Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Shadery v GLSL

2017

Martina Kukulová

Vedoucí práce: RNDr. Eduard Bartl, Ph.D.

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Martina Kukulová
Název práce:	Shadery v GLSL
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2017
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	RNDr. Eduard Bartl, Ph.D.
Počet stran:	57
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Martina Kukulová
Title:	Shaders in GLSL
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2017
Study field:	Computer Science, full-time form
Supervisor:	RNDr. Eduard Bartl, Ph.D.
Page count:	57
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Bakalářská práce se zabývá shadery napsanými v OpenGL Shading Language (GLSL). Práce se podrobně zaměří na popis integrace GLSL objektů v rámci OpenGL a zapojení shaderů v různých fázích zobrazovacího procesu (především vertex a fragment). Součástí bude také návrh a implementace shaderů realizujících různé typy transformací, vizuální efekty nad texturami, dynamické textury apod. Vytvořené shadery budou integrovány do grafické automatizace služby Stream Circle (streamcircle.com).

Synopsis

The thesis deals with shaders in the OpenGL Shading Language (GLSL). It is focused on an integration of GLSL objects with an OpenGL context and shaders linkage in different phases of its rendering proccess (especially vertex and fragment). Next part of this work includes shader design and implementations for different types of transformations, dynamics texture effects and others. All the shaders are programmed for a graphics editor of a Stream Circle service (streamcircle.com).

Klíčová slova: OpenGL; GLSL; shader; vertex; fragment; rendering; real-time; Stream Circle

Keywords: OpenGL; GLSL; shader; vertex; fragment; rendering; real-time; Stream Circle

Ráda bych poděkovala svému vedoucímu bakalářské práce RNDr. Eduardu Bartlovi, Ph.D., který mi umožnil pracovat na vlastním tématu shaderů v GLSL a také celé své rodině za poskytnutou oporu. Speciální poděkování pak patří Mgr. Josefu Vašicovi, který mi dal příležitost pracovat pro Stream Circle, motivoval mě a poskytoval mi cenné rady (nejen) při psaní této práce.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

Počítačová grafika je obor pokrývající mnoho oblastí. Jednou z nich je zobrazování (rendering), proces zpracování dat popisujících modely objektů ve výsledný digitální obraz, který je vykreslen na monitor. Zpracování těchto dat probíhá v grafické kartě (GPU). Moderní GPU zvládají složité výpočty ve velmi krátkém čase, čehož využívají především aplikace běžící v reálném čase (real-time). Zasahovat do jejich vykreslovacího procesu přímo za běhu pak umožňují shadery (krátké programy zapojované do grafického vykreslovacího řetězce) a dovolují tak ovlivňovat vykreslovací proces v reálném čase. Shadery mají velký potenciál při zobrazování náročné počítačové grafiky.

Právě shadery jsou tématem této bakalářské práce. Ta nejdříve popisuje proces renderování pomocí grafické pipeline, její jednotlivé fáze, zapojení shaderů do těchto fází, představuje jazyk GLSL pro psaní shaderů, demonstruje možnosti shaderů na příkladech a zaměřuje se na jejich využití při editování obrazu živého televizního vysílání. Vše pomocí standardu OpenGL a programovacího jazyka pro psaní shaderů OpenGL Shading Language (GLSL).

2 Počítačová grafika a renderování

Počítačová grafika se zabývá zpracováním a zobrazením digitálního obrazu. Obrazem může být fotografie, kresba, 3D model, video a další. Může zachycovat jak předměty reálného světa, tak i zcela abstraktní, neexistující objekty. Proto je počítačová grafika velmi rozsáhlý obor. Jedním z odvětví počítačové grafiky je zobrazování (rendering). Jedná se o proces zpracování 2D nebo 3D modelu na základě dat definujících virtuální scénu a výsledkem tohoto zpracování je pak digitální obraz. Scéna je popsána pomocí objektů, jejich geometrie, barev a textur a pomocí dalších informací, jako je například osvětlení, stínování nebo pozice pozorovatele.

3 Grafická renderovací pipeline

Data, která definují scénu jsou počítačem zpracována ve výsledný obraz, který je zobrazen na monitor. Toto zpracování je provedeno v grafické kartě pomocí grafického vykreslovacího řetězce (graphics rendering pipeline).

3.1 Grafická pipeline OpenGL

Grafická pipeline, neboli vykreslovací řetězec, je série instrukcí, kterou zpracovává grafická karta a transformuje tak vstupní data do výstupního obrazu. [1]

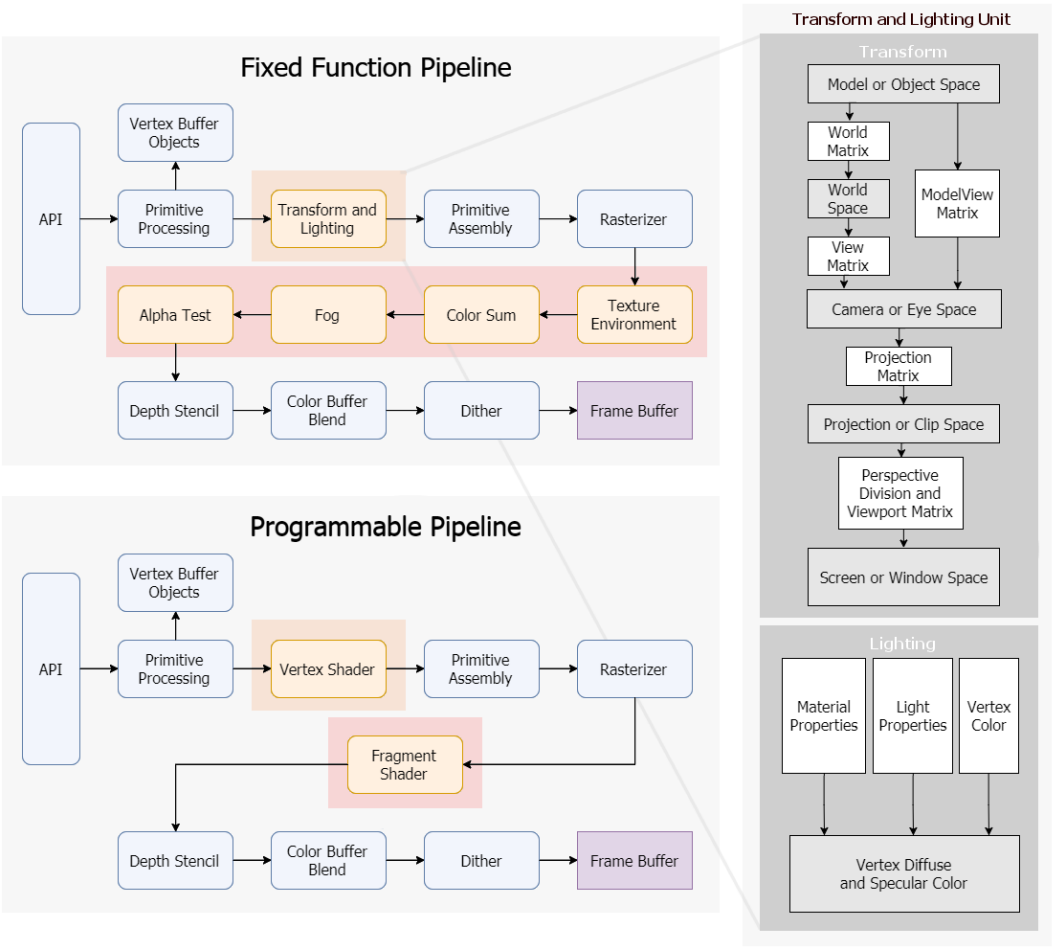
Instrukce grafické karty jsou hardwarově závislé a pro pohodlné a platformově nezávislé programování lze využít OpenGL API[1]. OpenGL je standard specifikující rozhraní pro tvorbu aplikací počítačové grafiky. Implementace OpenGL existují pro prakticky všechny počítačové platformy, na kterých je možné vykreslovat grafiku. Kromě implementací vestavěných na grafické kartě existují také softwarové implementace, které umožňují používat OpenGL i na hardwaru, který ho sám o sobě nepodporuje[4]. Verze OpenGL pak definují způsob zpracování dat včetně grafické pipeline.

Grafická pipeline OpenGL verze 1.x byla fixní, to znamená, že nebylo možné ovlivnit proces vykreslování přímo za běhu. Převážná funkcionalita, jako jsou vertexové transformace nebo způsoby stínování, byla definována v knihovnách OpenGL a programátor byl touto knihovnou značně omezený. Zásah do vykreslování nad rámec těchto knihoven byl značně komplikovaný. Navíc tak OpenGL aplikace pracovala s daty převážně nad CPU, což znamenalo nutnost přenosu dat před vykreslením do paměti GPU a to je při velkém množství dat neefektivní.

Rozvoj grafických karet a změna jejich architektury umožnila zavedení programovatelných jednotek a z původně fixní pipeline tak vznikla pipeline programovatelná. Programování těchto jednotek probíhá pomocí shaderů, programů, které jsou zapojované do grafické pipeline. Standard OpenGL tak byl od verze 2.x rozšířen o OpenGL Shading Language (GLSL), jazyk pro programování shaderů.

[1] Application Programming Interface, rozhraní pro programování aplikací

Obrázek 1 ilustruje rozdíly mezi fixní a programovatelnou pipeline. Značně rozsáhlá jednotka transform and lighting byla nahrazena vertex shaderem a jednotky texture environment, color sum, fog, alpha test splynuly do jedné jednotky fragment shaderu. Došlo tak ke značnému zjednodušení programování grafické pipeline a rozšíření jejích možností. Podrobný popis jednotlivých fází vykreslovací pipeline následuje v další sekci.



Obrázek 1: srovnání fixní a programovatelné pipeline

3.2 Popis fází grafické pipeline

Proces zpracování dat vykreslovacím řetězcem (grafickou pipeline) lze rozdělit do několika fází.

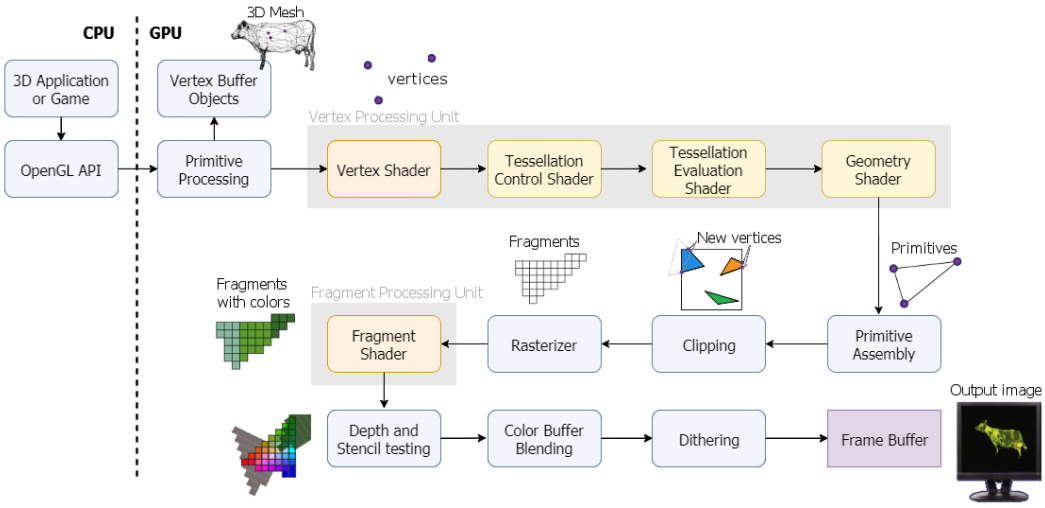
Vstupní data jsou uložena v paměti v tzv. bufferech. Buffery jsou úseky paměti spravované OpenGL a obsahují především vertexová[2] data.

[2]vertex je bod v prostoru, primitivum, nemá rozměr, obsahuje informace o souřadnicích, normále popř. barvě a další

První částí řetězce, která zpracovává vertexy, je vertexová jednotka. Je plně programovatelná a tvoří ji tři shadery - vertex, tesellation[3] (tesellation control a tesellation evaluation) a geometry. Všechny umožňují manipulaci s pozicemi vertexů.

Po průchodu vertexovou jednotkou jsou ze sekvencí jednotlivých vertexů sestavena geometrická primitiva (body, přímky, trojúhelníky a polygony). O toto zpracování se stará jednotka primitive assembly. Pokud zobrazovaná primitiva leží mimo viewport[4], jsou v další fázi zvané clipping odstraněna. Zároveň také dojde k odstranění primitiv, která nejsou viditelná z důvodu opačné orientace v prostoru (normála směřuje „od“ oka) (culling) a primitiv, která jsou ve scéně umístěna příliš daleko od oka (z-clipping).

Takto zpracovaná primitiva jsou poslána do rasterizační jednotky. Tam jsou primitiva promítnuta do vhodné diskrétní rasterizační mřížky a dojde k vytvoření fragmentů. Nejedná se však o pixel a je vhodné pojmy fragment a pixel rozlišovat. Fragmentu je při průchodu fragmentovou jednotkou (fragment shaderem) přiřazena barva a je tak definovaný pomocí barevných kanálu r, g, b, a a z-souřadnice. Po průchodu fragmentu fragmentovou jednotkou dochází ke zpracování fragmentů pomocí depth a stencil testing, color a alpha blending. Takto zpracované fragmenty jsou uloženy do framebufferu a zobrazeny na monitor. V této chvíli se již jedná o pixely.



Obrázek 2: OpenGL grafická pipeline

4 Shadery

Shadery jsou programy zapojované do grafické renderovací pipeline a rozdělují se podle toho, do které části (neboli jednotky) renderovací pipeline jsou zapojeny

[3]teselace je proces zjemňování trojúhelníkové sítě tvořící model objektu

[4]polygon ohraničující oblast scény, která bude vykreslena

a podle toho nad jakými daty operují.

