ChatGPT

Psaní a propojování vertex a fragment shaderů v OpenGL

Úvod do shaderů v OpenGL

Shadery jsou malé programy běžící na grafické kartě, které umožňují upravit chování grafického řetězce (pipeline) a dosáhnout různých vizuálních efektů. V moderním OpenGL se využívají zejména dva typy shaderů: **vrcholový (vertex) shader** a **fragmentový (pixel) shader**. Každý má odlišnou roli:

- **Vertex shader** zpracovává **jednotlivé vrcholy** geometrie. Na vstupu dostává atributy vrcholu (např. pozici, barvu, normálu). Jeho úkolem je provádět geometrické transformace vrcholu (pomocí modelové, pohledové a projekční matice), transformovat a normalizovat normály, případně vypočítat osvětlení na vrcholu a další operace. Každý vrchol je zpracován nezávisle (vertex shader nezná sousední vrcholy). Povinným výstupem vertex shaderu je vestavěná proměnná `gl_Position`, která určuje ztransformovanou pozici vrcholu v prostoru clip-space.

- **Fragment shader** zpracovává **jednotlivé fragmenty (pixely)** vykreslovaného obrazce. Jeho úkolem je vypočítat výslednou barvu fragmentu – typicky kombinuje barvu materiálu a osvětlení, případně aplikuje texturu, mlhu apod.. Vstupem fragment shaderu jsou hodnoty interpolované z výstupů vertex shaderu (například interpolovaná barva, normála nebo texturovací souřadnice). Podobně jako u vertex shaderu, každý fragment se zpracovává odděleně a shader nemá informace o ostatních pixelech. Výstupem fragment shaderu je barva pixelu; ve starších verzích GLSL se zapisuje do vestavěné proměnné `gl_FragColor`, v novějších verzích se používá uživatelsky definovaná výstupní proměnná (např. `out vec4 outColor`).

Poznámka: Existují i další typy shaderů (např. geometry, tessellation shadery), ale pro základní kreslení postačuje vertex a fragment shader.

Vytvoření a propojení shaderů v OpenGL

Abychom mohli shader využít, musíme jej v aplikaci (typicky v C/C++ kódu) vytvořit, zkompilovat a linkovat do shaderového programu. Následuje přehled kroků integrace shaderů do OpenGL aplikace:

1. **Vytvoření objektů shaderů:** Pomocí funkce `glCreateShader` vytvoříme prázdný shader objekt pro každý typ shaderu. Například `glCreateShader(GL_VERTEX_SHADER)` vytvoří vertex shader a `glCreateShader(GL_FRAGMENT_SHADER)` vytvoří fragment shader. Tyto objekty budou držet zdrojový kód a zkompilovaný binární kód shaderu.
2. **Přiřazení zdrojového kódu:** Funkce `glShaderSource(shader, count, &source, NULL)` nahraje zdrojový kód GLSL (jako řetězec) do daného shaderového objektu. Typicky načteme kód shaderu ze souboru nebo pole řetězců a předáme jej shaderu touto funkcí (parametr `count` udává počet řetězců).

3. **Kompilace shaderů:** Zavoláme `glCompileShader(shader)` pro každý shader (vertex i fragment). Tím se zdrojový kód přeloží (kompiluje) na GPU. Doporučuje se následně zkontrolovat, zda kompilace proběhla úspěšně (např. pomocí `glGetShaderiv` a `glGetShaderInfoLog` pro získání chybových hlášek).
4. **Vytvoření programu:** Pomocí `glCreateProgram()` vytvoříme prázdný **shaderový program**. Shaderový program slouží jako kontejner, do kterého připojíme zkompilované shadery.
5. **Připojení shaderů k programu:** Pomocí `glAttachShader(program, shader)` připojíme zkompilovaný vertex a fragment shader k programu. Tím se tyto dva shadery stanou součástí jednoho programu. (Lze připojit i více shaderů různých typů – minimálně jeden vertex a jeden fragment shader; geometry shader případně navíc apod.)
6. **Linkování programu:** Zavoláme `glLinkProgram(program)`. Linker v tu chvíli prováže výstupy a vstupy jednotlivých shaderů mezi sebou (např. proměnné z vertex shaderu s odpovídajícími vstupy fragment shaderu) a vytvoří finální binární program, který lze spustit na GPU. Opět je vhodné zkontrolovat úspěch linkování (`glGetProgramiv` / `glGetProgramInfoLog`).
7. **Použití shaderového programu:** Nakonec aktivujeme program voláním `glUseProgram(program)`. Tím se začne tento shaderový program používat pro veškeré následující vykreslování. (Pokud potřebujeme přepnout zpět na fixní funkce či jiný program, volá se `glUseProgram(0)` nebo jiný program.)

