MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Jazyky pro programování shaderů

Bakalářská práce

Lukáš Gregor

Brno, 2008

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: Mgr. Petr Tobola, Ph.D.

Shrnutí

Tato práce slouží jako výukový materiál pro programování shaderů. Práce vysvětluje princip a možnosti shaderů a jejich použití na současném hardwaru. Součástí práce jsou i demonstrační příklady, na kterých je vysvětlena práce se shadery v jazyku GLSL a princip vybraných grafických efektů.

Klíčová slova

OpenGL, GLUT, Direct3D, cg, GLSL, HLSL, pixel, vertex, shader

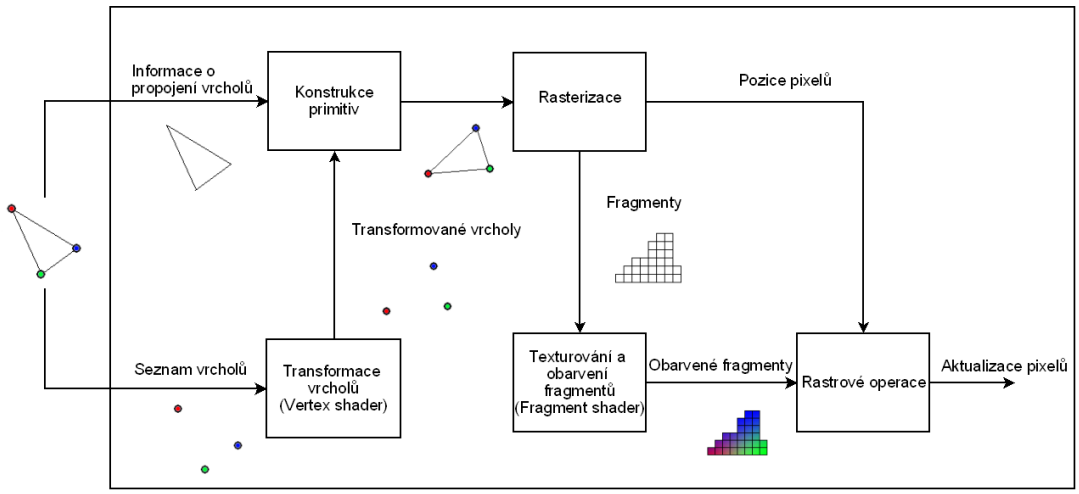
Obsah

Kapitola 1

Úvod

1.1 Popis grafické pipeline

Pro zobrazení 3D scény na obrazovce počítače je potřeba nemalého množství matematických výpočtů. Vzhledem k těmto nárokům není možné všechny výpočty provádět v reálném čase pomocí CPU počítače a proto se o velkou část stará grafický akcelerátor. Grafický akcelerátor je soustava na sebe navazujících jednotek (obrázek 1.1), které si mezi sebou předávají zpracovaná data, aby ve výsledku mohla být vykreslena výsledná scéna na obrazovku.



Obrázek 1.1.1 Zjednodušený model grafické pipeline

Pro zobrazení objektu ve scéně je potřeba nejdříve daný objekt sestavit z jednoduchých primitiv, vetšinou trojúhelníků nebo čtverců. Seznam těchto primitiv je předán na vstup grafickému akcelerátoru jako seznam vrcholů (bodů) a informace k jakému primitivu tento vrchol patří. Samotný vrchol obsahuje informaci o pozici ve scéně, svoji barvu a několik dalších parametrů. Každý vrchol z vstupního seznamu projde první jednotkou, ve které se transformuje jeho pozice na pozici z pohledu kamery, vygenerují se texturovací souřadnice a vypočte se nasvětlení vrcholů. Transformované vrcholy jsou předány další jednotce, která dostane na vstup i informaci o propojení vrcholů. Na výstupu jsou pak jednotlivá primitiva (trojúhelníky, obdélníky...) která se předají rasterizační jednotce, ve které jsou vygenerovány fragmenty. Fragmenty jsou ve své podstatě potenciální pixely výsledného obrazu, mají svoji barvu, souřadnici na obrazovce a hloubku (vzdálenost od kamery). Fragmenty jsou poslány do další jednotky, ve které se vypočítá výsledná barva fragmentu. V poslední jednotce jsou ze seznamu fragmentů pomocí různých testů vybrány jen ty fragmenty, které mají být vykresleny (jsou nejblíže kameře...). Výsledkem je pak seznam pixelů, které se mají na obrazovce vykreslit.

Kapitola 2

Popis a možnosti shaderů

U prvních grafických akcelerátorů nebylo možné ovlivnit chování těchto jednotek, jednalo se o tzv. fixed pipeline. Dnešní akcelerátory už umožňují změnit chování některých jednotek pomocí krátkých programů – shaderů. Pomocí shaderů je možné upravit chování jednotky pro transformaci vrcholů a jednotku pro zpracování fragmentů. U nejnovějších grafických akcelerátorů je možné použít i geometry shader, který je součástí jednotky pro konstrukci primitiv.

2.1 Vertex shader

Vertex shader je krátký program, který se provede nad každým vrcholem v jednotce pro zpracování vrcholů, předaným do grafické karty. Tento program může ovlivnit některé vlastnosti vrcholu jako jsou pozice, texturovací souřadnice, normálový vektor a barva. Nelze ale pomocí vertex shaderu vrcholy přidávat nebo odebírat. Při zpracování vrcholu nejsou dostupné informace o sousedních vrcholech, to znamená, že není známa topologie zpracovávaného objektu. Standardní vertex shader převádí pozici vrcholu ze světových souřadnic na souřadnice na obrazovce a je odpovědný za výpočet osvětlení.

2.2 Fragment shader

Někdy nazývaný pixel shader je program prováděný nad každým zpracovávaným fragmentem. Umožňuje měnit barvu fragmentu a jeho hloubku. Pokud je na objekt použita textura, je aplikována právě v této jednotce. Pozice fragmentu je sice v shaderu k dispozici ale nelze ji měnit. Na rozdíl od vertex shaderu mohou být fragmenty pomocí shaderu odebírány, to znamená, že je dále posláno méně fragmentů než bylo předáno na vstup. Protože fragment shader je volán relativně častěji než vertex shader (i jeden trojúhelník může generovat velké množství fragmentů), je u tohoto shaderu důležitá optimalizace.

2.3 Geometry shader

Geometry shader je zatím novinkou, která je podporována nejnovějšími kartami splňujícími DirectX 10. Jedná se o novou jednotku, která se nachází v jednotce pro konstrukci primitiv. Pomocí tohoto shaderu je možné generovat ze zpracovávaných primitiv další vrcholy a vytvářet z nich další primitiva která jsou pak předávána dál k rasterizaci.

Kapitola 3

Jazyky pro programování shaderů

Podobně jako pro CPU bylo pro programování shaderů vyvinuto několik jazyků. Základním jazykem je assembler, který podobně jako u assembleru pro CPU není příliš přehledný a programování složitějších efektů je poměrně náročné. Proto byly vyvinuty vyšší jazyky, které umožňují psát shadery přehledněji. V následujících podkapitolách jsou uvedeny tři nejpoužívanější z nich.

3.1 GLSL

Jazyk GLSL (OpenGL Shading Language) byl vyvinut pro rozhraní OpenGL a je velmi podobný programovacímu jazyku C. Syntaxe je pro jednotku vrcholů a fragmentů velmi podobná, rozdíly jsou pouze v dostupných funkcích a proměnných. Každý program, podobně jako u jazyku C, musí obsahovat hlavní funkci. Tato funkce musí mít název „main“, nesmí mít žádné parametry a nesmí vracet žádnou hodnotu. Příkazy v této funkci se provedou nad každým vstupním vrcholem (fragmentem). Parametry vrcholu (fragmentu) se do programu předávají pomocí speciálních globálních proměnných. Obsah vstupních proměnných je nastaven podle vstupních parametrů zpracovávaného vrcholu (fragmentu) před spuštěním funkce main. V programu se vypočítají nové hodnoty a uloží se do výstupních globálních proměnných. Název i typ těchto proměnných je pevně dán a nelze je programově měnit.

Příklad jednoduchého vertex shaderu:

```
void main()
{
    gl_Position = gl_Vertex;
    gl_FrontColor = vec4(0.0,1.0,0.0,1.0);
}
```

V tomto vertex shaderu jsou použity některé globální proměnné. První z nich je *gl_Position*, do které vertex shader zapisuje výslednou pozici vrcholu. Každý korektní vertex shader musí do této proměnné zapsat hodnotu. Další proměnnou je *gl_Vertex*. Jedná se o vstupní proměnnou, ze které lze jen číst a je v ní uložena vstupní pozice vrcholu. Poslední použitou proměnnou je *gl_FrontColor*, do které se ukládá výsledná barva vrcholu.

Tento krátký vertex shader vezme vstupní pozici vrcholu (*gl_Vertex*) a vloží jí do výstupní proměnné (*gl_Position*). Na druhém řádku se do proměnné *gl_FrontColor* zapíše vektor představující zelenou barvu.

3.2 CG

Jazyk CG byl vyvinut společností nVidia ve spolupráci se společností Microsoft. Stejně jako GLSL je i jeho syntaxe podobná programovacímu jazyku C. Jazyk byl vyvíjen jako univerzální nástroj a proto je použitelný jak s rozhraním OpenGL tak Direct3D. Hlavní rozdíl mezi CG a GLSL je v předávání vstupních a výstupních hodnot. Hodnoty nejsou předávány v globálních proměnných, ale jako parametry hlavní funkce. Protože vstupních i výstupních hodnot může být více (pozice, barva, normála), bývá na začátku programu deklarována struktura, obsahující všechny potřebné parametry vrcholu (fragmentu). V této struktuře jsou pomocí speciálních klíčových slov označeny proměnné, do kterých se mají uložit hodnoty vstupního vrcholu (fragmentu), nebo naopak obsahují výsledné hodnoty. Jako parametr hlavní funkce bývá struktura tohoto typu, obsahující vstupní hodnoty, stejně jako struktura vrácená hlavní funkcí, která obsahuje hodnoty výstupní.

Příklad jednoduchého vertex shaderu:

```
struct OutStruct {
        float4 position : POSITION;
        float3 color    : COLOR;
};

OutStruct main(float3 position : POSITION)
{
        OutStruct OUT;
        OUT.position = float4(position,1);
        OUT.color    = float3(0, 1, 0);

        return OUT;
}
```

Tento příklad má stejnou funkčnost jako příklad v popisu GLSL. Na začátku programu je deklarace struktury *OutStruct*, která slouží jako uložiště výstupních dat a obsahuje pozici a barvu výsledného vrcholu. Důležitou částí této definice je klíčové slovo *POSITION* a *COLOR*. Tím je definováno, která proměnná obsahuje výslednou pozici nebo barvu vrcholu. Následuje definice hlavní funkce programu, která vrací strukturu *OutStruc* (výsledné parametry vrcholu) a vstupní vektor position, označený klíčovým slovem *POSITION* (vstupní pozice vrcholu). V tělu funkce se vytvoří instance struktury *OutStruc,* její proměnné se nastaví na vstupní pozici vrcholu a zelenou barvu. Posledním příkazem se struktura obsahující upravené hodnoty vrací jako návratová hodnota funkce.

3.3 HLSL

Jazyk HLSL (High level shader language) byl vyvinut společností Microsoft. Jedná se o jazyk pro rozhraní Direct3D a syntakticky je totožný s jazykem CG. Jeho výhodou je přímá podpora v Direct3D bez potřeby dodatečných knihoven, a proto se velmi často používá právě v aplikacích pracujících s tímto rozhraním (Xbox).

