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

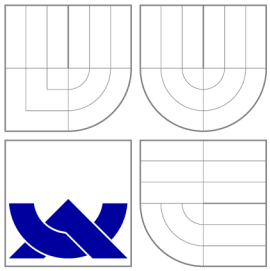
SIMULACE VLNĚNÍ VODY V REÁLNÉM ČASE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

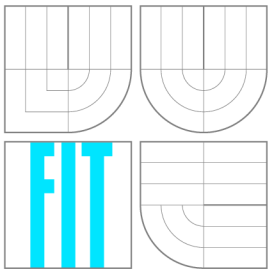
AUTOR PRÁCE
AUTHOR

Bc. MARTIN PILCH

BRNO 2010



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

SIMULACE VLNĚNÍ VODY V REÁLNÉM ČASE

SIMULATION OF WATER WAVES IN REAL-TIME

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MARTIN PILCH
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. ADAM HEROUT, Ph.D.
SUPERVISOR

BRNO 2010

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2010/2011

Zadání diplomové práce

Řešitel: **Pilch Martin, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Simulace vlnění vody v reálném čase**
Simulation of Water Waves in Real-Time
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte algoritmy simulace vlnění a toku vody v reálném čase.
2. Vyberte vhodný algoritmus simulace vlnění a toku vody a implementujte ho.
3. Vytvořte demonstrační aplikaci pro ilustrování možností implementovaného algoritmu.
4. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek pro prezentování projektu.

Literatura:

- dle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- Bod 1 a značné rozpracování bodu 2.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Herout Adam, doc. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 20. září 2010
Datum odevzdání: 25. května 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2
L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Předmětem práce je vytvoření simulace vlnění vody v reálném čase. Implementační platformou je Mac OS X a rozhraní OpenGL. Základem práce je vodní hladina, tvořená výškovou mapou. Metoda pro výpočet výškové mapy je založená na výpočtu sumy sinusoid s komplexními, časově závislými amplitudami. Pro výpočet je použita rychlá Fourierova transformace, Phillipsovo spektrum a gaussovský generátor náhodných čísel. Práce je implementována také pro platformu iOS a optimalizována pro běh na mobilních zařízeních díky použití programovatelného grafického řetězce a optimalizacích při výpočtech a vykreslování.

Abstract

Task of this thesis is creation of real-time simulation of the water waves. It is implemented on Mac OS X platform using OpenGL. This thesis is based on height map surface. Heigh map is computed by suming of sinusoids with complex, time-based amplitudes. Fast Fourier transformation, Phillips spectrum and gauss random generator are used to solve this problem. The thesis is also implemented on iOS platform and optimized to run on mobile devices thanks to using programmable graphic pipeline and other drawing and computing optimizations.

Klíčová slova

Navier-Stokes rovnice, OpenGL, Apple Mac OS X, iOS, Phillipsovo spektrum, Gerstnerovy vlny, FFT, cube map, frustum culling, Vertex Buffer Object, shader, programovatelný grafický řetězec, Phongův světelný model

Keywords

Navier-Stokes equations, OpenGL, Apple Mac OS X, iOS, Phillips spectrum, Gerstner waves, FFT, cube map, frustum culling, Vertex Buffer Object, shader, programmable graphic pipeline, Phong lighting model

Citace

Martin Pilch: Simulace vlnění vody v reálném čase, diplomová práce, Brno, FIT VUT v Brně, 2010

Simulace vlnění vody v reálném čase

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Adama Herouta.

.....
Martin Pilch
23. května 2011

Poděkování

Děkuji svému vedoucímu, panu Adamu Heroutovi, za pomoc při vypracování práce, především za jeho podnětné rady a myšlenky.

Obsah

Kapitola 1

Úvod

Tématem diplomové práce je simulace vlnění vody v reálném čase. Jako implementační platformu jsem si vybral Apple Mac OS X a grafické rozhraní Open GL [2]. Hlavním důvodem, který mě vedl k výběru platformy, je její dobrá podpora rozhraní Open GL, možnost relativně jednoduchého přepsání aplikace pro mobilní platformu iOS a velká komunita vývojářů.

Cílem práce není vytvořit fyzikálně realistickou simulaci vody, ale její co možná nejpřesvědčivější vizuální reprezentaci, která bude působit dojmem vlnící se hladiny moře. Během implementace jsem kladl důraz na rychlost, vizuální stránku a přepsání pro mobilní platformu iOS tak, aby vše běželo pokud možno plynule. I v době neustále rostoucího výkonu procesorů a grafických karet jsou totiž mobilní zařízení omezena svým výkonem, prostor pro optimalizace výpočtů je proto obrovský. Je jednoduché tyto optimalizace použít i na stolních počítačích, a proto je možné na tuto práci možné nahlížet jako na způsob, jak otestovat hranice výkonnosti mobilních zařízení a zároveň jak optimalizovat grafické aplikace napsané pomocí knihovny Open GL.

V této práci nejdříve popíšu jednotlivé metody reprezentace vody a vlnění vody jak v reálném čase, tak metody vhodné pro renderování.

Konkrétně rozeberu Navier-Stokes rovnice, které se používají k výpočtu chování vody. Jedná se o rovnice popisující realistické chování kapalin jako rychlost kapaliny v čase a bodě prostoru.

Dále popíšu Gerstnerovy vlny, rovnice českého matematika a fyzika Františka Josefa Gerstnera, které popisují vlnění vody pomocí funkcí času. Jejich výsledkem je v podstatě výšková mapa.

Nakonec se dostanu k práci Jerryho Tessendorfa, který rozvádí předešlé rovnice a pomocí Fourierovy transformace vypočítává výškovou mapu. Ke generování hodnot se používá Phillipsovo spektrum, což je statistický model vlnění vody.

V další kapitole popíšu, jak jsem z těchto poznatků vycházel a jak jsem je konkrétně použil ve své práci. Proberu teoretické předpoklady pro implementaci světelného modelu a s tím souvisejícího výpočtu normál. Dále rozeberu programovatelný grafický řetězec a také operace s maticemi, které jsou důležité pro implementaci pohledu kamery.

