ChatGPT

Reaktivní GLSL shadery v Qt/OpenGL

Moderní OpenGL umožňuje pomocí shaderů vytvářet působivé vizuální efekty, které mohou reagovat na různé vnější podněty. V prostředí Qt (např. s použitím `QOpenGLWidget` a `QOpenGLFunctions`) můžeme do GPU předávat data jako **uniformy** (globální vstupy pro shadery) či **atributy** (vstupy per-vertex) a dosáhnout tak interaktivních efektů. Níže uvádíme několik příkladů **GLSL vertex a fragment shaderů** zaměřených na vizuálně efektní jevy reagující na čas, polohu kamery či kurzoru myši. Každý příklad obsahuje důkladně okomentovaný kód (komentáře jsou psány česky, kód zůstává v angličtině), vysvětlení principu efektu a poznámky k implementaci v C++/Qt aplikaci (např. jak nastavit uniformní proměnné přes `QOpenGLShaderProgram`).

Poznámka: Předpokládáme znalost základů OpenGL v Qt – např. vytvoření třídy odvozené od `QOpenGLWidget`, implementace metod `initializeGL()` (pro načtení shaderů, vytvoření geometrie atd.) a `paintGL()` (pro vykreslení scény). Ve všech uvedených příkladech se využívá moderní shaderový přístup (core profile OpenGL), takže je potřeba mít připravený `QOpenGLShaderProgram` s načtenými vertex a fragment shadery, navázané atributy (pozice vrcholů, případně normály) a nastavené uniformy před vykreslením.

Shader s pulsující barvou podle času (`u_time`)

První ukázkou je velmi častý efekt: **pulsování barvy v čase**. Shader průběžně mění barvu objektu (např. mezi dvěma odstíny) na základě sinusové funkce času. K tomu využívá uniformní proměnnou `u_time`, do níž aplikace každým snímkem posílá aktuální čas (typicky v sekundách). Výsledkem je například objekt, který plynule mění svou barvu (např. pulzuje mezi červenou a modrou) s určitou periodou.

GLSL kód (pulsující barva)

```
#version 330 core

// **Vertex Shader: Pulsující barva podle času**
// Přijímá pozice vrcholů a transformuje je standardním způsobem pomocí matice MVP.
layout(location = 0) in vec3 a_position;   // pozice vrcholu (atribut, location 0)
uniform mat4 u_mvp;                        // Model-View-Projection matice pro transformaci
void main() {
    // Transformace vrcholové pozice do klip-prostoru
    gl_Position = u_mvp * vec4(a_position, 1.0);
    // (Tento shader nepotřebuje předávat žádné další údaje do fragment shaderu)
}
```

```
// **Fragment Shader: Pulsující barva podle času**
// Mění výslednou barvu fragmentu na základě času pomocí sinusovky.
uniform float u_time;  // aktuální čas v sekundách (zadávaný z aplikace)
out vec4 FragColor;    // výstupní barva fragmentu
void main() {
    // Výpočet interpolačního parametru t v rozsahu 0.0-1.0 na základě času.
    // Používáme sinus; expression (sin(u_time) + 1.0) / 2.0 bude oscilovat mezi
0 a 1.
    float t = (sin(u_time) + 1.0) / 2.0;
    // Definice dvou barev mezi kterými budeme přecházet (červená a modrá).
    vec3 colorA = vec3(1.0, 0.0, 0.0);  // červená
    vec3 colorB = vec3(0.0, 0.0, 1.0);  // modrá
    // Lineární interpolace mezi colorA a colorB řízená hodnotou t.
    vec3 pulsatingColor = mix(colorA, colorB, t);
    FragColor = vec4(pulsatingColor, 1.0);  // finální barva fragmentu s alfa =
1
}
```

Vysvětlení efektu: Ve vertex shaderu probíhá jen standardní transformace vrcholu pomocí matice model-view-projection (`u_mvp`). Celý efekt se odehrává ve fragment shaderu: uniformní proměnná `u_time` (čas v sekundách) je použita v sinusové funkci k výpočtu hodnoty mezi 0 a 1. Tato hodnota `t` plynule osciluje (sinus dává hladký průběh) a slouží k míchání dvou předem daných barev (`colorA` a `colorB`) pomocí vestavěné funkce `mix`. Když je sinus rovný -1, t vyjde 0 (barva bude `colorA`), při sinusu 1 vyjde t = 1 (barva `colorB`), pro mezilehlé hodnoty se barva postupně mění. Výsledkem je pulsující přechod mezi červenou a modrou s periodou 2π (což odpovídá ~6.28 sekundám; rychlost lze ovlivnit násobením argumentu u_time nějakým faktorem pro zrychlení/zpomalení).

