Pokročilé techniky GLSL: Dynamická změna barvy a průhlednosti objektů pomocí rotace a translace

Tato zpráva se zabývá tvorbou vlastních vertex a fragment shaderů v GLSL (OpenGL Shading Language) se zaměřením na dynamické vizuální efekty. Konkrétně se bude věnovat tomu, jak měnit barvu a průhlednost objektu v závislosti na jeho rotaci, jak implementovat odlišnou změnu barvy na základě translace objektu a jak tyto dva efekty kombinovat. Cílem je poskytnout srozumitelný výklad a praktické příklady ve formě pseudokódu, které mohou sloužit jako základ pro implementaci v různých grafických aplikacích využívajících OpenGL.

Sekce 1: Úvod do GLSL shaderů pro dynamické vizuály

Moderní grafické programování se neobejde bez shaderů, malých specializovaných programů, které běží přímo na grafickém procesoru (GPU) a tvoří klíčovou součást programovatelného grafického pipeline. Umožňují vývojářům přímou kontrolu nad tím, jak je geometrie zpracovávána a jak jsou výsledné pixely obarveny, což otevírá dveře k široké škále vizuálních efektů.

Vertex Shader Primární zodpovědností vertex shaderu je zpracování každého jednotlivého vertexu (bodu) 3D modelu. Jeho základním výstupem je finální transformovaná pozice vertexu v tzv. clip space, reprezentovaná vestavěnou proměnnou gl_Position. Tato pozice určuje, kde se vertex objeví na obrazovce. Právě ve vertex shaderu se typicky aplikují geometrické transformace jako rotace a translace pomocí maticových operací, což je stěžejní pro požadavky této zprávy.

Fragment Shader (Pixel Shader) Fragment shader, někdy označovaný jako pixel shader, vstupuje do hry po rasterizaci, což je proces převodu geometrických primitiv na fragmenty (potenciální pixely na obrazovce). Úkolem fragment shaderu je určit finální barvu a hodnotu hloubky pro každý takový fragment. Jádrem výstupu fragment shaderu je vestavěná proměnná gl_FragColor (typ vec4 reprezentující RGBA – červenou, zelenou, modrou složku a alfu pro průhlednost). V tomto shaderu budou implementovány požadované dynamické změny barvy a průhlednosti na základě rotace a translace.

Shadery umožňující dynamické efekty založené na vlastnostech objektu
Programovatelnost shaderů posouvá možnosti grafiky daleko za statické renderování. Umožňuje vytvářet vizuály, které dynamicky reagují na měnící se vlastnosti objektů nebo parametry scény. Data jako aktuální úhel rotace objektu nebo jeho translační vektor mohou být předávána z hlavní aplikace (běžící na CPU) do shaderů (běžících na GPU). Shadery pak mohou tato data využít ve svých výpočtech k modifikaci vzhledu objektu v reálném čase. Tento princip je základem pro efekty, které si uživatel přeje vytvořit.

Dynamické vizuální efekty, jako je změna barvy či průhlednosti v závislosti na rotaci nebo translaci, nejsou výhradně záležitostí GPU a shaderů. Vyžadují úzkou spolupráci mezi CPU a GPU. Logika aplikace na straně CPU je zodpovědná za sledování nebo výpočet aktuálního stavu objektu (např. jeho úhlu rotace, translačního vektoru, často v reakci na uživatelský vstup, fyzikální simulaci nebo časovače animace). Tyto stavové informace musí být následně kontinuálně přenášeny na GPU, kde k nim shadery mohou přistupovat. Proměnné typu uniform jsou primárním mechanismem pro tento přenos dat z CPU na GPU. Tvorba těchto dynamických efektů tedy zahrnuje těsnou zpětnovazební smyčku: CPU vypočítá stav -> CPU aktualizuje uniform proměnné na GPU -> GPU vykoná shadery s použitím aktualizovaných uniform proměnných -> vizuální změna na obrazovce. Příklady jako použití glfwGetTime() pro aktualizaci uniform proměnné nebo myShader.setUniform('time', millis()) v p5.js demonstrují tuto CPU řízenou aktualizaci.

Ve své podstatě jsou shadery programy, které transformují vstupní data na výstupní data. Vertex shadery transformují atributy vertexů (jako pozice v modelovém prostoru) na pozice v clip space. Fragment shadery transformují interpolovaná data a uniform proměnné na finální barvy pixelů. Požadavek na "změnu barvy na základě rotace" je specifickým případem této transformace: shader bude naprogramován tak, aby transformoval vstupní data o rotaci (a translaci) na výstupní RGBA hodnoty barvy. Pochopení shaderů jako flexibilních transformátorů dat pomáhá konceptualizovat, jak různé vstupy (nejen rotace/translace, ale také čas, vlastnosti světla, materiálové vlastnosti atd.) mohou být mapovány na rozmanité vizuální výstupy.

Sekce 2: Základní koncepty GLSL pro tvorbu efektů

Pro efektivní tvorbu dynamických vizuálních efektů v GLSL je nezbytné porozumět klíčovým typům proměnných, datovým typům a vestavěným funkcím, které tento jazyk nabízí.

Klíčové kvalifikátory proměnných a jejich role:

- **uniform**: Tyto proměnné slouží k předávání dat z aplikace běžící na CPU do shaderů na GPU. Jejich hodnota je konstantní pro všechny vertexy zpracovávané v jednom vykreslovacím volání (pro vertex shadery) a pro všechny fragmenty zpracovávané pro dané primitivum v tomto volání (pro fragment shadery).
    - *Použití v kontextu dotazu:* Nezbytné pro předávání parametrů jako uModelMatrix, uViewMatrix, uProjectionMatrix a, což je pro tento dotaz nejdůležitější, vlastních dat jako u_rotationAngle (typ float reprezentující úhel rotace), u_translationVector (typ vec3 reprezentující posunutí objektu) nebo u_time (typ float pro časově závislé animace).