Prvními shadery, které vznikly, byly vertex a fragment shadery. Jejich zapojení do grafické pipeline je nyní povinné. V dalších letech pak přišly další, nepovinné shadery. Byly to tesellation, geometry a computation shadery.

4.1 Výhody shaderů

OpenGL API poskytuje pohodlné programování nad grafickou kartou bez ohledu na její hardwarovou architekturu. Síla grafické karty spočívá v jejím paralelním způsobu pracování dat. Navíc je přizpůsobena výpočetně náročným matematickým výpočtům jako je například počítání s vektory a maticemi. Je tedy výhodné převést co nejvíce operací pracujících nad daty přímo do paměti grafické karty.

Vykreslování funguje v renderovací smyčce „process - update - render“[5][7]. Pokud je nad daty operováno v OpenGL aplikaci a tedy v CPU paměti, musí být každý vertex zvlášť transformován, převeden z CPU do GPU a poté vykreslen. Pokud je stejná transformace prováděna v GPU, probíhají transformace jednotlivých vertexů paralelně, tedy současně a přímo v GPU paměti, není tedy potřeba data kamkoliv přenášet. Navíc je většina transformací přímo zabudována v GPU jako již dříve zmíněné maticové a vektorové operace.

Zajímavostí je, že díky paralelní architektuře GPU operuje shader právě nad jedním vertexem/fragmentem v čase a proto o sobě jako celek navzájem nevědí. Je tedy nemožné zjistit pozici či barvu konkrétních sousedních vertexů/fragmentů.

V následující části je stručně představenou všech pět druhů shaderů avšak v implementacích se bakalářská práce dále věnuje už jen vertex a fragment shaderům.

4.2 Vertex Shader

Vertex shader je první částí programovatelné pipeline. Umožňuje manipulaci s jednotlivými existujícími vertexy, v každém okamžiku právě nad jedním. Není možné přidávat nové vertexy a nebo existující odebírat. Vertex je bod v 3D prostoru, nemá rozměr, ale nese informace o pozici, barvě, souřadnicích v textuře a normále. Vstupem i výstupem vertex shaderu je jeden vertex.

Častými operacemi prováděnými ve vertex shaderu jsou maticové transformace realizující souřadnicové transformace vertexů:

- **Modelovací transformace** umístí a natočí objekty ve scéně. Jedná se o převedení objektu z model space[6] do world space[7].
- **Zobrazovací transformace** objekt zpracují vzhledem k poloze a natočení kamery. Jde o převedení objektu ve world space do camera space[8].

[5] načtení dat do bufferů - aktualizace dat v bufferech - zobrazení
[6] vertexy definovány vzhledem ke středu modelu
[7] vertexy definovány vzhledem ke středu světa
[8] vertexy definovány vzhledem ke kameře

- **Projekční transformace** nastaví pohled na objekt kvádrem nebo komolým jehlanem převedením z camera space do clip space.

```
1 // SimpleVertexShader.vertexshader
2 // nastaví do proměnné gl_Position souřadnice vertexu v model space
3 #version 330 core
4 layout(location = 0) in vec3 vertexPosition_modelspace;
5
6 void main() {
7   gl_Position = vec4(vertexPosition_modelspace.xyz, 1.0);
8 }
```

Zdrojový kód 1: ukázka vertex shaderu

4.3 Tessellation Shader

Tessellation shader je volitelnou fází pipeline, umožňuje přidávání nových vertexů. Teselace rozdělí jednoduchý geometrický útvar do menších částí podle zadaných parametrů (velikost, zakřivení) a tím zvětší detaily vykreslovaného objektu, který se bude skládat z více vertexů. Vytváření nových vertexů až v grafické pipeline je velmi výhodné, protože manipulace s vertexy před vstupem do GPU jsou náročné.

Proces teselace lze řídit dvěma shadery:

- **Tessellation control shader** transformuje vstupní data (mřížku, patch) podle zadaných parametrů (například velikost, zakřivení) a přepočítá atributy vertexů (pozice, normály).

Tessellation primitive generator je fixní jednotkou mezi dvěma teselačními shadery a jeho úkolem je rozdělení mřížky, která je výstupem tessellation control shaderu, na jemnější mřížku, která je pak vstupem do Tessellation evaluation shaderu.

- **Tessellation evaluation shader** je zavolán na každý vertex výstupní mžížky z jednotky tessellation primitive generator a provádí interpolaci jejich atributů.

4.4 Geometry Shader

Geometry shader je také volitelný a je poslední částí pipeline, která se věnuje geometrii objektů před rasterizací. Operuje nad primitivy, jeho vstupem i výstupem jsou primitiva (bod, úsečka, trojúhelník). Tím umožňuje vertexy jak přidávat tak i odebírat, vznikají nové tvary, ať už jednodušší či složitější, případně může původní tvar zcela zaniknout. Není však optimální pro generování složité geometrie, k tomu je vhodnější předchozí fáze teselace.

4.5 Fragment Shader

Fragment shader je poslední částí programovatelné pipeline a je povinný. Jeho vstupem je pozice jednoho fragmentu a výstupem je jeho barva. Fragment vznikl rasterizací, promítnutím vertexů na rastrovou mřížku a proto má, narozdíl od vertexu, rozměr. Ještě se ale nejedná o pixel, protože je potřeba provést depth a stencil testing, alpha a color blending.

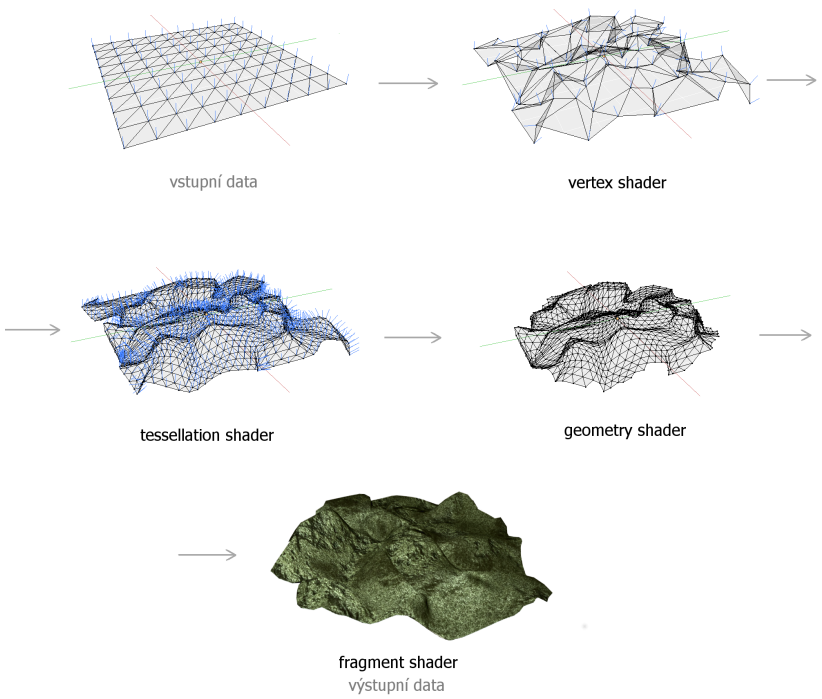
```
1 // přiřadí fragmentu červenou barvu
2 in vec2 position;
3 out vec4 color;
4
5 void main()
6 {
7   red = vec3(1.0, 0.0, 0.0);
8   color = vec4(red, 1.0);
9 }
```

Zdrojový kód 2: ukázka fragment shaderu

4.6 Příklad

Příkladem demonstrujícím využití jednotlivých shaderů je například modelování terénu. Do vertex shaderu vstupují vertexy umístěné v rovině. Aplikací vhodné náhodné funkce lze vertexy přesunout ve směru osy z. Vznikne tak náhodně zakřivená rovina. Tessellation shader pak na základě vhodných parametrů terén vyhladí a zjemní. Dostává tak tvar odpovídající skutečnému terénu. Geometry shaderem pak lze odstranit nepotřebné vertexy. V poslední fázi, ve fragment shaderu, jsou na fragmenty naneseny textury či barvy.

Proces zpracování dat tohoto příkladu jednotlivými fázemi ilustruje obrázek 3.



Obrázek 3: zpracování primitiv jednotlivými shadery

5 GLSL

OpenGL Shading Language neboli GLSL je vyšší programovací jazyk pro psaní shaderů pro OpenGL API. Syntaxí je velmi blízký jazyku C. Navíc poskytuje nové datové typy a funkce zjednodušující práci nad vertexy či fragmenty v GPU.

V této sekci je popsána verze GLSL vycházející z OpenGL verze 4.3.

5.1 Datové typy a operátory

Datové typy jsou shodné s datovými typy jazyka C. Navíc je zde typ `sampler` pro reprezentaci textur.

typ	popis
`void`	pro funkce bez návratové hodnoty
`bool`	pravdivostní datový typ, hodnoty true/false
`int`	znaménkové celé číslo, 32 bitů včetně znaménkového bitu
`uint`	neznaménkové celé číslo, 32 bitů
`float`	desetinné číslo s pohyblivou řadovou čárkou
`double`	desetinné číslo s pohyblivou řadovou čárkou s dvojitou přesností
`sampler`	textura

Příkladem použití vektorového datového typu je například uložení pozice vertexu `vec3 position;` nebo barvy fragmentu `vec4 color = vec4(1.0, 0.0,`

`0.0, 1.0);`. Barvy fragmentů jsou uloženy ve formátu RGBA s hodnotami v rozmezí $0 - 1$.

5.1.1 Vektorové typy

Pro datové typy `bool`, `int`, `uint`, `float` a `double` existují vektorové datové typy. Jsou tvořeny dvěma, třemi nebo čtyřmi složkami daného typu a značně usnadňující práci a výpočetní operace.

základní datový typ	2D	3D	4D
`float`	`vec2`	`vec3`	`vec4`
`double`	`dvec2`	`dvec3`	`dvec4`
`int`	`ivec2`	`ivec3`	`ivec4`
`uint`	`uvec2`	`uvec3`	`uvec4`
`bool`	`bvec2`	`bvec3`	`bvec4`

Konstruktory

Hodnoty jednotlivých složek vektoru lze inicializovat výčtem hodnot

```
vec4 color;
vec3 cyan = vec3(0.0, 1.0, 1.0);
```

nebo zkráceným zápisem, jsou-li hodnoty všech složek shodné

```
vec3 white = vec3(1.0);
```

a nebo pomocí vektorů

```
color = (cyan, 0.5);
```

a v proměnné `color` je uložena modrozelená barva s průhledností 0.5.

5.1.2 Matice

Matice jsou dalším datovým typem o více složkách, které mohou být typu `float` nebo `double`. Jedná se o sloupcové matice.

základní datový typ	typy matic
`float`	`mat2`, `mat2x2`, `mat3x2`, `mat4x2`
	`mat3`, `mat2x3`, `mat3x3`, `mat4x3`
	`mat4`, `mat2x4`, `mat3x4`, `mat4x4`
`double`	`dmat2`, `dmat2x2`, `dmat3x2`, `dmat4x2`
	`dmat3`, `dmat2x3`, `dmat3x3`, `dmat4x3`
	`dmat4`, `dmat2x4`, `dmat3x4`, `dmat4x4`

Matice lze inicializovat výčtem jednotlivých složek

```
mat2 M1 = mat2(0.5, 1.0, 0.25, 0.7);
```

nebo pomocí vektorů

```
vec2 col1 = vec2(0.5, 1.0);
vec2 col2 = vec2(0.25, 0.7);
```

```
mat2 M2 = mat2(col1, col2);
```
a v obou případech je výsledkem stejná matice

$$M1 = M2 = \begin{bmatrix} 0.5 & 0.25 \\ 1.0 & 0.7 \end{bmatrix}$$

Swizzling

Swizzling je mocným nástrojem jazyka GLSL vhodným pro práci s vektorovými a maticovými typy. K jednotlivým složkám lze přistupovat pomocí tečkové notace nebo pomocí indexů. Přístup pomocí indexů odpovídá přístupu k jednotlivým pvkům pole v jazyce C.

pojmenované složky	**popis**
(x, y, z, w)	složky spojené s pozicemi
(r, g, b, a)	složky spojené s barvami
(s, t, p, q)	složky spojené se souřadnicemi textur

```
vec4 color = texture2D(Texture, textureUV).rgba;
uv1.xyz = uv2.zyx //prohozenísouřadnic
```

5.1.3 Transparentní datové typy

Transparentní datový typy jsou ve skutečnosti handlery na nějaký objekt, například samplery jsou handlery pro přístup k texturám.

základní datový typ	**1D**	**2D**	**3D**
	`sampler1D`	`sampler2D`	`sampler3D`
`int`	`isampler1D`	`isampler2D`	`isampler3D`
`uint`	`usampler1D`	`usampler2D`	`usampler3D`

5.1.4 Struktury a pole

Deklarace struktur a přístup k jejich hodnotám je stejná jako v jazyce C.

Rozdíl oproti jazyku C je u polí. Ta mohou být pouze jednorozměrná a statická.[9] Navíc lze délku pole snadno zjistit pomocí funkce `length()`:

```
uniform float pole[12];
pole.length(); //vracíl2
```

[9] Vyjímku tvoří vstupní pole geometry a tessellation shaderů, které nemusí mít předem stanovenou velikost.

