPointer`) se dělalo v metodě draw() [Groff]

Vertex Array Object (VAO) - vytvoření




```
GLuint vertex_array;
glGenVertexArrays( 1,  &vertex_array );     // 4
glBindVertexArray( vertex_array );          // 5a

// buffery s atributy vrcholů - vytvořeny dříve
glBindBuffer( GL_ARRAY_BUFFER, positionBuffer );
glEnableVertexAttribArray( pos_location ); // 6
glVertexAttribPointer( pos_location, 2,GL_FLOAT, GL_FALSE,0,0);// 7
glBindBuffer( GL_ARRAY_BUFFER, 0 );

glBindBuffer( GL_ARRAY_BUFFER, colorBuffer );
glEnableVertexAttribArray(color_location); // 6
glVertexAttribPointer(color_location,3,GL_FLOAT,GL_FALSE,0,0); // 7
glBindBuffer(GL_ARRAY_BUFFER, 0);

// pouze jeden buffer s indexy vrcholů!
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, elementBufferNameHouse2 );

glBindVertexArray( 0 ); // 5b detach
```

Vertex Array Object – 4. generování jmen



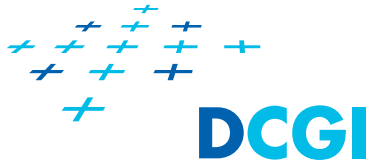
void **glGenVertexArrays**(GLsize *n*, GLuint * *arrays*); `// 4`

- Vytvoří *n* jmen polí v polí *arrays*

void **glDeleteVertexArrays** (GLsize *n*, const GLuint **arrays*);

- Uvolní *n* jmen polí v polí *arrays* a jejich obsah smaže

Vertex Array Objects – 5. připojení

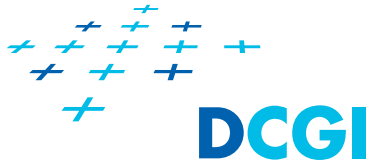


void **glBindVertexArray** (GLuint **array**); `// 5`

- Připojí pole se jménem **array**
- Při prvním volání se objekt pole vytvoří
- Volá se podruhé při vykreslování – před použitím v poli definovaných bufferů a v nich uložených atributů vrcholů
- (nemá parametr target – existuje jediný typ vertex array)
- Připojení se zruší parametrem 0

```
glBindVertexArray(0);                    // 5b
```

Vertex Array Objects –
6. Povolení atributu se zadaným indexem



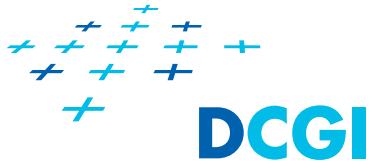

```
void glEnableVertexAttribArray(GLuint index); // 6
```

- Povolí atribut (location) s indexem **index**
- Každý vrchol dostane na vstup vertex shaderu jiná data odpovídající jeho umístění v bufferu

```
void glDisableVertexAttribArray(GLuint index);
```

- Zakáže atribut (location) s indexem **index**
- Všechny vrcholy dostávají konstantní vstupní hodnotu
  - Implicitně (0.0, 0.0, 0.0, 1.0)
  - Nebo příkazem glVertexAttrib*(Gluint index, TYPE value);

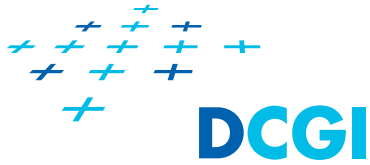
Vertex Array Objects –
7. Popis pole atributů vrcholů pro VS




```
void glVertexAttribPointer(GLuint index, GLint size,
                           GLenum type, GLboolean normalized,
                           GLsizei stride, const GLvoid * pointer);   // 7
```

- **index** atributu ve vertex shaderu
  (location atributové proměnné)
- **size** - počet složek atributu (1, 2, 3 nebo 4)
- **type** - datový typ složek (GL_BYTE, GL_FLOAT,…)
- **normalized** převádí celá čísla na interval 0.0f ...1.0f
- **stride -** offset mezi atributy v bufferu
- **pointer** - offset první hodnoty v rámci bufferu přetypovaný na (void *) – pro data od začátku bufferu (void*)0

Postup vykreslení jednoho snímku



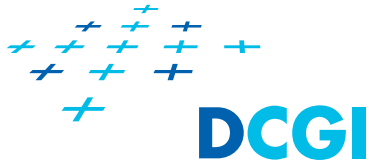
Příprava

- Nastaví se stav OpenGL
- Připraví se balíky dat – přenos do GPU (buffery)
- Připraví se informace o obsahu bufferů a jejich propojení na vstupy vertex shaderů (vertex arrays)

Vykreslování (např. 60x za sekundu)

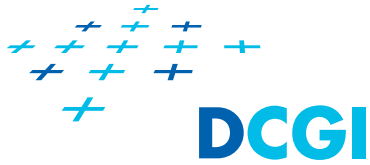
- Smazání obrazovky
- Propojení vstupů VS (bind vertex array)
- Spustí se vykreslení (draw)

Dva způsoby vykreslování geometrických primitiv



- Data o vrcholech jsou uložena v buffer objektech, tj. v polích v adresovém prostoru serveru (GPU)
- Jedním příkazem lze vybrat část z nich a vykreslit je (např. jednu stěnu krychle)
  - glDrawArrays() vykreslí *n* po sobě jdoucích vrcholů
  - glDrawElements() vykreslí *n* po vrcholů napřeskáčku, podle pole indexů vrcholů
- Před vykreslováním je nutno informovat OpenGL o tom, jak jsou data v polích poskládána, jak jsou propojena se vstupy vertex shaderů a v jakém pořadí se budou vrcholy uložené v datech vykreslovat - to je uloženo ve Vertex Array
  - glBindVertexArray(…) => připojit Vertex Array

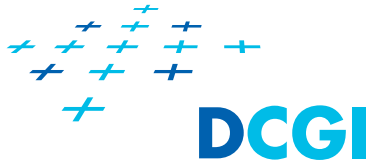
Vykreslení *n* po sobě jdoucích vrcholů




```
void glDrawArrays(GLenum mode,
                  GLint first, GLsizei count);
```

- Vykreslí **count** po sobě jdoucích vrcholů ze všech povolených polí
  - Začne na vrcholu s indexem **first**
  - Poslední vykreslený má index **first+count-1**
- **mode** udává typ primitiva GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES

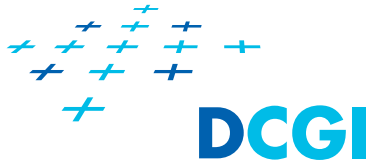
Vykreslení *n* po vrcholů napřeskáčku



void **glDrawElements**(GLenum **mode**, GLsizei **count**,
GLenum **type**, void ***indices**);

- Vykreslí **count** vrcholů ze všech připojených polí ARRAY_BUFFER, jejichž indexy jsou uloženy v připojeném poli indexů ELEMENT_ARRAY_BUFFER na offsetu **indices**
  (dříve to byl opravdu odkaz na pole indexů v klientské paměti, proto se offset přetypovává na odkaz `(void*) offset`)
- **type** udává datový typ indexů a může nabývat hodnot GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT
- **mode** udává typ primitiva GL_POINTS. GL_LINES, …