V páté kapitole se zabývám vlastní implementací, konkrétně specifiky programování pod operačním systémem Mac OS X a iOS z hlediska vývoje aplikace pod Open GL a tvorby aplikací obecně, použitím programovacího jazyka Objective-C a také srovnáním výkonnosti vlastní a systémové implementace FFT. Dále v ní probírám výpočet osvětlení a zobrazení prostředí scény. S tím souvisí implementace pohledu kamery a jednoduchý Level-Of-Detail. Popíšu i strukturu aplikace a rozdělení výpočtů mezi grafickou kartu a procesor. Rozeberu

i způsob, jak vytvořit pro operační systémy Mac OS X a iOS jednoduše přenositelnou aplikaci.

Zmíním i několik technik a přístupů, které jsem použil pro optimalizaci, jako jsou Frustum culling, Vertex arrays a Vertex buffer object.

Celou práci také otestuji na několika mobilních zařízeních s platformou iOS a stolních i přenosných počítačích s operačním systémem Mac OS X. Některé naměřené výsledky jsou velice zajímavé.

Práce navazuje na Semestrální projekt, ze kterého přebírá především teorii výpočtu vlnění hladiny vody popsanou v kapitolách 2 a 3 a také některé části kapitoly 5 zabývající se implementací. Oproti semestrálnímu projektu je implementováno mnoho optimalizací, které se příznivě projevují na výkonu, a především je celá aplikace přenositelná na mobilní zařízení s operačním systémem iOS.

Kapitola 2

Metody simulace vody

V této kapitole proberu jednotlivé metody reprezentace vody a vlnění vody. Nejdříve vysvětlím reprezentace pomocí funkcí, konkrétně Gerstnerovy vlny. Ty jsou méně náročné na výpočet, neposkytují však dostatečně kvalitní výsledky.

Následně proberu Navier-Stokes rovnice, které představují nejpřesnější známý model pro reprezentaci kapalin. Ten je však příliš náročný pro reprezentaci v reálném čase a použití v počítačové grafice.

2.1 Gertsnerovy vlny

Existují i jiné reprezentace, než NSE. V drtivé většině vychází z funkcí, které popisují povrch kapaliny nějakou funkcí a pomocí rychlé inverzní Fourierovy transformace je vypočtena hodnota v čase a prostoru.

Gerstnerovy vlny, jako aproximace rovnic pro vyjádření dynamiky kapalin, byly poprvé popsány před 200 lety Františkem Josefem Gerstnerem. Povrch kapaliny je tvořen pohybujícími se body. Vzorec pro výpočet souřadnic je pak následující:

$$\mathbf{x} = \mathbf{x_0} - (\mathbf{k}/k)A\sin(\vec{k}\mathbf{x_0} - \omega t) \tag{2.1}$$

$$y = A\cos(\mathbf{k}\mathbf{x_0} - \omega t) \tag{2.2}$$

V těchto rovnicích je ω frekvence vektoru vlny, $\vec{k}$ je vektor vlny, $\mathbf{k}$ je velikost vektoru vlny nebo také vlnové číslo, $\mathbf{x_0}$ je počáteční hodnota nehybného bodu (x_0, z_0) a $y_0 = 0$.

Pomocí těchto rovnic získáme v podstatě výškovou mapu. Nevýhodou Gerstnerových vln je fakt, že se jedná o jednu sinusovou vlnu v horizontálním a jednu ve vertikálním směru. Tento problém je možné vyřešit součtem několika vln.

2.2 NSE - Navier-Stokes rovnice

Navier-Stokes rovnice popisují proudění *nestlačitelné Newtonovské kapaliny*. Newtonovská kapalina je model, který se řídí *Newtonovým zákonem viskozity*. Ten stanovuje vztah mezi napětím a rychlostí deformace jako přímou úměru. Konstantou úměrnosti je *dynamická viskozita*. Tento model je vhodný především pro popis vody a jiných tekutin. Pro úplnost dodám, že existují i látky, které není možné tímto modelem popsat.

Navier-Stokes rovnice byly popsány nezávisle na sobě dvěma muži. Byli to Claude-Louis Navier a George Gabriel Stokes. Samotné rovnice neřeší pozici jako spíše rychlost nebo

pohyb tekutiny. Řešením rovnic je *velocity field* (pole rychlostí), tedy rychlost tekutiny v daném bodě v prostoru a čase [5].

2.2.1 Matematická reprezentace

Obecný vzorec pro výpočet pohybu tekutiny vypadá následovně [5]:

$$\rho\left(\frac{\partial v}{\partial t} + v \bigtriangledown v\right) = - \bigtriangledown p + \bigtriangledown \top + f \tag{2.3}$$

Kde **v** je rychlost toku, ρ je hustota kapaliny, **p** představuje tlak, $\bigtriangledown$ diferenciální operátor, $\top$ tensor a **f** sílu působící na kapalinu.

Pro každý specifický problém musí být NSE upraveny. U aplikací, které mohou počítat s určitou nepřesností, jako například v počítačové grafice, se problém a výpočty aproximují.

2.2.2 Použitelnost NSE

Navier-Stokes rovnice jsou *nelineární parciálně diferenciální rovnice*. Nelinearita způsobuje, že většina problémů je obtížně řešitelná. Nelinearita je způsobena prouděním tepla a hmoty, neboli *konvekcí*. Pro některé případy (jako jsou velmi pomalé toky) je možné rovnice linearizovat.

Turbulence je jedním z problémů, který stěžuje popis chování kapalin. Jedná se o časově závislé chaotické chování, které se vyskytuje v mnoha kapalinách. Ve většině případů se jeho výskyt a vliv zjišťuje experimentálně. Pomocí NSE je možné turbulence zahrnout do výpočtu ale je to časově velice náročné.