Implementace v Qt/C++: Aplikace musí zajistit aktualizaci uniformy `u_time` v každém snímku. V praxi to znamená v metodě `paintGL()` před vykreslením získat aktuální čas (např. od začátku aplikace) a nastavit jej:

- Vytvořte si v `initializeGL()` instanci `QOpenGLShaderProgram`, přidejte zdroják vertex a fragment shaderu (např. voláním `addShaderFromSourceCode`), zlinkujte program. Získejte lokace uniformních proměnných (např. `program->uniformLocation("u_time")` a `program->uniformLocation("u_mvp")`) nebo ještě lépe, použijte pohodlné `program->setUniformValue()` s názvy proměnných.
- Vykreslování: v `paintGL()` při každém framu zavolejte `program->bind()` a nastavte `u_time`. Aktuální čas můžete získat například pomocí `QElapsedTimer` (který jste spustili v `initializeGL()`) nebo voláním `QTime::currentTime()` či `QDateTime::currentMSecsSinceEpoch()` – získaný čas převedete na sekundy s plovoucí čárkou. Poté volejte `program->setUniformValue("u_time", timeSeconds)`.
- Matice `u_mvp` bude nastavena podle kamery/scény: například pomocí `QMatrix4x4 model, view, proj;` se vypočítá kombinovaná projekční matice a předá shaderu (`program->setUniformValue("u_mvp", proj * view * model)`).
- Atribut pozice `a_position` je třeba svázat s daty modelu: buď přes `program->bindAttributeLocation("a_position", 0)` před linkováním a poté povolit a naplnit atribut

na location 0, nebo využít `program->enableAttributeArray` a `program->setAttributeBuffer` s názvem atributu. (V našem kódu jsme použili `layout(location=0)` pro zjednodušení, takže očekáváme, že pozice vrcholu je bindenutá na index 0.)
- Pro plynulé animování lze využít např. `QTimer` spouštějící `update()` ~60x za sekundu; v každém takovém cyklu se paintGL vykreslí s novou hodnotou času, tím dojde k animaci pulzující barvy.

Shader s vlněním pozice (deformace vrcholů sinusovkou)

Druhá ukázka demonstruje **deformaci objektu ve vertex shaderu** pomocí sinusové funkce, což může vytvořit efekt vlnění nebo kymácení. Vertex shader zde posune vrcholy objektu nahoru a dolů podle sinusovky, jejíž argument závisí na poloze vrcholu a čase. Tím docílíme například vlnící se plochy (jako vlna na vodě nebo vlající prapor) čistě pomocí GPU výpočtu, bez nutnosti měnit geometrie na CPU.

GLSL kód (vlnící se deformace)

```
#version 330 core

// **Vertex Shader: Vlnění pozice sinusovou funkcí**
layout(location = 0) in vec3 a_position;  // vstupní pozice vrcholu
uniform mat4 u_mvp;                       // matice model-view-projection
uniform float u_time;                    // čas (sekundy) pro animaci vlny
uniform float u_amplitude;               // amplituda vlnění (maximální výchylka)
uniform float u_frequency;               // frekvence vln (jak "husté" vlny jsou)
void main() {
    // Kopie vstupní pozice, abychom ji mohli upravit
    vec3 pos = a_position;
    // Aplikace sinusové deformace na Y souřadnici vrcholu:
    // - pos.x * u_frequency určuje vlnovou délku (menší u_frequency = delší
vlny)
    // - u_time uvnitř sin zajistí časovou animaci posunu vln (cestují v čase)
    // - vynásobení u_amplitude určuje výšku vlny
    pos.y += sin(pos.x * u_frequency + u_time) * u_amplitude;
    // Standardní transformace vrcholu do clip-space pomocí MVP matice
    gl_Position = u_mvp * vec4(pos, 1.0);
}

// **Fragment Shader: Vlnící se objekt (jednolitá barva)**
uniform vec3 u_color;    // uniformní barva objektu
out vec4 FragColor;
void main() {
    FragColor = vec4(u_color, 1.0);  // vykreslíme objekt jednolitou barvou
}
```

Vysvětlení efektu: Klíčová část se odehrává ve vertex shaderu, kde každému vrcholu měníme jeho výšku (`pos.y`) podle sinusové funkce. Sinus má dvě složky vstupu: - **Prostorová složka:** `pos.x * u_frequency` způsobí, že pozice podél osy X ovlivňuje fázi sinusovky. To znamená, že různé vrcholy (různá

X) budou v daném čase posunuty nahoru/dolů různě podle své X souřadnice – vzniká tak vlna rozprostřená v prostoru. Parametr `u_frequency` určuje **vlnovou délku**: vyšší frekvence znamená, že na stejné délce objektu bude více cyklů sinusovky (hustší, kratší vlny), nižší hodnota dává delší táhlé vlny. - **Časová složka:** přičtení `u_time` do argumentu sinus zajistí posun fáze v čase – vlna se animuje. V čase 0 a 2π budou deformace stejné. Pokud bychom chtěli ovlivnit rychlost pohybu vln, můžeme `u_time` násobit nějakým parametrem (např. rychlostí); v této jednoduché verzi se bere přímo čas v sekundách, což odpovídá periodě 2π sekund pro jednu kompletní oscilaci. - **Amplituda:** výsledek sinusovky (rozmezí -1 až 1) násobíme uniformou `u_amplitude`. Ta nastavuje maximální výchylku nahoru či dolů od původní polohy. Například při `u_amplitude = 0.5` se vrcholy budou pohybovat ±0.5 jednotky v ose Y oproti původnímu modelu.