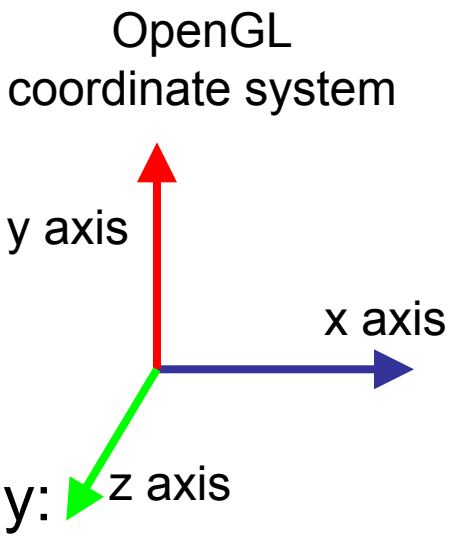
Použití vertex arrays



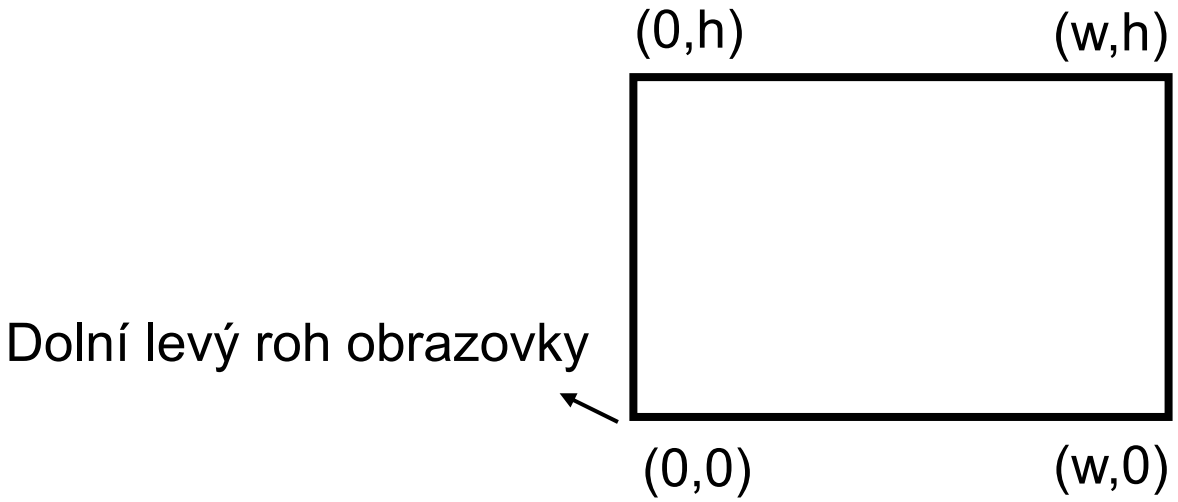

```
glBindVertexArray( vertex_array );       // 5
glDrawArrays( GL_LINES, 7, 777 ); //od 7.bytu, 777 hodnot
glBindVertexArray( 0 );                  // 5
```

Souřadnicový systém v OpenGL - základy

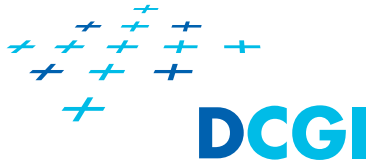
- Pravotočivá souřadná soustava



- Souřadnicový systém zobrazované plochy:



Souřadnice vrcholů



- **Souřadnice vrcholů**
  - Typicky se s nimi pracuje v polích vrcholů
  - Předávají se přes ANSI C pole do tzv. vertex *buffer objektů*
- Počet složek souřadnic

  2 $\Rightarrow$ 2D souřadnice (x, y), *z je implicitně rovno 0 and w rovno 1*

  3 $\Rightarrow$ 3D souřadnice (x, y, z), *w je implicitně rovno 1*

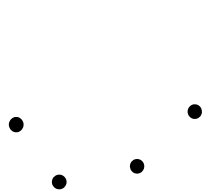
  4 $\Rightarrow$ homogenní souřadnice (x, y, z, w) – vlastně reprezentují 3D souřadnice podle vztahu (x/w, y/w, z/w) **- vhodnější pro transformace, detaily později.**

Vnitřní reprezentace vrcholů v homogenních souřadnicích (x, y, z, w).

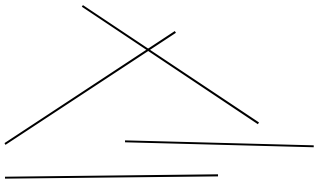
specifikace (x, y, z) $\Rightarrow$ reprezentace (x, y, z, 1)

(x, y) $\Rightarrow$ reprezentace (x, y, 0, 1)

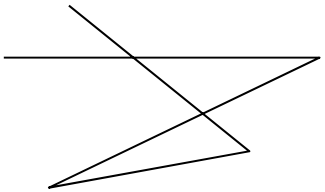
Grafická primitiva v OpenGL



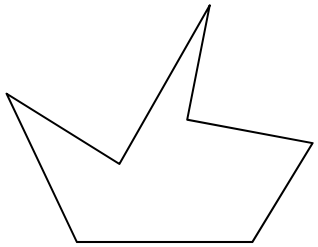
GL_POINTS



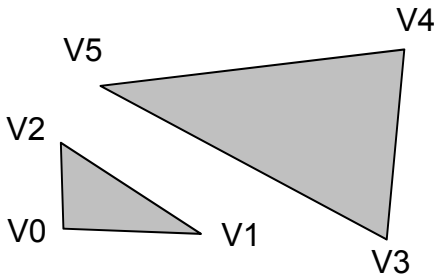
GL_LINES



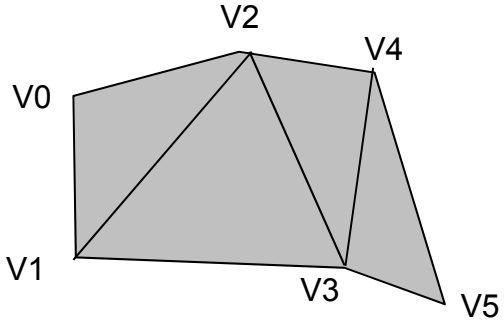
GL_LINE_STRIP



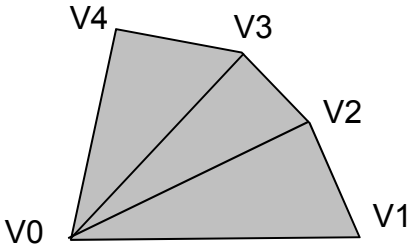
GL_LINE_LOOP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

PLUS (pro geometry shader a teselace):