   Tip: Existuje mnoho knihoven a tříd, které celý tento proces správy shaderů usnadňují – například zabalí vytvoření, kompilaci a linkování shaderů do několika funkcí či metod. Příkladem může být Qt s třídou `QOpenGLShaderProgram`, GLU/GLUT utility, nebo moderní grafické engine. Ruční postup výše však pomáhá pochopit, co se v OpenGL děje.

Struktura vertex a fragment shaderu (GLSL)

OpenGL využívá jako jazyk shaderů GLSL (*OpenGL Shading Language*), což je jazyk podobný jazyku C. Shader kód se typicky píše do textových souborů s příponou `.vert` (vertex shader) a `.frag` (fragment shader). Základní struktura kódu shaderu obsahuje:

- **Direktivu verze GLSL:** na začátku souboru se udává např. `#version 330 core` (pro OpenGL 3.3), případně `#version 130` (pro OpenGL 3.0) apod., podle požadované verze GLSL. Tato direktiva musí být na úplném prvním řádku souboru.
- **Deklarace vstupů a výstupů:** pomocí *kvalifikátorů* (viz GLSL tahák níže) definujeme proměnné, které přicházejí do shaderu nebo odcházejí z něj do další fáze. Ve vertex shaderu tedy typicky deklarujeme vstupní atributy (pozice, normály, barvy, …), ve fragment shaderu zase vstupy, které jsou výstupem z vertex shaderu (tzv. interpolované *varying* proměnné), a výstupní barvu pixelu.
- **Funkci** `main()`**:** stejně jako v jazyce C/C++ je vstupním bodem shader programu funkce `void main()`. Uvnitř této funkce zapíšeme postup výpočtu pro každý vrchol či fragment.

Jako ilustraci uvádíme jednoduchý příklad shaderů, který očekává na vstupu 3D souřadnice vrcholu a jeho barvu a posílá interpolovanou barvu do fragment shaderu:

Vertex shader (GLSL 1.30+):

```glsl
#version 130
in vec3 a_Vertex;
in vec3 a_Color;
out vec4 color;

void main() {
    gl_Position = vec4(a_Vertex, 1.0);
    color = vec4(a_Color, 1.0);
}
```

(Ukázkový vertex shader: vezme vstupní souřadnice vrcholu `a_Vertex` *a nastaví výstupní pozici* `gl_Position`*. Zároveň vezme vstupní barvu* `a_Color` *a předá ji jako výstupní proměnnou* `color` *do fragment shaderu* [1] [2] *.)*

Fragment shader (GLSL 1.30+):

```glsl
#version 130
in vec4 color;
out vec4 outColor;

void main() {
    outColor = color;
}
```

(Ukázkový fragment shader: přijímá interpolovanou barvu `color` *z vertex shaderu a ukládá ji do výstupní proměnné* `outColor`*, čímž určuje barvu vykresleného pixelu.)*

Ve výše uvedené dvojici shaderů se používají moderní kvalifikátory `in` a `out` (zavedené od GLSL 1.30). Pro úplnost dodejme, že ve starších verzích GLSL (1.20 a nižší) se místo nich používaly kvalifikátory `attribute` (pro vstupy do vertex shaderu) a `varying` (pro proměnné předávané z vertex shaderu do fragment shaderu) [3] . Fragment shader ve starším GLSL zapisoval barvu do vestavěné proměnné `gl_FragColor` namísto používání vlastní `out` proměnné. Moderní přístup s `in/out` však poskytuje více flexibility (můžeme mít např. více výstupů z fragment shaderu při využití vícero renderovacích cílů).

