VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

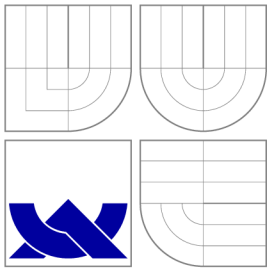
# VIZUÁLNÍ EFEKTY VE 3D APLIKACÍCH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

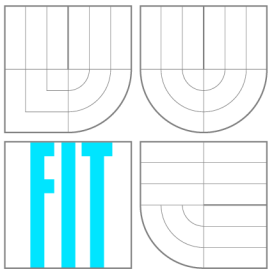
AUTOR PRÁCE
AUTHOR

Bc. MARTIN DUŽÍ

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# VIZUÁLNÍ EFEKTY VE 3D APLIKACÍCH

VISUAL EFFECTS IN 3D APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN DUŽÍ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN NAVRÁTIL

BRNO 2014

# Abstrakt

Tato diplomová práce se zabývá tvorbou vizuálních efektů v 3D grafických aplikacích. Předpokládá vykreslení scény metodou rasterizace pomocí knihovny OpenGL. Teoretická část popisuje několik vybraných efektů a následně analyzuje přístup používaný pro jejich implementaci. Dále se práce zaměřuje na vývoj softwarové knihovny s účelem usnadnění procesu programování efektů. Výsledná knihovna redukuje čas i znalosti potřebné ke tvorbě efektů. Provádí automatické generování kódu shaderů. Podstatnou funkci představuje také možnost kombinovat definice efektů do jediného celku.



# Abstract

This master's thesis deals with the creation of visual effects in 3D graphics applications. Rendering scenes using rasterization method and OpenGL library is assumed. The theoretical part describes several selected effects and then analyzes the approach used for their implementation. It focuses on the principles of rendering passes. Subsequently, the thesis focuses on the development of a software library which aims to simplify the process of programming effects. The resulting library reduces the time and knowledge required for the creation of effects. Automatic shader code generation is performed. Substantial feature is also the possibility to combine definitions of effects into a single unit.



# Klíčová slova

3D aplikace, 3D grafika, vizuální efekt, vykreslovací průchod, vykreslování, shader, OpenGL, GLSL

# Keywords

3D application, 3D graphics, visual effect, rendering pass, rendering, shader, OpenGL, GLSL

# Citace

Martin Duží: Vizuální efekty ve 3D aplikacích, diplomová práce, Brno, FIT VUT v Brně, 2014

# Vizuální efekty ve 3D aplikacích

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Navrátila. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Duží
18. května 2014

## Poděkování

Na tomto místě bych rád poděkoval Ing. Janu Navrátilovi, který mě vždy navedl správným směrem a poskytl mi mnoho podnětů užitečných pro dokončení této práce.

# Obsah

# Kapitola 1

# Úvod

Jedním z hlavních cílů počítačové grafiky se od počátku stala tvorba vizuálně přitažlivých aplikací. Za jedny z technologicky nejvyspělejších programů dnešní doby lze považovat hry, simulace a vizualizace. Jejich společným účelem je kvalitní prezentace 3D scény. Pod rukama výtvarníků vznikají detailní modely, které definují tvar a podobu zobrazovaných objektů. Až při vykreslování však dostáváme celkový dojem o výsledném obraze. Abychom zajistili požadovaný vzhled scény, přidáváme do procesu zpracování grafiky vizuální efekty. Těmi se snažíme přiblížit fotorealismu nebo naopak nerealismu. Příkladem může být osvětlení, stíny, mlha aj. Vzhledem k tomu, že lidský zrak přijímá podněty z dopadajícího světelného záření, často simulujeme efekty související právě s šířením světla. Konečný výběr samozřejmě závisí na zaměření aplikace.

Z pohledu výkonnosti a velikosti paměti jsou dnešní grafické karty na značně vyšší úrovni, než tomu bylo ještě před několika lety. Současně došlo k rozvoji možností pro programování vykreslovacího řetězce. Můžeme si tak dovolit implementovat vizuální efekty ve větší kvalitě, množství a počtu kombinací. S pomocí paralelní architektury grafického adaptéru lze snadněji splnit požadavky na rychlost, které kladou aplikace běžící v reálném čase. Implementace grafických algoritmů není přímočará. Obvykle vyžaduje napsání většího množství kódu, přestože některé jeho části se mohou opakovat. Sloučení různých efektů musíme uvažovat předem. Také je nutné zahrnout kód pro další aspekty programu, například načítání modelů scény ze souborů. Tento proces tedy přináší množství práce, a to i v případech, kdy nás zajímá pouze samotný efekt.

Cílem této práce je vytvoření nástroje pro usnadnění implementace vizuálních efektů v 3D aplikacích běžících v reálném čase. Zaměřuje se na práci s knihovnou OpenGL a programování shaderů v jazyce GLSL (více informací lze nalézt v [23, 22]). Obsahuje analýzu efektů, které se dnes často vyskytují v grafických programech. Dále nabízí řešení v podobě softwarové knihovny (inspirace pro vývoj kódu čerpána z [16]).

Kapitola 2 popisuje principy a implementaci vybraných efektů. Kapitola 3 shrnuje teoretické poznatky získané analýzou efektů popsaných dříve. Kapitola 4 se zabývá návrhem knihovny, definuje její nezbytné součásti a funkčnost. Kapitola 5 popisuje schopnosti implementace a využité algoritmy. Kapitola 6 se zabývá testováním implementace, ověřením a hodnocením funkčnosti. Také uvádí možnosti dalšího vývoje. Kapitola 7 shrnuje dosažené výsledky.

# Kapitola 2

# Efekty v 3D grafice

Pod pojmem vizuální efekt uvažujme algoritmus nebo metodu, po jejichž aplikaci očekáváme odlišný dojem z vykreslené scény. Následující podkapitoly popisují vybrané představitele efektů.

## 2.1 Osvětlení a stíny

Abychom dostali z vykreslované geometrie více než samotnou barvu objektu, musíme začít řešit osvětlení. Do scény proto přidáváme světelné zdroje. Ty mohou být statické i dynamické. Každý zdroj působí na své okolí zářením o určité intenzitě, která zároveň nese barevné spektrum (nejčastěji RGB). Využíváme 3 základní typy světelných zdrojů:

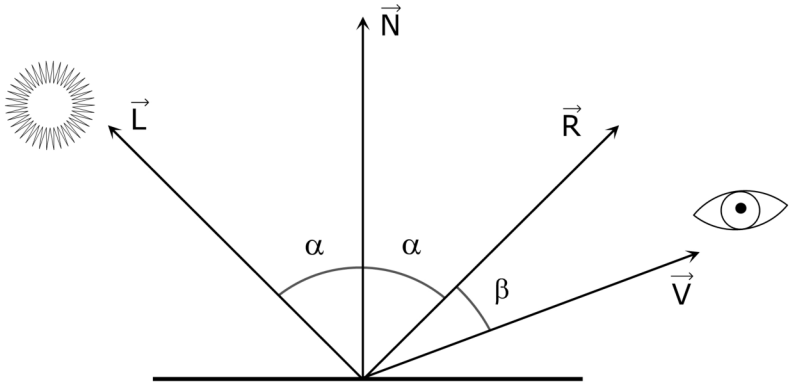
- **Směrové světlo** (angl. directional light) aproximuje zdroj, který je teoreticky nekonečně daleko (např. Slunce). Pro všechny body povrchu jsou směr a intenzita příchozího záření stejné. Z hlediska implementace se jedná o nejjednodušší případ.
- **Bodové světlo** (angl. point light) má určenu svou pozici a generuje záření do všech směrů. Intenzita pak klesá se vzdáleností povrchu od zdroje[1]. Jako příklad si můžeme představit lampu osvětlující setmělou ulici.
- **Spotlight** se podobá bodovému světlu, ale jeho oblast působení je omezena tvarem definovaným nějakým tělesem. Nejčastěji se jedná o kužel, proto se tento typ zdroje připodobňuje k baterce. Záření je nasměrováno na bod[2], který se nachází ve středu oblasti vymezené daným tělesem. Intenzita pak obvykle klesá na základě vzdálenosti od spojnice tohoto bodu s pozicí světla.

Pro účely implementace dále volíme tzv. *osvětlovací model*. Ten určuje, jakým způsobem bude vypočítána hodnota osvětlení pro každý bod povrchu. Fyzikální modely jsou schopny korektně popsat distribuci světla v prostoru. Častěji se ale používají modely empirické, které byly navrženy s ohledem na rychlost a přijatelný výsledek. Pravděpodobně nejznámější je *Phongův model* [20], jehož kompletní řešení vyjadřuje vykreslovací rovnice:

$$I = K_e + K_a I_a + \sum_{i=1}^{N} \{K_d L_d(i)(\overrightarrow{N} \cdot \overrightarrow{L}(i)) + K_s L_s(i)(\overrightarrow{V} \cdot \overrightarrow{R}(i))^n\} \tag{2.1}$$

[1] Hovoříme o tzv. útlumu světla, angl. *attenuation*
[2] V angl. užíváme výraz *spot* – proto spotlight

Pro daný bod získáme intenzitu $I$ světelného záření odraženého směrem k pozorovateli. Ve výpočtu figurují 4 vektory znázorněné na obrázku 2.1. $\overrightarrow{N}$ zastupuje normálový vektor (normálu), který je vždy kolmý k povrchu a určuje jeho orientaci. $\overrightarrow{L}$ směřuje ke světlu, zatímco $\overrightarrow{V}$ k pozorovateli. $\overrightarrow{R}$ představuje směr úplného odrazu světla od povrchu.




Obrázek 2.1: Princip Phongova osvětlovacího modelu.

Parametr $n$ se nazývá *shininess* a kontroluje matnost materiálu. Konstanty $K$ a $L$ značí barvy materiálu a světelného spektra. Celková intenzita se skládá ze 4 složek, které jsou v rovnici 2.1 vyjádřeny indexy $e$, $a$, $d$, $s$:

- **Emisní složka** označuje světlo vyzářené samotným objektem. Můžeme ji využít pro simulaci jevů fluorescence nebo fosforescence. Tato hodnota se často nevyužívá, protože osvětlení řešíme v osamocených bodech a nedosáhneme tak žádaného efektu záře v okolí předmětu[3].
- **Ambientní složka** simuluje světelné záření odražené od jiných povrchů. Úroveň intenzity ovládá člen $I_a$. Aproximace pomocí konstanty není přesná, ale zabrání skrytí částí objektů, které se nacházejí mimo oblast působnosti světelných zdrojů.

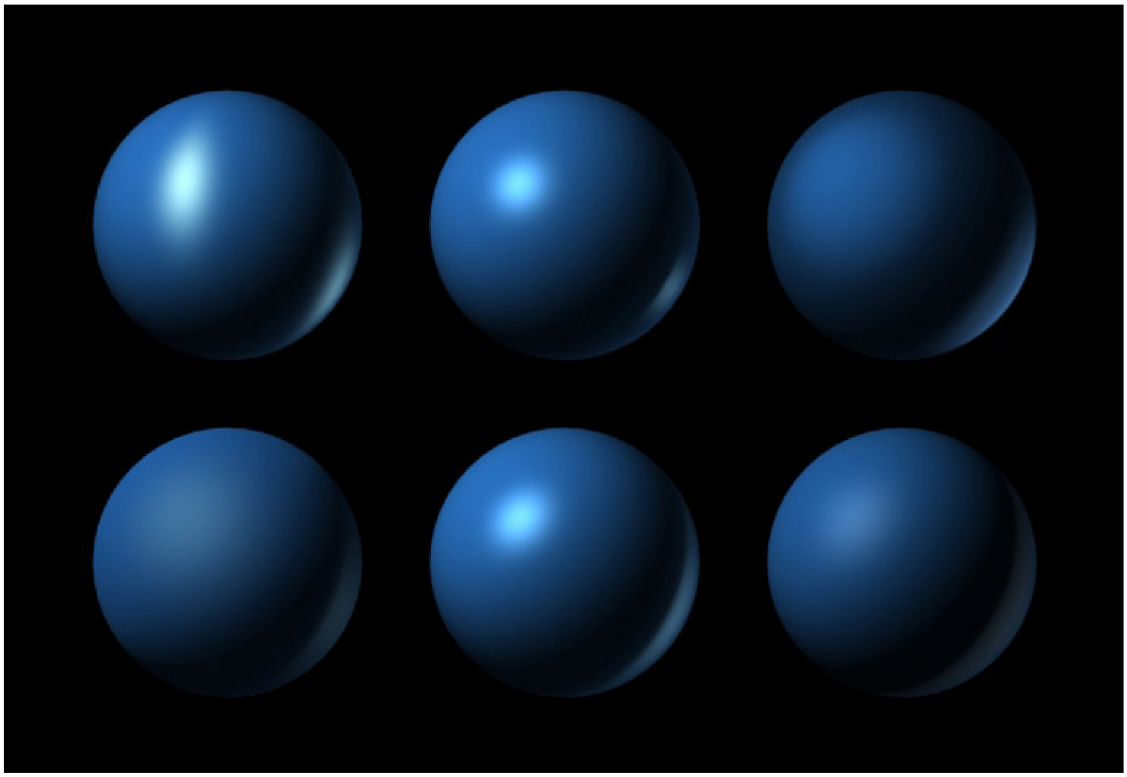
Následující složky jsou započteny pro každý světelný zdroj ve scéně:

- **Difúzní složka** zahrnuje světelné záření odražené pod povrchem. Směr odrazu je náhodný. Při interakci s přítomnými částicemi dochází k částečné absorbci a světlo tak přejímá barvu objektu. Podle *Lambertova zákona* [18] míra odraženého záření klesá s rostoucím úhlem vzhledem k normálovému vektoru, což implementuje skalární součin $\overrightarrow{N} \cdot \overrightarrow{L}$ v rovnici 2.1.
- **Spekulární složka** narozdíl od difúzní uvažuje odraz na povrchu. Směr je určen dle zákona odrazu[4]. Míra sledovaných odlesků závisí na pozici pozorovatele a matnosti materiálu. Světelné záření nevstupuje pod povrch objektu, proto si výsledný efekt často zachová barvu světla (výjimku tvoří např. kovy, které absorbují světlo na svém povrchu).

[3] Pro simulaci lze využít *glow efekt* (viz 2.3)
[4] Také nazýváme zrcadlový odraz

Pro výpočet osvětlení tedy musíme znát informace o povrchu objektu, světelných zdrojích a pozici pozorovatele.

Osvětlovacích modelů samozřejmě existuje větší množství. Liší se v implementaci difúzní nebo spekulární složky (ambientní a emisní jsou zde konstanty). *Blinn* [3] navrhuje ve spekulárním výrazu nahradit operandy skalárního součinu za normálu a normalizovaný vektor $\overrightarrow{H} = \overrightarrow{L} + \overrightarrow{V}$. Výsledkem budou pozvolnější odlesky. *Cook* a *Torrance* [5] se zaměřili na přesnější definici celého spekulárního výrazu. Jejich model produkuje kvalitnější osvětlení kovových a plastových objektů, ovšem za cenu zvýšené výpočetní náročnosti. *Oren* a *Nayar* [18] vytvořili modifikaci Lambertova modelu pro zobrazení drsných povrchů, jako např. beton nebo písek.



Obrázek 2.2: Srovnání osvětlovacích modelů. Zleva shora:
anisotropický, Blinnův, pro kovy, Oren-Nayarův a Blinnův, Phongův, Straussův.[5]

Výpočet osvětlení můžeme provést pro každý pixel, který vznikne rasterizací (*Phongovo stínování*, fragment shader), nebo interpolujeme hodnotu mezi vrcholy polygonů (*Gouraudovo stínování*, vertex shader). První způsob produkuje kvalitnější výsledek, ale pokud potřebujeme urychlit vykreslování, je praktické předpočítat alespoň některé výrazy už při zpracování vrcholů.

Podstatnou součást zobrazované scény tvoří stíny. Jejich přítomnost je důležitá pro správné vnímání hloubky v obraze. Stín vzniká, pokud je mezi objekt - příjemce (angl. receiver) a světelný zdroj umístěn další objekt - okluzor (angl. occluder). Na povrch nedopadá světelné záření a proto zde vynecháváme část výpočtu osvětlení (ambientní složka se stále

[5]Zdroj: http://www.camillotrevisan.it/cad2002/rendering_03/Rendering_03.htm

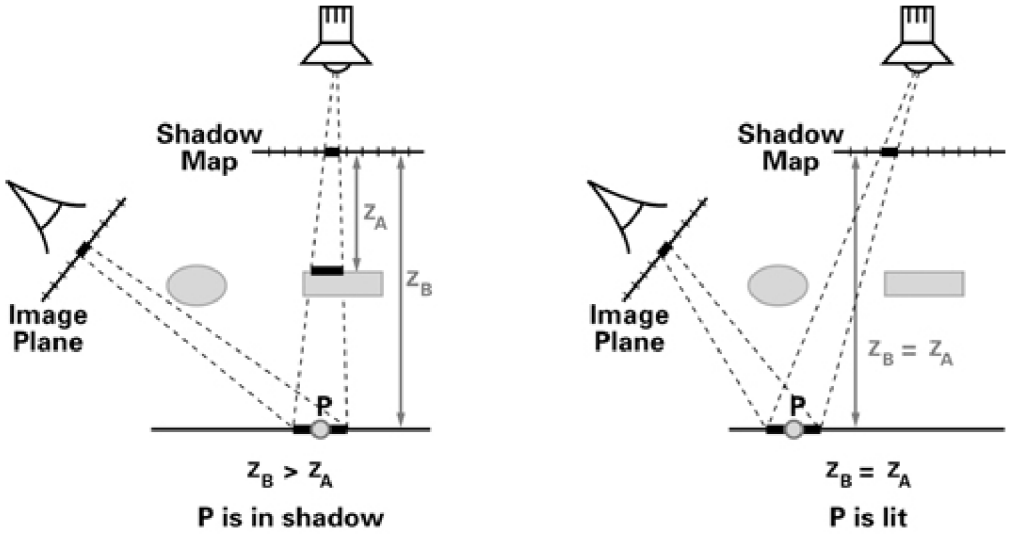
projeví, viz výše). Přestože se jedná o základní efekt, není obsažen v API OpenGL.

## Shadow mapping

Shadow mapping [9] představuje nejpoužívanější algoritmus pro výpočet stínů. Vyžaduje zpracování scény ve dvou průchodech. V prvním je vykreslena geometrie z pohledu světla. Musíme tedy ze známé pozice světla a směru jeho působení sestavit pohledovou matici. Projekci nastavíme podle typu světelného zdroje (viz 2.1) – ortogonální pro směrová světla, perspektivní pro spotlight. Bodová světla jsou komplikovanější, protože nemůžeme určit směr jejich působení. Řešení nabízí využití cube mapy [23] nebo pokrytí okolí bodového zdroje pomocí šesti spotlight zdrojů. K cílovému framebufferu připojíme texturu jako buffer hloubky. Ta bude představovat stínovou mapu. Její velikost nemusí odpovídat rozměrům defaultního framebufferu. V tomto průchodu nepotřebujeme provádět vykreslení barvy, zajímá nás pouze hloubka fragmentů.

V druhém průchodu připojíme na vstup fragment shaderu získanou texturu (stínovou mapu). Scénu nyní vykreslíme z pohledu pozorovatele. Pozici fragmentů transformujeme znovu do souřadného systému světla. Následně porovnáme hloubku s hodnotou získanou ze stínové mapy[6]. Pokud je hloubka větší, bod se nachází ve stínu. Vzhledem k tomu, že porovnáváme hodnoty s omezenou přesností (datový typ `float`), odečteme od transformované hloubky fragmentu experimentálně zjištěnou konstantu (v angl. *bias*), abychom předešli problému zvanému *z-fighting* [23].

Pro větší množství světelných zdrojů musíme opakovat postup pro vytvoření stínové mapy a proces srovnání ve fragment shaderu.




Obrázek 2.3: Princip metody shadow mapping.[7]

Tato metoda přináší výhody v jednoduchosti implementace a rychlosti. Na druhou stranu není snadné dosáhnout kvalitního výsledku. Okraje stínů si uchovávají blokovitý tvar[8], protože sousední body ve scéně jsou mapovány na stejné hodnoty v textuře. Větší

[6] Jako texturovací souřadnice využijeme souřadnice x a y transformované pozice
[7] Zdroj: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html
[8] Tento problém se nazývá *aliasing*

rozměry stínové mapy obecně vyvažují nepřesnost hodnot získaných jejím vzorkováním. S rozlišením ovšem roste časová náročnost vykreslování v prvním průchodu a také paměťová náročnost pro uložení textury. Další řešení poskytuje technika PCF (*Percentage Closer Filtering*) [4], která vzorkuje a porovnává více hodnot z okolí původního vzorku a průměruje výsledky jejich srovnání. Dojde tak k rozmazání okrajů stínu.

Existuje množství variant algoritmu shadow mapping. Většina z nich se zaměřuje na zvýšení kvality stínů. Následuje několik příkladů:

- **Cascaded shadow maps** (CSM) [7] využívá více stínových map, které dělí pohled na několik částí na základě vzdálenosti od pozorovatele. Bližší textury mají vyšší rozlišení než vzdálenější. Je zde tedy zohledněna myšlenka přesunutí většího množství detailů před pozorovatele.
- **Parallel-split shadow maps** (PSSM) [30] je modifikací CSM. Vzdálenosti pro dělení pohledu upravuje dynamicky tak, aby byl zohledněn celý rozsah hloubky scény.
- **Perspective shadow maps** (PSM) [24] provádí generování stínové mapy a následné porovnání až po perspektivní projekci. V prvním průchodu je potřeba transformovat světlo společně s geometrií. Po projekci jsou všechny souřadnice mapovány do rozsahu $\langle -1, 1 \rangle$. Proto jsou zde stíny objektů menší, což znamená i snížení blokovitosti jejich okrajů.
- **Variance shadow maps** (VSM) [8] ukládá do stínové mapy kromě hloubky také její druhou mocninu. Hodnota pro porovnání se pak získá pomocí funkce rozložení pravděpodobnosti. Hlavním přínosem této metody je možnost filtrovat texturu (např. pomocí Gaussova filtru) a rozmazat tím hrany stínů.