Navier-Stokes rovnice se používají pro výpočet proudění tekutin a plynů, modelování počasí, mořských proudů aj. Z pozorování pak vyplývá, že NSE odpovídají reálnému prostředí. Dokonce i turbulence je možné poměrně věrně simulovat.Protože jsou však velice náročné na výpočet, jejich použití pro reprezentaci vody v počítačové grafice, zvlášť v reálném čase, je nemožné [11].

Kapitola 3

Statistické modely vln

Pro simulaci vlnění vody existují dvě skupiny metod. Takové, které jsou reálnými simulacemi chování kapalin a používají se v hydrodynamice a jiných oborech. Navier-Stokes, popsané v předchozí kapitole, jsou právě touto metodou. Nejsou však vhodné pro použití v počítačové grafice kvůli své náročnosti

Druhou skupinou jsou potom takové, které předchozí metody aproximují, případně nejsou přesnými simulacemi ale vychází ze statistických hodnot.

V některých případech je ve statistických modelech výška vlny náhodnou proměnnou horizontálního bodu v čase. Jerry Tessendorf se ve své práci zabývá metodou, která jako počáteční data používá právě pseudonáhodná čísla, která se použijí pro vygenerování sinusoid. Ty jsou následně rychlou Fourierovou transformací sečteny [9].

3.1 Popis šíření vlny

Pro popis šíření vlny je možné použít několik známých vztahů mezi frekvencí a velikostí pro vektory vln k_i, v závislosti na hloubce vodní masy. Pro hlubokou vodu, kde můžeme ignorovat dno, má tento vztah následující tvar:

$$w^2(k) = gk \tag{3.1}$$

kde g je konstanta gravitačního zrychlení. Tento vztah platí jak pro výše uvedené Gerstnerovy vlny 2.1, tak i pro statistické modely vln.

Pro mělkou vodu, s hloubkou D se používá následující vzorec:

$$w^2(k) = gk \tanh(kD) \tag{3.2}$$

Popis šíření vlny se využívá později při generování výškového pole v závislosti na čase.

3.2 Sinusoidy

Jak už jsem uvedl výše, statistické modely vln jsou reprezentovány sinusoidy, které se následně pomocí FFT sečtou. Reprezentace výškového pole vlny je pak výška vlny $h = (\mathbf{x}, t)$ v horizontálním bodě $\mathbf{x} = (x, z)$ jako suma sinusoid s komplexními, časově závislými amplitudami:

$$h(\mathbf{x}, t) = \sum_k \tilde{h}(\mathbf{k}, t) exp(i\mathbf{k} \cdot \mathbf{x}) \tag{3.3}$$

Kde t je čas, $\mathbf{k}$ je vektor se složkami $\mathbf{k} = (k_x, k_z)$, $k_x = 2\pi n/L_x$, $k_z = 2\pi m/L_z$ a n a m jsou celá čísla z rozsahu $-N/2 \leq n < N/2$ a $-M/2 \leq m < M/2$. Hodnoty N a M se volí z rozsahu od 16 do 2048 jako mocnina dvou, aby bylo možné sumu vypočítat pomocí FFT. Význam výrazu $\tilde{h}(\mathbf{k}, t)$ vysvětlím později.

Oceánografický výzkum založený na statistické analýze fotografií a radarových snímků prokázal rovnici 3.3 jako vhodnou pro reprezentaci vln na otevřeném moři. Stejně tak bylo prokázáno, že pro generování komplexních amplitud $\tilde{h}(\mathbf{k}, t)$ je vhodné použít *Phillipsovo spektrum.*

3.3 Phillipsovo spektrum

Phillipsovo spektrum je model vln hnaných větrem v otevřeném moři [12]. Vyhází ze statistické analýzy fotografií a radarových snímků a měření na otevřeném moří:

$$P_h(k) = A\frac{exp(-1/(kL)^2)}{k^4}\left|\hat{\mathbf{k}} \cdot \hat{w}\right|^2 \tag{3.4}$$

V tomto vzorci $L = V^2/g$ představuje největší možnou vlnu, kterou může vítr o síle V vytvořit. g představuje gravitační konstantu, $\hat{w}$ pak směr větru. A je numerická konstanta a faktor kosinu $\left|\hat{\mathbf{k}} \cdot \hat{w}\right|^2$ eliminuje vlny, které se pohybují kolmo ke směru větru.

3.4 Generování počátečních hodnot

Phillipsovo spektrum se používá ve vzorci pro generování počátečních hodnot komplexních amplitud $\tilde{h_0}(\mathbf{k})$:

$$\tilde{h_0}(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(k)} \tag{3.5}$$

kde ξ_r, ξ_i jsou hodnoty z gaussovského generátoru náhodných čísel. Je možné trochu experimentovat a zkusit použít jiné generátory a tím získat různou charakteristiku vln.

3.5 Generování výškového pole vln

Použitím popisu šíření vln a vygenerovaných počátečních hodnot vygeneruji výškové pole amplitud vln podle následujícího vzorce:

$$\tilde{h}(\mathbf{k}, t) = \tilde{h_0}(\mathbf{k})exp\{i\omega(k)t\} + \tilde{h_0}^*(-\mathbf{k})exp\{-i\omega(k)t\} \tag{3.6}$$

Toto výškové pole určuje strukturu vln. Pro úplnost ještě zopakuji vzorec 3.3, ve kterém se používá výsledek předchozí rovnice.

$$h(\mathbf{x}, t) = \sum_k \tilde{h}(\mathbf{k}, t)exp(i\mathbf{k} \cdot \mathbf{x})$$

Výhodou celého tohoto postupu je fakt, že k výpočtu nepotřebujeme znát a uchovávat hodnoty vypočítané v předchozích krocích. Stačí nám pouze jednou vygenerované počáteční hodnoty a poté se již jen vypočítá výšková mapa jako funkce času.

Kapitola 4

Teoretické předpoklady pro implementaci

Během studia teorie simulace vlnění vody jsem prostudoval spoustu různých metod, ty nejdůležitější z nich jsem uvedl v předchozích kapitolách. Jako nejvhodnější jsem si vybral statistický model vln, který ve své práci popisuje Jerry Tessendorf [9] a kterou jsem blíže rozebral v kapitole 3.
V této kapitole proberu způsob reprezentace osvětlení, možnosti využití programovatelného grafického řetězce a také transformace matic, důležité pro práci s kamerou.