Fragment shader zde jen vyplní každý fragment jednolitou barvou `u_color` (např. modrou) – díky tomu vynikne samotná změna tvaru objektu. Pokud by objekt měl vlastní texturu nebo barevný atribut, mohl by fragment shader být složitější; pro demonstrační účely stačí uniformní barva.

Implementace v Qt/C++: Aby byl efekt vlnění dobře patrný, je vhodné mít model s dostatečným počtem vrcholů podél směru vln (např. jemně tesslovaná rovina nebo dlouhý úzký pásek s více segmenty). V Qt zajistíme:

- **Nastavení atributů:** Stejně jako v předchozím příkladu musíme naplnit buffer s vrcholovými pozicemi (`a_position`) a povolit jej. Pro vlnění nepotřebujeme žádné speciální atributy navíc (postačí pozice vrcholů, případně normály pokud bychom chtěli provádět osvětlení, ale to zde neděláme).
- **Uniformy:** `u_time` se bude aktualizovat každý snímek (podobně jako v pulsující barvě). `u_amplitude` a `u_frequency` lze nastavit staticky nebo také měnit (třeba umožnit uživateli je regulovat sliderem v aplikaci). Například:
- `program->setUniformValue("u_amplitude", 0.2f);`
- `program->setUniformValue("u_frequency", 5.0f);` (tyto hodnoty by vytvořily relativně malé vlnky s pěti sinusovými periodami rozprostřenými podél jednotkové délky v ose X).
- **Animace:** Pro plynulé vlnění opět zajistíme volání `update()` ~60x za sekundu nebo podle potřeby. V každém framu aktualizujeme `u_time`. Pokud by animace neměla běžet neustále, lze `u_time` zastavit či resetovat podle potřeby.
- **Transformace:** `u_mvp` se nastaví obdobně jako dříve. Pokud se objekt či kamera hýbe, je třeba matici počítat každým snímkem. Pokud je scéna statická, stačí jednou.
- Při vykreslování voláme `program->bind()`, pak `program->setUniformValue()` pro čas (a případně i pro barvu, amplitudu, frekvenci pokud by se měnily), poté vykreslíme geometrii voláním např. `glDrawArrays` nebo `glDrawElements`.

(Tip: Chcete-li vidět výraznější efekt, vyzkoušejte nastavit různou amplitudu či frekvenci. Také je možné vlnit jinou osu než Y – např. posouvat vrcholy v osách X nebo Z pro různé efekty – stačí upravit výpočet pos.)

Shader s barevným zvýrazněním podle vzdálenosti od kamery (`u_camPos`)

Třetí shader ukazuje využití polohy kamery k **barevnému zvýraznění objektů podle jejich vzdálenosti**. Tento efekt se dá připodobnit například k **hloubkové mapě nebo mlze**, kde blízké objekty mají jinou barvu než vzdálené. Pomocí uniformy `u_camPos` (pozice kamery ve světových souřadnicích) můžeme v shaderu

spočítat vzdálenost každého fragmentu od kamery a podle této vzdálenosti měnit jeho barvu. Typicky můžeme definovat dvě barvy: jednu pro blízké povrchy a druhou pro daleké, a plynule mezi nimi přecházet. Alternativně lze zvýraznit jen určité vzdálenosti (např. objekty v určitém rozsahu obarvit speciální barvou), ale v našem příkladu zvolíme jednoduchý plynulý přechod.

GLSL kód (barva vs. vzdálenost od kamery)

```
#version 330 core

// **Vertex Shader: Předání světové pozice pro výpočet vzdálenosti**
layout(location = 0) in vec3 a_position;
uniform mat4 u_model;        // modelová matice (model -> svět)
uniform mat4 u_view;         // view matice (svět -> kamera)
uniform mat4 u_proj;         // projekční matice (kamera -> clip-space)
out vec3 v_worldPos;         // výstupní proměnná nesoucí světovou souřadnici
vrcholu
void main() {
    // Převod vrcholové pozice do světových souřadnic:
    vec4 worldPos4 = u_model * vec4(a_position, 1.0);
    v_worldPos = worldPos4.xyz;
    // Standardní transformace do clip-space pro zobrazení
    gl_Position = u_proj * u_view * worldPos4;
}

// **Fragment Shader: Barevné zvýraznění podle vzdálenosti od kamery**
uniform vec3 u_camPos;      // pozice kamery ve světě
uniform float u_nearDist;   // vzdálenost, od které začíná přechod barev
(blízko)
uniform float u_farDist;    // vzdálenost, na které končí přechod (daleko)
uniform vec3 u_nearColor;   // barva pro blízké objekty (u_nearDist a blíže)
uniform vec3 u_farColor;    // barva pro vzdálené objekty (u_farDist a dále)
in vec3 v_worldPos;         // světová pozice fragmentu (interpolovaná z vertex
shaderu)
out vec4 FragColor;
void main() {
    // Výpočet vzdálenosti tohoto fragmentu od kamery (Eukleidovská délka
vektoru)
    float dist = distance(v_worldPos, u_camPos);
    // Normalizace vzdálenosti do intervalu 0-1 vzhledem k zadanému rozsahu
[u_nearDist, u_farDist]
    float t = clamp((dist - u_nearDist) / (u_farDist - u_nearDist), 0.0, 1.0);
    // Lineární interpolace barvy mezi blízkou a vzdálenou barvou
    vec3 color = mix(u_nearColor, u_farColor, t);
    FragColor = vec4(color, 1.0);
}
```