Kapitola 4

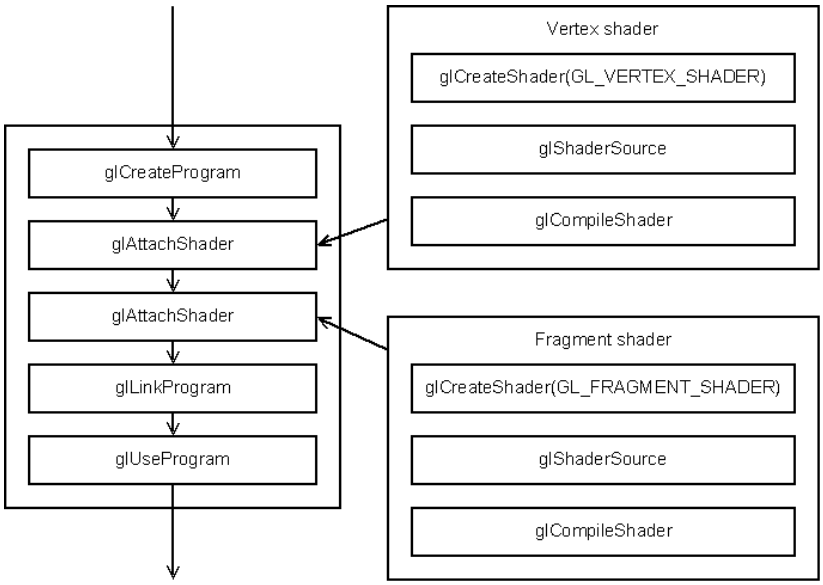
Praktické použití shaderů

Pro spuštění napsaného shaderu, musí být zdrojový kód podobně jako kód v jazyku C přeložen do binární podoby. Všechny tři jazyky umožňují překlad přímo v kódu aplikace pomocí funkcí, obsažených v potřebných knihovnách. V následujících třech kapitolách bude popsán stručný postup pro kompilaci a spuštění shaderů v jazyku C.

4.1 GLSL

V OpenGL je od verze 2.0 implementována podpora pro práci s jazykem GLSL. Standardně nejsou tyto funkce dostupné a proto je potřeba jejich definici do programu přidat, nebo využít některou z rozšiřujících knihoven, které pro OpenGL vznikly. V případě GLSL jsou potřebné funkce implementovány v knihovně GLEW (The OpenGL Extension Wrangler Library), která je volně dostupná a nabízí veškeré potřebné funkce pro práci s GLSL.

Postup pro úspěšné přeložení je zobrazen na následujícím obrázku.



Obrázek 4.1.1 Postup kompilace v GLSL

V praxi pak aplikace pro práci s jednotlivými objekty potřebuje několik proměnných. Nejdůležitější z nich jsou textové řetězce, které obsahují zdrojový kód shaderů.

*const char *vertexCode; //< obsahuje kod vertex shaderu*
*const char *fragmentCode; //< obsahuje kod fragment shaderu*

Dále je potřeba si uložit id jednotlivých objektů. Celkem budou tři, jeden pro vertex shader, jeden pro fragment shader a jeden pro výsledný program.

```
GLuint vertexShader;
GLuint fragmentShader;
GLuint program;
```

Při samotném překladu jsou nejdříve vytvořeny prázdné objekty pro vertex a fragment shader pomocí funkce *glCreateShaderObject*. V parametru funkce se určí typ vytvářeného objektu a id vytvořeného objektu se uloží do připravené proměnné.

```
vertexShader = glCreateShader(GL_VERTEX_SHADER);
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
```

Tím vzniknou prázdné objekty, do kterých se funkcí *glShaderSource* nahraje zdrojový text vertex a fragment shaderu.

```
glShaderSource(vertexShader, 1, &vertexCode, NULL);
glShaderSource(fragmentShader, 1, &fragmentCode, NULL);
```

Poslední částí je kompilace funkcí glCompileShader.

```
glCompileShader(vertexShader);
glCompileShader(fragmentShader);
```

Pokud vše proběhlo korektně, jsou vytvořeny dva objekty. Další částí je vytvoření objektu pro hlavní program funkcí *glCreateProgramObject.*

```
program = glCreateProgram();
```

Do programu se přidají dříve vytvořené objekty funkcí *glAttachShader*.

```
glAttachShader(program,vertexShader);
glAttachShader(program,fragmentShader);
```

Výsledný program se sestaví příkazem *glLinkProgram.*

```
glLinkProgram(program);
```

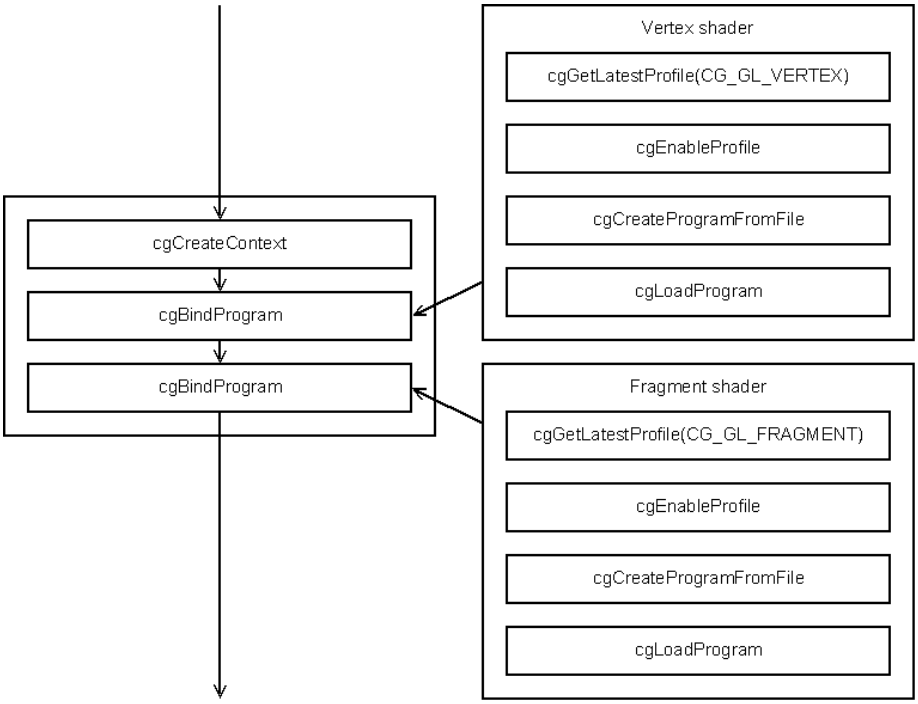
Tím je výsledný program sestaven. Poslední funkcí je *glUseProgram*, kterou se program nastaví jako aktivní.

```
glUseProgram(program);
```

Jakmile je program nastaven jako aktivní, nahradí pevně danou funkci standardní jednotky pro zpracování vrcholů a fragmentů, dokud není aplikace ukončena nebo není zavolána funkce *glUseProgram* s nulovým parametrem.

4.2 CG

Jazyk CG není standardně podporován ani jedním z rozhraní (OpenGL ani Direct3D) a pro práci s ním je nutno nainstalovat potřebné knihovny, které jsou volně k dispozici ke stažení na stránkách společnosti nVidia. Postup pro kompilaci a spuštění shaderu v jazyku CG je velmi podobný spuštění v GLSL.



Obrázek 4.2.1 Postup kompilace v CG

Podobně jako u jazyku GLSL jsou pro spuštění potřeba tři objekty. Prvním je objekt typu *CGcontext*, kontejner, který bude obsahovat všechna potřebná data. Další dva objekty slouží pro uchování fragment a vertex shaderu. Poslední dvě proměnné slouží pro získání profilů (verze shaderů), které jsou aktuálním hardwarem podporovány.

CGcontext context;
CGprogram vertexProgram;
CGprogram fragmentProgram;

CGprofile vertexProfile;
CGprofile fragmentProfile;

Nejdříve je potřeba vytvořit hlavní kontejner funkcí *cgCreateContext* a zjistit dostupnou verzi shaderů pomocí c*gGLGetLatestProfile.* Tato funkce vrací nejvyšší dostupnou verzi vertex nebo fragment shaderu.

```
context = cgCreateContext();
vertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);
cgGLEnableProfile(vertexProfile);
fragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
cgGLEnableProfile(fragmentProfile);
```

Následuje načtení a kompilace zdrojového kódu.

```
vertexProgram = cgCreateProgramFromFile(context,
                                        CG_SOURCE,
                                       "vertex.sh",
                                        vertexProfile,
                                        "main",
                                        NULL);
fragmentProgram = cgCreateProgramFromFile(context,
                                        CG_SOURCE,
                                       "fragment.sh",
                                        fragmentProfile,
                                        "main",
                                        NULL);
```

Poslední částí je nahrání programů do grafické karty.

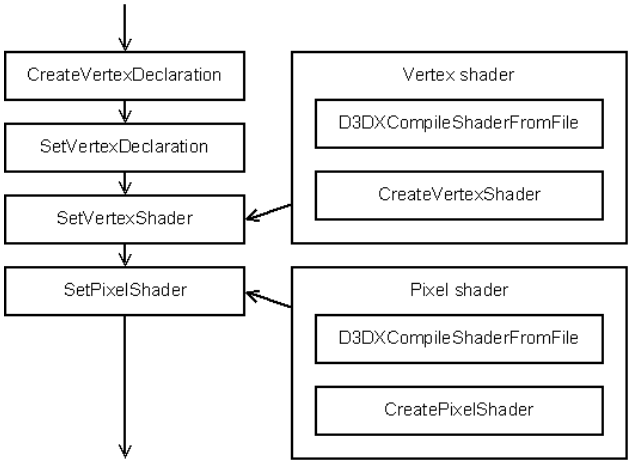
```
cgGLLoadProgram(vertexProgram);
cgGLLoadProgram(fragmentProgram);

cgGLBindProgram(vertexProgram);
cgGLBindProgram(fragmentProgram);
```

Uvedený příklad je určen pro rozhraní OpenGL. Funkce pro rozhraní Direct3D mají stejný název, liší se pouze nahrazením zkratky GL za D3D (*cgD3DGetLatestProfile, cgD3DLoadProgram, cgD3DBindProgram*).

4.3 HLSL

Funkce, potřebné pro úspěšnou kompilaci a spuštění shaderu v jazyku HLSL, jsou implementovány přímo v DirectX SDK, a proto není třeba instalovat dodatečné knihovny.

Postup kompilace je opět znázorněn na následujícím obrázku a praktickém příkladu.



Obrázek 4.3.1 Postup kompilace v HLSL

Na rozdíl od předchozích dvou jazyků jsou funkce pro práci se shadery implementovány jako metody objektu *LPDIRECT3DDEVICE9*, který slouží ke komunikaci s rozhraním Direct3D.

Deklarace proměnných, potřebných pro překlad a práci se shadery.

LPDIRECT3DVERTEXSHADER9 vertexShader, pixelShader;

Před kompilací je potřeba nastavit formát dat vrcholů, které budou předávány z aplikace do grafické karty. V tomto případě bude každý vrchol obsahovat trojrozměrný vektor (*D3DDECLTYPE_FLOAT3)*, který bude vyjadřovat jeho pozici (*D3DDECLUSAGE_POSITION*) a barvu (*D3DDECLUSAGE_COLOR*) typu *D3DDECLTYPE_D3DCOLOR*.

```
D3DVERTEXELEMENT9 declaration[] =
{
        {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 0},

        {0, 12, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_COLOR,   0},

        D3DDECL_END()
};

d3dDevice->CreateVertexDeclaration( declaration, &vertexDeclaration );
d3dDevice->SetVertexDeclaration( vertexDeclaration );
```

Následuje samotná kompilace a vytvoření shaderů. Použité proměnné vertexCode a pixelCode slouží jako dočasné uložiště přeložených programů. Kompilace funkcí *D3DXCompileShaderFromFile* je velmi podobná kompilaci pomocí *cgCreateProgramFromFile* v jazyku CG.

```
LPD3DXBUFFER vertexCode, pixelCode;

D3DXCompileShaderFromFile( L"vertex.sh",
                           NULL,
                           NULL,
                           "main",
                           "vs_1_1",
                           NULL,
                           &vertexCode,
                           &errors,
                           NULL);
D3DXCompileShaderFromFile( L"pixel.sh",
                           NULL,
                           NULL,
                           "main",
                           "ps_1_1",
                           NULL,
                           &pixelCode,
                           NULL,
                           NULL);

d3dDevice->CreateVertexShader( (DWORD*)code->GetBufferPointer(), &vertexShader );
d3dDevice->CreatePixelShader((DWORD*)code->GetBufferPointer(), &pixelShader);
```

Uvolnění nepotřebných prostředků. Shadery v binární formě jsou už uloženy v proměnných *vertexShader* a *pixelShader*.

```
vertexCode->Release();
pixelCode->Release();
```

Nastavení aktivních shaderů.

```
d3dDevice->SetVertexShader( vertexShader );
d3dDevice->SetPixelShader( pixelShader );
```

Kapitola 5

Programování shaderů

V následujících kapitolách budou popsány vlastnosti shaderů. Protože všechny tři jazyky jsou si velmi podobné, budou příklady popsány pouze v jenom z nich a to v GLSL.