4.1 Výpočet vln

Pro generování počátečních hodnot jsem tedy použil *Phillipsovo spektrum* 3.4. Jako zdroj náhodných dat jsem použil gaussovský generátor náhodných čísel [7] se střední hodnotou 0 a standardní odchylkou 1.

Pro popis šíření vln jsem použil vzorec popisující vlny na otevřeném moři, kde je možné hloubku vody zanedbat 3.1.

Jedním z důvodů, proč jsem si vybral tuto metodu, je ten, že je možné vypočítat výslednou hodnotu 3.3 rychle pomocí rychlé Fourierovy transformace. Celá metoda je navržena s ohledem na to, že výpočty je potřeba provádět v reálném čase. Proto také není přesná z hlediska fyzikálního, ale přesto poskytuje přesvědčivé výsledky.

Postup, který je tedy potřeba pro vygenerování výškové mapy, je následující:

- Pomocí gaussovského generátoru náhodných čísel a Phillipsova spektra vygenerovat počáteční hodnoty.
- Vypočítat amplitudy v závislosti na čase.
- Vypočítat konečnou výškovou mapu bodu v prostoru a čase.

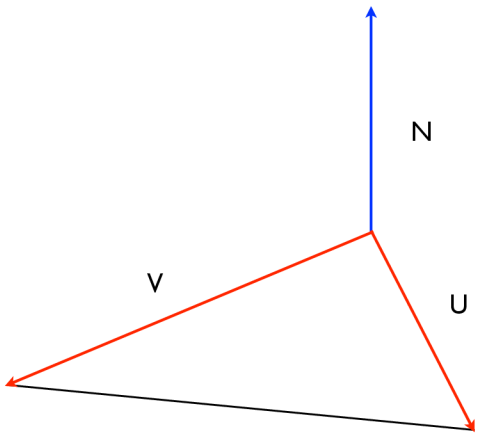
4.2 Reprezentace vln

Vlny jsou reprezentovány jako výšková mapa, proto je tak i vykresluji. Protože je třeba vykreslovat vlny co nejefektivněji a neplýtvat výpočetním výkonem procesoru a grafické karty, zobrazuji vlnu jak onu vypočítanou výškovou mapu. Některé operace je potřeba provést pouze jednou, proto je provádím při inicializaci, jiné se provádějí při každém překreslení

obrazovky, více o tomto v kapitole 5.
V této podkapitole rozeberu především teorii výpočtu osvětlení, konkrétně výpočet *Phongova světelného modelu* a *cube mapping*, použitý pro výpočet odlesků.

4.2.1 Výpočet normál

Vypočítat normály vrcholů je možné dvěma způsoby.
V prvním z nich se vypočítá normála pro každý trojúhelník. Jedná se o normálu zvanou `surface normal`. V druhém se použijí normály trojúhelníků pro výpočet normály společného vrcholu jejich zprůměrováním.
Rozhodl jsem se použít první způsob, protože poskytuje dostatečné výsledky a nezatěžuje tolik procesor.
Pro výpočet normály trojúhelníku je potřeba provést vektorový součin dvou vektorů tvořících hrany daného trojúhelníku. Pro trojúhelník na obrázku 4.1 tedy platí, že jeho normála $N = U \times V = (u_2v_3 - u_3v_2, u_3v_1 - u_1v_3, u_1v_2 - u_2v_1)$



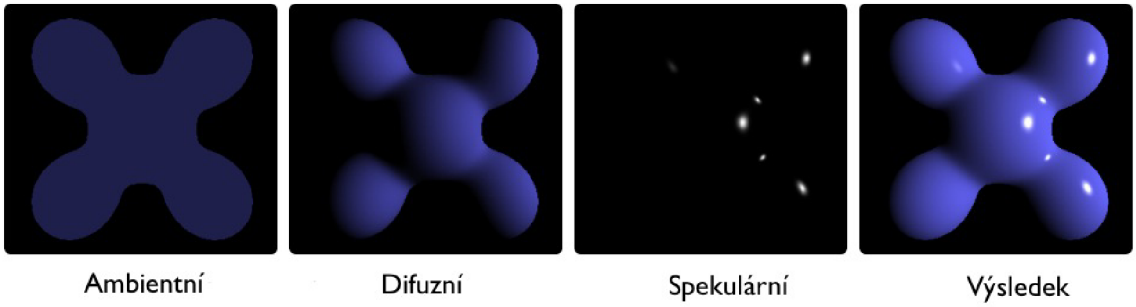
Obrázek 4.1: Výpočet normály trojúhelníku

4.2.2 Světelný model

Pro výpočet osvětlení používám modifikovaný `Phongův světelný model` [4], [14].
Jedná se o výpočet barvy každého pixelu povrchu předmětu, který je velice efektivní a lze jej provádět jednoduše v grafické kartě ve `fragment shaderu`, kde jsou normály jednotlivých vrcholů interpolovány pro každý pixel.
Vlastnosti materiálu povrchu předmětu nebo světla se popisují pomocí tří složek:

- Ambientní - rozptýlené světlo, reprezentuje konstantní barvu povrchu.
- Difuzní - odražené světlo podle normály povrchu, dodává předmětu hloubku.
- Spekulární - odlesk předmětu na nejvíce lesklých místech.

Pro lepší představu přikládám obrázek 4.2.



Obrázek 4.2: Phongův světelný model (Autor: Brad Smith, *llustration of the Phong Reflection Model.* 2006.)

