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

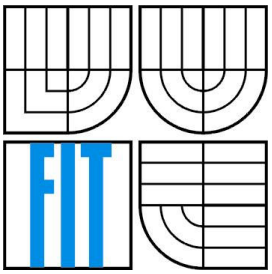
VERTEX / PIXEL SHADERY V REALTIME RENDERINGU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JIŘÍ LIGMAJER
AUTHOR

BRNO 2009

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

VERTEX / PIXEL SHADERY V REALTIME RENDERINGU

VERTEX / PIXEL SHADERS IN REALTIME RENDERING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Jiří Ligmajer
AUTHOR

VEDOUCÍ PRÁCE — Ing. Miroslav Švub
SUPERVISOR

BRNO 2009

Abstrakt

Moje bakalářská práce se zabývá vertex a pixel shadery. Jejich vznikem, jak fungují a jakým programovacím jazykem se dají implementovat. Součástí této práce je i tutoriál, který popisuje teorii a implementaci tří mnou zvolených realtime zobrazovacích technik. Na závěr jsou zhodnoceny dosažené výsledky u každé techniky.

Abstract

My bachelor's thesis is about vertex and pixel shaders. It is engaged in evolution, funcionality and usage of shaders in realtime graphics. This bachelor's thesis also includes tutorials, that are describing theory and implementation of three realtime rendering techniques, which were chosen by me. Results of my realtime techniques are evaluated at the end of my book.

Klíčová slova

Realtime, vertex, fragment, pixel, shader, motion blur, self-shadowing, Phong shading, transformace, OpenGL, GLSL, fixed, processing, pipeline, stínová mapa, depth mapa.

Keywords

Realtime, vertex, fragment, pixel, shader, motion blur, self-shadowing, Phong shading, transformations, OpenGL, GLSL, fixed, processing, pipeline, shadow map, depth map.

Citace

Ligmajer Jiří: Vertex / pixel shadery v realtime renderingu, bakalářská práce, Brno, FIT VUT v Brně, 2009

Vertex / pixel shadery v realtime renderingu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Miroslava Švuba
Další informace mi poskytli Ing. Miroslav Švub
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jiří Ligmajer
12.5. 2009

Poděkování

Chtěl bych poděkovat mému vedoucímu bakalářské práce Ing. Miroslavu Švubovi za poskytnuté informace, a odborné vedení při zpracovávání mé práce.

Obsah

1 Úvod

Grafické karty jsou nedílnou součástí hardware počítače. Dokážou totiž zobrazit obraz našich grafických rozhraní a aplikací na monitoru. Tak jako ostatní prvky počítače se i grafická karta musela vyvíjet do podoby, jak ji známe dnes. Dříve byl grafický hardware velice pomalý a zobrazování tak kvalitního obrazu, jak ho vídáme dnes, bylo prakticky nemožné, protože by trvalo hodiny než by se zobrazil jeden snímek. Ovšem s rozvojem tzv. programovatelných aplikačních prostředí jako je OpenGL nebo později DirectX se začal i grafický hardware více zlepšovat. Začalo být možné zobrazovat grafiku v reálném čase, bylo ale těžké naprogramovat lepší zobrazovací techniky. Toto se změnilo až s příchodem grafických karet, které umožňovali naprogramovat vlastní pokročilejší zobrazovací techniky pro vertex a fragment procesory v grafické kartě. Takovéto programy se nazývají shadery. Touto možností se výstupní obraz grafických aplikací stal o třídu lepší a vylepšuje se do dnes.

A právě vertex a fragment shadery jsou tématem mé bakalářské práce. V jednotlivých kapitolách se budu zabývat, jak shadery fungují, k čemu se používají a poté i jejich využitím v real-time grafice. V praktické části si vyberu tři zobrazovací techniky, které se používají v real-time grafice a ty naimplementuju. Pro každou techniku vytvořím tutoriál, který bude popisovat, jak se dají implementovat pomocí vertex a fragment shaderů a jejich následné použití v OpenGL aplikaci.

2 Princip shaderů

V následujících kapitolách se budu zabývat principem vertex a pixel (fragment) shaderů. Uvedu, jak shadery vznikly, kde se používají a k čemu jsou dobré. Zaměřím se také na jedno z grafických API (application programming interface) OpenGL a jeho rozšíření v podobě jazyka GLSL, který umožňuje vytváření programů pro vertex a pixel shadery.

2.1 Vznik OpenGL

Jazyk OpenGL je průmyslový standard přenosného API na jíné operační systémy. Vytvořeno bylo v roce 1992. Využíváno bylo v průmyslovém software, jako jsou CAD systémy. Časem však muselo být upraveno kvůli stále více prosazujícímu se konkurenčnímu rozhraní zvanému DirecX od Microsoftu.

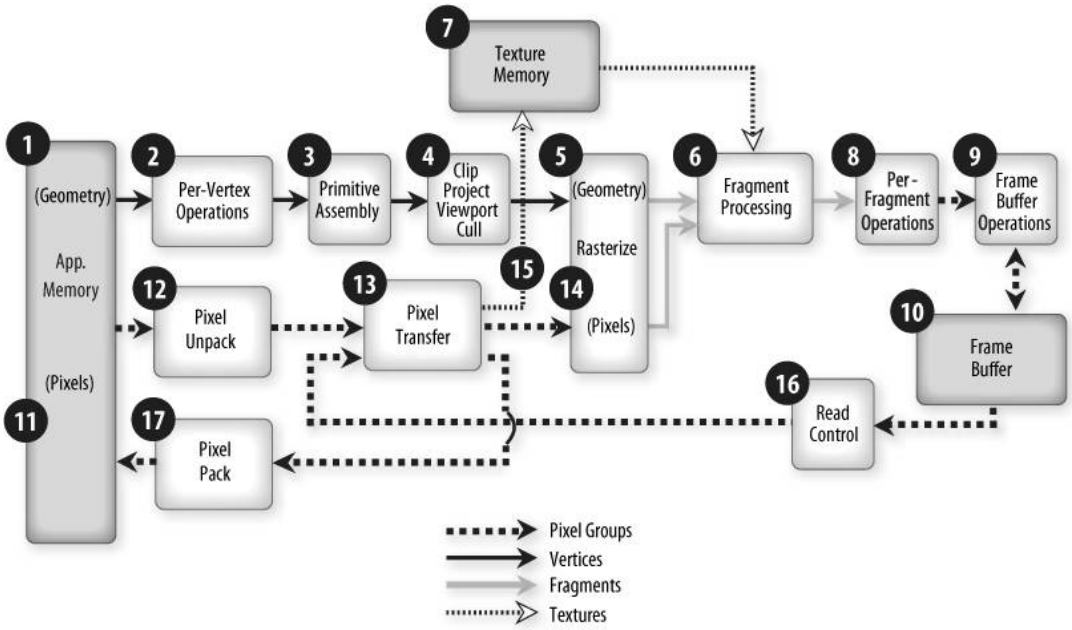
Vývoj OpenGL je řízen skupinou firem, které se říká OpenGL Architecture Review Board. Odtud pochází také zkratka ARB, která se vyskytuje u všech doplňkových funkcí. Současnými členy skupiny ARB jsou 3Dlabs, Apple, Dell, IBM, Intel, NVIDIA, ATI, SGI a Sun Microsystems.

Záměrem OpenGL je poskytnout přístup k možnostem grafického hardware na co nejnižší programovatelné úrovni, která je stále oddělená od samotné znalosti grafického hardware. OpenGL je implementováno pro několik různých operačních systémů zahrnující Mac OS, MS Windows a UNIX systémy. První verze OpenGL 1.0 se objevila v roce 1993 a do roku 2003 bylo vydáno dalších 5 specifikací. Všechny tyto specifikace OpenGL byly založeny na fixed-function pipeline (bude probírána dále). V roce 2004 přineslo OpenGL 2.0 možnost naprogramovat vertex a fragment procesory vlastním zobrazovacím algoritmem. Uveden byl proto také jazyk GLSL (OpenGL shading langure), který umožnil psát programy pro vertex a pixel shadery. Další verze OpenGL 2.1, přinesla rozšíření specifikace jazyka GLSL na verzi 1.20, kterou používám ve své práci pro implementaci shaderů. Současná verze OpenGL 3.1 je vylepšením OpenGL 2.0 a je konkurencí k Microsoft DirectX 10.0. Vyžaduje totiž grafické karty s podporou DX10 nebo vyšší.

2.2 Fixed pipeline

V této sekci se zaměřím na popsání principu fixed pipeline. Takto se nazývá stav zpracovávání a zobrazování scény, kdy nebylo možné programovat vertex a fragment procesory, tj. jejich funkce byla dána výchozím způsobem, tak jak byla definována v OpenGL, než přišla specifikace 2.0. Schéma fixed pipeline je zobrazeno na obrázku 1. Rozeberu hlavně části, které jsou v obrázku označeny čísly 1, 2, 3, 4, 5, 6, 7, 8, 9 a 10. Tyto části se zabývají zpracováním a transformacemi vertexu a jejich převodem na fragmenty. Toto je dobré znát před tím, než se pustím do vysvětlování principu vertex a pixel shaderů.

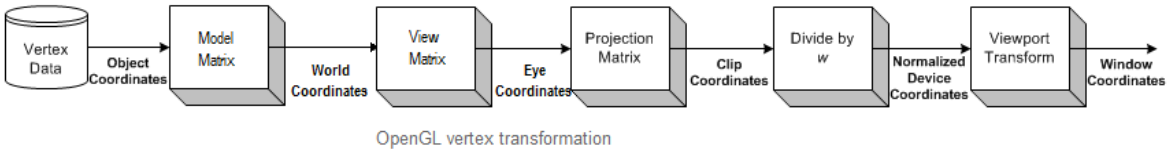
Nejprve bych popsal jednotlivé fáze transformování vertexů do jednotlivých oblastí, při zpracovávání geometrie objektů.



Obrázek 1. Schéma OpenGL fixed pipeline

2.2.1 Transformace vertexů

Každý vertex, než je zobrazen v okně OpenGL aplikace, prochází několika transformacemi, tak abychom dostali jeho výslednou polohu ve scéně. Tyto transformace jsou uvedeny na obrázku 2.



Obrázek 2. Transformace vertexů v OpenGL

Před zpracováním jsou jednotlivé vertexy resp. normály, v OpenGL aplikaci definovány pomocí funkce **glVertex()** resp. **glNormal()**, uloženy v paměti aplikace (viz (1) na obrázku č. 1) a postupně posílány do grafické karty. Definované souřadnice vertexu a normál v aplikaci jsou v objektovém prostoru.

V grafické kartě jsou na každý vertex aplikovány modelové transformace (viz (2) na obrázku č. 1) např.: rotace, posunutí, zvětšení, zmenšení, atd, přičemž si musíme uvědomit, že OpenGL zpracovává transformace od poslední k první. Po aplikování modelových transformací na vertex dostáváme souřadnice vertexu v tzv. světovém prostoru (world space).

$$v_{world} = \mathbf{M}_{\mathbf{model}} \cdot v_{obj}$$

Následně jsou na takto transformovaný vertex aplikovány pohledové transformace, což znamená posunutí vertexu ve scéně vůči kameře, nebo oku pozorovatele. Výsledkem je pozice vertexu z pohledu kamery resp. pozorovatele (eye space).

$$v_{eye} = \mathbf{M}_{\mathbf{view}} \cdot v_{world}$$

Další transformací, která navazuje na pohledovou transformaci je projekční transformace. Ta má za úkol rozhodnout, zda bude vertex viditelný ve scéně nebo nikoliv. Každý prostor, který je ve scéně vidět, je určen pomocí frustum, což je ohraničená oblast, která je vidět z okna OpenGL aplikace. Získané souřadnice jsou v tzv. clip space.

$$v_{clip} = \mathbf{M}_{\mathbf{projection}} \cdot v_{eye}$$

Následně je tyto souřadnice třeba normalizovat. Dostáváme souřadnice vertexu v okně, které stále neurčují pozici pixelu (normalized device coordinates). O získání souřadnic pixelu v okně se stará funkce **glViewport()**, která upraví NDC souřadnice vertexu, tak aby byly v rámci zobrazovaného okna.

Jednotkou fixed pipeline, která se stará o transformace vertexů je vertex procesor. Dokáže i další věci, jako je počítání osvětlení v rámci vertexu (per vertex-lighting) nebo generování a transformování texturovacích souřadnic. Po provedení všech transformací se dostáváme k dalšímu zpracovávání grafických dat. Tím je část primitive assembly.

2.2.2 Primitive assembly

Po zpracování vertexů vertex procesorem, přichází na řadu složení vertexů do jednotlivých primitiv (viz (3) na obrázku č. 1). Primitiva jsou základní geometrické struktury, jako jsou body, čáry, trojúhelníky, čtverce a mnohoúhelníky. Každý bod je tvořen jedním vertexem, čára dvěma,

trojúhelník třemi, čtverec čtyřmi a mnohoúhelník definovaným počtem vertexů. Funkcí **glBegin()** se pozná, jaké primitivum je třeba složit.

2.2.3 Zpracování primitiv

Následující pasáží je zpracování grafických primitiv (viz (4) na obrázku č. 1). Zahrnuje dvě důležité funkce. První z nich je učení, zda grafické primitivum leží vně, uvnitř nebo je rozděleno, uživatelsky definovaným frustumem.

Frustum je v OpenGL definováno pomocí modelview-projection matice, která je složena konkatenací modelview a projection matic. Každé primitivum je potom porovnáváno z frutumem a uživatelsky definovanými ořezávacími hranicemi. Pokud všechny tyto testy projdou, je primitivum připraveno k další fázi zpracování. Pokud jedna z těchto částí neprojde, primitivum nebude ve výsledné scéně vidět a je zahozeno. Při rozdělení primitiva na viditelnou a neviditelnou část, je třeba ho oříznout a k dalšímu zpracování poslat jen tu část, která bude vidět.