Předávání atributů a uniform proměnných ze C++ do shaderů

Aby shader dělal něco užitečného, je potřeba mu **předat data** z aplikace – například pole vrcholů, barvy, transformační matice, parametry materiálu apod. V GLSL k tomu slouží dva hlavní mechanismy: **atributy** (per-vertex data) a **uniformy** (jednotné proměnné).

Atributy (atributní proměnné, `in` **ve vertex shaderu):** představují data, která se mohou lišit pro každý vrchol. Typicky sem patří souřadnice vrcholu, jeho normála, barva vrcholu, texturovací souřadnice atd. V moderním OpenGL se atributy předávají prostřednictvím **vertex bufferů** (VBO). Postup je následující:

- **Navázání dat na atributy:** Každá atributní proměnná ve shaderu má svůj název (např. `a_Vertex`). V kódu C++ musíme zajistit, že této proměnné přiřadíme zdroj dat. To lze udělat buď:
- **(a)** Před kompilací/linkováním programu přiřadit atributu **index** pomocí `glBindAttribLocation(program, index, "nazev")`. Tím určíte, že např. `a_Vertex` bude očekávat data na indexu 0, `a_Color` na indexu 1 atd. **Nebo:**
- **(b)** Použít v kódu shaderu *layout qualifier*, např. `layout(location = 0) in vec3 a_Vertex;`, což staticky nastaví index přímo ve shaderu (vyžaduje GLSL 3.30+). Tím pádem volání `glBindAttribLocation` není třeba – indexy jsou dány v kódu shaderu.
- **Povoleni a nastavení pole vrcholů:** Pro přiřazený index atributu povolíme čtení dat voláním `glEnableVertexAttribArray(index)`. Poté musíme specifikovat, odkud se data vezmou: k tomu slouží funkce `glVertexAttribPointer(index, size, type, normalized, stride, pointer)` [4]. Tato funkce nastaví, že na daném indexu se mají číst například trojice hodnot typu float ze současně navázaného bufferu:
- `index` – index atributu (např. 0 pro `a_Vertex`),
- `size` – počet komponent (např. 3 pro 3D souřadnice),
- `type` – datový typ v bufferu (např. `GL_FLOAT`),
- `normalized` – zda se mají celočíselné hodnoty normalizovat (pro float dáváme `GL_FALSE`),
- `stride` – krok (v bajtech) mezi dvěma po sobě jdoucími položkami v bufferu (0 znamená, že data jsou uložena těsně za sebou),
- `pointer` – offset nebo ukazatel na začátek dat v bufferu (např. `(void*)0` pro start bufferu). [5]

Typicky se před voláním `glVertexAttribPointer` **naváže příslušný VBO** s vrcholovými daty pomocí `glBindBuffer(GL_ARRAY_BUFFER, vboId)`. Pokud jsou data v klientské paměti (tzv. immediate mode, již se nepoužívá v moderním OpenGL), `glBindBuffer` se nevolá a `pointer` by ukazoval přímo na pole v RAM. V moderním přístupu však data vždy posíláme do grafické paměti do bufferů.

- **Příklad:** Pokud máme VBO s polem souřadnic vrcholů `vertices` (pole float xyzxyz…), nastavíme např.:

```
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
glEnableVertexAttribArray(0);
```

a obdobně pro barvy:

```
glBindBuffer(GL_ARRAY_BUFFER, colorBuffer);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
glEnableVertexAttribArray(1);
```

Tím jsme řekli, že `a_Vertex` (location 0) čte vždy 3 floaty z bufferu `vertexBuffer` a `a_Color` (location 1) čte 3 floaty z bufferu `colorBuffer`. Po vykreslení můžeme atributy případně

deaktivovat voláním `glDisableVertexAttribArray(0/1)` [6] (není nutné, pokud je budeme používat opakovaně, ale je to dobrá praxe uvolnit stav).