Barva pixelu je dána vztahem:

$$I_p = k_a i_a + \sum_{m=1}^{lights} (k_d(L_m \cdot N) i_{m,d} + k_s(R_m \cdot V)^{\alpha} i_{m,s}) \tag{4.1}$$

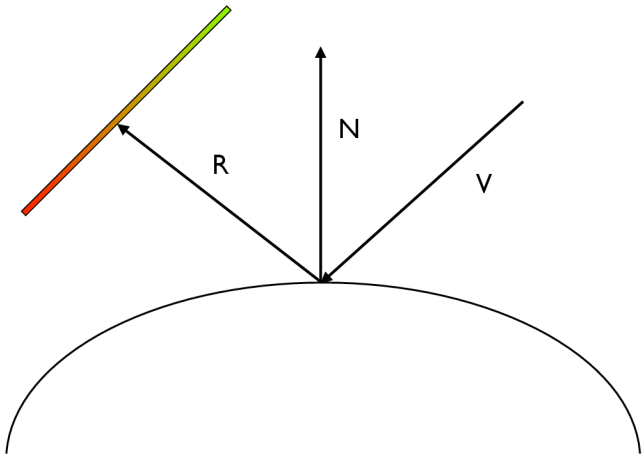
- i_m reprezentuje jedno ze světel, $i_{m,s}$ jeho spekulární, $i_{m,a}$ ambientní a $i_{m,d}$ difuzní složku.
- k_s spekulární, k_a ambientní a k_d difuzní složku materiálu.
- α je konstanta udávající odrazivost materiálu. Čím je větší, tím je materiál lesklejší a naopak.
- N je normála a V je vektor směřující od pixelu k pozorovateli.
- L_m je vektor směřující od pixelu ke zdroji světla m.
- R_m je vektor vypočítán jako odraz vektoru $-L_m$ vztahem $R_m = 2(L_m \cdot N)N - L_m$.

Výsledná barva pixelů tedy reprezentuje součet ambientní složky materiálu a sumy odrazů všech světel. Ta je dána vlastnostmi materiálu, jeho normálami, vlastnostmi světla a jeho polohou.

Pro své potřeby ještě přičítám k vypočítané barvě barvu pozadí. Tím dosáhnu jednoduchého efektu odrazu. Princip je znázorněn na obrázku 4.3 a je velice jednoduchý. Vypočítám odraz vektoru v, který vedu z kamery na požadovaný pixel, podle normály N, čímž získám vektor r, pomocí kterého získám barvu pixelu z textury reprezentující pozadí scény.

Výsledný vektor je dán vztahem:

$$r = 2(v \cdot N)N - v \tag{4.2}$$



Obrázek 4.3: Odraz vektoru podle normály

4.3 Programovatelný grafický řetězec

Programovatelný grafický řetězec je způsob, jak v moderních grafických kartách modifikovat data, která prochází *grafickou pipeline* [6]. Programy, které tyto data modifikují, se nazývají *shadery*. Existují tři základní druhy shaderů, každý pracuje s jiným druhem primitiv:

- `Vertex shader` - tento program je vykonán pro každý vrchol. Je možné měnit barvu, pozici nebo pozici textury daného vrcholu. Není však možné odebírat nebo vytvářet nové vrcholy. Při vykonávání vertex shaderu pro daný vrchol není možné přistupovat k ostatním vrcholům, jejich vlastnostem ani není možné je modifikovat. Po zpracování jsou vrcholy předány k zpracování geometry shaderu, případně rovnou k rasterizaci a následně fragment shaderu.

- `Geometry shader` - tento program umožňuje odebírat nebo přidávat nové vrcholy. Pomocí něj je možné modifikovat detaily objektu, které by byly příliš náročné pro výpočet na CPU. Ideální využití geometry shaderu je pro naprogramování Level-Of-Detail.

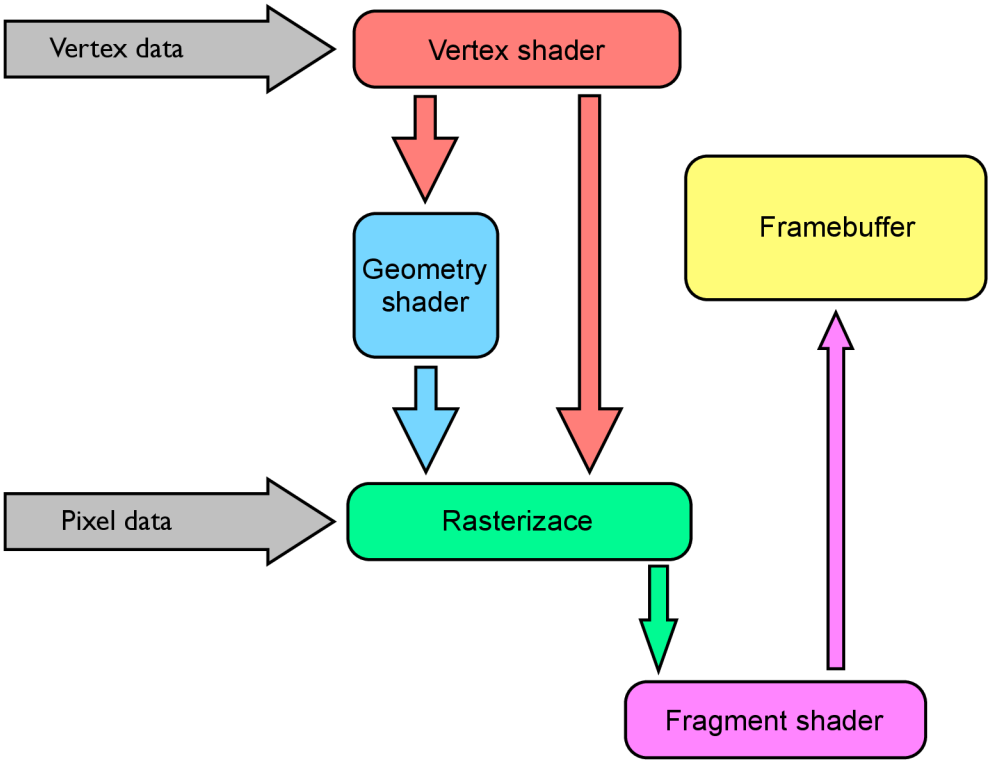
- `Fragment shader` - také nazývaný pixel shader pracuje s jednotlivými body scény po rasterizaci a umožňuje měnit jejich barvu nebo pracovat s texturami. Vstupem jsou mu jednotlivé pixely rasterizované z primitiv pozměněných pomocí vertex, případně geometry shaderu. Typickým příkladem použití fragment shaderu je výpočet osvětlení nad každým fragmentem, neboli *per fragment lighting*. Takovým druhem osvětlení je například Phongův osvětlovací model.