Obrázek 2.4: Scéna se stíny vytvořenými pomocí metody VSM.[9]

[9]Zdroj: http://downloads.guru3d.com/download.php?det=1628

## Shadow volumes

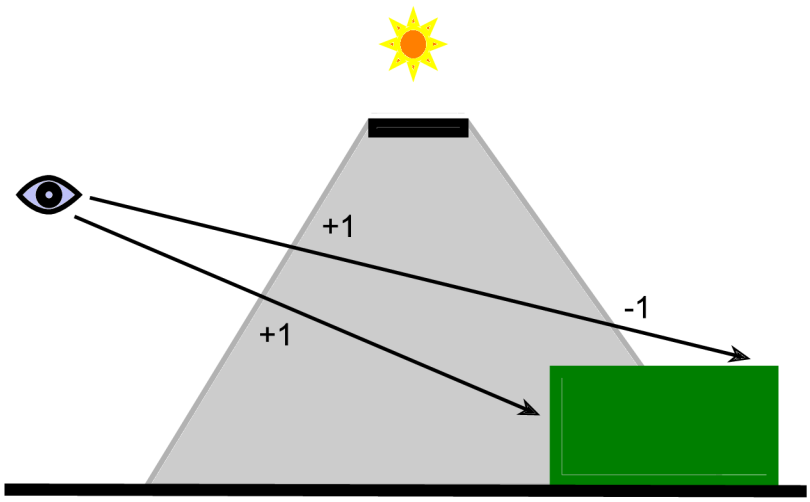
Metoda Shadow volumes (stínová tělesa) [17] přináší alternativní přístup pro generování stínů. Implementace následuje tento postup:

1. Vykresíme scénu se zahrnutím emisní a ambientní složky světla pro zobrazení oblastí ve stínu. Získáme buffer hloubky pro budoucí fáze algoritmu. Připravená informace o hloubce zároveň slouží pro optimalizaci dalšího vykreslování - nyní již nemusíme přepisovat pixely ve fragment shaderu.

Další kroky opakujeme pro všechny světelné zdroje ve scéně:

2. Na CPU sestrojíme siluety objektů pro vykreslení stínových těles.
3. Vynulujeme stencil buffer.
4. Pro všechny objekty vykreslíme zadní stranu stínových těles do stencil bufferu. Inkrementujeme hodnotu pokud test hloubky selže (*z-fail*).
5. Provedeme vykreslení předních stran stínových těles, tentokrát dekrementujeme hodnotu ve stencil bufferu.
6. Konečně vykreslíme celou scénu a vypočítáme osvětlení v místech, kde stencil buffer obsahuje hodnotu 0. Výslednou barvu přidáme do framebufferu pomocí aditivního blendingu [23].

Principem je reprezentace oblasti zastíněné objektem pomocí stínového tělesa. Tvar pro jeho vykreslení získáme protažením paprsků vycházejících ze světla přes siluetu objektu. Kroky 4, 5 a 6 si můžeme představit jako sledování dráhy světelného záření mezi pozorovatelem a bodem na povrchu objektu (viz obrázek 2.5). Jakmile tento pomyslný paprsek vstoupí dovnitř stínového tělesa, inkrementujeme hodnotu stencil bufferu. Naopak vystoupí-li ven, operace se změní na dekrementaci. Pokud tedy stencil buffer obsahuje hodnotu 0, je zřejmé, že tento bod se nachází před nebo až za stínovým tělesem.




Obrázek 2.5: Princip kumulování hodnoty ve stencil bufferu.

Algoritmus vytváří stíny na pixel přesné a bez aliasingu. Snadno řeší všechny typy světelných zdrojů. Umožňuje také zpracování průsvitných objektů. V rychlosti se však nevyrovná metodě shadow mapping (viz 2.1), zejména u větších scén. Pro kvalitní obrysy stínů vyžaduje detailní geometrii, ze které lze snadno získat siluetu (výpočet zatěžuje CPU). Další nevýhodu představuje velké množství vykreslovacích průchodů, protože vykreslujeme nejen geometrii, ale také stínová tělesa z obou stran. Některé z těchto problémů lze optimalizovat, viz [25].

## Ambient occlusion

Předcházející kapitoly řeší osvětlení v lokálním kontextu, tedy vždy pro jistý bod na povrchu. V reálném světě se však světelné záření často odráží a dopadá na povrch z jiných úhlů, než jen přímo ze zdroje. Tento jev nahrazuje v uvedených modelech ambientní složka. Abychom dostali přesnější hodnoty, musíme přidat výpočet *globálního osvětlení*. Pro tento účel existuje řada specializovaných metod, jako např. raytracing, radiozita nebo mapování fotonů. Produkují velmi kvalitní výsledky blízké fotorealismu. Jejich nevýhoda spočívá ve výpočetní náročnosti. Kvůli tomu se většinou nepoužívají u aplikací běžících v reálném čase přímo. Často se ale setkáme s předpočítanými výsledky uloženými v texturách – tzv. *světelných mapách*. S výkonností textur již shadery problém nemají. V tomto případě se ovšem jedná o statické řešení.

Algoritmus *ambient occlusion*[10] [19] nabízí implementaci globálního osvětlení vhodnou pro dynamické scény. Obvyklá aproximace ambientní složky neuvažuje zastínění bodu okolní geometrií, které způsobuje útlum osvětlení v záhybech, koutech a rozích. Objekty pak vypadají ploše, neboť ztrácejí část informace o členitosti povrchu. Ambient occlusion zjišťuje zakrytí bodu vzorkováním hloubky v jeho okolí. Vzorky vybíráme v rozsahu polokoule nad počítaným bodem. Jejich množství ovlivňuje kvalitu efektu. Nejmenší zakrytí očekáváme ve směru normály. Pro výpočet intenzity $I_a$ ambientní složky osvětlení použijeme rovnici 2.2. Výsledkem je hodnota v rozsahu $\langle 0, 1 \rangle$.

$$I_a(p, \overrightarrow{N}) = \frac{1}{N} \sum_{i=1}^{N} v(p, \overrightarrow{\omega}_i)(\overrightarrow{N} \cdot \overrightarrow{\omega}_i) \tag{2.2}$$

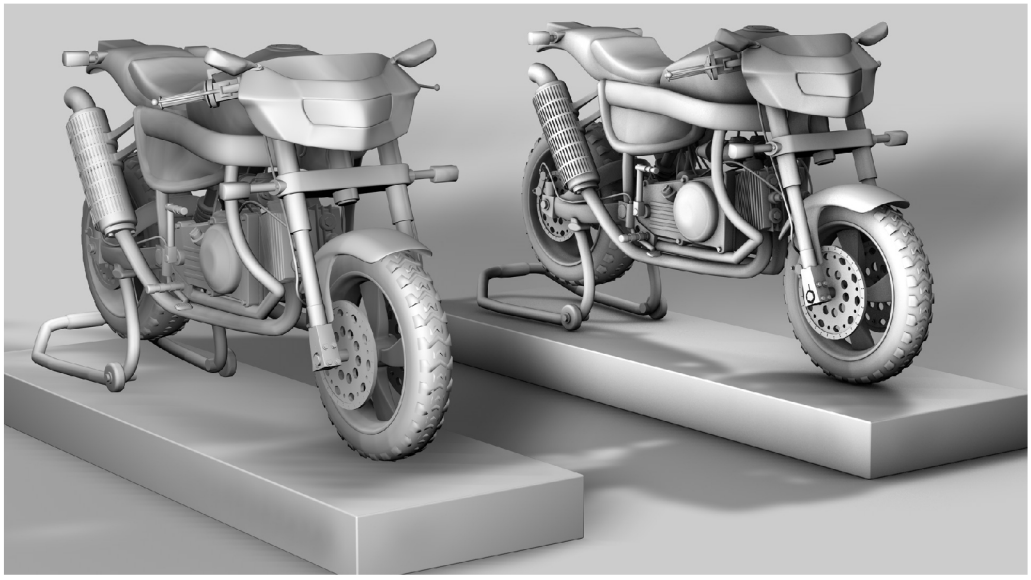
Pro pozici bodu $p$ a normálu $\overrightarrow{N}$ získáme intenzitu jako průměr příspěvků všech vzorků. Funkce viditelnosti $v$ vrací hodnotu 0 pro bod zakrytý okolím, jinak 1. $\overrightarrow{\omega}$ vyjadřuje vektor směřující k $i$-tému vzorku. Skalární součin $\overrightarrow{N} \cdot \overrightarrow{\omega}$ zohledňuje sklon zkoumaného směru vzhledem k normále.

**Screen space ambient occlusion** (SSAO) [12] patří k nejrychlejším variantám algoritmu. Uvedený postup lze snadno implementovat v shaderech. Pracuje v prostoru obrazu, tedy až po vykreslení geometrie. Není tak ovlivněn složitostí geometrie objektů ve scéně. Před výpočtem vyžaduje uložení normál a hloubky do textur. Pomocí hloubky okolních pixelů můžeme rekonstruovat jejich pozici. Pro výběr vzorků se obvykle využívá vyhledávací tabulka s předem definovanými ofsety pozice.

[10] V překladu "zastínění okolím"

**Horizont based ambient occlusion** (HBAO) [1] ukládá normálové vektory v souřadném systému pozorovatele, tedy ještě před jejich transformací. Dále v prostoru obrazu vzorkuje buffer hloubky tak, že vždy podle zvoleného úhlu prochází s definovaným krokem pixely na přímce směřující přes zpracovávaný pixel. Snaží se tak spočítat tzv. *úhel horizontu*, který slouží pro výpočet hodnoty zastínění v bodě.

**Fast ray-cast ambient occlusion** (FRAO) [14] využívá geometry shader pro sestrojení útvaru, který pokrývá oblast potencionálně zastíněnou vykreslovaným polygonem. Podle potřebné velikosti volí šestiboký hranol nebo půlkruh. Následně rasterizuje přední strany těchto útvarů. Pro každý fragment určí zakrytí původní geometrie, jejíž normály a hloubky jsou uloženy v texturách. Úroveň zastínění se akumuluje do tzv. *occlusion bufferu*. Hodnotu intenzity ambientní složky pak získáme z rovnice 2.2.



Obrázek 2.6: Příklad modelu vykresleného bez (vlevo) a s (vpravo) ambient occlusion.[11]

## 2.2 Varianty metody antialiasing

Aliasing je v 3D grafice známý problém. Způsobuje zubatost úseček a hran polygonů, jejichž sklon neodpovídá natočení diskrétní matice pixelů u displeje. Prakticky se jedná o všechny hrany kromě vodorovných nebo svislých. Vykreslený obraz pak nepůsobí zcela realisticky. Abychom přesvědčili náš zrak, potřebujeme aplikovat korekci postižených pixelů rozmazáním jejich okolí.

Nejstarším hardwarovým řešením byl **Supersample Antialiasing** (SSAA, supersampling) [2]. Vykresluje obraz v násobně větším rozlišení a následně jej zmenší na velikost defaultního framebufferu. Výsledné hodnoty obsahují průměr z několika okolních vzorků. S rozlišením však roste i časová náročnost vykreslování, proto se tato metoda často nepoužívá.

[11] Zdroj: http://www.haag-roland.de/2011/02/10/ambient-occlusion-in-cinema-4d

Naproti tomu **Multisample Antialiasing** (MSAA, multisampling) [23, 27] dnes implementují všechny grafické karty. Využívá se zde podobný princip jako u SSAA – průměrování více vzorků. Rozdíl spočívá v množství zpracovaných fragmentů. Fragment shader je spuštěn pouze jednou pro každý fragment, zatímco test hloubky a stencil test se provádí pro každý vzorek. Výsledná barva je následně uložena u vzorků, které projdou zmíněnými testy. Při zobrazování nebo čtení z framebufferu se hodnota pixelu získá průměrovací operací zvanou *MSAA resolve*.

S rozvojem programování vykreslovacího řetězce vzniklo několik metod, které aplikují antialiasing v prostoru obrazu až po výpočtu osvětlení. Poskytují zrychlení oproti MSAA, protože nejsou vázané komplexností geometrie objektů ve scéně. Navíc nevyžadují speciální buffery pro uložení většího množství vzorků. Fungují na principu detekce hran ve 2D obraze. Výsledný efekt je víceméně shodný s MSAA, ikdyž může obsahovat silnější rozmazání (textury uvnitř polygonů částečně ztrácejí ostrost)[12]. Následuje příklad dvou variant:

- **Morphological Antialiasing** (MLAA) [21] detekuje viditelné přechody[13] v obraze, které si ukládá do textury. Z označených pixelů přítomných hran pak vytváří váhy pro blending v posledním kroku. Algoritmus vyžaduje zpracování tří textur. Tato implementace je modifikovaná pro využití v shaderech, původní verze byla cílena na CPU.
- **Fast Approximate Antialiasing** (FXAA) [15] transformuje texturu s barvou v jediném vykreslovacím průchodu. Rozmazává hrany detekované pomocí filtru s adaptivním poloměrem a směrem. Pracuje s jasovou složkou obrazu. Jedná se o častou implementaci v počítačových hrách.


Obrázek 2.7: Ve hře Saboteur byla pro vyhlazení hran použita metoda MLAA na platformě PlayStation 3 narozdíl od platformy Xbox 360.[14]

[12] Uživatel si vytváří subjektivní názor při hodnocení výsledku
[13] Pro detekci se používají barvy, hloubky, normály nebo id objektů.
[14] Zdroj: http://www.eurogamer.net/articles/digitalfoundry-saboteur-aa-blog-entry

## 2.3 Glow efekt

Objekty, které vyžařují světelné záření, kolem sebe generují viditelnou auru. Může se jednat o samotné světelné zdroje, lesklé materiály, neonové nápisy aj. Ve hrách a filmech se podobný efekt využívá pro navození „sci-fi atmosféry". V počítačové grafice jej můžeme snadno simulovat v shaderech jako tzv. *glow efekt* [10]. Nahrazujeme tak emisní složku světla, která je pro tento účel nedostatečná (viz 2.1).

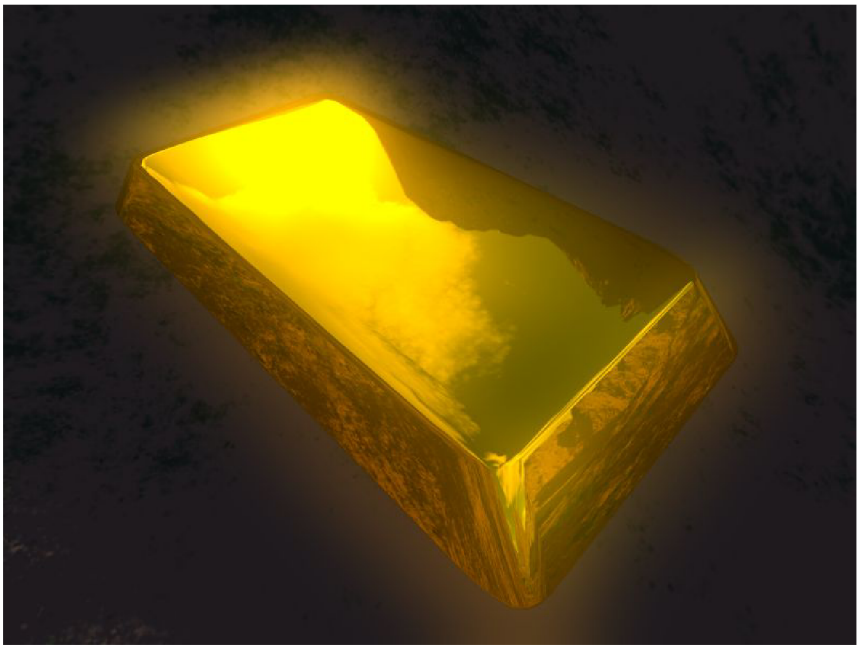
Nejprve budeme potřebovat identifikaci zářících objektů – příznak, nebo jejich částí – textura. V textuře s formátem RGBA je ideální použít kanál alfa. Dále postupujeme tímto způsobem:

1. Vykreslíme scénu do defaultního framebufferu a zároveň vytvoříme tzv. *glow texturu*. Ta slouží jako maska zářících oblastí ve výsledném obraze. Obsahuje barvu a intenzitu aur objektů.

2. (Volitelně) snížíme rozlišení glow textury. Vzhledem k filtraci v kroku 3 si můžeme dovolit ztratit část detailu. V praxi se využívá zmenšení na $\frac{1}{3}$ až $\frac{1}{4}$. Pro převzorkování je ideální bilineární filtrace dostupná v texturovacích jednotkách GPU. Předejdeme tak aliasingu, který by se projevil blikáním aury mezi snímky.

3. Na glow texturu aplikujeme rozmazávací filtr, např. Gaussův filtr. Pro kvalitu efektu potřebujeme aplikovat větší jádro filtru, proto s úspěchem využijeme jeho separovatelnosti. Počet čtených vzorků na pixel tím zredukujeme z $n^2$ na $2n$. Texturu rozmažeme ve dvou samostatných průchodech - horizontální směr, pak vertikální.

4. Pomocí aditivního blendingu přidáme glow texturu do defaultního framebufferu.



Obrázek 2.8: Glow efekt na modelu zlaté cihly.[15]

[15] Zdroj: http://www.humus.name/index.php?page=3D&&start=56

Variantu tohoto efektu tvoří *bloom* [29]. Ten zesiluje a rozmazává v obraze oblasti s vyšší intenzitou osvětlení. Glow textura je generována prahováním obrazu s vykreslenou scénou. Při filtraci se používá jádro s menším průměrem, aby výsledek nebyl přehnaně výrazný. Tento algoritmus lze využít v souvislosti s vykreslováním HDR obrazu [23].

## 2.4 Algoritmus Depth of field

V 3D grafických aplikacích vytváříme ideální projekci přes tzv. dírkovou komoru (angl. *pinhole camera*) [6]. Získáváme tak dokonale ostrý obraz[16], ale neuvažujeme velikost a vlastnosti čočky (kamera, oko). V reálném světě zaostření na určitý objekt nastavuje ohniskovou vzdálenost čočky, která dále definuje ohniskovou rovinu kolmou ke směru pohledu. Všechny objekty mimo tuto rovinu se jeví pozorovateli rozmazané. Světelné paprsky, které z nich vycházejí, se v čočce promítají na oblast, nikoliv do bodu. Tuto oblast nazýváme *circle of confusion* (dále jen CoC) [6] a její velikost roste se vzdáleností od ohniskové roviny.

Obrázek 2.9: Efekt depth of field ve hře Witcher 2 : Assassins of Kings.[17]
Pohled je zaostřen na postavy v popředí.

Počítačová grafika tento efekt simuluje pomocí algoritmu *Depth of field* [6]. Musíme zahrnout parametry kamery a projekce. Rozsah CoC lze zjistit například z bufferu hloubky. Existuje několik způsobů implementace:

- *Raytracing* vysílá a následně průměruje větší množství paprsků vržených přes virtuální čočku do scény. Tato metoda nesouvisí s rasterizací a dále se jí nebudeme zabývat.
- Pomocí *accumulation bufferu* můžeme napodobit raytracing opakovaným vykreslením scény s mírným posunem. S velikostí CoC bohužel roste potřebný počet vykreslovacích průchodů[18] pro získání kvalitního výsledku. Proto není lehké dosáhnout efektu v reálném čase. Na druhou stranu tato technika využívá fyzikálně korektní postup.

[16] Pokud nezpůsobíme rozmazání vlastními nebo nežádoucími efekty
[17] Zdroj: http://www.geforce.com/whats-new/articles/witcher-2-deep-dive-part-2
[18] Jedná se vždy o kompletní vykreslení geometrie scény, výpočet osvětlení atd.

- Objekty můžeme vykreslit do různých *vrstev* (textur) podle jejich vzdálenosti od ohniskové roviny. Následně rozmažeme textury podle určené velikosti CoC a pomocí blendingu spojíme výsledný obraz. Tímto přistupem však nelze úspěšně řešit objekty přesahující více hloubkových úrovní (např. stěny budov) nebo jejich překrývání. Implementace vyžaduje optimální dělení scény na vrstvy.

- **Dopředné mapování bufferu hloubky** funguje na principu mapování pixelů do cílového obrazu. Nejprve vykreslíme scénu a uložíme barvy a hloubky do textur. Určíme CoC z bufferu hloubky. Pomocí blendingu vložíme do barevné textury každý pixel jako kruh o velikosti CoC. Rozptýlení CoC vykreslením velkého množství kruhů ovšem není ideální pro implementaci na GPU.

- **Zpětné mapování bufferu hloubky** vyžaduje na vstupu stejné textury jako dopředné. Rozmazává barevné pixely na základě porovnání hloubky se vzdáleností ohniskové roviny. Rozdíl definuje velikost jádra filtru. Filtraci lze nahradit vzorkováním mipmap [23]. Tato metoda se v současné době využívá pro implementaci na GPU kvůli její rychlosti.

# Kapitola 3

# Technické zpracování efektů

U grafických aplikací určených pro běh v reálném čase závisí výkon především na způsobu implementace a použitých technologiích. Výběr nevhodné struktury dat nebo volání nesprávné funkce může znamenat podstatné zpoždění. Většinou se snažíme dosáhnout alespoň 30 *snímků za vteřinu* (angl. frames per second, FPS).

Kapitola 2 obsahuje shrnutí několika používaných vizuálních efektů. Můžeme si všimnout, že se v jednotlivých postupech opakují podobné techniky. Pro získání požadovaného výsledku musíme provést sérii kroků, které nazýváme vykreslovací průchody (viz 3.2). Průchody si předávají data přes textury (viz 3.1). Obvykle se jedná o jistou reprezentaci vykreslené scény.

Následující podkapitoly popisují principy podstatné pro implementaci efektů. Vychází z prostředků dostupných v knihovně OpenGL [22, 23].

## 3.1 Buffery a textury

Na grafické kartě se vykreslený obraz ukládá do tzv. *framebufferu* (neboli paměťi snímku). Zde data setrvávají dokud nejsou nahrazena jinými nebo je paměť uvolněna. Za povšimnutí stojí fakt, že prezentace obrazu na displeji není povinná. Můžeme tedy provádět s bufferrem různé *off-screen operace*[1], aniž bychom znehodnotili obsah aplikačního okna. Framebuffer samotný se skládá z několika částí:

- **Color buffer** byl původně určen pro ukládání barvy ve fragment shaderu. Pro implementaci efektů se však často nahrazuje libovolnými potřebnými daty (normály, pozice aj.). Lze využít více instancí současně, každá má svůj index (v OpenGL `GL_COLOR_ATTACHMENTn`, kde $n$ je v rozsahu 0 až `GL_MAX_DRAW_BUFFERS`).
- **Depth buffer** obsahuje hloubky (souřadnice z) uložených fragmentů. Tato informace má široké využití u vykreslování (řeší překrývání polygonů) i efektů. Připojujeme jako `GL_DEPTH_ATTACHMENT`.
- **Stencil buffer** kumuluje hodnotu na základě výsledku zvoleného testu (viz [23]). Přestože OpenGL obsahuje konstanty (`GL_STENCIL_ATTACHMENT`) pro definici samostatného stencil bufferu, v praxi je podporována pouze kombinace s depth bufferem.