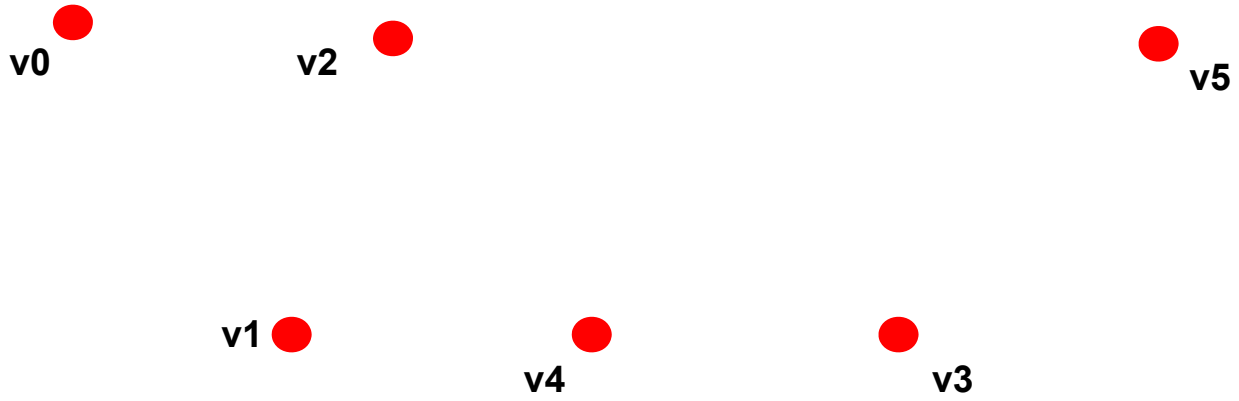
GL_LINES_ADJACENCY GL_TRIANGLES_ADJACENCY GL_TRIANGLE_STRIP_ADJACENCY

GL_PATCH_VERTICES

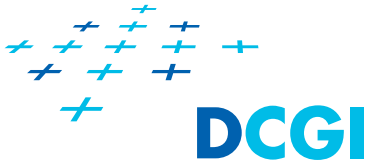
Body

GL_POINTS

- Vrcholy jsou specifikovány v poli, vrcholů je libovolný počet
- Lze specifikovat velikost bodu v počtu pixelů
- Nezáleží tedy na vzdálenosti bodu ke kameře, velikost bodu bude vždy stejná.

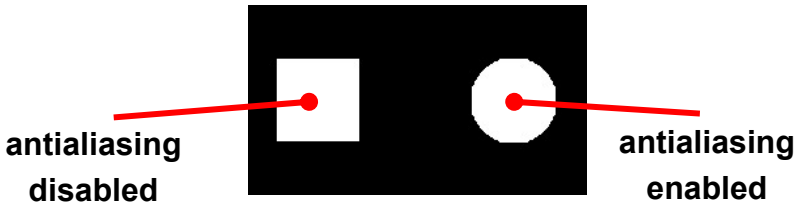


Velikost bodu



void **glPointSize(**GLfloat **size);**

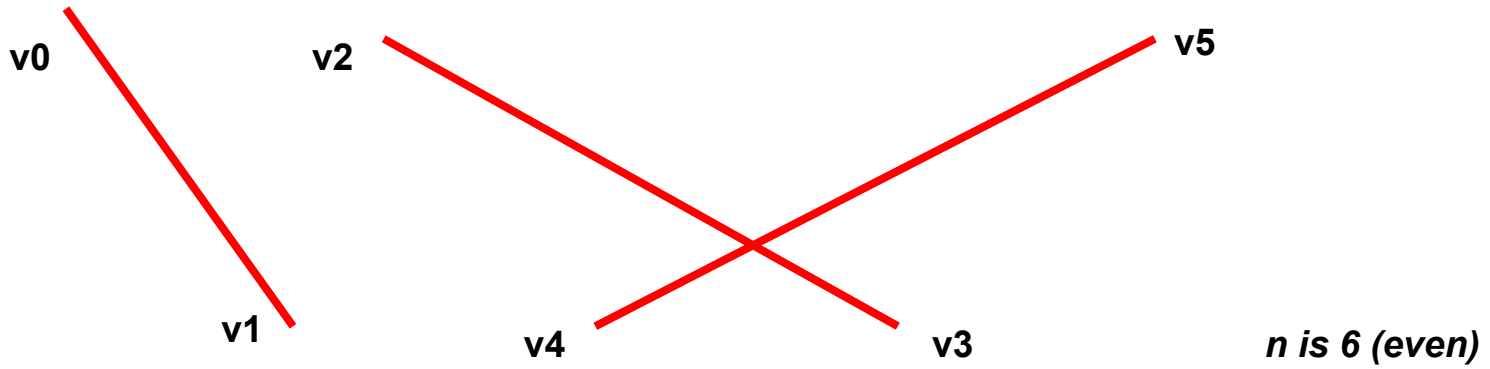
- nastaví šířku zobrazovaných bodů v pixelech, šířka musí být větší než 0, implicitní hodnota je 1.0
- Je-li však nastaveno glEnable(GL_PROGRAM_POINT_SIZE), příkaz se ignoruje a velikost každého bodu nastavuje vertex shader v proměnné gl_pointSize
- Dříve možnost zapnout aliasing glEnable(GL_POINT_SMOOTH)
- Nyní všechny body jako čtverce s texturou – point sprite



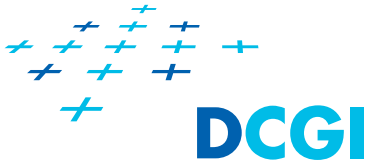
Samostatné úsečky

GL_LINES

- Sekvence samostatných úseček
- Úsečka je mezi v0 a v1, mezi v2 a v3, mezi v4 a v5 atd.
- Je-li počet vrcholů *n lichý*, poslední vrchol je ignorován.