    - *Charakteristiky:* Deklarují se v globálním rozsahu. Uchovávají si svou hodnotu, dokud nejsou explicitně aktualizovány aplikací. Aplikace získává jejich "umístění" (identifikátor) pomocí funkce glGetUniformLocation() a poté aktualizuje jejich hodnoty pomocí funkcí jako glUniform*() (např. glUniform1f() pro float, glUniformMatrix4fv() pro matici 4x4) poté, co je shader program aktivován pomocí glUseProgram(). Pokud je uniform proměnná deklarována, ale není použita, kompilátory ji často optimalizují (odstraní).

- **in (Vstup vertex shaderu - dříve attribute)**: Tyto proměnné reprezentují data pro jednotlivé vertexy dodávaná aplikací, jako jsou pozice vertexů (a_position), normály (a_normal) nebo texturovací souřadnice (a_texCoord).
    - *Zdroj dat:* Tato data typicky pocházejí z Vertex Buffer Objects (VBOs) na GPU a aplikace konfiguruje, jak jsou tato data strukturována a jak k nim přistupovat pomocí ukazatelů na atributy vertexů.
    - *Propojení:* Kvalifikátory layout (location = N) se často používají k explicitnímu propojení těchto in proměnných s konkrétními sloty atributů.
- **out (Výstup vertex shaderu) / in (Vstup fragment shaderu - dříve varying)**: Tyto proměnné se používají k předávání dat z vertex shaderu do fragment shaderu. Klíčovou charakteristikou je, že tato data jsou během rasterizace interpolována přes povrch primitiva (např. trojúhelníku).
    - *Použití v kontextu dotazu:* Nezbytné pro posílání hodnot vypočítaných pro každý vertex (jako je faktor intenzity odvozený od rotace, transformované normály nebo pozice ve světovém prostoru) do fragment shaderu, kde budou dostupné pro každý fragment po interpolaci.
    - *Propojení:* Aby data správně prošla, musí mít out proměnná ve vysílajícím shaderu (vertex shader) a odpovídající in proměnná v přijímajícím shaderu (fragment shader) naprosto stejný název a typ. Pro starší verze GLSL se používalo klíčové slovo varying.

Základní datové typy GLSL:

- **Vektory (vec2, vec3, vec4)**: Používají se k reprezentaci vícesložkových dat, jako jsou 2D/3D/4D pozice, směry, barvy (RGBA) a texturovací souřadnice (UV/STPQ).
    - *Přístup ke komponentám (Swizzling):* GLSL umožňuje flexibilní přístup a přeskupování komponent vektorů pomocí tečkové notace s písmeny jako

.x,.y,.z,.w (pro pozice/vektory), .r,.g,.b,.a (pro barvy) nebo .s,.t,.p,.q (pro texturovací souřadnice). Například aColor.rgb nebo aPosition.xyz.

- **Matice (mat2, mat3, mat4)**: Primárně se používají pro geometrické transformace. mat4 (matice 4x4) je standardní pro modelové, pohledové a projekční transformace.
- **Skaláry (float, int, bool)**: Pro jednotlivé číselné nebo logické hodnoty.

Nezbytné vestavěné funkce GLSL pro tvorbu efektů:

- **Trigonometrické a exponenciální:** sin(), cos() (životně důležité pro vytváření periodických efektů z úhlů rotace nebo času), abs() (pro získání absolutních hodnot, často pro normalizaci oscilačních výstupů), pow() (pro nelineární odezvy).
- **Vektorové operace:** length() (vypočítá velikost vektoru, užitečné pro translační efekty), distance() (vzdálenost mezi dvěma body), normalize() (škáluje vektor na jednotkovou délku, nezbytné pro směrové vektory), dot() (skalární součin, užitečný pro projekce, výpočty úhlů, osvětlení).
- **Interpolace a omezování:** mix(x, y, a) (lineární interpolace mezi x a y pomocí faktoru a - základní pro plynulé prolínání barev nebo hodnot), clamp(val, minVal, maxVal) (omezí val na rozsah [minVal, maxVal], klíčové pro udržení hodnot barev/alphy v platném rozsahu), step(edge, x) (vrací 0.0 pokud x < edge a 1.0 pokud x >= edge, pro vytváření ostrých přechodů).
- **Výstupy shaderových fází:** gl_Position (vec4 ve vertex shaderu pro souřadnice v clip space), gl_FragColor (vec4 ve fragment shaderu pro výstupní RGBA barvu).

Opakujícím se vzorem v příkladech GLSL pro dynamické efekty je normalizace nebo mapování vstupních hodnot na předvídatelný rozsah, typicky nebo [-1, 1]. Například souřadnice myši jsou děleny rozlišením obrazovky, nebo časově řízené hodnoty procházejí funkcemi sin() či cos(), aby poskytly hodnoty v rozsahu [-1, 1], které mohou být dále zpracovány (např. abs() nebo 0.5 * value + 0.5) pro mapování na . Vstupní parametry – úhel rotace (který může být neomezený nebo v různých jednotkách jako stupně/radiány) a velikost translace (která může být také libovolná) – nejsou přímo vhodné pro řízení barevných komponent (R, G, B, A) nebo faktorů funkce mix(), jelikož ty obvykle očekávají hodnoty v rozsahu . Proto bude klíčovým krokem v logice shaderu transformace nebo normalizace těchto surových dat o rotaci a translaci na omezený, předvídatelný faktor.

Funkce mix(start_value, end_value, factor) je často používána pro vytváření plynulých přechodů mezi dvěma stavy (např. barvami, skalárními hodnotami) na základě interpolačního factor v rozsahu od 0.0 do 1.0. Tato funkce je výjimečně vhodná pro

požadavek na změnu barvy na základě rotace/translace a následné kombinování těchto efektů. Může plynule interpolovat mezi dvěma definovanými barvami pomocí normalizovaného faktoru rotace nebo translace. Navíc, mix() může sama o sobě být metodou pro kombinování dvou samostatně vypočítaných barevných efektů.