5.1 Datové typy a proměnné

Práce s datovými typy a proměnnými je prakticky totožná s programovacím jazykem C. Datové typy jsou rozšířeny především o matice a vektory. Tyto typy vycházejí z celočíselného i reálného typu int a float a jejich název se liší podle použitého jazyka. Přehled dostupných číselných typů je v následující tabulce.

GLSL	CG a HLSL	Popis
int	int	Celočíselný typ
ivec2, ivec3, ivec4	int2, int3, int4	Vektor celých čísel
float	float	Reálné číslo
vec2, vec3, vec4	float2, float3, float4	Vektor reálných čísel
bool	bool	Logický typ
bvec2, bvec3, bvec4	bool2, bool3, bool4	Vektor logických typů
mat2, mat3, mat4	float2x2, float3x3, float4x4	Matice reálných čísel

Práce s jednoduchými typy se neliší od jazyku C. Výraznější rozdíl je až při práci s vektory a maticemi. Pro přístup k jednotlivým složkám lze použít buď tečkovou konvenci (třída) nebo hranaté závorky (pole). Zadání hodnot do vektoru je možné i pomocí konstruktoru.

Zápis hodnoty do proměnné:

vec3 position = vec3(1.0, 2.5, 8.0);
věc3 newPostion = postition;

Složky vektoru jsou přístupné pomocí hranatých závorek nebo pomocí čtveřice atributů *x, z, y, w,* pokud se jedná o vektor vyjadřující barvu *r, g, b, a,* v případě texturovacích souřadnic *s, t, p, q*.

float positionX = position.x;
float positionY = position[1];

přístup k více složkám

vec2 position2D = postion.xy;

V případě matic je postup velmi podobný.

mat2 matrix = mat2(1.0, 0.0, 1.0, 0.0);
vec2 row = matrix[0];
float i = matrix [1][0];

5.2 Jednoduchý vertex shader

Jak už bylo zmíněno v popisu grafické pipeline, vertex shader je spuštěn nad každým zpracovávaným vrcholem. V jazyku GLSL jsou vstupní a výstupní hodnoty předávané pomocí globálních proměnných. Význam těch nejpoužívanějších je popsán v následujících tabulkách.

Vstupní proměnné:

Název	Význam hodnoty
vec4 gl_Vertex;	Pozice vrcholu ve světových souřadnicích
vec3 gl_Normal;	Normála vrcholu
vec4 gl_Color;	Barva vrcholu
vec4 gl_MultiTexCoord0 - 7;	Texturovací souřadnice

Výstupní proměnné:

Název	Význam hodnoty
vec4 gl_Position;	Výsledná pozice vrcholu, korektní vertex shader musí do této proměnné zapsat hodnotu
vec4 gl_FrontColor;	Přední barva vrcholu
vec4 gl_BackColor;	Zadní barva vrcholu
vec4 gl_TexCoord[ ];	Texturovací souřadnice

Standardní jednotka pro zpracování vrcholu je zodpovědná za transformaci každého vrcholu. V praxi to znamená, že vstupní pozice vrcholu je ve světových souřadnicích a úkolem jednotky je převést tuto pozici na souřadnicový systém z pohledu kamery a správně spočítat efekt perspektivy. Proto jsou ve vertex shaderu k dispozici proměnné *gl_ModelViewMatrix* a *gl_ProjectionMatrix*. V těchto proměnných jsou uloženy transformační matice, pomocí kterých lze jednoduše a hlavně rychle danou transformaci provést.

Vynásobením pozice maticemi se pozice transformuje na souřadný systém z pohledu kamery.

*gl_Position = gl_ModelViewMatrix * gl_ProjectionMatrix * gl_Vertex;*

Součin těchto dvou matic lze nahradit proměnnou *gl_ModelViewProjectionMatrix*, ve které je

součin transformační a pro projekční matice předpočítán.

*gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;*

Tento jednoduchý vertex shader, nastaví všem vrcholům červenou barvu.

```
void main()
{
        // transformace vrcholu
        gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
        // nastaveni barvy vrcholu na cervenou
        gl_FrontColor = vec4(1.0,0.0,0.0,1.0);
}
```

Obrázek 5.2.1 Jednoduchý vertex shader.

5.3 Jednoduchý fragment shader

Podobně jako u vertex shaderu i ve fragment shaderu jsou dostupné speciální proměnné.

Vstupní proměnné:

Název	Význam hodnoty
vec4 gl_Color;	Vstupní barva fragmentu
vec4 gl_TexCoord[ ];	Texturovací souřadnice
vec4 gl_FragCoord;	Pozice fragmentu

Výstupní proměnné:

Název	Význam hodnoty
vec4 gl_FragColor;	Výsledná barva
float gl_FragDepth;	Výsledná hloubka

Na rozdíl od vertex shaderu, fragment shader má možnost zpracovávaný fragment zrušit (neposlat k dalšímu zpracování) příkazem *discard*. Tím se ukončí běh shaderu nad tímto fragmentem a fragment se zahodí.

Následující příklad vytváří stejný efekt jako výše popsaný příklad pro vertex shader. V tomto případě se ale výsledná barva nastaví až pří zpracování fragmentů.

```
void main()
{
        // nastaveni barvy fragmentu na cervenou
        gl_FragColor = vec4(1.0,0.0,0.0,1.0);
}
```

5.4 Ladění shaderů

Ladění především složitějších shaderů není jednoduchou záležitostí. Protože přeložený kód běží na grafické kartě, není možné program krokovat ani jinak ladit. Přesto jsou ale k dipozici metody pro výpis výsledku kompilace shaderu i celého programu, díky kterým lze v kódu rychleji odhalit syntaktické chyby.

K dispozici jsou dvě funkce, glGetShaderiv pro dotazování na stav vertex nebo fragment shaderu a glGetProgramiv pro zjištění stavu výsledného programu v závislosti na typu parametru předaného funkci.

*void glGetShaderiv(GLuint shader, GLenum type, GLint *value);*

Název typu	Význam
GL_SHADER_TYPE	Vrací typ objektu (GL_VERTEX_SHADER nebo GL_FRAGMENT_SHADER)
GL_COMPILE_STATUS	Vrací GL_TRUE pokud byl shader úspěšně přeložen.
GL_INFO_LOG_LENGTH	Vrací počet znaků, které budou vráceny funkcí glGetShaderInfoLog (viz. níže).

*void glGetProgramiv(GLuint program, GLenum type, GLint *value);*

Název typu	Význam
GL_LINK_STATUS	Vrací GL_TRUE pokud byl výsledný program úspěšně slinkován.

GL_INFO_LOG_LENGTH	Vrací počet znaků, které budou vráceny funkcí glGetProgramInfoLog (viz. níže).
GL_ATTACHED_SHADERS	Vrací počet shader objektů v programu.
GL_ACTIVE_ATTRIBUTES	Vrací počet proměnných typu attribute v programu
GL_ACTIVE_UNIFORMS	Vrací počet proměnných typu uniform v programu

Pokud překlad selhal, vrací funkce pro parametr GL_COMPILE_STATUS nebo GL_LINK_STATUS v případě funkce glGetProgramiv hodnotu *GL_FALSE*. Pro výpis chybové zprávy překladače jsou dostupné funkce *glGetShaderInfoLog* a *glGetProgramInfoLog*.

*void glGetShaderInfoLog(GLuint shader, Glsizei maxLen, GLsizei *length, GLchar *log);*

*void glGetProgramInfoLog(GLuint program, GLsizei maxLen,GLsizei *length, GLchar *log)*

Následující kód obsahuje funkce pro výpis chybového hlášení překladače pro objekt zadaný v parametru.

```
void PrintShaderLog(GLuint obj)
{
	int logLength = 0;
	glGetShaderiv(obj, GL_INFO_LOG_LENGTH, &logLength);

	if (logLength > 0)
	{
		int length = 0;
		char *infoLog = new char[logLength];
		glGetShaderInfoLog(obj, logLength, &length, infoLog);
		std::cout << infoLog << "\n";
		delete infoLog;
	}
}

void PrintProgramLog(GLuint obj)
{
	int logLength = 0;
	glGetProgramiv(obj, GL_INFO_LOG_LENGTH, &logLength);

	if (logLength > 0)
	{
		int length = 0;
		char *infoLog = new char[logLength];
		glGetProgramInfoLog(obj, logLength, &length, infoLog);
		std::cout << infoLog << "\n";
		delete infoLog;
	}
}
```

5.5 Předávání hodnot

Z úvodního příkladu by se mohlo zdát, že veškeré vstupní hodnoty musí být shaderu předány pouze v pevně definovaných proměnných. To není tak úplně pravda, protože hodnoty mohou být předávány z aplikace i pomocí sdílených proměnných. Tato proměnná musí být v shaderu definována jako globální a při deklaraci musí být před jejím typem klíčové slovo.

Uniform
Pokud je proměnná uvozena klíčovým slovem *uniform*, jedná se o vstupní proměnnou. Takto může být deklarována proměnná ve vertex i fragment shaderu ale hodnotu této proměnné nelze v aplikaci měnit během definice primitiva (v OpenGL mezi příkazy *glBegin* a *glEnd*), její hodnota je pro všechny vrcholy jednoho primitiva konstantní. Do této proměnné zapisuje pouze aplikace, v obou shaderech je tato proměnná pouze pro čtení.

Attribute
Pro proměnné s klíčovým slovem *attribute* neplatí předchozí omezení a její hodnota se může v aplikaci měnit prakticky kdykoliv. Tyto proměnné lze deklarovat pouze ve vertex shaderu a používají se pro zadávání dodatečných hodnot ke konkrétním vrcholům. I do této proměnné může zapisovat pouze aplikace a ve vertex shaderu je pouze pro čtení.

Varying
Jedná se o speciální proměnnou pro předávání hodnot z vertex shaderu do fragment shaderu. Tato proměnná musí mít stejnou deklaraci v obou shaderech a vertex shader do ní musí zapsat hodnotou. V fragment shaderu je tato proměnná pouze pro čtení. Hodnota této proměnné je při rasterizaci lineárně interpolována pro každý fragment podle pozice podobně jako například barva.

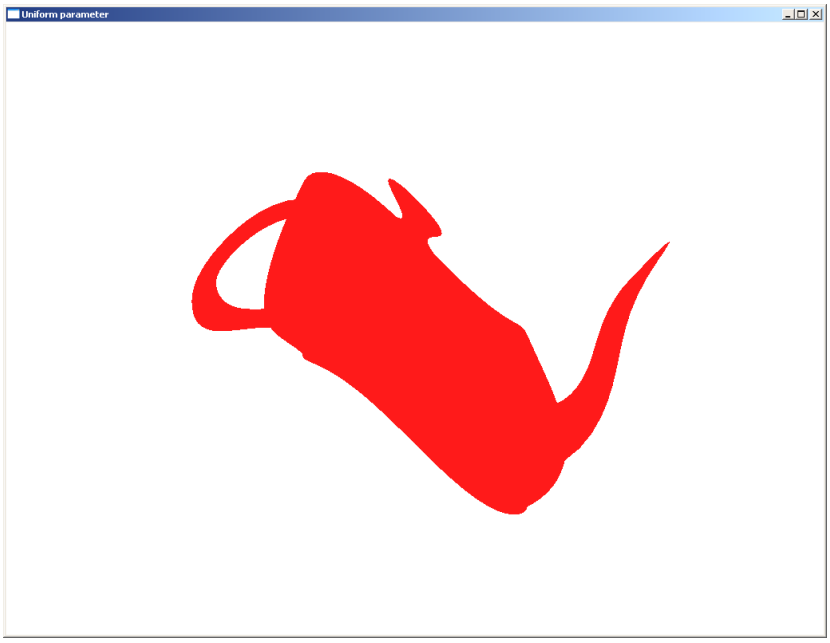
Následující příklad ilustruje použití proměnné typu *uniform*. Podle hodnoty proměnné je upravena souřadnice v ose *z* každého z vrcholů.