Vysvětlení efektu: Vertex shader zde předává do fragment shaderu pozici každého vrcholu ve **světových souřadnicích** (`v_worldPos`). K tomu využívá uniformní matici modelu `u_model` (převádí z model-space do world-space). Tuto světovou pozici pak rasterizace náležitě perspektivně interpoluje pro každý pixel fragmentu. Ve fragment shaderu pak:

- Pomocí `u_camPos` (ve světových souřadnicích) spočítáme vzdálenost fragmentu od kamery: `distance(v_worldPos, u_camPos)`. To je standardní Eukleidovská vzdálenost mezi dvěma body v prostoru.
- Tuto vzdálenost upravíme do normovaného rozsahu 0 až 1 pomocí parametru t. Zde přicházejí ke slovu uniformy `u_nearDist` a `u_farDist`. Ty představují prahové vzdálenosti, mezi kterými chceme měnit barvu:
- Pokud `dist <= u_nearDist`, po clamping bude t = 0 (objekt je považován za "blízký" – použije se plně `u_nearColor`).
- Pokud `dist >= u_farDist`, pak t = 1 (objekt je "daleko" – dostane barvu `u_farColor`).
- Pro vzdálenosti mezi těmito hodnotami se t lineárně změní od 0 do 1 a barva bude plynulým mixem obou koncových barev.
- Výsledek `color` se získá interpolací (`mix`) mezi `u_nearColor` a `u_farColor` podle váhy t.
- Funkce `clamp` zajišťuje, že hodnoty t mimo [0,1] jsou oříznuty (tj. cokoliv blíž než nearDist dostane t=0, cokoliv dál než farDist t=1, aby se nepřekračovala daná paleta).

Tímto způsobem můžeme snadno například nastavit blízkou barvu `u_nearColor` na jasně zelenou a vzdálenou `u_farColor` na modrou – objekty u kamery by byly zelené a plynule by přecházely do modré s rostoucí vzdáleností. Lze tím simulovat i mlhavý efekt (pokud zvolíme vzdálenou barvu stejnou jako pozadí, objekty splývají s pozadím do ztracena).

Implementace v Qt/C++:

- **Výpočet vzdáleností:** Hodnoty `u_nearDist` a `u_farDist` volíme podle měřítka scény. Například v jednotkách světových souřadnic: pokud chceme, aby do 5 jednotek od kamery byla objektům přiřazena `u_nearColor` a ve 20 jednotkách a dál už `u_farColor`, nastavíme `u_nearDist = 5.0f; u_farDist = 20.0f`. Tyto uniformy lze nastavit staticky nebo je případně měnit (např. posuvníkem v UI) pro různé efekty.
- **Pozice kamery (`u_camPos`):** Musíme určit pozici kamery v souřadnicích světa a tu předat shaderu. Pokud v Qt používáte vlastní kameru (např. definovanou maticí pohledu), získáte pozici kamery jako **inverzi** view matice. Příklad: máte-li `QMatrix4x4 view` (kamera), pak světovou pozici kamery lze získat jako `QVector3D camPos = view.inverted().column(3).toVector3D();` (v případě, že view je rigidní transformace). Alternativně, pokud používáte QCamera nebo trackball, můžete mít pozici kamery přímo k dispozici. Každopádně tuto pozici (camPos.x, camPos.y, camPos.z) vložíte do uniformy `u_camPos` (např. `program->setUniformValue("u_camPos", camPos)`).
- **Matice transformace:** Všimněte si, že ve vertex shaderu nyní používáme samostatně model, view a projekční matici. V C++ kódu tedy musíme nastavit tři uniformy: `u_model`, `u_view`, `u_proj`. Např.:

```
program->setUniformValue("u_model", modelMatrix);
program->setUniformValue("u_view", viewMatrix);
program->setUniformValue("u_proj", projMatrix);
```

(kde `modelMatrix` je `QMatrix4x4` transformace objektu, `viewMatrix` je kamera a `projMatrix` projekce).