Současně může být aktivní pouze jeden framebuffer. Programátor jej nastaví voláním funkce `glBindFramebuffer`. Samotný framebuffer ukládá pouze stavové informace. Pro

[1] Bez zobrazení výsledku na displeji

získání užitečného výsledku k němu musíme připojit některé z výše zmíněných bufferů. Ty se označují jako cíle vykreslování (angl. *render targets*[2]). OpenGL rozlišuje 2 typy:

- **Renderbuffer** původně simuloval prostředek pro off-screen vykreslování. Bohužel hodnoty nelze zpětně číst v shaderech, proto je jeho užitečnost omezená.
- **Textura** představuje současný trend. Jedná se o stejné entity, které materiály využívají např. pro obarvení povrchu. Podstatný rozdíl představuje možnost vzorkovat textury na vstupu shaderů. Jak bylo zmíněno v úvodu kapitoly 3, tato funkčnost dovoluje předávání dat mezi jednotlivými vykreslovacími průchody v rámci efektu. Textury navíc podporují struktury s více dimenzemi - 1D/2D/3D/cube, pro připojení k framebufferu používáme funkce `glFramebufferTexture1D/2D/3D`.

Ne každý color buffer musí být vždy aktivní. OpenGL vyžaduje zvolit množinu povolených výstupů pomocí funkcí `glDrawBuffer/s`. Voláním `glDrawBuffer(GL_NONE)` vyřadíme z provozu zápis hodnot ve fragment shaderu. Stále však může být naplněn depth/stencil buffer, což využijeme např. při tvorbě stínové mapy (viz 2.1).

Před zahájením vykreslování, nebo libovolně v průběhu efektu, je potřeba vyčistit obsah připojených bufferů, abychom odstranili obsah vygenerovaný pro předchozí snímek. Obvykle se jedná o vynulování odkazované paměti. To zajistíme funkcí `glClear`, kde parametr představuje or-kombinace konstant `GL_COLOR_BUFFER_BIT` (všechny aktivní, viz výše), `GL_DEPTH_BUFFER_BIT` a `GL_STENCIL_BUFFER_BIT` pro příslušné buffery.

Při inicializaci[3] kontextu OpenGL vzniká automaticky *defaultní framebuffer*. Ten asociujeme s oblastí aplikačního okna, do kterého vykreslujeme. Změna velikosti okna se vždy promítne i do rozměrů přítomných bufferů. Parametry (přítomnost depth/stencil bufferu, multisampling aj.) lze nastavit při vytváření kontextu.

Defaultní framebuffer obsahuje odlišné složení vykreslovacích cílů, než klasický. Jeho strukturu nelze měnit a z připojených bufferů nelze číst (jedná se o renderbuffery). Color buffer obsahuje 2 části[4] - zadní (`GL_BACK`) pro vykreslování a přední (`GL_FRONT`) pro zobrazení v okně. Tato technika se nazývá *double buffering*. Obsah se zobrazí, pokud dojde k přesunu dat ze zadního bufferu do předního. OpenGL podporuje u defaultního framebufferu rovněž *accumulation buffer*, jeho užitečnost ovšem klesla s rozvojem shaderů a blendingu.

Schopnost vykreslovat současně do více bufferů se nazývá *Multiple Render Targets* (MRT) [23]. Programátoři tuto techniku využívají pro implementaci víceprůchodových efektů. Aby se zobrazil výsledek v okně, musí (nejčastěji) poslední průchod zapsat barvu do defaultního framebufferu.

## 3.2 Vykreslovací průchody

Mnoho efektů vyžaduje pro dokončení více vykreslovacích průchodů. Průchodem rozumíme inicializaci a vykreslení zvolené geometrie. Na nejvyšší úrovni lze rozlišit dva základní typy:

- **3D** průchod vykresluje kompletní geometrii scény. Počet objektů může být optimalizován při inicializaci, např. v rámci řešení viditelnosti. Definujeme zde pohled a projekci,

[2] Tento název používá knihovna Direct3D
[3] Vytvoření kontextu OpenGL zajišťují externí knihovny, tato operace je závislá na cílové platformě
[4] Ve skutečnosti se jedná o 2x 2 buffery pro stereoskopické vykreslování

často zapouzdřené jako kamera. Tento typ průchodu obvykle slouží k uložení hodnot do framebufferu pro další zpracování.

- **2D** průchod používáme pro zpracování 2D obrazu. Není vázán komplexností geometrie objektů scény, proto můžeme zpracovat větší množství dat za kratší čas. Pro správnou činnost vyžaduje textury na vstupu i výstupu[5]. Vykreslovaný objekt představuje obdélník, který pokryje rozměry textur.

Pro každou logickou část vykreslované geometrie je v rámci průchodu zavolán jeden *vykreslovací příkaz* (výjimka viz 3.4), který zahájí zpracování na GPU. V OpenGL pro vykreslování slouží příkazy `glDrawArrays` nebo `glDrawElements` pro indexované buffery, popř. jejich varianty. Následnou funkčnost zajišťuje vykreslovací řetězec OpenGL a definované shadery (viz 3.3).

Pořadí průchodů v rámci efektu musí být zachováno. Každý efekt obsahuje jeden průchod, jehož výsledkem je stínování povrchu (angl. surface). Nejedná se vždy o kompletní řešení, ale ostatní průchody vždy vztahují svoji pozici v posloupnosti právě ke zpracování povrchu. Proto rozdělme průchody na (angl.):

- **Pre-pass** předchází řešení povrchu. Protože v této fázi ještě neexistují žádné textury pro zpracování, jedná se většinou o 3D průchod. Často se zde ukládají hodnoty pro pozdější výpočty. Do této kategorie náleží také vytváření stínové mapy (viz 2.1). Pro optimalizaci opakovaného vykreslování rozsáhlé scény lze předem naplnit depth buffer (*z-prepass*), čímž následně zvýšíme počet zahozených fragmentů na základě testu hloubky. Užitečnost tohoto přístupu závisí na složitosti scény a architektuře GPU.

- **Surface pass** vytváří aproximaci stínování povrchu vykreslovaných objektů. Volitelně využívá hodnoty uložené v texturách v předchozích průchodech. Po tomto kroku by měla být zpracována světla a materiály. Zahrnuje aplikaci stínů.

- **Post-processing** označuje sérii 2D průchodů, které přidávají efekty do obrazu získaného při řešení povrchu. Vzhledem k relativní výpočetní nenáročnosti tohoto zpracování se programátoři snaží co největší množství efektů převést na post-processing. Jednu z možných operací představuje také blending.

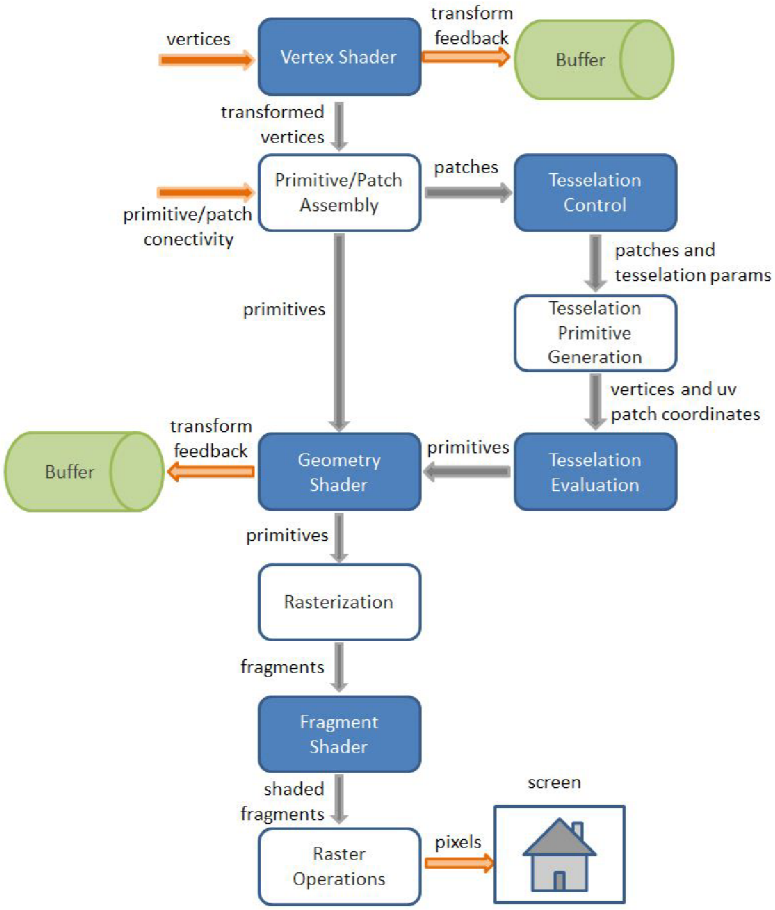
## 3.3 Shadery a programy

OpenGL realizuje vykreslování přes tzv. *vykreslovací řetězec* (angl. rendering pipeline) [23]. Úkolem této posloupnosti kroků je převod 3D geometrie (vrcholy) na 2D obraz (framebuffer). Za účelem rozšíření možností implementace různých efektů došlo k nahrazení některých částí vykreslovacího řetězce za programovatelné bloky. Jejich funkčnost definuje kód v programech, které nazýváme *shadery*.

Pro OpenGL programujeme shadery v jazyce GLSL. Syntaxí i systémem překladu se podobá jazyku C. Pro grafické výpočty disponuje datovými typy jako jsou vektory a matice. Hlavní část programu vkládáme do funkce `main`. Zdrojový kód shaderu musí být nejprve přeložen příkazem `glCompileShader`. Funkce `glLinkProgram` následně sestaví kompletní

[5] Výstup lze směrovat i do renderbufferu, popř. defaultního framebufferu

program[6] pro GPU, který obsahuje přeložený kód připojených shaderů. Před zahájením vykreslovacího příkazu je nutné nastavit aktivní program voláním `glUseProgram`, jinak operace skončí s chybou.




Obrázek 3.1: Vykreslovací řetězec používaný v OpenGL 4. Modré bloky lze programovat.[7]

Při implementaci vlastních efektů závisí výsledek vykreslování zcela na výstupu shaderů. OpenGL umožňuje definovat následující typy shaderů [23]:

- **Vertex shader** zpracovává vstupní geometrii po jednotlivých vrcholech. Probíhají zde transformace pozice a dalších atributů do cílového souřadného systému.
- **Control** a **Evaluation shader** souvisí s procesem teselace, který dělí geometrická primitiva (body, úsečky, polygony) na větší množství menších útvarů. Účelem je generování detailnější reprezentace objektů z menšího množství dat na vstupu.
- **Geometry shader** pracuje s celými geometrickými primitivy. Může generovat nebo naopak zahazovat primitiva a vrcholy. Pro stejný vstup lze spustit více instancí tohoto shaderu.
- **Fragment shader** tvoří jádro vykreslovacího řetězce. Zde dochází ke zpracování fragmentů získaných rasterizací primitiv. Výsledné hodnoty (barva, hloubka aj.) ukládáme

[6] Přestože shader sám je program, OpenGL považuje za program sadu shaderů propojených navzájem
[7] Zdroj: http://www.lighthouse3d.com/2011/03/opengl-4-1-pipeline/

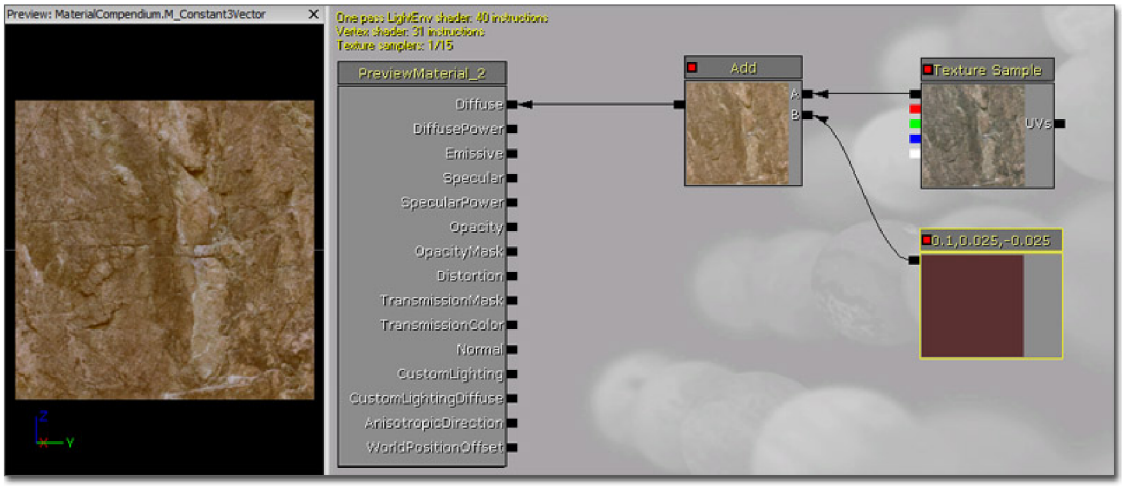
do framebufferu po výpočtech provedených ve fragment shaderu.

- **Compute shader** leží mimo vykreslovací řetězec. Slouží pro zpracování dat v bufferech, což mohou být táke objekty použitelné pro vykreslování (např. vertex buffer s vrcholy). Využívá se pro post-processing nebo obecné výpočty[8].

Vertex shader a fragment shader představují povinně definovanou část, ostatní shadery lze vynechat. Na shadery lze nahlížet jako na klasické programy, které transformují vstup na výstup. Mezi jednotlivými stupni předáváme data pomocí proměnných, které označujeme kvalifikátory `in` a `out`. Při rasterizaci dochází k interpolaci těchto hodnot mezi vrcholy primitiv. Konstanty a parametry představující podstatnou součást efektů předáváme do shaderů přes uniformní proměnné (kvalifikátor `uniform`). Tato část paměti je určena pro čtení v shaderech, zápis provádíme z aplikace variantami funkce `glUniform`.

## 3.4 Materiály

Předcházející kapitoly zmiňují vykreslování geometrie jako iteraci přes vybrané objekty ve scéně. V praktických aplikacích se často využívá odlišný přístup. Objekty se zpracovávají ve skupinách po materiálech, které určují vzhled jejich povrchu. Pokud tedy máme ve scéně například pouze kovové a dřevěné předměty, procházíme vždy dvě skupiny. Omezíme tak počet nutných změn nastavení shaderů v rámci vykreslovacího průchodu.




Obrázek 3.2: Skládání parametru materiálu v nástroji Unreal Material Editor.[9]

Materiál obsahuje sadu parametrů a textur. Z jejich kombinace získáme hodnoty (barva, průhlednost aj.) potřebné pro výpočet stínování povrchu. Postup těchto efektů definujeme při tvorbě modelu v editoru, např. ve formě stromové struktury (viz obrázek 3.2). Podstatný je fakt, že pro každý materiál dostaneme samostatné shadery (respektive program v GLSL). S rostoucím počtem materiálů tak roste i časová náročnost přepínání shaderů za běhu aplikace. Musíme vždy změnit aktivní program, nahrát parametry přes uniformní proměnné, připojit textury atd. Implementace velkého množství shaderů pro odlišné efekty rovněž není snadná záležitost, protože počet jejich kombinací opět rychle narůstá. Následuje přehled nejčastějších technik pro správu shaderů v grafických aplikacích:

[8] Jedná se o alternativu ke knihovnám CUDA a OpenCL

[9] Zdroj: http://udn.epicgames.com/Three/MaterialsCompendium.html

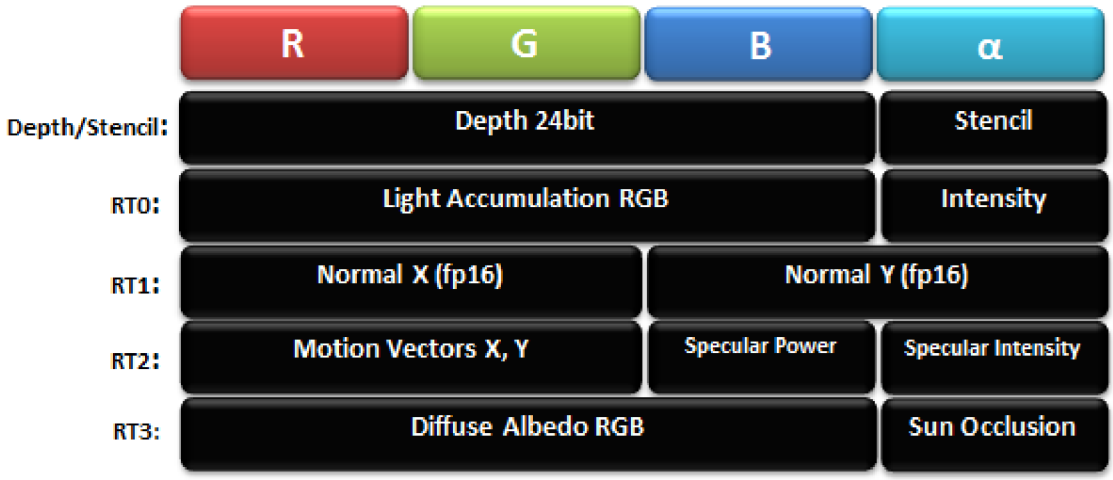
- **Übershader** představuje řešení všech efektů sloučené do jediného zdrojového souboru. Přítomné funkce povolujeme pomocí definice symbolů, které ovlivňují odstranění nebo ponechání kódu při zpracování preprocesorem (konstrukce `#ifdef`). Překládáním tohoto souboru tak lze vygenerovat shadery s různým kódem.

- Principem **mikro shaderů** je umístění každého efektu do samostatného souboru. Kombinace provádíme na úrovni spojování řetězců. Bohužel se vzrůstající složitostí shaderů lze těžko sledovat duplicitní symboly a nadbytečný kód, proto může ve výsledku vzniknout méně efektivní řešení.

- Pokud známe množinu požadovaných efektů předem, můžeme definovat omezenou sadu shaderů. Při tvorbě modelů se pouze upraví parametry zvoleného shaderu. Toto řešení dostačuje pro implementaci specifického vizuálního stylu.

- OpenGL od verze 4.0 dovoluje využít funkčnost podobnou volání funkce přes ukazatel v jazyku C++. Nejprve napíšeme kód pro **podprogramy** označené klíčovým slovem `subroutine` [23], které jsou syntaxí podobné obyčejným funkcím. Všechny podprogramy musí mít stejný typ. Implementaci zvolíme předáním příslušného indexu do unifomní proměnné příkazem `glUniformSubroutineuiv`. V důsledku tedy pracujeme pouze s jedním shaderem, kde přepínáme mezi implementacemi funkcí, což je jednodušší i relativně rychlejší řešení.

Zpracování materiálu a výpočet osvětlení představují podstatné a časově náročné operace. Tradičně se provádí při zpracování geometrie (3D průchod). Tato technika se nazývá **dopředné stínování** (angl. *forward shading*) [27]. Hlavní nevýhoda spočívá ve zbytečných výpočtech u fragmentů, které jsou následně přepsány jiným fragmentem ve framebufferu.




Obrázek 3.3: Formát G-bufferu pro odložené stínování ve hře Killzone.[10]

Vzhledem k tomu, že dnes již lze využít vykreslení scény do textur a následné zpracování v prostoru obrazu (2D průchod), rozvinuly se alternativní metody aplikující tento přístup. Neznámější je **odložené stínování** (angl. *deferred shading*) [27]. Parametry získané z materiálů a další hodnoty potřebné pro výpočet osvětlení v prvním průchodu zapíšeme do skupiny textur označovaných jako *G-buffer*. V následujícím průchodu vypočítáme osvětlení

[10] Zdroj: http://www.spuify.co.uk/?cat=11

již nad 2D reprezentací vykreslené scény. Kromě zrychlení tak získáme i snazší možnost řešit větší množství světelných zdrojů.

# Kapitola 4

# Struktura a funkčnost knihovny pro tvorbu efektů

Hlavní cíl této práce představuje tvorba softwarového frameworku ve formě knihovny tříd. Za účel lze považovat usnadnění vývoje vizuálních efektů pro 3D grafické aplikace. Pro implementaci byl vybrán jazyk C++. Objektově orientovaný přístup (OOP [28]) poskytuje vhodné prostředky pro vystavění struktury knihovny a následné využití třetí stranou.

Pro programování grafického rozhraní bylo zvoleno API OpenGL, které oproti Direct3D nabízí přenositelnost kódu na více platforem. Vzhledem ke snaze přinést inovativní a funkční řešení, spíše než zaměřit se na zpětnou kompatibilitu, bude vyžadována minimálně verze OpenGL 4.2. Framework by měl poskytovat vyšší úroveň abstrakce a to tak, aby uživatel[1] nemusel psát vlastní kód v OpenGL. Dále bude umět zobrazit 3D scénu s vestavěnými nebo definovanými efekty. Měla by existovat možnost zapínat a vypínat efekty za běhu aplikace, což zároveň znamená nutnost kombinovat efekty dohromady. Framework bude samostatně generovat kód shaderů v jazyku GLSL, přičemž uživatel doplní pouze pasáže specifické pro definovaný efekt. Přitom budou automaticky doplněny často opakované úkony, jako např. transformace pozice vrcholu ve vertex shaderu.

Následující podkapitoly popisují návrh knihovny, zejména z pohledu zpracování efektů.