Tabulka 1: Klíčové GLSL proměnné a funkce pro dynamické efekty

GLSL Klíčové slovo/Funkce	Typ/Signatura	Účel v tomto kontextu	Stručný příklad
uniform	(různé typy)	Předávání dat z CPU do shaderů (např. transformační matice, čas, vlastní parametry).	uniform float u_time;
in (atribut vertexu)	vec3, vec2, atd.	Vstupní data pro každý vertex (pozice, normála, tex. souřadnice).	layout (location = 0) in vec3 a_position;
out (výstup vertex shaderu)	(různé typy)	Předávání interpolovaných dat do fragment shaderu.	out vec3 v_worldNormal;
in (vstup fragment shaderu)	(stejný typ jako out z VS)	Příjem interpolovaných dat z vertex shaderu.	in vec3 v_worldNormal;
vec2, vec3, vec4	vecN	Reprezentace pozic, směrů, barev, texturovacích souřadnic.	vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
mat4	mat4	Reprezentace 4x4 transformačních matic.	uniform mat4 u_modelMatrix;
gl_Position	vec4	Výstupní pozice vertexu v clip space (pouze ve VS).	gl_Position = mvp * a_position;
gl_FragColor	vec4	Výstupní RGBA barva fragmentu (pouze ve FS).	gl_FragColor = finalColor;

mix(genType x, genType y, a)	genType	Lineární interpolace mezi x a y pomocí faktoru a (0 až 1).	vec3 c = mix(colorA, colorB, factor);
sin(float angle)	float	Sinus úhlu (v radiánech).	float s = sin(u_rotationAngle);
cos(float angle)	float	Kosinus úhlu (v radiánech).	float c = cos(u_rotationAngle);
length(genType x)	float	Délka (velikost) vektoru.	float dist = length(translationVector);
normalize(genType x)	genType	Vrátí normalizovaný vektor (jednotková délka).	vec3 n = normalize(a_normal);
clamp(genType x, min, max)	genType	Omezí x na rozsah [min, max].	float a = clamp(value, 0.0, 1.0);
abs(genType x)	genType	Absolutní hodnota x.	float factor = abs(sinVal);
dot(genType x, genType y)	float	Skalární součin dvou vektorů.	float d = dot(normal, lightDir);
step(genType edge, genType x)	genType	Vrátí 0.0 pokud x < edge, jinak 1.0.	float onOff = step(0.5, value);

Export to Sheets

Sekce 3: Předávání transformačních dat do shaderů

Pro dosažení dynamických efektů závislých na rotaci a translaci je klíčové efektivně předat relevantní transformační data z aplikace do shaderů.

Standardní transformace: MVP maticový pipeline V konvenčním grafickém pipeline hrají ústřední roli Modelová (Model), Pohledová (View) a Projekční (Projection) matice, často předávané do vertex shaderu jako uniform mat4 uModelMatrix;, uniform mat4 uViewMatrix; a uniform mat4 uProjectionMatrix;. Tyto matice postupně transformují pozice vertexů z lokálního modelového prostoru přes světový prostor a pohledový (kamerový) prostor až do konečného clip space, který je použit pro rasterizaci. Typická operace ve vertex shaderu je: gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix * vec4(aPosition, 1.0f); Porozumění tomuto standardnímu toku je důležité, protože rotace a translace objektu jsou

inherentně součástí uModelMatrix. Požadované efekty jsou založeny právě na těchto transformacích. Pořadí násobení matic je klíčové: obvykle se aplikuje škálování, poté rotace a nakonec translace (TranslationMatrix * RotationMatrix * ScaleMatrix) pro sestavení modelové matice.

Strategie pro předávání dat o rotaci pro efekty:

- **Metoda 1: Přímé uniform proměnné pro úhel/osu (např. uniform float u_rotationAngleY;)**
    - *Popis:* Aplikace posílá aktuální úhel rotace (např. kolem osy Y) přímo jako uniform float. Pokud je rotace kolem libovolné osy, mohou být předány uniform float u_rotationAngle; a uniform vec3 u_rotationAxis;.
    - *Použití ve vertex shaderu:* Lze použít k sestavení rotační matice pro transformaci aPosition a aNormal, pokud již nejsou součástí uModelMatrix, nebo jednoduše předat dále.
    - *Použití ve fragment shaderu:* u_rotationAngleY (nebo varying proměnná nesoucí tuto hodnotu) může být přímo použita s funkcemi sin() nebo cos() k vygenerování faktoru, který ovlivňuje barvu/alphu. Toto je často nejpřímočařejší pro efekty vázané na specifický úhel.
- **Metoda 2: Odvození informací o rotaci z uModelMatrix nebo transformovaných vektorů**
    - *Popis:* Pokud je již plná uModelMatrix (která zahrnuje rotaci) posílána pro transformaci vertexů, lze zvážit extrakci informací o rotaci z ní. Dekompozice matice pro získání Eulerových úhlů v shaderu je však komplexní a obecně se pro tento účel nedoporučuje.
    - *Alternativa:* Praktičtějším přístupem je, aby aplikace vypočítala "faktor rotace" (např. abs(sin(angle))) na CPU a předala jej jako jednoduchý uniform float u_rotationEffectFactor;.
    - *Jiná alternativa (založená na orientaci):* Pokud efekt závisí na orientaci objektu spíše než na spojitém úhlu rotace (např. "horní plocha je zelená, dolní modrá"), vertex shader může transformovat normály vertexů pomocí modelové matice (vec3 worldNormal = mat3(uModelMatrix) * aNormal;) a předat worldNormal (nebo normálu v pohledovém prostoru) do fragment shaderu. Fragment shader pak může použít komponenty této normály (např. v_worldNormal.y) k určení barvy.