Zdrojový kód pro vertex shader:

```
uniform float delta;
void main()
{
        vec4 position = gl_Vertex;

        // posun vrcholu v ose z.
        position.z = position.z + sin(position.x + delta);

        // transformace pozice
        gl_Position = gl_ModelViewProjectionMatrix * position;

        // nastaveni barvy
        gl_FrontColor = gl_Color;
}
```

Na prvním řádku je deklarována proměnná *delta.* Tato proměnná je označena slovem uniform, do této proměnné bude hodnotu zapisovat aplikace. V hlavní funkci se upraví souřadnice vrcholu v ose *z*. K současné souřadnici se připočte hodnota funkce *sin*, která je závislá na pozici

vrcholu v ose *x* a hodnotě proměnné *delta*.



Obrázek 5.5.1 Ukázka vlnění objektu pomocí vertex shaderu

Na straně aplikace je potřeba před zápisem do proměnné získat potřebný identifikátor funkcí *glGetUniformLocation*. Prvním parametrem této funkce je identifikátor objektu programu, ve kterém se daná proměnná nachází. Jedná se o hodnotu vrácenou proměnnou *glCreateProgram* při kompilaci shader programu. Druhým parametrem je název této proměnné v shaderu. Funkce vrací identifikátor proměnné, který bude použit později při zápisu hodnoty.

GLint deltaRef = glGetUniformLocation(p,"delta");

Tím je dokončena inicializační část. Samotný zápis se provádí funkcí *glUniform*. Tato funkce má několik variant v závislosti na typu proměnné, v tomto případě použita funkce *glUniform1f* (jedna proměnná typu float).

glUniform1f(deltaRef, delta);

Protože v jazyku C není k dispozici datový typ představující vektor nebo matici, zadávají se hodnoty buď po jednotlivých složkách nebo jako pole hodnot. V následující tabulce je přehled nejpoužívanějších variant funkce *glUniform.*

glUniform1f - glUniform4f	Má dva až pět parametrů, používá se pro zápis jedné proměnné a dvou až čtyř složkových vektorů.
glUniform1i - glUniform4i	Používá se stejně jako předchozí funkce, předávané hodnoty jsou celá čísla.
glUniform1fv - glUniform4fv	Od předchozích funkcí se liší pouze formátem předávaných hodnot. Hodnoty se funkcím předávají jako pole hodnot.

glUniform1iv - glUniform4iv	Celočíselná varianta předchozích funkcí.
glUniformMatrix2fv - glUniformMatrix4fv	Funkce používané pro zadávání matic, hodnoty jsou předávány jako pole hodnot

5.6 Práce s texturou

V následujícím příkladě bude demonstrován postup pro nanesení bitmapy na vykreslovaný objekt. Tento efekt je aplikován při zpracování fragmentů, kde se výsledná barva fragmentu upraví podle pixelu bitmapy, který podle texturovacích souřadnic odpovídá danému fragmentu.

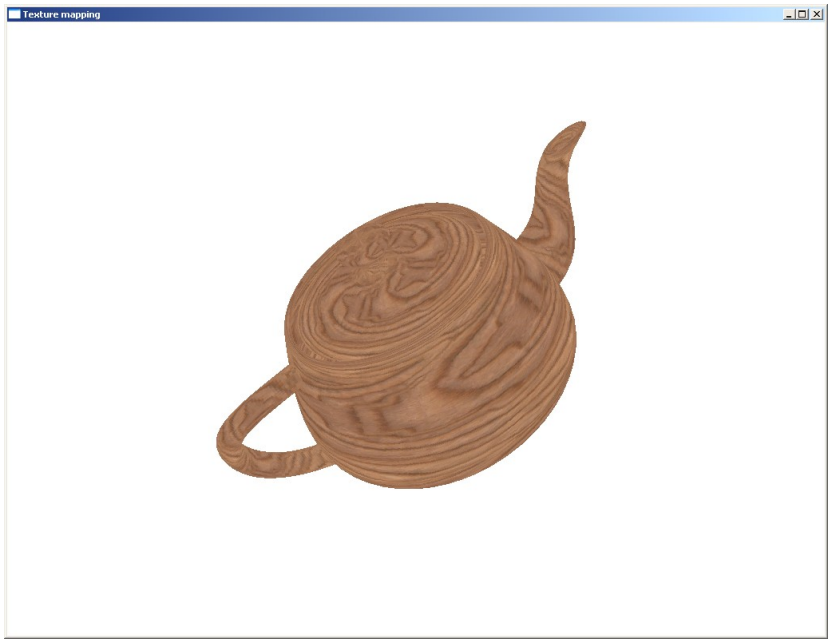
Vertex shader bude mít v tomto příkladu za úkol pouze předání texturovacích souřadnic do fragment shaderu.

```
void main()
{
	// transformace pozice
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

	// predani texturovacich koordinatu
	gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Texturovací souřadnice jsou zadány aplikací pro každý vrchol a stejně jako proměnné typu *varying* se hodnota pro jednotlivé fragmenty mezi vrcholy lineárně interpoluje. Následující kód vypočte výslednou barvu fragmentu podle nanášené textury.

```
uniform sampler2D tex;
void main()
{
	vec4 color = texture2D(tex, gl_TexCoord[0].st);
	gl_FragColor = color;
}
```

Zde je použita proměnná typu sampler2D. Hodnota proměnné tohoto typu je většinou nastavena aplikací a určuje, se kterou texturou má fragment shader pracovat. Novou funkcí je *texture2D*, která vrací barvu odpovídající zadaným souřadnicím v textuře. Získaná hodnota se použije jako výsledná barva fragmentu.



Obrázek 5.6.1 Aplikace textury na objekt.

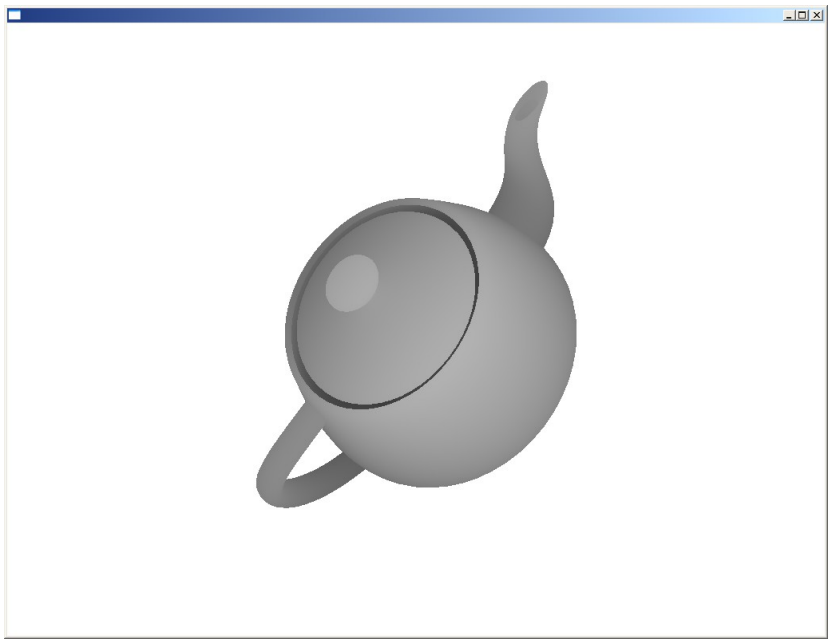
Kapitola 6

Příklady efektů

Následující část práce se bude zabývat aplikací shaderů pro dosažení jednoduchých efektů.

6.1 Hloubková mapa

Pomocí hloubkové mapy lze jednoduše vyjádřit vzdálenost jednotlivých objektů od kamery. Vykreslená scéna má většinou jednotnou barvu a světlost každého pixelu vyjadřuje právě vzdálenost od pozorovatele. Čím je bod obrazu světlejší, tím je daný objekt blíže pozorovateli.



Obrázek 6.1.1 Hloubková mapa.

Výpočet tohoto efektu lze provést ve vertex shaderu. Výhodou je možnost využití transformovaných souřadnic. Jakmile je pozice vrcholu transformována na souřadný systém z pohledu kamery, je vzdálenost od pozorovatele určena přímo souřadnicí vrcholu v ose *z*.

Pro výpočet bude použita funkce smoothstep, která má následující syntaxi:

float smoothstep (float edge0, float edge1, float x);

Pokud je hodnota *x* v intervalu od *edge0* do *edge1*, je návratová hodnota funkce lineárně interpolována v intervalu od 0.0 do 1.0. Pokud je *x* menší než *edge0*, je návratová hodnota rovna nule, v případě kdy je *x* větší jak *edge1* je výsledek roven jedné.

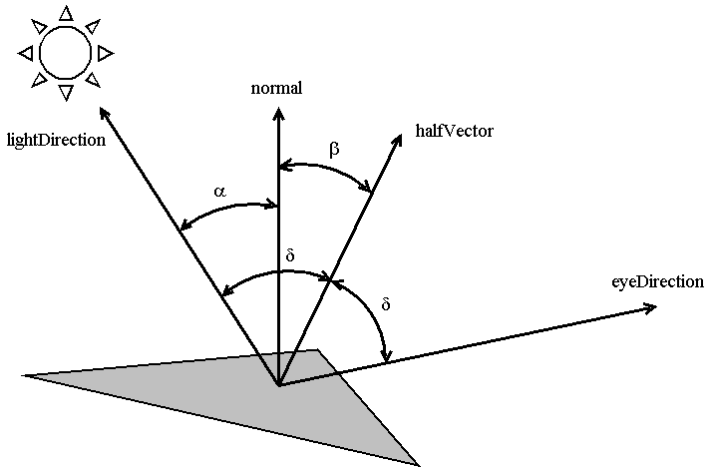
Kód vertex shaderu:

```
void main()
{
        // transformace pozice vrcholu
        vec4 position = gl_ModelViewProjectionMatrix * gl_Vertex;

        // vypocet barvy vrcholu v zavislosti na vzdalenosti od kamery
        float intensity = smoothstep(-5.0, 0.0, -position.z);

        // nastaveni vypoctene barvy
        gl_FrontColor = vec4(intensity, intensity, intensity, 1.0);