## 4.1 Konstrukce jádra aplikace

Dříve než se začneme zabývat samotnými efekty, potřebujeme definovat strukturu ostatních částí aplikace. Knihovna by měla umožnit vytvoření samostatné aplikace. Kód grafických programů obvykle obsahuje sekvenci operací zahrnující podobný postup:

1. Inicializace grafického kontextu
2. Vytvoření okna pro zobrazení výsledku
3. Načtení scény a převedení na interní reprezentaci
4. Definice kódu pro vykreslování
5. Definice kódu pro zpracování uživatelských vstupů (myš, klávesnice aj.)
6. Zahájení smyčky pro zpracování událostí okna, aktualizace a vykreslení scény

[1] Za uživatele jsou považováni programátoři grafických aplikací

Zaměřme se tedy na jádro knihovny, které poskytne prostředky pro implementaci této struktury. Jednotlivé součásti zde nahradí logické objekty, které mohou být později převedeny na třídy. Následuje výčet potřebných objektů:

- **Kontext** zapouzdřuje ostatní složky aplikace. Řídí běh programu prováděním hlavní smyčky (viz bod 6 výše). Pro svoji činnost vyžaduje aktivní *okno* a *renderer*.
- **Okno** zajišťuje manipulaci se systémovým oknem, které bude sloužit jako cíl pro vykreslování. Vytváří kontext OpenGL. Zahrnuje zpracování uživatelských vstupů.
- **Renderer** má za úkol aktualizaci a vykreslování scény. Využívá *kameru* pro 3D průchody a *světla*. Dále spolupracuje se *správcem scény* a *správcem zdrojů*.
- **Správce scény** ukládá objekty přítomné ve scéně, tj. *modely* a *světla*. Manipuluje s nimi na vyšší úrovni abstrakce. Možné operace představuje vystavění grafu scény, určení viditelnosti aj. Je svázán se *správcem zdrojů*.
- **Správce zdrojů** načítá *meshe*, *materiály* a *textury* ze souborů. Zajišťuje jejich uložení a převod na hardwarovou reprezentaci.
- **Kamera** uchovává nastavení pohledu a projekce pro transformace při vykreslování. Umožňuje uživateli aplikace pohyb ve scéně. K tomu je potřeba propojení s *oknem* kvůli zpracování vstupů.
- **Model** zapouzdřuje objekt scény jako celek. Skládá se z 1 nebo více *meshů*.
- **Světlo** reprezentuje světelný zdroj, který je určen svými parametry. Podle orientace a typu zdroje lze vytvořit příslušnou *kameru*.
- **Mesh** představuje geometrickou jednotku na nejnižší úrovni. Obsahuje odkazy na hardwarové buffery nutné pro její vykreslování. Má přiřazen právě jeden *materiál*.
- **Materiál** ukládá parametry a *textury* potřebné pro vykreslení *meshe*. Více meshů může využívat stejný materiál (viz 3.4).
- **Textura** zapouzdřuje stejnojmenný prostředek z OpenGL.

## 4.2 Definice prvků efektu

Programátor, jakožto uživatel frameworku, nebude psát příkazy OpenGL, ani celý kód shaderů v jazyku GLSL. Místo toho definuje struktury potřebné pro provedení efektu a doplní specifický kód na příslušná místa. Následuje seznam základních prvků, ze kterých bude možné efekt složit. Každý efekt obsahuje sadu vykreslovacích průchodů, proto tyto prvky představují právě součásti průchodů, popř. komponenty shaderů.

### Sémantika průchodu

Sémantika určuje pořadí vykreslovacího průchodu v posloupnosti, vzhledem k výpočtu stínování povrchu. Možné hodnoty zahrnují alespoň pre-pass, surface pass a post-processing (viz 3.2).

### Vstupní a výstupní textury

Jedná se o textury připojené do texturovacích jednotek na vstupu shaderů, nebo k framebufferu na výstupu (viz 3.1). Stejnou texturu lze použít ve více vykreslovacích průchodech, ovšem čtení i zápis zároveň ve stejném shaderu vede na nedefinované chování. Každá textura musí definovat:

- Rozměry
- Formát
- Identifikátor pro referenci v shaderech

### Interpolované proměnné

Popisují rozhraní mezi vertex shaderem a fragment shaderem. Může se jednat o atributy získané z bufferů na vstupu vertex shaderu (pozice, normála aj.) nebo uživatelem definované proměnné. Pro každou proměnnou mohou být definovány 2 *transformační funkce* (syntaxe GLSL). První funkce bude zavolána před uložením hodnoty na výstup vertex shaderu, druhá na začátku fragment shaderu. Transformace samozřejmě není povinná, ale také může obsahovat část kódu efektu. Navíc framework volitelně doplní standardní transformace pro geometrické atributy. Následuje příklad vygenerovaného kódu v GLSL, který obsahuje standardní transformaci pozice ve vertex shaderu (násobení maticí).

```
in  vec3 input_position;
out vec3 output_position;

uniform mat4 model_view_matrix;

// standardní transformační funkce, může být předefinována nebo vynechána
vec3 transform_position(vec3 p)
{
  return (model_view_matrix * vec4(p, 1.0)).xyz;
}

void main()
{
  ...

  output_position = transform_position(input_position);
}
```

Zdrojový kód 4.1: Příklad vygenerovaného kódu pro interpolovanou proměnnou.

### Uniformní proměnné

Programátor definuje vlastní uniformní proměnné pro předání konstant do shaderů. Framework doplní proměnné pro transformační matice, parametry materiálu a světla. Každá proměnná musí obsahovat:

- Identifikátor
- Datový typ
- Cílový shader pro umístění
- Zdroj dat pro aktualizaci (např. adresa v paměti)

### Funkce

Programátorem definované funkce se syntaxí GLSL pro vlastní využití v shaderech. Funkce by měly definovat prioritu, aby bylo možné nahradit funkčnost při kombinování efektů.

Představme si například efekt řešící osvětlení, který ve svém výpočtu pracuje s úrovní stínu pro daný bod tak, že definuje funkci `shadow` vracející konstantní hodnotu. Pak definujeme efekt pro algoritmus stínových map (viz 2.1), kde funkce `shadow` obdrží korektní hodnotu aplikací použité metody. Při spojení efektů budeme chtít využít osvětlení z prvního a zároveň funkci `shadow` z druhého, což zajistíme pomocí priority.

### Zápis hodnot do výstupních bufferů

Aby textury na výstupu shaderu plnily svůj účel, musí být do nich zapsána hodnota. Pro každou takovou texturu by proto měla být definována sekvence kódu, která v posledním řádku provede požadovaný zápis. Tento úsek tvoří další z podstatných částí kódu efektu a bude umístěn do funkce `main` ve fragment shaderu.

```
in vec2 texcoord;

uniform sampler2D texture1;
uniform sampler2D texture2;

out vec3 color_buffer;

void main()
{
  ...

  // Tato sekvence kódu zapíše do výstupní textury color_buffer
  // kombinaci barev ze 2 vstupních textur
  vec3 color1 = texture2D(texture1, texcoord).rgb;
  vec3 color2 = texture2D(texture2, texcoord).rgb;
  color_buffer = mix(color1, color2, 0.5);
}
```

Zdrojový kód 4.2: Příklad zápisu hodnoty do výstupního bufferu.

### Vykreslovací kontext

Obsahuje množinu konstant, které určují pro daný průchod požadované nastavení vykreslovacího řetězce OpenGL. Předáváním tohoto kontextu mezi průchody lze zabránit nadbytečným změnám stavu OpenGL. Definované hodnoty ovlivňují například:

- Test hloubky
- Stencil test
- Blending
- Odstranění zadních nebo předních stran polygonů

## 4.3 Generování shaderů

Ze součástí definovaných v podkapitole 4.2 framework automaticky vygeneruje kód pro vertex shader a fragment shader. Ty poslouží pro jeden vykreslovací průchod. Při jakékoliv

změně struktury shaderů je potřeba znova provést generování, překlad a sestavení programu. Vzhledem k tomu, že kód efektů se za běhu aplikace nemění, neměl by zde vzniknout výkonnostní skok. Následuje rozpis sekvencí úseků kódu umístěných do jednotlivých shaderů. Některé části se opakují.

**Vertex shader**

1. Verze GLSL (direktiva `#version`)
2. Vstupní geometrie získaná z vertex bufferu. Proměnné jsou označené klíčovým slovem `in`. Nevyužité hodnoty odstraní překladač GLSL. Zahrnuje:
   - Pozici
   - Normálu, tangentu, bitangentu
   - Texturovací souřadnice (více sad)
   - Barvy (více sad)
3. Interpolované proměnné, zde označené jako výstup (`out`)
4. Uniformní proměnné (`uniform`)
5. Vstupní textury (`uniform` + typ sampleru [22], např. `sampler2D`)
6. Uživatelské funkce (viz 4.2)
7. Funkce pro transformaci vstupní geometrie (souvisí s materiály, viz 4.6)
8. Funkce pro transformaci interpolovaných proměnných
9. Funkce `main`
   (a) Volání funkcí pro transformaci vstupní geometrie
   (b) Povinná transformace pozice, tj. zápis do proměnné `gl_Position`
   (c) Volání funkcí pro transformaci a zápis do interpolovaných proměnných

**Fragment shader**

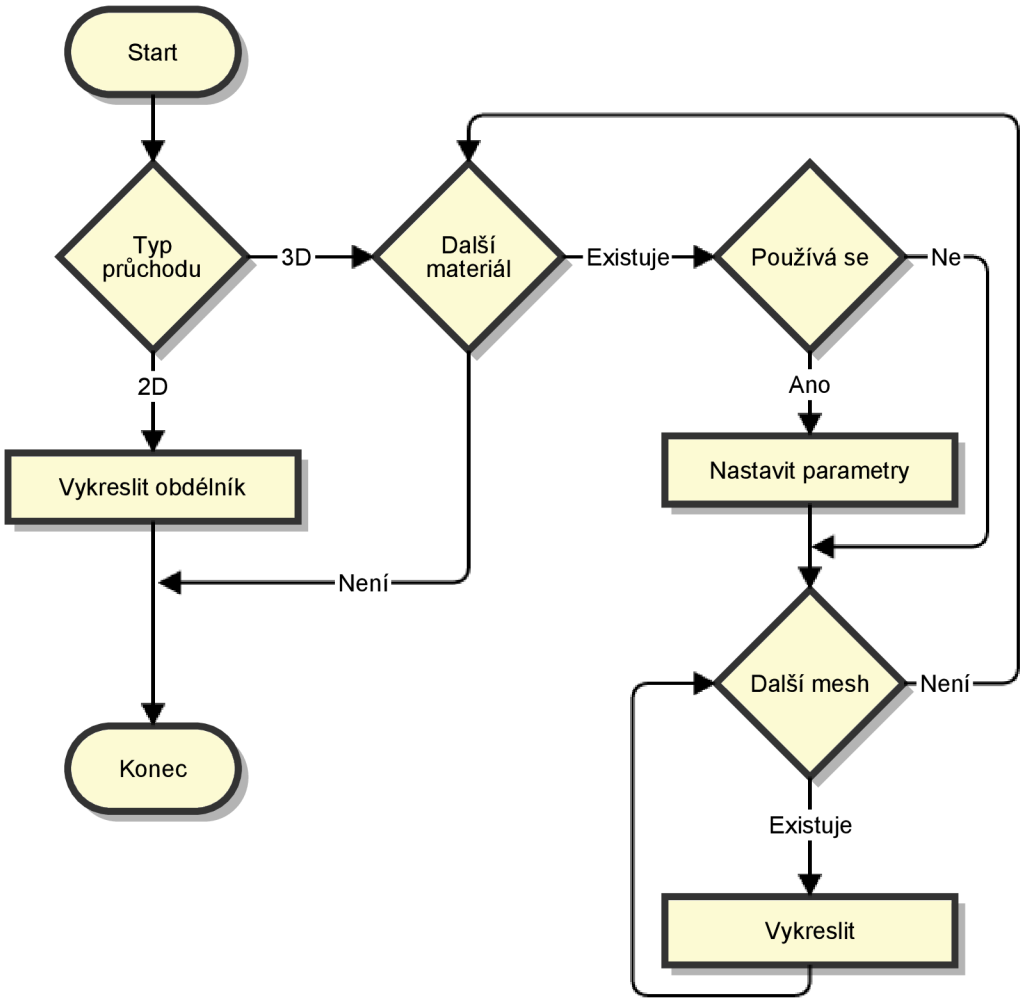
1. Verze GLSL (direktiva `#version`)
2. Interpolované proměnné, zde označené jako vstup (`in`)
3. Uniformní proměnné (`uniform`)
4. Vstupní textury (`uniform` + typ sampleru, např. `sampler2D`)
5. Výstupní textury (`out`), tj. color buffery připojené k framebufferu
6. Uživatelské funkce
7. Funkce `main`
   (a) Volání funkcí pro transformaci interpolovaných proměnných
   (b) Úseky kódu pro zápisy do color bufferů

## 4.4 Interpretace efektu

Provedení efektu představuje iteraci přes definované vykreslovací průchody. Každý průchod se skládá z inicializace a vykreslení. Inicializace zahrnuje tyto kroky:

- Přenastavení vykreslovacího kontextu
- Aktivace programu (shadery)
- Aktivace framebufferu
- Vyčištění cílových bufferů ve framebufferu
- Nastavení uniformních proměnných
- Přiřazení textur k texturovacím jednotkám

Princip vykreslování vyjadřuje následující diagram:




Obrázek 4.1: Postup vykreslování v rámci průchodu

## 4.5 Skládání efektů

Skládáním rozumíme sloučení vykreslovacích průchodů dvou efektů do jediného výsledného efektu. Pořadí průchodů v rámci jednotlivých efektů musí být zachováno. Framework při tom vyhledá kandidátní dvojice průchodů, které by mohl spojit dohromady. Následující vlastnosti průchodů se musí shodovat:

- Typ (2D nebo 3D)
- Sémantika
- Nastavení vykreslovacího kontextu
- Rozměry výstupních textur

Dále dojde ke sloučení prvků definovaných v podkapitole 4.2. Jedinou komplikaci představují konflikty jmen identifikátorů. V takovém případě je nutné provést přejmenování jednoho ze symbolů některých z těchto způsobů:

- Automatické přejmenování frameworkem. Dále by ovšem bylo nutné nahradit všechny výskyty identifikátoru v kódu shaderu, což vyžaduje minimálně jeho lexikální analýzu.
- Upozornění programátora na konflikt pomocí signalizace chybového stavu. Přejmenování je pak potřeba udělat ručně.

Kombinace efektů je vhodné skladovat v paměti pro ušetření času při opakované aktivaci nebo deaktivaci jednotlivých efektů.

## 4.6 Materiály v efektech

Materiály tvoří nezanedbatelnou součást efektů, kterou však každý vykreslovací průchod nemusí využít. Proto je praktické uchovat struktury potřebné pro přidání materiálů do shaderů pohromadě. Kdykoliv pak bude možné snadno spojit materiály s vybraným vykreslovacím průchodem. Materiály definují v shaderech následující prvky:

- Textury
- Uniformní proměnné, tj. parametry materiálu (barvy, průhlednost ad.)
- Funkce pro transformaci vstupní geometrie ve vertex shaderu
- Funkce pro přístup k parametrům materiálu

Transformace vstupní geometrie (pozice, normála aj.) umožňuje přidat efekty jako např. *bump mapping* [23]. Přístupové funkce k parametrům zakrývají rozdílné způsoby získávání těchto hodnot v jednotlivých materiálech. Zde musíme vyřešit problém diverzity kódu shaderů pro různé materiály. Vzhledem k tomu, že framework předpokládá s jediným programem na vykreslovací průchod, jeví se jako ideální implementace pomocí podprogramů OpenGL (viz 3.4). Úsek kódu 4.3 obsahuje názorný příklad.

```
in vec2 texcoord;

uniform vec3 color;
uniform sampler2D color_texture1;
uniform sampler2D color_texture2;

// rozhraní podprogramu
subroutine vec3 get_color();

// aktuálně vybraná implementace podprogramu
uniform subroutine get_color selected_get_color;

// materiál 1 získává barvu z parametru
subroutine (get_color) vec3 get_color_material_1()
{
  return color;
}

// materiál 2 kombinuje barvy ze dvou textur
subroutine (get_color) vec3 get_color_material_2()
{
  vec3 color1 = texture2D(color_texture1, texcoord).rgb;
  vec3 color2 = texture2D(color_texture2, texcoord).rgb;
  return 0.7 * color1 + 0.3 * color2;
}
```

Zdrojový kód 4.3: Příklad definice odlišných podprogramů pro různé materiály.

# Kapitola 5

# Funkční popis implementace

Tato kapitola popisuje samotnou implementaci knihovny pro tvorbu efektů. Jako základ k její tvorbě sloužil návrh z kapitoly 4.

Knihovna je naprogramována v jazyku C++ s podporou standardní knihovny STL [11]. Opírá se o standard C++11 [26]. Tento fakt dovoluje využití užitečných konstrukcí jazyka jako jsou například lambda funkce, formátované řetězcové literály, „chytré" ukazatele s počítáním referencí, atomické operace, automatické odvození datového typu proměnné a další. Nevýhodu představuje omezení na použití novějších verzí překladačů.

Implementace provádí grafické operace pomocí knihovny OpenGL (viz kapitola 3). Sekundární úkony jsou delegovány na pomocné (externí) knihovny, konkrétně:

- **GLM**[1] poskytuje funkce pro matematické operace, práci s vektory a maticemi.
- **GLEW**[2] načítá funkce příslušící rozšířením OpenGL.
- **GLFW**[3] umožňuje vytvoření a správu aplikačního okna.
- **Assimp**[4] načítá 3d modely ze souborů.
- **DevIL**[5] načítá a ukládá obrazové formáty (zde především textury).

Vzhledem k uplatnění objektově orientovaného přístupu [28] je kód strukturován do tříd a struktur.

## 5.1 Základ aplikace

Na začátku a na konci aplikace je nutné provést jisté operace, které povedou ke správnému běhu a ukončení programu. Externí knihovny (viz výše) vyžadují před zahájením jejich používání inicializaci a nastavení. Pro zobrazení vykreslených snímků budeme potřebovat okno, se kterým je zároveň svázán kontext OpenGL. Vytvoření tohoto kontextu představuje nezbytnou podmínku pro volání funkcí knihovny OpenGL. O všechny zmíněné kroky se ve frameworku stará třída `Context` (dále jako *kontext*). Instance této třídy by tedy měla být vytvořena v aplikaci co nejdříve.

[1] Viz http://glm.g-truc.net
[2] Viz http://glew.sourceforge.net
[3] Viz http://www.glfw.org
[4] Viz http://assimp.sourceforge.net
[5] Viz http://openil.sourceforge.net

Po definici všech ostatních datových struktur, které určují chování programu, předá uživatel řízení frameworku voláním metody `Context::MainLoop`. Zde je zahájena smyčka běžící až do okamžiku uzavření aplikačního okna. Následující akce jsou provedeny v každé iteraci:

- Výpočet časového rozdílu vzhledem k předchozí iteraci
- Přepočítání FPS (viz úvod kapitoly 3)
- Aktualizace stavu aplikace
- Vykreslení nového snímku
- Zobrazení snímku
- Zpracování událostí okna

Počet snímků za vteřinu (FPS) slouží jako metrika vhodná pro hodnocení rychlosti implementovaného algoritmu. O vykreslování a aktualizace se vždy stará instance třídy odvozené od třídy `Renderer` (dále jako *renderer*). Přítomnost kontextu a rendereru jsou jediné podmínky k zahájení činnosti frameworku. Můžeme tak napsat minimální kostru programu pomocí úseku kódu 5.1. Lze si všimnout, že nezotavitelné chyby způsobí zahození výjimky, proto je vhodné zajistit kód bloky `try - catch`.

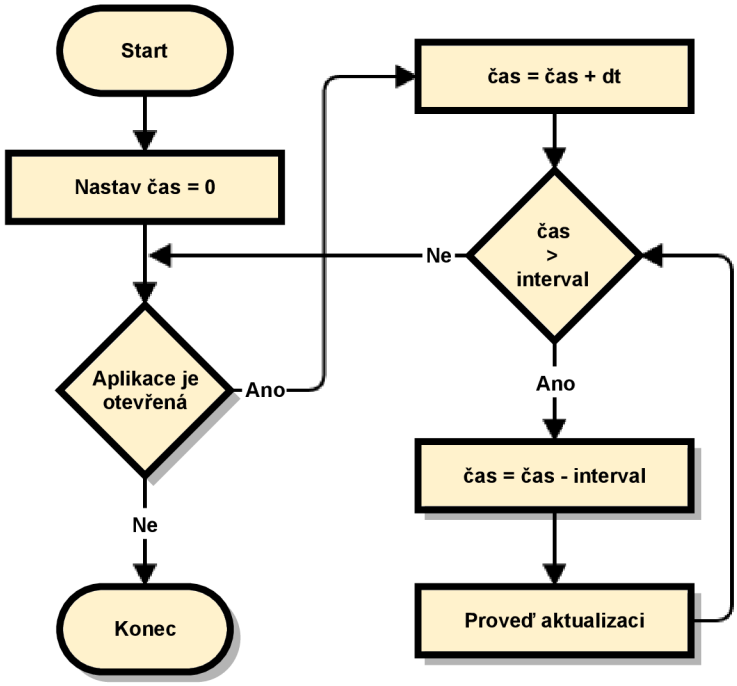
```
#include <exception>
#include <visegl/context.h>
#include <visegl/effect_renderer.h>

// obsahuje všechny symboly frameworku
using namespace visegl;

int main()
{
  try
  {
    // inicializace + vytvoření okna a kontextu OpenGL
    Context context(800, 600, "Minimal program");

    // objekt zajišťující vykreslování a aktualizace
    // EffectRenderer dědí Renderer
    auto renderer = std::make_shared<EffectRenderer>();
    context.SetRenderer(renderer);

    // zahájení hlavní smyčky aplikace
    context.MainLoop();
  }
  catch(std::exception & exc)
  {
    // ... ošetření chybového stavu
    return 1;
  }
}
```

Zdrojový kód 5.1: Minimální funkční program napsaný ve frameworku.

Aktualizace stavu zahrnuje úkony, které vyžadují periodické provádění. Může se jednat například o animace, přenastavení kamery, přepočítání parametrů algoritmu aj. Aby si chování programu uchovalo svůj determinismus (předvídatelnost), jsou aktualizace vykonávány v pravidelných časových intervalech[6]. Vzhledem k tomu, že nelze zajistit stejnou

[6]Tento interval lze změnit, výchozí hodnota je *16,6 ms* (60 aktualizací za vteřinu)

dobu trvání iterací aplikační smyčky, dochází ke kumulaci časového rozdílu mezi jednotlivými iteracemi. Pokud hodnota přesáhne velikost daného intervalu, provede se aktualizace. Framework tak může provést 0, 1 i více aktualizací každou iteraci. Celý postup znázorňuje diagram na obrázku 5.1.




Obrázek 5.1: Princip aktualizace stavu aplikace. `dt` představuje časový rozdíl mezi dvěma iteracemi. Časový krok aktualizací reprezentuje konstanta `interval`.

Okno slouží jako cíl pro zachytávání vstupu od uživatele. Podporován je vstup z klávesnice a myši. Framework reaguje na následující akce:

- Stisknutí/uvolnění klávesy
- Stisknutí/uvolnění tlačítka myši
- Skrolování myší
- Změna pozice kurzoru

Uživatel může nastavit lambda funkce zpracovávající příslušné události (ukázka viz úsek kódu 5.2).