5.2 Proměnné a kvalifikátory

typ	popis
`const`	proměnná, jejíž hodnota nemůže být změněna
`in`	vstupní proměnná shaderu nebo funkce
`out`	výstupní proměnná shaderu nebo funkce
`inout`	vstupně-výstupní proměnná shaderu nebo funkce
`location`	odkazuje na umístění proměnné
`uniform`	proměnná, jejíž hodnota je neměnná v celé fázi renderování, tedy její hodnota je stejná pro každý vertex či fragment; slouží pouze pro čtení ve vertex a fragment shaderech [10]
`attribute`	proměnná, jejíž hodnota se může lišit mezi jednotlivými vertexy předávanými z OpenGL do vertex shaderu; slouží pouze ke čtení ve vertex shaderech
`varying`	proměnná, jejíž hodnota se mění mezi vertex a fragment shadery; ve vertex shaderu je definována pro každý vertex a při rasterizaci je její hodnota vlivem interpolace vertexů na fragmenty změněna; do proměnné lze ve vertex shaderu zapisovat, ve fragmentu shaderu z ní lze pouze číst

Vestavěné proměnné mají předponu `gl_` a slouží pro sdílení hodnot mezi OpenGL aplikací a shadery a nebo mezi shadery samotnými. Mohou to být různé konstanty a nebo vstupní a výstupní proměnné shaderů.

```
int gl_MaxViewports; //konstanta
in int gl_VertexID; //vstupníproměnnávertex shaderu
out float gl_FragDepth; //výstupníproměnnáfragment shaderu
```

5.3 Operátory

V GLSL jsou k dispozici stejné matematické a logické operátory jako v jazyce C, navíc jsou přizpůsobené počítání s vektory a maticemi. Dale je tu k dispozici již dříve zmíněný operátor pro swizzling.

5.4 Podmínky a cykly

V GLSL lze používat podmínky a cykly řídící běh programu stejně jako v jazyce C.

iterace	`for`, `while`, `do-while`
podmínky	`if-else`, `switch-case`
ukončení	`return`, `discard`

[10] např. proměnná uniform float Frame; udávající pořadí snímku od počátku pro synchronizaci času v celém procesu renderování

5.5 Funkce

Také deklarace funkcí je shodná s deklarací funkcí jazyka C. Navíc GLSL obsahuje velké množství vestavěných funkcí představených v následující podsekci.

5.5.1 Vestavěné funkce

GLSL obsahuje širokou škálu vestavěných funkcí umožňující snadnou a rychlou práci s nejrůznějšími transformacemi a efekty. Následuje krátký přehled nejpoužívanějších funkcí.[11]

Trigonometrické

funkce	popis
T `sin`(T *angle*)	standardní funkce sinus
T `cos`(T *angle*)	standardní funkce kosinus
T `tan`(T *angle*)	standardní funkce tangens
T `asin`(T x)	funkce arkus sinus; vrací úhel, jehož sinus je x
T `acos`(T x)	funkce arkus kosinus; vrací úhel, jehož kosinus je x
T `atan`(T x, y)	funkce arkus tangens; vrací úhel jehož tangens je y/x
T `atan`(T y^x)	arkus tangens; vrací úhel, jehož tangens je y^x

Exponenciální

funkce	popis
T `pow`(T x, *typ* y)	vrací hodnotu x umocněno na y, tedy x^y
T `exp`(T x)	přirozená exponenciální funkce, výsledkem je hodnota e^x
T `log`(T x)	přirozený logaritmus, výsledkem je hodnota y splňující rovnici $x = e^y$
T `sqrt`(T x)	odmocnina, výsledkem je hodnota $\sqrt{x}$

Geometrické

funkce	popis
T `length`(T x)	vrací Euklidovskou délku vektoru
T `distance`(T *p0*, T *p1*)	vrací vzdálenost mezi body p0 a p1, tedy délku vektoru d = p0 - p1
T `dot`(T x, T y)	skalární součin vektorů x a y
T `normalize`(T x)	vrací normálový vektor, tedy vektor stejného směru ale délky 1

[11] T je zkratkou za některý datový typ z následujících - `float, vec2, vec3, vec4`

Operace s maticemi

funkce	popis
mat[12] `matrixCompMult`(*mat x, mat y*)	vrací výsledek vynásobení matice x s maticí y po složkách, tedy $z[i][j] = x[i][j] \times y[i][j]$
mat `transpose`(*mat m*)	vrací matici transponovanou k matici m
float `determinant`(*mat m*)	vrací determinant m
mat `inverse`(*mat m*)	vrací matici inverzní k matici m

Práce s texturami

funkce	popis
vec4 `texture2D`(*samplerT sampler*, vec2 *coord*)	vrací texel, tedy barvu textury na pozici *coord*

Další

funkce	popis
typ `abs`(T x)	vrací absolutní hodnotu x, tedy x pokud $x >= 0$, jinak -x
T `floor`(T x)	vrací hodnotu zaokrouhlení x na celé číslo, které je menší nebo rovno x
T `ceil`(T x)	vrací hodnotu zaokrouhlení x na celé číslo, které je větší nebo rovno x
T `fract`(T x)	vrací $x -$ `floor`(x)
T `mod`(T x, T y)	operace modulo; vrací $x - y \times$ `floor`$(\frac{x}{y})$
T `min`(T x, T y)	vrací y, pokud $y < x$, jinak vrací x
T `max`(T x, T y)	vrací y, pokud $x < y$, jinak vrací x
T `clamp`(T x, T y, T $minV$, T $maxV$)	vrací `min`$(max(x, minV), maxV)$, výsledek je nedefinovaný pokud $minV > maxV$
T `mix`(T x, T y, T a)	vrací lineární přechod x a y; tedy $x * (1 - a) + y * a$
T `step`(T *edge*, T x)	vrací 0.0 pokud $x <$ *edge*, jinak 1.0
T `smoothstep`(T *edge0*, T *edge1*, T x)	vrací 0.0 pokud $x <=$ *edge0*, 1.0 pokud $x >=$ *edge1* jinak výsledek Hermitovy interpolace[13] mezi 0 a 1 pokud *edge0* $< x <$ *edge1*

[12] *mat* představuje datový typ matice

[13] Hermitova interpolace je ekvivalentní s

6 Zapojení shaderů v OpenGL

Tato sekce nejdříve popisuje přípravu dat OpenGL aplikace pro jejich zpracování pomocí shaderů a potom představuje způsoby předávání hodnot mezi samotnými shadery.

6.1 Příprava dat a nastavení shaderů

Tato část popisuje přípravu dat v OpenGL aplikaci podle OpenGL knihovny (nejedná se o jazyk GLSL).

Nejprve je potřeba data určená ke zpracovávání načíst z CPU do GPU bufferů. To probíhá v OpenGL aplikaci pomocí funkce `glBufferData()`, která alokuje buffer objekt a uloží do něj příslušná data. Současně tato funkce slouží i pro přístup k datům.

Po inicializaci bufferů je zavolána funkce `glDrawArrays()`, která spustí proces renderování, tedy pošle vertexová data z bufferů do renderovací pipeline.

Dále je v OpenGL aplikaci potřeba vytvořit shader objekt a připojit k němu příslušný zdrojový kód shaderu. Funkce `glCreateShader()` alokuje shader objekt. Je to funkce jednoho argumentu, který určuje o jaký typ shaderu se jedná. Následně funkce `glShaderSource()` spojí zdrojový kód shaderu s alokovaným shader objektem. Objekt je spojen se shaderem a je potřeba jej zkompilovat pomocí funkce `glCompileShader()`.

Nyní je potřeba vytvořit spustitelný program. Po tom, co jsou vytvořeny a zkompilovány všechny jednotlivé shader objekty, je pomocí `glCreateProgram()` a `glAttachShader()` vytvořen výsledný shader program a k němu jsou připojeny všechny předchozí shader objekty. `glLinkProgram()` zpracuje shader objekty v jeden spustitelný soubor a ten je pomocí `glUseProgram()` nastaven jako aktuální pro vykreslování. Tento spustitelný program definuje veškeré zpracování dat vykreslovacím procesu pomocí shaderů.

6.2 Předávání hodnot mezi shadery

Následující část již opouští programování v OpenGL a zabývá se programováním v GLSL.

6.2.1 Pomocí proměnných

Základním způsobem předáváním hodnot mezi shadery je pomocí proměnných stejných názvů. Hodnota proměnné out jednoho shaderu je čtena ve vstupní proměnné druhého shaderu, pokud se shodují její názvy. Například pro předání barvy vertexu z vertex shaderu do fragment shaderu lze provést následovně:

```
T t = clamp ((x - edge0) / (edge1 - edge0), 0, 1);
 return t * t * (3 - 2 * t);
```

```
1  // vertex shader
2  out vec4 color;
3  // fragment shader
4  in vec4 color;
```

Zdrojový kód 3: sdílení proměnných pomocí názvu

6.2.2 Pomocí lokace

Název výstupní proměnné jednoho shaderu, který je vstupní proměnnou navazujícího shaderu musí být shodný, avšak nesmí být shodný mezi dalšími dvěma či více shadery. Například pokud by bylo požadováno dodatečné přidání geometry shaderu mezi vertex a fragment shadery, je nutné proměnné zpětně přejmenovat a to je zbytečná komplikace. Řešením je použití jiného způsobu předávání hodnot a to pomocí kvalifikátoru `location`. Ten odkazuje na uložení proměnné v paměti a zároveň není závislý na názvu proměnných a tak nemusí být shodné.

```
1  // vertex shader
2  layout (location = 0) out vec3 normalOut;
3  // fragment shader
4  layout (location = 0) in vec3 normalIn;
```

Zdrojový kód 4: sdílení proměnných pomocí kvalifikátoru layout

Přidání nových proměnných do geometry shaderu pomocí kvalifikátoru `location` nemá vliv na dříve vytvořené vertex a fragment shadery:

```
1  // vertex shader
2  layout (location = 0) out vec3 normalOut;
3  // geometry shader
4  layout (location = 0) in vec3 normalIn[];
5  // fragment shader
6  layout (location = 0) in vec3 normalIn;
```

Zdrojový kód 5: sdílení proměnných pomocí kvalifikátoru layout

Předávání hodnot tímto způsobem však může být nebezpečné a je potřeba dbát na to, jaký prostor v paměti zabírají konkrétní datové typy používaných proměnných.

6.2.3 Pomocí vestavěných proměnných

Jedná se o proměnné s předponou `gl_`, například:

```
// vstupní proměnné vertex shaderu
in int gl_VertexID;
in int gl_InstanceID;
// výstupní proměnné vertex shaderu
out gl_PerVertex {
vec4 gl_Position;
float gl_PointSize;
float gl_ClipDistance[];
};
// vstupní fragment shaderu
in vec4 gl_FragCoord;
in bool gl_FrontFacing;
in vec2 gl_PointCoord;
// výstupní fragment shaderu
out float gl_FragDepth;
```

Zdrojový kód 6: vestavěné proměnné sdílené mezi OpenGL a shadery

7 Stream Circle

7.1 O službě Stream Circle

Grafické efekty v této bakalářské práci byly programovány pro využití v editoru scén služby Stream Circle. Je to služba pro profesionální internetové televizní vysílání. Umožňuje nahrávání souborů, plánování vysílání, náhledy vysílání a editování televizní grafiky. Tu lze skládat z jednotlivých panelů, jako je například panel loga, textu, hodin nebo dalších geometrických prvků (koule, zakřivené plochy apod.). Obraz vysílání pak lze skrze shaderů upravovat pomocí filtrů a dalších efektů.

7.2 Editor scén pro Stream Circle

PBA editor

Scény v grafickém editoru PBA jsou tvořeny OpenGL panely nebo dalšími geometrickými primitivy. Na ně je pak aplikován shader s konkrétním efektem. Vertex shadery jsou převážně použity pouze pro základní operace (transformace, výpočet normál vertexů ve 3D prostoru), samotné efekty jsou pak naprogramovány především ve fragment shaderech.

8 Příklady efektů

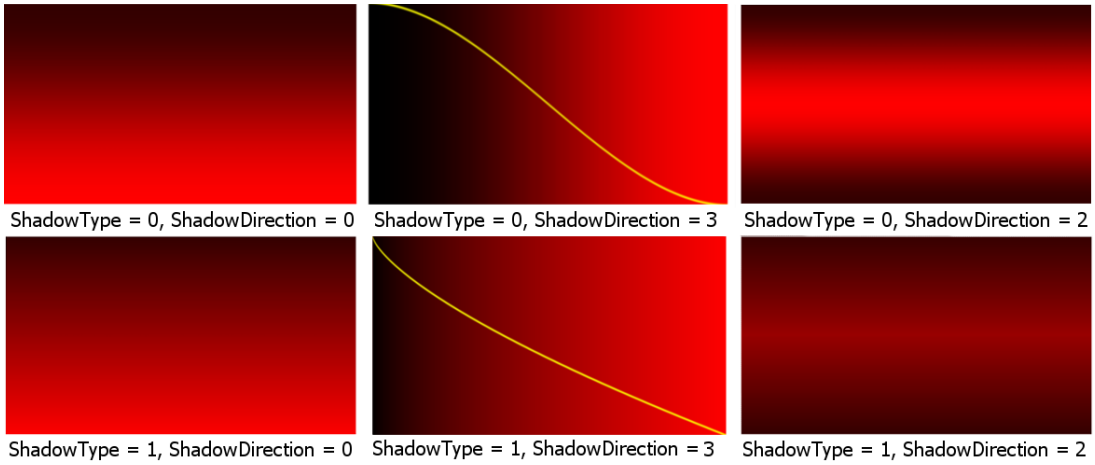
V následující části bakalářské práce jsou předvedeny a popsány grafické efekty, které byly programovány pro použití v grafickém editoru scén služby Stream Circle. Efekty mohou být aplikovány přímo na obraz živého vysílání nebo na jednoduché panely, jež lze kombinovat a doplnit tak scénu vysílaného obrazu o mnoho efektů jako například o dynamické pozadí, doplňující prvky jako jsou např. ručičkové hodiny a další.

8.1 2D shadery

8.1.1 Stínované pozadí

Stínované pozadí je ukázkou základního statického shaderu. Vykresluje přechod černé barvy a barvy zadané uživatelem v proměnné `MaterialAmbient`. Lze nastavit typ stínovaného přechodu `ShadowType`, směr přechodu `ShadowDirection`, rozmezí přechodu `ShadowRange` a intenzitu přechodu `ShadowIntensity`.