        //zapis transformovane pozice vrcholu
        gl_Position = position;
}
```

Výše uvedený kód nejprve transformuje pozici vertexu vynásobením transformační matice. Po té je vypočtena intenzita podle velikosti souřadnice *z* a nastavena výsledná barva vrcholu.

6.2 Osvětlení per-vertex

V následujícím příkladu je popsán jednoduchý světelný model. Princip jednoduchého osvětlení je založen na změně intenzity barvy vrcholu v závislosti na pozici světelného zdroje.



Obrázek 6.2.1 Ilustrace výpočtu jednoduchého osvětlení.

Tento jednoduchý model upraví barvu vrcholu podle úhlu, který svírá jeho normála a vektor směřující k světelnému zdroji. Pokud bude tento úhel nulový, bude mít vrchol svou původní barvu. Čím větší tento úhel bude, tím tmavší odstín původní barvy bude vrchol mít. Tento úhel ale není jediným faktorem ovlivňujícím výslednou barvu vrcholu. Ve skutečnosti je výsledná barva skládána z difúzní, spekulární a ambientní složky. Výsledná barva je spočítána jako součet těchto složek.

$$C = D + S + A$$

Hodnota difúzní složky je rovna součinu difúzních složek světla a materiálu objektu. Tím je určen výsledný odstín, který je upraven podle úhlu mezi normálou a vektorem směřujícím k světelnému zdroji.

$$D = D_o * D_l * \cos(\alpha)$$

Nejsložitější částí je výpočet spekuláru, lesku na povrchu objektu. Tento efekt závisí i na pozici kamery. Výsledná hodnota je opět dána součinem spekulární složky světla a materiálu objektu, která je upravena podle úhlu mezi normálou a vektorem *halfVector*. Jedná se o speciální vektor, který svírá s vektorem kamery a vektorem světelného zdroje stejný úhel. Tento koeficient je pak dále upraven podle parametru *shininess*, který je součástí definice materiálu objektu.

$$S = S_o * S_l * (\cos(\beta))^{shininnes}$$

Výpočet ambientní složky je roven součinu ambientních složek světla, materiálu objektu a globální hodnoty ambientu.

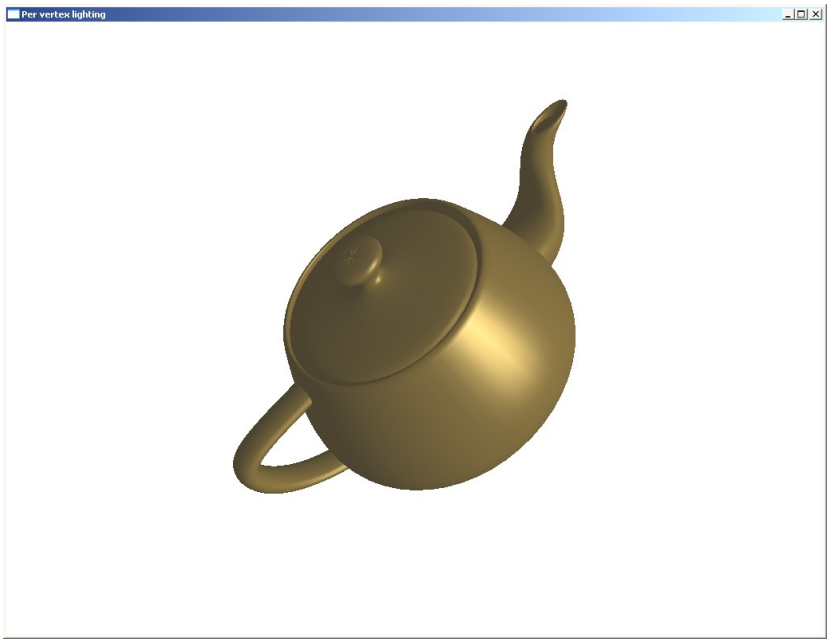
$$A = A_o * A_l * A_g$$

Kód pro vertex shader:

```
void main()
{
	// ziskani normalizovaneho vektoru udavajici smer nasvetleni objektu.
	vec3 lightDirection = normalize(gl_LightSource[0].position.xyz);

	// normala vrcholu
	vec3 normal = normalize(gl_NormalMatrix * gl_Normal);

	// vypocet intenzity pro difuzni slozku
	float intensity = dot(normal, lightDirection);
	// omezeni intenzity na interval <0-1>
	intensity = max(intensity, 0.0);

	float specIntensity = dot(normal, gl_LightSource[0].halfVector.xyz);
	specIntensity = max(specIntensity, 0.0);
	specIntensity = pow(specIntensity, gl_FrontMaterial.shininess);

	vec4 diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;;
	vec4 ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
	ambient += gl_LightModel.ambient * gl_FrontMaterial.ambient;
	vec4 specular = gl_FrontMaterial.specular * gl_LightSource[0].specular;

	// vypocet vysledne barvy
	gl_FrontColor = (diffuse * intensity) + (specular * specIntensity) + ambient;

	// transformace pozice
```

```
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

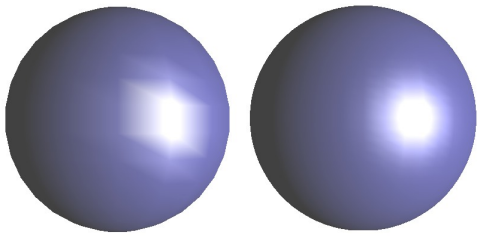
První část obsahuje výpočet vektoru směrem ke světelnému zdroji a normalizaci normály vrcholu. Pro získání pozice a barvy světelného zdroje je použita proměnná gl_LightSource. Jedná se o pole struktur, obsahující parametry všech světelných zdrojů ve scéně. V tomto případě je osvětlení počítáno pouze pro jeden světelný zdroj (*gl_LightSource[0]*). Pro výpočet intenzity se provede vektorový součin (funkce *dot*) normálového vektoru a vektoru směřujícího k světelnému zdroji. Protože jsou oba vektory normalizované, výsledná hodnota odpovídá kosinu úhlu mezi vektory. Výslednou hodnotou je tedy číslo v intevalu <-1 - 1>, kde hodnota 1 odpovídá nulovému úhlu a hodnota -1 úhlu maximálnímu (180 stupňů). Tato hodnota se omezí na interval <0 - 1> pomocí funkce *max*, která vrací vetší z čísel zadaných v parametru. Stejný postup se použije i pro výpočet spekulární složky světla, jen místo směru k světelnému zdroji je použit předpočítaný parametr *halfVector*. V poslední části jsou všechy složky sečteny a výsledná hodnota nastavena jako barva vrcholu. Ve fragment shaderu se nastaví vstupní barva na výstup bez jakýchkoliv úprav.



Obrázek 6.2.2 Výsledný efekt jednoduchého světelného modelu.

6.3 Osvětlení per-pixel

V předchozím příkladu byl popsán princip jednoduchého osvětlení, počítaného pomocí vertex shaderu. Přestože výsledný efekt vypadá věrohodně, má tato technika jednu velkou nevýhodu a to nasvětlení objektů s malým počtem vrcholů. Na následujícím obrázku je srovnání dvou modelů. Oba objekty mají stejný materiál, liší se pouze počtem vrcholů, ze kterých jsou složeny. Jejich nasvětlení je počítáno pomocí shaderu, popsaného v minulém příkladu.



Obrázek 6.3.1 Nasvětlení objektů s nízkým a vysokým počtem vrcholů

Z obrázku je patrné, že pokud bude mít model menší počet vrcholů, začne se především spekulární efekt rozpadat na jednotlivá primitiva, ze kterých je objekt sestaven. To je dáno právě tím, že osvětlení je počítáno pro vrcholy a výsledná barva je pak lineárně interpolována k dalšímu vrcholu. Proto tento model nefunguje příliš dobře u větších ploch. Řešením je tedy počítat osvětlení ne pro každý vrchol ale až na úrovni fragmentů.

Kód pro vertex shader:

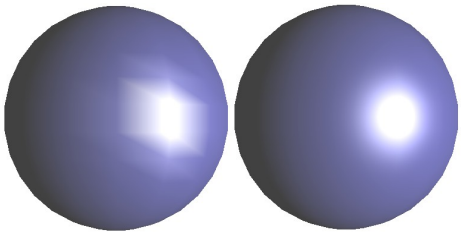
```
varying vec3 normalAtr, lightDirectionAtr, halfDirectionAtr;
varying vec4 ambientAtr, diffuseAtr, specularAtr;

void main()
{
	// ziskani normalizovaneho vektoru udavajici smer nasvetleni objektu.
	lightDirectionAtr = normalize(gl_LightSource[0].position.xyz);
	halfDirectionAtr = normalize(gl_LightSource[0].halfVector.xyz);

	// normala vrcholu
	normalAtr = gl_NormalMatrix * gl_Normal;

	// vypocet slozek barvy vrcholu.
	diffuseAtr = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
	specularAtr = gl_FrontMaterial.specular * gl_LightSource[0].specular;
	ambientAtr = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
	ambientAtr += gl_LightModel.ambient * gl_FrontMaterial.ambient;

	// transformace pozice
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Pro výpočet jsou použity naprosto stejné vzorce jako při výpočtu osvětlení popsané v předchozím příkladu, pouze výpočet je prováděn pro každý zpracovávaný fragment. Protože při vykreslení scény bývá vetšinou počet fragmentů mnohem větší než počet vrcholů, je optimálnější co nejvíce hodnot, které se mezi fragmenty nemění, vypočítat už ve vertex shaderu. Proto jsou vektor ke světelnému zdroji i *halfVector* normalizovány už zde a předávány jako parametr fragment shaderu, stejně jako výpočet odstínů jednotlivých barevných složek.

Kód fragment shaderu:

```
varying vec3 normalAtr, lightDirectionAtr, halfDirectionAtr;
varying vec4 ambientAtr, diffuseAtr, specularAtr;

void main()
{
        vec3 normal = normalize(normalAtr);

        // vypocet intenzity
        float intensity = dot(normal, lightDirectionAtr);
        // omezeni intenzity na interval <0-1>
        intensity = max(intensity, 0.0);

        float specIntensity = dot(normal, halfDirectionAtr);
        specIntensity = max(specIntensity, 0.0);
        specIntensity = pow(specIntensity, gl_FrontMaterial.shininess);

        gl_FragColor = diffuseAtr * intensity + specularAtr * specIntensity + ambientAtr;
}
```

Ve fragment shaderu je vypočtena intenzita ovlivňující difůzní a spekulární složku podle stejných vzorců jako u per-vertex osvětlení. Rozdíl mezi per-vertex a per-fragment osvětlením je znázorněn na následujícím obrázku.



Obrázek 6.3.2 Srovnání osvětlení metodou per-vertex a per-pixel

6.4 Cartoon shading

Cartoon shading je efekt, který patří do kategorie non-photorealistic effects. Účelem těchto efektů není zobrazení co nejrealističtější scény ale naopak její stylizace například do malby nebo kresby. V následujícím příkladu bude popsán jednoduchý shader, pomocí kterého lze vykreslovat objekty v komiksovém stylu.

Výsledný obraz bude složen ze dvou na sobě nezávislých efektů. Prvním z nich je obtažení obrysu objektu černou linkou. Tohoto efektu je dosaženo jednoduchým postupem v aplikaci, kdy jsou nejdříve vykresleny hrany trojúhelníků, ze kterých se objekt skládá, a které jsou natočeny směrem od kamery (zadní plochy objektu). Tím vznikne černý obrys, do kterého je vykreslen objekt znovu, tentokrát pouze s použitím předních ploch a při použití shaderu.

Následující kód je součástí funkce pro vykreslení objektu v aplikaci:

```
glLineWidth(8);
glColor3f(0.0,0.0,0.0);

glUseProgram(0);
glDisable(GL_LIGHTING);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glCullFace(GL_FRONT);
DrawObject()

glUseProgram(p);
glEnable(GL_LIGHTING);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glCullFace(GL_BACK);
DrawObject()
```

Pro vykreslení obrysu je použit fixní vertex a fragment shader, protože se jedná pouze o vykreslení černého objektu, bez potřebných efektů. Dále je nastavena tloušťka čáry a barva obrysu, je vypnuto osvětlení, nastavení vykreslovacího módu na kreslení čar, nastavení ořezávání předních ploch a vykreslen objekt. Druhá část nastaví změněné parametry zpět a vykreslí objekt znovu za použití shaderů.

Druhá část efektu obsahuje vykreslení samotné plochy objektu tak, aby měl co nejvíce komiksový vzhled tím, že pro vykreslení je použit pouze omezený počet barev. V tomto případě pouze čtyři odstíny původní barvy.

Vertex shader pouze předá potřebné hodnoty.

```
varying vec3 normalAtr;

void main()
{
        normalAtr = gl_NormalMatrix * gl_Normal;
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
        gl_FrontColor = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
}
```

Výpočet barvy fragmentu je podobný výpočtu per-pixel osvětlení, podle hodnoty intenzity je vybrán výsledný odstín.