Po fázi ořezání přichází na řadu pasáž, kde jsou jednotlivé vertexy primitiv převedeny do souřadnic okna OpenGL aplikace. Toto je provedeno pomocí funkce **glViewport** a **glDepthrange**. Každý vertex musí být tedy nejprve normalizován, abychom dostali souřadnice v rozmezí [-1, 1], ve všech třech osách x, y, z. Tyto NDC souřadnice jsou poté transformovány pomocí již dvou zmíněných funkcí, aby padly do rozhraní okna aplikace. Nyní jsme získali souřadnice vertexů v okně, v tuto chvíli by byli vidět pouze jednotlivé body, ze kterých je primitivum sestaveno. Musíme ho ještě rasterizovat.

2.2.4 Rasterizace

Proces rasterizace (viz (5) na obrázku č. 1) bere jednotlivá grafické primitiva, které prošly fází Primitive processing. Každá skupina vertexů, která tvoří grafické primitivum, je rozložena do menších základních částí, které se nazývají fragmenty. Fragmenty jsou jednotlivé pixely v cílovém framebufferu a jsou vypočítány pomocí rasterizačních algoritmů (přímka, kružnice, elipsa). Každý fragment má stejně, jako vertex definované atributy. Těmi hlavními jsou souřadnice v framebufferu, hloubka, která je uložena v depth bufferu, dále to mohou být interpolované barvy mezi jednotlivými vertexy, texturovací souřadnice a další. Zda má být barva pixelů interpolována nebo přiřazena defaultně podle vertexů, je definováno OpenGL funkcí **glShadeModel** a konstantami GL_SMOOTH resp. GL_FLAT.

Poslední částí je samotné zpracování fragmentů, operace s nimi a jejich zápis do framebufferu.

2.2.5 Zpracování fragmentů

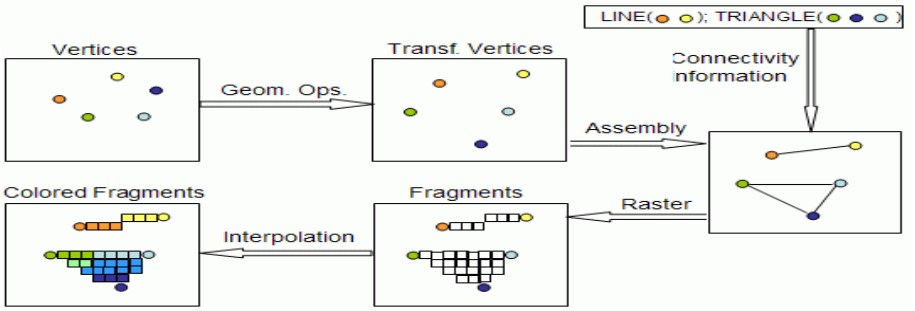
Poslední část zpracování geometrických primitiv jsem nazval zpracování fragmentů a uvedu zde operace, které se provádějí se všemi fragmenty naráz a potem per-fragment. Dále se zmíním o zápisu do framebufferu (viz (6, 7, 8, 9, 10) na obrázku č. 1).

Po rasterizaci grafických primitiv získáme skupinu fragmentů, které tvoří zpracované grafické primitivum. Nejdůležitější proces, který se zde vykonává je mapování textury na jednotlivé fragmenty, přiřazení barvy fragmentům nebo modifikování barvy podle vzdálenosti fragmentu od okna (viewport). Dále je to také součet primární a sekundární barvy fragmentu.

Další zpracování probíhá jako per-fragment, pro každý fragment zvlášť. Nejdůležitějšími jsou alpha test, stencil test a depth test. Alpha test má za úkol zjistit podle alpha složky barvy fragmentu a definované funkce pomocí glAlphaFunc, zda se má fragment zahodit nebo nikoliv. Stencil test porovnává hodnotu fragmentu s hodnotou v stencil bufferu a rozhoduje, zda se má fragment vykreslit, nebo zakryt jiným grafickým primitivem. Poslední operací je depth test, který porovnává z-ovou souřadnici fragmentu s hodnotou uloženou v depth bufferu a rozhodne, zda je vidět, nebo leží za jiným tělesem.

Dokončením fáze zpracování fragmentů se dostáváme k operacím nad framebufferem. Framebuffer je prostor definovaný výškou a šířkou okna, uložený v paměti aplikace. Zapisují se do něj výsledné fragmenty podle jejich definované pozice. Lze zde provádět aktualizaci, vymazání určitých atributu fragmentu, kterými jsou hloubka nebo stencil hodnota. Accumulation buffer zase zpracovává dohromady jednotlivé zobrazení framebufferu a poté výsledek nahraje do framebufferu. Takto se dá vytvořit například motion blur v OpenGL. Výsledný framebuffer je poté zobrazen do okna OpenGL aplikace.

V kapitole 2.2 byl popsán princip OpenGL fixed pipeline, která se v OpenGL používala od verze 1.0 do 1.5. Uvedl jsem, jak se transformují vertexy, tak jak prochází zpracováním na grafické kartě a jak se z nich stanou fragmenty, které tvoří výsledný obraz scény v OpenGL aplikaci. Obrázek 3. graficky znázorňuje, jak jsou jednotlivé vertexy zpracovávány. Toto je základ, který je třeba znát, abychom porozuměli, jak pracuje tzv. processing pipeline, která přišla ve verzi OpenGL 2.0 a znamenala velký skok kupředu. Začalo být totiž možné programovat vertex a fragment procesory vlastními algoritmy pomocí nového jazyka GLSL.

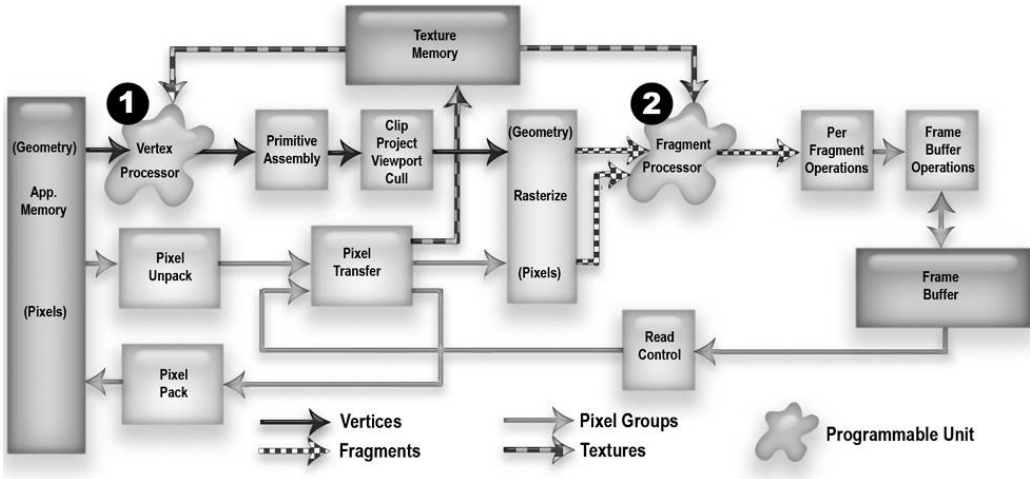


Obrázek 3. Zpracování vertexů v pipeline

2.3 OpenGL processing pipeline

Uvedení programovatelných procesorů, je největší změnou od vzniku OpenGL. S tím souvisí i potřeba High Level shading jazyka, který se v rámci OpenGL nazývá GLSL (OpenGL Shading Language). V předchozí kapitole 2.2 jsem se zabýval fixed pipeline, která vykonává operace, které jsou přesně dané. V processing pipeline je umozněno do vertex a fragment procesorů, nahrát vlastní algoritmus, který je napsán v jazyku GLSL. Používání fixní funkcionality vertex a fragment procesorů je zakázáno pouze tehdy, když je do těchto procesorů nahrán program shaderu.

Obrázek 4. ukazuje stav, kdy jsou naprogramovány vertex (1) a fragment procesory (2). Ostatní části processing pipeline zůstávají fixní.



Obrázek 4. Schéma processing pipeline

Vidíme, že data putují z paměti aplikace do vertex procesoru, poté do fragment procesoru a konečně jsou zapsána do framebufferu, který je zobrazen do okna aplikace. OpenGL je navrhnuto tak, aby poskytovalo paralelní zpracování vertexů a fragmentů, což umožňuje výrobcům grafických karet vytvářet stále rychlejší hardware.

Chtěl bych také zmínit, že dnes se v grafických kartách vyskytuje tzv. unifikovaná architektura, kde vedle vertex a fragment procesorů existují ještě geometry procesory. Geometry shader je novinkou v konkurenčním rozhraní DirecX 10, ale jsou podporovány i v OpenGL 3.0 a vyšším. Tyto procesory umožňují grafické kartě vytvářet další vertexy, což znamená, že můžeme změnit úplně geometrii zobrazovaného objektu, nebo vytvářet další geometrické struktury. Geometrické procesory se potom v návrhu grafické karty nacházejí mezi vertex a fragment procesory. Těmito částmi se ale zabývat nebudu, protože to není předmětem mé práce.

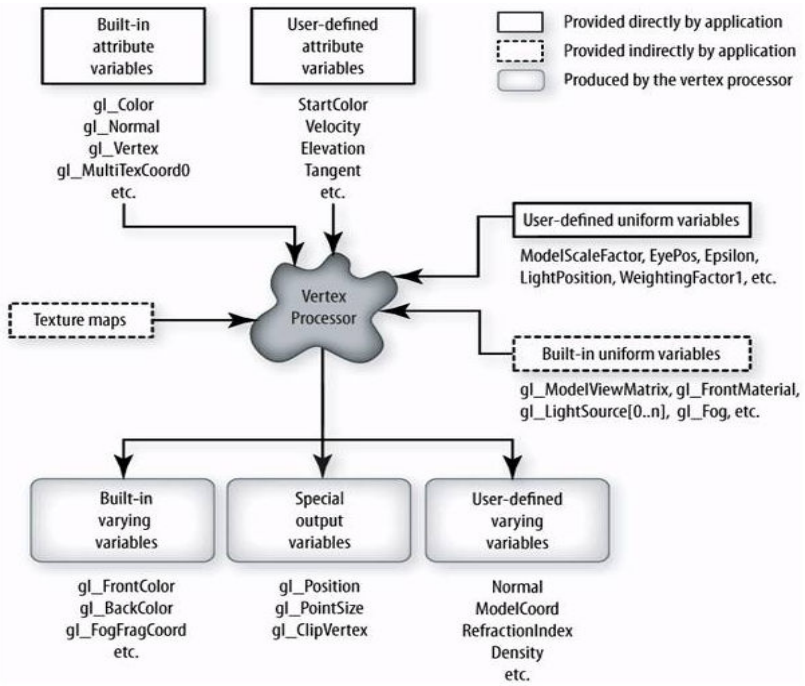
2.3.1 Vertex procesory

Vertex procesor provádí operace nad příchozími vertexy a jejich atributy nebo daty. Nezná žádné souvislosti mezi dalšími vertexy, proto nemůže provádět jiné operace než per-vertex. K tomu slouží další fáze zpracování. Vertex procesor může provádět následující úkoly:

- Transformace vertexů
- Transformace normál a jejich normalizace
- Generování a transformaci texturovacích souřadnic
- Výpočet osvětlení per-vertex
- Aplikaci barvy materiálu na daný vertex

Programy, které je možné nahrát do vertex procesoru, se nazývají vertex shadery. Tento shader implementuje uživatelsky napsaný algoritmus, který říká, jaké operace se mají s každým vertexem provést. Nemůžeme ovšem chtít, aby náš vertex shader transformoval vertex a vygeneroval texturovací souřadnice, načež následně fixní funkcionalita vertex procesoru spočítala osvětlení. Toto nelze, z toho důvodu, že pokud je do procesoru nahrán nějaký shader, fixní funkcionalita automaticky odpadá. Je tedy nutnost mít všechny úkony definované v našem vertex shaderu.

Vertex procesor, tedy vezme vstupní data, provede s nimi požadované operace a uloží je do výstupních proměnných. Schéma Vertex procesoru a jeho registrů je na obrázku 5.



Obrázek 5. Schéma vertex procesoru

Proměnné, které se vyskytují ve vertex shaderu, můžou být rozděleny, jako attribute proměnné. Tyto proměnné definují atributy každého vertexu a jsou děleny do dvou skupin, vestavěné a uživatelsky definované. Dají se definovat mezi voláním funkcí **glBegin** a **glEnd** nebo voláním vertexového pole, protože jsou často používány, mění se jejich obsah s každým vertexem. Vestavěné attribute proměnné definují barvu vertexu (*gl_Color*), pozici vertexu (*gl_Vertex*), normály (*gl_Normal*), texturovací proměnné (*glMultiTexCoord*), a další. Kromě vestavěných proměnných může uživatel definovat další atributy vertexu, které jsou Vertex shaderu předány voláním funkce **glVertexAttrib** z OpenGL aplikace. Nejprve je třeba si zjistit jejich umístění pomocí **glBindAttribLocation**.

Další proměnné, pomocí kterých se dají definovat vstupní proměnné, jsou uniform proměnné. Uniform mohou posílat data z OpenGL aplikace jak do vertex tak i fragment shaderů. Ve vertex shaderu se vyskytují zabudované proměnné, které poskytují informace nepřímo. Nejčastěji to jsou modelview matice, projection matice, textury, atd. Všechny tyto proměnné začínají pomocí předpony „gl_“. Uniform proměnné nemění svoji hodnotu tak často, jako attribute, protože jsou definovány pro alespoň jedno grafické primitivum nebo celou scénu. Můžeme vytvářet i vlastní uniform proměnné. Použitím funkce **glGetUniformLocation** si zjistíme místo, kde se námi deklarovaná proměnná nachází. Poté zavoláním funkce **glUniform** můžeme do ní nahrát námi požadovaná data. Lze předávat datové typy float, double, int, boolean, vektory a matice. Vektory můžou mít maximálně 4 komponenty a matice má maximální rozměr 4x4.