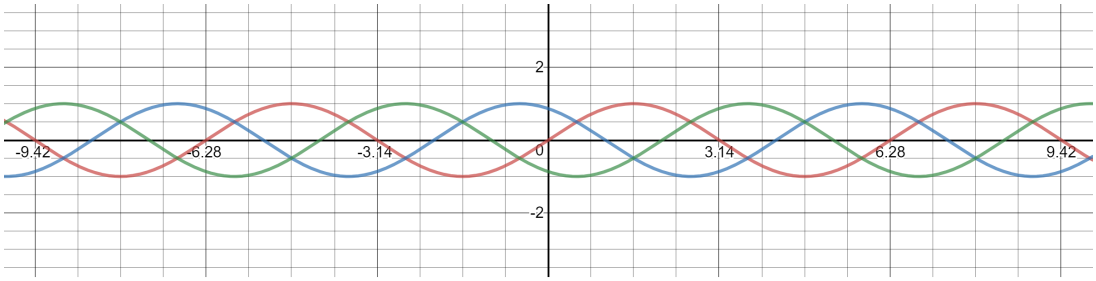
Typy přechodu jsou dva. První, `ShadowType = 0`, realizuje barevný přechod Hermitovou interpolací pomocí vestavěné funkce `smoothstep`. Druhý, `ShadowType = 1`, vykresluje přechod exponenciální funkcí `pow`. Srovnání vykreslení přechodu pomocí těchto funkcí znázorňuje obrázek 4 .



Obrázek 4: srovnání přechodu pomocí interpolace smoothstep (nahoře) a exponenciální funkce (dole)

8.1.2 Barevné pulzování

Jednoduchý dynamický shader, který je ukázkou použití proměnné `time`. Pozadí je vykreslené stejnou barvou která se postupně mění. Pozadí tak plynule a harmonicky přechází z jedné barvy do druhé. Toho je dosaženo rovnoměrnou změnou všech tří barevných kanálů pomocí funkce sinus. Základní proměnnou této funkce je proměnná `time`. Proměnná `time` je spočítána pomocí dvou proměnných typu `uniform` - `Frame`, která udává frame rate[14] GPU a `FrameToTime`, která převádí frame rate na čas. Aby byly barevné přechody co nejplynulejší a pokrývaly co nejširší škálu barev, je důležité rovnoměrné rozmístění hodnot jednotlivých kanálů. To je zajištěno pomocí funkce sinus se třemi různými argumenty - `sin(time)`, `sin(time+2*PI/3)` a `sin(time+4*PI/3)`




```
vec3 fragColor = vec3(sin(time), sin(time+(2*PI/3)), sin(time+(4*PI/3)));
```

Obrázek 5: rovnoměrné přiřazení hodnot jednotlivým kanálům pomocí tří funkcí

Zdrojový kód tohoto shaderu lze nalézt v příloze A Zdrojové kódy vybraných shaderů.

8.1.3 Pohyblivé bodové světlo

Jedná se o vykreslení bodového světla, které projíždí obrazovkou zleva doprava či obráceně. Při průchodu obrazovkou, světlo s klesající vzdáleností ke středu obrazovky zvyšuje svůj jas a naopak s rostoucí vzdáleností od středu obrazovky svůj jas snižuje až do ztracena. To vytváří dojem, jako by se světlo přibližovalo a poté vzdalovalo.

Výpočet bodového světla je inspirován z jednoduchého poměru mezi intenzitou záření a druhou mocninou vzdálenosti fragmentu od středu záření. Tento předpis se používá při výpočtu ztlumení bodového světla (point light attenuation).

$$light = \frac{lightPower}{d^2}$$

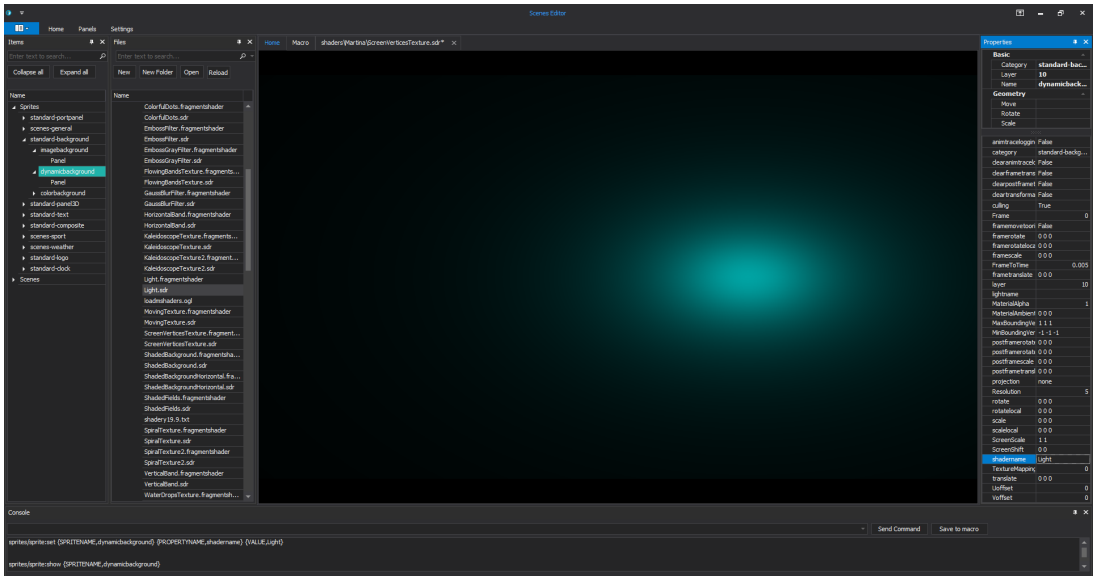
Pro efekt postupného rozjasňování a tlumení světla je ve výsledném výpočtu navíc zohledněna vzdálenost fragmentu od středu obrazovky. Pro větší rozptyl

[14] pořadí snímku od počátku renderování

světla od středu záření a snížení ostrého přechodu samotného zdroje světla od jeho záření je navíc ke jmenovateli přičtena intenzita záření $lightPower$ vynásobená konstantou $0,5$, která byla zvolena jako vizuálně nejvhodnější parametr pro výsledný efekt.

$$light = \begin{cases} lightCenter * \frac{lightPower}{d^2 + 0.5 * lightPower} & \text{pro } lightCenter <= 0.5 \\ (1 - lightCenter) * \frac{lightPower}{d^2 + 0.5 * lightPower} & \text{pro } lightCenter > 0.5 \\ lightCenter * lightPower & \text{pro } x = 0.5 \end{cases}$$

Výsledný předpis ilustruje obrázek 6.



Obrázek 6: bodové světlo

Zdrojový kód tohoto shaderu lze nalézt v příloze A Zdrojové kódy vybraných shaderů.

8.1.4 Stínovaný pohyblivý pruh

Shader zobrazuje pruh, který je na okrajích vystínovaný do ztracena a probíhá obrazovkou horizontálním nebo vertikálním směrem. Aby bylo dosaženo co nejjemnějšího přechodu mezi pruhem a pozadím, bylo stínování řešeno pomocí funkce `smootherstep`, která je úpravou vestavěné fuknce `smoothstep` jazyka GLSL. Ta je definována pomocí Hermitovy interpolace.

Pohyb pruhu obrazovkou se opakuje, od počátečního okraje ke koncovému. Zajímavým problémem, který bylo u tohoto shaderu potřeba vyřešit, bylo vykreslení pruhu v koncovém okraji obrazovky a jeho návaznost ve vykreslení v počátečním okraji obrazovky. Požadovaná návaznost byla taková, aby se část pruhu, která je již za okrajem obrazovky, objevila na jejím začátku.

```
1   // smoothstep : Hermitova interpolace 3. řádu
2   float smoothstep(float edge0, float edge1, float x)
3   {
4     x = clamp((x - edge0)/(edge1 - edge0), 0.0, 1.0);
5     return x*x*(3 - 2*x);
6   }
7
8   // smootherstep : Hermitova interpolace 6. řádu
9   float smootherstep(float edge0, float edge1, float x)
10  {
11    x = clamp((x - edge0)/(edge1 - edge0), 0.0, 1.0);
12    return x*x*x*(x*(x*6 - 15) + 10);
13  }
```

Zdrojový kód 7: srovnání funkce smoothstep a smootherstep

Nastavitelnými parametry tohoto shaderu jsou barva a průhlednost pozadí, barva a průhlednost pruhu, rychlost a směr pohybu pruhu, šířka pruhu a počet pruhů.

Tento shader byl implementován jako doplněk pro oddělení textu od ostatních částí obrazu. Ukázka použití shaderu ve scéně s více panely zobrazuje obrázek 7.

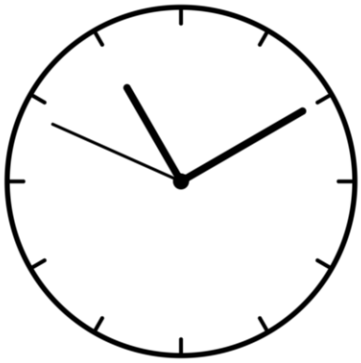
Obrázek 7: použití stínovaného pruhu

8.1.5 Ručičkové hodiny

Shader vykreslující ručičkové hodiny ukazující přesný čas.

Tento shader byl implementován ve webovém prostředí shadertoy.com[15], ve kterém je k dispozici proměnná `uniform vec4 iDate;`, ve formátu (rok, měsíc, den, čas v sekundách). Poslední složka je převedena na aktuální čas v hodinách, minutách a sekundách (`UTCHour`, `UTCMinute`, a `UTCSecond`) a tyto proměnné jsou použity pro pohyb odpovídajících hodinových ručiček.

Samotné hodiny jsou vykresleny pomocí jednoduchých geometrických tvarů, tedy pomocí kružnice a přímek. Shader umožňuje nastavovat barvu a průhlednost hodin `ClockAmbient, ClockAlpha`, barvu výplně hodin `ClockInAmbient` a barvu a průhlednost pozadí `MaterialAmbient, MaterialAlpha`.



Obrázek 8: ručičkové hodiny

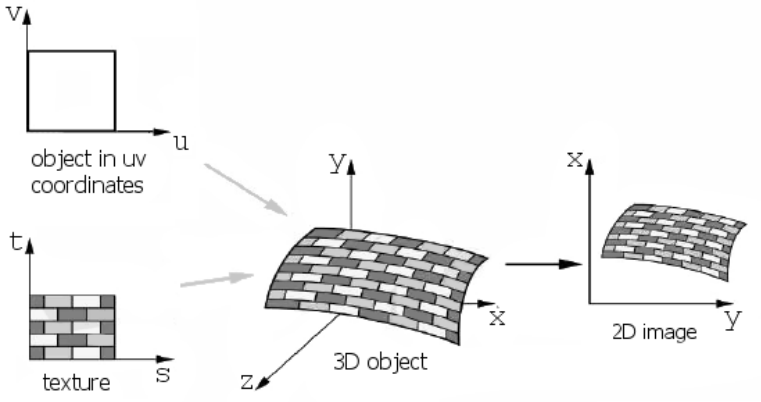
Zdrojový kód tohoto shaderu lze nalézt v příloze A Zdrojové kódy vybraných shaderů.

8.2 Textury

Textura je obrázek uložený v texture buffer. Textury se používají k mapování na povrch objektů za účelem přidání barev nebo průhlednosti. Mohou být mapovány na 2D plochu, jako je tomu v následujících příkladech bakalářské práce, a nebo na povrch 3D objektu pro dosažení realističtějšího nebo vizuálně zajímavějšího zobrazení objektu.

Mapování textury je proces nanesení textury (obrázku) na povrch 2D nebo 3D objektu. Každý objekt v prostoru má přidělené souřadnice (u, v). Ty odpovídají souřadnicím povrchu objektu rozloženém ve 2D prostoru. Textura má jednotlivé pixely (texely) umístěny na souřadnicích (t, s). Každý texel na souřadnicích (t, s) je pak nanesen na odpovídající souřadnice objektu (u, v) a tím dojde k namapování textury na povrch objektu. Tento proces zjednodušeně znázorňuje obrázek 9.

[15] více v sekci použité nástroje



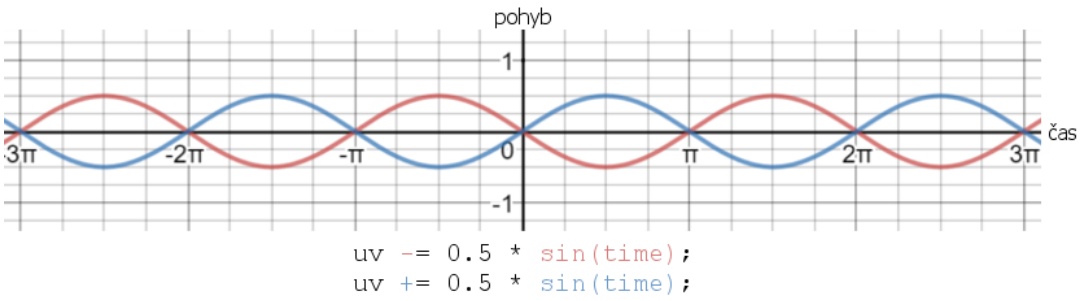
Obrázek 9: mapování textury

V GLSL je v proměnné `uniform sampler2D Texture;` uložený odkaz na texturu v paměti. Pomocí funkce `fragColor = texture2D(Texture, uv).rgba;` je do fragmentu na souřadnicích uv nanesen příslušný texel se složkami rgba.

8.2.1 Rozbití textury na pohyblivé pruhy

Tento shader je první ukázkou práce nad texturou vykreslenou na 2D panel. Jedná se o dynamický shader, který rovnoměrně rozdělí texturu na stejně velké pruhy, které se pak střídavě pohybují opačnými směry nahoru a dolů nebo doprava a doleva. Počet pruhů lze zadat v proměnné `BandsNumVertical` či `BandsNumHorizontal`.