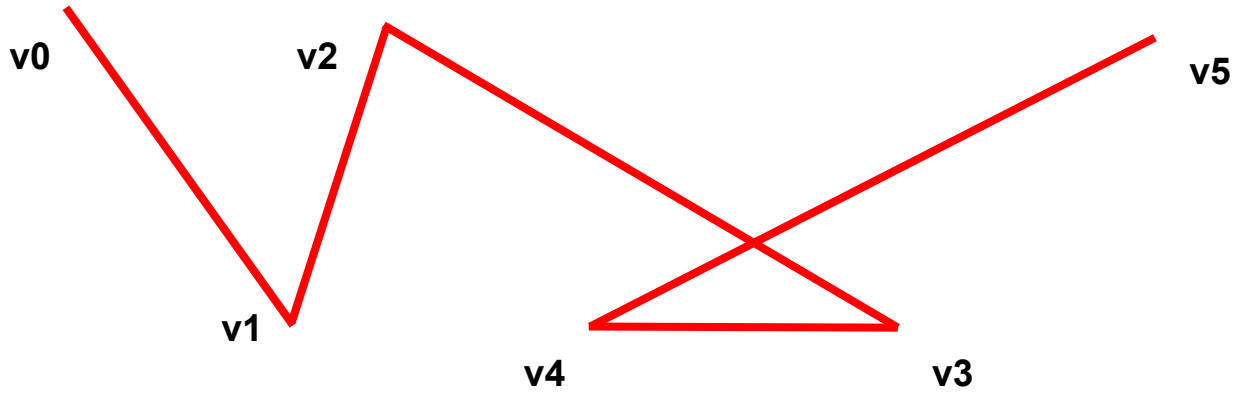


Lomená čára



GL_LINE_STRIP

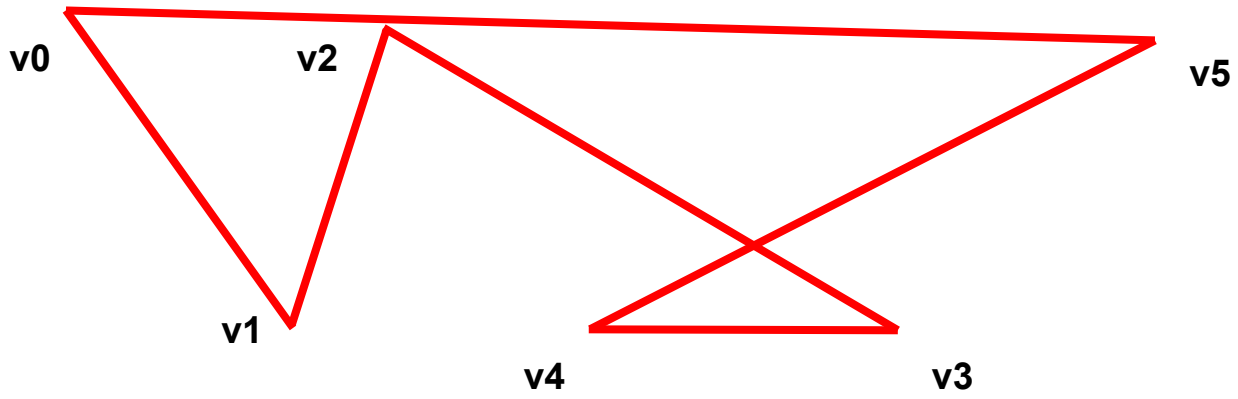
- Úsečka z v0 do v1, potom z v1 do v2, atd., konečně pak z vn-2 to vn-1
- Celkem *n-1* line úseček, pokud n=1, nekresli nic
- Nejsou stanovena žádná omezení na hodnoty vrcholů,
  úsečky se mohou protínat a body opakovat



Uzavřená posloupnost úseček

GL_LINE_LOOP

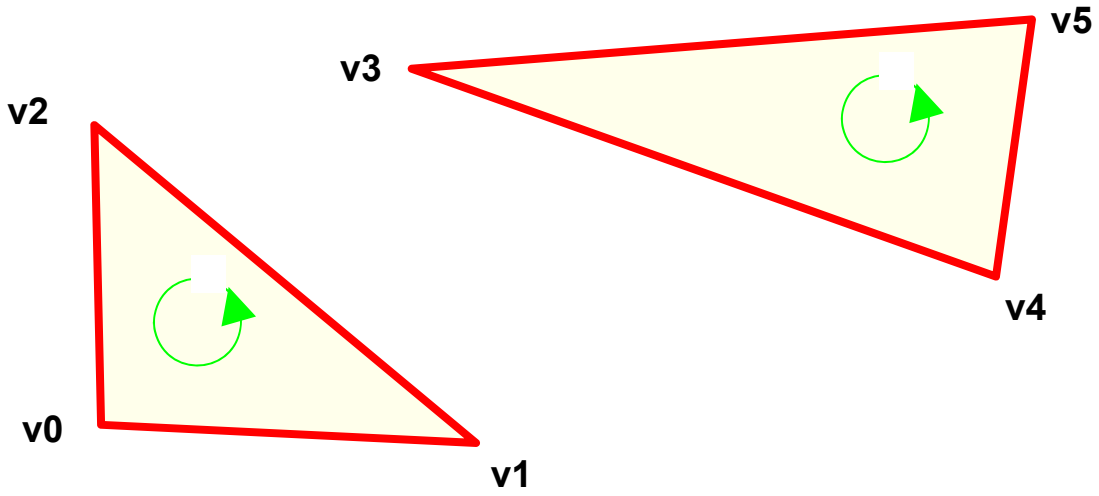
- Ozavřená lomená čára
- tj., jako GL_LINE_STRIP, plus poslední úsečka z vn-1 do v0 uzavírá smyčku



Trojúhelník

GL_TRIANGLES

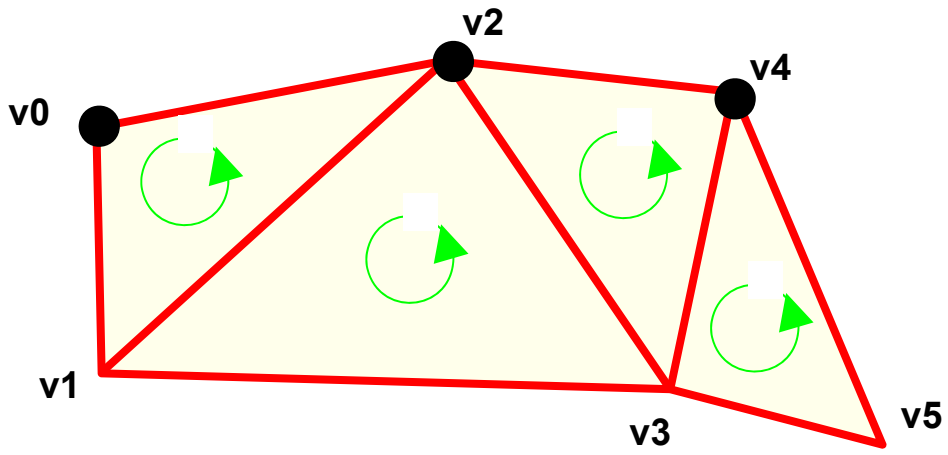
- zobrazí sekvenci trojúhelníků pomocí vrcholů v0, v1, v2, pak trojúhelník v3, v4, v5, atd.
- Pokud *n* není násobkem tří, je poslední vrchol či oba poslední dva vrcholy ignorovány



TRIANGLE STRIP

GL_TRIANGLE_STRIP

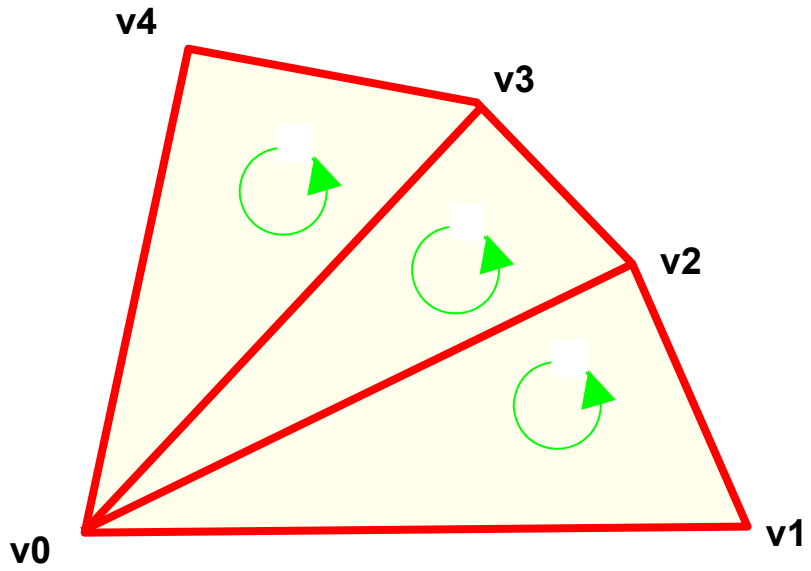
- Zobrazí sekvenci trojúhelníků o vrcholech trojúhelník v0, v1, v2, pak trojúhelník v2, v1, v3 (pozor na pořadí), pak v2, v3, v4, atd.
- Uspořádání vrcholů zajistí správnou orientaci každého trojúhelníku, takže je správně vykreslena k pozorovateli přivrácená část povrchů objektů
- *n* musí být větší nebo rovno třem
- Dochází k úspoře počtu vrcholů – kolik vrcholů je potřeba na jeden trojúhelník, pokud *n* by bylo velké číslo?