Vertex procesory jsou konstruovány taky, aby zpracovávali jeden vertex v daném čase. Pokud má grafická karta více vertex procesorů, je možné paralelně zpracovávat větší množství vertexů. Primárně jsou zaměřeny na transformaci a výpočet osvětlení pro každý vertex. Vertex procesory ukládají výsledky do speciálních výstupních proměnných. Každý vertex musí mít definovánu výslednou transformovanou pozici v homogenních clip-space souřadnicích. Tuto hodnotu získáme konkatenací modelview-projection matice a zpracovávaného vertexu. Vypočtená hodnota je uložena do proměnné *gl_Position*. Další výstupní proměnné, které je možné defaultně nastavit jsou *gl_ClipVertex*, která definuje uživatelsky vytvořené frustum, a *gl_PointSize*, což je velikost fragmentu.

Proměnné definující data jsou předávány do fragment shaderu pomocí varying proměnných. Jejich název vyplývá z toho, že do fragment shaderu nepřijde stejná hodnota, kterou vypočítá vertex shader, ale hodnota interpolovaná pro daný fragment. Varying proměnné jsou vestavěné a uživatelsky definované. Vestavěné proměnné definují texturovací souřadnice, každé z osmi aktivních texturovacích jednotek. Dále interpolovanou barvu pro vertex pomocí (*gl_FrontColor* a *gl_BackColor*) atd. Uživatelsky definované varying proměnné, mohou předávat texturovací souřadnice pro shadow mapu nebo různé mapovací techniky (bump, displacement mapping), modelové souřadnice, normály a další.

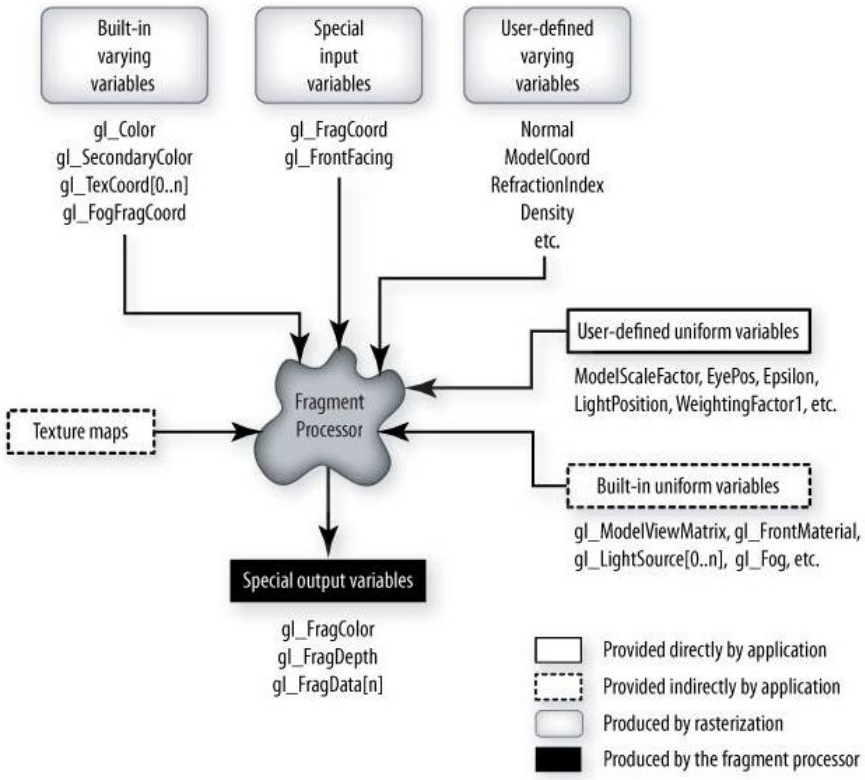
2.3.2 Fragment procesor

Fragment procesor je programovatelná jednotka, která zpracovává příchozí fragmenty. Fragmenty byly vytvořeny ve fázi rasterizace jednotlivých grafických primitiv, úseček, polygonů, trojúhelníků, atd. Prostřednictvím fragmentů se v rámci zpracování vykonávají následující techniky:

- Operace na interpolovaných souřadnicích
- Náhled do textury
- Aplikace textury na povrch těles
- Per-fragment osvětlení
- Nastavení barvy fragmentu

Programy, které provádí fragment procesor, se nazývají fragment shadery. Definují uživatelské zobrazovací algoritmy. Každý fragment je zpracován tímto algoritmem. Nelze ovšem, aby fragment procesor vykonal nějaké operace, například spočítal per-pixel osvětlení a poté fixní funkcionalita aplikovala texturu na objekt. U fragment procesoru platí stejné zásady jako u vertex procesoru. Fragment procesory jsou stavěny tak, aby zpracovávali jeden fragment v daný čas. V grafické kartě může být více fragment procesorů, které se starají o paralelní zpracovávání fragmentů, tj. v daný čas může být zpracováno více fragmentů.

Na obrázku 6. je vidět schéma fragment procesoru a jaké proměnné používá.



Obrázek 6. Schéma fragment procesoru

Primárním vstup shaderu jsou interpolované proměnné, které vzniknou jako následek rasterizačního procesu. Rozlišujeme vestavěné a uživatelsky definované proměnné. Uživatelsky definované proměnné musí mít stejný název a datový typ, jaký měli při výstupu z vertex shaderu.

Vstupní proměnné fragment shaderu, které byly vytvořeny během cesty mezi vertex a fragment procesorem se nazývají speciální vstupní proměnné. Zahrnují pozici fragmentu v okně (*gl_FragCoord*) a zda příchozí fragment náleží grafickému primitivu, které je tzv. front facing (*gl_FrontFace*).

Další uživatelské proměnné mohou být typu uniform, ke kterým se dá přistupovat jak z vertex tak i fragment shaderu. Zahrnují nejčastěji textury a další data, která jsou třeba předat z aplikace do fragment shaderu.

Největší výhodou fragment shaderu je, že může nahlížet do textury několikrát za sebou, může aplikovat i multi-texturing, což je aplikace dvou a více textur naráz. Dále mohou být hodnoty získané náhledem do jedné textury, vstupními souřadnicemi textury jiné. Náhledy nemají mezi sebou žádné společné nebo zděděné informace, proto může být fragment shader použit k implementaci ray-casting algoritmů, nebo ambient occlusion. Postup, jak se má shader chovat při přístupu do textury je popsáno v OpenGL aplikaci, parametry jako: wrapping, borders, filtering nebo compare mode (použití v shadow mapě).

Výstupní proměnné fragment shaderu jsou definovány speciálními proměnnými *gl_FragColor*, *gl_FragDepth*, *gl_FragData*. Každý fragment shader je zodpovědný alespoň za zapsání hodnoty do *gl_FragColor*, která nastavuje barvu danému fragmentu. Pokud není zapsána žádná výstupní hodnota fragment automaticky zahozen. Proměnné *gl_FragDepth* slouží k definici z-ové souřadnice fragmentu, která je poté uvedena v depth bufferu, pokud není zapsána, je definována pomocí fixní funkcionality podle OpenGL aplikace. Poslední výstupní proměnnou jsou *gl_FragData*, což jsou další libovolná doplňková data pro daný fragment.

Po zpracování fragmentu fragment procesorem jsou výstupní data zapsána právě do speciálních výstupních proměnných. Každý fragment je poté dále pracováván podle fixed pipeline. Tyto funkce jsou relativně složité, protože vyžadují mnoho čtení, zápisu a aktualizací hodnot, s kterými pracují (např.: blending, dithering, depth testing, stencil testing, alpha testing, atd). Tato funkcionalita se dá naprogramovat i v shaderu, ale doba zpracování by byla neúměrně dlouhá oproti standardním postupům. Všechny shadery (vertex i fragment) jsou totiž vytvářeny za účelem co nejrychlejšího zpracování grafických dat, ale také implementace nových zobrazovacích algoritmů, které nedokáže fixed pipeline vykonat.

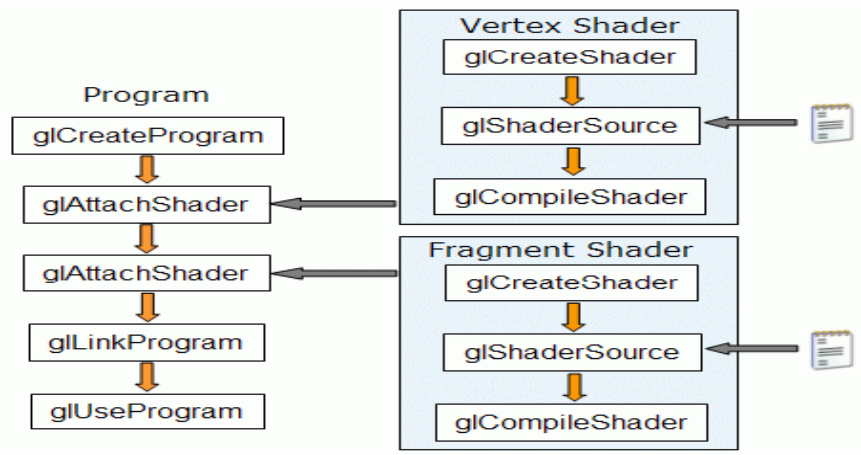
2.4 Použití shaderů v OpenGL

V této kapitole se budu zabývat nastavením OpenGL aplikace, abychom mohli použít námi naprogramované shadery. Jazyk, který slouží v OpenGL pro psaní vertex a fragment shaderů, se nazývá OpenGL Shading Language (GLSL).

Nejprve je třeba mít vytvořený nějaký vertex a fragment shader, který budeme používat pro naprogramování vertex a fragment procesorů. Shadery jsou implementovány pomocí jazyka GLSL, který je podobný jazyku C. Obsahuje specifické datové typy, funkce a proměnné, které jsou používány v shaderech.

Nastavení OpenGL aplikace se dá přirovnat k psaní programu v jazyku C. Každý shader má svůj algoritmus, jehož implementace je podobná jazyku C. Tak jako zdrojové kódy jazyka C musí být přeloženy překladačem a nalinkovány do výsledného binárního souboru, je i shader třeba vytvořit, přiřadit mu zdrojový kód, přeložit ho a nakonec nalinkovat do vytvořeného shader programu. Každý shader program se musí skládat z vertex a fragment shaderu. Lze použít jeden vytvořený vertex shader, který bude dodávat data více různým fragment shaderům nebo naopak, několik různých vertex shaderů dodává data jednomu fragment shaderu. Těchto možností lze využít při implementaci různých druhů osvětlení, kreslené grafiky a dalších.

OpenGL aplikace potřebuje pro používání shaderů ARB rozšíření, které přišlo od verze 2.0. Bohužel OpenGL 2.0 nemá implementovány funkce pro vytvoření, překlad a sestavení shaderů. Je tedy třeba si sehnat knihovnu GLEW (OpenGL Extension Wrangler library). Tato knihovna definuje všechny používané funkce pro správný chod shaderů s OpenGL aplikací. Pro sestavení shader programu je třeba si vedle knihoven OpenGL, vložit do aplikace i knihovnu GL/glew.h. Nyní je možné používat, pro vytvoření shader programu, OpenGL syntaxi nebo ARB syntaxi.
Následující obrázek 7. popisuje schéma vytvoření vertex a fragment shaderu, jejich spojení v programu, který je následně nahrán do grafické karty. Vidíme dále OpenGL syntaxi funkcí,



Obrázek 7. Postup vytvoření shader prog.

které jsou pro jednotlivé kroky použity.

Nejprve musíme mít k dispozici zdrojové kódy vertex a fragment shaderu psané v jazyku GLSL. Tyto kódy si přečteme ze souboru do nějakého textového řetězce. Musíme mít ovšem na paměti, že nesmíme použít binární mód pro čtení souboru, jinak nám shader nepozná, co jsme přečetli.

Následuje samotné vytvoření shader objektu. Každý vertex a fragment shader musí mít vytvořen svůj vlastní objekt. Toto se provede funkcí **glCreateShader** a jako parametr se zadá typ shaderu (GL_VERTEX_SHADER resp. GL_FRAGMENT_SHADER). Vertex shader může obsahovat několik objektových kontejnerů, z nichž každý má vlastní kód. Musí být pouze dodrženo, aby celý shader složený z více kontejnerů měl jednu hlavní funkci (main). Totéž platí pro fragment shader. Po vytvoření shader objektu, je třeba mu přiřadit nějaký zdrojový kód, který jsme si načetli ze souboru. Toto provedeme pomocí **glShaderSource**. Nakonec je třeba shader objekt zkompilovat (**glCompileShader**). Schéma vytvoření funkčního shader objektu (vertex, fragment) je ukázáno na obrázku 7. vpravo.

Vytvořením objektového kontejneru se zdrojovým kódem a jeho zkompilováním máme připraveny funkční vertex a fragment shadery. Nyní je musíme přiřadit do vytvořeného programového kontejneru, který je bude spouštět. Nejprve je třeba si vytvořit objekt, který se bude chovat jako program kontejner, docílíme tím funkcí **glCreateProgram**. Programů můžeme mít v OpenGL aplikaci několik, které se ve fázi zobrazení mohou vyměňovat nebo se může použít i fixní funkcionalita. Např. máme program, který mapuje stíny do scény a druhý program počítá osvětlení jednotlivých modelů ve scéně. Dále k vytvořenému kontejneru připojíme oba vertex a fragment shadery funkcí **glAttachShader**. Shaderů, které připojíme k programu, můžeme mít několik, stejně jako program jazyka C může mít několik modulů. Platí ale, že hlavní funkce shaderu může být pouze jedna v rámci všech modulů shaderu. Takto naplněný programový kontejner lze pomocí funkce **glLinkProgram** zkompilovat a vytvořit tím funkční program, který používá k zobrazování scény určené vertex a fragment shadery. Výsledný program se automaticky sám nezačne používat, ale musíme pomocí funkce **glUseProgram** sdělit OpenGL aplikaci, aby používala náš vytvořený program.