- **Geometrie a atributy:** Potřebujeme pouze atribut pozice (`a_position`); normály zde nevyužíváme, protože neděláme osvětlení. Při rasterizaci se interpoluje světová poloha `v_worldPos`, což funguje dostatečně pro efekt barvy – drobná poznámka: interpolace polohy není zcela lineární v perspektivní projekci (způsobí menší odchylky při výpočtu vzdálenosti uprostřed trojúhelníků oproti přesné světové vzdálenosti), ale pro vizuální efekt to nevadí. Pro zcela přesný výpočet by bylo možné provádět výpočet vzdálenosti již ve vertex shaderu a ten předat jako hladký interpolant, nebo rekonstruovat světovou pozici ve fragmentu z hloubky – to je však zbytečně složité pro tento účel.
- Po nastavení všech potřebných uniform (`u_camPos`, `u_nearDist`, `u_farDist`, `u_nearColor`, `u_farColor`, a matic) a po bindnutí bufferů s vrcholy zavoláme vykreslení. Výsledkem bude, že objekty na scéně budou obarveny podle své vzdálenosti od kamery.

(Možné rozšíření: tento shader lze kombinovat s klasickým osvětlovacím shaderem – stačí vypočítaný mix barev použít jako modulaci *materiálové barvy nebo intenzity. Také by šlo místo plynulého mixu použít prahové funkce (step) a zvýraznit tak třeba jen objekty v určité vzdálenostní vrstvě jinou barvou.)*

Shader s obvodovým zvýrazněním (silueta/outline efekt)

Nyní se zaměříme na **obvodové zvýraznění** objektu – tedy vykreslení barevné kontury kolem jeho siluety. Tento efekt je známý z comicsových či toon shaderů jako černá linka kolem modelu, případně jako záře kolem objektu (edge glow). Realizace v čistém shaderu typicky vyžaduje trošku trik: obrys není nic jiného než druhý vykreslovací průchod, v němž se objekt vykreslí zvětšený a jednobarevný, **za** tím původním. Náš shader tedy zajistí zvětšení objektu (všemi směry od povrchu) a vykreslí jej jednolitou barvou obrysu. K tomu potřebujeme znát normály povrchu, abychom mohli vrcholy **odsunout ve směru normály**.

GLSL kód (obvodový outline shader)

```
#version 330 core

// **Vertex Shader: Extruze vrcholů pro outline**
layout(location = 0) in vec3 a_position;
layout(location = 1) in vec3 a_normal;      // normála vrcholu (jednotkový
vektor)
uniform mat4 u_model;
uniform mat4 u_view;
uniform mat4 u_proj;
uniform mat3 u_normalMatrix;   // matice pro transformaci normál (inverzně
transponovaná modelová matice)
uniform float u_outlineThickness;  // tloušťka obrysu (velikost odsunutí)
void main() {
    // Převod vrcholové pozice a normály do světových souřadnic
    vec3 worldPos    = (u_model * vec4(a_position, 1.0)).xyz;
    vec3 worldNormal = normalize(u_normalMatrix * a_normal);
    // Posunutí (extrudování) vrcholu ve směru normály o požadovanou tloušťku
```

```
    vec3 extrudedPos = worldPos + worldNormal * u_outlineThickness;
    // Projekce posunuté pozice do clip-space pro vykreslení
    gl_Position = u_proj * u_view * vec4(extrudedPos, 1.0);
}

// **Fragment Shader: Jednobarevný obrys**
uniform vec3 u_outlineColor;   // barva obrysu (např. černá pro toon outline)
out vec4 FragColor;
void main() {
    FragColor = vec4(u_outlineColor, 1.0);
}
```

Vysvětlení efektu: Tento shader (vertex+fragment) je určen pro vykreslení obrysů. Samotný trik spočívá ve **vertex shaderu**, který posune vrcholy modelu směrem ven od povrchu:

- Vstupní normála `a_normal` (kterou každému vrcholu musí poskytnout aplikace, obvykle vypočtená z geometrie modelu) udává směr, kterým daný vrchol "trčí" z povrchu. Normály transformujeme do světového prostoru pomocí `u_normalMatrix` (což bývá inverzní transpozice modelové matice; zajistí správný převod normál i při nejednotném měřítku modelu). Pro jistotu normalizujeme (`normalize(...)`) výsledek, aby normála zůstala jednotkové délky.
- Pozici vrcholu převedeme do světových souřadnic (vynásobením `u_model` maticí). Následně k této pozici přičteme normálu vynásobenou skalárem `u_outlineThickness`. Tím vznikne nová světová pozice, která je o danou vzdálenost posunutá směrem ven z povrchu. Parametr `u_outlineThickness` tedy určuje tloušťku obrysu v jednotkách světových souřadnic. Typicky půjde o malou hodnotu (např. 0.05 či 0.1 vzhledem k velikosti objektu).
- Upravenou pozici pak transformujeme view a projekční maticí stejně jako normální vrcholy. Výsledkem je, že tento vykreslovací průchod zobrazí objekt o něco větší, **odsazený všemi směry od původního povrchu**.

Fragment shader pro outline je triviální – všem fragmentům obrysového objektu přiřadí jednotnou barvu `u_outlineColor` (často černá pro kreslený efekt, nebo libovolná zářivá barva pro "glow"). Neřešíme zde osvětlení ani textury; obrys bývá prostě plochý barevný.