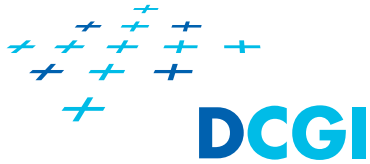
TRIANGLE FAN

GL_TRIANGLE_FAN

- Podobné jako je GL_TRIANGLE_STRIP, ale pořadí vrcholů pro první trojúhelník je v0, v1, v2, pak trojúhelník v0, v2, v3, pak trojúhelník v0, v3, v4, atd.

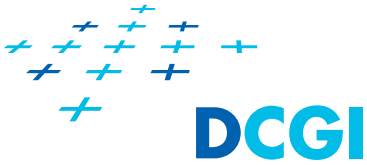


PATCH



- Jediné vstupní primitivum pro teselační shadery
- N-tice vrcholů – nebo řídících bodů nějaké plošky
- GPU pak generuje jemnější reprezentaci např. podle vzdálenosti plošky od pozorovatele (LOD)

Předání dat do shaderu pomocí proměnných s kvalifikátorem uniform



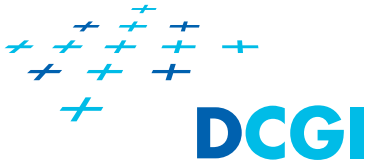
- **uniform** kvalifikátor pro proměnné
  - Vhodný pro hodnoty, které se mění málo (např. jsou stejné pro celý objekt)
  - Hodnoty se nastavují v OpenGL API, v C++
  - Vždy jsou vždy použity jako vstupní proměnné
  - V kódu shaderu je nelze měnit (read-only)

Deklarace proměnné s kvalifikátorem uniform a jménem RotationMatrix (*matice k transformaci vrcholů*)

Příklad zdrojového kódu pro Vertex shader

```
#version 400

in vec3 VertexPosition;
in vec3 VertexColor;
out vec3 Color;
uniform mat4 RotationMatrix;

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

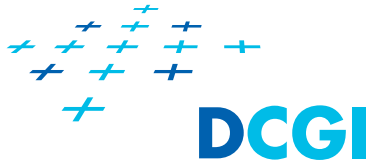
Předání dat do shaderu pomocí proměnných s kvalifikátorem uniform (pokr.)



Příklad části programu v C++

```
// clear the color buffer
glClear(GL_COLOR_BUFFER_BIT);

// use glm (part of PGR-framework) function to create a transformation matrix
mat4 rotationMatrix = glm::rotate(mat4(1.0f), angle, vec3(0.0f,0.0f,1.0f));

// find location of a uniform variable
GLuint location = glGetUniformLocation(programHandle, "RotationMatrix");

if (location >= 0) { // location == -1 if uniform is not active

  // assign a value to the uniform variable
  glUniformMatrix4fv(location, 1, GL_FALSE, &rotationMatrix[0][0]);
}

// bind to the vertex array object and call glDrawArrays to initiate rendering
glBindVertexArray(vaoHandle);
glDrawArrays(GL_TRIANGLES, 0, 3 );
```

Varianty příkazu glUniform



- glUniform{1|2|3|4}{f|i|ui}
  - 1 pro float, int, unsigned int, bool;
  - 2 pro vec2, ivec2, uvec2, bvec2, atd.
  - f pro float (float, vec2, vec3, vec4, nebo jejich pole)
  - i pro int (int, ivec2, ivec3, ivec4, nebo jejich pole)
  - ui pro unsigned int, uvec2, uvec3, uvec4, nebo jejich pole)
  - i, ui or f pro bool (bool, bvec2, bvec3, bvec4, nebo jejich pole)
    false pro 0 a 0.0f a *true* pro nenulové hodnoty

Příklady příkazu glUniform

Jednotlivé **hodnoty** – 1 až 4x float

- glUniform1f(location, 7.5f);
- glUniform4f(location , 0.5f , 0.5f , 0.2f , 1.0f);

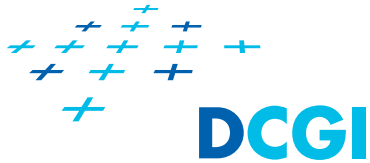
Pole hodnot – int pole[count] – pole o délce count

- glUniform1iv(location, count, pole);

Matice či pole matic

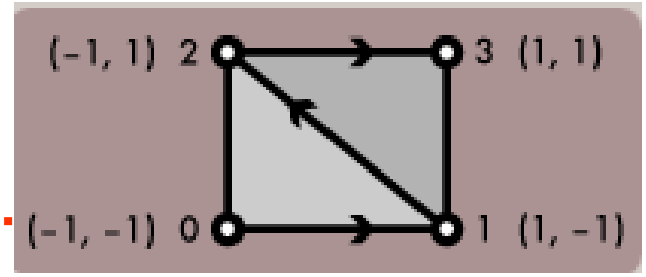
- glUniformMatrix4fv(location, 1, GL_FALSE, &rotationMatrix[0][0]);
  - 1 matice
  - Nebude se transponovat

Příklady různého uložení hodnot ve VBO



1. Jeden atribut v jednom VBO
2. Dva atributy v jednom VBO za sebou
3. Dva atributy v jednom VBO za prokládaně
4. Dva atributy, každý v jiném VBO

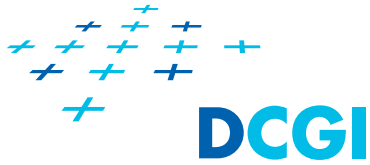
1. Př. – Jeden atribut v jednom VBO – příprava dat




```
static const Glfloat g_vertex_buffer_data[] = {
    -1.0f, -1.0f,
     1.0f, -1.0f,
    -1.0f,  1.0f,
     1.0f,  1.0f   };

static const GLushort g_element_buffer_data[] = { 0,1,2,3 };

vertex_buffer = make_buffer(   //souřadnice vrcholů
        GL_ARRAY_BUFFER,
        sizeof(g_vertex_buffer_data,
        g_vertex_buffer_data)
    );
element_buffer = make_buffer(   // indexy vrcholů
        GL_ELEMENT_ARRAY_BUFFER,
        sizeof(g_element_buffer_data,
        g_element_buffer_data)
```

[Groff]

1. Příklad – rozvinutý make_buffer



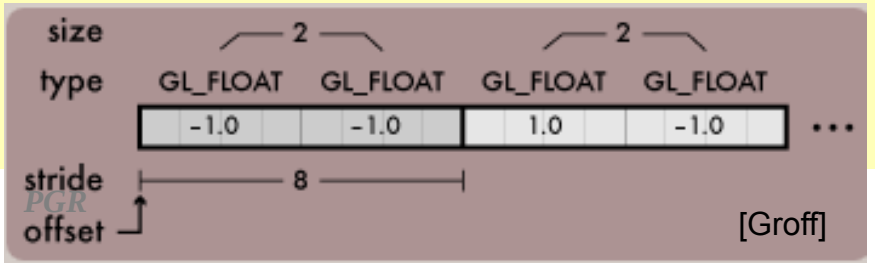