Strategie pro předávání dat o translaci pro efekty:

- **Metoda 1: Přímý uniform vektor translace (např. uniform vec3 u_translationOffset;)**
    - *Popis:* Aplikace posílá aktuální translaci objektu (např. jeho posunutí od počátku nebo jeho absolutní světovou pozici) jako uniform vec3.
    - *Použití ve vertex shaderu:* Tento vektor se používá při výpočtu gl_Position.
    - *Použití ve fragment shaderu:* Samotný u_translationOffset, jeho délka (length()) nebo jedna z jeho komponent (např. u_translationOffset.x) může být předána přes varying proměnnou nebo použita jako uniform přímo ve fragment shaderu k odvození "faktoru translace".
- **Metoda 2: Extrakce translace z uModelMatrix**
    - *Popis:* Translační komponenty standardní modelové matice se nacházejí ve čtvrtém sloupci (indexy 12, 13, 14 pro matici uloženou po sloupcích). V GLSL je to pro mat4 M výraz M.xyz.
    - *Použití ve vertex shaderu:* Vertex shader může tuto informaci extrahovat: vec3 worldPosition = uModelMatrix.xyz; a předat ji do fragment shaderu přes varying vec3 v_worldPosition;.
- **Metoda 3: Předání transformované pozice vertexu**
    - *Popis:* Vertex shader vypočítá světovou pozici každého vertexu (vec4 worldPos = uModelMatrix * vec4(aPosition, 1.0);) a předá worldPos.xyz do fragment shaderu přes varying vec3 v_worldPos;. Fragment shader pak obdrží interpolovanou světovou pozici pro každý fragment.

Zatímco uModelMatrix inherentně obsahuje všechny informace o rotaci a translaci, její přímé předání do fragment shaderu pro extrakci jednoduchého úhlu nebo posunutí může být neefektivní a zbytečně komplikovat logiku fragment shaderu. Pokud efekt fragmentu závisí *pouze* například na úhlu rotace kolem osy Y, je mnohem efektivnější, aby CPU tento úhel vypočítalo a předalo ho jako jedinou uniform float u_rotationAngleY;. Fragment shader pak pracuje s přímou, relevantní hodnotou. Naopak, pokud fragment shader potřebuje více komponent transformace (např. plnou orientaci a pozici), předání matice (nebo relevantních částí jako transformované normály a pozice z VS) může být stručnější než mnoho jednotlivých skalárních/vektorových uniform proměnných. Je vhodné volit nejpřímější a sémanticky nejrelevantnější formu dat pro efekty ve fragment shaderu.

Vertex shader, který běží pro každý vertex, může provádět výpočty nebo transformace, jejichž výsledky jsou poté interpolovány a použity fragment shaderem (který běží pro každý fragment/pixel). To může být optimalizace, pokud by se výpočet jinak redundantně prováděl pro každý fragment. Může také zjednodušit logiku fragment shaderu tím, že mu poskytne snadno použitelná data. Například místo toho, aby fragment shader přijímal surový úhel a poté počítal abs(sin(angle)), mohl by vertex shader tento rotationFactor vypočítat jednou pro každý vertex a předat ho jako varying. Interpolovaný v_rotationFactor by pak byl přímo použitelný ve fragment shaderu. To platí zejména, pokud je výpočet faktoru rotace netriviální. Podobně pro translaci může vertex shader snadno vypočítat pozice ve světovém prostoru nebo extrahovat translační vektory.

Tabulka 2: Data pro efekty rotace/translace

Parametr efektu	**Datový typ**	**Zdroj (CPU/VS)**	**Příklad Uniform/Varying v GLSL**	**Typické použití v shaderu**
Úhel rotace	float	CPU	uniform float u_rotationAngle;	sin(u_rotationAngle) ve FS
Osa rotace	vec3	CPU	uniform vec3 u_rotationAxis;	(Použito s úhlem k sestavení matice nebo pro komplexní efekty)
Modelová matice	mat4	CPU	uniform mat4 uModelMatrix;	VS: uModelMatrix * aPosition, FS: (méně přímé)
Translační vektor	vec3	CPU	uniform vec3 u_translationVector;	VS: aPosition + u_translationVector, FS: length(u_translationVector)
Světová pozice (objektu/vertexu)	vec3	VS z ModelMatrix	varying vec3 v_worldPosition;	FS: v_worldPosition.x pro řízení gradientu
Čas	float	CPU	uniform float u_time;	FS: sin(u_time) pro animaci

Export to Sheets

Sekce 4: Efekt 1 - Barva a průhlednost z rotace

Tato sekce popisuje implementaci efektu, kde se barva a průhlednost objektu mění v závislosti na jeho rotaci. Jako příklad použijeme rotaci kolem osy Y.

Logika Vertex Shaderu: Vertex shader má za úkol provést standardní transformaci pozice vertexu a vypočítat "faktor rotace", který bude předán fragment shaderu.

- **Vstupy:** in vec3 aPosition; (pozice vertexu), volitelně in vec3 aNormal; (pokud jsou normály použity pro jiné efekty nebo pokud je efekt rotace založen na orientaci).
- **Uniformy:** uniform mat4 uModelMatrix; uniform mat4 uViewMatrix; uniform mat4 uProjectionMatrix; uniform float u_rotationAngleY; (příklad: úhel rotace kolem osy Y, předaný z CPU).
- **Výstupy:** out float v_rotationFactor; (pro předání vypočítaného faktoru do fragment shaderu).
- **Funkce main():**
    1. Standardní transformace vertexu: gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix * vec4(aPosition, 1.0);
    2. Výpočet faktoru rotace: v_rotationFactor = abs(sin(u_rotationAngleY));. Tato operace mapuje úhel na oscilující hodnotu v rozsahu . Jiné mapování je možné, např. (sin(u_rotationAngleY) * 0.5) + 0.5; pro plynulou oscilaci 0-1-0.
    3. *Alternativa pro efekt založený na orientaci:* Pokud uModelMatrix obsahuje rotaci, vypočítá se vec3 worldNormal = normalize(mat3(transpose(inverse(uModelMatrix))) * aNormal); a předá se například v_orientationFactor = worldNormal.y; (nebo abs(worldNormal.y)).

Logika Fragment Shaderu: Fragment shader přijme interpolovaný faktor rotace a použije jej k určení finální barvy a průhlednosti fragmentu.

- **Vstupy:** in float v_rotationFactor; (interpolovaný z vertex shaderu).
- **Uniformy:** Volitelné, např. uniform vec3 u_colorA; uniform vec3 u_colorB;.
- **Výstupy:** out vec4 gl_FragColor;.
- **Funkce main():**
    1. Definice základních barev pro efekt: vec3 colorStart = vec3(1.0, 0.0, 0.0); // Červená a vec3 colorEnd = vec3(0.0, 0.0, 1.0); // Modrá.