Původní souřadnice obrazovky $< 0, 1 >$ jsou nejdříve rozšířeny buď ve směru osy x na rozmezí $< 0,$ `BandsNumVertical`$>$ a nebo ve směru osy y na rozmezí $< 0,$ `BandsNumVertical`$>$. Fragmenty jsou pak rozpohybovány v příslušném vertikálním nebo horizontálním směru. Fragmenty ležící v pruzích lichého pořadí jsou posunuty o `(-1)* sin(time)` a fragmenty v sudých pruzích o `sin(time)`. Takto je dosaženo střídavého pohybu v opačných směrech.



Obrázek 10: střídavý pohyb jednotlivých pruhů textury

Zdrojový kód tohoto shaderu lze nalézt v příloze A Zdrojové kódy vybraných shaderů.

8.2.2 Spirálová deformace textury

Jedná se o efekt deformace textury od středu k okrajům obrazovky spirálovitým pohybem. Textura se postupně od středu zakřivuje a v určitém čase se deformace zastaví a směrem od okrajů se textura zase vrací zpět do původní podoby.

Spirál, jakožto křivek, je definováno velké množství a pro vizuálně nejvhodnější efekt byla vybrána logaritmická spirála. Její poloměr roste exponenciálně se vzdáleností od jejího středu a to vytváří efekt postupného slábnutí spirálovité deformace. Její rovnice má v polárních souřadnicích tvar[16]

$$r = a \times e^{b \times \phi}$$

kde a, b jsou konstanty r poloměr a ϕ úhel mezi spojnicí fragmentu a počátkem. Po zlogaritmování vypadá předpis následovně

$$ln\frac{r}{a} = b \times \phi$$

a v shaderu je použit ve tvaru

$$\phi = ln\frac{r}{a} \times 1/b.$$

Počátek souřadnic je nejdříve z původního levého horního okraje posunut do středu obrazovky a souřadnice jsou z původních $< 0, 1 >$ posunuty do intervalu $< -1, 1 >$.

Protože je spirála definována pomocí poloměru a úhlu, je výhodnější počítat s polárními souřadnicemi, které nám přímo udávají vzdálenost fragmentu od počátku a úhel, který svírá spojnice fragmentu s počátkem (osa x).

Pro daný fragment je pak vypočten úhel, který odpovídá rotaci ve směru spirály. Úhel je součinem uživatelské proměnné `SpiralDensity`, která určuje hustotu spirály (hutnost deformace), sinusem času a logaritmem vzdálenosti bodu od počátku souřadnic. Podle tohoto vypočteného úhlu a rotační matice je spočítán odpovídající bod ležící ve spirále.

```
1 // převedení kartézských souřadnic fragmentu na polární
2 float radius = length(uv);
3 float angle = atan(uv.y, uv.x);
4
5 // úprava souřadnic fragmentu podle spirály
6 angle = SpiralDensity * sin(1,5 * time) * log(radius) * (1 -
      smoothstep(0, 1, radius));
7 uv = uv * mat2(cos(angle), -sin(angle), sin(angle), cos(angle));
```

Zdrojový kód 8: spirálová deformace textury

Textury jsou mapovány na objekty v souřadnicích $< 0, 1 >$ a tak je posledním krokem převedení souřadnic zpět z intervalu $< -1, 1 >$ do intervalu $< 0, 1 >$. Při ponechání souřadnic v intervalu $< -1, 1 >$ by textura byla namapována na 2D ploše celkem čtyřikrát.

Obrázek 11: deformace textury efektem spirály

Zdrojový kód tohoto shaderu lze nalézt v příloze A Zdrojové kódy vybraných shaderů.

8.2.3 Interpolace obrazu podle nastavení jeho vrcholů

Cílem bylo vytvoření shaderu, který na základě pozic vrcholů textury zadaných uživatelem interpoluje obraz tak, aby namapování textury odpovídalo zadaným vrcholům. To znamená, že dojde k deformaci textury. Původní souřadnice jednotlivých fragmentů jsou známé a je potřeba najít pro každý fragment nové souřadnice tak, aby fragment zapadal do nové, deformované textury. Nová pozice fragmentu R mezi vrcholy textury A, B, C, D je nalezena pomocí bilineární interpolace.

Lineární interpolace obrazu mezi dvěma body vypadá následovně

$$R.x : P = A + (B - A) \times uv.x$$
$$R.x : Q = D + (C - D) \times uv.x$$
$$R.y : R = P + (Q - P) \times uv.y$$

a z ní odvozená bilineární interpolace mezi čtyřmi body

$$R = A + (B - A) \times uv.x + (D - A) \times uv.y + (A - B + C - D) \times uv.x \times uv.y$$

a pojmenováním vektorů lze rovnici převést na následující tvar

$$H = E \times uv.x + F \times uv.y + G \times uv.x \times uv.y$$

kde

$$e = B - A$$
$$g = D - A$$
$$g = A - B + C - D$$
$$h = X - A$$

Protože e, f, g, h jsou vektory ve dvourozměrné dimenzi, rovnice se rozpadá na dvě a vznikají dvě rovnice o dvou neznámých. Jednotlivé složky pozice fragmentu uv lze vyjádřit jako

$$uv.x = \frac{(H.x - F.x \times uv.y)}{(E.x + G.x \times uv.y)} \text{ (a)}$$

$$uv.y = \frac{(H.y - F.y \times uv.x)}{(E.y + G.y \times uv.x)} \text{ (b)}$$

a po dosazení (a) za proměnnou $uv.x$ dostává rovnice následující tvar

$$H.y = E.y \times \frac{(H.x - F.x \times uv.y)}{(E.x + G.x \times uv.y)} + F.y \times uv.y + G \times \frac{(H.x - F.x \times uv.y)}{(E.x + G.x \times uv.y)} \times uv.y$$

a po rozepsání

$$0 = F.y \times G.x \times uv.y^2 - G.y \times F.x \times uv.y^2 + F.y \times E.x \times uv.y - E.y \times F.x \times uv.y + G.y \times H.x \times uv.y - H.y \times G.x \times uv.y + E.y \times H.x - H.y \times E.x$$

To je kvadratická rovnice s koeficienty

$$k2 = F.y \times G.x - G.y \times F.x$$
$$k1 = F.y \times E.x - E.y \times F.x + G.y \times H.x - H.y \times G.x$$
$$k0 = E.y \times H.x - H.y \times E.x$$

Speciální případ nastává pokud $k2 = 0$. Může se tak stát v případě, že je alespoň jedna ze souřadnic vrcholu A shodná s příslušnou souřadnicí vrcholu D a nebo pokud vrcholy textury tvoří rovnoběžník.

Rovnice pak dostává lineární tvar

$$0 = F.y \times E.x \times uv.y - E.y \times F.x \times uv.y + G.y \times H.x \times uv.y - H.y \times G.x \times uv.y + E.y \times H.x - H.y \times E.x$$

s kořenem rovnice

$$uv.y = \frac{-k0}{k1}$$

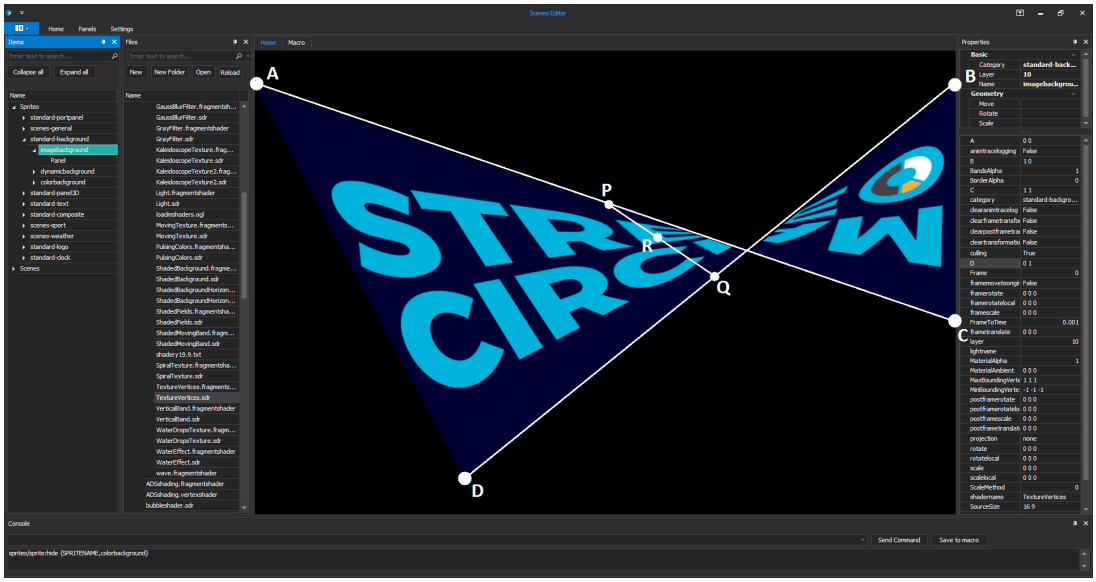
V ostatních případech je potřeba vyřešit původní kvadratickou rovnici s diskriminantem

$$D = k1 - 4 \times k0 \times k2$$

a kořenem rovnice

$$uv.y = \frac{-k1 \pm \sqrt{D}}{2 \times k2}$$

Pokud je diskriminant záporný, rovnice nemá řešení a znamená to, že nová pozice fragmentu leží mimo deformovanou texturu.



Obrázek 12: interpolovaná textura pro vrcholy $A = [0, 0], B = [1, 0.6], C = [1, 0], D = [0.3, 1]$

Zdrojový kód tohoto shaderu lze nalézt v příloze A Zdrojové kódy vybraných shaderů.

8.2.4 Šedotónový filtr

Filtr převádějící původní barevný obraz na šedotónový je jeden z implementačně nejjednodušších filtrů. K určení výsledné barvy fragmentu, respektive k odpovídajícímu stupni šedi, je potřeba znát pouze původní barvu tohoto fragmentu a nejsou potřebné žádné informace o okolí tohoto fragmentu.

Metod pro implementaci filtru převádějící obraz na šedotónový existuje několik.

- **Metoda zprůměrování barevných složek**
  Desaturace je základní metodou pro převod barevného fragmentu na fragment v odpovídající šedé barvě. Spočívá ve zprůměrování všech tří složek R, G, B a následně zpětným dosazením do R, G, B kanálů.

  $pixelcolor = (red + green + blue)/3$

  Metodu zprůměrováním barevných složek lze v shaderu nastavit parametrem `GrayscaleMethod = 1`.

```
1    // v proměnné texCol je uložena barva fragmentu
2    vec4 texCol = texture2D(Texture, textureUV).rgba;
3    // průměr z RGB složek daného fragmentu
4    float avgCol = (texCol.r + texCol.g + texCol.b) / 3.0;
5    Color = vec4(vec3(avgCol), texCol.a);
```

Zdrojový kód 9: výpočet šedé barvy fragmentu průměrováním RGB složek

- **Metoda luminance**
  Druhá metoda zohledňuje vnímání barev lidským okem. To vnímá intenzitu jednotlivých barevných složek odlišně, nejcitlivější je na zelenou, pak na červenou a nejméně vnímá barvu modrou. Následující výpočet proto zohledňuje vnímání barev lidským okem:

  $pixelcolor = 0.299 * red + 0.587 * green + 0.114 * blue$

  Tuto metodu lze v shaderu nastavit parametrem `GrayscaleMethod = 2`.

- **Metoda výběru podle barevného kanálu**
  Poslední metoda má spíše umělecké než praktické využití. Šedý odstín vrací podle jednoho zvoleného barevného kanálu a výsledná kvalita šedotónového obrazu tak závisí na konkrétní textuře.

  V shaderu lze převést obraz na šedotónový pomocí parametrů `GrayscaleMethod = 3` pro kanál červené barvy, `GrayscaleMethod = 4` pro kanál zelené barvy a `GrayscaleMethod = 5` pro kanál modré barvy.

[16]

Zdrojový kód tohoto shaderu lze nalézt v příloze A Zdrojové kódy vybraných shaderů.

[16]Dalšími metodami převedení barev obraz do odstínů šedi jsou například desaturace nebo

8.2.5 Filtr Gaussovského rozostření

Filtr Gaussovského rozostření se liší od předchozího filtru převádějící barevný obraz na šedotónový tím, že je potřebné znát okolí fragmentu. Programování tohoto filtru (a mnohých dalších) pak vychází ze stejné techniky. Nejdříve je potřeba vybrat k fragmentu vhodné diskrétní okolí a na něj je pak aplikována příslušná konvoluční matice filtru.