```
vertex_buffer = make_buffer(
        GL_ARRAY_BUFFER,
        sizeof(g_vertex_buffer_data) ,
        g_vertex_buffer_data
    );
```

Provede tedy tyto akce (bez pomocné proměnné):

```
glGenBuffers(1, &vertex_buffer );                  // 1
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer );     // 2a
glBufferData(GL_ARRAY_BUFFER,                      // 3
             sizeof(g_vertex_buffer_data),
             g_vertex_buffer_data,
             GL_STATIC_DRAW );
glBindBuffer(GL_ARRAY_BUFFER, 0);                  // 2b
```

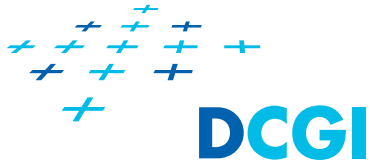
1. Příklad – navázání před vykreslením

```
pos_location  = glGetAttribLocation(program, "position");

glGenVertexArrays( 1, &vertex_array ); );     // 4
glBindVertexArray( vertex_array ); );         // 5
  glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);     // 2
  glEnableVertexAttribArray(pos_location);          // 6
  glVertexAttribPointer(                            // 7
    pos_location,                 /* VS attribute index */
    2,                            /* size */
    GL_FLOAT,                     /* type */
    GL_FALSE,                     /* normalized */
    sizeof(GLfloat)*2,            /* stride */
    (void*)0                      /* array buffer offset */
  );
glBindVertexArray( 0 );//4b
```



[Groff]

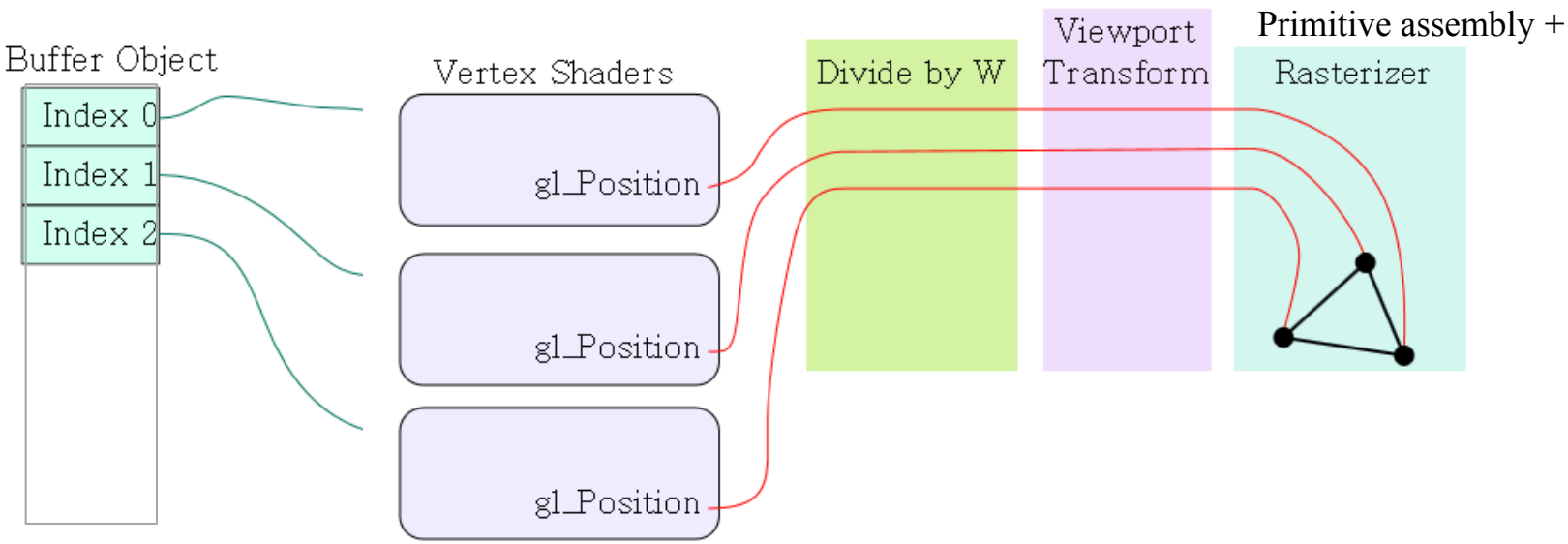
1. Příklad - vykreslení



[Groff]

```
glBindVertexArray( vertex_array );

  glDrawElements(
      GL_TRIANGLE_STRIP,   /* mode   */
      4,                   /* count */
      GL_UNSIGNED_SHORT,   /* type   */
      (void*)0             /* element array buffer offset */
  );

glBindVertexArray(0);
```

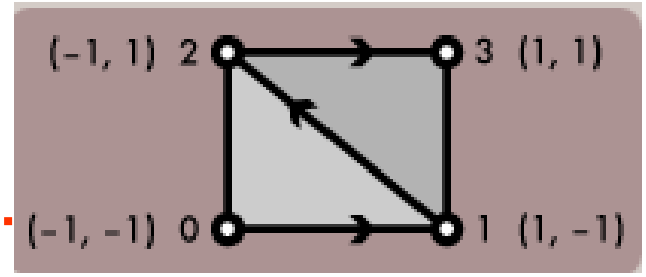
1. Příklad – Vizualizace zpracování geometrických dat na GPU



Přenos dat
z buffer object
do vertex shader

Přenos dat
z vertex shader
do jednotky rasterizace

2. Příklad – přidání druhého atributu v jednom VBO




```
static const GlfloatG_vertex_buffer_data[] = {
 -1.0f, -1.0f, 0.0f, 1.0f,
  1.0f, -1.0f, 0.0f, 1.0f,
 -1.0f,  1.0f, 0.0f, 1.0f,
  1.0f,  1.0f  0.0f, 1.0f,
  0.0f,  0.5f, 0.0f, 1.0f,
  1.0f,  0.0f, 0.0f, 1.0f,
  0.0f,  1.0f, 0.0f, 1.0f,
  0.0f,  0.0f, 1.0f, 1.0f, };
```

vrcholy

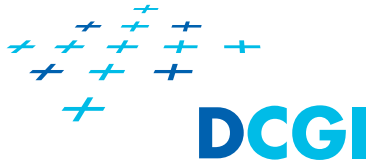
barvy

[Groff]

- Doplníme proměnnou ve vertex shaderu
- A její navázání