2. **Změna barvy na základě rotace:** Interpolace mezi colorStart a colorEnd pomocí v_rotationFactor: vec3 effectColor = mix(colorStart, colorEnd, v_rotationFactor);.
3. **Změna průhlednosti na základě rotace:** Odvození hodnoty alfa z v_rotationFactor. Například: float effectAlpha = v_rotationFactor; (objekt se stává více neprůhledným, jak se faktor blíží 1). Nebo float effectAlpha = 1.0 - v_rotationFactor; (objekt se stává více průhledným). Lze použít pow pro nelineární odezvu: float effectAlpha = pow(v_rotationFactor, 2.0);.
4. Je důležité zajistit, aby hodnota alfa byla omezena na platný rozsah : effectAlpha = clamp(effectAlpha, 0.0, 1.0);. Může být žádoucí minimální alfa, např. 0.1, pokud objekt nemá být nikdy zcela neviditelný.
5. Nastavení finálního výstupu: gl_FragColor = vec4(effectColor, effectAlpha);.

Surové úhly rotace (např. 0 až 360 stupňů nebo 0 až 2π radiánů) často nejsou přímo vhodné pro řízení změn barvy nebo alfa, protože jsou neomezené nebo jejich surové hodnoty se špatně mapují na rozsah očekávaný pro barevné komponenty nebo faktory funkce mix. Použití periodických funkcí jako sin() nebo cos() transformuje úhel na omezený rozsah [-1,1]. Další operace jako abs() nebo (value * 0.5) + 0.5 mohou toto mapovat na rozsah , čímž se vytvářejí plynulé, oscilující nebo pulzující vizuální efekty. Volba mapovací funkce (sin, cos, abs, pow, modulo aritmetika pro úhly) přímo určuje charakter vizuálního efektu.

Matematické operace v shaderech, zejména ty zahrnující čísla s plovoucí desetinnou čárkou nebo komplexní sekvence funkcí, mohou někdy vést k hodnotám mírně mimo očekávaný rozsah pro barevné komponenty nebo interpolační faktory kvůli problémům s přesností nebo povaze samotných funkcí. Hodnoty mimo pro RGBA komponenty mohou vést k nedefinovanému chování nebo artefaktům při renderování. Funkce clamp(value, 0.0, 1.0) explicitně omezuje hodnotu na požadovaný rozsah , čímž zajišťuje, že výstupy barvy a alfa zůstanou platné a předvídatelné.

Příklad pseudokódu:

Vertex Shader:

OpenGL Shading Language

#version 330 core

layout (location = 0) in vec3 aPosition;

```
uniform mat4 uModelMatrix;
uniform mat4 uViewMatrix;
uniform mat4 uProjectionMatrix;
uniform float u_rotationAngleY; // Úhel rotace kolem osy Y v radiánech

out float v_rotationFactor;

void main() {
   gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix * vec4(aPosition, 1.0);

   // Výpočet faktoru rotace (0 až 1, osciluje s rotací)
   v_rotationFactor = abs(sin(u_rotationAngleY));
}
```

Fragment Shader:

OpenGL Shading Language

```
#version 330 core
in float v_rotationFactor; // Interpolovaný faktor rotace

uniform vec3 u_colorA_rot = vec3(1.0, 0.0, 0.0); // Červená
uniform vec3 u_colorB_rot = vec3(0.0, 0.0, 1.0); // Modrá

out vec4 gl_FragColor;

void main() {
   // Změna barvy na základě rotace
   vec3 effectColor = mix(u_colorA_rot, u_colorB_rot, v_rotationFactor);
```

```
    // Změna průhlednosti na základě rotace
    // Objekt je více neprůhledný, když je v_rotationFactor blíže k 1
    float effectAlpha = v_rotationFactor;
    // Alternativa: objekt je více průhledný, když je v_rotationFactor blíže k 1
    // float effectAlpha = 1.0 - v_rotationFactor;

    // Omezení alfa na rozsah , případně nastavení minimální alfy
    effectAlpha = clamp(effectAlpha, 0.1, 1.0); // Minimální alfa 0.1

    gl_FragColor = vec4(effectColor, effectAlpha);
}
```

Sekce 5: Efekt 2 - Barva z translace

Tato sekce se zaměřuje na změnu barvy objektu na základě jeho translace (posunutí) ve scéně.

Logika Vertex Shaderu: Vertex shader vypočítá standardní transformaci a připraví "faktor translace" pro fragment shader.

- **Vstupy:** in vec3 aPosition;.
- **Uniformy:** uniform mat4 uModelMatrix; uniform mat4 uViewMatrix; uniform mat4 uProjectionMatrix; uniform vec3 u_objectWorldPosition; (příklad: střed objektu ve světových souřadnicích, předaný z CPU; může to být také u_translationVector, pokud reprezentuje plnou světovou pozici). Alternativně, pokud efekt závisí na pozici každého vertexu, může být uModelMatrix použita k výpočtu světové pozice vertexu.
- **Výstupy:** out float v_translationFactor; (nebo out vec3 v_worldVertexPos;).
- **Funkce main():**
    1. Standardní transformace vertexu: gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix * vec4(aPosition, 1.0);

2. Výpočet faktoru translace. To vyžaduje definování, který aspekt translace řídí barvu.
    - *Možnost 1 (Velikost XZ translace od počátku):* float distXZ = length(u_objectWorldPosition.xz); v_translationFactor = clamp(distXZ / u_maxExpectedDistance, 0.0, 1.0); (Vyžaduje uniform float u_maxExpectedDistance; pro normalizaci).
    - *Možnost 2 (Na základě světové X souřadnice):* v_translationFactor = (u_objectWorldPosition.x / u_worldBoundsX) * 0.5 + 0.5; (Normalizováno na na základě předpokládaných hranic světa u_worldBoundsX).
    - *Možnost 3 (Předání světové pozice vertexu):* out vec3 v_worldVertexPos;... v_worldVertexPos = (uModelMatrix * vec4(aPosition, 1.0)).xyz;. Fragment shader pak použije v_worldVertexPos. Příklad v ukazuje použití u_offset (translace) k modifikaci gl_Position a předání v_positionWithOffset.