Obecně by se měly shadery skládat pouze s několika instrukcí, aby nebyly příliš náročné. Provádějí se totiž pro každý vertex, případně fragment. Každá optimalizace se tedy projeví skokovým nárůstem výkonu.
Program pro vertex shader může obsahovat i několik stovek příkazů. Program pro fragment shader by jich měl obsahovat maximálně několik desítek, protože se provádí několikanásobně častěji, pro každý fragment.

Postup při zpracování dat pomocí programovatelného grafického řetězce je zjednodušeně znázorněn na obrázku 4.4 a dal by se rozdělit do následujících částí:

- Poslání vertex dat grafické kartě - vertex shaderu jsou poslány vrcholy, neboli `vertex data`, pro zpracování.
- Vertex shader - provedení programu vertex shaderu pro každý vrchol. Provádí se změna souřadnic, barvy vrcholu, souřadnic pro textury.
- `Primitive assembly` - jedná se o transformaci dat vertex shaderu na primitiva vhodná pro další zpracování. Jedná se o čáru, bod a trojúhelník. Pokud je tedy do vertex shaderu poslán čtverec, jsou z něj vytvořeny dva trojúhelníky.
- Geometry shader - pokud není použitý, provádí se rovnou rasterizace. Geometry shader přidává nebo odebírá vrcholy. Tím je možné měnit strukturu objektu.
- Rasterizace - stručně řečeno se jedná o převedení vektorových dat na rastrová. Při rasterizaci jsou interpolována data, jako jsou normály a barva.
- Poslání pixel dat grafické kartě - fragment shaderu jsou poslána `pixel data`, například textury nebo `cubemap` textury.
- Fragment shader - provedení fragment shader programu pro každý z pixelů. Fragment shader je zvláště vhodný pro realizaci výpočtu osvětlení nad pixely. V mém případě jsem v něm implementoval upravený Phongův světelný model.
- Per fragment testy - před odesláním k vykreslení jsou ještě jednotlivé fragmenty z důvodu optimalizace vykreslování testovány. Jedná se o `stencil test`, kdy jsou fragmenty testovány na přítomnost v zorném úhlu kamery a `early depth-test`, při kterém jsou fragmenty testovány na viditelnost, jestli nejsou překryty jinými fragmenty. Všechny tyto testy je možné provést a konfigurovat i programově, a tím docílit menší zátěže grafické karty.
- Framebuffer - odeslání dat do `framebufferu`, který se postará o vykreslení.

Důvod, proč jsou vertex shader a fragment shader operace tak rychlé, je ten, že se dají velice dobře paralelizovat. Moderní grafické karty obsahují několik vertexovacích a fragmentovacích jednotek, které dokáží v jednom cyklu provádět operace nad mnoha vertexy či fragmenty současně. To je však také důvod, proč není možné ve vertex shader programu přistupovat k jinému vertexu, než je ten aktuálně zpracovávaný. To platí i pro fragment shader.



Obrázek 4.4: Programovatelný grafický řetězec

Pro použití v kombinaci s Open GL se používá shaderovací jazyk *GLSL*. Jeho syntaxe je velice podobná jazyku C. Umožňuje například provádět cykly nebo vyhodnocovat podmínky. Verze pro mobilní zařízení obsahuje několik omezení, například je potřeba každou proměnnou deklarovat se zvolenou přesností.

Krátce shrnu několik vlastností shaderovacího jazyk GLSL ve verzi 4.0 [10]:

- Standardní datové typy - `void`, `int`, `uint`, `bool`, `float`, `double`.
- Vektorové datové typy - `vec2`, `vec3`, `vec4` pro vektory s čísly float, analogicky `dvec2`, `dvec3`, `dvec4` pro double, `uvec2`, `uvec3`, `uvec4` pro integer bez znaménka, `ivec2`, `ivec3`, `ivec4` pro znaménkový, `bvec2`, `bvec3`, `bvec4` pro vektory booleovských hodnot.
- Datové typy pro matice - `mat2`, `mat3`, `mat4` pro matice s čísly float.
- Datové typy pro textury - `sampler2D` pro texturu, `samplerCube` pro texturu namapovanou na krychli.
- Základní cykly jako `while` nebo `for`, vyhodnocování podmínek pomocí `if-else`.
- Vestavěné proměnné - popisují daný vertex nebo fragment, jako jsou `gl_Position`, `gl_TexCoord`, `gl_FrontColor`, `gl_BackColor`, `gl_FragColor`, takové, které popisují matice, jako `gl_ModelViewMatrix`, `gl_ProjectionMatrix` a jiné.

- Vestavěné funkce - pro práci s vektory, maticemi. Dále funkce pro výpočet odrazu nebo lomu paprsku a jiné.

Při programování shaderů se používají tři druhy parametrů pro posílání dat do shaderu a předávání si dat mezi vertex a fragment shadery. Jsou to `attribute`, `uniform` a `varying`:

- Typ `attribute` se používá pro předání dat vertex shaderu. Pomocí tohoto druhu parametru se předávají souřadnice vertexů, jejich normály nebo texturovací souřadnice. Vertex shader program se poté vykoná pro každý element.
- Typ `uniform` se používá pro předání například nastavení osvětlení nebo jiných parametrů a může být přítomen jak ve fragment tak i vertex shader programu. Hodnota parametru uniform je uchována, dokud není znovu přepsána, je tedy možné ji nastavit pouze jednou při inicializaci nebo při každém překreslení.
- Typ `varying` slouží pro předávání dat mezi vertex a fragment shader programem. Používám jej pro předání normál vertexů. Data takto předaná jsou interpolována z vertexů na fragmenty. Ve fragment shader jsou data určena pouze ke čtení, ve vertex shaderu je možné do nich zapisovat i číst. Pokud nejsou do proměnné typu varying uložena před čtením data, je jejich obsah nedefinován.