Pokaždé, když přestaneme shadery a jejich programy používat, měli bychom je odpojit od programového kontejneru a poté smazat. K této úloze slouží tři funkce. První funkce **glDetachShader** odpojí vertex nebo fragment shader od programového kontejneru. Pouze shadery, které nejsou připojeny k programům, můžou být smazány. Odstranění shaderu z programovatelných procesorů grafické karty se provede pomocí funkce **glDeleteShader**. Celý program se smaže funkcí **glDeleteProgram**.

Všechny tyto funkce se dají použít i v syntaxi ARB. Většina z nich má stejné jméno, jako v syntaxi OpenGL a stačí pouze přidat koncovku ARB. Pouze funkce, které vytváří programový kontejner nebo objektový kontejner pro shader, jsou zapsány funkcí **glCreateProgramObjectARB** resp. **glCreateShaderObjectARB**.

3 Teorie

Součástí mé bakalářské práce je i praktická část, kde jsem si vyzkoušel implementovat vertex a pixel shadery. Mým úkolem bylo vytvořit tutoriály k třem technikám, které se používají real-time grafice a dají se implementovat pomocí shaderů. Mnou vybrané techniky zahrnují:

- Phong shading
- Motion blur
- Self-shadowing

V této kapitole popíši a rozeberu teoretickou a matematickou část jednotlivých mnou vybraných technik. Uvedu, jak se dají tyto techniky používat v real-time grafice. Na konci mé práce porovnám zobrazované scény pomocí shaderů se scénami, které byly zobrazeny pomocí OpenGL fixed pipeline.

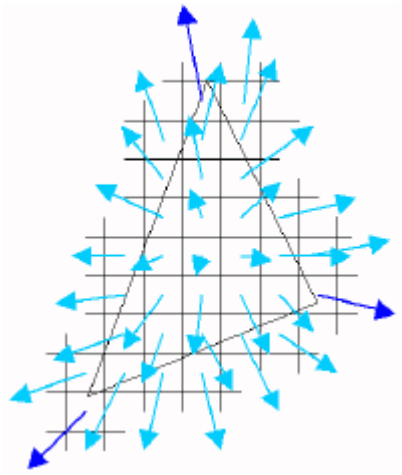
3.1 Phong shading

Phong shading je technika stínování objektů. Vymyslel ji a poprvé implementoval Bui Tuong Phong v roce 1973. Phong shading je založen na interpolaci normál na povrchu polygonu pro každý pixel a následného vypočítání odrazu světla. Mezi techniky shadingu patří také Flat shading a Gouraud shading. Flat shading je založen na principu výpočtu úhlu mezi normálou povrchu polygonu a vektorem směru světla. Následně je vypočítáno osvětlení pro celý polygon podle toho úhlu a barvy polygonu. Gouraud shading vypočte osvětlení podle normál v každém vrcholu polygonu a poté lineární interpolací vypočte barvu pro každý pixel polygonu. Tyto dvě techniky osvětlení mají, ale horší kvalitu než Phong shading.

Phong shading pracuje s tzv. Phongovým odrazovým modelem světla od povrchu objektů. Tomuto odrazovému modelu se také říká Phong lighting, protože počítá odrazy světla podle normál jednotlivých pixelů. Technika Phong shadingu je tedy realizována pro každý pixel, a je třeba ji implementovat ve fragment shaderu. Jelikož Phong shading zahrnuje tzv. Phongův model, popíšu právě tento model odrazu světla.

Každý objekt ve scéně je tvořen polygony. Každý polygon je definován vertexy. Všechny vertexy polygonu mají svoji normálu. Tuto normálu je třeba interpolovat pro každý pixel, který tvoří daný polygon. Podle obrázku 8. můžeme vidět, jak jsou normály interpolovány pro každý pixel

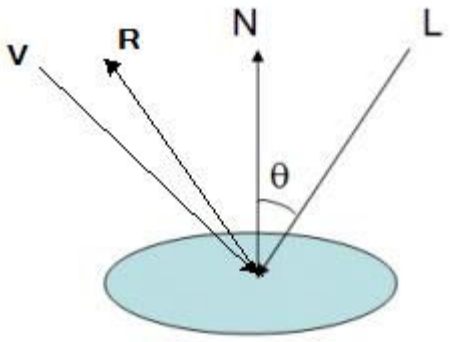
polygonu. Tmavě modrou barvou jsou označeny vektory normál, které jsou definovány ve vrcholech polygonu, tedy každého vertexu. Světle modrou barvou jsou vidět vektory normál, které jsou interpolované pro každý fragment daného polygonu. Interpolovány jsou na základě normál jednotlivých vertexů.



Obrázek 8. Výpočet normál per-pixel

Získání normál není nijak složité, je třeba si ale uvědomit, že normály musí mít souřadnice v eye-space. Pro výpočet Phongova modelu odrazu světla nám nestačí znát pouze normály jednotlivých fragmentů polygonu. Další vektory, které jsou zahrnuty do výpočtu, jsou uvedeny na obrázku 9.

Potřebujeme také vědět, zda je zpracovávaný fragment polygonu přivrácen, nebo odvrácen ke zdroji světla. Jelikož známe normálu, stačí nám vypočítat si vektor L, což je vektor od pozice světla k danému fragmentu. Úhel mezi normálou a vektorem světla zjistíme vektorovým součinem, jehož výsledkem je cosinus hledaného úhlu. Vektor R nám značí odraz vektoru světla od normály fragmentu. Posledním vektorem, který potřebuje znát, je vektor V, což je pozice fragmentu v eye-space souřadnicích, pouze je obrácená.



Obrázek 9. Phong reflection model

Pomocí vektorů světla L a normály N bychom dokázali vypočítat stínování objektů, tj. které fragmenty jsou přivrácené světlu a které nikoliv. Pozorovatel, který se na daný objekt dívá (značí ho vektor V na obrázku 9.), by měl přijímat odražené světlo od povrchu objektu. Jak moc velký odlesk světla na povrchu objektu bude pozorovatel vnímat, je udáno úhlem, který svírá vektor V a R. Můžeme se sami přesvědčit že, bude-li odraz světla totožný nebo skoro stejný jako vektor V, pod kterým se pozorovatel dívá, bude i výsledný odlesk na povrchu objektu velký. Naopak bude-li rozdíl mezi vektory V a R příliš velký, téměř 90°, pozorovatel skoro žádný odlesk vnímat nebude. Tento úhel opět vypočteme pomocí vektorového součinu vektorů V a R. Nyní jsem principielně popsal Phongův model odrazu světla.

Každý objekt a světlo v OpenGL má jasně definované složky. Každý objekt je popsán materiálem, který definuje jeho vlastnosti, jakou má barvu, jak reaguje na dopadající světlo, atd. V OpenGL je materiál jednotlivých objektů definován čtyřmi složkami a jednou konstantou. Tyto složky se nazývají ambient, diffuse, specular a emission.

Ambient složka materiálu definuje barvu celého objektu, tzn., jakou barvou vnímáme celý povrch objektu. Další složka materiálu se jmenuje diffuse. Tato složka má přímou návaznost na ambientní složku. Definuje barvu, kterou jsou zastíněny fragmenty odvrácené od zdroje světla. Tato barva by měla být stejná jako barva ambientní složky, neboť tvoří taktéž výslednou barvu objektu.

Specular složka definuje, jakou barvu budou odrážet odlesky od světla na povrchu objektu. Tyto tři složky jsou nezbytné pro výpočet Phongova modelu. Dále OpenGL definuje emission složku, která udává, jakou barvu bude daný objekt vyzařovat. Poslední informací, která popisuje materiál v OpenGL, je konstanta shininess. Tato konstanta umocňuje specular komponentu materiálu a světla.

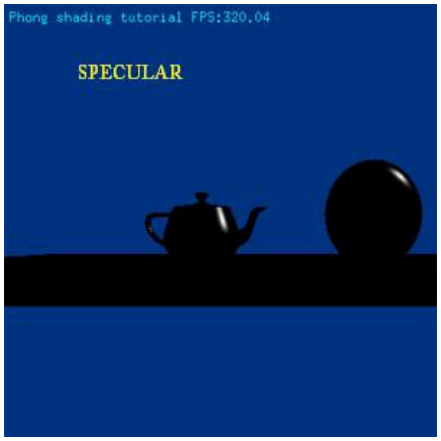
Nyní jsem definoval materiál, tak jak je popsán v OpenGL. Světla můsí být také popsána složkami. Tyto složky jsou tři a nazývají se stejně jako u materiálu ambient, diffuse a specular. Každá z těchto složek ovlivňuje stejnou složku materiálu. Ambient složka světla tedy odráží ambientní složku dopadajícího světla atd.

Ambientní složka světla udává, jakou bude mít světlo barvu. Například, chceme li nasimulovat světlo v lampičce, které svítí žlutě, přiřadíme tuto barvu právě ambient komponentě světla. Diffuse a specular složky se nejčastěji nastavují na bílou nebo černou. Podle toho, zda je světlo v OpenGL aplikaci povoleno nebo nikoliv. Tyto dvě složky reagují a odrážejí diffuse a specular složku materiálu.

Další detaily o definici světel a materiálu jsou popsány v OpenGL dokumentaci. Já jsem vyjmenoval a popsal pouze ty složky, které jsou nezbytné, abychom pochopili, jak funguje Phongův model odrazu světla. Na obrázcích 10a, 10b a 10c, jsou zobrazeny, jak vypadají jednotlivé složky ambient, diffuse a specular ve scéně, kterou zobrazujeme. Složením těchto tří obrázků dostaneme výsledné zobrazení scény.

Obrázek 10a.

Obrázek 10b.



Obrázek 10c.

Jak bylo řečeno v předchozím odstavci, ambient složka materiálu a světla spolu interagují a vytvářejí tak viditelnou barvu na povrchu každého objektu. Intenzita ambientní složky celé scény se vypočte podle rovnice:

$$i_a = m_a l_a \qquad (2.0)$$

V rovnici 2.0 m_a je ambient složka materiálu vynásobená stejnou složkou světla l_a. Výsledek poté tvoří proměnná i_a, která udává intenzitu ambientní složky ve výsledném fragmentu. Je třeba si uvědomit, že ambientní složka je stejná pro všechny světla ve scéně, proto ji nepotřebujeme počítat pro každé světlo.

Další složkou, která tvoří výslednou intenzitu fragmentu, je diffuse. Diffuse složka se spočítá vynásobením materiální a světelné složky (diffuse), dále je třeba rozhodnout, zda je fragment v polygonu přivrácen nebo odvrácen od světla. Toto docílíme, již popsaným postupem o pár odstavců výše. Musíme znát vektor světla L a normálu fragmentu N, mezi nimi si zjistit úhel, který svírají.

Provedeme tedy skalární součin a získáme cosinus námi hledaného úhlu. Výsledná diffuse intenzita fragmentu je dána matematickým zápisem:

$$i_d = m_d \left(\vec{l} \cdot \vec{n}\right) l_d \tag{2.1}$$

Rovnice 2.1 udává diffuse komponenty materiálu resp. světla m_d a l_d. L je vektor od pozice světla k zpracovávanému fragmentu v eye-space a N je normála fragmentu. Výsledek je poté uložen do intenzity diffuse složky i_d. Pokud máme ve scéně více aktivních světel, je třeba pro každé světlo tuto diffuse intenzitu spočítat zvlášť a následně je sečíst. Diffuse složka nám ve výsledné scéně vytváří stínování objektů, které se ve scéně nacházejí. Odtud také pochází název Phong shading.

Poslední složkou, kterou je třeba znát, je specular složka. Specular komponenta nám vytváří na povrchu objektu tzv. odrazy světelných paprsků, které pozorovatel vnímá. Jak moc budou silné, záleží na konstantě shininess (viz výše nebo dokumentace k OpenGL) a také na tom kolik odraženého světla pozorovatel scény vnímá. Rovnice, která vypočte specular intenzitu fragmentu je dána podle:

$$i_s = m_s \left(\vec{r} \cdot \vec{v}\right)^s l_s \tag{2.2}$$

V rovnici 2.2 m_s a l_s jsou specular složky materiálu respektive světla a s je konstanta shininess. Vektor R nám udává směr odraženého světla podle normály N fragmentu a vektor V definuje směr pohledu kamery v eye-space souřadnicích. Pro každé světlo ve scéně se musí specular složka spočítat zvlášť a vypočítané specular intenzity sečíst. Výsledná specular intenzita je uložena do proměnné i_s.

Výslednou barvu fragmentu dostaneme součtem všech tří vypočítaných intenzit složek ambient, diffuse a specular. Uvedená rovnice 2.3 se dá použít pro výpočet Phongova modelu, jednoho nebo více světel.

$$i_p = i_a + \sum_{lights} i_d + i_s \tag{2.3}$$

Pokud do této rovnice (2.3) dosadíme za i_a, i_d, i_s rovnice 2.0, 2.1 a 2.2 dostaneme výslednou rovnici pro výpočet Phongova modelu odrazu světla (2.4).

$$i_p = m_a l_a + \sum_{lights} m_d \left(\vec{l} \cdot \vec{n}\right) l_d + m_s \left(\vec{r} \cdot \vec{v}\right)^s l_s \tag{2.4}$$

V této kapitole 3.1 jsem uvedl princip Phongova stínování (Phong shading). Popsal jsem také model Phongova odrazu světla, jinak nazývaný Phong lighting. Dále jsem se zabýval matematickou částí popisu Phongova modelu, která je nezbytná pro pochopení a správnou implementaci ve vertex a fragment shaderu, které budou použity v OpenGL aplikaci. Bylo také potřeba uvést popis definice materiálu a světla v OpenGL. Více o jejich složkách a hodnotách se dá najít v dokumentaci OpenGL.