Logika Fragment Shaderu: Fragment shader použije interpolovaný faktor translace k určení barvy.

- **Vstupy:** in float v_translationFactor; (nebo in vec3 v_worldVertexPos;).
- **Uniformy:** Volitelné, např. uniform vec3 u_colorC_trans; uniform vec3 u_colorD_trans;.
- **Výstupy:** out vec4 gl_FragColor;.
- **Funkce main():**
    1. Definice základních barev pro tento efekt: vec3 colorC = vec3(0.0, 1.0, 0.0); // Zelená a vec3 colorD = vec3(1.0, 1.0, 0.0); // Žlutá.
    2. **Změna barvy na základě translace:**
        - vec3 translationEffectColor = mix(colorC, colorD, v_translationFactor);
        - Pokud se používá v_worldVertexPos: float factor = clamp(v_worldVertexPos.x / 10.0, 0.0, 1.0); // Příklad: barva se mění přes 10 jednotek v ose X
        - translationEffectColor = mix(colorC, colorD, factor);

3. Nastavení finálního výstupu (předpokládá se plná neprůhlednost pro tento specifický efekt, nebo by alfa mohla být také modulována): gl_FragColor = vec4(translationEffectColor, 1.0);.

Požadavek "při posunu měnit jinak barvu" je kvalitativní. Pro implementaci musí být "translace" kvantifikována způsobem, který může řídit změnu barvy. Může to být absolutní světová X, Y nebo Z souřadnice objektu, jeho vzdálenost od světového počátku, velikost jeho vektoru pohybu od posledního snímku nebo jeho pozice vzhledem k nějaké definované zóně ve světě. Volba kvantifikace přímo určuje povahu a chování vizuálního efektu. Pokud se barva mění na základě absolutní X světové souřadnice objektu, vytvoří to prostorový gradientní efekt napříč světem. Pokud je založena na *velikosti* translačního vektoru (např. length(u_translationOffset)), barva se může měnit, jak se objekt vzdaluje od svého výchozího bodu, bez ohledu na směr.

Efekty mohou být řízeny vlastnostmi objektu ve světových souřadnicích (pevné ve 3D scéně) nebo v souřadnicích obrazovky (relativní k viewportu). Pro efekt vázaný na "translaci objektu" se obecně předpokládá, že se to týká pohybu nebo pozice objektu v rámci 3D světa, nejen jeho pozice na 2D obrazovce (která se také mění s pohybem kamery). Pokud by se pro určení barvy použila gl_FragCoord.x (X souřadnice v prostoru obrazovky, jak je vidět v pro gradienty obrazovky), barva by se měnila, jak by se *kamera* pohybovala kolem stacionárního objektu, nebo jak by se objekt pohyboval po obrazovce. Avšak "při posunu [objektu]" silně naznačuje, že efekt by měl být vázán na vlastní transformaci objektu ve světě. To vyžaduje použití světových souřadnic (např. předaných jako varying z vertex shaderu po modelové transformaci, jak naznačuje) nebo uniform proměnné reprezentující světovou translaci objektu.

Příklad pseudokódu:

Vertex Shader:

OpenGL Shading Language

#version 330 core

layout (location = 0) in vec3 aPosition;

uniform mat4 uModelMatrix;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

```
// Předpokládáme, že uModelMatrix již obsahuje translaci objektu
// Alternativně lze předat uniform vec3 u_objectWorldPosition;

out float v_translationFactor;
// Maximální očekávaná vzdálenost od počátku pro normalizaci (např. poloměr scény)
uniform float u_maxDistance = 20.0;

void main() {
  gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix * vec4(aPosition, 1.0);

  // Získání světové pozice objektu z modelové matice
  vec3 objectWorldPos = uModelMatrix.xyz;

  // Faktor založený na vzdálenosti od světového počátku v rovině XZ
  float distanceXZ = length(objectWorldPos.xz);
  v_translationFactor = clamp(distanceXZ / u_maxDistance, 0.0, 1.0);
}
```

Fragment Shader:

```
OpenGL Shading Language
#version 330 core
in float v_translationFactor; // Interpolovaný faktor translace

uniform vec3 u_colorC_trans = vec3(0.0, 1.0, 0.0); // Zelená
uniform vec3 u_colorD_trans = vec3(1.0, 1.0, 0.0); // Žlutá
```

```
out vec4 gl_FragColor;


void main() {
  // Změna barvy na základě translace
  vec3 translationEffectColor = mix(u_colorC_trans, u_colorD_trans, v_translationFactor);


  gl_FragColor = vec4(translationEffectColor, 1.0); // Plná neprůhlednost
}
```

Sekce 6: Kombinování efektů rotace a translace

Po implementaci jednotlivých efektů pro rotaci a translaci je dalším krokem jejich kombinace, aby objekt reagoval na oba typy transformací současně.

Základní strategie: Izolovat, vypočítat, poté zkombinovat. Efektivní přístup spočívá v tom, že vertex shader vypočítá a předá všechny potřebné faktory pro oba efekty. Fragment shader poté samostatně vypočítá barevné příspěvky od rotace a translace a následně tyto příspěvky zkombinuje.

- Vertex shader by měl předávat například: out float v_rotationFactor; out float v_translationFactor;.
- Fragment shader provede následující kroky:
    1. Vypočítá barevný příspěvek a alfu z rotace (např. vec3 rotColor, float rotAlpha).
    2. Vypočítá barevný příspěvek z translace (např. vec3 transColor).
    3. Zkombinuje rotColor a transColor a určí finální alfu.

Metody pro kombinování barevných příspěvků ve fragment shaderu:

- **Aditivní kombinace:** vec3 combinedRGB = rotColor.rgb + transColor.rgb;
    - *Efekt:* Má tendenci zesvětlovat barvy, podobně jako přidávání světelných zdrojů. Může vést k přesycení, pokud není řízeno. Tento přístup je analogický tomu, jak se sčítají příspěvky více světel.
- **Multiplikativní kombinace (Modulace):** vec3 combinedRGB = rotColor.rgb * transColor.rgb;

    - *Efekt:* Má tendenci ztmavovat barvy, protože každá komponenta je škálována druhou. Užitečné pro efekty podobné filtrům.