Parametr si tedy nadefinuji například takto: `attribute vec4 position;`.
Způsob, jakým se předávají parametry shaderu, záleží na tom, jestli se jedná o `attribute` nebo `uniform`. U attribute se používá funkce `glVertexAttribPointer` spolu s funkcí `glDrawElements` nebo `glDrawArrays`. U uniform potom `glUniformXY`, kde X určuje velikost vectoru a Y jeho datový typ, tedy například `glUniform3f`, u matic potom obdobně `glUniformMatrix4fv` pro matici o rozměrech 4x4 a datovém typu float.

Shaderovací jazyk pro mobilní zařízení GLSL ES ve verzi 1.0 je odlišný v několika věcech:

- Není možné použít vestavěné proměnné s maticemi, které určují pohled na scénu, jako jsou gl_ModelViewMatrix nebo gl_ProjectionMatrix. Je potřeba si je vypočítat na straně procesoru a následně je poslat grafické kartě.
- Deklarace proměnných vyžaduje určení přesnosti pomocí klíčových slov `lowp`, `mediump` a `highp`
- Vertex data není možné vykreslovat jako čtyřúhelníky nebo polygony. Je potřeba je vykreslovat pomocí trojúhelníků.

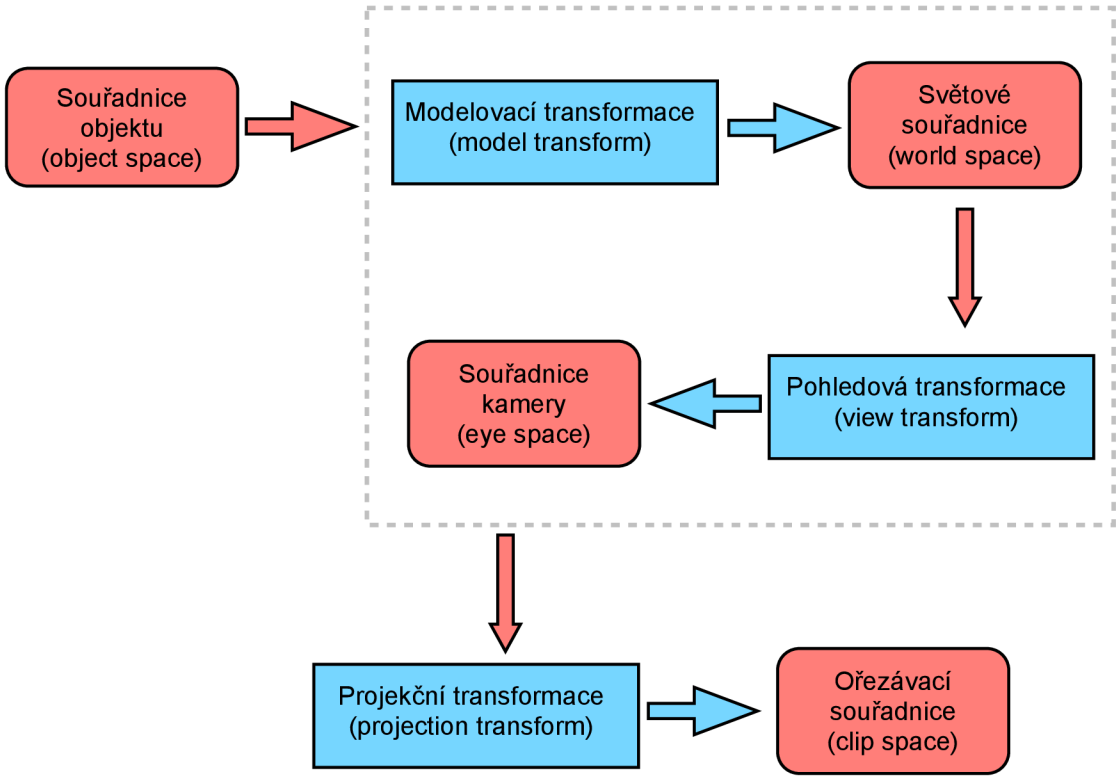
V zásadě však mezi standardní verzí GLSL a verzí pro mobilní zařízení není velký rozdíl. Není sice možné používat shodné programy, ale je velice snadné přepsat je z jedné verze na druhou.

4.4 Maticové transformace

Knihovna Open GL pro mobilní zařízení verze 2.0 nepodporuje standardní transformační funkce, jako jsou `gl_Scale` nebo `gl_Rotate`. Proto jsem byl nucen implementovat jejich ekvivalenty. Při vykreslování jsou prováděny transformace v následujícím pořadí, jak je vidět na obrázku 4.5:

- Model transform - transformací souřadnic modelu, tedy objektu, změníme jeho pozici ve scéně.
- View transform - transformací pohledu umístíme do scény kameru a nasměrujeme ji požadovaným směrem.
- Projection transform - projekční transformací určíme úhel záběru, nejbližší a nejvzdálenější plochu vykreslování.

Pro implementaci této funkcionality bylo potřeba vyřešit operaci s maticemi násobení, výpočet projekční matice, výpočet pohledové matice a výpočet modelové matice, dále také operace s vektory.