V moderním OpenGL (Core profile) výše popsaný postup nahrazuje dřívější funkce pevné funkce jako `glVertexPointer`, `glColorPointer`, `glEnableClientState(GL_VERTEX_ARRAY)` apod., které se používaly v Fixed Pipeline [7] [8]. Nyní máme obecný mechanismus *vertex atributů*, který funguje pro libovolná uživatelská data.

Uniformy (jednotné proměnné): jsou globální proměnné v shaderu, které mají pro všechny vrcholy či fragmenty *stejnou* hodnotu (typicky po celou dobu vykreslování jednoho objektu). Uniform může představovat například transformační matici, světelné parametry, čas, index aktuální snímku, barvu materiálu atd. Použití uniform proměnných probíhá takto:

- Ve shaderu (GLSL) se uniform deklaruje pomocí klíčového slova `uniform`. Například: `uniform mat4 u_ModelViewProjMatrix;` v vertex shaderu může představovat model-view-projection matici pro transformaci souřadnic vrcholu.
- V C++ po linkování shaderového programu zjistíme **location** (umístění) uniformy pomocí `glGetUniformLocation(program, "u_ModelViewProjMatrix")`. Tato funkce vrátí index (číslo), pod kterým je uniform v programu uložen.
- Následně **nastavíme hodnotu uniformy** voláním odpovídající funkce `glUniform*`. Pro matice je to `glUniformMatrix4fv(location, count, transpose, data)`. Například pro poslání jedné 4x4 matice:

```
GLuint loc = glGetUniformLocation(prog, "u_ModelViewProjMatrix");
glUseProgram(prog);
glUniformMatrix4fv(loc, 1, GL_FALSE, glm::value_ptr(matrix));
```

  Funkce `glUniformMatrix4fv` zde na danou location pošle 1 matici 4x4 (netransponovanou) z pole hodnot matice. Pro skalární uniformy existují funkce `glUniform1f` (float), `glUniform1i` (integer) atd., pro vektory `glUniform3fv` (poslat 3 floaty) apod. Důležité je, že před voláním `glUniform*` musíme mít aktivní správný shader program (`glUseProgram`), do kterého chceme uniformu nahrát.
- Uniformy lze nastavovat kdykoliv po linkování programu a jeho aktivaci. Jejich hodnota zůstává v programu uložena, dokud ji nezměníme nebo znovu nelinkujeme program. Typicky se uniformy nastavují před vykreslením objektu (např. nastavíme matici transformace ještě před voláním `glDrawArrays`).

Shrnutí: **atributy** slouží k posílání **pole dat** (odlišných pro každý vrchol), zatímco **uniformy** slouží k posílání **jednotných dat** (stejná hodnota pro všechny vrcholy/pixely v daném draw call). Správné využití obou typů je klíčové pro řízení chování shaderů z aplikační úrovně.

Příklad: průhlednost při rotaci a ztmavení při translaci

Nyní si ukážeme aplikaci výše uvedených principů na konkrétním problému. Zadání říká: implementujte ve shaderech efekt, aby objekt při **rotaci** zesvětlal (stal se průhledným) a při **translaci** ztmavl. To znamená,

že pokud objekt otáčíme, jeho viditelnost (alfa) by se měla snížit, a pokud ho posouváme po scéně, měla by se snižovat jeho barevná intenzita (ztmavne).

K realizaci takového efektu potřebujeme dvou věcí: **detekovat míru rotace a translace** objektu a **předat tuto informaci shaderům**, aby podle ní upravily barvu/transparentnost.