Pro nastavení pohledu a projekce se v rendereru využívá instance některé z tříd implementujících funkčnost kamery. Ty zapouzdřují transformace pohledu a projekce (ortogonální nebo perspektivní). Na tomto místě je vhodné zmínit třídu `FreeCamera`, která dovoluje pozici kamery ve scéně ovládat právě pomocí klávesnice (směrový pohyb) a myši (otáčení). Otáčení lze provádět v ose $x$ a $y$. Implementace využívá *quaterniony* pro vyjádření rotací a tím zabraňuje výskytu problému zvaného *Gimbal lock*.

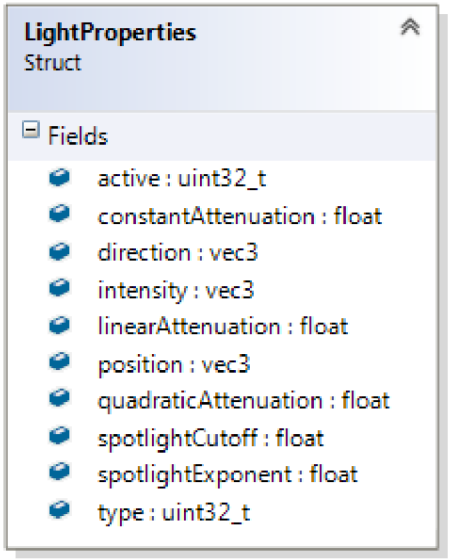
```
auto window = context.GetWindow();

window->SetKeyCallback([&](int key, int scancode, int action, int mode)
{
  if(action == GLFW_PRESS && key == GLFW_KEY_ESCAPE)
  {
    window->Close();
  }
});
```

Zdrojový kód 5.2: Nastavení lambda funkce pro uzavření okna klávesou [Esc]. Konstanty jsou přejaty z knihovny GLFW.

## 5.2 Formování scény

Vizualizace efektů se neobejde bez definice scény, na níž budou grafické algoritmy demonstrovány. Scénu tvoří dva hlavní typy objektů, tj. světla a geometrické modely. Tyto entity jsou pod správou instance třídy `SceneManager` (dále jako *správce scény*). Zde dochází k jejich uložení, popř. načtení. Správce scény pak slouží jako zdroj dat při vykreslování. Jeho instance lze za běhu prohodit a změnit tak obsah zobrazované scény.

Světla reprezentují světelné zdroje tak, jak jsou popsány v podkapitole 2.1. Každá instance obsahuje parametry pro směrová, bodová i spotlight světla [7]. Parametry jsou definovány ve struktuře `LightProperties` (viz obrázek 5.2). Jsou zde zohledněny koeficienty pro vyjádření klesající intenzity (*attenuation*). Nastavením hodnoty `active` lze jednoduše zapínat/vypínat zdroje za běhu aplikace.




Obrázek 5.2: Diagram tříd pro strukturu `LightProperties`.

Vzhledem k tomu, že světla může využívat více shaderů ve více efektech, jsou uloženy do uniformního bufferu (*uniform buffer object* viz [23]). Buffer obsahuje pole struktur, které využívají stejné rozvržení položek jako struktura `LightProperties`. Tento layout splňuje

[7] Typ zdroje lze libovolně měnit nastavením konstanty do atributu `LightProperties::type`

standard `std140` (viz [22]) a tudíž umožňuje blokový přesun dat do bufferu, protože adresy jednotlivých atributů jsou známé předem. Velikost pole v bufferu odpovídá maximálnímu počtu využívaných světelných zdrojů. Při velikosti struktury 64B lze do bufferu umístit nejméně 256 položek[8].

V dnešní době se ve velké míře využívá možnosti tvorby komplexních 3D modelů pomocí nástrojů jako například 3ds Max, Maya, Blender ad. Výsledky takovéto činnosti se ukládají do souborů se speciálními formáty[9]. Jeden z cílů frameworku představuje automatizace načtení scény právě z takových souborů. Tuto činnost lze provést voláním jediné metody, viz úsek kódu 5.3. Uživatel se tak nemusí zabývat konverzí dat a může v rychlosti přejít k vytváření efektů.

```
auto sceneManager = std::make_shared<SceneManager>();
sceneManager->LoadModel("model.obj");
```

Zdrojový kód 5.3: Načtení 3d modelu ze souboru *model.obj*.

Správce scény však pouze uchovává výsledný načtený model. Pro proces načítání využívá *správce zdrojů* (třída `ResourceManager`). Ten provádí zpracování geometrie modelu a dále načtení a transformaci materiálů.

## Zpracování geometrie

Správce zdrojů nejprve předá soubor pro načtení knihovně *Assimp*. Model zde bude rozdělen na samostatně vykreslitelné jednotky, tj. *meshe* (viz podkapitola 4.1). Za účelem získání vhodné reprezentace dat jsou dodatečně provedeny následující kroky:

- Všechny tvary jsou převedeny na trojúhelníky nebo odstraněny.
- Indexy pro skládání polygonů jsou vytvořeny.
- Pokud mesh neobsahuje normály, budou vygenerovány (volitelně také tangenty).

Výsledkem je tedy vždy seznam vrcholů a indexů. Vzhledem k tomu, že ve větších modelech množství polygonů sdílí vrcholy navzájem, nabízí se indexovaný způsob vykreslování jako ideální varianta. Načtená data následně framework transformuje do podoby vhodné ke zpracování pomocí OpenGL.

Každý mesh využívá vlastní *vertex array object* (viz [23]) pro uložení stavu potřebného pro jeho vykreslení. Přítomné atributy vrcholů jsou sloučeny do spojité paměti, nakopírovány do *vertex bufferu* a aktivovány jako vstup do vertex shaderu. Stejně tak indexy jsou umístěny do *element bufferu*. Vrcholy mohou obsahovat některé z následujících atributů:

- Pozice
- Normála
- Tangenta
- Texturovací souřadnice

[8] Implementace OpenGL musí podporovat velikost uniformního bufferu nejméně 16kB
[9] Hovoříme o 3D souborových formátech (např. Collada, Wavefront Object aj.)

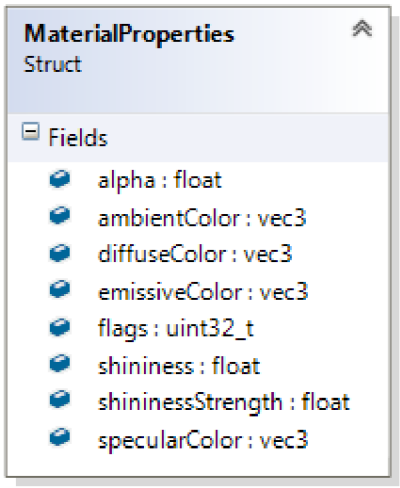
- Barva (až 4×)

Pozice a normála jsou však vždy přítomny. Za účelem umístění objektu do scény lze na celý model nebo samostatné meshe dále aplikovat lineární transformace, tj. posunutí, rotaci a změnu měřítka (scale).

## Zpracování materiálů

Každý mesh je asociován s materiálem, který určuje vlastnosti povrchu objektu využitelné při jeho stínování. Definici materiálu může uživatel měnit za běhu a tím přizpůsobit vizuální stránku aplikace svým potřebám. Mezi dvěma i více meshi ve vhodných případech dochází ke sdílení stejného materiálu. Omezí se tak četnost přenosů dat na grafickou kartu, které jsou nutné pro střídání materiálů v shaderech. Třída `Material` si pro tento účel mimo jiné uchovává také seznam všech meshů, které daný materiál využívají. To dovoluje aplikovat princip vykreslování po materiálech zmíněný v podkapitole 3.4.

Materiál se skládá z parametrů a textur. Parametry obsahují především barvy určené pro stínování netexturovaného objektu, průhlednost (*alpha*) a koeficienty matnosti/lesklosti povrchu (*shininess*). Uživatel má možnost nastavit až 24 příznaků (*flags*) s cílem odlišit materiály v shaderu. Dalších 8 příznaků[10] (bity 24 – 31) indikuje přítomnost nebo nepřítomnost textur (viz dále). Hodnoty příznaků získáme pomocí bitových operací. Kompletní soupis atributů lze vidět na diagramu struktury `MaterialProperties` na obrázku 5.3.




Obrázek 5.3: Diagram tříd pro strukturu `MaterialProperties`.

Parametry materiálu využívají stejný způsob uložení v uniformním bufferu jako světla (popis začíná na straně 33). Narozdíl od světelných zdrojů však buffer vždy obsahuje pouze jedinou instanci příslušné struktury. Celkově zabírá pouze 64B paměti.

Druhou část definice materiálů tvoří textury. Jejich načtení probíhá automaticky pomocí knihovny *DevIL* během zpracování materiálů. Textury, jejichž zdrojem je stejný soubor, sdílí datové struktury (např. difúzní a ambientní textura často bývají identické). Uživatel může vždy nechat načíst další textury, popř. uložit stávající do souborů. K tomu slouží

[10] Příznaky celkově zabírají 32 bitů (4 byty) v atributu s datovým typem `uint` (GLSL)

třída `Texture2D`. Manipulace s texturami nevyžaduje vazbu na materiál. Lze tak například provést i konverze souborů do jiného obrazového formátu, jak demonstruje úsek kódu 5.4. Programová dokumentace obsahuje seznam podporovaných formátu.

```
Texture2D image;
image.LoadFromFile("original.jpg");
image.SaveToFile("result.png");
```

Zdrojový kód 5.4: Konverze obrazových formátů pomocí třídy `Texture2D`.

Každá textura uchovává informace o svých rozměrech, interním formátu a počtu barevných kanálů. Implicitně zůstává nastavena bilineární filtrace, popř. trilineární, pokud je požadováno využití mipmap (viz [23]). Filtraci i jiné parametry textur lze volitelně přenastavit[11].

Materiály podporují následující typy textur:

1. Difúzní
2. Ambientní
3. Spekulární
4. Emisní
5. S koeficientem *shininess*
6. Alpha
7. Normálová mapa
8. Výšková mapa

Typy 1 až 4 slouží jako barvy pro výpočet složek osvětlovací rovnice. Společně s typy 5 a 6 nahrazují parametry definované ve struktuře `MaterialProperties`. *Normálová mapa* a *výšková mapa* se využívají u efektů, které modifikují atributy vrcholů (např. *bump mapping* [23]).

Materiál může obsahovat více textur stejného typu, kde každá instance představuje další vrstvu. Zde se implementace odlišuje od návrhu (viz podkapitola 4.6). Navrhovaný systém skládání vrstev za běhu v shaderech s pomocí podprogramů OpenGL byl shledán jako zbytečně komplikovaný a neefektivní. Vzhledem k tomu, že se textury za běhu nemění, nemá smysl kombinovat hodnoty texelů při každém vyvolání fragment shaderu. Místo toho byl adaptován systém pro tzv. *offline* zpracování textur. Všechny vrstvy jsou pomocí blendingu spojeny do jediné textury, která je následně opakovaně využívána v shaderech. Zároveň tak dojde k uvolnění některých texturovacích jednotek pro jiné účely.

Každá textura má definovánu operaci pro připojení hodnoty texelu k výsledku (sčítání, násobení, atd.) a násobící koeficient. Příspěvek se připojí k cílové hodnotě pomocí vztahu:

$$vysledek = vysledek \ \langle op \rangle \ (koeficient * hodnota\,texelu) \tag{5.1}$$

kde *op* představuje zmíněnou operaci.

[11] Jedná se o parametery odpovídající argumentům funkce `glTexParameteri`, viz [22]

## 5.3 Výstavba efektu

Tvorba efektu začíná vždy u třídy `Effect`. Lze pracovat přímo s instancí této třídy, nebo vytvořit přenositelnou definici pomocí dědičnosti. Každý efekt má určen svůj název, především pro rozeznání v textových výpisech. Unikátní[12] číselný identifikátor pak odlišuje jednotlivé instance efektů. Pomocí lambda funkcí lze aktualizovat dynamicky se měnící datové složky podstatné pro daný efekt.

Samotná třída `Effect` nehraje význačnou roli ve funkčnosti efektu, ale spíše poskytuje kontejner pro shromáždění *vykreslovacích průchodů* (dále jen průchod/y). Ty jsou řazeny do sekvence a při vykreslování vykonávány postupně. Předtím ovšem musí být všechny průchody přeloženy, tj. jejich definice převedena na kód shaderů a pomocné datové struktury, které pak poslouží pro konstrukci výsledného snímku. Definice průchodů lze sestavit z prvků navržených v podkapitole 4.2 a níže popsaných z hlediska implementace.

Jako základ vykreslovacího průchodu slouží abstraktní třída `RenderingPass`, jejíž metody by měly představovat hlavní cíl soustředění uživatele frameworku (detailní popis se nachází v programové dokumentaci). V praxi lze využít dvě implementace této třídy s názvy `RenderingPass3D` a `RenderingPass2D`, které zpracovávají rozdíly mezi základními typy průchodů (2D a 3D, viz podkapitola 3.2). Úsek kódu 5.5 znázorňuje kostru efektu se třemi vykreslovacími průchody.

```
auto effect = std::make_shared<Effect>("testing effect");

// 1. průchod, 3D průchody potřebují znát kameru a správce scény
auto pass1 = std::make_shared<RenderingPass3D>(sceneCamera, sceneManager);

// 2. průchod
auto pass2 = std::make_shared<RenderingPass3D>(sceneCamera, sceneManager);

// 3. průchod, tentokrát typu 2D - např. jistá forma post-processingu
auto pass3 = std::make_shared<RenderingPass2D>();

// přidání průchodů do efektu
effect.AddRenderingPass(pass1);
effect.AddRenderingPass(pass2);
effect.AddRenderingPass(pass3);

// překlad efektu (a tím všech vykreslovacích průchodů)
effect.Compile();
```

Zdrojový kód 5.5: Úkázka kostry implementace efektu.

## Definice prvků vykreslovacího průchodu

Následující definice se vztahují k bázové třídě `RenderingPass`, pokud není uvedeno jinak.

### Uniformní proměnné

Struktura `Uniform` zastupuje uniformní proměnné v shaderech. Jedná se o šablonu, jejímž parametrem je konstanta zastupující datový typ proměnné (např. `GL_FLOAT`). Reprezentace datových typů konstantami používají i ostatní prvky definic průchodů. Důvod pro využití šablon v tomto případě představuje rozlišnost argumentů variant funkce `glUniform`, která slouží pro kopírování dat na grafickou kartu.

[12] Zajištěno pomocí atomického čítače

Proměnnou lze umístit do vertex shaderu, fragment shaderu nebo obou, v tom případě se ovšem jedná o jednu sdílenou instanci. Aby bylo možné hodnotu aktualizovat, musí být znám zdroj dat v paměti. Není vhodné předávat adresu proměnné, která bude později přemístěna (např. položky v kontejneru `std::vector`). Podporovány jsou také statická pole (počet položek znám za překladu) s tím, že pole o velikosti 1 bude převedeno na skalár. Úsek kódu 5.6 demonstruje přidání uniformní proměnné k definici vykreslovacího průchodu.

```
auto pass = std::make_shared<RenderingPass2D>();

...

// náhodný vektor, zdroj dat pro uniformní proměnnou
glm::vec3 randomVector = glm::linearRand(0.0f, 1.0f);

// přidání uniformní proměnné s datovým typem vec3 do fragment shaderu
pass->AddUniform<GL_FLOAT_VEC3>(
  "random_vector",        // identifikátor proměnné
  VISEGL_SHADER_FRAGMENT, // cílový shader
  &randomVector);         // zdroj dat
```

Zdrojový kód 5.6: Definice uniformní proměnné.

### Interpolované proměnné

Mnoho grafických algoritmů využívá interpolaci atributů mezi vrcholy polygonů. Často se jedná o optimalizace přenášející segmenty výpočtů do vertex shaderu[13]. Vyjádření takových proměnných dovoluje struktura `Interpolated`. Mezi její atributy patří především datový typ (vyjádřený konstantou, viz výše) a název identifikátoru. Pro spárování proměnných mezi vertex shaderem a fragment shaderem se ale využívá pozice (*location*, viz [22]), nikoliv identifikátor.

Uživatel může definovat 2 funkce pro transformaci proměnné před a po interpolaci. Jejich syntaxe podléhá pravidlům jazyka GLSL. Ve frameworku jsou uloženy ve formě řetězce a ke kontrole dochází až při překladu vygenerovaných shaderů. Ukázku definice interpolované proměnné a jejich transfomačních funkcí znázorňuje úsek kódu 5.7.

Transformace ve vertex shaderu slouží pro vytvoření počáteční hodnoty proměnné. Jedná se tedy o podstatnou součást kódu. Návratový typ funkce se musí shodovat s datovým typem proměnné, název funkce kopíruje identifikátor proměnné s prefixem `tout_`.

Naopak transformace na vstupu fragment shaderu není povinná a uživatel ji může vynechat předáním prázdného řetězce. V tom případě bude dostupná hodnota převzata z procesu interpolace. Na druhou stranu lze tento mechanismus využít pro definici části implementovaného algoritmu, případně pro korekci hodnoty po interpolaci (např. normály je často třeba znovu normalizovat). Syntaxe funkce se podobá tvaru užitému ve vertex shaderu (viz výše). Odlišnosti představuje změna prefixu na `tin_` a přidání jediného parametru s datovým typem proměnné.

3D průchody poskytují snadnější způsob interpolace pro atributy vrcholů (seznam v podkapitole 5.2). Pozice, normály a tangenty navíc mohou využít předdefinované transformační funkce s následující sémantikou:

- Pozice jsou ve vertex shaderu transformovány do souřadného systému světa nebo kamery[14].

[13] Většinou dochází k četnějšímu vyvolání fragment shaderu než vertex shaderu

[14] Známější jsou anglické názvy *world space* a *view space*

- Normály a tangenty jsou ve vertex shaderu vynásobeny příslušnou transformační maticí a normalizovány v obou shaderech.

```
auto gpass = std::make_shared<RenderingPass3D>(lightCamera, sceneManager);

...

// transformace ve vertex shaderu
std::string vsTransform = R"(
vec4 tout_lposition()
{
  return light_vp_matrix * model_matrix * vec4(position, 1.0);
})";

// transformace ve fragment shaderu
std::string fsTransform = R"(
vec4 tin_lposition(vec4 lposition)
{
  lposition.xyz /= lposition.w;
  lposition.xyz = 0.5 * lposition.xyz + 0.5;
  return lposition;
})";

// přidání interpolované proměnné s definovanými transformačními funkcemi
gpass->AddInterpolated("lposition", GL_FLOAT_VEC4, vsTransform, fsTransform);

// přímá interpolace atributů vrcholu s využitím předdefinovaných transformací
gpass->AddInterpolatedGeometry(VISEGL_VERTEX_POSITION); // pozice
gpass->AddInterpolatedGeometry(VISEGL_VERTEX_NORMAL);   // normála
gpass->AddInterpolatedGeometry(VISEGL_VERTEX_TEXCOORD); // texturovací souřadnice
```

Zdrojový kód 5.7: Definice interpolované proměnné (součást efektu *shadow mapping*).

### Funkce

Přestože framework generuje kód shaderů automaticky, uživatel by měl doplnit části související s implementovaným algoritmem. Pro tento účel lze definovat libovolné funkce se syntaxí jazyka GLSL. Ty následně mohou být umístěny do vertex shaderu, fragment shaderu nebo obou. Každá funkce má určenu svou prioritu, která se využije při skládání efektů (viz podkapitola 5.4).

Přidání funkce vyžaduje 3 řetězce představující atributy struktury `ShaderFunction`:

- Název
- Prototyp (hlavička)
- Tělo (kompletní kód)

Tělo by mělo obsahovat prototyp a ten by měl obsahovat název. Framework ovšem neprovádí analýzu kódu, proto se chyby projeví až při překladu. Prototyp přidává možnost volat funkce navzájem[15]. Název umožňuje kontrolu kolizí jmen. Úsek kódu 5.8 uvádí příklad definice funkce.

[15]Rekurze však není podporována v jazyku GLSL

```
auto pass = std::make_shared<RenderingPass2D>();

...

// přidání funkce
pass->AddShaderFunction(
  "diffuse_term",                                  // název
  "vec3 diffuse_term(vec3, vec3, vec3)",           // prototyp
  R"(
vec3 diffuse_term(vec3 color, vec3 N, vec3 L)
{
  float dp = max(0.0, dot(L,N));
  return dp * lights[0].intensity * color;
})",                                               // tělo
  VISEGL_SHADER_FRAGMENT,                          // cílový shader
  VISEGL_PRIORITY_AVERAGE);                        // priorita
```

Zdrojový kód 5.8: Definice funkce (pro výpočet difúzní složky osvětlení).

### Textury

Textury poskytují způsob předávání dat mezi vykreslovacími průchody. Třída `RenderTarget` dovoluje vytvořit instanci s vhodným formátem a volitelně i multisamplingem. Každá textura musí vlastnit název umožňující její referenci v shaderech.

Vstupní textury jsou mapovány na `samplery` v jazyku GLSL. Hodnotu texelu lze získat pomocí GLSL funkcí `texture`, `texelFetch` aj. (viz [13]) s pomocí texturovacích souřadnic. Sampler může být k dispozici ve fragment shaderu nebo ve vertex shaderu.

Pro připojení color bufferu na výstup framebufferu (viz podkapitola 3.1) definujeme výstupní textury. Pomocí sekvence GLSL kódu uživatel určí hodnotu zapsanou do textury. Kód by tedy měl obsahovat ve svém závěru přiřazení s názvem textury na levé straně výrazu. Dodatečně je možno uvést parametry pro blending nebo maskování jednotlivý barevných složek RGBA, které budou pro příslušnou texturu nastaveny. Úsek kódu 5.9 demostruje vytváření textur a jejich předávání mezi vykreslovacími průchody.

Framework podporuje rovněž formáty pro uložení hloubky a případně hodnoty stencil. Takovou texturu lze využít k příslušným testům u 3D průchodů nebo i jako vstup. Připojení depth bufferu není povinné. Pokud průchod vyžaduje provádění testů, neexistující textura bude vytvořena automaticky.

```
// vytvoření textury
auto texture = std::make_shared<RenderTarget>("texture_name");
texture->CreateRenderTexture(800, 600, GL_RGB, 1); // 1 = počet vzorků

// 1. průchod
auto pass1 = std::make_shared<RenderingPass2D>();
pass1->AddOutputTexture(
  texture,
  R"(
  vec3 result = diffuse_color * lights[0].intensity;
  texture_name = result;)"                         // kód pro zápis hodnoty
);

// 2. průchod
auto pass2 = std::make_shared<RenderingPass2D>();
pass2->AddInputTexture(texture, VISEGL_SHADER_FRAGMENT);
```

Zdrojový kód 5.9: Definice vstupní/výstupní textury.