3.2 Motion blur

Motion blur je realistický efekt, který vzniká při vysoké rychlosti, kdy rychle pohybující objekty se nám z našeho pohledu jeví jako rozmazané. Tento efekt můžeme ve skutečnosti pozorovat při jízdě autem na okolní krajině, nebo když se díváme z mostu na rychle jedoucí vlak. Myslím si, že motion blur je jeden z efektů, který dodává počítačové hře na realističnosti. V závodní hře můžeme například vnímat pocit z rychlosti nebo v akční hře se může vyskytnout taková situace, že nevíme, kde nám hlava stojí. Takováto počítačová hra může mít bez efektu motion bluru přibližně 60 nebo více fps (snímků za sekundu), ale s motion blurem sice ztrácí na plynulosti, však výsledný dojem vypadá o poznání lépe.

Motion blur se dá realizovat několika technikami, jako je změna geometrie, aplikace rozmazané textury, atd. Tyto techniky pracují a rozmazávají konkrétní objekt. Dalšími technikami jsou tzv. image-space techniky, které neprovádí generování motion bluru pouze na objektu ale celém obrazu scény. Jednou z těchto technik je image-space motion blur s použitím post-processing efektu a právě tuto techniku jsem implementoval ve svém tutoriálu.

3.2.1 Image-space Motion blur

Termín post-processing znamená, že celá scéna je nejprve zobrazena bez motion blur shaderů. Poté je na takto zobrazenou scénu povolán fragment shader, který provede nezbytné kroky, pomocí kterých vznikne motion blur. Tato technika je velmi jednoduchá a nevyžaduje modifikaci vertex a fragment shaderů, které se starají o zobrazování scény bez motion bluru. Je tedy jednoduché ji implementovat do již vytvořeného herního enginu přidáním jednoho pouhého fragment shaderu.

Technika image-space motion bluru se dá realizovat pomocí depth bufferu. Depth buffer používá OpenGL aplikace, do kterého jsou ukládány z-ové souřadnice jednotlivých pixelů obrazu. Z těchto hodnot depth bufferu je po zobrazení celé scény potřeba vygenerovat depth texturu (shadow mapu). Můžeme říct, že je to převedení jednotlivých pixelů z depth bufferu na texturu. Depth textura mé scény je ukázána na obrázku 11. Tato textura bude sloužit jako vstup texturovací jednotky fragment shaderu, za účelem získat vzdálenost, kterou urazil každý fragment z předchozího do současného bodu (pozice). Tyto hodnoty se jinak nazývají velocity mapa.

Obrázek 11. Depth textura

Abychom byli schopní získat uraženou vzdálenost pixelu (velocity mapu), je třeba si vytvořit v OpenGL aplikaci depth texturu. Jak už bylo řečeno depth textura je obraz depth bufferu, převedený na texturu (viz Obrázek 11., depth textura krychle). Hodnoty depth bufferu jsou interpolované hodnoty z-ových souřadnic jednotlivých fragmentů, které tvoří polygony našeho objektu. Extrahováním těchto hodnot z detph bufferu jsme schopni získat z-ovou souřadnici pixelu a k ní přidat interpolovanou texturovací souřadnici, která byla použita pro náhled do depth textury k získání oné z-ové souřadnice. Takto sestavený bod má souřadnice, které leží v souřadném systému okna OpenGL aplikace, tj. jsou to souřadnice po aplikaci viewport transformací. Rovnice 3.0 definuje bod H, který jsme právě získali. Souřadnice všech jeho os musí být v rozmezí [-1, 1], kde bod [0, 0] je střed okna aplikace.

$$H = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1 \right) \tag{3.0}$$

Bod H definuje současnou pozici fragmentu v okně aplikace. Tuto pozici je třeba přetransformovat na předchozí pozici, kterou fragment zaujímal v předchozím snímku. K tomu lze použít současná modelview-projection matice. Tato matice definuje všechny transformace modelu, pohledu a projekce, které jsou aplikovány na právě zpracovaný fragment. Tuto modelview-projection matici je třeba invertovat, tj. vytvořit z ní inverzní modelview-projection matici. Takto definovanou maticí vynásobíme bod H a dostaneme pozici fragmentu, která je ekvivalentní k souřadnému systému, který dostaneme po aplikování modelových transformací (world-space). Výslednou rovnici na převod bodu H pomocí inverzní modelview-projection matice udává matematický výraz 3.1.

$$W = \mathbf{M}^{-1} \times H \tag{3.1}$$

Získaný bod W je bod fragmentu v world-space. Tento bod je třeba podělit 4. komponentou w, abychom ho dostali v rozsahu [-1, 1], protože ho použijeme k výpočtu pozice fragmentu v okně. Takto upravený bod W transformujeme pomocí předchozí modelview-projection matice, kterou si zjistíme pomocí OpenGL aplikace. Rovnice 3.2 ukazuje výsledek.

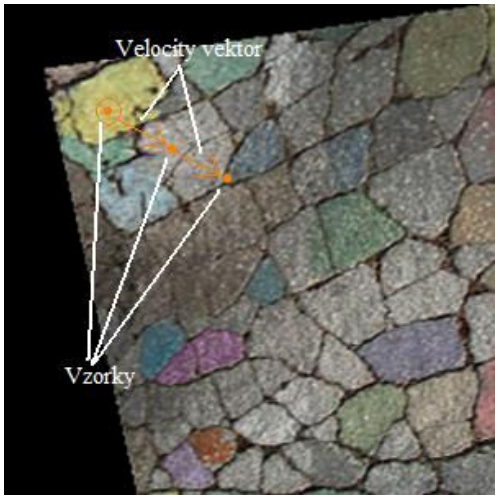
$$H_{prev} = \mathbf{M}_{\mathbf{prev}} \times W \tag{3.2}$$

Získaný bod H_{prev} nám udává souřadnice fragmentu v předchozím snímku. Stejně jako současná pozice fragmentu H, musí být i předchozí pozice podělena 4. Komponentou w, abychom získali rozsah -1 a 1. Tyto rozsahy jsou nezbytné, abychom mohli získat vzdálenost, kterou daný fragment urazil z předchozího do současného snímku.

Máme získané body H a H_{prev}, což je současná resp. předchozí pozice fragmentu v okně. Tyto dva body následně použijeme pro výpočet vzdálenosti, kterou fragment urazil (velocity vektor). Tento výpočet nám popisuje rovnice 3.3, kde rozdílem současné a předchozí pozice fragmentu vypočteme per-pixel velocity pro daný fragment, který je ale v rozmezí -2 a 2, proto ho musíme podělit dvěma, abychom dostali hodnotu mezi -1 a 1, protože budeme per-pixel velocity používat při náhledu do scény. Takto vypočítaná per-pixel velocity udává vzdálenost, kterou urazil fragment z předchozího do současného snímku.

$$\vec{v} = \frac{H - H_{prev}}{2.0} \tag{3.3}$$

Nyní jsem vysvětlil, jak získat velocity vektor, který budu používat pro náhled do tzv. textury scény. Texturu scény získáme pomocí OpenGL aplikace prostým zkopírováním zobrazeného framebufferu do textury. Jelikož se jedná o post-processing efekt, tak výsledný motion blur je právě docílen zpracováním výsledného obrazu scény. Tento obraz se právě nazývá textura scény. Obrázek 12. Popisuje, jak lze docílit motion bluru.



Obrázek 12. Vytvoření motion bluru

Princip je zcela jednoduchý, vezmeme vygenerovanou texturu scény, souřadnice, které jsme použili při náhledu do depth textury, a uděláme náhled do scény. Tím zjistíme barvu prvního vzorku. Počet vzorků nám udává výslednou kvalitu zobrazení scény s motion blurem. Další vzorky dostaneme

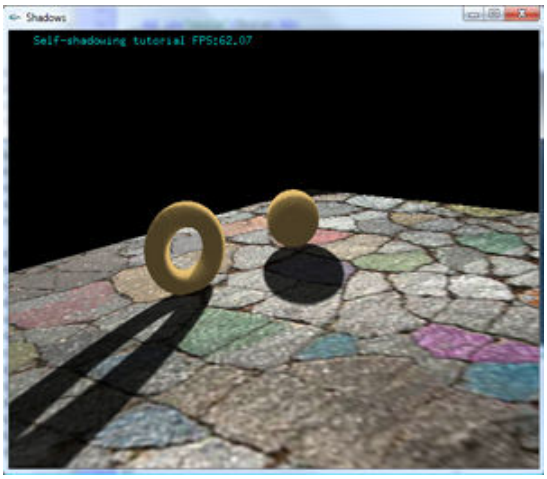
tak, že k současné texturovací souřadnici přičteme velocity vektor (vzdálenost, kterou urazil fragment) a uděláme další náhled. Po sérii námi zvolených náhledů získáme sérii barev, z kterých uděláme průměr a dostaneme výslednou barvu fragmentu. Tuto funkci popisuje matematický zápis rovnice 3.4.

$$c_{fin} = \sum_{i=1}^{n=8} f_{sampler}\left(T + i * \vec{v}\right) \tag{3.4}$$

V rovnici 3.4 definuji c_{fin} jako výslednou barvu fragmentu, $f_{sampler}$ jako funkci pro náhled do textury scény. Bod T značí texturovací souřadnici, i počet iterací (vzorků) a vektor V je vzdálenost, kterou urazil fragment (velocity vektor).

V kapitole 3.2, jsem popsal teoreticky princip motion bluru, který jsem implementoval pomocí post-processing efektu. Mnou zvolená technika vykonává image-space motion blur, tj. motion blur je generován v celém obrazu scény, ne na jednotlivých objektech. Dále jsem uvedl matematickou část, kterou využiji při implementaci mého motion blur efektu pomocí fragment shaderu.

3.3 Self-shadowing



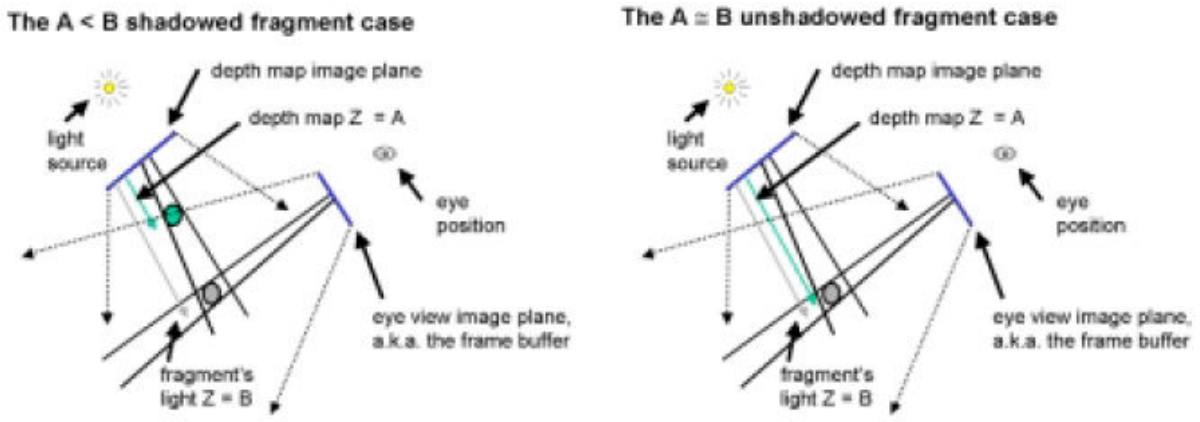
Stíny dokážou udělat 3D grafiku, aby vypadala lépe. Bez nich vypadá zobrazená scéna nepřirozeně a všechny objekty se nám jeví jako ploché a nemáme u nich pojetí o tom, jak daleko jsou od pozorovatele. Je dobré použít některou z technik, která dokáže stíny vytvořit. Těchto technik je velké množství, ale pouze pár z nich dokáže udělat opravdu kvalitní stíny a také brát v potaz vlastnosti objektů, které stíny vrhají. Realistická scéna by neměla umět pouze zobrazovat stíny jednotlivých objektů, ale měla by dokázat generovat samostínění a stínit jeden objekt druhým. Samostínění nebo-li, anglický termín Self-shadowing je vlastnost objektů, které vrhají stíny na samy sebe. Tento jev můžeme pozorovat u objektů, které mají konkávní tvar, nebo-li jsou prohnuté. Pěkný příklad self-shadowing můžeme vidět na vnitřní straně kulatého prstence na motivačním obrázku zobrazeném výše. Technik generující stíny je mnoho, ale jen pár z nich dokáže vytvořit samostínění objektů. Například můžu uvést shadow mapping, shadow volumes nebo hybrid shadows, atd.

3.3.1 Shadow mapping

Techniku shadow mapping vynalezl Lance Williams v roce 1978. Shadow mapping je technika, která dokáže vygenrovat stíny, které poté aplikuje, pomocí texturovací jednotky pixel shaderu, na zobrazenou scénu z pohledu kamery. Tato technika, se dá realizovat pomocí hardware grafické karty použitím verte a pixel shaderů, proto se jí také říká hardware shadow mapping. Stíny vytvořené touto technikou, jsou jednoduché na použití a na rozdíl od shadow volumes nepotřebují generování další pomocné geometrie jednotlivých objektů.

Základem shadow mappingu je tzv. shadow mapa. Možná si vzpomínáte na depth texturu, kterou jsme vytvářeli v kapitole 3.2.1, pomocí které jsme vytvářely motion blur. A právě tato depth textura je jiný název pro shadow mapu.