Detekce rotace a translace: Uvažujme, že pro každý objekt máme jeho transformační matici modelu (model matrix), která kombinuje rotace, posunutí atd. Z této matice můžeme získat jednotlivé složky transformace. Například pomocí knihovny **GLM** (OpenGL Mathematics) lze snadno extrahovat posunutí a rotaci:

- **Posunutí (translaci)** získáme z 4×4 modelové matice jako **čtvrtý sloupec** (pokud používáme konvenci posledního sloupce jako translace) – v GLM jednoduše: `glm::vec3 translation = glm::vec3(modelMatrix[3]);` získá vektor posunutí [9]. Velikost posunu od počátku spočteme jako délku tohoto vektoru: `float distance = glm::length(translation);`.
- **Rotaci** můžeme získat například pomocí **kvaternionu**: GLM nabízí `glm::quat rotationQuat = glm::quat_cast(modelMatrix);`, který extrahuje rotační složku matice jako kvaternion [10]. Z kvaternionu lze získat úhel rotace okolo určité osy (funkcí `glm::angle(rotationQuat)` – výsledný úhel v radiánech, případně převedeme na stupně). Pokud víme, že rotujeme kolem jedné osy, můžeme jednoduše sledovat úhel `angle` (např. v radianech či stupních) přímo jako parametr.
- **Alternativní přístup:** Pokud v aplikaci provádíme rotaci a translaci odděleně (např. ovládáním uživatele), můžeme si **přímo udržovat proměnné** jako aktuální úhel rotace a posunutí na ose X/Y/Z. Tyto hodnoty pak můžeme použít k určení efektu (např. při stisknutí klávesy pro rotaci zvyšovat úhel, při pohybu ukládat ujetou vzdálenost).

Jakmile máme nějakou míru rotace a translace, rozhodneme se, **jak ovlivní průhlednost a tmavost**. Řešení může být různých podob podle kreativního záměru; my zvolíme jednoduchý přístup:

- **Průhlednost při rotaci:** Čím větší rotace (resp. rychlost otáčení), tím více objekt zprůhledníme. Můžeme například stanovit, že při otočení o 180° bude objekt zcela průhledný. Podle aktuálního úhlu rotace spočteme **faktor průhlednosti** v intervalu 0–1. Například: `transparencyFactor = angle / 180°` (při 0° rotace faktor 0 – žádná průhlednost; při 180° faktor 1 – plná průhlednost). Úhel nad 180° můžeme saturovat na 1 (objekt už nemůže být „více než úplně průhledný"). Případně lze použít i jinou funkci (např. sinus pro plynulé opakované pulsování průhlednosti s rotací).
- **Ztmavení při translaci:** Čím dále objekt posuneme od výchozí polohy, tím tmavší bude. Spočteme například vzdálenost objektu od počáteční pozice (např. délku posunu od originu) a stanovíme **faktor ztmavení** 0–1. Např. pro vzdálenost 0 bude `darkeningFactor = 0` (žádné ztmavení, původní barvy), pro nějakou maximální uvažovanou vzdálenost (např. 10 jednotek) dáme `darkeningFactor = 1` (objekt úplně žčerná). Opět hodnotu případně ořízneme do intervalu [0,1]. Můžeme volit i lineární závislost nebo jinou křivku podle požadovaného efektu.

Tyto vypočtené faktory (říkejme jim třeba `u_transparency` a `u_darkening`) pak **pošleme jako uniformy** do shaderu. Budeme je interpretovat tak, že `u_transparency = 0` znamená žádnou průhlednost (objekt plně viditelný) a `u_transparency = 1` znamená objekt plně průhledný; obdobně `u_darkening = 0` žádné ztmavení (původní barva), `u_darkening = 1` úplně tmavý (černý) objekt.