Jak ale zajistit, aby se obrys objevil jen skutečně na okrajích modelu, a nepřekryl celý objekt? K tomu slouží správné nastavení vykreslování v C++: - **Dvojí vykreslení:** Obrysový shader se obvykle použije jako druhý pas vykreslování objektu. Nejdříve můžete vykreslit běžným shaderem samotný model, a poté obrys, nebo naopak. Častější je vykreslit obrys **pod** modelem, aby nebyl jím překryt. Toho docílíme pomocí nastavení *cullingu* (odstranění odvrácených stěn): - Zapněte zobrazení pouze zadních stěn pro obrys: `glEnable(GL_CULL_FACE); glCullFace(GL_FRONT);`. Tím se při vykreslování obrysového pasu **vykreslí jen back-faces** (zadní strany trojúhelníků). Protože obrysový model je roztažený ven, zadní strany budou viditelné okolo okrajů původního modelu jako obrys. (Přední stěny obrysového modelu bychom viděli přes celý objekt, to nechceme, proto je cullujeme.) - Po vykreslení obrysu přepněte culling zpět na výchozí `glCullFace(GL_BACK)` pro standardní vykreslování předních stran u ostatních objektů. - Alternativní technika je vykreslit nejprve obrys (back-faces roztažené) a pak normální model *nad ním*. V takovém případě obrys "vykukuje" jen tam, kde přesahuje původní model. Tato metoda také potřebuje culling front-face pro obrys a zachování hloubkového testu. - **Hloubkový test:** Chceme, aby obrys neprosvítal skrz objekt nebo jiné objekty blíže kameře. Proto ponecháme `glEnable(GL_DEPTH_TEST)`

celou dobu. Při vykreslení obrysu je dobré do hloubkového bufferu zapisovat, nebo použít lehký *offset* dozadu, aby nedocházelo k boji o hloubku na hraně (z-fighting) mezi obrysem a modelem. Můžeme použít `glEnable(GL_POLYGON_OFFSET_FILL)` s malým posunem, nebo jednoduše obrys posunout o něco málo více (zvětšit `u_outlineThickness` nepatrně), aby byl jistě za modelem. - **Nastavení uniform a atributů:** V Qt aplikaci musíme pro obrysový shader nastavit: - `u_outlineThickness` – např. konstantu 0.05. Můžeme ji také udělat závislou na měřítku modelu nebo vzdálenosti od kamery, pokud chceme, aby tloušťka na obrazovce byla konstantní, ale to vyžaduje pokročilejší výpočet. - `u_outlineColor` – typicky `QVector3D(0,0,0)` pro černou, nebo jakákoli jiná barva kontury. - `u_model`, `u_view`, `u_proj`, `u_normalMatrix` – modelová, pohledová a projekční matice stejně jako u běžného shaderu. Normálová matice je možno získat z modelové: např.

```
QMatrix4x4 modelMatrix = ...;
QMatrix3x3 normalMatrix = modelMatrix.normalMatrix();
programOutline->setUniformValue("u_normalMatrix", normalMatrix);
```

Qt umí přímo z QMatrix4x4 vypočítat 3x3 normálovou matici metodou `normalMatrix()`. Ta provede inverzní transpozici pro správný převod normál. - Atributy: Kromě `a_position` musíme obrysovému shaderu předat také `a_normal`. To znamená, že ve vertextovém bufferu modelu musíme mít i normálové vektory a povolit je (např. `programOutline->enableAttributeArray("a_normal")` a nastavit pointer do VBO s normálami). Bez správných normál nebude extruze fungovat očekávaně. - **Použití:** Jednoduchá sekvence vykreslení jednoho objektu s obrysem by mohla vypadat takto:

```
// 1. Normální model (bez obrysu)
glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);                // vykreslujeme jen přední strany
programMain->bind();
programMain->setUniformValue(...);  // nastav matice, materiál etc.
// ... nastav atributy pozice, normál, uv ...
glDrawElements(...);                // vykresli model

// 2. Outline obrys
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);               // vykresluj jen zadní strany (siluetu)
programOutline->bind();
programOutline->setUniformValue(...); // matice, barva, tloušťka
// ... povol atributy (pozice+normála) ...
glDrawElements(...);                // vykresli "nafouknutý" model
glCullFace(GL_BACK);                // (reset cull-face na default)
```

Výsledkem bude, že druhý průchod vykreslí roztažený model jen tam, kde vykukuje za hranami prvního průchodu – to vytvoří viditelný obrys. Při správném nastavení hloubky nebude obrys překreslovat přední stranu modelu, pouze pozadí okolo něj.

Tento outline shader dává charakteristický "comic" vzhled. Lze jej dále rozšířit například tak, že `u_outlineColor` nebude konstantní, ale třeba závislý na původní barvě objektu nebo dokonce textuře, čímž vznikne barevný lem. Také je možné implementovat obdobný efekt v post-processingu (detekce hran v obrazových datech), ale náš přístup s druhým vykreslením je názorný a efektivní pro jednotlivé modely.