```
attributes.position
  = glGetAttribLocation(program, "position"); // VS input
attributes.color
  = glGetAttribLocation(program, „color");    // VS input
```

2. Příklad - navázání dvou atributů za sebou

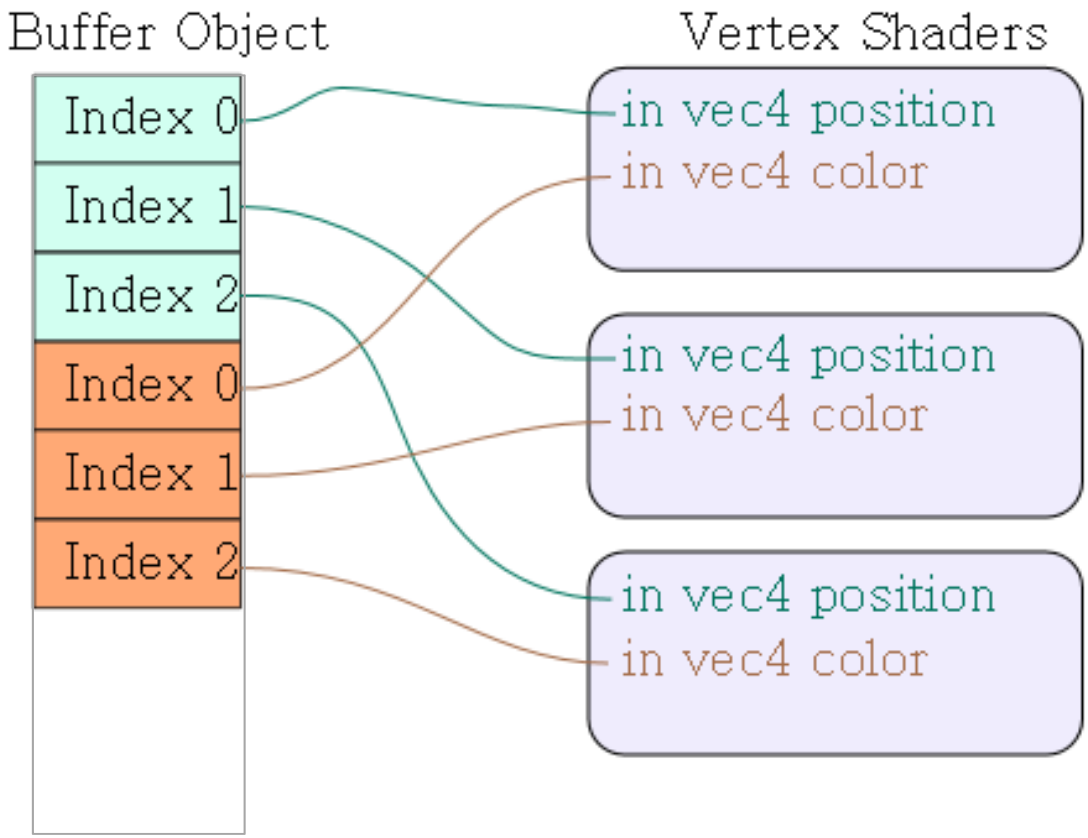



```
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glVertexAttribPointer(
  attributes.position,                  /* attribute */
  4,                                    /* size */
  GL_FLOAT,                             /* type */
  GL_FALSE,                             /* normalized */
  sizeof(GLfloat)*4,                    /* stride */
  (void*)0                              /* array buffer offset */
);
glVertexAttribPointer(
  attributes.color,                     /* attribute */
  4,                                    /* size */
  GL_FLOAT,                             /* type */
  GL_FALSE,                             /* normalized */
  sizeof(GLfloat)*4,                    /* stride */
  (void*) sizeof(GLfloat)*4*4           /* array buffer offset */
);
```

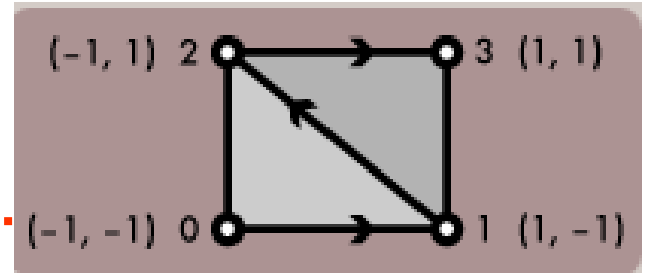
[Groff]

4 vrcholy

2. Předání dat vrcholu do „vertex shaderu“



3. Př. Dva atributy v jednom VBO prokládaně (*interlaced*)




```
static const GlfloatG_vertex_buffer_data[] = {
 -1.0f, -1.0f, 0.0f, 1.0f,    0.0f,  0.5f, 0.0f, 1.0f,
  1.0f, -1.0f, 0.0f, 1.0f,    1.0f,  0.0f, 0.0f, 1.0f,
 -1.0f,  1.0f, 0.0f, 1.0f,    0.0f,  1.0f, 0.0f, 1.0f,
  1.0f,  1.0f  0.0f, 1.0f,    0.0f,  0.0f, 1.0f, 1.0f, };
```

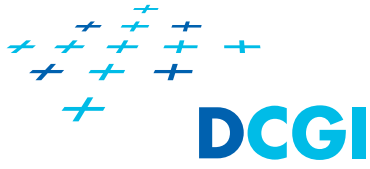
vrcholy

barvy

- Doplníme proměnnou ve vertex shaderu – to je stejné
- A její navázání

```
attributes.position
  = glGetAttribLocation(program, "position"); // VS input
attributes.color
  = glGetAttribLocation(program, "color");    // VS input
```

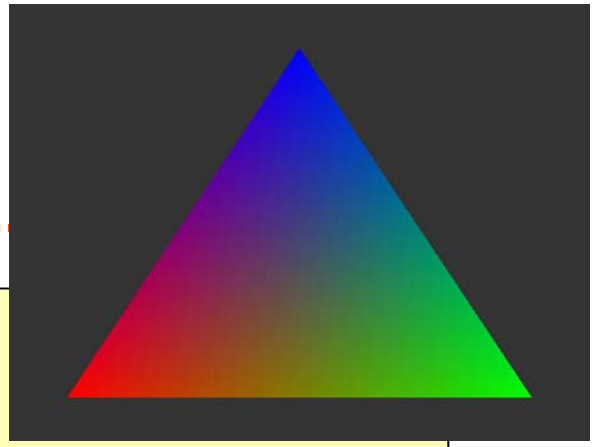
3. Příklad - použití i druhého atributu




```
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glVertexAttribPointer(
  attributes.position,              /* attribute */
  4,                                /* size */
  GL_FLOAT,                         /* type */
  GL_FALSE,                         /* normalized */
  sizeof(GLfloat)*8,                /* stride */
  (void*)0                          /* array buffer offset */
);
glVertexAttribPointer(
  attributes.color,                 /* attribute */
  4,                                /* size */
  GL_FLOAT,                         /* type */
  GL_FALSE,                         /* normalized */
  sizeof(GLfloat)*8,                /* stride */
  (void*) sizeof(GLfloat)*4         /* array buffer offset */
);
```

[Groff]

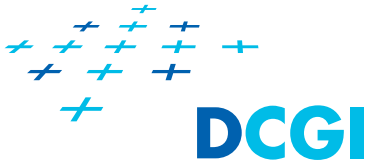
4. Př. dva atributy ve dvou VBO



Příklad části programu v C++