Nyní upravíme fragment shader tak, aby tyto uniformy využil a modifikoval výslednou barvu pixelu:

```
// Fragment shader – aplikuje průhlednost a ztmavení
uniform float u_transparency;  // hodnota 0..1
uniform float u_darkening;     // hodnota 0..1

in vec4 color;       // interpolovaná vstupní barva z vertex shaderu
out vec4 outColor;   // výstupní barva pixelu

void main() {
    // Začneme s původní barvou
    vec4 finalColor = color;
    // Aplikujeme průhlednost: zvýšíme průhlednost podle u_transparency
    finalColor.a *= (1.0 - u_transparency);
    // Aplikujeme ztmavení: snížíme všechny složky RGB podle u_darkening
    finalColor.rgb *= (1.0 - u_darkening);
    // Výslednou barvu po úpravách zapíšeme na výstup
    outColor = finalColor;
}
```

Tento shader vezme vstupní barvu (`color`) a upraví její alfa a RGB složky. Pro **průhlednost** platí, že pokud `u_transparency = 0.0`, pak `(1.0 - u_transparency) = 1.0` a alfa zůstane nezměněna (objekt je plně neprůhledný). Pokud `u_transparency = 1.0`, pak `(1.0 - u_transparency) = 0.0` a výsledná alfa bude 0 (objekt zcela průhledný). Hodnoty mezi tím způsobí částečnou průhlednost úměrně nastavenému faktoru. Pro **ztmavení** obdobně: `u_darkening = 0` vede na zachování původní barvy, `u_darkening = 1` vynuluje složky RGB (černá), mezihodnoty ztmaví barvu lineárně.

Aby se průhlednost projevila vizuálně, nezapomeňte v OpenGL zapnout **míchání barev** (blending): typicky `glEnable(GL_BLEND); glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`. Fragment shader pak produkuje barvu s upravenou alfou a díky zapnutému blendingu se pixel smíchá s pozadím podle alfa hodnoty.

Předávání uniform hodnot: Zbývá vysvětlit, kdy a jak nastavovat uniformy `u_transparency` a `u_darkening` z aplikace. To může záviset na interakci uživatele nebo animaci: - Pokud např. uživatel právě rotuje objektem (držme třeba úhel rotace v proměnné `currentAngle`), spočítáme `transp = currentAngle / 180.0` (pokud to má smysl) a před vykreslením nastavíme `glUniform1f(locationTransp, transp)`. Pokud objekt stojí (nerotuje), můžeme `u_transparency` nechat 0. - Pokud objekt přesouváme (mění se jeho pozice), spočteme vzdálenost od původní polohy nebo rychlost pohybu a podle toho nastavíme `u_darkening`. Např. `float dark = glm::clamp(distance / 10.0f, 0.0f, 1.0f); glUniform1f(locationDark, dark);`. Když objekt zase zastavíme nebo vrátíme na původní místo, nastavíme `u_darkening` zpět k 0.

Tímto způsobem aplikační logika rozhoduje o míře efektu a **shader pouze aplikuje** zadané parametry na výslednou barvu. Kombinací uniformních proměnných a shaderové logiky tak lze docílit mnoha stavových efektů (průhlednost, pulsování barvy, změna odstínu při akcích atd.).

GLSL tahák – syntaxe, typy a funkce

Závěrem uvádíme stručný *tahák* se základní syntaxí a užitečnými konstrukcemi jazyka GLSL:

- **Datové typy:** GLSL disponuje skalárními typy jako `int`, `float`, `bool` (a jejich unsigned variantami `uint`), dále **vektory** (např. `vec2`, `vec3`, `vec4` pro vektory floatů, obdobně `ivec2/3/4`, `uvec2/3/4`, `bvec2/3/4` pro vektory integerů, unsigned a booleanů). Nechybí **matice** (`mat2`, `mat3`, `mat4` = 2x2, 3x3, 4x4; nebo obecné obdélníkové matice `mat2x3`, `mat3x4` atd.). Pro práci s texturami existují typy vzorkovačů (**sampler**): např. `sampler2D` pro 2D texturu, `samplerCube` pro cubemap texturu apod..
  – *Příklady:* `float x = 1.0; vec3 pos = vec3(0.0, 5.0, -3.0); mat4 m = mat4(1.0); sampler2D tex;`

- **Konstrukce a kombinování vektorů:** Vektorové typy mají v GLSL pohodlné konstruktory. Lze je skládat z jednotlivých hodnot i menších vektorů. Např. máme-li `vec3 rgbColor;`, můžeme vytvořit `vec4 RGBA` tak, že přidáme alfa složku: `vec4 rgba = vec4(rgbColor, 1.0);` [11]. Naopak z většího vektoru lze získat menší výběrem složek (*swizzling*, viz níže). Lze také konvertovat mezi typy pokud to dává smysl (int ↔ float, atd.) pomocí konstruktorů.