Shader reagující na pozici kurzoru myši (`u_mouse`)

Poslední ukázka předvádí využití interakce s uživatelem – **reakci shaderu na pohyb myši**. Typickým příkladem může být "světlomet" či "zvýraznění" v místě, kam ukazuje kurzor. Shader může podle polohy kurzoru měnit barvu fragmentů, vytvořit kolem kurzoru světlejší oblast, vlny, nebo jiné efekty. My se zaměříme na jednoduchý efekt: **kruh zvýraznění kolem pozice kurzoru**. Budeme potřebovat uniformy pro pozici myši a pro rozměry okna (plátna), abychom mohli správně dopočítat relativní polohu fragmentu vůči kurzoru.

GLSL kód (interakce s kurzorem myši)

```
#version 330 core

// **Vertex Shader: Passthrough (bez speciální deformace)**
layout(location = 0) in vec3 a_position;
uniform mat4 u_mvp;
void main() {
    gl_Position = u_mvp * vec4(a_position, 1.0);
    // Vertex shader zde jen transformuje vrcholy; veškerá interakce se
vyhodnotí ve fragment shaderu
}

// **Fragment Shader: Zvýraznění podle vzdálenosti od kurzoru**
uniform vec2 u_mouse;      // pozice kurzoru myši (v pixelech, souřadnice v
okně)
uniform vec2 u_resolution; // rozlišení okna (šířka, výška v pixelech)
out vec4 FragColor;
void main() {
    // Normalizace souřadnic fragmentu (gl_FragCoord.xy) do rozsahu [0,1]
    vec2 normCoord = gl_FragCoord.xy / u_resolution;
    // Normalizace zadané polohy myši obdobně
    vec2 normMouse = u_mouse / u_resolution;
    // Vzdálenost normalizované pozice tohoto fragmentu od pozice kurzoru (v 0-1
prostoru)
    float dist = distance(normCoord, normMouse);
    // Parametry pro velikost zvýraznění
    float radius = 0.1;         // poloměr kruhového zvýraznění (v norm.
souřadnicích ~10% velikosti okna)
    float edge =
0.02;          // poloměr "okraje" pro plynulé vytracení (šířka přechodu)
    // Výpočet intenzity zvýraznění pomocí hladkého přechodu (smoothstep dává 0-
```

```
>1 v rozmezí, invertujeme 1.0-... pro inverzní efekt)
    float highlight = 1.0 - smoothstep(radius - edge, radius + edge, dist);
    // Definice barev: základní barva pozadí a barva zvýraznění
    vec3 baseColor = vec3(0.0, 0.0, 0.3);       // tmavě modrá (pozadí)
    vec3 highlightColor = vec3(1.0, 1.0, 1.0);  // bílá (zvýrazněný "světlý"
bod)
    // Smíchání výsledné barvy - v okolí kurzoru přimícháme bílou podle
intenzity highlight
    vec3 finalColor = mix(baseColor, highlightColor, highlight);
    FragColor = vec4(finalColor, 1.0);
}
```

Vysvětlení efektu: Tento shader předpokládá, že vykreslujeme nějakou plochu pokrývající oblast okna (např. fullscreen quad, nebo velký obdélník). Ve vertex shaderu opět jen převedeme souřadnice do obrazovky pomocí `u_mvp` (pro 2D efekt by MVP mohla být jednotková nebo ortografická projekce pokrývající okno).

Ve fragment shaderu pak využíváme vestavěnou proměnnou `gl_FragCoord`, která obsahuje souřadnice aktuálního fragmentu v okně (v pixelech). Uniformy `u_mouse` a `u_resolution` jsou také v jednotkách pixelů: - `u_mouse` by měl nést [x, y] pozici kurzoru v okně. **Pozor:** souřadnice `gl_FragCoord` mají počátek vlevo dole (0,0 je levý spodní roh okna), zatímco v Qt typicky dostaneme pozici myši s počátkem vlevo *hore*. Je proto nutné při nastavování `u_mouse` provést převod: `u_mouse = QVector2D(mouseX, height - mouseY)`, aby Y souřadnice seděla. - `u_resolution` nese šířku a výšku okna. Použijeme ji k normování souřadnic do intervalu [0,1]. `normCoord` pak vyjadřuje polohu fragmentu relativně v okně (0,0 v levém spodním rohu a 1,1 v pravém horním). - Vypočítáme vzdálenost `dist` mezi normalizovanou pozicí fragmentu a normalizovanou pozicí kurzoru. To nám dává, jak daleko je pixel od aktuálního kurzoru, v relativních jednotkách. Například fragment přímo pod kurzorem má dist ~0, fragment na opačné straně okna může mít dist ~1.4 (pokud vezmeme úhlopříčku jako maximum). - Proměnné `radius` a `edge` definují velikost zvýrazněné oblasti: do vzdálenosti `radius` od kurzoru bude fragment plně zvýrazněn, za hranicí `radius+edge` už nebude zvýrazněn vůbec. Funkcí `smoothstep(radius-edge, radius+edge, dist)` získáme plynulý přechod z 0 na 1 v okolí hrany; od 0 do (radius-edge) vrací 0, od (radius+edge) do nekonečna vrací 1, a mezi tím hladce roste. My však chceme intenzitu **1 uprostřed a 0 daleko**, proto bereme `1.0 - smoothstep(...)`. Výsledná hodnota `highlight` je tedy 1 pro fragmenty velmi blízko kurzoru a klesá k 0 za hranicí poloměru. - Máme zvoleny dvě barvy: `baseColor` (tmavá modrá) a `highlightColor` (bílá). Pomocí `mix` je smícháme podle hodnoty `highlight`. Tedy pixel daleko od kurzoru bude baseColor, pixel přímo na kurzoru bude téměř bílý, a mezi tím bude docházet k plynulému zesvětlování. - Finální barva se výstupem FragColor aplikuje na fragment.