```
varying vec3 normalAtr;

void main()
{
        vec3 normal = normalize(normalAtr);
        vec3 lightDirection = normalize(gl_LightSource[0].position.xyz);
        float intensity = max(dot(lightDirection, normal), 0.0);

        vec4 color = gl_Color;
```

```
        if (intensity > 0.7)
                color *= 1.0;
        else
        if (intensity > 0.4)
                color *= 0.8;
        else
        if (intensity > 0.1)
                color *= 0.6;
        else
                color *= 0.4;

        gl_FragColor = color;
}
```

Obrázek 6.4.1 Cartoon shading

6.5 Normal mapping

Normal mapping je jedna z metod, jak přidat na povrch objektu dodatečné detaily bez potřeby dalších vrcholů a tím zvýšení složitosti modelu. Tato metoda vytváří iluzi nerovnosti povrchu změnou vektoru normály při výpočtu osvětlení.

Při výpočtu osvětlení se podobně jako při nanášení textury zjistí barva pixelu v odpovídajícím místě speciální bitmapy nazývané normálová mapa. Barevné složky pixelu představují hodnotu normálového vektoru, který je použit při výpočtu osvětlení metodou per-pixel.

Při výpočtu osvětlení fragmentu ale nelze použít normálový vektor přesně ve formátu, ve kterém byl uložen v bitmapě. Aby bylo nasvětlení vypočteno korektně, musí být normálový vektor přepočítán do souřadného systému konkrétního fragmentu, který je zpracováván. Tento souřadný systém je definován třemi vektory (normála, binormála, tangenta). Matice pro převod do tohoto systému se nazývá TBN matice.

$$tangent = (t_x t_y t_z)$$
$$binormal = (b_x b_y b_z)$$
$$normal = (n_x n_y n_z)$$

$$TBMmatrix = \begin{pmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{pmatrix}$$

Výsledná normála pro konkrétní fragment se vypočítá jako součin TBN matice a normály načtené z bitmapy.

$$fragmentNormal = TBNmatrix * bitmapNormal$$

Výpočet normály, tangenty i binormály je záležitostí aplikace a většinou se zadává jako proměnná typu *attribute* pro konkrétní vrchol. Pro definici TBN matice jsou potřeba minimálně dva z těchto vektorů. Protože jsou vektory navzájem na sebe kolmé, může být třetí z nich vypočítán pomocí vektorového součinu.

$$tangent = normal \times binormal$$

Celý efekt je počítan ve fragment shaderu, vertex shader pouze předá potřebné hodnoty.

Kód vertex shaderu:

```
attribute vec3 binormal;
varying vec3 position;
varying vec3 n, t, b;

void main(void)
{
	// transformace pozice
	vec4 newPosition = gl_ModelViewProjectionMatrix * gl_Vertex;
	gl_Position = newPosition;

	// predani texturovacich koordinatu
	gl_TexCoord[0] = gl_MultiTexCoord0;

	// predani pozice
	position = gl_NormalMatrix * gl_Vertex.xyz;

	// vypocet TBN matice
	n = gl_NormalMatrix * gl_Normal;
	b = gl_NormalMatrix * binormal;
	t = cross(n, b);
}
```

Kód fragment shaderu:

```
uniform sampler2D difuse;
uniform sampler2D bump;

varying vec3 position;
varying vec3 n, t, b;

void main()
{
	// vypocet normaly podle hodnoty v mape
	vec3 bumpNormal = texture2D(bump,gl_TexCoord[0].st).xyz * 2.0 - 1.0;

	// definice TBN matice
	mat3 tbnMatrix = mat3(normalize(t), normalize(b), normalize(n));

	// vysledna normala fragmentu
	bumpNormal = normalize(tbnMatrix * bumpNormal);

	// vypocet smeru svetelneho zdroje
	vec3 lightDirection = normalize(gl_LightSource[0].position.xyz - position);

	// vypocet intenzity
	float intensity = dot(lightDirection, bumpNormal);

	// nacteni barvy z textury
	vec4 color = texture2D(difuse, gl_TexCoord[0].st);

	// vypocet vysledne barvy fragmentu
	gl_FragColor = color * intensity;
}
```

První částí je získání normály pro fragment z bitmapy. V tomto případě se nebude jednat o interpolovanou hodnotu předávanou z vertex shaderu ale normála je dána hodnotou v bitmapě (funkce *texture2D*). Načtený vektor je potřeba upravit, protože jeho hodnota je v intervalu 0.0 – 1.0 ale hodnota jednotlivých složek vektoru musí být v intervalu -1.0 – 1.0. Následně je spočítána TBM matice pro konkrétní fragment a načtená normála je vynásobena touto maticí. Ve zbývající části je vypočteno osvětlení metodou per-pixel.

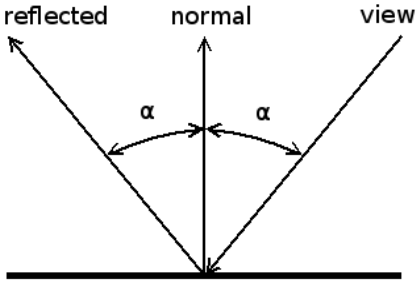
Obrázek 6.5.1 Normal mapping

6.6 Reflexe a refrakce

Tyto dva efekty jsou založeny na principu lomu světelného paprsku při přechodu z jednoho prostředí do druhého. V běžném životě se nejčastěji objevují ve formě odrazů (reflexe) a lomu světla (refrakce).

Reflexe

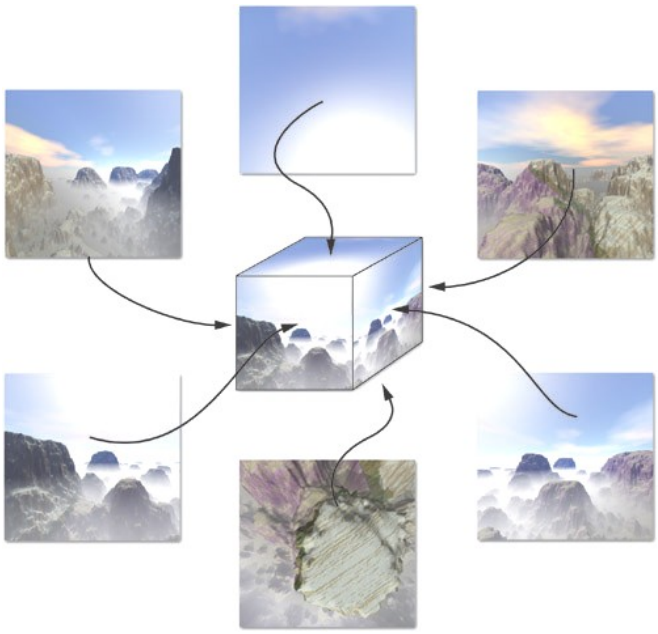
V praxi je nejčastějším příkladem reflexe zrcadlo nebo jakýkoliv lesklý předmět, ve kterém se odráží jeho okolí. Celý efekt lze popsat jednoduchou rovnicí znázorněnou na obrázku 6.6.1.



Obrázek 6.6.1 Výpočet odrazu světla

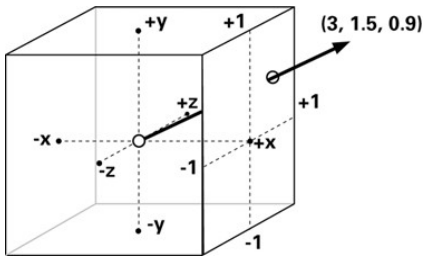
Úhel pod kterým dopadá světelný paprsek na povrch objektu je roven úhlu paprsku odraženého. V případě shaderů lze tento vzorec jednoduše implementovat pomoci fragment shaderu. K výpočtu je potřeba znát směr paprsku (směr od kamery), normálu v místě dopadu a místo, kam odražený paprsek dopadne do scény. Předat do shaderu první dva parametry není složité,

komplikace ale nastávají s místem dopadu odraženého paprsku. Jak plyne z omezení shaderů, v programu není možnost přistupovat k okolním fragmentům a dostupná není ani jakákoliv informace o okolní geometrii nebo ostatních objektech ve scéně. Proto se využívá speciálních textur zvaných cube maps (krychlové mapy). Tento typ textury se sestává z na sebe navazujících šesti čtvercových textur, které dohromady tvoří krychli (viz. Obrázek 6.6.2).



Obrázek 6.6.2 Cube map

Stěny této krychle představují pohled na okolí objektu ve všech směrech a proto se v tomto případě také někdy označuje jménem *environment map*. Práce s tímto typem textury se liší od klasické 2D textury. Nejzásadnější rozdílem je mapovaní a orientace v této textuře. U klasické textury je v fragment shaderu k dispozici funkce *texture2D*, které se do parametru předají souřadnice ve formě dvourozměrného vektoru a funkce vrací barvu pixelu v zadaném místě textury. Pro práci s krychlovou mapou je k dispozici příkaz *textureCube,* kterému se předá trojrozměrný vektor určující směr od středu pomyslné krychle. Výsledná barva, vrácená touto funkcí, je brána z místa, kde se polopřímka ve směru daném vektorem od středu krychle protne se stěnou krychle (obr. 6.6.3).



Obrázek 6.6.3 Princip texturovacích souřadnic u krychlových map

Před samotným výpočtem je potřeba si uvědomit jednu podstatnou věc. Použitá cube mapa zobrazuje okolí objektu a proto texturovací koordináty pro získaní barvy musí být ve světových

souřadnicích. Normála vrcholu a jeho pozice je však v souřadném systému objektu. Proto je potřeba obě tyto hodnoty do světového souřadného systému převést pomocí transformační matice. Standardně není matice pro tento převod v shaderech k dispozici a proto musí být z hlavního programu předána v podobě *uniform* parametru. Jakmile jsou oba parametry převedeny do světových souřadnic, provede se výpočet odrazu pomocí příkazu *reflect*, který je v GLSL k dispozici a vypočtený směr se předá do fragment shaderu. Výsledný směr je pak použit k získaní výsledné barvy z krychlové mapy funkcí *textureCube*.

Samotný výpočet odrazu se dá provést buď ve vertex shaderu a výsledný směr paprsku předat do fragment shaderu nebo počítat odraz pro každý fragment a z vertex shaderu pouze předávat potřebnou normálu a směr ke kameře ve světových souřadnicích. Výpočet ve vertex shaderu bude rychlejší, ale ve výsledku bude méně přesnější kvůli interpolaci odraženého paprsku. Následující kód představuje přesnější variantu kdy výpočet odrazu se provádí ve fragment shaderu.

Kód pro vertex shader:

```
vvarying vec3 worldViewAtr;
varying vec3 worldNormalAtr;
uniform vec3 camera;
uniform mat3 worldMatrix;

void main()
{
	// vektor smerem ke kamere ve svetovych souradnicich.
	worldViewAtr = worldMatrix*gl_Vertex.xyz - camera;
	// normalovy vektor ve svetovych souradnicich
	worldNormalAtr = worldMatrix*gl_Normal.xyz;

	// transformace pozice
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

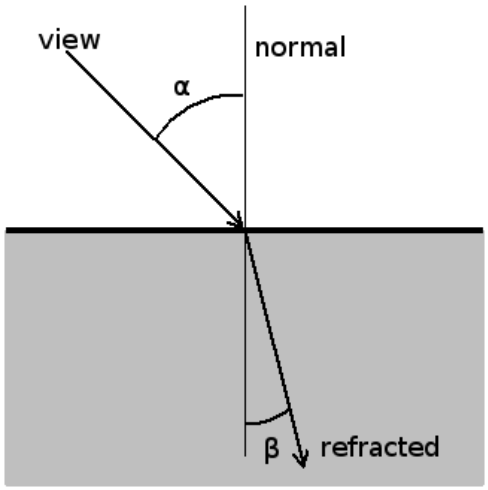
Kód pro fragment shader:

```
varying vec3 worldViewAtr;
varying vec3 worldNormalAtr;

uniform samplerCube texture;

void main()
{
	// ziskani vysledneho vektoru podle vektoru dopadu a normaloveho vektoru
	vec3 reflected = reflect(worldViewAtr, normalize(worldNormalAtr));

	// ziskani vysledne barvy z cube mapy.
	vec4 texColor = textureCube(texture, reflected);

	// nastaveni vysledne barvy fragmentu
	gl_FragColor = texColor;
}
```

Obrázek 6.6.4 Reflexe

Refrakce

Princip tohoto efektu vzjadřuje Snellův zákon, který popisuje šíření vlnění (v tomto případě světelného paprsku) při přechodu z jednoho prostředí do druhého. Při přechodu se paprsek zlomí a velikost lomu závisí na poměru indexů lomu obou prostředí, jak ukazuje obrázek 6.6.5.