### Vykreslovací kontext

Každý vykreslovací průchod spravuje instanci struktury `RenderingContext`, která obsahuje atributy pokrývající podstatné stavové proměnné OpenGL. Kontext nepřidává žádný kód do shaderů, ale ovlivňuje výsledek vykreslování. Nastavení hodnot probíhá automaticky nebo voláním příslušných metod (viz programová dokumentace). Použitelné atributy zahrnují:

- **Viewport** označuje pozici a rozměry vykreslované plochy ve framebufferu.
- **Culling** ovlivňuje, které strany polygonů budou zahozeny (přední/zadní, oboje, žádné).
- Zapnutí/vypnutí **testu hloubky**.
- Zapnutí/vypnutí **zápisu hloubky**.
- Zapnutí/vypnutí **stencil testu**.
- **Parametry pro stencil test**.
- Zapnutí/vypnutí **scissor testu** pro případné omezení viewportu.
- **Viewport pro vymezení scissor testu**.
- **Multisampling** určuje, zda-li výstupní textury obsahují více než 1 vzorek.

U 2D průchodů nelze nastavit *culling*, ani zapnout test/zápis hloubky a stencil test. Zmíněné omezení vychází z principu vykreslování ve 2D, které se blíží spíše zpracování obrazu. Naopak 3D průchody dovolují dodatečně využít předčasný test hloubky (*early z-test*), který umožňuje na základě výsledku zahodit fragment ještě před fragment shaderem.

## Překlad vykreslovacího průchodu

Aby bylo možné pracovat s průchodem, musí ten nejprve projít procesem překladu. Zde dochází k transformaci uživatelem vytvořené definice na struktury OpenGL, které poslouží k realizaci odpovídajícího výsledku. Postup pak nebude opakován až do té doby, než se opět změní definice průchodu. Překlad sestává ze 3 hlavních kroků:

1. Zpracování a kontrola definice
2. Generování a překlad shaderů
3. Inicializace struktur OpenGL

První fáze tedy především kontroluje chyby v definici, které by způsobily problemy později. Framework může pracovat pouze s dostupnými informacemi, neprovádí kontrolu předaného kódu atd. Jedná se především o zjištění kolize jmen identifikátorů proměnných, funkcí a textur. Mezi další provedené úkony patří:

- Ověření, že název funkce se nerovná `main`, což je vyhrazené jméno pro vstupní bod shaderu.

- Kontrola počtu vzorků ve výstupních texturách. Framebuffer nepodporuje textury s odlišným nastavením multisamplingu. Toto omezení platí i pro depth buffer.
- 2D průchod musí používat alespoň 1 vstupní a 1 výstupní texturu. I bez splnění tohoto pravidla by byl průchod funkční, ovšem negeneroval by žádný použitelný výsledek. Princip 2D průchodu představuje zpracování dat ze vstupu na výstup.
- Neexistuje-li u 3D průchodu depth buffer a je vyžadován test/zápis hloubky nebo stencil test, bude definice rozšířena o takový buffer se standardním formátem (32b pro hloubku, nebo 24b + 8b pro stencil hodnotu).

V tuto chvíli je definice předáná systému pro generování shaderů. Cílový jazyk představuje GLSL verze 4.2 (viz [13]), který odpovídá použité verzi OpenGL 4.2. Framework postupně umisťuje části definice do kódu podle pořadí určeného návrhem (viz podkapitola 4.3). Všechny symboly (proměnné, textury ad.) jsou dostupné k využití v uživatelských funkcích.

Definice těl funkcí následují až za všemi prototypy, aby bylo možné volat funkce navzájem. Od transformace atributů vrcholu na vstupu vertex shaderu bylo upuštěno z důvodu nedostatečného uplatnění. Rovněž byla odstraněna bitangenta jakožto atribut vrcholu. Tento vektor lze dopočítat pomocí normály a tangenty. Pro interpolované proměnné jsou automaticky volány transformační funkce před koncem vertex shaderu a na začátku fragment shaderu. Pokud uživatel využívá světla, fragment shader obsahuje příslušný uniformní buffer (viz podkapitola 5.2).

Některé části shaderů se u 2D a 3D průchodů liší. 3D průchody pracují ve vertex shaderu s atributy vrcholů a poskytují pro jejich transformaci často využívané matice, např. matice převádějící pozici do souřadného systému kamery (`mv_matrix`). Navíc je zde možnost přidat konstrukce reprezentující materiál, které se skládají z:

- Uniformního bufferu s parametry
- Textur
- Funkcí pro čtení korektní hodnoty

Textury jsou dostupné pouze za přítomnosti texturovacích souřadnic. Funkce vrací hodnotu získanou z textury (je-li k dispozici) nebo parametr. Rozhodování se provádí na základě automaticky nastaveného příznaku.

Úsek kódu 5.10 znázorňuje konstrukce určené pro práci s parameterem *difúzní barva*. Kompletní popis rozhraní materiálů lze nalézt v programové dokumentaci.

2D průchody naproti tomu nevyužívají materiály. Z atributů vrcholu se zde vyskytuje pouze pozice, která je dále interpolována jako texturovací souřadnice (proměnná `texcoord`) určené ke vzorkování textur ve fragment shaderu. Pro usnadění přístupu k sousedním texelům obsahuje kód navíc konstanty `tex_offset_hor` a `tex_offset_vert`, kterými lze souřadnice posunout právě o jeden texel.

Kompletně vygenerovaný kód shaderů je předán překladači GLSL dostupnému přes API OpenGL. Zde budou nalezeny chyby v definici, které nebyly odhaleny dřívější kontrolou (především syntaxe funkcí aj.).

```
layout( binding = 1, std140 ) uniform Material
{
  vec3 diffuse_color;
  ...
  uint flags;
};

// Následující úseky kódu jsou podmíněny ne/přítomností texturovacích souřadnic

// 1) Texturovací souřadnice jsou k dispozici
uniform sampler2D diffuse_texture;

vec3 get_diffuse()
{
  if((flags & (1 << 24)) != 0) return texture(diffuse_texture, texcoord).rgb;
  else return diffuse_color;
}

// 2) Texturovací souřadnice nejsou k dispozici
vec3 get_diffuse()
{
  return diffuse_color;
}
```

Zdrojový kód 5.10: Součásti kódu fragment shaderu související s parametrem materiálu zastupujícím difúzní barvu.

Poslední fáze překladu vykreslovacího průchodu nastává po úspěšném sestavení programu OpenGL. Nyní je třeba inicializovat datové struktury potřebné k vykreslování.

Lokace uniformních proměnných musí být zjištěny za účelem pozdější aktualizace hodnot. Textury rovněž vyžadují přiřazení texturovacích jednotek pro správné mapování v shaderech. Pokud uživatel využívá světla a/nebo materiály, provede se aktivace uniformních bufferů.

Dále framework vytvoří framebuffer a připojí k němu color buffery, u 3D průchodů také depth buffer. Nakonec probíhá kontrola stavu framebufferu.

Uživatel si následně může prohlédnout záznam zkonstruovaný v průběhu překladu. Ten obsahuje zprávy o úspěšnosti jednotlivých fází a případně zmiňuje nalezené problémy. Chyby odhalené překladem kódu shaderů jsou doplněny o výstup překladače GLSL. Záznamy všech průchodů v rámci efektu jsou spojeny dohromady a dostupné přes metody třídy `Effect`.

Následuje příklad záznamu z překladu efektu, jehož druhý průchod obsahuje syntaktickou chybu ve funkci (lze dohledat v kódu):

```
compiling effect: testing_effect

compiling rendering pass: first_pass (3D)
processing definitions ... OK
compiling vertex shader ... OK
compiling fragment shader ... OK
linking program ... OK
processing uniforms ... OK
initializing framebuffer ... OK

compiling rendering pass: second_pass (2D)
processing definitions ... OK
```

```
compiling vertex shader ... FAILED
log:
0(19) : error C1008: undefined variable "positionn"
```

## Provedení efektu

Efekt zprovozníme jeho předáním rendereru (viz podkapitola 5.1) odvozenému od třídy `EffectRenderer`. Postup znázorňuje úsek kódu 5.11.

```
auto renderer = std::make_shared<EffectRenderer>();
...
auto effect = std::make_shared<Effect>();
...
// pokud nebyl efekt přeložen, proběhne zde překlad automaticky
effect->SetActiveEffect(effect);

// snímky budou nyní reflektovat nastavený efekt
```

Zdrojový kód 5.11: Aktivace efektu pro vykreslování.

Vykreslení snímku pak využívá datové struktury a shadery vytvořené při překladu průchodů. Celý proces sestává z následujících kroků:

1. Načtení světel do uniformního bufferu (pokud jsou použity)
2. Pro každý průchod:
   (a) Aktivace vykreslovacího kontextu
   (b) Nastavení blendingu a maskování barevných složek pro výstupní textury
   (c) Řízení je předáno průchodu
      i. Aktivace OpenGL programu
      ii. Aktivace framebufferu a vyčištění připojených textur (pouze pokud nevyužívají blending)
      iii. Provázání textur s texturovacími jednotkami
      iv. Aktualizace hodnot uniformních proměnných
      v. Vykreslení obsahu, řízené podle diagramu na obrázku 4.1
3. Kopie vykresleného obsahu do defaultního framebufferu

Renderer si udržuje instanci struktury `RenderingContext`, která slouží jako cache hodnot stavových proměnných. V kroku 2a dochází vždy nejprve k porovnání hodnoty mezi kontextem v rendereru a průchodu. Pouze pokud jsou hodnoty rozdílné, provede se změna voláním příslušných funkcí OpenGL.

Definované textury mohou mít určenu významovou sémantiku vyjádřenou řetězcem. Tento mechanismus se využívá především u skládání efektů (viz podkapitola 5.4). Aby bylo možné zobrazit výsledek vykreslování, musí uživatel alespoň u jedné výstupní textury (ideálně v posledním průchodu efektu) nastavit sémantiku `"surface"`. Obsah této textury bude dle kroku 3 přesunut do color bufferu v defaultním framebufferu za pomoci OpenGL funkce `glBlitFramebuffer`.

## 5.4 Kombinace efektů

Možnost sloučit výsledky několika efektů představuje jeden z hlavních cílů frameworku. Pro tento účel byla vytvořena třída `EffectLibrary`. Její využití není povinné a uživatel může bez omezení pracovat s oddělenými instancemi efektů. Na druhou stranu kombinování efektů poskytuje příležitost pozorovat rozdíly, které přináší jednotlivé efekty do zobrazení scény.

Třída `EffectLibrary` pracuje s klasickými objekty třídy `Effect`. Uživatel nemusí jejich definici příliš měnit, pouze je vhodné doplnit informace ovlivňující proces skládání (viz dále). O ukládání efektů se stará třída `std::map`, kde jsou instance řazeny podle jejich číselného id pro rychlé vyhledávání. Efekty jsou automaticky přeloženy při jejich přidání, viz úsek kódu 5.12.

```
auto effect1 = std::make_shared<Effect>("effect1");
...
auto effect2 = std::make_shared<Effect>("effect2");
...
EffectLibrary effectLibrary;
effectLibrary->AddEffect(effect1); // uložení a automatický překlad
effectLibrary->AddEffect(effect2); // uložení a automatický překlad
```

Zdrojový kód 5.12: Registrace efektů u třídy `EffectLibrary`.

### Skládání na úrovní efektů

Při skládání efektů dochází ke sloučení jejich vykreslovacích průchodů do výsledného efektu. Jedná se o sekvenční proces, tedy vždy pouze dva efekty mohou být zpracovány zároveň. Pořadí průchodů z pohledu původních efektů musí zůstat stejné. Framework se snaží redukovat počet vykreslovacích průchodů slučováním jejich definic. Pokud to není možné, použije se originální průchod. Výsledný efekt tedy obsahuje sekvenci spojených a původních průchodů.

Podkapitola 3.2 definuje dělení vykreslovacích průchodů na 3 typy. Pro zopakování se jedná o:

- **Pre-pass** neboli předzpracování za účelem získání dat pro pozdější úkony.
- **Surface pass**, který slouží pro výpočet osvětlení/stínování.
- **Post-processing** zajišťující dodatečnou transformaci vykresleného obrazu.

Příslušnost průchodu k jedné z těchto kategorií určuje jeho sémantika[16]. Podle sémantiky jsou posloupnosti spojovaných efektů rozděleny do 3 skupin. Nelze spojit průchody s odlišnou sémantikou a zachovat tak jejich korektní pořadí. Systém pro spojování sekvencí průchodů dostává na vstupu iterátory ohraničující začátek a konec dvou ekvivalentních skupin. Výstup pak představuje sloučená sekvence průchodů, která tvoří základ nového efektu. Vhodné kombinace průchodů jsou zde již nahrazeny spojenými instancemi. Nově vzniklé průchody jsou přeloženy.

Algoritmus pro skládání sekvencí průchodů vyjadřuje následující pseudokód:

[16] Zde se jedná o konstanty, protože existují pouze 3 varianty

```
INIT begin1   // iterátor na začátek první sekvence
INIT end1     // iterátor za konec první sekvence
INIT begin2   // iterátor na začátek druhé sekvence
INIT end2     // iterátor za konec druhé sekvence
INIT result   // výsledná posloupnost

PROCEDURE MergeRanges (begin1, end1, begin2, end2, result)
BEGIN
    FOR ptr1 = begin1 to (end1 - 1)
        SET ptr2 = begin2
        WHILE ptr2 != end2
            CALL CanMerge with ptr1 and ptr2 RETURNING cond
            IF cond THEN
                WHILE begin2 < ptr2
                    result = result + begin2
                    INCREMENT begin2
                CALL Merge with ptr1 and ptr2 RETURNING ntr
                result = result + ntr
                INCREMENT begin2
                BREAK
            ELSE
                INCREMENT ptr2
            END IF
        IF ptr2 == end2 THEN
            result = result + ptr1
        END IF
    WHILE begin2 != end2
        result = result + begin2
        INCREMENT begin2
END PROCEDURE
```

*Poznámky k pseudokódu*:

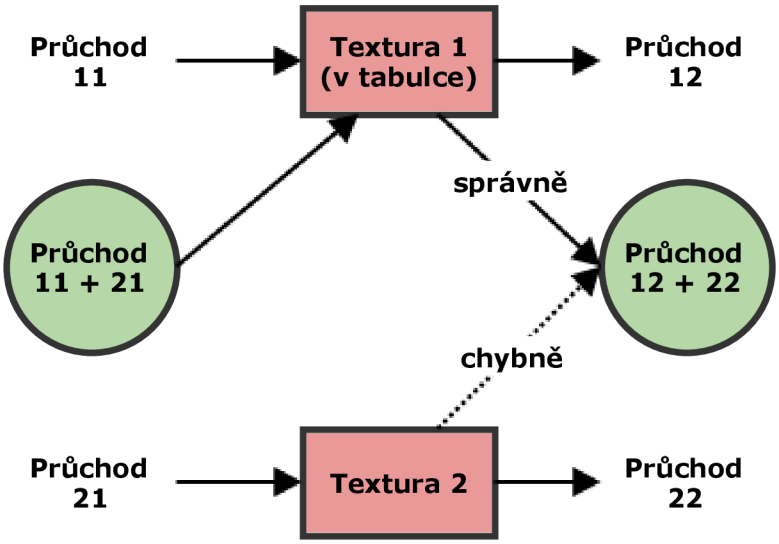
Operátor `+` má v tomto případě význam připojení položky na konec seznamu. Kód naznačuje provedení této operace na iterátorech, ovšem v praxi pracuje s ukazateli na instance průchodů. Dále je zřejmé, že budeme potřebovat rozhodovací funkci pro výběr spojitelných průchodů (`CanMerge`) a funkci, která sloučení provede (`Merge`). Tento problém rozebírá následující text.

## Skládání na úrovní průchodů

Sloučením definic dvou vykreslovacích průchodů do jediného celku lze zkombinovat jejich výsledky a navíc redukovat počet vykreslovacích operací. Ideální řešení představuje sdílení stejných prvků definice a ponechání rozdílných. V některých případech však nemůžeme akceptovat ani jednu z variant, protože dojde ke kolizím, které framework není schopen samostatně vyřešit. Příkladem může být dvojice uniformních proměnných se stejným názvem, ale odlišným datovým typem – zde nelze zajistit zachování funkčnosti a zároveň korektnost

při překladu shaderů. Někdy naopak může být žádoucí ponechat průchod nespojitelný s jiným.

Pro rozlišení funkcí a výstupních textur slouží celočíselná priorita. Shodují-li se tyto prvky v názvu, framework vybere ten s vyšší prioritou a umístí jej do výsledného průchodu. Pokud je priorita stejná, dojde ke kolizi a průchody nepůjdou spojit. Všechny textury navíc mohou mít určenu významovou sémantiku, která povoluje sdílení obsahově stejných instancí. Toto sdílení je kritické při předávání textur mezi průchody. Při chybném nastavení mohou části kódu zpracovávat rozdílná data a výsledek pak nebude korektní. Proto jsou využité výstupní textury registrovány podle jejich sémantiky v tabulce. Vstupní textury se sémantikami jsou nahrazeny položkami z tabulky. Důležitost nastavení správných textur demonstruje obrázek 5.4.




Obrázek 5.4: Demonstrace správného mapování textur při skládání průchodů.

**Prázdné definice průchodů lze spojit vždy, pokud mají stejný typ** (2D/3D) **a sémantiku** (viz výše). Po přidání prvků k definici a modifikaci možných nastavení již nelze tento fakt zaručit. Zajištění spojitelnosti průchodů vyžaduje splnění následujícího seznamu podmínek:

1. Rozměry *viewportu* jsou shodné.
2. Nastavení pro *culling* (viz strana 41) je shodné nebo alespoň jeden z průchodů jej má vypnutý.
3. Nastavení pro vypnutí/zapnutí *testu hloubky* a *zápisu hloubky* je shodné.
4. Ani jeden z průchodů nepoužívá *stencil test*.
5. Ani jeden z průchodů nepoužívá *scissor test* nebo jej používají oba a vymezený viewport je shodný.
6. Ani jeden z průchodů nepoužívá *multisampling* u výstupních textur nebo jej používají oba a počet vzorků je shodný.

7. Shodují-li se *uniformní proměnné* v názvu, musí se shodovat i datový typ a velikost pole.

8. Shodují-li se *interpolované proměnné* v názvu, musí se shodovat i datový typ.

9. Shodují-li se *funkce* v názvu, jsou umístěny v jiných shaderech nebo mají rozdílnou prioritu.

10. *Vstupní textury* mají odlišné názvy od výstupních textur v druhém průchodu. Pokud se vstupní textury shodují v názvu a jsou umístěny ve stejném shaderu, musí se shodovat i jejich sémantika, počet vzorků (multisampling), počet barevných kanálů a buď obě slouží pro uložení hloubky (depth buffer) nebo žádná z nich.

11. Shodují-li se *výstupní textury* v sémantice a tato se nerovná prázdnému řetězci[17], pak se musí shodovat i počet barevných kanálů a maskování barevných složek RGBA. Dále musejí mít textury rozdílnou prioritu a ani jedna nevyužívá blending. Mají-li textury rozdílnou sémantiku, nesmí se shodovat v názvu.

**Tyto podmínky platí pouze pro 3D průchody:**

12. *Kamera* je shodná (stejný objekt).

13. *Správce scény* je shodný (stejný objekt).

14. Transformuje-li jeden z průchodů světla do souřadného systému kamery, musí druhý průchod obsahovat stejné nastavení.

15. Pokud oba průchody obsahují *depth buffer*, pak se musí shodovat v sémantice (je-li definována) a v počtu vzorků.

16. *Interpolované proměnné příslušící atributům vrcholů* obsahují totožné transformační funkce.

Většina podmínek zabraňuje kolizím, které by znemožnily překlad/funkčnost výsledného průchodu, nebo se snaží lehce rozlišit prvky, jejichž sloučení by bylo nežádoucí. Porušení kterékoliv z vypsaných podmínek vede k zamezení možnosti spojit průchody.

Framework provede složení definic dvou průchodů do jediného za předpokladu, že všechny výše zmíněné podmínky jsou splněny. Většina hodnot nastavení a samostatné prvky jsou zkopírovány nebo sdíleny s původními průchody. Sloučitelné prvky se řídí tímto postupem:

- Je-li *culling* vypnut u jednoho z průchodů, pak stejné nastavení platí pro výsledný průchod.
- U počtu využívaných *světel* se vybere maximum.
- *Uniformní proměnné* se shodným názvem jsou sloučeny, včetně jejich umístění do cílových shaderů.
- *Interpolované proměnné* se shodným názvem jsou sloučeny.
- Z *funkcí* se shodným názvem umístěných do stejného shaderu je vybrána ta s větší prioritou.

[17] Sémantika textur je určena řetězcem, pokud není definována uživatelem, má hodnotu prázdného řetězce

- *Vstupní textury* pro něž existuje položka se stejnou sémantikou v tabulce (viz strana 47) jsou nahrazeny touto položkou. Textury se stejným názvem jsou sloučeny do jedné.
- *Výstupní textury* se stejnou sémantikou jsou sloučeny.

**Následující body platí pouze pro 3D průchody:**

- Pro nastavení předčasného testu hloubky (*early depth test*) má přednost jeho vypnutí.
- *Části kódu shaderů určené pro materiály* budou umístěny pokud je alespoň jeden z průchodů využívá.
- Volba *depth bufferu* preferuje přítomnost stencil hodnot, jinak se snaží vybrat formát s největším bitovým rozsahem. Pokud pouze jedna z textur má definovánu sémantiku, pak má tato přednost nehledě na formát.
- *Interpolované proměnné příslušící atributům vrcholů* jsou sloučeny.

## Přepínání efektů

V předchozím textu bylo vysvětleno, jakým způsobem framework skládá efekty dohromady. Tuto činnost vykonává třída `EffectLibrary` vždy, když je některý z efektů aktivován nebo deaktivován. Aktivovaný efekt přidá svou funkčnost k současně využívané kombinaci. Skládání efektů není reverzibilní operace. Proto nelze při jejich deaktivaci pouze odstranit část definice, ale je potřeba vytvořit kombinaci neobsahující odstraňovaný efekt.

Spojování vykreslovacích průchodů vyžaduje kopírování nebo sdílení datových složek a přegenerování kódu shaderů (včetně překladu). Jsou-li efekty aktivovány opakovaně, dochází ke zbytečnému opakování těchto úkonů, nehledě na časovou režii. Framework proto využívá cache kombinací efektů, do níž jsou uloženy všechny výsledky operace skládání. Její položky nezabírají velké množství paměti, neboť množství dat je sdíleno. Přesto uživatel může omezit velikost cache (až na 0 položek = vypnutí cache).