- **Interpolace (mix):**
    - vec3 combinedRGB = mix(rotColor.rgb, transColor.rgb, 0.5f); (Jednoduché prolnutí 50/50).
    - vec3 combinedRGB = mix(rotColor.rgb, transColor.rgb, u_blendFactor); (Použití další uniform proměnné u_blendFactor k dynamickému řízení míry prolnutí).
    - vec3 combinedRGB = mix(rotColor.rgb, transColor.rgb, v_rotationFactor); (Použití jednoho z existujících faktorů k řízení prolnutí – např. jak se objekt více otáčí, translační barva se stává výraznější). Klíčovým odkazem pro funkci mix() je.
- **Kombinace alfa kanálu:**
    - Typicky může být alfa z rotačního efektu (rotAlpha) primární alfou.
    - Nebo je lze kombinovat: float finalAlpha = rotAlpha * u_someOtherFactor; nebo min(rotAlpha, translationEffectAlpha), pokud translace také ovlivňuje alfu.
    - Funkce prolnutí nastavená pomocí glBlendFunc (viz Sekce 7) určí, jak tato finální alfa interaguje s framebufferem.

Finální výstup: gl_FragColor = vec4(clamp(combinedRGB, 0.0, 1.0), clamp(finalAlpha, 0.0, 1.0)); Omezení hodnot pomocí clamp() je klíčové pro zajištění platných hodnot barvy a alfa.

Na rozdíl od fyzikálně založeného renderování, kde často existují "správné" vzorce, kombinování uměleckých vizuálních efektů je vysoce subjektivní a závisí na požadované estetice. Neexistuje jediný "správný" způsob, jak prolnout změnu barvy založenou na rotaci se změnou barvy založenou na translaci. Volba matematické operace (+, *, mix(), vlastní logika) významně mění konečný vzhled. Aditivní prolnutí obecně zesvětluje výsledky , multiplikativní prolnutí má tendenci ztmavovat nebo vytvářet efekty podobné filtrům, a mix() poskytuje řízený přechod.

V shaderovém programování, stejně jako v obecném programování, je pořadí operací kritické. Při kombinování více výpočtů barev určí konečný výsledek sekvence, v níž jsou aplikovány nebo prolnuty. Například (colorFromRotation + colorFromTranslation) * masterBrightness se liší od (colorFromRotation * masterBrightness) + colorFromTranslation. Pokud má rotační efekt definovat základní paletu barev a translační

efekt má tuto paletu tónovat nebo modifikovat, logický tok by byl: 1. Vypočítat základní barvu z rotace. 2. Vypočítat modifikaci/tónování z translace. 3. Aplikovat modifikaci na základní barvu.

Tabulka 3: Přístupy ke kombinaci efektů

Metoda kombinace	Koncepční logika	Příklad GLSL fragmentu (zaměření na Fragment Shader)
Aditivní	Akumulace světla/barvy, zesvětlení	vec3 finalColor = colorA + colorB;
Multiplikativní	Filtrování, ztmavení	vec3 finalColor = colorA * colorB;
Alfa prolnutí (pomocí mix)	Plynulé prolnutí mezi dvěma barvami	vec3 finalColor = mix(colorA, colorB, factor);
Modulace jednoho efektu druhým	Jeden efekt ovlivňuje intenzitu/charakter druhého	vec3 finalColor = baseEffectColor * modulatingFactor;

Export to Sheets

Příklad pseudokódu fragment shaderu pro kombinovaný přístup: Předpokládá se, že vertex shader předává v_rotationFactor a v_translationFactor.

OpenGL Shading Language

```
#version 330 core

in float v_rotationFactor;

in float v_translationFactor;


// Uniformy pro barvy definované v předchozích sekcích

uniform vec3 u_colorA_rot = vec3(1.0, 0.0, 0.0); // Červená pro rotaci

uniform vec3 u_colorB_rot = vec3(0.0, 0.0, 1.0); // Modrá pro rotaci

uniform vec3 u_colorC_trans = vec3(0.0, 1.0, 0.0); // Zelená pro translaci

uniform vec3 u_colorD_trans = vec3(1.0, 1.0, 0.0); // Žlutá pro translaci
```

```
// Uniforma pro řízení způsobu kombinace (0.0 = pouze rotace, 1.0 = pouze translace)
uniform float u_combinationFactor = 0.5;

out vec4 gl_FragColor;

void main() {
  // Příspěvek od rotace
  vec3 rotEffectColor = mix(u_colorA_rot, u_colorB_rot, v_rotationFactor);
  float rotEffectAlpha = clamp(v_rotationFactor, 0.1, 1.0); // Alfa z rotace

  // Příspěvek od translace
  vec3 transEffectColor = mix(u_colorC_trans, u_colorD_trans, v_translationFactor);
  // Pro jednoduchost předpokládáme, že translace neovlivňuje alfu přímo,
  // ale mohla by, např. float transEffectAlpha =...;

  // Kombinace barevných příspěvků (např. interpolací)
  vec3 combinedRGB = mix(rotEffectColor, transEffectColor, u_combinationFactor);

  // Finální alfa je zde brána z rotačního efektu
  float finalAlpha = rotEffectAlpha;
  // Alternativně lze alfy také kombinovat:
  // float finalAlpha = mix(rotEffectAlpha, 1.0, u_combinationFactor);

  gl_FragColor = vec4(clamp(combinedRGB, 0.0, 1.0), clamp(finalAlpha, 0.0, 1.0));
}
```

Sekce 7: Praktické aspekty

Implementace shaderových efektů vyžaduje nejen správný kód v GLSL, ale také odpovídající nastavení na straně aplikace a pochopení určitých konceptů.