Obrázek 6.6.5 Snellův zákon

Matematická rovnice popisující lom má následující tvar:

$$n1 * \sin(\alpha) = n2 * \sin(\beta)$$

V této rovnici *n1* a *n2* představuje index lomu jednotlivých prostředí, úhly α a β jsou jsou počítány od normály kolmé k povrchu přechodu. Může ovšem i nastat situace, kdy výsledný úhel β bude větší jak 90º a paprsek se odrazí zpět do prvního prostředí. Tato situace se nazývá totální odraz.

Implementace refrakce je totožná s reflexí, jediný rozdíl nastává v místě, kde se u reflexe počítal směr odraženého paprsku. V případě refrakce je výsledný vektor počítán podle Snellova zákona. Podobně jako pro reflexi je pro výpočet refrakce k dispozici funkce *refract*. Tato funkce má tři parametry: vstupní vektor, normálový vektor a poměr indexů lomů obou prostředí. Funkce ale nepokrývá případy, kdy dochází k totálnímu odrazu a vrací pro tyto situace nulový vektor. Proto je potřeba tyto případy zvlášť v shaderu ošetřit nebo provést výpočet Shnellova zákona bez pomoci této funkce. V praxi se spolu s refrakcí počítá i reflexe a pro případ totálního odrazu se vrací vektor spočítaný funkcí *reflect*.

Případy, kdy došlo k totálnímu odrazu, se dají spočítat úpravou Snellova zákona.

$$\frac{\sin(\beta)}{\sin(\alpha)} = \frac{n2}{n1}; \frac{n2}{n1} = r$$

$$\frac{\sin(\beta)^2}{\sin(\alpha)^2} = r^2$$

$$\frac{1-\cos(\beta)^2}{1-\cos(\alpha)^2} = r^2$$

$$\cos(\beta)^2 = 1 - r^2(1-\cos(\alpha)^2)$$

K totálnímu odrazu dochází, pokud je β >= 90º, tedy případy kdy platí cos(β) <= 0.

$$1 - r^2(1-\cos(\alpha)^2) \leq 0$$

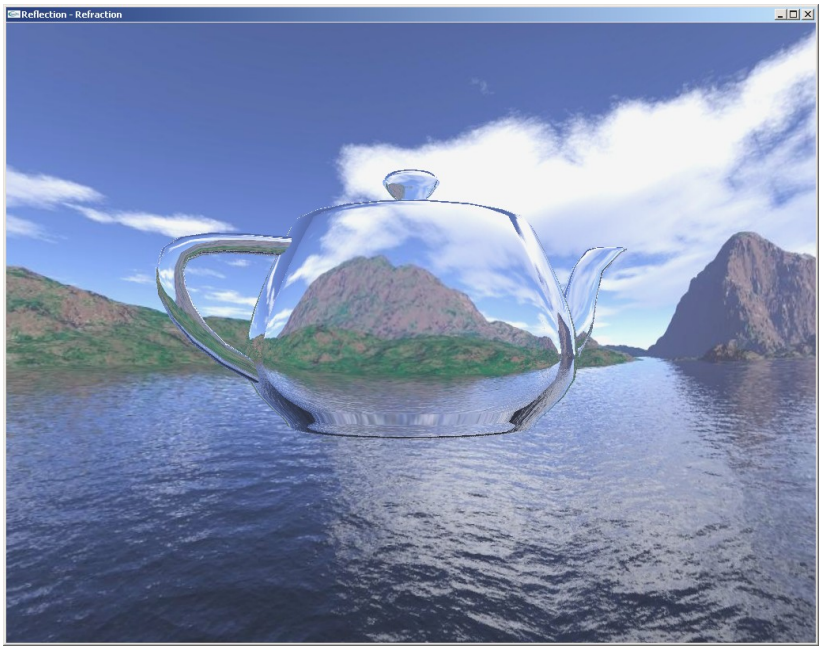
Pokud platí tato nerovnice, je paprsek odražen zpět a musí se počítat jako odraz pomocí funkce *reflect*.

Kód pro vertex shader je shodný pro reflexi i refrakci.

Kód pro fragment shader:

```
varying vec3 worldViewAtr;
varying vec3 worldNormalAtr;

uniform samplerCube texture;

// pomer indexu lomu obou prostredi
const float ratio = 1.1;
```

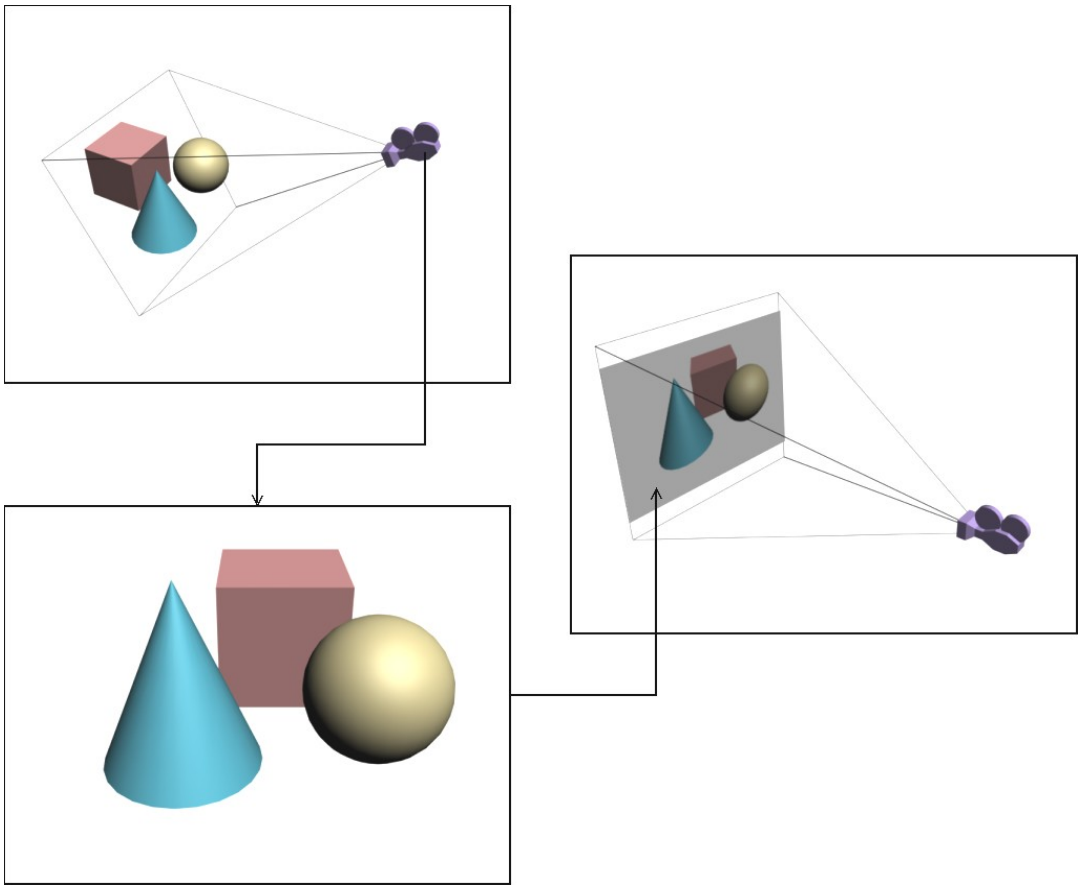
```
void main()
{
	// oba vektory musi byt normalizovany
	vec3 worldView = normalize(worldViewAtr);
	vec3 worldNormal = normalize(worldNormalAtr);

	// vypocet cos beta pro nalezeni pripadu totalniho odrazu
	float cosa = dot(-worldView, worldNormal);
	float cos2b = 1.0 - (ratio * ratio *(1.0 - cosa * cosa));

	// vypocet vysledneho smeru svetelneho paprsku
	vec3 refracted;
	if(cos2b > 0.0)
		refracted = refract(worldView, worldNormal, ratio);
	else
		refracted = reflect(worldView, worldNormal);

	// ziskani vysledne barvy z cube mapy
	vec3 refractColor = textureCube(texture, refracted).rgb;

	// nastaveni barvy fragmentu
	gl_FragColor = vec4(refractColor, 1.0);
}
```



Obrázek 6.6.6 Refrakce

6.7 Efekty post-processingu

Post-processing efekty je skupina efektů, které jsou aplikovány na výslednou scénu až před finálním zobrazením. Jedná se většinou o 2D transformace počítané těsně před zobrazením výsledné scény. Tyto efekty jsou počítány v fragment shaderu a často je potřeba při jejich výpočtu znát hodnoty okolních fragmentů. Tato funkce není standardně u fragment shaderu k dispozici a proto se pro tento druh efektů používá zvláštní technika. Celá scéna se nevykreslí do frame bufferu jako obvykle, ale do klasické 2D textury. Při vykreslení výsledného obrazu je do prázdné scény umístěn jeden obdélník, který z pohledu kamery přesně vyplňuje celý pohled. Na tento obdélník je aplikována textura s původní scénou a fragment shader s post-processing efektem. Základní princip je naznačen na obrázku 6.7.1. Celý proces vykreslení a uložení scény do textury je záležitostí aplikace. V OpenGL je pro tento účel k dispozici funkce *glCopyTexImage2D*, případně rozšíření Frame Buffer Object.



Obrázek 6.7.1 Princip post-processing efektu

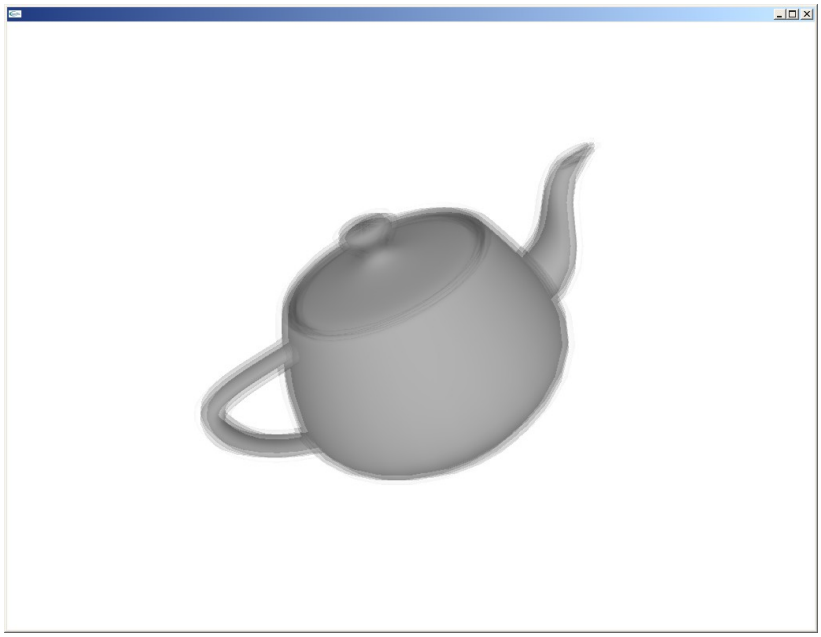
Následující příklad demonstruje použití post-processingu pro efekt rozostření výsledného obrazu. Jak bylo popsáno výše, fragment shader s výpočtem rozostření bude aplikován až v závěru vykreslovací funkce. V první fázi se za použití standardního vertex a fragment shaderu vykreslí celá scéna a výsledek je uložen do textury. V druhé části se zapne shader s výpočtem rozostření a před kameru se vykreslí obdélník s texturou scény.

Pro výpočet výsledné barvy fragmentu je brána v úvahu barva okolních fragmentů. Výsledná barva je spočítána jako průměrná barva původního a v tomto případě okolních čtyř fragmentů.