- **Přístup ke složkám vektorů (swizzling):** Ke komponentám vektorů se přistupuje pomocí tečkové notace. GLSL nabízí několik ekvivalentních názvů složek:

- **Pro souřadnice:** `.x, .y, .z, .w` (např. `vec4 v; float a = v.x;`),
- **Pro barvy:** `.r, .g, .b, .a` (např. `vec4 color; color.r = 1.0;` odpovídá nastavení x složky),

- **Pro texturové souřadnice:** `.s, .t, .p, .q` (méně časté, .s/.t se používá pro 2D textury místo .x/.y).
  Můžeme také vybírat více složek zároveň a tvořit tak podvektory: např. `vec3 rgb = rgba.rgb;` vezme první tři složky ze `vec4`, nebo `vec2 xz = pos.xz;` vytvoří 2D vektor z x a z složky 3D vektoru. Je možné měnit pořadí či opakovat složky: třeba `vec3 bgr = color.bgr;` prohodí pořadí na modrá-zelená-červená. **Není** však povoleno míchat různé sady označení najednou nebo uvést stejnou složku vícekrát v jedné čtyřici (např. `vec4 v = pos.xyar;` je nesmysl). Swizzling lze využít i pro přiřazení: např. `color.rgb = vec3(1.0, 0.0, 0.0);` změní první tři složky vektoru `color`.

- **Kvalifikátory proměnných:** GLSL používá klíčová slova k určení role proměnné:

- `in` – vstupní proměnná shaderu (přichází z předchozího stupně pipeline; v případě vertex shaderu typicky z aplikačního kódu jako atribut) [12].
- `out` – výstupní proměnná shaderu (posílá se do dalšího stupně; např. z vertex do fragment shaderu, nebo z fragment shaderu jako výstup barvy) [13].
- `uniform` – jednotná proměnná, nemění se napříč výpočtem všech vrcholů/fragmentů jednoho objektu (nastavuje se z aplikace) [14].
- `const` – konstantní proměnná (hodnota známá už při kompilaci shaderu, nelze ji uvnitř programu měnit) [15].

- *(žádný kvalifikátor)* – lokální proměnná v rámci funkce (existuje jen během provádění shaderu pro daný fragment/vrchol) [16].
- `attribute`, `varying` – zastaralé kvalifikátory z GLSL 1.20 a starších; `attribute` se používalo místo `in` pro vstup do vertex shaderu a `varying` místo kombinace `out` (ve vertex) / `in` (v fragmentu) [12].

- `centroid in`, `centroid out` – varianty vstupů/výstupů používané při *centroidní interpolaci* (speciální případ pro korektní interpolaci např. u anti-aliasingu na trojúhelnících; pokročilé použití nad rámec základu).

- **Kvalifikátory parametrů funkcí:** GLSL umožňuje definovat vlastní funkce a u jejich parametrů lze specifikovat:

- `in` (nebo nic) – parametr se předává *hodnotou do funkce* (only input) [17].
- `out` – parametr je využit pro *výstup z funkce* (funkce do něj zapíše výsledek, podobně jako návratová hodnota) [18].
- `inout` – parametr funguje jako vstup i výstup (do funkce jde s nějakou hodnotou a může být uvnitř změněn) [19].

- *(Pozn.:* Toto je podobné jako reference v C++. Vestavěná funkce `main()` žádné parametry nemá.)