Systém pro přepínání efektů vyhledává potřebné kombinace v cachi a ukládá zpět výsledky. Není-li k dispozici shoda, je možné použít i položku obsahující podmnožinu žádaných efektů, tj. nejlepší shodu. Mechanismus aktivace/deaktivace efektů pracuje dalé popsaným způsobem:

1. Neexistuje-li efekt, nebo již byl aktivován/deaktivován, neprovede se nic.
2. Na základě prováděné akce se:
   (a) použije aktivovaný efekt beze změny, pokud není aktivní žádný jiný efekt.
   (b) pouze zruší aktivace deaktivovaného efektu, pokud se jedná o jediný aktivní efekt.
3. Vytvoří se seznam požadovaných aktivních efektů.
4. Vyhledá se nejlepší shoda v cache.
5. Připojí se zbývající efekty.
6. Výsledek se uloží do cache.

Uživatel si může zkontrolovat záznam operace skládání efektů. Ten obsahuje pro zúčastněné efekty údaje o tom, zda-li byly kontrolované dvojice průchodů spojeny nebo ne, a zdůvodnění tohoto výsledku (uvedení porušené podmínky).

Úsek kódu 5.13 demonstruje kombinování efektů pomocí jejich aktivace/deaktivace.

```
auto effect1 = std::make_shared<Effect>("effect1");
...
auto effect2 = std::make_shared<Effect>("effect2");
...
auto effect3 = std::make_shared<Effect>("effect3");
...

EffectLibrary effectLibrary;

// uložení efektů - nyní budou k dispozici pro aktivaci/deaktivaci
effectLibrary->AddEffect(effect1);
effectLibrary->AddEffect(effect2);
effectLibrary->AddEffect(effect3);

// aktivace efektu
effectLibrary->ActivateEffect(effect1->GetEffectID());

// aktivace druhého efektu - zde dojde ke spojení "effect1" a "effect2"
effectLibrary->ActivateEffect(effect2->GetEffectID());

// deaktivace efektu
effectLibrary->DeactivateEffect(effect1->GetEffectID());

effectLibrary->ActivateEffect(effect3->GetEffectID());

// efekt je již aktivní - nic se neprovede
effectLibrary->ActivateEffect(effect3->GetEffectID());

// reaktivace efektu
effectLibrary->ActivateEffect(effect1->GetEffectID());
```

Zdrojový kód 5.13: Manipulace efektů pomocí třídy `EffectLibrary`.

# Kapitola 6

# Zhodnocení implementace

Framework umožňuje vytvořit množství vizuálních efektů běžně využívaných v 3D aplikacích. Jako příklad, ale nikoliv omezení, mohou sloužit efekty popsané v kapitole 2 (námátkově shadow mapping, ambient occlusion a další). Jako princip zde slouží více-průchodové vykreslování. Uživatel může k definici průchodů využít prvky, jako jsou např. textury, funkce, uniformní proměnné, světelné zdroje, materiály ad. Příslušné konstrukce jsou pak dostupné v automaticky generovaném kódu shaderů. Vzhledem k oddělenému principu definic může v některých případech dojít k neefektivnímu přepočítávání mezivýsledků.

Mezi další přednosti patří redukce obtížnosti načtení scény. Zpracování souboru s 3D modelem lze provést voláním jediné metody. Vykreslované objekty scény následně slouží pro demonstraci efektů. Vykreslování probíhá automaticky v režii frameworku.

Jednotlivé definice efektů mohou být sloučeny dohromady za účelem získání kombinovaného výsledku. Tento proces probíhá na úrovní vykreslovacích průchodů a uživatel jej ovlivňuje úpravou atributů jednotlivých prvků. Většina prvků (textury, proměnné ad.) jsou sdíleny, proto nedochází k výráznému nárůstu paměťové náročnosti. Efekty lze aktivovat nebo deaktivovat za běhu programu a sledovat jejich přínos na zobrazené scéně.

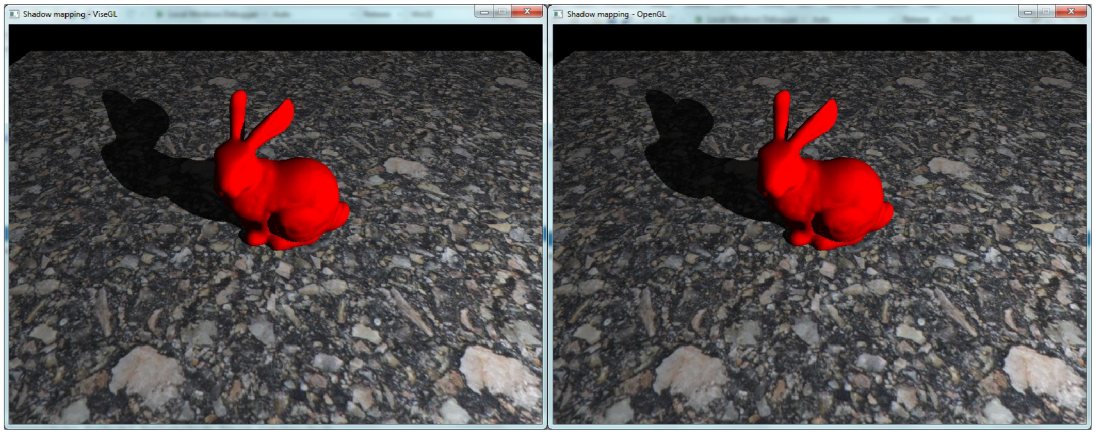
Framework byl napsán v jazyce C++. Z důvodu zaměření na standard C++11 [26] musí být překlad proveden na novější verzích překladačů, konkrétně alespoň: *GCC verze 4.7* nebo *Visual C++ 12.0* (2013). Produkt je tedy přenositelný mezi operačními systémy *Microsoft Windows* a *Linux*. Výsledek sestává z hlavičkových souborů a sdílené knihovny určené k sestavení programu. Uživatel používá při práci výhradně třídy frameworku a je odstíněn od volání funkcí knihovny OpenGL. Cílovou hardwarovou platformu tvoří grafické karty s podporou OpenGL verze 4.2. Shadery jsou generovány v jazyku GLSL. Pro přidání vlastních funkcí do shaderů musí znát uživatel příslušnou syntaxi v GLSL, která se ovšem velmi blíží jazyku C, s rozšířením o pomocné funkce (viz [13]).

## 6.1 Testování

Následující text obsahuje výsledky testů, které byly provedeny za účelem ověření funkčnosti frameworku. Testy byly provedeny na počítačové soustavě s procesorem *Intel Core i7-2630QM* o frekvenci 2.00 GHz a grafickou kartou *GeForce GT 525M*. Jako cíl vykreslování sloužil framebuffer s rozměry $800 \times 600$ pixelů. Měření byla realizována na platformě *Microsoft Windows* využitím nativní funkce `QueryPerformanceCounter`. Rozlišení časovače bylo ověřeno jako *513 ns*;

### Test1 – srovnání implementace efektu s a bez využití frameworku

Pro tento test byly vytvořeny 2 verze efektu *shadow mapping* se stejnou funkčností. První varianta využívala implementovaný framework a její kompletní zdrojový kód lze nalézt na straně 53. Druhá varianta, jejíž kód zde není uveden z prostorových důvodů, byla napsána pouze pomocí knihovny OpenGL. Na obrázku 6.1 lze vidět, že obě verze produkují prakticky totožný výsledek.




Obrázek 6.1: Srovnání výsledků verze využívající framework (vlevo) a bez něj (vpravo).

Hlavní cíl testu představovala analýza náročnosti implementace efektu z pohledu uživatele. Byl změřen počet řádků[1] v původním kódu obou verzí a porovnány potřebné počáteční znalosti. Souhrn zjištěných poznatků obsahuje tabulka 6.1.

| | **Framework** | **OpenGL** |
|---|---|---|
| **Řádky kódu pro definici efektu** | 35 | 118 |
| **Řádky kódu pro provedení efektu** | 2 | 49 |
| **Závislost na jiných knihovnách** | Ne | Ano |
| **Vyžadovaná znalost OpenGL** | Ne | Ano |
| **Vyžadovaná znalost GLSL** | Jen funkce | Ano |
| **Nutnost vytvořit kompletní kód shaderů** | Ne | Ano |
| **Ošetření chybových stavů** | Automaticky | Ručně |

Tabulka 6.1: Poznatky získané analýzou definic obou verzí implementace.

Do měření nebyly zahrnuty části kódu pro vytvoření okna, načtení scény atd. Z tabulky je zřejmé, že naprogramování efektu s pomocí frameworku povede na ušetření času a uživatel se obejde bez detailní znalosti knihovny OpenGL. K podstatným výhodám patří i automatické generování shaderů a automatické ošetření chybových stavů (vynecháno u druhé verze jako optimalizace). Framework dále redukuje práci tím, že odstraňuje nutnost při vývoji připojit externí knihovny pro vedlejší úkony.

[1] Jedná se pouze o relativní odhad

```
auto effect = make_shared<Effect>("shadow mapping");

auto lightCamera = make_shared<StaticCamera>();
lightCamera->SetViewport(1024, 1024);
lightCamera->SetPositionAndTarget(light->GetPosition(), light->GetDirection());
lightCamera->SetPerspective(
  2.0f * light->GetSpotlightCutoff(), 1.0f, 1.0f, 21.0f);

auto pass1 = make_shared<RenderingPass3D>(lightCamera, sceneManager);
pass1->SetCulling(GL_FRONT);

auto shadowMap = make_shared<RenderTarget>("shadow_map");
shadowMap->CreateDepthStencilTexture(1024, 1024, 32);
pass1->SetDepthStencilTexture(shadowMap);

auto pass2 = make_shared<RenderingPass3D>(sceneCamera, sceneManager);
pass2->UseLights(1);
pass2->EnableMaterials(true);
pass2->AddInputTexture(shadowMap, VISEGL_SHADER_FRAGMENT);

mat4 lightVPMatrix = lightCamera->GetViewProjectionMatrix();
pass2->AddUniform<GL_FLOAT_MAT4>(
  "light_vp_matrix", VISEGL_SHADER_VERTEX, &lightVPMatrix);

pass2->AddInterpolatedGeometry(VISEGL_VERTEX_POSITION);
pass2->AddInterpolatedGeometry(VISEGL_VERTEX_NORMAL);
pass2->AddInterpolatedGeometry(VISEGL_VERTEX_TEXCOORD);

pass2->AddInterpolated("lposition", GL_FLOAT_VEC4, R"(
  vec4 tout_lposition()
    { return light_vp_matrix * model_matrix * vec4(position, 1.0); } )", R"(
  vec4 tin_lposition(vec4 p)
    { p.xyz /= p.w; p.xyz = 0.5 * p.xyz + 0.5; return p; } )");

pass2->AddShaderFunction("shadow", "float shadow()", R"(float shadow(){
  return (texture(shadow_map, lposition.xy).r < lposition.z)? 0.0 : 1.0;
})", VISEGL_SHADER_FRAGMENT);

auto colorBuffer = make_shared<RenderTarget>("color_buffer");
colorBuffer->SetSemantics("surface");
colorBuffer->CreateRenderTexture(800, 600, GL_RGBA);

pass2->AddOutputTexture(colorBuffer, R"(
  vec3 L = normalize(lights[0].position - position);
  vec3 ambient_term = 0.1 * get_diffuse();
  vec3 diffuse_term =
    shadow() * get_diffuse() * lights[0].intensity * max(0.0, dot(L,normal));
  color_buffer = vec4(ambient_term + diffuse_term, 1.0); )");

effect->AddRenderingPass(pass1);
effect->AddRenderingPass(pass2);
```

Zdrojový kód 6.1: Definice efektu *shadow mapping* s využitím frameworku.

Poslední část testu zahrnovala srovnání výkonnosti obou variant. Pro tento účel byla změřena doba vykreslení 7 nezávislých snímků pomocí každé z verzí. Výsledky znázorňuje tabulka 6.2, hodnoty jsou v mikrosekundách.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Framework** | 21,5523 | 24,1180 | 25,6575 | 23.6049 | 23.6049 | 23.0917 | 23.6049 |
| **OpenGL** | 20,5260 | 18,9865 | 26,6838 | 20,0128 | 20,0128 | 22,5786 | 17,9602 |

Tabulka 6.2: Výsledky měření doby vykreslení jedno snímku pro obě verze.

Průměrná doba provedení snímku za pomoci frameworku byla **23,604 $\mu$s**, zatímco u druhé verze se jednalo o **20,9658 $\mu$s**. Rozdíl tedy činil **2,6382 $\mu$s**, což je **12,58335%**. Vezmeme-li v potaz možné odchylky v různých definicích efektů, **můžeme odhadnout režii frameworku vzhledem k vykreslení snímku na cca 10 – 15 %**.

### Test 2 – analýza doby aktivace/deaktivace efektů

Cíl testu představovalo ověření schopnosti frameworku kombinovat definice efektů za běhu programu. Pro tento účel bylo vytvořeno 10 efektů, kde každý obsahuje 5 vykreslovacích průchodů. Definice průchodů se dále skládály z následujích prvků:

- 1 – 2 vstupní textury
- 1 – 2 výstupní textury
- 2 interpolované proměnné
- 2 uniformní proměnné
- 2 funkce

Ostatní prvky a nastavení nebyly zahrnuty, protože jejich přítomnost by nepřinesla podstatnou změnu do výsledku. Mezi efekty byly zastoupeny průchody všech typů (2D/3D) a rozdělení (viz podkapitola 3.2). Možnost spojitelnosti průchodů byla ponechána náhodnému výběru. Výsledek vykreslování nebyl podstatný pro měření, proto nebyl sledován.

Test měl rovněž prokázat účinnost ukládání kombinací efektů do cache a následného vyhledávání.

#### Případ 1) Aktivace efektů v daném pořadí a jejich deaktivace v opačném pořadí

| Id efektu | Akce | Zásah v cache | Čas | SE | SP | AE | AP |
|---|---|---|---|---|---|---|---|
| 0 | Aktivace | N/A | 3,0788 $\mu s$ | 0 | 0 | 1 | 5 |
| 1 | Aktivace | Výpadek | 6,1536 $ms$ | 1 | 4 | 2 | 6 |
| 2 | Aktivace | Částečný | 3,6213 $ms$ | 1 | 3 | 3 | 8 |
| 3 | Aktivace | Částečný | 4,0847 $ms$ | 1 | 3 | 4 | 10 |
| 4 | Aktivace | Částečný | 2,8680 $ms$ | 1 | 3 | 5 | 12 |
| 4 | Deaktivace | Úplný | 2,0526 $\mu s$ | 0 | 0 | 4 | 10 |
| 3 | Deaktivace | Úplný | 2,0526 $\mu s$ | 0 | 0 | 3 | 8 |
| 2 | Deaktivace | Úplný | 1,5394 $\mu s$ | 0 | 0 | 2 | 6 |
| 1 | Deaktivace | N/A | 1,0263 $\mu s$ | 0 | 0 | 1 | 5 |
| 0 | Deaktivace | N/A | 513 $ns$ | 0 | 0 | 0 | 0 |

Tabulka 6.3: Výsledky měření pro případ 1.

*V tabulkách jsou použity tyto zkratky*:

- **SE** ... počet efektů spojených při provedení akce
- **SP** ... počet průchodů spojených při provedení akce
- **AE** ... počet aktivních efektů po provedení akce
- **AP** ... počet průchodů v aktivních efektech

V tomto případě byla využita cache o velikosti 5. Z tabulky 6.3 lze vyvodit, že ukládání kombinací redukuje počet skládaných efektů v budoucnu. Navíc je-li nalezena přesná shoda, klesá časová náročnost aktivace/deaktivace z milisekund na mikrosekundy. V případech, kdy je výsledkem akce jediný nebo žádný efekt, prohledávání cache se neprovádí.

**Případ 2) Aktivace/deaktivace efektů bez využití cache**

| Id efektu | Akce | Zásah v cache | Čas | SE | SP | AE | AP |
|---|---|---|---|---|---|---|---|
| 0 | Aktivace | N/A | 513 $ns$ | 0 | 0 | 1 | 5 |
| 1 | Aktivace | Výpadek | 6,9059 $ms$ | 1 | 4 | 2 | 6 |
| 2 | Aktivace | Výpadek | 9,6913 $ms$ | 2 | 7 | 3 | 8 |
| 3 | Aktivace | Výpadek | 13,9006 $ms$ | 3 | 10 | 4 | 10 |
| 4 | Aktivace | Výpadek | 17,0411 $ms$ | 4 | 13 | 5 | 12 |
| 4 | Deaktivace | Výpadek | 15,0501 $ms$ | 3 | 10 | 4 | 10 |
| 3 | Deaktivace | Výpadek | 10,3989 $ms$ | 2 | 7 | 3 | 8 |
| 2 | Deaktivace | Výpadek | 5,2464 $ms$ | 1 | 4 | 2 | 6 |
| 1 | Deaktivace | N/A | 9,2367 $\mu s$ | 0 | 0 | 1 | 5 |
| 0 | Deaktivace | N/A | 513 $ns$ | 0 | 0 | 0 | 0 |

Tabulka 6.4: Výsledky měření pro případ 2.

Tabulka 6.4 jasně ukazuje závislost doby aktivace/deaktivace efektu na počtu skládaných vykreslovacích průchodů v kombinovaných efektech. Ze znalosti implementace lze největší časové spoždění přiřadit překladu shaderů, sestavení programu a vytváření framebufferu. Z obou předchozích tabulek byla určena **průměrná doba nutná pro složení dvou průchodů cca 1,4 ms**.

I bez využití cache tak lze spojit stovky průchodů za vteřinu, což by mělo být dostatečné kritérium pro přepínání efektů v reálném čase. Na druhou stranu bylo prokázáno, že **využití cache kombinací efektů má přínos pro redukci doby aktivace/deaktivace efektů**.

### Test3 – testování frameworku uživateli

Vzhledem k tomu, že framework slouží především jako prostředek pro podporu práce jiných lidí, bylo žádoucí získat názor potencionálních uživatelů. Knihovna byla poskytnuta skupině 8 programátorů, kteří byli vybaveni různými úrovněmi zkušenosti s vývojem grafických

aplikací. Tito uživatelé pak po vlastnoručně provedené analýze poskytli hodnocení práce. Největší důraz byl přikládán komentářům popisujícím klady a nedostatky řešení. Následuje přehled nejdůležitějších poznatků:

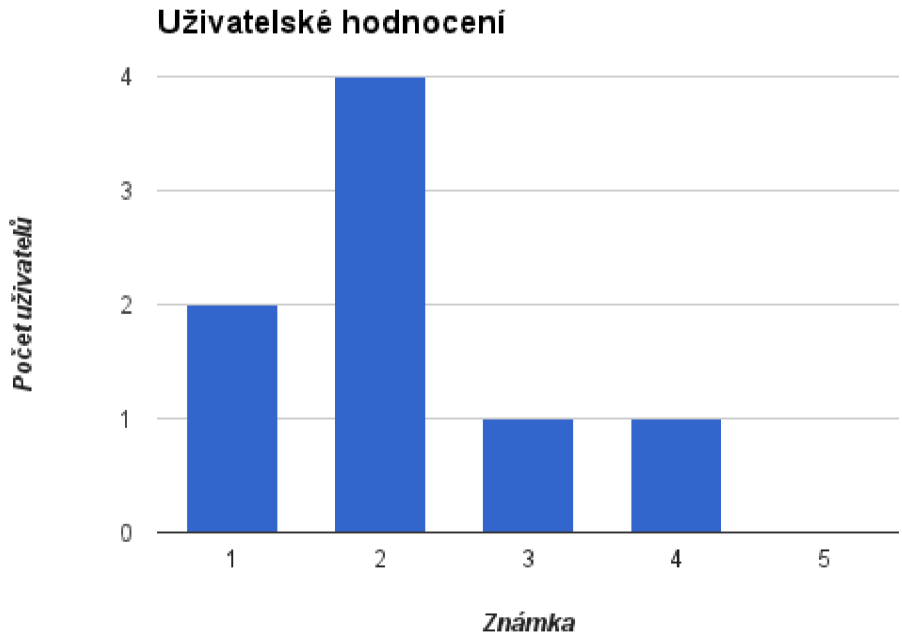
**Pozitiva:**

- Vytvoření efektů je snadné i pro uživatele s minimální znalostí knihovny OpenGL.
- Podpora více-průchodových algoritmů představuje jeden ze základů funkčnosti.
- Velké množství práce je provedeno automaticky (načtení scény, vykreslování atd.).
- Není potřeba dohledávat a zprovozňovat další knihovny.
- Framework představuje vhodný prostředek pro demonstraci scény.

**Negativa:**

- Implementace nezahrnuje celý rozsah funkčnosti knihovny OpenGL.
- Modifikovatelnost některých částí frameworku by zvýšila jeho využitelnost.
- Omezení verze OpenGL na 4.2 a vyšší je limitující na počet využitelných zařízení.
- Pochopení práce s frameworkem vyžaduje návod nebo alespoň studium dokumentace.

Dále měli uživatelé vyjádřit celkové hodnocení práce s frameworkem pomocí známek, od 1 (nejlepší) do 5 (nejhorší). Obrázek 6.2 obsahuje graf znázorňující souhrn výsledků.




Obrázek 6.2: Graf obsahující uživatelské hodnocení frameworku pomocí známek.

Lze tedy říct, že uživatelé byli se zpracováním frameworku ve větší míře spokojeni. Bylo získáno cenné hodnocení, které poslouží pro budoucí zkvalitnění nebo rozšíření implementace.

## 6.2 Možnosti dalšího vývoje

Výsledný framework odpovídá návrhu, je zde však množina funkcí, které do něj nebyly zahrnuty. Jedná se především o prvky knihovny OpenGL, např. 3D textury, cube mapy nebo accumulation buffer (viz [23]). Kód shaderů je generován pro vertex shader a fragment shader, ovšem některé efekty by mohly využít i jiné typy, jmenovitě geometry shader.

Ke zprovoznění frameworku na PC se staršími typy grafických karet, a případně i na jiných platformách, by bylo vhodné vytvořit fallback verzi zohledňující dostupnou verzi knihovny OpenGL. Rovněž přidání možnosti volby práce s knihovnou Direct3D by přineslo příležitost srovnat schopnosti obou API. Za účelem odstranění závislosti na externích knihovnách (GLEW, GLFW aj.) by bylo vhodné vytvořit vlastní implementaci poskytované funkčnosti.