Definice Gaussovy funkce:

Hodnota v bodě x je podle Gaussovy funkce rovna

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

Koeficienty konvoluční matice pro Gaussovské rozostření jsou pro matice 3x3, 5x5 a 7x7:

$$M = \begin{bmatrix} \frac{2}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{1}{16} & \frac{4}{16} & \frac{2}{6} \\ \frac{1}{16} & \frac{2}{6} & \frac{1}{6} \end{bmatrix}$$

$$M = \begin{bmatrix} \frac{1}{273} & \frac{4}{273} & \frac{7}{273} & \frac{4}{273} & \frac{1}{273} \\ \frac{4}{273} & \frac{16}{273} & \frac{26}{273} & \frac{16}{273} & \frac{4}{273} \\ \frac{7}{273} & \frac{26}{273} & \frac{41}{273} & \frac{26}{273} & \frac{7}{273} \\ \frac{4}{273} & \frac{16}{273} & \frac{26}{273} & \frac{16}{273} & \frac{4}{273} \\ \frac{1}{273} & \frac{4}{273} & \frac{7}{273} & \frac{4}{273} & \frac{1}{273} \end{bmatrix}$$

$$M = \begin{bmatrix} \frac{1}{273} & \frac{4}{273} & \frac{7}{273} & \frac{4}{273} & \frac{1}{273} & \frac{4}{273} & \frac{1}{273} \\ \frac{4}{273} & \frac{16}{273} & \frac{26}{273} & \frac{16}{273} & \frac{4}{273} & \frac{4}{273} & \frac{1}{273} \\ \frac{7}{273} & \frac{26}{273} & \frac{41}{273} & \frac{26}{273} & \frac{7}{273} & \frac{4}{273} & \frac{1}{273} \\ \frac{4}{273} & \frac{16}{273} & \frac{26}{273} & \frac{16}{273} & \frac{4}{273} & \frac{4}{273} & \frac{1}{273} \\ \frac{1}{273} & \frac{4}{273} & \frac{7}{273} & \frac{4}{273} & \frac{1}{273} & \frac{4}{273} & \frac{1}{273} \\ \frac{4}{273} & \frac{16}{273} & \frac{26}{273} & \frac{16}{273} & \frac{4}{273} & \frac{4}{273} & \frac{1}{273} \\ \frac{1}{273} & \frac{4}{273} & \frac{7}{273} & \frac{4}{273} & \frac{1}{273} & \frac{4}{273} & \frac{1}{273} \end{bmatrix}$$

dekompozice pomocí maximální hodnoty barevného kanálu. Obě metody však pracují s hodnotami barev všech fragmentů najednou, což přesahuje rámec fragment shaderu, který má na vstupu a výstupu pouze jeden fragment.

S větší maticí lze dosáhnout efektu hlubšího rozostření. S tím však souvisí větší náročnost výpočtu a může docházet ke zpomalení vykreslování. Hlubšího rozmazání zde dosáhnout pomocí techniky techniky post-processing[17]. Nejdříve se řeší rozmazání v horizontálním a potom ve vertikálním směru (nebo obráceně). Výpočet rozmazání pouze v jednom směru je mnohonásobně rychlejší a jednodušší, počítá se pouze s vektory (nebo hodnotami uloženými v poli) a ne s celými maticemi.

Zdrojový kód tohoto shaderu lze nalézt v příloze A Zdrojové kódy vybraných shaderů.

8.2.6 Doplnění okrajů obrazu jiného formátu

Shader byl implementován za účelem vyplňování okrajů obrazovky při vkládání obrazového vstupu jiného formátu než je výchozí formát obrazovky. Příkladem je vložení příspěvku (obrázek, fotografie, video) formátu 4:3 do obrazu TV vysílání formátu 16:9. Jednobarevné vyplnění barvou, nejčastěji černou, může působit rušivě a proto je na místo černých ploch namapována část z okrajů obrazového vstupu, která je následně rozostřena. Okraje jsou tak doplněný barvami přímo z vloženého obrazu a to působí přirozeněji.

Na okraje jsou nejprve přeneseny odpovídající fragmenty a následně je na ně aplikován filtr Gaussova rozostření.

Obrázek 13: doplnění okrajů obrazu

[17]proces aplikování efektů po renderování, například vícenásobná aplikace shaderů

8.3 3D shadery

8.3.1 Stínování 3D objektů

Stínování je proces úpravy barvy a tmavosti objektu ve 3D scéně na základě pozice a vzdálenosti vzhledem objektu ke zdroji světla, jeho pozici vzhledem ke kameře, vlastnostem jeho materiálu jako jsou povrch, struktura, barva. Protože je stínování prováděno na základě vlastností objektů, které jsou vypočítány až během renderovacího procesu, jsou ke stínování využívány shadery.

Proces stínování povrchu objektu je ovlivněn třemi faktory - rozptylem materiálu (diffuse), vlivem prostředí (ambient) a zrcadlením materiálu (specular).

- **Diffuse složka**
  Světelný paprsek dopadne na povrch objektu a od něj je odražen do všech okolních stran. Intenzita tohoto odrazu závisí na úhlu dopadu světelného paprsku vzhledem k povrchu. To je spočítáno pomocí vektoru směru paprsku a normály povrchu.

```
1  float cosTheta = clamp(dot(n,l),0, 1);
2  color = LightColor * cosTheta;
3
4  // difRef : intenzita absorbce světla objektem
5  // 0 úplná absorbce, 1 žádná absorbce
6  diffuse = difRef * LightColor * cosTheta;
```

Zdrojový kód 10: diffuse

Barva odražených pavrchů pak závisí na barvě povrchu objektu

```
color = MaterialDiffuseColor * LightColor * cosTheta;
```

Dále je potřeba započítat vlastnosti světla. Existují tři základní zdroje světla, bodové (point), směrové (directional) a reflektor (spotlight). V této části se dále používá bodové světlo. To vyzařuje paprsky do všech okolních stran a jeho vzdálenost od objektu ovlivňuje intenzitu osvětlení objektu.

```
1  // LightPower : síla světla
2  // 1/(distance*distance) je vzdálenost světla od objektu
3  color = MaterialDiffuseColor * LightColor * LightPower * cosTheta
       / (distance*distance);
4  ambient = LightPower / (distance*distance);
```

Zdrojový kód 11: diffuse

- **Ambient složka**
  Na intenzitu osvětlení objektu má vliv nejen samotný zdroj světla, ale také jeho okolí. Objekty v okolí taktéž odraží světlo a tím ovlivňují osvětlení objektu z dalších stran. Protože počítat vliv osvětlení jednotlivých objektů ve scéně je komplikované, jednodušše se docílí stejného efektu tím, že stínovaný objekt nepohltí všechno světlo, které na něj dopadá. To působí jakoby objekt sám nějaké světlo vyzařoval.

```
1  // emit : síla vyzařovaného světla objektem
2  vec3 emit = vec3(0.1);
3  vec3 MaterialAmbientColor = emit * MaterialDiffuseColor;
4  color = MaterialAmbientColor + MaterialDiffuseColor * LightColor
       * LightPower * cosTheta / (distance*distance);
```

Zdrojový kód 12: ambient

- **Specular složka**
  Intenzita odraženého světla od místa dopadu paprsku na povrch objektu závisí na pozici, ze které je na objekt pohlíženo (poloha pozorovatele, eye vector). Rozptyl odražených paprsků pak určuje míru zrcadlení, čím menší rozptyl, tím větší zrcadlení, efekt se více blíží odrazu světla v zrcadle.

```
1  // E : vektor oka skrze kameru
2  vec3 E = normalize(EyeDirection_cameraspace);
3
4  // R : směr, ve kterém se odráží světlo
5  vec3 R = reflect(-l,n);
6  float cosAlpha = clamp( dot( E,R ), 0,1 );
7
8  // mirrorPower : čím menší hodnota, tím vyšší zrcadlení
9  float mirrorPower = 5;
10
11 // specRef : vlastnost materiálu reflexe světla
12 specular = LightColor * specRef * pow(cosAlpha, mirrorPower);
```

Zdrojový kód 13: specular

Spojení všech tří komponent definuje výsledné stínování objektu:

```
color = diffuse + ambient + specular;
```

Následuje popis tří základních renderovacích technik stínování:

- **Flat shading**
  Výpočetně nejjednodušší a nejrychlejší model stínování. Výsledná barva je

spočítána vždy pro jeden vertex v polygonu a tato barva je pak přiřazena všem ostatním vertexům v polygonu. Stínování tímto způsobem je rychlé a efektivní, ale poskytuje nejméně kvalitní výsledek.

- **Gouraudovo stínování**
  Neboli stínování přes vertexy (per vertex). Ve vertex shaderu je pro každý vertex spočítána intenzita stínování a pokud je k dispozici i barva vertexu, je zkombinována s intenzitou stínu a poslána do rasterizéru. Ten interpoluje hodnoty souřadnic vertexů, texturové souřadnice vertexů a intenzitu stínování jednotlivých vertexů a pošle interpolované hodnoty do fragment shaderu. Je-li k dispozici textura, je namapována na příslušné fragmenty a k nim je pak přiřazena intenzita stínování.

- **Phongovo stínování**
  Neboli stínování přes fragmenty (per fragment). Vertex shader rovnou pošle informace o souřadnicích vertexů, normálách vertexů a texturových souřadnicích vertexů do rasterizéru. Ten interpoluje souřadnice, normály a texturové souřadnice vertexů a předá hodnoty fragment shaderu. Fragment shader následně provede výpočet pro intenzitu stínování objektu pro jednotlivé fragmenty a zkombinuje ji s barvou fragmentu či barvou textury. Phongovo stínování je výpočetně nejnáročnější, avšak poskytuje nejpřesnější a vizuálně nejkvalitnější výsledek.

8.3.2 Bump mapping

Bump mapping je technika pro realističtější rendrování složitějšího povrchu objektů či scény bez vytváření dalších polygonů. Je vhodná pro vykreslování těles s nerovnoměrným povrchem za pomocí textury a bump mapy. Bump mapa je textura tří barev, kde každá ze složek určuje směr normály. Například fragment bump mapy má modrou barvu, směřuje-li normála kolmo nahoru k povrchu textury. Uložení normál do textury má velkou slabinu. Nemění geometrii objektu a neovlivní tak ani jeho stín a je proto vhodná pro nepohyblivé 2D objekty. Pro pohyblivé 3D objekty je vhodnější technika displacement mapping.

9 Použité nástroje

V této části bakalářské práce je uveden stručný přehled nástrojů, které byly použity při programování shaderů pro tuto bakalářskou práci.

9.1 PBA editor

PBA editor je editor scén pro Stream Circle. Byl představen v kapitole 7.2.

9.2 Shadertoy

Shadertoy je webová aplikace pro programování fragment shaderů ve WebGL. Umožňuje nejen programování shaderů se snadnou a rychlou kompilací, ale také jejich sdílení a prohlížení, včetně zdrojových kódů, jiných uživatelů.
https://www.shadertoy.com/

9.3 Desmos

Webový nástroj pro znázorňování grafů matematických funkcí. Při programování shaderů pro bakalářskou práci byl využit především pro jednoduché znázornění průběhu více funkcí.
https://www.desmos.com/

9.4 Shadershop

Podobně jako Desmos je Shadershop nástrojem pro znázorňování matematických funkcí, avšak je navíc specializovaný přímo pro jazyk GLSL. Umožňuje tak jednoduše pracovat přímo s funkcemi jazyka GLSL, sčítat je, násobit, skládat a provádět nad nimi další operace. Kromě průběhu funkcí pak nabízí výstup odpovídající aplikaci funkcí ve fragment shaderu. Názorně zobrazuje převedení x, y souřadnic fragmentu na odpovídající rgb barvu na základě průběhu funkcí.
http://tobyschachman.com/Shadershop/

Závěr

Cílem bakalářské práce byl především podrobný popis shaderů (krátkých programů určených k programování grafické karty) a jejich implementace pro využití v grafickém editoru živého televizního vysílání.

První část práce se podrobně zaměřila na celkový popis zobrazovacího procesu, jednotlivé fáze grafické pipeline včetně zapojení konkrétních shaderů (vertex, geometry, tessellation, fragment). Představila programovací jazyk GLSL (OpenGL Shading Language) pro psaní shaderů včetně jeho nejpoužívanějších typů proměnných, kvalifikátorů a vestavěných funkcí, které značně usnadňují práci při programování grafiky. V další části pak práce uvedla zapojení shaderů do kontextu OpenGL, tedy přípravu dat v OpenGL aplikaci, předání dat do shaderů a zapojení shaderů do OpenGL.

Druhá část práce se pak věnovala praktickému využití předchozích poznatků. Začínala jednoduššími 2D shadery, statickými a dynamickými, vykreslujícími zajímavá pozadí, pokračovala efekty nad texturami, ať už se jednalo o dynamické deformace textury nebo statické filtry. Všechny tyto shadery byly implementovány pro konkrétní využití, tedy doplnění vysílaného obrazu o nové prvky a vytvoření vizuálně zajímavých efektů. Nakonec byly v praktické části bakalářské práce stručně představeny 3D shadery a zajímavé efekty s nimi spojené. Těmi jsou například stínování objektů nebo technika bump mapping.

Technik a možností při programování shaderů je mnoho a s neustále se vyvíjejícími grafickými kartami se budou možnosti počítačové grafiky i nadále rozrůstat. Tato bakalářská práce shrnula základní možnosti shaderů a je úvodem do této oblasti.

Conclusions

The aim of this thesis was detailed description of shaders (short programs used to program a graphics card behavior) and their implementations for application in a broadcast video editing.

The first part of this work has focused on rendering, individual phases of a rendering pipeline and an integration of vertex, geometry, tessellation and fragment shaders with the rendering pipeline. It has introduced GLSL (OpenGL Shading Language) for shaders programming with its most used variables, qualifiers and built-in functions for effective graphics programming.

The second part has aimed on practical use of the shaders. It has started with static and dynamic 2D shaders generating interesting effects and continued with texture effects, including dynamic texture deformations and static filters. All these shaders have been implemented for a specific application to alter broadcast video with new elements and create visually intriguing effects. Finally, 3D shaders and interesting effects created by them have been briefly introduced including objects shading and bump mapping technique.

There are many techniques and possibilities in shaders programming. Because of progressive development of graphics card, the potentialities of computer graphics are still growing. This bachelor work summarized basic possibilities of shaders and is an introduction into this part of computer graphics.