- **Vestavěné funkce:** GLSL poskytuje bohatou sadu matematických funkcí pro usnadnění výpočtů přímo na GPU [20]:

- Převody úhlů: `radians(x)` (stupně → radiány), `degrees(x)` (radiány → stupně) [21].
- Goniometrické funkce: `sin(x)`, `cos(x)`, `tan(x)`, a také jejich inverzní varianty `asin`, `acos`, `atan` [22].
- Exponenciální a logaritmické: `pow(x,y)` (x^y), `exp(x)` (e^x), `log(x)` (přirozený logaritmus), `sqrt(x)` (odmocnina) [22].
- Zaokrouhlování a absolutní hodnota: `abs(x)` (absolutní hodnota), `floor(x)` (zaokrouhlení dolů), `ceil(x)` (zaokrouhlení nahoru) [23].
- Operace s celými čísly: `mod(x,y)` (zbytek po dělení x/y, funguje i pro float) [24].
- Porovnání a vzdálenosti: `min(a,b)`, `max(a,b)` (minimální, maximální hodnota), `clamp(x, minVal, maxVal)` (ořízne hodnotu do intervalu), `length(v)` (délka vektoru v), `distance(p,q)` (vzdálenost mezi body p a q) [25].
- Vektorové operace: `dot(u,v)` (skalární součin), `cross(u,v)` (vektorový produkt), `normalize(v)` (normalizace vektoru na jednotkovou délku) [25].
- Míchání a výběr: `mix(x,y,a)` (lineární interpolace mezi x a y, $x(1-a)+y \cdot a$), `step(edge,x)` (komponentově 0/1 podle prahu), `smoothstep(edge1,edge2,x)` (plynulý přechod 0→1).
- Texturovací funkce: `texture(sampler, coords)` vzorkuje texturu (2D, 3D, Cube) danou samplerem na zadaných souřadnicích [26] a vrací texel (barvu). Pro stínové mapy existuje `textureProj` atd.

- ...a mnoho dalších (bitové operace, matrix funkce, noise, atd.). Detailní výpis všech funkcí je v oficiální dokumentaci GLSL. Uvedené jsou jen nejčastěji používané.

- **Vestavěné proměnné:** Kromě uživatelských `in` / `out` proměnných má GLSL i některé *vestavěné*. Již jsme zmínili:

- `gl_Position` – (`vec4`) speciální výstup vertex shaderu, určuje pozici vrcholu v clip-space [27] (je nutné ji ve vertex shaderu nastavit, jinak se vrcholy považují za neplatné).
- `gl_FragCoord` – (`vec4`) ve fragment shaderu obsahuje souřadnice aktuálního pixelu na obrazovce (x, y) a hloubku z-bufferu (z). Hodí se pro efekty závislé na poloze fragmentu na obrazovce.
- `gl_FragColor` – (`vec4`) výstupní barva fragment shaderu ve starších verzích GLSL. V nových verzích se místo něj definují vlastní výstupní proměnné (jak jsme ukázali s `outColor`).
- `gl_PointSize` – (float) ve vertex shaderu nastavuje velikost vykreslovaného bodu (pro point primitives).
- A další specializované (např. `gl_InstanceID`, `gl_VertexID` pro čísla instancí a vrcholů, dostupné ve vertex shaderu; nebo `gl_FrontFacing` ve fragment shaderu indikující, zda fragment patří lícové straně trojúhelníka).

Tímto přehledem byste měli získat základní orientaci v psaní shaderů v OpenGL. Doporučujeme si vyzkoušet jednoduché příklady – např. obarvení trojúhelníku ve fragment shaderu, průchod interpolované barvy z vertex do fragment shaderu, použití uniformní transformační matice pro otáčení či posun objektu atd. Psaní shaderů vyžaduje určitý cvik, ale nabízí obrovskou flexibilitu v ovlivnění finálního vzhledu vykreslené scény. Hodně štěstí u zkoušky a nebojte se experimentovat!

---

1 2 3 4 5 6 7 8 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
3_opengl_shaders.pdf
file://file-435KcdajWGrT74g9TRReCq

9 10 matrix - glm - Decompose mat4 into translation and rotation? - Stack Overflow
https://stackoverflow.com/questions/17918033/glm-decompose-mat4-into-translation-and-rotation