K vytvoření shadow mapy se v OpenGL používa depth buffer, do kterého jsou ukládány jednotlivé hodnoty z-ových souřadnic, tak jak jsou fragmenty zapisovány do framebufferu. Musíme si však uvědomit, že pokud vytváříme shadow mapu, která má generovat stíny, musíme nejprve scénu zobrazit z pozice světla. U takto zobrazené scény se nám do depth bufferu zapíšou z-ové souřadnice jednotlivých fragmentů, které nám říkají, jaká je vzdálenost fragmentu od světla. Takto vytvořený depth buffer poté zkopírujeme do textury, která se nazývá shadow mapa. Následně je scéna zobrazena z pohledu kamery a pro kazdý fragment v okně je zjištěno, zda leží ve stínu nebo nikoliv. Tohoto se docílí porovnáním s shadow mapu, vygenerovanou v předchozím kroku. Na obrázku 13. je uvedeno, jak se zjistí, zda je fragment ve stínu nebo ne.



Obrázek 13. stínovací porovnání

Schéma vlevo na obrázku 13. ukazuje porovnání fragmentu s shadow mapou, kde výsledný fragment bude zastíněn. Šedé kolečko představuje pozici fragmentu, která byla spočítána při zobrazení scény z pozice světla. Na obázku je označena bodem B. Modré kolečko představuje pozici fragmentu, která byla zjistěna náhledem do shadow mapy, která obsahuje vzdálenosti fragmentů od světla, kterému daná shadow mapa patří. Na obrázku tuto pozici označuje bod A. Nyní dojde

k porovnání těchto dvou pozic A a B. Jak vidíme pozice A je blíže světlu než pozice B, z toho vyplývá, že námi porovnávaný fragment s shadow mapou bude zastíněn.

Naopak schéma vpravo na obrázku 13 ukazuje případ, kdy výsledný fragment po porovnání bude nezastíněn, tj bude přijímat osvětlení. V tomto případě šedé kolečko a modré kolečko mají stejnou pozici, tzn. fragment po porovnání, bude nezastíněn. Fragmenty, které budou osvětleny, mohou mít pozici A vetší nebo rovnu pozici B (podle obrázku 13.).

Nyní je třeba si říci, jak toto porovnání zapsat matematicky, abychom ho poté mohli implementovat ve fragment shaderu. Představme si, že máme bod P, který představuje bod B na obrázku 13, což je pozice fragmentu ve viditelné oblasti světla. Rovnice 4.0 poté ukazuje porovnání pozice fragmentu s shadow mapou.

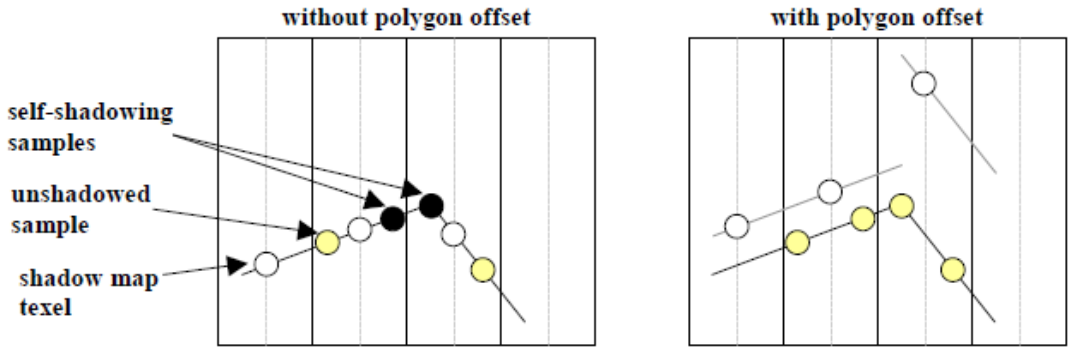
$$P_z \leq shadow\_mapa\left(P_x, P_y\right) \tag{4.0}$$

Podle rovnice 4.0 je porovnána hodnota shadow mapy určená indexy P_x a P_y s hodnotou P_z, což je z-ová souřadnice fragmentu a je rozhodnuto, zda je fragment zastíněn nebo nikoliv. Jelikož porovnání fragmentu s shadow mapou probíhá v texturovací jednotce fragment shaderu, je třeba rovnici 4.0, upravit, aby splňovala podmínky perspektivní projekce. Toto je popsáno rovnici 4.1.

$$\frac{P_r}{P_q} \leq f_{sampler}\left(\frac{P_s}{P_q}, \frac{P_t}{P_q}\right) \tag{4.1}$$

V rovnici 4.1 představuje bod P interpolovanou homogenní texturovací souřadnici, která je shodná s pozicí porovnávaného fragmentu. Odtud pochází jiné označení jednotlivých komponent bodu P. Porovnávací funkce texturovací jednotky, porovná hodnotu P_r (resp. P_z) s hodnotou shadow mapy zadanou interpolovanými texturovacími souřadnicemi P_s a P_t (ekvivalentní P_x, P_y), které jsou podělené hodnotou P_q (ekvivalent hodnoty P_w). Je to z toho důvodu, že všechny 2D textury mají rozměry od 0 po 1.

Tímto jsem popsal, jak funguje porovnávání z-ové souřadnice fragmentu s hodnotou v shadow mapě. Bohužel, při takovémto porovnávání vznikají na stěnách těles, která jsou samo stíněna tzv. artefakty. Tyto artefakty se projevují špatně namapovanou shadow mapou na povrch stíněného objektu, když porovnávaná hodnota v shadow mapě má stejnou hodnotu jako z-ová hodnota texturovací souřadnice. K odstranění těchto artefaktů se dá použít tzv. polygon offset. Polygon offset je malá hodnota, která se přidá k texturovacím souřadnicím, které se používají pro náhled do shadow mapy. Výsledek použití polygon offsetu, je vidět na obrázku 14.



Obrázek 14. Polygon offset

Na obrázku 14. vlevo jsou vidět patrné fragmenty, na kterých vznikají samo stínící artefakty v důsledku špatného porovnání hodnot s shadow mapou. Obrázek vpravo ukazuje řešení přičtením malého čísla k texturovacím souřadnicím, které se používají pro náhled do textury. Tímto vznikne, že porovnávaná hodnota bude mít vždy větší nebo menší hodnotu, než hodnota v shadow mapě. Výsledkem je korektně mapovaný stín na povrch objektu.

V další části teorie k shadow mapping je třeba si říci, jak spočítat texturovací souřadnice, které se použijí jako souřadnice fragmentu k porovnání a index k náhledu do shadow mapy.

Při zobrazování scény z pozice světla, je třeba si někde uložit všechny modelové, pohledové a projekční transformace, které jsme použili. Potřebujeme je totiž k získání pozice, kde byl fragment umístěn z pohledu světla. Transformace jsou v OpenGL uloženy v maticích o velikosti 4x4 (řádky, sloupce). Nejprve je třeba si připravit tzv. bias matici (4.5), kterou se budou násobit všechny transformace.

$$\mathbf{Bias} = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.5}$$

Nyní stačí vzít modeview matici a projekční matici, které byly použity při zobrazení scény z pozice světla a konkatenací s bias matici a pozicí vertexu V, vypočítat texturovací souřadnici T pro tento vertex. Toto provádí rovnice 4.6.

$$\begin{pmatrix} T_s \\ T_t \\ T_r \\ T_q \end{pmatrix} = Bias \times P_{light} \times MV_{light} \times \begin{pmatrix} V_x \\ V_y \\ V_z \\ V_w \end{pmatrix} \tag{4.6}$$

Tímto bych uzavřel teoretickou část a posunul se na část implementace.

4 Implementace

V kapitole 3. jsem se zabýval teorií jednotlivých mnou vybraných technik, které pomáhají přizpůsobovat počítačovou grafiku tak, aby byla více realistická. Tato kapitola nazvaná Implementace se bude zabývat tím, jak teoretickou část každé real-time zobrazovací techniky převést do praxe. Jelikož se moje bakalářská práce zabývá použitím shaderů v real-time grafice, je jasné, že nyní popíši, jak jednotlivé techniky implementovat pomocí vertex a fragment shaderů s tím, abychom je mohli použít v OpenGL aplikacích.

Kapitola Implementace je rozdělena do tří podkapitol, kde každá podkapitola se zabývá realizací technik Phong shading, motion blur a self-shadowing pomocí vertex a fragment shaderů. Následně bude implementace každé techniky rozdělena do částí Nastavení aplikace, Vertex shader a Fragment shader. V části Nastavení aplikace uvedu, jak jsem implementoval svou aplikaci v OpenGL, a nejdůležitější části, které jsou potřeba, abychom dostali správné zobrazení scény použitím shaderů. Další dvě části Vertex shader resp. Fragment shader se zabývají implementací těchto technik, tak jak jsme si je popsali v kapitole 3.

4.1 Phong shading

V této podkapitole se budu zabývat implementací real-time zobrazovací techniky nazvané Phong shading. Jak jsem již uvedl v teoretické části v kapitole 3.1, Phong shading je technika, která počítá stínování a osvětlení pro každý fragment objektu (per-pixel). Je založena na Phongově modelu odrazu světla a právě zde uvedu, jak tento model implementovat pomocí vertex a fragment shaderů v jazyce GLSL.

4.1.1 Nastavení aplikace

Moje aplikace pro zobrazení scény je vytvořena v jazyce OpenGL. Jelikož Phong shading vyžaduje práci se světlem, musel jsem naprogramovat a povolit v mé aplikaci světelné zdroje. Docílil jsem toho tak, že jsem nastavil parametry světla a materiálu, který udává vlastnosti povrchu objektu, pro který chceme vypočítat Phongův model. Nejprve jsem musel nastudovat dokumentaci k OpenGL, kde je popsáno, jak nastavit světelný zdroj a parametry materiálu.

Následně jsem si ve své aplikaci vytvořil dvě funkce **setupLight()**, která nastaví parametry světla a **set_material()**, jenž definuje materiál objektu. Tyto dvě funkce jsou uvedeny níže:

```
void setupLight(GLenum l, GLfloat *amb, GLfloat *diff, GLfloat *spec,
GLfloat *pos, GLfloat *dir) {
        glLightfv(l, GL_AMBIENT, amb);
        glLightfv(l, GL_DIFFUSE, diff);
        glLightfv(l, GL_SPECULAR, spec);
        glLightfv(l, GL_POSITION, pos);
        glLightfv(l, GL_SPOT_DIRECTION, dir);
        glLightf(l, GL_SPOT_CUTOFF, 15.0);
}

void set_material(GLfloat *l_mamb, GLfloat *m_amb, GLfloat *m_spec,
GLfloat *m_emiss, GLint shin) {
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT, l_mamb);
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, m_amb);
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, m_spec);
        glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, m_emiss);
        glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, shin);
}
```

Funkce **setupLight()** má sedm parametrů. Parametr „l“ udává světlo, které chceme nastavit, v OpenGL je definováno konstantou *GL_LIGHT0-7*. Další parametry udávají, jakou barvu a vlastnosti bude mít moje světlo, tj. definují složky ambient, diffuse, specular. Potřebuju znát také pozici světla (*pos*) a směr svitu (*dir*). Následně je těmito hodnotami světlo inicializováno. Toto se provede pomocí vestavěné OpenGL funkce **glLightfv**.

Nastavení materiálu objektu se provede funkcí **set_material()**, která má pět parametrů, které udávají ambient, diffuse, specular, emission a shininess komponenty materiálu. V OpenGL se vlastnosti materiálu nastaví pomocí vestavěné funkce **glMaterialfv**.

Nyní jsem popsal nejdůležitější úsek implementace mojí aplikace v OpenGL. Jelikož moje shadery pracují se světlem a materiálem, jsou tyto hodnoty velmi důležité, abych mohl spočítat Phongův model. Tyto parametry nejsou třeba do shaderů předávat pomocí vlastních proměnných, protože v jazyku GLSL jsou tyto proměnné vestavěny.

4.1.2 Vertex shader

V této části popíšu implementaci vertex shaderu. Jeho nejdůležitější část je uvedena níže:

```
varying vec3 normal, eyePos;
...
normal = normalize(gl_NormalMatrix * gl_Normal); // normala
eyePos = vec3(gl_ModelViewMatrix * gl_Vertex); // pozice vertexu v eye-
space
gl_Position = ftransform();
```

Vstupem vertex shaderu je normála vertexu, samotný vertex a modelview-projection matice, kterou sestavila aplikace v OpenGL. Všechna tato vstupní data jsou automaticky poskytována, protože jsou to vestavěné proměnné.

Nejprve je třeba převést normálu z object-space (prostor před aplikací transformací) a to tak, že ji vynásobíme modelovou a pohledovou transformací. Takto dostaneme normálu v prostoru, který vidí pozorovatel (eye-space). Jazyk GLSL používá pro transformaci normály do eye-space tzv. *gl_NormalMatrix*, což je 3x3 modelview matice, která je invertovaná a transponovaná (zaměněny řádky a sloupce v matici). Jelikož budu ve fragment shaderu používat normálu v skalárním součinu, je třeba ji normalizovat (jednotkový vektor). Pozici vertexu v eye-space dostanu vynásobením *gl_Vertex* modelovou a pohledovou transformační maticí.

Výstupní data vertex shaderu jsou uložena v proměnných *normal* (normála v eye-space) a *eyePos* (vertex v eye-space). Nakonec je třeba definovat pozici vertexu po perspektivní transformaci, která se uloží do vestavěné proměnné *gl_Position*. K tomuto je použita vestavěná funkce **ftransform**, což je ekvivalentní vynásobení modelview-projection matice a vertexu.