U efektů se nabízí přesunout definici do externího souboru. Definici by mohl reprezentovat zápis v jazyku XML. Uživatel by pak mohl jednoduše měnit podobu definice efektu bez nutnosti znovu překládat aplikaci. Tento princip by přinesl zjednodušení tvorby, kombinace a šíření efektů.

# Kapitola 7

# Závěr

Tato práce se nejprve zabývá rozborem vizuálních efektů přítomných v dnešních grafických aplikacích. Zaměřuje se na vykreslování 3D scény metodou rasterizace. Popisuje teoretické základy a postup implementace vybrané sady efektů. Dále jsou zde nastíněny principy tvorby efektů pomocí knihovny OpenGL. Důraz je kladen na koncept vykreslovacích průchodů, využití shaderů a textur.

Jako hlavní část práce byl vytvořen softwarový framework (knihovna) sloužící k usnadnění tvorby vizuálních efektů. Implementace byla realizována v jazyku C++, s využitím objektově orientovaného přístupu. Pro účely vykreslování byla zvolena knihovna OpenGL verze 4.2. Množství úkonů je prováděno automaticky, jako např. generování kódu shaderů nebo načítání scény. Uživatel definuje efekty instanciací tříd frameworku a voláním příslušných metod. Podstatnou funkčnost představuje také skládání/kombinace 2 a více definic efektů dohromady. Popis dostupného API se nachází v programové dokumentaci. Kód je přenositelný na platformy Microsoft Windows a Linux.

Testováním bylo ověřeno splnění podmínky snížení náročnosti programování efektů. Definice obsahují menší množství kódu a nevyžadují pokročilou znalost knihovny OpenGL. Režie frameworku při vykreslování způsobuje zpoždění o cca 10 – 15%. Efekty je možné aktivovat a deaktivovat za běhu programu (při využití cache v reálném čase).

# Literatura


[1] Bavoil, L.; Sainz, M.: Image-space Horizon-based Ambient Occlusion. In *ShaderX7: Advanced Rendering Techniques*, editace W. Engel, 2009.

[2] Beets, K.; Barron, D.: Super-sampling Anti-aliasing Analyzed. *Beyond3D*, 2000.

[3] Blinn, J. F.: Models of Light Reflection for Computer Synthesized Pictures. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '77, ACM, 1977, s. 192–198.

[4] Bunnell, M.; Pellacini, F.: Shadow Map Antialiasing. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, editace R. Fernando, 2004.

[5] Cook, R. L.; Torrance, K. E.: A Reflectance Model for Computer Graphics. *Computer Graphics*, ročník 15, č. 3, 1981: s. 307–316.

[6] Demers, J.: Depth of Field: A Survey of Techniques. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, editace R. Fernando, 2004.

[7] Dimitrov, R.: Cascaded Shadow Maps. Technická zpráva, NVIDIA Corporation, August 2007.

[8] Donnelly, W.; Lauritzen, A.: Variance Shadow Maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ACM, 2006, s. 161–165.

[9] Eisemann, E.; Schwarz, M.; Assarsson, U.; aj.: *Real-Time Shadows*. A K Peters/CRC Press, 2011, ISBN 978-1-56881-438-4.

[10] James, G.; O'Rorke, J.: Real-Time Glow. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, editace R. Fernando, 2004.

[11] Josuttis, N. M.: *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 2th edition, 2012, ISBN 0321623215.

[12] Kajalin, V.: Screen-Space Ambient Occlusion. In *ShaderX7: Advanced Rendering Techniques*, editace W. Engel, 2009.

[13] Kessenich, J.; Baldwin, D.; Rost, R.: *The OpenGL Shading Language*. Khronos Group, August 2011.

[14] Laine, S.; Karras, T.: Two Methods for Fast Ray-cast Ambient Occlusion. In *Proceedings of the 21st Eurographics Conference on Rendering*, Eurographics Association, 2010, s. 1325–1333.

[15] Lottes, T.: FXAA. Technická zpráva, NVIDIA Corporation, February 2009.

[16] Martin, R. C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008, ISBN 0132350882.

[17] McGuire, M.: Efficient Shadow Volume Rendering. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, editace R. Fernando, 2004.

[18] Oren, M.; Nayar, S. K.: Generalization of Lambert's Reflectance Model. In *Proceedings of the 21th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, ACM, 1994, s. 239–246.

[19] Pharr, M.; Green, S.: Ambient Occlusion. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, editace R. Fernando, 2004.

[20] Phong, B. T.: Illumination for Computer Generated Pictures. *Communications of the ACM*, ročník 18, č. 6, 1975: s. 311–317.

[21] Reshetov, A.: Morphological Antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, 2009, s. 109–116.

[22] Segal, M.; Akeley, K.: *The OpenGL Graphics System: A Specification (Version 4.2 (Core Profile) - August 22, 2011)*. Khronos Group, August 2011.

[23] Shreiner, D.; Sellers, G.; Kessenich, J.; aj.: *OpenGL Programming Guide - The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley, 8th edition, 2013, ISBN 978-0-321-77303-6.

[24] Stamminger, M.; Drettakis, G.: Perspective Shadow Maps. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, ACM, 2002, s. 557–562.

[25] Stich, M.; Wächter, C.; Keller, A.: Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders. In *GPU Gems 3*, editace H. Nguyen, 2007.

[26] Stroustrup, B.: *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013, ISBN 0321563840.

[27] Thibieroz, N.: Deferred Shading with Multisampling Anti-Aliasing in DirectX 10. In *ShaderX7: Advanced Rendering Techniques*, editace W. Engel, 2009.

[28] Wirfs-Brock, R.; Wilkerson, B.; Wiener, L.: *Designing Object-Oriented Software*. Prentice Hall, 1990, ISBN 0136298257.

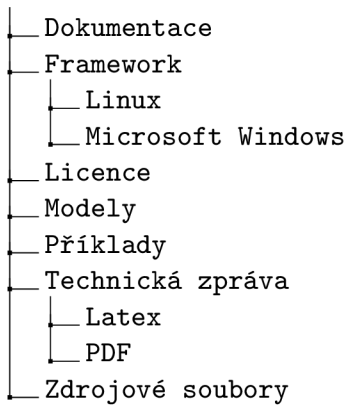
[29] Wolff, D.: *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing, 2011, ISBN 1849514763.

[30] Zhang, F.; Sun, H.; Nyman, O.: Parallel-Split Shadow Maps on Programmable GPUs. In *GPU Gems 3*, editace H. Nguyen, 2007.

# Příloha A

# Obsah DVD

**Adresářová struktura**:




**Obsah**:

- **Dokumentace** – programová dokumentace ve formátu HTML, viz *index.html*
- **Linux** – verze frameworku přeložitelná v OS Linux, obsahuje *Makefile*
- **Microsoft Windows** – verze frameworku přeložitelná ve Windows OS, obsahuje předkompilované soubory a *solution* pro Microsoft Visual Studio 2013
- **Licence** – licenční soubory frameworku a použitých knihoven
- **Modely** – 3D soubory pro vytvoření demonstrační scény
- **Příklady** – projekty pro rychlé otestování frameworku
- **Latex** – zdrojové soubory v LaTeXu použité pro generování tohoto dokumentu
- **PDF** – tento dokument
- **Zdrojové soubory** – zdrojové soubory frameworku napsané v jazyku C++

# Příloha B

# Instalace

Tato kapitola obsahuje instrukce pro zprovoznění frameworku na cílových platformách. Soubory se nacházejí na DVD v adresáři `Framework`.

### Microsoft Windows

*Potřebné soubory*:

- složka `visegl`
- `visegl.lib`
- `visegl.dll`, `Assimp32.dll`, `DevIL.dll`, `glew32.dll`, `glfw3.dll`

*Překladač*: Microsoft Visual C++ 12.0 (MSVS 2013) a vyšší

*Integrace frameworku do projektu*:

1. Vytvořte nový prázdný C++ projekt:

   *File* → *New* → *Project* → *Visual C++* → *Empty Project*

2. Otevřete dialogové okno pro úpravu nastavení projektu: *Project* → *Properties*
   (a) Nastavte cestu k hlavičkovým souborům frameworku (k adresáři `visegl`) do:

   *Configuration Properties* → *C/C++* → *Additional Include Directories*
   (b) Nastavte cestu k souboru `visegl.lib` do:

   *Configuration Properties* → *Linker* → *Additional Library Directories*
   (c) Přidejte soubor `visegl.lib` do seznamu závislých knihoven:

   *Configuration Properties* → *Linker* → *Input* → *Additional Dependencies*
   (d) Nastavte *Configuration Properties* → *Runtime Library* na *Multi-threaded DLL*
   (e) Nastavte *Configuration Properties* → *General* → *Platform Toolset* na *Visual Studio 2013 (v120)*
   (f) Potvrďte stisknutím tlačítka *OK*

Správnost instalace lze ověřit překladem minimální kostry aplikace:

```
#include <iostream>
#include <exception>

#include <visegl/context.h>
#include <visegl/effect_renderer.h>

using namespace visegl;


int main()
{
  try
  {
    Context context(800, 600, "Hello world!");

    auto renderer = make_shared<EffectRenderer>();
    context.SetRenderer(renderer);

    context.MainLoop();
  }
  catch(exception & exc)
  {
    cerr << exc.what() << endl;
    return 1;
  }
}
```

Zdrojový kód B.1: Kód pro ověření korektnosti instalace frameworku.

*Poznámky:*

- Před spuštěním aplikace nakopírujte soubory `visegl.dll`, `Assimp32.dll`, `DevIL.dll`, `glew32.dll` a `glfw3.dll` do adresáře, který obsahuje spustitelný soubor.
- Ujistěte se, že aplikace běží na grafické kartě podporující OpenGL verze 4.2.

### Linux

*Potřebné soubory*:

- `Makefile`
- složka `src`

*Překladač*: GCC verze 4.7 a vyšší

*Instalace*:

1. Nainstalujte nebo ověřte přítomnost následujících balíčků: `glm-devel`, `glew-devel`, `glfw-devel`, `assimp-devel` a `DevIL-devel`
2. Umístěte všechny potřebné soubory (viz výše) do zvoleného adresáře
3. Otevřete terminál, nastavte pracovní adresář podle bodu 2 a vyvolejte překlad zadáním příkazu `make`

4. Zadáním příkazu `make install` nakopírujte hlavičkové soubory a sdílenou knihovnu do systémových složek

*Poznámky*:

- Cílový adresář pro umístění instalovaných souborů lze změnit přepsáním následujících hodnot v souboru `Makefile`:
  - `HEADER_INSTALL_PATH` ... hlavičkové soubory
  - `LIB_INSTALL_PATH` ... sdílená knihovna
- Pro sestavení programu s využitím frameworku přidejte argumenty `-std=c++11 -lvisegl -lGLEW -lglfw` k příkazu `g++`
- Správnost instalace lze ověřit přeložením úseku kódu B.1
- Ujistěte se, že aplikace běží na grafické kartě podporující OpenGL verze 4.2.

# Příloha C

# Návod

Následující text slouží pro snadnější pochopení práce s frameworkem. Obsahuje kompletní kód programu implementujícího efekt *shadow mapping* (viz podkapitola 2.1) a myšlenkový postup, který vedl k jeho tvorbě.

Při programování aplikace v C++ je nejprve nutné uvést potřebné hlavičkové soubory. Třídy frameworku se nacházejí v souborech, jejichž název odpovídá názvu třídy, ovšem nahrazuje CamelCase za podtržítka (např. `StaticCamera` → `static_camera.h`). Implicitně se nacházejí v adresáři `visegl`.

```
// hlavičkové soubory STL
#include <iostream>
#include <exception>

// hlavičkové soubory frameworku
#include <visegl/context.h>
#include <visegl/effect_renderer.h>
#include <visegl/free_camera.h>
#include <visegl/static_camera.h>

// zjednodušení přístupu do jmenných prostorů
using namespace std;
using namespace glm;
using namespace visegl;
```

Je tedy čas přejít do funkce `main`. Veškerý kód by měl být vložen do bloku `try` pro zachycení případných výjimek.

```
int main(){
  try{
```

První důležitý krok představuje vytvoření okna a kontextu OpenGL. Pro tento účel využijeme třídu `Context`.

```
Context context(800, 600, "Shadow mapping", false);
auto window = context.GetWindow();
```

K zobrazení efektu budeme muset předat kontextu instanci třídy `EffectRenderer`.

```
auto renderer = make_shared<EffectRenderer>();
context.SetRenderer(renderer);
```

Pro nastavení pohledu a projekce slouží kamera. Třída `FreeCamera` nám poskytne možnost pohybovat se ve scéně. Nastavíme parametry kamery a perspektivní projekci. Velikost viewportu bude odvozena od předaného okna.

```
auto sceneCamera = make_shared<FreeCamera>(window);
sceneCamera->SetPosition(vec3(0.0f, 1.5f, 5.0f));
sceneCamera->SetRotationUnit(0.175);
sceneCamera->SetMovementUnit(2.0);
sceneCamera->SetPerspective(radians(60.0f), 0.01f, 100.0f);
renderer->SetSceneCamera(sceneCamera);
```

Následně načteme scénu pro demostraci efektu. K tomu nám poslouží správce scény (třída `SceneManager`), jež získáme přímo z rendereru. Pomocí linearních transformací umístíme objekty na vhodné místo ve scéně.

```
auto sceneManager = renderer->GetSceneManager();

auto bunny = sceneManager->LoadModel("bunny.obj");
bunny->Scale(vec3(2.0f));
bunny->Translate(0.0f, 0.95f, 0.0f);

auto ground = sceneManager->LoadModel("ground.obj");
ground->Scale(10.0f, 1.0f, 10.0f);
```

Osvětlení a především stíny tvoří základ tohoto efektu. Potřebujeme tedy umístit do scény zdroj světla. Pro jednoduchost zvolíme světelný zdroj typu *spotlight*.

```
auto light = make_shared<Light>(VISEGL_LIGHT_SPOTLIGHT);
light->SetPosition(10.0f, 10.0f, 10.0f);
light->SetSpotlightCutoff(radians(30.0f));
sceneManager->AddLight(light);
```

Nyní již máme dostatečný základ pro zahájení tvorby samotného efektu. Předem si zvolíme velikost stínové mapy.

```
auto effect = make_shared<Effect>("shadow mapping");
const unsigned SHADOW_MAP_SIZE = 1024; // velikost stínové mapy
```

V prvním průchodu budeme vykreslovat z pohledu světla, proto si pro tento účel vytvoříme kameru. Postačí nám třída `StaticCamera`, nepotřebujeme se zde pohybovat po scéně. Nastavíme viewport podle rozměrů stínové mapy a ostatní parametry získáme ze světelného zdroje. Ořezové roviny nastavíme tak, abychom pokryli celou scénu (zde 1 – 31).

```
auto lightCamera = make_shared<StaticCamera>();
lightCamera->SetViewport(SHADOW_MAP_SIZE, SHADOW_MAP_SIZE);
lightCamera->SetPositionAndTarget(light->GetPosition(), light->GetDirection());
lightCamera->SetPerspective(
  2.0f * light->GetSpotlightCutoff(), 1.0f, 1.0f, 31.0f);
```

Nyní vytvoříme první průchod (3D) a předáme mu nově vytvořenou kameru a správce scény. Nastavíme zahazování předních stran polygonů, do stínové mapy se uloží pouze hloubky zadních stran (redukujeme tak možnost chybných porovnání později).

```
auto pass1 = make_shared<RenderingPass3D>(lightCamera, sceneManager);
pass1->SetCulling(GL_FRONT);
```

Jeden z hlavních prvků efektu představuje stínová mapa. Musíme vytvořit texturu s formátem vhodným k uložení hloubky (zde 32b) a zvolenou velikostí. Připojíme ji jako *depth buffer* k prvnímu průchodu.

```
auto shadowMap = make_shared<RenderTarget>("shadow_map");
shadowMap->CreateDepthStencilTexture(SHADOW_MAP_SIZE, SHADOW_MAP_SIZE, 32);
pass1->SetDepthStencilTexture(shadowMap);
```

Dále se přesuneme k druhému průchodu (opět 3D). Zde využijeme již kameru z pohledu pozorovatele. Pro výpočet stínování si vyžádáme přítomnost našeho světelného zdroje a materiálů. Stínovou mapu připojíme pro vzorkování ve fragment shaderu.

```
auto pass2 = make_shared<RenderingPass3D>(sceneCamera, sceneManager);
pass2->UseLights(1);
pass2->EnableMaterials(true);
pass2->AddInputTexture(shadowMap, VISEGL_SHADER_FRAGMENT);
```

Vzorkování stínové mapy a následné porovnání vyžaduje přítomnost pozice transformované z pohledu světla. Nejprve budeme muset provést samotnou transformaci ve vertex shaderu. K tomu účelu vytvoříme příslušnou matici pro transformaci pohledu a projekce a předáme ji jako uniformní proměnnou právě do vertex shaderu. Pro transformaci ze souřadného systému modelu do souřadného systému světa již máme k dispozici matici `model_matrix`.

```
mat4 lightViewProjectionMatrix = lightCamera->GetViewProjectionMatrix();
pass2->AddUniform<GL_FLOAT_MAT4>(
  "light_vp_matrix", VISEGL_SHADER_VERTEX, &lightViewProjectionMatrix);
```

Následně interpolujeme pozici s definovanými transformačními funkcemi. Využijeme uniformní proměnnou k transformaci ve vertex shaderu. Ve fragment shaderu převedeme pozici zpět na homogenní.

```
pass2->AddInterpolated("lposition", GL_FLOAT_VEC4,
  R"(
vec4 tout_lposition()
{
  return light_vp_matrix * model_matrix * vec4(position, 1.0);
})",
  R"(
vec4 tin_lposition(vec4 lp)
{
  lp.xyz /= lp.w;
  lp.xyz = 0.5 * lp.xyz + 0.5;
  return lp;
})");
```

Pro účely stínování budeme potřebovat interpolovat také pozici, normálu a texturovací souřadnice (pro textury v materiálech).

```
pass2->AddInterpolatedGeometry(VISEGL_VERTEX_POSITION);
pass2->AddInterpolatedGeometry(VISEGL_VERTEX_NORMAL);
pass2->AddInterpolatedGeometry(VISEGL_VERTEX_TEXCOORD);
```

Definujme nyní funkci, která provede výpočet stínu porovnáním interpolované pozice v souřadném systému světla a hodnoty ze stínové mapy. Je-li fragment ve stínu, funkce vrátí hodnotu 0, jinak vrátí 1. Nesmíme zapomenout, že definice funkce vyžaduje její název, prototyp i tělo.

```
pass2->AddShaderFunction(
  "shadow",
  "float shadow()",
  R"(
float shadow()
{
  return (texture(shadow_map, lposition.xy).r < lposition.z)? 0.0 : 1.0;
})",
  VISEGL_SHADER_FRAGMENT);
```

Co se týče stínování, spokojíme se prozatím s ambientní a difúzní složkou světla. Podstatné je využití stínu v osvětlovací rovnici.

```
pass2->AddShaderFunction(
  "shading",
  "vec3 shading()",
  R"(
vec3 shading()
{
  vec3 L = normalize(lights[0].position - position);
  float dp = max(0.0, dot(L,normal));
  vec3 color = get_diffuse();

  return (0.1 * color) + (shadow() * dp * lights[0].intensity * color);
})",
  VISEGL_SHADER_FRAGMENT);
```

Pro uložení výsledku vytvoříme texturu o rozměrech okna a připojíme ji na výstup druhého průchodu. Po nastavení její sémantiky na `"surface"` je zajištěno kopírování obrazu do defaultního framebufferu.

```
auto colorBuffer = make_shared<RenderTarget>("color_buffer");
colorBuffer->SetSemantics("surface");
colorBuffer->CreateRenderTexture(
  window->GetWidth(), window->GetHeight(), GL_RGBA);
pass2->AddOutputTexture(colorBuffer, "color_buffer = vec4(shading(), 1.0);");
```

Oba průchody jsou nyní hotové, proto je přidáme k efektu a ten předáme rendereru pro vykreslení. Překlad proběhne automaticky, není třeba jej vyvolávat explicitně.

```
effect->AddRenderingPass(pass1);
effect->AddRenderingPass(pass2);
renderer->SetActiveEffect(effect);
```

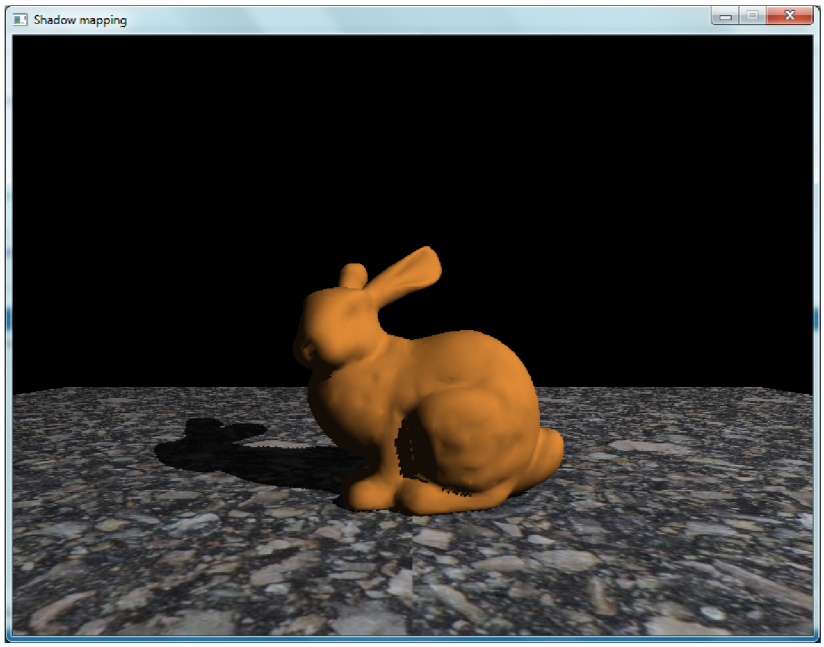
Nyní spustíme aplikační smyčku, která bude zajišťovat vykreslování a aktualizace.

```
context.MainLoop();
```

Jako poslední krok uzavřeme blok `try` a zapíšeme ošetření případné výjimky v bloku `catch`.

```
  } catch(exception & exc) {
    cerr << exc.what() << endl;
    return 1;
  }
}
```

Vytvořený program můžeme přeložit a spustit. Očekávaný výstup lze vidět na obrázku C.1. Definici efektu lze rozšířit mnoha způsoby, např. přidáním spekulární složky osvětlení, implementací filtru PCF ad.




Obrázek C.1: Očekávaný výsledek získaný implementací programu podle návodu.