A Zdrojové kódy vybraných shaderů

V této příloze jsou k nahlédnutí zdrojové kódy vybraných shaderů. Tyto kódy doplňují předchozí text a jsou zajímavým příkladem programování GPU.

```
1  // PulsingColors.fragmentshader
2  #version 400
3
4  in vec2 UV;
5  in vec3 VertexPosition;
6  out vec4 color;
7
8  uniform float MaterialAlpha;
9  uniform int TextureMapping;
10 uniform vec3 MinBoundingVertex;
11 uniform vec3 MaxBoundingVertex;
12 uniform float Frame;
13 uniform float FrameToTime;
14
15 @include ..\TextureUV.shader
16
17 #define PI 3.14159265359
18
19 void main()
20 {
21   vec2 uv = TextureUV(UV, TextureMapping, MinBoundingVertex,
        MaxBoundingVertex, VertexPosition);
22   float time = Frame*FrameToTime;
23   vec3 fragColor = vec3(sin(time) , sin(time + (2*PI/3)), sin(time +
         (4*PI/3)));
24   color = vec4(fragColor, MaterialAlpha);
25 }
```

Zdrojový kód 14: barevné pulzování

```
1  // Light.fragmentshader
2  #version 400
3  @include ..\TextureUV.shader
4
5  in vec2 UV;
6  in vec3 VertexPosition;
7  out vec4 color;
8  uniform int TextureMapping;
9  uniform vec3 MinBoundingVertex, MaxBoundingVertex;
10 uniform float Frame, FrameToTime;
11 uniform vec3 MaterialAmbient, LightAmbient;
12 uniform float MaterialAlpha, LightAlpha, LightPower;
13
14 float time = FrameToTime * Frame;
15
16 void light(vec2 uv, inout vec3 fragColor, inout float fragAlpha)
17 {
18   float l;
19   float lp = LightPower * 0.1;
20   vec2 lc = vec2(mod(time, 1.0) + floor(uv.x), 0.5); // lc : light
        center
21   float d = length(uv-lc);
22
23   if(d == 0.0) l = lc.x * lp;
24   else
25   {
26     if(lc.x <= 0.5) l = lc.x * lp/(d*d + lp);
27     else l = (1-lc.x)*lp/(d*d + 0.5*lp);
28   }
29   fragColor = mix(MaterialAmbient, LightAmbient, vec3(l));
30   fragAlpha = mix(MaterialAlpha, LightAlpha, l);
31 }
32
33 void main()
34 {
35   vec2 uv = TextureUV(UV, TextureMapping, MinBoundingVertex,
        MaxBoundingVertex, VertexPosition);
36   vec3 fragColor = MaterialAmbient;
37   float fragAlpha = MaterialAlpha;
38   if(LightPower > 0.0) light(uv, fragColor, fragAlpha);
39   color = vec4(fragColor, fragAlpha);
40 }
```

Zdrojový kód 15: pohyblivé bodové světlo

```
1  // shader byl implementován ve webovém prostředí shadertoy.com
2  // https://www.shadertoy.com/view/4ldXRB
3
4  vec3 MaterialAmbient = vec3(1.0);
5  vec3 ClockAmbient = vec3(0.0);
6  vec3 ClockInAmbient = vec3(1.0);
7  float MaterialAlpha = 1.0;
8  float ClockAlpha =1.0;
9  float Thickness = 1.2;
10
11 int UTCYear, UTCMonth, UTCDay, UTCHour, UTCMinute, UTCSecond;
12
13 #define PI 3.14159265358979323846
14
15 void setTime(out int UTCYear,out int UTCMonth,out int UTCDay,out int
       UTCHour,out int UTCMinute, out int UTCSecond)
16 {
17   UTCYear = int(iDate.x);
18   UTCMonth = int(iDate.y);
19   UTCDay = int(iDate.z);
20   UTCHour = int(mod(iDate.w / 3600., 24.));
21   UTCMinute = int(mod(iDate.w / 60., 60.));
22   UTCSecond = int(mod(iDate.w, 60.));
23 }
24
25 void circle(in vec2 uv, vec2 c, float r, in vec3 inc, in float ina,
      out vec3 outc, out float outa)
26 {
27   float x = 1. - smoothstep(r, r+0.01, length(uv-c));
28   outc = mix(outc, inc, vec3(x));
29   outa = mix(outa, ina, x);
30 }
31
32 void line(in vec2 uv, vec2 a, vec2 b, float th, in vec3 inc, in
      float ina, out vec3 outc, out float outa)
33 {
34   vec2 uva = uv - a;
35   vec2 ba = b - a;
36   float l = clamp(dot(uva,ba) / dot(ba,ba), 0., 1.);
37   float x = 1.0 - smoothstep(th, th + 0.01, length(uva-ba * l));
38   outc = mix(outc, inc, vec3(x));
39   outa = mix(outa, ina, x);
40 }
41 // ...
```

Zdrojový kód 16: ručičkové hodiny (1/2)

```
// ...
void mainImage( out vec4 Color, in vec2 UV )
{
  vec2 uv = (2.0 * UV.xy - iResolution.xy)/ iResolution.y;

  setTime(UTCYear, UTCMonth, UTCDay, UTCHour, UTCMinute, UTCSecond);

  vec2 center = vec2(0.0);
  float radius = 0.8;
  float thick = Thickness * 0.01;
  if(Thickness < 0.0) thick = 0.015;

  vec3 pixCol = MaterialAmbient;
  float pixAlpha = MaterialAlpha;

  circle(uv, center, radius+thick, ClockAmbient, ClockAlpha, pixCol,
      pixAlpha);
  circle(uv, center, radius-thick, ClockInAmbient, ClockAlpha,
     pixCol, pixAlpha);
  mix(pixCol, vec3(0.0), 1.-smoothstep(0.2, radius, length(center-uv
     )));
  circle(uv, center, thick+0.02, ClockAmbient, ClockAlpha, pixCol,
     pixAlpha);

  vec2 sec = vec2(sin(2.*PI*float(UTCSecond)/60.), cos(2.*PI*float(
     UTCSecond)/60.));
  line(uv, center, center + (radius - 0.15) * sec, thick-0.01,
     ClockAmbient, ClockAlpha, pixCol, pixAlpha);

  vec2 min = vec2(sin(2.*PI*float(UTCMinute)/60.), cos(2.*PI*float(
     UTCMinute)/60.));
  line(uv, center, center + (radius - 0.15) * min, thick,
     ClockAmbient, ClockAlpha, pixCol, pixAlpha);

  vec2 hour = vec2(sin(2.*PI*float(UTCHour)/12.), cos(2.*PI*float(
     UTCHour)/12.));
  line(uv, center, center + (radius - 0.3) * hour, thick,
     ClockAmbient, ClockAlpha, pixCol, pixAlpha);

  float lc;
  for(int i = 0; i < 12; i++)
  {
    vec2 a, b;
    a.x = center.x + radius * cos(float(i) * 2.*PI/12.);
    a.y = center.y + radius * sin(float(i) * 2.*PI/12.);
    b.x = center.x + (radius-0.07) * cos(float(i) * 2.*PI/12.);
    b.y = center.y + (radius-0.07) * sin(float(i) * 2.*PI/12.);
    line(uv, a, b, thick/3.0, ClockAmbient, ClockAlpha, pixCol,
       pixAlpha);
  }
  Color = vec4(pixCol, pixAlpha);
}
```

Zdrojový kód 17: ručičkové hodiny (2/2)

```
1  // BandsTexture.fragmentshader
2  #version 400
3  @include ..\TextureUV.shader
4
5  in vec2 UV;
6  in vec3 VertexPosition;
7  uniform sampler2D Texture;
8  out vec4 color;
9
10 uniform int TextureMapping, ScaleMethod;
11 uniform vec3 MinBoundingVertex, MaxBoundingVertex;
12 uniform vec3 MaterialAmbient;
13 uniform float MaterialAlpha, BandsAlpha;
14 uniform float Frame, FrameToTime;
15 uniform vec2 SourceSize, TargetSize;
16 uniform int BandsNumVertical, BandsNumHorizontal;
17
18 #define PI 3.14159265358979323846
19
20 void main()
21 {
22   vec2 textureUV = TextureUV(UV, TextureMapping, MinBoundingVertex,
       MaxBoundingVertex, VertexPosition);
23   textureUV = ResizeUV(textureUV, SourceSize, TargetSize,
       ScaleMethod);
24   vec4 fragColor;
25
26   if((BandsNumVertical > 0) && (BandsNumHorizontal == 0)) // svislé
       rozdělení textury
27   {
28     if(mod(floor(textureUV.x*BandsNumVertical), 2.0) == 1.0)
29     textureUV.y += 0.5*sin(Frame*FrameToTime-PI*0.5);
30     else textureUV.y += 0.5*cos(Frame*FrameToTime);
31   }
32
33   if((BandsNumHorizontal > 0) && (BandsNumVertical == 0)) //
       vodorovné rozdělení textury
34   {
35     if(mod(floor(textureUV.y*BandsNumHorizontal), 2.0) == 1.0)
36     textureUV.x += 0.5*sin(Frame*FrameToTime-PI*0.5);
37     else textureUV.x += 0.5*cos(Frame*FrameToTime);
38   }
39
40   if ((textureUV.x < 0.0) || (textureUV.y < 0.0) || (textureUV.x >
       1.0) || (textureUV.y > 1.0))
41     fragColor = vec4(MaterialAmbient, MaterialAlpha);
42   else
43   {
44     fragColor = texture2D(Texture, textureUV).rgba;
45     fragColor.a *= BandsAlpha;
46   }
47   color = fragColor.rgba;
48 }
```

Zdrojový kód 18: rozbití textury na pohyblivé pruhy

```
1  // SpiralTexture.fragmentshader
2  #version 400
3  @include ..\TextureUV.shader
4
5  in vec2 UV; in vec3 VertexPosition;
6  out vec4 color;
7
8  uniform sampler2D Texture;
9  uniform int TextureMapping, ScaleMethod;
10 uniform vec3 MinBoundingVertex, MaxBoundingVertex;
11 uniform vec2 SourceSize, TargetSize;
12 uniform float Frame, FrameToTime;
13 uniform vec3 MaterialAmbient;
14 uniform float MaterialAlpha, SpiralDensity;
15
16 #define PI 3.14159265358979323846
17 float time = Frame*FrameToTime;
18
19 vec2 rotate(vec2 uv, float angle) {
20   mat2 rotateMat = mat2(cos(angle), -sin(angle), sin(angle), cos(
       angle));
21   return uv*rotateMat;
22 }
23
24 void main() {
25   vec2 textureUV = TextureUV(UV, TextureMapping, MinBoundingVertex,
       MaxBoundingVertex, VertexPosition); // <0; 1>
26   textureUV = 2.0*textureUV-1.0; // <-1; 1>
27   textureUV = ResizeUV(textureUV, SourceSize, TargetSize,
       ScaleMethod);
28
29   if ((textureUV.x < -1.0) || (textureUV.y < -1.0) || (textureUV.x >
        1.0) || (textureUV.y > 1.0))
30   color = vec4(MaterialAmbient, 1.0);
31   else {
32     float radius = 0.0;
33
34     if(textureUV.x != 0.0 && textureUV.y != 0.0)
35       radius = length(textureUV);
36
37     float angle = atan(textureUV.y, textureUV.x);
38     float rotAngle = 0.0;
39
40     if(SpiralDensity>0.0)
41       rotAngle = SpiralDensity * sin(1.5*time) * log(radius);
42
43     vec2 spiralCoords = rotate(textureUV, rotAngle);
44     spiralCoords = (spiralCoords + 1.0) / 2.0; // <0;1>
45     color = texture2D(Texture, spiralCoords).rgba;
46   }
47   color.a *= MaterialAlpha;
48 }
```

Zdrojový kód 19: spirálová deformace textury

```
// TextureVertices.fragmentshader
#version 400
@include ..\TextureUV.shader

in vec2 UV;
in vec3 VertexPosition;
uniform sampler2D Texture;
out vec4 color;

uniform int TextureMapping, ScaleMethod;
uniform vec3 MinBoundingVertex, MaxBoundingVertex;
uniform float Frame, FrameToTime;
uniform vec2 SourceSize; TargetSize;
uniform vec3 MaterialAmbient;
uniform float MaterialAlpha;
uniform vec2 A, B, C, D;

vec2 bilinearCoords(vec2 uv, vec2 a, vec2 b, vec2 c, vec2 d) {
  vec2 e, f, g, h;
  e = b-a;
  f = d-a;
  g = a-b+c-d;
  h = uv-a;

  float k2 = g.x*f.y - g.y*f.x;
  float k1 = (e.x*f.y - e.y*f.x) + (h.x*g.y - h.y*g.x);
  float k0 = h.x*e.y - h.y*e.x;

  if(k2 == 0.0) {
    float v = -k0 / k1;
    float u = (h.x - f.x*v)/(e.x + g.x*v);
    return vec2(u, v);
  }

  float discr = k1*k1 - 4.0*k0*k2;
  if(discr < 0.0) return vec2(-3.0);
  discr = sqrt(discr);

  float v1 = (-k1 - discr) / (2.0*k2);
  float v2 = (-k1 + discr) / (2.0*k2);
  float u1 = (h.x - f.x*v1)/(e.x + g.x*v1);
  float u2 = (h.x - f.x*v2)/(e.x + g.x*v2);

  bool cond1 = v1 > 0.0 && v1 < 1.0 && u1 > 0.0 && u1 < 1.0;
  bool cond2 = v2 > 0.0 && v2 < 1.0 && u2 > 0.0 && u2 < 1.0;

  vec2 result = vec2(-1.0);
  if(cond1 && !cond2) result = vec2(u1, v1);
  if(!cond1 && cond2) result = vec2(u2, v2);

  return result;
}
// ...
```