4.1.3 Fragment shader

Fragment shader jsem implementoval tak, aby mi spočítal jednotlivé složky ambient, diffuse a specular, tak jak jsem je popsal v teoretické části. Kód je vidět níže:

```
varying vec3 normal, eyePos;

...
n = normalize(normal); // N vektor
view = normalize(-eyePos); // V vektor kamera nebo pozorovatel
aux = gl_LightSource[i].position.xyz -
 eyePos; // prokazde svetlo si vypocitam L vektor
light_dir = normalize(aux);

ambient += (gl_LightSource[i].ambient * gl_FrontMaterial.ambient); // ambi
ent slozka svetla
light_reflect = reflect(-light_dir, n); // jeho odraz podle normaly
NdotL = max(dot(n, light_dir), 0.0); // skalární součin N.L

// spocitam si diffuse slozku vysledne barvy
diffuse += NdotL * gl_FrontMaterial.diffuse * gl_LightSource[i].diffuse;
RdotV = max(dot(light_reflect, view), 0.0); // skalární součin R.V
// specular slozka
specular += pow(RdotV, gl_FrontMaterial.shininess) *
            gl_FrontMaterial.specular * gl_LightSource[i].specular;
gl_FragColor =  ambient + diffuse + specular; // vysledna barva pixelu
```

Vstupními hodnotami fragment shaderu jsou normála, která je interpolovaná pro každý fragment objektu a pozice vertexu z pohledu pozorovatele (opět interpolovaná pro fragment). Tento shader má za úkol nejprve spočítat diffuse složku, tj. určit, které fragmenty budou zastíněny. Toho jsem docílit tak, že jsem vypočítal skalární součin vektorů světla a normály fragmentu. Tím dostanu cosinus úhlu, pod jakým dopadá světlo na daný fragment. Následně tuto hodnotu vynásobím diffuse složkou materiálu a světla a dostávám výslednou hodnotu diffuse složky. Dále jsem spočítal specular

složku. Pomocí skalárního součinu vektorů odraženého světla a směru pohledu pozorovatele, si určím, pod jakým úhlem dopadá do pozorovatelova oka světlo. Dále tento úhel vynásobím specular složkami světla a materiálu. Výsledné složky jsou sečteny a zapsány do *gl_FragColor*.

4.2 Motion blur

V této podkapitole se budu zabývat implementací motion bluru použitím post-processing efektu. To znamená, že veškeré výpočty jsem prováděl ve fragment shaderu. Vertex shader mi poskytnul pouze potřebná data pro fragment shader.

4.2.1 Nastavení aplikace

V OpenGL aplikaci jsem si vytvořil texturu, do které budu ukládat hodnoty z depth bufferu. Tímto získám tzv. depth texturu, což je zkopírovaný obraz depth bufferu do textury. K tomuto účelu jsem implementoval funkci **generate_depthmap()** (viz. níže).

```
glGenTextures(1, &depth_tex);
glBindTexture(GL_TEXTURE_2D, depth_tex); // nastavime ji parametry a rekne
me
...
...
// ze chceme ukladat hodnoty depth bufferu tj hodnotu 'Z'
glTexImage2D( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, win_w, win_h, 0, GL_DE
PTH_COMPONENT, GL_UNSIGNED_BYTE, 0);
// zrusime linkovani textury
glBindTexture(GL_TEXTURE_2D, 0);
__glewGenFramebuffersEXT(1, &fboid); // vytvorime vlastni framebuffer
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboid);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);

// prilinkovani nasi depth textury k framebufferu do bodu depth bufferu
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
GL_TEXTURE_2D, depth_tex, 0);
```

Nejprve jsem si vytvořil vlastní framebuffer, který ukládá pouze z-ové souřadnice fragmentů (GL_DEPTH_COMPONENT). Dále jsem mu přiřadil texturu, do které se tyto hodnoty poté zkopírují. Tím dostanu obraz depth bufferu, k jehož hodnotám se dá přistupovat pomocí náhledových funkcí fragment shaderu. Důležité je, že výsledná textura musí mít rozměry okna aplikace.

```
scene_p = glGetUniformLocation(p, "sceneTex");
depth_p = glGetUniformLocation(p, "depthTex");
prevModelView_p = glGetUniformLocation(p, "prevModelView");

// funkce renderWithMotionShader()
glUniformMatrix4fvARB(prevModelView_p, 16, 0, ptr);
```

```
glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_2D, scene_tex);
glUniform1iARB(scene_p, 0);

glActiveTextureARB(GL_TEXTURE7_ARB);
glBindTexture(GL_TEXTURE_2D, depth_tex);
glUniform1iARB(depth_p, 7);
```

V další části aplikace jsem nadefinoval tři uživatelské uniform proměnné. První (*sceneTex*) mi posílá do fragment shaderu texturu scény, na které se bude vykonávat výsledný motion blur. Druhou proměnnou je *depthTex*, což je moje vygenerovaná depth textura, pomocí funkce popsané v předchozím odstavci. Poslední proměnnou je modelview transformační matice, která mi udává transformace v předchozím snímku.

4.2.2 Vertex shader

Vertex shader je u post-processing efektu velice jednoduchý, prakticky bych ho nepotřeboval. Implementoval jsem ho pouze na ukázku, jak předávat transformační matice pomocí varying proměnných do fragment shaderu.

```
uniform mat4 prevModelView;
varying mat4 prevMVPM;
varying mat4 MVPMinv;
varying vec4 depthCoord;
...
depthCoord = gl_TextureMatrix[7] * gl_Vertex;
depthCoord = depthCoord / depthCoord.w;
```

Vstupní proměnnou vertex shaderu je předchozí modelview matice, kterou vynásobím s projekční maticí a pošlu pomocí proměnné *prevMVPM* do fragment shaderu. Dále jsou výstupními proměnnými *MVPMinv*, což je invertovaná modelview-projection matice a *depthCoord* souřadnice, která udává pozici vertexu v rozmezí 0 a 1. Tato souřadnice slouží ve fragment shaderu k zjištění z-ové souřadnice fragmentu z depth textury. Pro její výpočet jsem použil matici *gl_TextureMatrix[7]*, což je právě v teorii popsaná matice, která vznikne konkatenací bias, modelview a projection matic. Bias matice slouží k převedení souřadnic vertexu do rozmezí [0, 1]. Samožřejmě se dá použít i jiná texturovací matice.

4.2.3 Fragment shader

Tento shader dělá nejdůležitější funkci. Vypočte mi vzdálenost (*velocity*), kterou fragment urazil a náhledem do textury scény (*sceneTex*) vypočítá výslednou barvu tohoto fragmentu:

```
uniform sampler2D sceneTex;
uniform sampler2D depthTex;
varying mat4 prevMVPM;
varying mat4 MVPMinv;
varying vec4 depthCoord;

float zOverW = texture2D(depthTex, depthCoord.st).z; //z-souradnice pixelu
// view-port pozice pixelu
vec4 H = vec4(depthCoord.x * 2.0 - 1.0, (1.0 - depthCoord.y) * 2.0 -
 1.0, zOverW, 1.0);
vec4 D = MVPMinv * H;
vec4 worldPos = D / D.w; // world pozice
vec4 previous = prevMVPM * worldPos; // predchozi pozice pixelu
...
vec2 velocity = (current.xy - previous.xy) / 2.0; // velocity vektor
// ziskame vzorky a vypocitame vyslednou barvu
for (int i = 0; i < samples; i++) {
        vec4 c_color = texture2D(sceneTex, texCoord); // nahled
        texCoord += velocity;
        f_color += c_color;
}

gl_FragColor = f_color / float(samples);
```

Vstupem pixel shaderu jsou dvě textury, textura scény (*sceneTex*) a depth mapa (*depthTex*). Dále mi vertex shader poskytne předchozí a současnou modelview-projection matici a vypočítanou souřadnici pro náhled do depth mapy.

Ve funkci pixel shaderu je nejprve třeba si zjistit z-ovou souřadnici fragmentu pomocí vestavěné funkce texture2D. Potom vytvořím bod H, což je pozice fragmentu v okně. Důležité je, aby souřadnice bodu H byly v rozmezí [-1, 1], protože z něho budu počítat per-pixel velocity. Získaný bod H přetransformuju do world-space pozice, kterou musím taktéž normalizovat, tj. musí být v rozmezí [-1, 1].

Nyní, když mám world-space pozici, můžu ji transformovat pomocí předchozí modelview-projection matice a získat pozici fragmentu v předchozím snímku. Mnou nalezený bod jsem pojmenoval *previous* a je třeba ho ještě normalizovat, aby měl souřadnice mezi -1, 1. Velocity vektor se spočítá rozdílem současné H (*current*) a předchozí *previous* pozice fragmentu. Jelikož velocity vektor používám k náhledu do scény, potřebuju znát pouze x a y souřadnici.

V posledním úseku fragment shaderu mám proměnnou *samples*, která mi udává počet vzorků, kolikrát je třeba přičíst velocity vektor a nahlédnout do *sceneTex*, abych získal sérii barev, ze kterých udělám průměr. Výslednou barvu poté uložím do *gl_FragColor*. U některých grafických karet, je třeba transformační matice nalinkovat přímo do fragment shaderu pomocí uniform proměnné.

4.3 Self-shadowing

V této podkapitole se budu zabývat implementací techniky samo stínění. Jelikož samo stínění se nedá realizovat samostatně, musel jsem použít a implementovat jednu z technik generování a mapování stínů na povrchy objektů. Techniku, kterou jsem si zvolil, se nazývá shadow mapping.

4.3.1 Nastavení aplikace

Nejprve jsem potřeboval v OpenGL aplikaci vytvořit shadow mapu, což je textura, která definuje z-ové souřadnice fragmentů, které jsou ukládány do depth bufferu (viz kapitola Teorie 3.3.1). K tomu jsem vytvořil funkci **generate_depthmap()** (viz níže):

```
// pro pouziti shadow2DProj v shaderu
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
            GL_COMPARE_R_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE, GL_INTENSITY);

// ze chceme ukladat hodnoty depth bufferu tj hodnotu 'Z'
glTexImage2D( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, shMapW, shMapH, 0, GL_
DEPTH_COMPONENT, GL_UNSIGNED_BYTE, 0);

// zrusime linkovani textury
glBindTexture(GL_TEXTURE_2D, 0);
glGenFramebuffersEXT(1, &fboid); // vytvorime vlastni framebuffer
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboid);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);

// prilinkovani nasi depth textury k framebufferu do bodu depth bufferu
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
                        GL_TEXTURE_2D, shadowTex, 0);
```

Tato funkce je podobná funkci, kterou jsem použil v tutoriálu o motion bluru. Vytvořím si tedy texturu, do které budu ukládat hodnoty depth bufferu. Následně jí nastavím parametry, ale navíc ještě musím povolit porovnávací mód při náhledu do shadow mapy (GL_TEXUTURE_COMPARE_MODE). Tento mód se dá použít pouze s shadow mapou (depth mapou). Abych mohl použít porovnávací mód je třeba ve fragment shaderu použít náhledovou funkci **shadow2Dproj** (viz implementace fragment shaderu).

Následně je třeba si v paměti zarezervovat místo pro tři hlavní uniform proměnné, *xOffset*, *yOffset* a *shadowTex*. Offsety se přičtou ve fragment shaderu k projektivním souřadnicím pro náhled do shadow mapy, protože jinak dostanu značně nepřesné mapování stínů na scénu (tzv. self-shadowing artefakty). *ShadowTex* proměnná mi slouží k předání shadow mapy do fragment shaderu. V mé aplikaci jsem si rezervoval i *isSelfShadowed* proměnnou, která říká, zda je právě zobrazovaný

objekt ve scéně samostíněn nebo nikoliv. Všechny tyto proměnné musí být inicializovány daty pomocí vestavěné funkce **glUniform1fiARB()**.

```
shadowTex_p = glGetUniformLocationARB(main_prog, "shadowTex");
xOffset_p = glGetUniformLocationARB(main_prog, "xOffset");
yOffset_p = glGetUniformLocationARB(main_prog, "yOffset");
isSelfShaded_p = glGetUniformLocationARB(main_prog, "isSelfShadowed");
...
glUniform1fARB(xOffset_p, 1.0 / (float)shMapW);
glUniform1fARB(yOffset_p, 1.0 / (float)shMapH);
glUniform1iARB(shadowTex_p, 7);
```

4.3.2 Vertex shader

Dostávám se k implementaci vertex shaderu. Vertex shader je velice jednoduchý, protože obsahuje jen pár řádků, zde jsou ty nejpodstatnější:

```
varying vec4 shadowCoord;
...
gl_Position = ftransform();
shadowCoord = gl_TextureMatrix[7] * gl_Vertex;
...
```

Jako vstup shaderu mi slouží modelview-projection matice a pozice vertexu, z kterých získám pozici vertexu po perspektivní transformaci (clip-space), kterou jsem uložil do *gl_Position*. Výstupem vertex shaderu je spočítaná pozice vertexu v clip-space a *shadowCoord*, což je clip-space souřadnice vertexu z pozice světla. Lze si také všimnout, že vypočtená texturovací matice je uložena v *gl_TextureMatrix[7]*, protože jsem pro shadow mapu použil sedmou texturovací jednotku.

4.3.3 Fragment shader

Fragment shader je poslední částí, která porovná hodnotu v shadow mapě se vzdáleností fragmentu od světla a mapuje stíny na zobrazenou scénu:

```
uniform sampler2DShadow shadowTex;
uniform float xOffset;
uniform float yOffset;
varying vec4 shadowCoord;
...
float lookup(vec2 offsetVect) {
        return shadow2DProj(shadowTex, shadowCoord + vec4
(offsetVect.x * xOffset * shadowCoord.w, offsetVect.y * yOffset *
```

```
shadowCoord.w, 0.005, 0.0)).w;
}
...
if (shadowCoord.w > 1.0) {
        // shadow = lookup(vec2(0.0, 0.0));
        float x, y;
        for (y = -3.5; y <= 3.5; y += 1.0) {
                for (x = -3.5; x <= 3.5; x += 1.0) {
                        shadow += lookup(vec2(x, y));
                }
        }

        shadow /= 64.0;
}
```

Vstupem fragment shaderu je texturovací souřadnice *shadowCoord*, předávaná z vertex shaderu. Dále je zde několik uniform proměnných, jejichž hodnoty jsou dodávány z OpenGL aplikace. Především to jsou offsety, pro texturovací souřadnici a shadow mapa.