Nastavení na straně aplikace: Povolení prolnutí (Blending) pro průhlednost Aby měla hodnota alfa menší než 1.0 ve gl_FragColor viditelný efekt průhlednosti, je klíčové povolit prolnutí (blending) v OpenGL. Jednotka prolnutí na GPU kombinuje výstup fragment shaderu (zdrojová barva) s barvou již přítomnou ve framebufferu (cílová barva) na základě nastavených faktorů. Bez správného nastavení prolnutí nebude alfa kanál vypočítaný fragment shaderem fungovat podle očekávání. Standardní nastavení pro běžnou alfa-průhlednost v C++ (nebo ekvivalentu v jiném jazyce) je:

C++

```
glEnable(GL_BLEND);

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Tato konfigurace typicky implikuje následující vzorec prolnutí:
$FinalColor = SourceColor \cdot SourceAlpha + DestinationColor \cdot (1 - SourceAlpha)$

Logika na straně aplikace: Aktualizace uniform proměnných každý snímek Pro dynamické efekty musí aplikace (CPU) aktualizovat relevantní uniform proměnné (např. u_rotationAngle, u_translationVector, u_time) každý snímek před vydáním vykreslovacího příkazu. Konceptuální příklad herní smyčky v C++:

C++

```
// V herní smyčce:

// 1. Aktualizovat currentRotationAngleY a currentTranslationVector objektu

//    na základě vstupu/animace.


// 2. Aktivovat shader program:

glUseProgram(shaderProgramID);


// 3. Získat umístění uniform proměnných (ideálně jednou při inicializaci a uložit je):

GLint rotAngleLoc = glGetUniformLocation(shaderProgramID, "u_rotationAngleY");

GLint transVecLoc = glGetUniformLocation(shaderProgramID, "u_translationVector");
```

```
// 4. Nastavit uniform proměnné:
glUniform1f(rotAngleLoc, currentRotationAngleY);
glUniform3fv(transVecLoc, 1, &currentTranslationVector);

// 5. Vykreslit objekt:
glBindVertexArray(objectVAO);
glDrawElements(/*...*/); // nebo glDrawArrays
//...
glUseProgram(0); // Deaktivovat shader program
```

Souřadnicové systémy, normalizace a mapování Je důležité si připomenout význam normalizace faktorů odvozených z rotace (úhly) nebo translace (vzdálenosti, pozice) na rozsah pro předvídatelné použití s funkcí mix(), barevnými komponentami nebo alfou. Volba souřadnicového systému pro odvození faktorů (např. použití lokální rotace objektu vs. orientace ve světovém prostoru, nebo lokální posunutí vs. pozice ve světovém prostoru) může vést k odlišnému vizuálnímu chování.

Shadery neoperují izolovaně. Jejich úspěšné provedení a správný vzhled jejich efektů závisí na širším stavu OpenGL konfigurovaném aplikací. Pro průhlednost jsou glEnable(GL_BLEND) a vhodná nastavení glBlendFunc nezbytnými předpoklady. Podobně i testování hloubky (glEnable(GL_DEPTH_TEST)) a ořezávání (glEnable(GL_CULL_FACE)) interagují s tím, jak a které fragmenty jsou nakonec renderovány.

Ačkoliv jsou požadované efekty relativně jednoduché, fragment shader běží pro každý pixel renderovaného objektu. Jak se efekty stávají složitějšími (více výpočtů, více čtení textur, více podmíněné logiky), mohou významně ovlivnit výkon. Pokud by se do fragment shaderu přidalo mnoho komplexních výpočtů, mohlo by to nevědomky vytvořit úzké hrdlo výkonu. Některé výpočty lze přesunout do vertex shaderu (který běží méněkrát, pro každý vertex), pokud se výkon stane problémem.

Sekce 8: Závěr a další možnosti průzkumu

Tato zpráva představila základní principy a techniky pro tvorbu vlastních GLSL vertex a fragment shaderů schopných dynamicky měnit barvu a průhlednost objektů na základě jejich rotace a translace, a také metody pro kombinování těchto efektů.

Shrnutí klíčových technik: Byla zdůrazněna ústřední role uniform proměnných pro předávání dynamických dat o rotaci a translaci z aplikace do shaderů. Dále byla vysvětlena funkce varying proměnných (nebo moderní syntaxe out/in) pro usnadnění toku dat z vertex shaderu do fragment shaderu, včetně klíčové interpolace. Připomněli jsme sílu matematických funkcí GLSL (sin, cos, length, mix, clamp atd.) při transformaci těchto vstupů na smysluplné faktory, které řídí změny barvy a alfa. Byly demonstrovány strategie pro kombinování jednotlivých efektů rotace a translace.

Povzbuzení k experimentování a kreativnímu průzkumu: Poskytnutý pseudokód slouží jako výchozí bod. Skutečná síla shaderů spočívá v experimentování. Doporučuje se prozkoumat další matematické funkce GLSL pro různá mapování a chování (např. smoothstep pro plynulejší nelineární přechody, fract a mod pro opakující se vzory založené na pozici nebo úhlu, jak je vidět v). Je vhodné vyzkoušet rozmanité barevné palety a různé metody kombinování příspěvků rotColor a transColor. Pro sofistikovanější efekty lze začlenit další dostupné vstupy shaderu, jako jsou:

- Normály vertexů (např. pro interakce s osvětlením nebo efekty založené na orientaci).
- Texturovací souřadnice a textury (pro aplikaci těchto dynamických změn barev na texturované objekty).
- uniform proměnná u_time pro vytváření animací, které se vyvíjejí nezávisle nebo ve spojení s rotací/translací.

Diskutované techniky – parametrizace vizuálního výstupu, použití matematických funkcí k definování vzhledu a kombinování jednoduchých pravidel k vytvoření komplexního chování – jsou základními principy v oblasti procedurálního generování obsahu, vizuálního umění v reálném čase a efektů ve stylu demoscény. Uživatel se v podstatě učí "instruovat" GPU, aby generovalo vizuály na základě pravidel a vstupů, nikoli jen renderovalo předem definovaná aktiva. Proces mapování parametrů (rotace, translace) na vizuální atributy (barva, alfa) pomocí matematických funkcí je přesně to, jak se vytváří mnoho pokročilých shaderových efektů, včetně těch, které lze vidět na platformách jako ShaderToy. Zvládnutí shaderů tak otevírá dveře k širším možnostem kreativního kódování a procedurální grafiky.

Sources used in the report

Thoughts