Tento efekt vytvoří kolem kurzoru světlou "svatozář", která plynule mizí s vzdáleností. Pokud kurzor pohybujeme, světlá oblast se bude přesouvat spolu s ním.

Implementace v Qt/C++:

- **Nastavení uniformy myši:** V Qt můžeme zachytávat pohyb myši (např. překrytím `QOpenGLWidget::mouseMoveEvent`). Při pohybu myši aktualizujeme nějakou proměnnou s pozicí.

Před vykreslením (v `paintGL`) pak nastavíme uniformu `u_mouse`. Jak bylo zmíněno, je nutné převést souřadnice:

```
QPoint mousePos = mapFromGlobal(QCursor::pos()); // nebo z eventu v rámci widgetu
float mx = mousePos.x();
float my = mousePos.y();
float w = width();
float h = height();
QVector2D mouseUniform(mx, h - my);
program->setUniformValue("u_mouse", mouseUniform);
program->setUniformValue("u_resolution", QVector2D(w, h));
```

Tím dostane shader korektní souřadnice. Pokud byste chtěli myš nepřetržitě (i bez stisknutého tlačítka), nezapomeňte povolit ve widgetu `setMouseTracking(true)`.

- **Geometrie:** Můžeme vykreslovat například celý obraz jako jeden velký trojúhelník pokrývající obrazovku (tzv. screen-space triangle) nebo čtverec. V jednodušším podání lze jako geometrii použít čtverec 2x2 v clip space (vrcholy [-1,-1], [1,-1], [1,1], [-1,1]), pak by MVP mohla být identita. Pokud používáte `u_mvp`, můžete vykreslovat třeba fullscreen quad v modelových souřadnicích 0-1 a použít ortho projekci. Na způsobu nezáleží, hlavně aby fragmenty pokryly celé okno, jinak by se efekt projevil jen na objektu který vykreslujete.
- **Barvy a parametry:** V našem fragment shaderu jsou barvy natvrdo vepsané. Můžete je samozřejmě také dát jako uniformy (`u_baseColor`, `u_highlightColor`) pro větší flexibilitu. Stejně tak poloměr zvýraznění a šířku okraje by bylo vhodné mít jako uniformní floaty, abyste je mohli ladit z aplikace. Princip zůstává stejný.
- **Výkonnost:** Výpočet vzdálenosti a smoothstep pro každý pixel je triviální pro GPU. I tak se vyplatí zvážit, zda nepotřebujete optimalizovat – např. pokud byste chtěli ostré ohraničení místo plynulého, je možné to udělat pomocí jednoduchého podmíněného přiřazení (if/else nebo lépe step funkce). Náš plynulý přechod ale vypadá lépe, neboť nemá aliasing na okraji kruhu.

Tento "kurzorový" shader lze různě obměňovat: může třeba vykreslovat v kruhu texturu, ripple efekt (vlny od kurzoru při kliknutí), nebo ovlivňovat jiné vlastnosti materiálu (např. lom světla pod "lupou" kolem myši apod.). Základní princip je však vždy stejný – aplikace předá polohu myši do shaderu a ten podle ní moduluje výstup.

Závěrem: Uvedené příklady ilustrují, jak lze relativně snadno obohatit OpenGL aplikaci o interaktivní a vizuálně atraktivní efekty pomocí GLSL shaderů. V Qt prostředí můžeme tyto shadery pohodlně spravovat skrze `QOpenGLShaderProgram` a aktualizovat jejich uniformy podle vstupů od uživatele (čas, kamera, myš, atd.). Důkladné okomentování kódu by mělo pomoci pochopit, *jak* každý efekt funguje a co je potřeba na straně C++/Qt udělat pro jeho zprovoznění. Experimentováním s těmito shadery (např. změnou funkcí, přidáním dalších uniform pro ovládání efektu) se dá naučit mnoho o tom, jak GPU vykreslovací řetězec pracuje a jak dosáhnout požadovaných vizuálních jevů. Hodně štěstí při dalším zkoumání shaderů!