Funkce **lookup** zajišťuje projektivní náhled do textury a zjištění, jak moc je daný pixel zastíněn. Vstupem funkce jsou offsetové hodnoty, které získáme z OpenGL aplikace. Důležité upozornění je, že hodnoty, které jsem použil, k přičtení k texturovacím souřadnicím, se mění v závislosti na scéně, kde shadow mapping provádíme. Nezaručuji tedy, že dostanete s touto **lookup** funkcí stejné výsledky jako já. Tělo této funkce tvoří vestavěná funkce GLSL **shadow2Dproj**, kde Shadow2D znamená, že ji můžu použít pouze pro náhled do shadow mapy. Proj definuje, že souřadnice, které použiju při náhledu do shadow mapy, budou poděleny 4. komponentou *w*.

Na posledním úseku kódu mám použití lookup funkce. Moje implementace shadow mappingu zahrnuje techniku PCF (Percentage Closer Filtering). Funguje na principu získání sady okolních vzorků pro zpracovávaný fragment. Vzorky se získají náhledem do shadow mapy kolem aktuálního zpracovávaného fragmentu. Získané vzorky jsou následně zprůměrovány a dostanu výslednou hodnotu zastínění aktuálního fragmentu.

Výsledná barva fragmentu je poté dána součtem zastínění a barvou fragmentu. Výsledek je uložen do proměnné *gl_FragColor*.

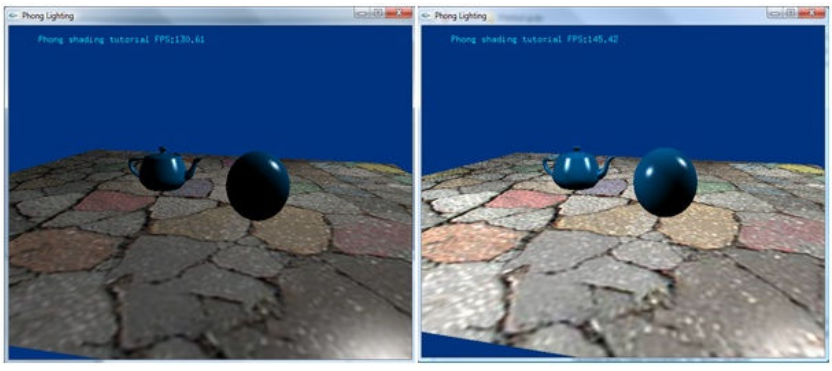
5 Závěr a zhodnocení

Na závěr mé práce bych v několika podkapitolách zhodnotil dosažené výsledky mnou implementovaných real-timových zobrazovacích technik. Tyto techniky nebylo v minulosti možné uvést do praxe, protože výkon tehdejšího grafického hardware byl opravdu žalostný. Celkové zpracování obrazu totiž trvalo velice dlouhou dobu. To se s nástupem moderních grafických karet změnilo. Nejen že grafické karty mají dostatek výkonu, leckdy jím dokonce překypují. Ovšem také k tomu výrazně přispělo vyvinutí high-level shading jazyka, pomocí kterého dokáže programátor napsat svůj zobrazovací algoritmu za několik minut. V podání OpenGL aplikací se tento jazyk nazývá OpenGL shading language.

Nyní bych přistoupil ke shrnutí a zhodnocení mých dosažených výsledků v implementaci Phong shading, motion blur a self-shadowing.

5.1 Phong shading

Technika Phong shading se používá ke stínování a osvětlení objektů ve scéně. Založena je na Phongově modelu odrazu světla. Ukázku Phong shadingu je vidět na obrázku 15. vlevo. Tento obrázek ukazuje scénu z mého tutoriálu zobrazenou pomocí vertex a fragment shaderů, které počítají per-pixel osvětlení za použití Phongova modelu. Scéna zobrazená pomocí fixní funkcionality OpenGL se liší oproti zobrazení za pomoci shaderů tím, že všechny objekty ve scéně jsou zatemněny. Je to z toho důvodu, že OpenGL aplikace nemá definovány žádné algoritmy pro výpočet osvětlení, tak jak to definuje OpenGL dokumentace. O to se starají v mé aplikaci právě shadery.

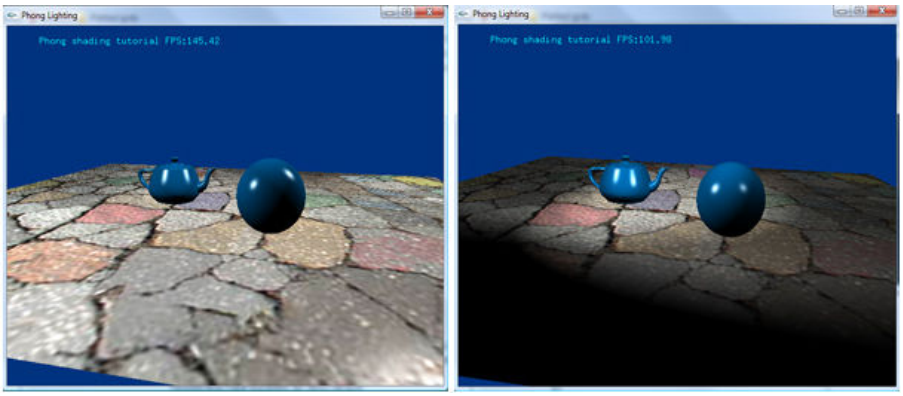


Obrázek 15. Výsledek Phong shadingu

Na obrázku 15. vpravo, je ta samá scéna, kde jsou zapnuty 2 a více světel, tj. Phong shading se dá použit i pro výpočet osvětlení z několika světelných zdrojů.

Jelikož je Phong shading jednoduší na implementaci, snažil jsem se svoji aplikaci rozšířit. Vytvořil jsem dva typy světel, directional a spot. Poté jsem pomocí těchto světel,

nechal zobrazit scénu s Phong shading. Výsledky jsou vidět na obrázku 16. Directional světlo (obrázek vlevo) se dá přirovnat k tomu, jak svítí slunce, tj. všude stejně. Spot světlo naopak můžeme pozorovat při zapnuté lampičce nebo u předních světel u aut, kdy je vidět pouze to co je v kuželu, který světlo vymezuje, všechno ostatní je zastíněno. Spot světlo v OpenGL je tvořeno pouze jedním kuželem, což neodpovídá skutečnosti. Implementoval jsem ho jako v DirectX, kde je tvořeno vnějším a vnitřním kuželem, mezi kterými se oblast lineární interpolací zastiňuje. Můžeme si také všimnout, že odlesky na povrchu objektů, které světlo vytváří, mají na hranici kužele světla menší intenzitu nebo nejsou vidět vůbec.



Obrázek 16. Directional, Spot light

5.2 Motion blur

Při implementaci motion bluru jsem popsal techniku, jak získat souřadnice fragmentu po modelové transformaci pomocí depth bufferu. Ukázal jsem také, že použitím této techniky se dá celkem jednoduše naimplementovat motion blur jako post-processing efekt, který je poté snadno zakomponovatelný do již vytvořeného herního enginu.

Výsledné zobrazení scény s tímto efektem je na obrázku 16. uprostřed. Vlevo je ta samá scéna, ale zobrazená bez použití motionu bluru.

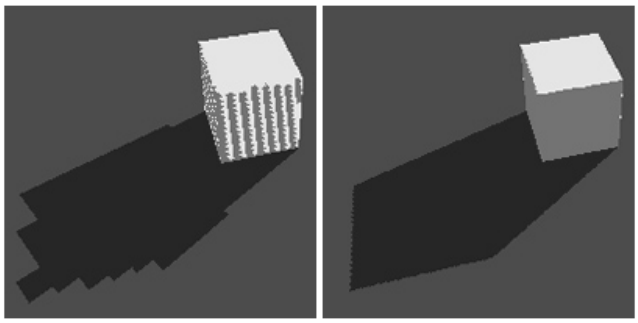
Obrázek 17. Scéna bez/s motion blurem

Můžeme si celkem snadno všimnout, že obraz vypadá o mnoho realističtěji. Při bližším prozkoumání si lze také všimnou, že části objektu, které mají blíže ke kameře pozorovatele, se jeví více rozmazané než části, které jsou v dálce. Samozřejmě části, kde se objekt pohybuje rychleji, jsou taktéž více rozmazány.

Na prvním obrázku 16. uprostřed je použit motion blur s 8 vzorky. Vidíme, že už při takto malém počtu vzorků je výsledný motion blur opravdu hezký a nespotřebuje tolik výkonu našeho hardwaru. Obrázek vpravo ukazuje ten samý objekt s použitím 16 vzorků, kdy výsledná kvalita zobrazení je ještě lepší. Obecně tedy platí, že čím více vzorků tím je motion blur kvalitnější, ale pohyblivost zobrazované scény je čím dál tím menší, klesá tedy počet fps, což se projevuje i na zatížení hardwaru.

5.3 Self-shadowing

K implementaci samostínění jsem použil techniku shadow mapping. Shadow mapping je sama o sobě technika pro vytváření stínů v 3D scéně. Získáme tak dojem opravdového 3D zobrazení. Další techniky, které mají vlastnost samostínění, jsou, shadow volumes (použita v Doom3) nebo hybrid shadows, což je kombinace shadow mapping a shadow volumes. Techniku shadow mapping jsem zvolil, protože má dvě výhody oproti dalším zmiňovaným. Těmi jsou, rychlost a možnost implementace pomocí shaderů na hardware grafické karty. Jedinou nevýhodou je, že potřebujeme co největší velikost shadow mapy, abychom dostali přijatelné detaily stínů, což je ostatně vidět na obrázku 17. (vlevo shadow mapa 160x120, vpravo 1280x960).



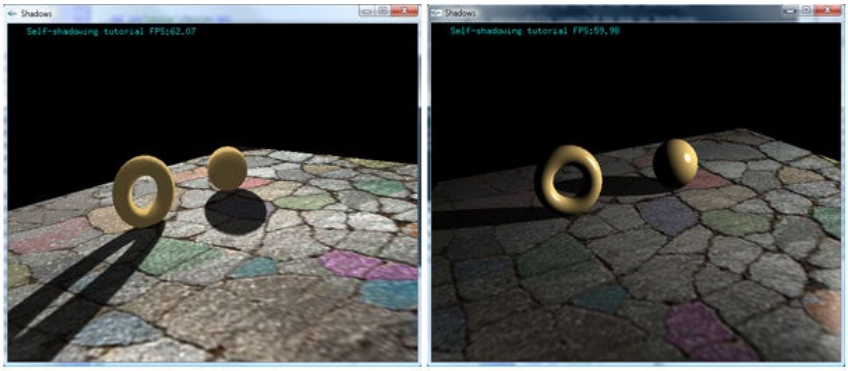
Obrázek 18. Shadow mapping

Moje implementace shaderů používá shadow mapping s PCF (Percentage Closer Filtering). Tento přístup aplikace shadow mapy vytváří jemné a kvalitní stíny. Bohužel musíme také počítat s výraznou ztrátou výkonu. Kvalita výsledných stínů potom závisí na počtu vzorků. Jednotlivé rozdíly mezi kvalitou stínů jsou vyobrazeny na obrázku 18. Obrázek vlevo používá PCF 4 vzorky, uprostřed je obrázek s PCF 16 vzorků a vpravo PCF 64 vzorků.

Obrázek 19. PCF

Výsledné obrázky shadow mapping s mojí scénou jsou vedeny na obrázku 20. V tomto zhodnocení jsem uvedl pouze rozdíly v kvalitě a zobrazení jednotlivých stínů, tak aby byly zřetelné detaily okrajů a celé provedení stínů.

Na závěr bych chtěl ukázat, jak vypadá moje scéna s shadow mapping PCF 64 vzorků. Výsledek je vidět na obrázku 20. vlevo. Kvalitní stíny, ale stále nedělají výsledné zobrazení scény ničím zajímavé, chybí zde totiž výpočet osvětlení. Za tímto účelem jsem se pokusil vyobrazit scénu pomocí dvou shaderů. První vypočítá stíny a druhý pomocí Phongova osvětlovacího modelu vypočítá osvětlení. Takováto scéna je uvedena na obrázku vpravo.



Obrázek 20. Stíny a osvětlení

V této práci jsem se zabýval pouze třemi real-time technikami. Technik, které dělají grafiku realističtější je opravdu velké množství. Doufám tedy v to, že mnoho dalších lidí najde v této práci užitečné informace a možná se i pokusí nějaké další techniky implementovat.

Literatura

[1] Randi J. Rost: OpenGL Shading Language, Second edition, Addison Wesley Professional 2006, ISBN 978-0-321-33489-3

[2] NEIDER, Jackie, DAVIS, Tom, WOO, Mason. *OpenGL Programming Guide* [online]. 1993. Addison-Wesley Publishing Company, 1994 [cit. 2009-05-06]. Dostupný z WWW: <http://unreal.srk.fer.hr/theredbook/>. ISBN 0-201-63274-8.

[3] NGUYEN, Hubert. *GPU Gems 3* [online]. 2007. 2007 , 12.5. 2009 [cit. 2009-05-07]. Dostupný z WWW: <http://http.developer.nvidia.com/GPUGems3/ gpugems3_ch27.html>.ISBN 0-321-51526-9.

[4] SONG HO, Ahn. *OpenGL Transformation* [online]. 2008. 2008 , 12.5. 2009 [cit. 2009-05-06]. Dostupný z WWW: <http://www.songho.ca/opengl/gl_transform.html>.

Seznam příloh

Příloha 1. CD se zdrojovými kódy jednotlivých tutoriálů a elektronická verze bakalářské práce.