```
// 1. create and bind vertex array object
// variable to hold our handle to the vertex array object
GLuint vaoHandle;
// create and bind to a vertex array object (stores the relationship
between the buffers and the input attributes)
glGenVertexArrays( 1, &vaoHandle );
glBindVertexArray(vaoHandle);

// 2. set the data as arrays
float positionData[] = {                    // coordinates of triangle vertices
            -0.8f, -0.8f, 0.0f,
            0.8f, -0.8f, 0.0f,
            0.0f, 0.8f, 0.0f };
float colorData[] = {                       // colors of triangle vertices
            1.0f, 0.0f, 0.0f,
            0.0f, 1.0f, 0.0f,
            0.0f, 0.0f, 1.0f };
```

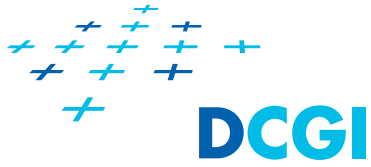
4. Př. dva atributy ve dvou VBO



Příklad části programu v C++

```
// 3. Create and bind the buffer objects, populate them with data
GLuint vboHandles[2];
glGenBuffers(2, vboHandles);
GLuint positionBufferHandle = vboHandles[0]; // 0. buffer name
GLuint colorBufferHandle    = vboHandles[1]; // 1. buffer name
// Populate the position buffer
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glBufferData(GL_ARRAY_BUFFER,
             9*sizeof(float), positionData, GL_STATIC_DRAW);

// Populate the color buffer
glBindBuffer(GL_ARRAY_BUFFER, colorBufferHandle);
glBufferData(GL_ARRAY_BUFFER,
             9*sizeof(float), colorData, GL_STATIC_DRAW);
```

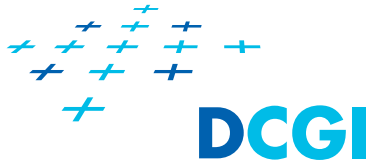
4. Př. dva atributy ve dvou VBO



Příklad části programu v C++

```
// 4. Make connection between data on a CPU and GPU
// obtain location of each attribute
GLint posLoc = glGetAttribLocation(programHandle, “VertexPosition”);
// This is the name in shader
GLint colorLoc = glGetAttribLocation(programHandle, “VertexColor”);
// This is the name in shader

// enable the vertex attribute arrays
glEnableVertexAttribArray(posLoc);        // vertex position
glEnableVertexAttribArray(colorLoc);      // vertex color

// map attribute with index posLoc to the position buffer
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glVertexAttribPointer(posLoc, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte *)NULL);

// map attribute with index colorLoc to the color buffer
glBindBuffer(GL_ARRAY_BUFFER, colorBufferHandle);
glVertexAttribPointer(colorLoc, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte *)NULL);
```

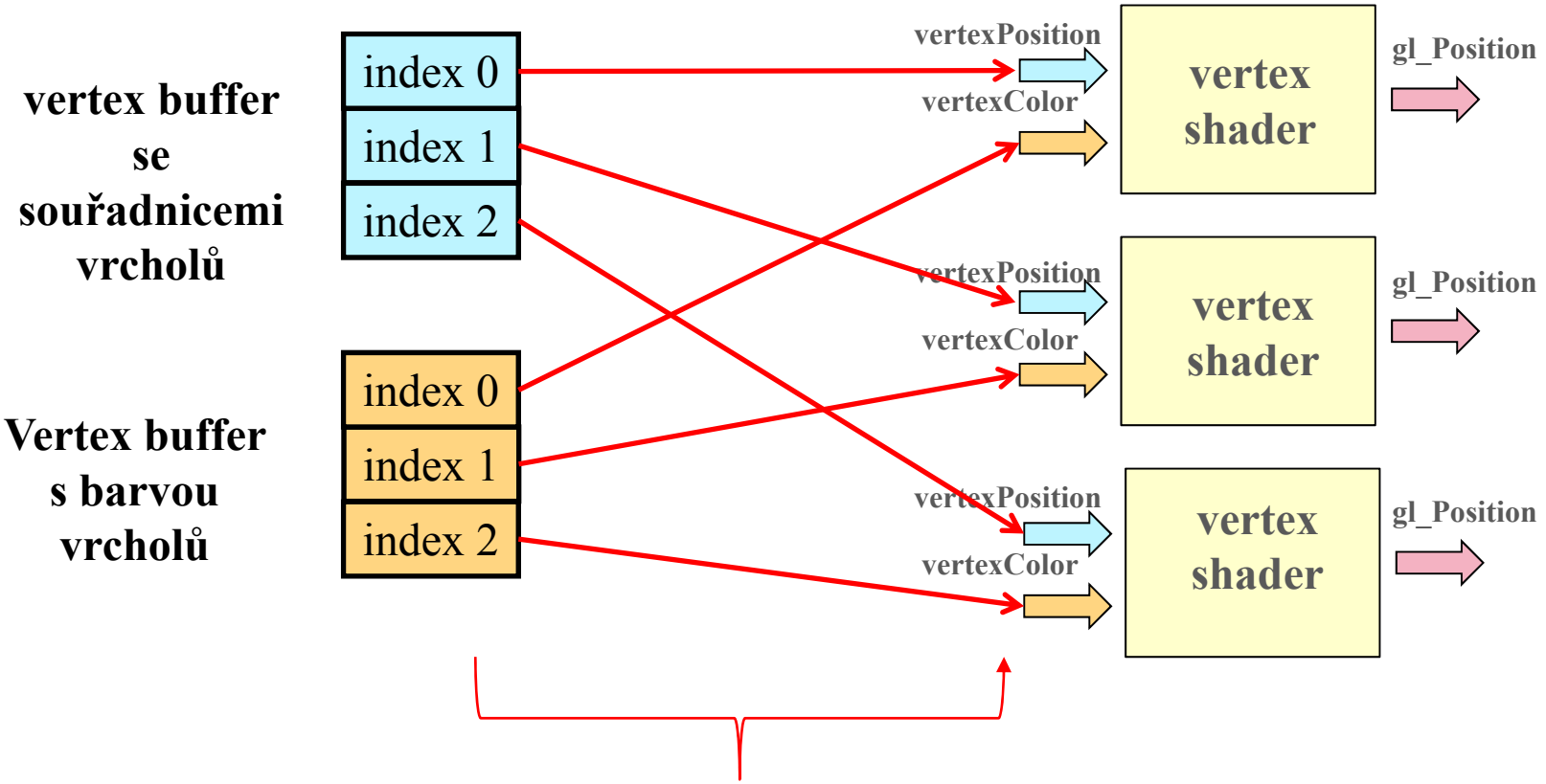
4. Př. dva atributy ve dvou VBO



Příklad části programu v C++

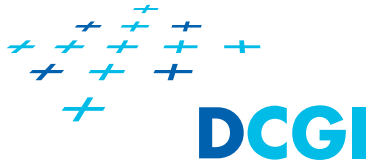
```
// 5. Use the data in the application to render them
// in the render function, bind to the vertex array object and call glDrawArrays to initiate rendering
glBindVertexArray(vaoHandle);
glDrawArrays(GL_TRIANGLES, 0, 3 );
```

4. Př. dva atributy ve dvou VBO



Přenos dat z buffer objektů v C++ do vertex shaderů na GPU

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