Zdrojový kód 20: interpolace obrazu podle nastavení jeho vrcholů (1/2)

```
1  // ...
2  void main()
3  {
4    vec2 textureUV = TextureUV(UV, TextureMapping, MinBoundingVertex,
       MaxBoundingVertex, VertexPosition);
5    textureUV = ResizeUV(textureUV, SourceSize, TargetSize,
       ScaleMethod);
6
7    vec2 newUV = textureUV;
8
9    vec2 a = A;
10   vec2 b = B;
11   vec2 c = C;
12   vec2 d = D;
13   \
14   if(a.x < 0.0) a.x = 0.0;
15   if(a.x > 1.0) a.x = 1.0;
16   if(a.y < 0.0) a.y = 0.0;
17   if(a.y > 1.0) a.y = 1.0;
18
19   if(b.x < 0.0) b.x = 0.0;
20   if(b.x > 1.0) b.x = 1.0;
21   if(b.y < 0.0) b.y = 0.0;
22   if(b.y > 1.0) b.y = 1.0;
23
24   if(c.x < 0.0) c.x = 0.0;
25   if(c.x > 1.0) c.x = 1.0;
26   if(c.y < 0.0) c.y = 0.0;
27   if(c.y > 1.0) c.y = 1.0;
28
29   if(d.x < 0.0) d.x = 0.0;
30   if(d.x > 1.0) d.x = 1.0;
31   if(d.y < 0.0) d.y = 0.0;
32   if(d.y > 1.0) d.y = 1.0;
33
34   newUV = bilinearCoords(textureUV, a, b, c, d);
35
36   if(newUV.x < 0.0 || newUV.y < 0.0 || newUV.x > 1.0 || newUV.y >
       1.0)
37   color = vec4(MaterialAmbient, 1.0);
38   else
39   color = texture2D(Texture, newUV);
40
41   color.a *= MaterialAlpha;
42 }
```

Zdrojový kód 21: interpolace obrazu podle nastavení jeho vrcholů (2/2)

```
1  // GrayFilter.fragmentshader
2  #version 400
3  @include ..\TextureUV.shader
4
5  in vec2 UV;
6  in vec3 VertexPosition;
7  out vec4 Color;
8
9  uniform sampler2D Texture;
10 uniform float MaterialAlpha;
11 uniform vec3 MaterialAmbient;
12 uniform int TextureMapping;
13 uniform vec3 MinBoundingVertex, MaxBoundingVertex;
14 uniform vec2 SourceSize, TargetSize;
15 uniform int ScaleMethod, GrayscaleMethod;
16
17 void main() {
18   vec2 textureUV = TextureUV(UV, TextureMapping, MinBoundingVertex,
        MaxBoundingVertex, VertexPosition);
19   textureUV = ResizeUV(textureUV, SourceSize, TargetSize,
        ScaleMethod);
20
21   if ((textureUV.x < 0.0) || (textureUV.y < 0.0) || (textureUV.x >
        1.0) || (textureUV.y > 1.0)) {
22     Color.rgba = vec4(MaterialAmbient.r, MaterialAmbient.g,
         MaterialAmbient.b, 1.0);
23   }
24   else
25   {
26     vec4 texCol = texture2D(Texture, textureUV).rgba;
27     if(GrayscaleMethod == 0) Color = texCol; // barevný obraz
28     else if(GrayscaleMethod == 1) { // metoda zprůměrování hodnot
29       float avgCol = (texCol.r + texCol.g + texCol.b) / 3.0;
30       Color = vec4(vec3(avgCol), texCol.a);
31     }
32     else if(GrayscaleMethod == 2) { // metoda luminance
33       float gray = dot(texCol.rgb, vec3(0.299, 0.587, 0.114));
34       Color = vec4(vec3(gray), texCol.a);
35     }
36     else if(GrayscaleMethod == 3) // metoda výběru barevného kanálu
37       Color = vec4(vec3(texCol.r), texCol.a);
38     else if(GrayscaleMethod == 4)
39       Color = vec4(vec3(texCol.g), texCol.a);
40     else if(GrayscaleMethod == 5)
41       Color = vec4(vec3(texCol.b), texCol.a);
42   }
43   Color.a = Color.a * MaterialAlpha;
44 }
```

Zdrojový kód 22: šedotónový filtr

```
1  // GaussBlurFilter.fragmentshader
2  #version 400
3  @include ..\TextureUV.shader
4
5  in vec2 UV;
6  in vec3 VertexPosition;
7  out vec4 color;
8
9  uniform sampler2D Texture;
10 uniform int TextureMapping, ScaleMethod;
11 uniform vec3 MinBoundingVertex, MaxBoundingVertex;
12 uniform vec2 SourceSize, TargetSize;
13 uniform float Frame, FrameToTime;
14 uniform vec3 MaterialAmbient;
15 uniform float MaterialAlpha;
16 uniform float EdgeDist;
17
18 float time = Frame*FrameToTime;
19
20 float rand(vec2 n)
21 {
22   return 0.5 + 0.5 * sin(dot(n.xy, vec2(12.9898, 78.233)))*
       43758.5453;
23 }
24
25 void main()
26 {
27   vec2 textureUV = TextureUV(UV, TextureMapping, MinBoundingVertex,
       MaxBoundingVertex, VertexPosition);
28   textureUV = ResizeUV(textureUV, SourceSize, TargetSize,
       ScaleMethod);
29
30   if ((textureUV.x < 0.0) || (textureUV.y < 0.0) || (textureUV.x >
       1.0) || (textureUV.y > 1.0))
31   color.rgba = vec4(MaterialAmbient.rgb, 1.0);
32   else
33   {
34     float ed = EdgeDist;
35     vec4 texCol;
36     vec4 fragCol = vec4(0.0);
37
38     if(EdgeDist > 0.03) ed = 0.03;
39     if(EdgeDist < 0.0) ed = 0.0;
40
41     vec2 raster3[9] = { vec2(-ed, -ed), vec2(0.0, -ed), vec2(ed, -ed)
         ,
42       vec2(-ed, 0.0), vec2(0.0, 0.0), vec2(ed, 0.0),
43       vec2(-ed, ed), vec2(0.0, ed), vec2(ed, ed) };
44
45     float filter3[9] = { 1., 2., 1.,
46       2., 4., 2.,
47       1., 2., 1. }; // sum = 16
48   // ...
```

Zdrojový kód 23: filtr Gaussovského rozostření (1/3)

```
// ...
  vec2 raster5[25] = { vec2(-2.0*ed, -2.0*ed), vec2(-ed, -2.0*ed),
     vec2(0.0, -2.0*ed), vec2(ed, -2.0*ed), vec2(2.0*ed, -2.0*ed),
    vec2(-2.0*ed, -ed), vec2(-ed, -ed), vec2(0.0, -ed), vec2(ed, -
       ed), vec2(2.0*ed, -ed),
    vec2(-2.0*ed, 0.0), vec2(-ed, 0.0), vec2(0.0, 0.0), vec2(ed,
       0.0), vec2(2.0*ed, 0.0),
    vec2(-2.0*ed, ed), vec2(-ed, ed), vec2(0.0, ed), vec2(ed, ed),
       vec2(2.0*ed, ed),
    vec2(-2.0*ed, 2.0*ed), vec2(-ed, 2.0*ed), vec2(0.0, 2.0*ed),
       vec2(ed, 2.0*ed), vec2(2.0*ed, 2.0*ed) };

  float filter5[25] = { 1., 4., 7., 4., 1.,
    4., 16., 26., 16., 4.,
    7., 26., 41., 26., 7.,
    4., 16., 26., 16., 4.,
    1., 4., 7., 4., 1. }; // sum = 273.0

  vec2 raster7[49] = { vec2(-3.0*ed, -3.0*ed), vec2(-2.0*ed, -3.0*
     ed), vec2(-ed, -3.0*ed), vec2(0.0, -3.0*ed), vec2(ed, -3.0*ed
     ), vec2(2.0*ed, -3.0*ed), vec2(3.0*ed, -3.0*ed),
    vec2(-3.0*ed, -2.0*ed), vec2(-2.0*ed, -2.0*ed), vec2(-ed, -2.0*
       ed), vec2(0.0, -2.0*ed), vec2(ed, -2.0*ed), vec2(2.0*ed,
       -2.0*ed), vec2(3.0*ed, -2.0*ed),
    vec2(-3.0*ed, -ed), vec2(-2.0*ed, -ed), vec2(-ed, -ed), vec2
       (0.0, -ed), vec2(ed, -ed), vec2(2.0*ed, -ed), vec2(3.0*ed, -
       ed),
    vec2(-3.0*ed, 0.0), vec2(-2.0*ed, 0.0), vec2(-ed, 0.0), vec2
       (0.0, 0.0), vec2(ed, 0.0), vec2(2.0*ed, 0.0), vec2(3.0*ed,
       0.0),
    vec2(-3.0*ed, ed), vec2(-2.0*ed, ed), vec2(-ed, ed), vec2(0.0,
       ed), vec2(ed, ed), vec2(2.0*ed, ed), vec2(3.0*ed, ed),
    vec2(-3.0*ed, 2.0*ed), vec2(-2.0*ed, 2.0*ed), vec2(-ed, 2.0*ed)
       , vec2(0.0, 2.0*ed), vec2(ed, 2.0*ed), vec2(2.0*ed, 2.0*ed),
        vec2(3.0*ed, 2.0*ed),
    vec2(-3.0*ed, 3.0*ed), vec2(-2.0*ed, 3.0*ed), vec2(-ed, 3.0*ed)
       , vec2(0.0, 3.0*ed), vec2(ed, 3.0*ed), vec2(2.0*ed, 3.0*ed),
        vec2(3.0*ed, 3.0*ed) };

  float filter7[49] = { 1., 1., 2., 2., 2., 1., 1.,
    1., 2., 2., 4., 2., 2., 1.,
    2., 2., 4., 8., 4., 2., 2.,
    2., 4., 8., 16., 8., 4., 2.,
    2., 2., 4., 8., 4., 2., 2.,
    1., 2., 2., 4., 2., 2., 1.,
    1., 1., 2., 2., 2., 1., 1. }; // sum = 130.0
// ...
```

Zdrojový kód 24: filtr Gaussovského rozostření (2/3)

```
  // ...
    if(ed <= 0.006)
    {
      for(int i=0; i<9; i++) {
        texCol = texture2D(Texture, textureUV+raster3[i] / 3.0).rgba;
        fragCol += texCol * vec4(filter3[i] / 16.0);
      }
    }
    else if((ed > 0.006) && (ed <= 0.01))
    {
      for(int i=0; i<25; i++)
      {
        vec4 texCol = texture2D(Texture, textureUV+raster5[i]/5.0).
            rgba;
        fragCol += texCol * vec4(filter5[i] / 273.0);
      }
    }
    else if (ed > 0.01)
    {
      vec2 rasterRand[14];
      float filterRand[14];
      float sum = 0.0;

      float r;
      r = rand(textureUV);
      r = floor(mod(r, 48.0));
      rasterRand[0] = raster7[int(r)];
      filterRand[0] = filter7[int(r)];
      sum += filterRand[0];

      for(int i=1; i<14; i++)
      {
        r = rand(textureUV*rasterRand[i-1]);
        r = floor(mod(r, 48.0));
        rasterRand[i] = raster7[int(r)];
        filterRand[i] = filter7[int(r)];
        sum += filterRand[i];
      }

      for(int i=0; i<14; i++) {
        texCol = texture2D(Texture, textureUV+rasterRand[i]/7.0).rgba;
        fragCol += texCol * vec4(filterRand[i] / sum);
      }
    }
    if(fragCol.r == 0.0 && fragCol.g == 0.0 && fragCol.b == 0.0 &&
        fragCol.a == 0.0) color = texture2D(Texture, textureUV).rgba;
    else color = vec4(fragCol);
  }
  color.a = color.a * MaterialAlpha;
}
```

Zdrojový kód 25: filtr Gaussovského rozostření (3/3)

B Obsah přiloženého CD/DVD

Přiložené DVD obsahuje dva adresáře:

`shadery/`
: Zde se nacházejí zdrojové kódy shaderů této bakalářské práce.

`pba/`
: Adresář obsahující důležité soubory pro PBA editor. Podadresář Debug/ obsahuje spustitelný soubor pba.editor.exe pro instalaci editoru. Dále lze v podadresáři pba-server/Config/A0E83EE7-6C2E-1014-A7F5-790168EA06EE/ nalézt vložené soubory k vykreslování grafiky, jako jsou obrázky (images/), objekty (objects/), scény (scenes/) a hlavně shadery (shaders/). V posledním zmíněném adresáři jsou vertex a fragment shadery patřící k editoru spolu s dalšími potřebnými soubory. V dalším podadresáři Bakalářská práce/ jsou umístěny fragment shadery, které byly implementovány pro tuto bakalářskou práci

Literatura

[1] SHREINER Dave, SELLERS Graham, KESSENICH John, LICEA-KANE Bill. OpenGL Programming Guide, Eighth Edition, The Official Guide to Learning OpenGL, Version 4.3

[2] BAILEY Mike, CUNNINGHAM Steve. Graphics Shaders: Theory and Practice, Second Edition

[3] https://cs.wikipedia.org/wiki/Shader

[4] https://cs.wikipedia.org/wiki/OpenGL

[5] https://en.wikipedia.org/wiki/Rendering_(computer_graphics))

[6] https://cs.wikipedia.org/wiki/OpenGL_Shading_Language

[7] http://gameprogrammingpatterns.com/game-loop.html

[8] https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/

[9] http://www.sallyx.org/sally/c/opengl/glsl

[10] https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)

[11] http://ivl.calit2.net/wiki/index.php/Discussion3S16

[12] https://www.khronos.org/opengl/wiki/Tessellation

[13] http://web.engr.oregonstate.edu/ mjb/cs519/Handouts/tessellation.1pp.pdf

[14] https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_tessellation_shader.txt

[15] http://nehe.gamedev.net/article/glsl_an_introduction/25007/

[16] http://fyzikalniolympiada.cz/texty/matematika/mkrivek.pdf

[17] http://www.iquilezles.org/www/articles/ibilinear/ibilinear.htm

[18] https://www.shadertoy.com/view/lsBSDm

[19] https://en.wikipedia.org/wiki/Shading

[20] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-8-basic-shading/

[21] http://www.csc.villanova.edu/ mdamian/Past/csc8470sp15/notes/shading.htm

[22] http://www.csc.villanova.edu/ mdamian/Past/csc8470sp15/notes/PhongLighting.pdf

[23] http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

[24] http://apoorvaj.io/exploring-bump-mapping-with-webgl.html

[25] http://lambdacube3d.com/slides.html

[26] https://www.cs.cmu.edu/ fp/courses/graphics/pdf-color/10-texture.pdf