Program pro fragment shader:

```
uniform sampler2D tex;

void main()
{
	// seznam pozic okolnich vzorku
	vec2 samples[4];
	samples[0] = vec2(-0.005, 0.0);
	samples[1] = vec2( 0.005, 0.0);
	samples[2] = vec2( 0.0,-0.005);
	samples[3] = vec2( 0.0, 0.005);

	// nacteni puvodni barvy fragmentu
	vec4 color = texture2D(tex, gl_TexCoord[0].st);

	// pricteni barev ctyr okolnich fragmentu
	for(int j=0;j<4;++j)
		color += texture2D(tex, gl_TexCoord[0].xy + samples[j] );

	// podeleni vysledku poctem vzorku (puvodni + 4x okolni)
	color = color / 5.0;

	// nastaveni vysledne barvy fragmentu.
	gl_FragColor = Color;
}
```

Tento příklad reprezentuje nejjednodušší způsob implementace rozostření. V praxi se používají složitější techniky pro aplikaci tohoto efektu. Většinou je výsledná barva fragmentu počítána z více vzorků nebo na několik průchodů, kdy je výsledná scéna opět vykreslena do textury a celý efekt aplikován opakovaně.



Obrázek 6.7.2 Rozostření pomocí post-processingu

Kapitola 7

Příklady složitějších efektů

Následující kapitola obsahuje příklady použití shaderů pro tvorbu složitějších efektů, většinou se jedná o spojení dvou nebo více již dříve popsaných shaderů. Vzhledem k větší složitosti a délce zdrojového kódu bude u každého příkladu popsán jen základní princip efektu. Všechny příklady jsou pak podrobně komentovány přímo ve zdrojovém kódu, který je k práci přiložen.

7.1 Hloubkové rozostření scény

Jedná se o klasický efekt, kdy objekty mimo prostor ostření jsou zobrazeny rozostřeně. V principu se jedná o zaostření na ty objekty ve scéně, které jsou vzdáleny od kamery na předem danou vzdálenost. Čím blíže nebo dále od této hranice se objekt nachází, tím více je vidět jako rozostřený.

V případě implementace pomocí shaderů se dají použít dva již popsané efekty a to hloubková mapa a efekt rozostření. Celý princip spočívá v úpravě algoritmu pro výpočet rozostření z minulé kapitoly. Kromě textury se scénou však bude potřeba do shaderu předat i hloubkovou mapu scény z pohledu kamery, podle které se bude určovat, které fragmenty mají být rozostřeny a které nikoliv.

Obrázek 7.1.1 Hloubková mapa scény

Na obrázku 7.1.1 je zobrazena upravená hloubková mapa testovací scény. Černá barva představuje vzdálenost, na kterou je zaostřeno. Čím světlejší barva v hloubkové mapě, tím bude fragment více rozostřen. Při výpočtu rozostření je pak brána v úvahu nejen barva ale i hloubka v místě fragmentu.

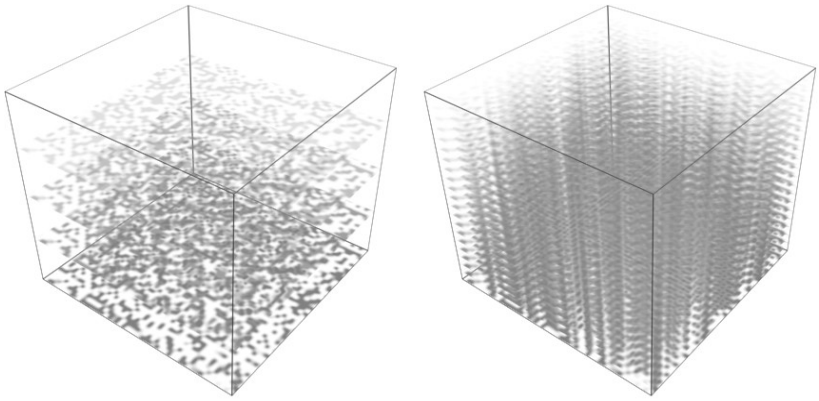
Obrázek 7.1.2 Rozostření pomocí post-processingu

7.2 Efekt chlupů a vlasů

Zobrazení chlupů a vlasů, například zvířecí srsti, je velmi náročný efekt. Aby byl výsledný dojem co nejvíce podobný realitě, je potřeba zobrazovat velké množství jednotlivých vláken. Použít pro každý tento vlas vlastní geometrii (trojúhelník nebo polygon) není z hlediska výkonu reálné, protože počet vlasů i na menším objektu může stoupnout do řádu několika stovek tisíců vláken. Přesto ale existují metody, jak tento efekt úspěšně napodobit.

První podmínkou je zobrazení dostatečných detailů při zachování minimálních požadavků na množství ploch potřebných pro vykreslení. Celý efekt je založen na principu vrstvených textur, kdy je celý objekt opakovaně vykreslován se speciální texturou šumu. Důležitou podmínkou pro správnou funkci je přítomnost alfa kanálu u této textury. Při každém dalším vykreslení je objekt o malý kousek zvětšen spolu se zvýšením průhlednosti textury šumu. Následující obrázek 7.2.1 ukazuje tento princip použitý na jedné čtvercové ploše.

V případě že bude mezi vrstvami dostatečně malá vzdálenost, budou fragmenty jednotlivých vrstev nad sebou vytvářet jednotlivá vlákna. Počet vrstev potřebných pro simulaci tohoto efektu, je závislý na požadované délce jednotlivých vláken, druhu použité textury a vzdálenosti, z jaké je objekt pozorován. V případě, že mezi vrstvami bude vzdálenost příliš velká, budou mezi fragmenty jednotlivých vrstev viditelné mezery. S přibývajícím počtem vrstev bude efekt vypadat reálněji ale každá další vrstva představuje další vykreslení celého objektu a tím i vzrůstající počet ploch ve scéně.



Obrázek 7.2.1 Princip vrstvené textury

Při vykreslení scény musí program vykreslit objekt opakovaně a do shaderů předat informaci o tom, pro kterou vrstvu je právě model vykreslován. Ve vertex shaderu se posune pozice vrcholu ve směru normály a tím se objekt zvětší. Velikost zvětšení je dána pořadím právě vykreslované vrstvy. Zde může být vypočteno i jednoduché osvětlení a jeho výsledek předán do fragment shaderu. Ve fragment shaderu se načte z textury šumu především hodnota alfa kanálu (průhlednosti), která je pro celý efekt klíčová. Barva fragmentu pak může být načtena z druhé textury nebo zvolena podle vrstvy, ve které se fragment nachází. Odstín fragmentu je následně upraven podle intenzity osvětlení, vypočtené ve vertex shaderu. Celkový efekt je pak možné ještě doplnit o simulaci větru nebo jiných vnějších vlivů upravením pozice při zvětšování objektu ve vertex shaderu. Výsledek pak může vypadat jako na obrázku 7.2.2.

Obrázek 7.2.2 Simulace chlupů

7.3 Simulace reálných objektů

Posledním příkladem této práce je ukázka použití shaderů pro zobrazení konkrétního objektu z běžného života, v tomto případě automobilu. V praxi by mohl být tento program použit například pro prezentační nebo reklamní účely.

Postupů, jak dosáhnout co možná nejvěrnějšího zobrazení, je velmi mnoho. Následující postup popisuje pouze jednu z možností, jak se k výslednému efektu přiblížit. Nejsložitějším prvkem automobilu je jeho karoserie. Pro její zobrazení je potřeba spojit hned několik efektů.

1. Prvním z nich je efekt zvaný fall-off. Jedná je o změnu odstínu podle úhlu pohledu. Při pohledu kolmo na povrch objektu bude mít lak svou barvu, ale při pohledu z většího úhlu se jeho odstín změní. Výpočet tohoto efektu není příliš náročný a je velmi podobný výpočtu osvětlení.
2. Odraz okolí. Obzvláště v nové karoserii se velmi dobře odráží okolí automobilu. Tento efekt opět není náročné implementovat pomocí krychlové mapy.
3. Poslední aplikovaný efekt je spekulární odraz. Přidá na karoserii automobilu světlá místa a zdůrazní především detaily na povrchu. Opět se dá použít jednoduchý výpočet použitý v příkladu pro výpočet osvětlení.

Pomocí těchto tří efektů lze vykreslit reálně vypadající karoserii. Co se týká ostatních částí automobilu, dají se pro jejich vykreslení použít jen některé z těchto efektů (například pouze odrazy pro chromované části) a proto je potřeba shader navrhnout tak, aby šla kterákoliv část při výpočtu vynechat.

Důležitá část scény, která nebyla zmíněna, je osvětlení. Až na spekulární odraz není v tomto příkladu žádné osvětlení počítáno. Místo toho jsou použity předem vypočítané stínové mapy. Jedná se o klasické textury, vygenerované k modelu pomocí externího programu. Textury přidávají na model tmavá místa tam, kam například dopadá stín od jiného modelu. Výhodou těchto map je velmi malá náročnost a velmi reálný vzhled. Hlavní nevýhoda této metody je použití pouze na statických modelech se statickým osvětlením.
Pro dodání větší hloubky je na výsledný obraz použit efekt hloubkového rozostření na tu část automobilu, která je nejvíce vzdálená od kamery.

Obrázek 7.3.1 Příklad zobrazení automobilu

Závěr

Celá práce je zpracována jako učební materiál pro výuku shaderů. V první části jsou popsány typy shaderů, jejich funkce a možnosti při vykreslování scény. Následující kapitoly obsahují stručný popis dostupných jazyků pro jejich programování. Poslední, největší část práce, je věnována praktickým ukázkám. Popsán je postup pro přeložení shaderů přímo v kódu aplikace, podrobnější popis jazyku GLSL a praktické aplikace shaderů pro výpočet efektů. Poslední kapitola práce je věnována popisu složitějších efektů. Všechny popsané příklady jsou k dispozici i ve formě podrobně komentovaných zdrojových kódů na CD, které je součástí práce.

V budoucnu by mohla být práce ještě rozšířena o další sadu složitějších příkladů, případně ukázek využití shaderů pro negrafické výpočty.

Obsah přiloženého CD

Součástí práce je i přiložené CD, obsahující všechny příklady popsané v této práci. Příklady jsou dostupné ve formě souboru projektů pro vývojové prostředí *Microsoft Visual Studio 2005*. Jednotlivé příklady jsou uloženy v adresáři *Examples*, kde každý příklad má samostatný podadresář, obsahující zdrojové kódy, potřebné datové soubory a přeložený binární soubor pro spuštění bez potřeby vývojového prostředí. V adresáři *Utils* je umístěna pomocná knihovna, vytvořená konkrétně pro tuto práci. Kvůli přehlednosti kódu jednotlivých příkladů jsou zde umístěny společné funkce například pro načítání bitmap nebo modelů ze souborů.

Každý příklad po spuštění zobrazuje okno se scénou a textové okno, do kterého se vypisuje informace o ovládání a aktuální stav volitelných parametrů. Pro spuštění příkladů je dostačující grafická karta s podporou DirectX 9.0 a vyšší. Příklady byly odladěny a testovány na grafických kartách ATI Radeon 9700 PRO, ATI Radeon X1400, ATI Radeon X1600 a NVIDIA GeForce 9600GT.

Literatura

[1] Randima Fernando, Mark J.Kilgard : The Cg Tutorial - The Definitive Guide to Programmable Real-Time Graphics, NVIDIA Corporation, Addison-Wesley, 2003

[2] Fernandes, António Ramires : GLSL Tutorial Dokument dostupný na URL http://www.lighthouse3d.com/opengl/glsl/index.php?intro (prosinec 2008)

[3] DirectXTutorial.com. Dokument dostupný na URL: https://www.directxtutorial.com/index.aspx (prosinec 2008)

[4] The OpenGL Extension Wrangler Library. Dokument dostupný na URL http://glew.sourceforge.net/ (prosinec 2008)

[5] Itskov, Costa; Wong, Rick : Two-kings. Dokument dostupný na URL: http://www.two-kings.de/tutorials/dxgraphics/ (prosinec 2008)

[6] Fur Effects - Teddys, Cats, Hair .... : Dokument dostupný na URL: http://www.xbdev.net/directx3dx/specialX/Fur/index.php (prosinec 2008)