Obrázek 4.5: Transformace souřadnic

4.4.1 Modelová matice

Pro výpočet modelové matice je potřeba provést dílčí transformace nad jednotkovou maticí. Jsou to posun, rotace a změna měřítka. Matice vzniklé těmito transformacemi se mezi sebou násobí, čímž dosáhnu výsledné modelové matice, přičemž záleží na pořadí násobení.
Posun modelu je možné provést pomocí jednotkové matice, kde v posledním řádku jsou

uvedeny hodnoty posunu v osách x, y, z:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{pmatrix} \tag{4.3}$$

Změnu měřítka modelu je možné provést pomocí jednotkové matice, které změním hodnoty na diagonále na poměr, o který se má měřítko změnit v jednotlivých osách:

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

Rotace modelu je trochu složitější, protože záleží na ose, podle které chci objekt rotovat. Ve všech případech využiji sinus a cosinus úhlu, o který budu rotovat, $s = \sin(r)$, $c = \cos(r)$. Rotace kolem osy **x**:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.5}$$

Rotace kolem osy **y**:

$$\begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.6}$$

Rotace kolem osy **z**:

$$\begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.7}$$

Násobením těchto dílčích matic umístím a natočím model na požadované místo ve scéně. Výsledkem násobení je zmiňovaná **modelová matice**.

4.4.2 Pohledová matice

Pomocí pohledové matice určím polohu a směr natočení kamery. Výsledná matice vypadá takto:

$$\begin{pmatrix} x.v_1 & y.v_1 & z.v_1 & 0 \\ x.v_2 & y.v_2 & z.v_2 & 0 \\ x.v_3 & y.v_3 & z.v_3 & 0 \\ -(x \cdot eye) & -(y \cdot eye) & -(z \cdot eye) & 1 \end{pmatrix} \tag{4.8}$$

Kde **eye** je poloha kamery, **target** je souřadnice, na kterou je kamera natočena a **up** je určení orientace horní hrany kamery, který může nabývat hodnot $(1, 0, 0)$, $(0, 1, 0)$ nebo $(0, 0, 1)$. Vektory **x**, **y** a **z** se vypočtou ze vztahů:

$$z = (eye - \hat{}target) \tag{4.9}$$

$$x = (up \hat{\times} z) \tag{4.10}$$

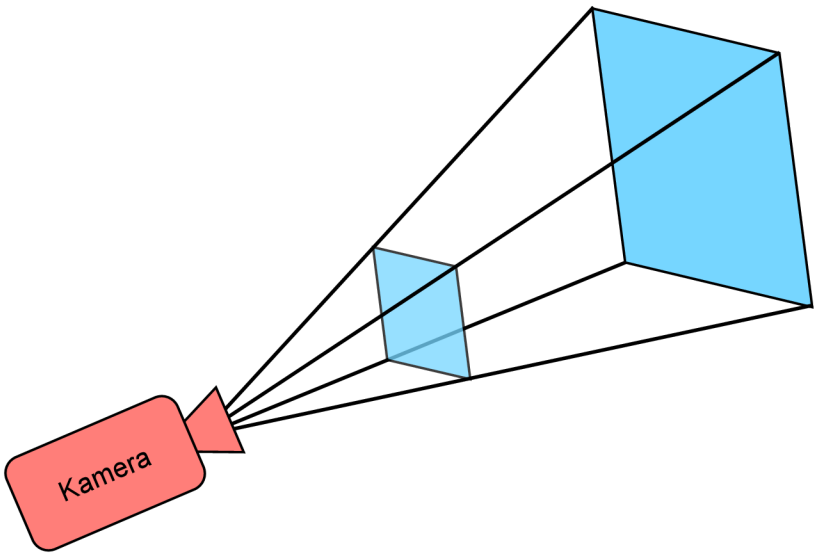
$$y = (z \hat{\times} x) \tag{4.11}$$

Pro jistotu doplním, že všechny vektory jsou normalizované.

4.4.3 Projekční matice

Projekční matice se používá pro nastavení pohledu kamery, tedy úhlu záběru a nejbližší a nejvzdálenější roviny vykreslení, jak je vidět na obrázku 4.6. K výpočtu potřebujeme znát **z** souřadnice označující vzdálenou a blízkou ořezávací rovinu **f** a **n**, dále **y** souřadnici definující horní a dolní hranu blízké ořezávací roviny **t** a **b** a také **x** souřadnici pro její levou a pravou hranu **l** a **r**. Výsledná matice je potom tohoto tvaru:

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.12}$$



Obrázek 4.6: Projekce pohledu kamery

Výsledné transformace se poté provádí ve vertex shaderu postupem naznačeným na obrázku 4.5:

- Z pohybu, rotace nebo ze změny měřítka objektu vypočítám modelovou matici.
- Z pozice kamery a její orientace vypočítám pohledovou matici. Tuto matici je kvůli optimalizaci dobré přepočítávat pouze při pohybu nebo natočení kamery.

- Vynásobím modelovou matici pohledovou, čímž získám `modelview` matici.
- Projekční matici stačí vypočítat pouze jednou, při inicializaci.
- Pozici vertexu vynásobím nejdříve modelview a následně projekční maticí.

Tímto postupem získám výslednou pozici vertexu.

Kapitola 5

Implementace

V této kapitole blíže popíšu vlastní implementaci. Nejdříve krátce rozeberu specifika programování pod operačním systémem Mac OS X a iOS, způsob jakým jsem rozložil výpočet mezi grafickou kartu a procesor, popíšu vlastní implementaci a nakonec zmíním i vliv na rychlost výpočtu při použití systémových funkcí pro výpočet FFT a již zmíněného rozložení výpočtu mezi grafickou kartu a procesor.
Aplikace běží na operačních systémech Mac OS X 10.6.x a iOS 4.3.x.

5.1 Specifika programování pod Mac OS X

Operační systém Mac OS X obsahuje nativní podporu Open GL rozhraní. Mimo jiné obsahuje nástroje pro profilování grafických aplikací nebo ladění shaderů. Hlavním programovacím jazykem pro vývoj na tomto operačním systému je *Objective-C*, které je také programovacím jazykem pro vývoj na platformě iOS. Je sice možné programovat i v C++, ale spousta frameworků a knihoven není pro tento jazyk dostupná. Je však možné kombinovat zdrojové kódy v jazyce Objective-C a C++.

Jazyk Objective-C se vyvíjel paralelně s C++ jako objektová nadstavba jazyka C. Jedná se o nadmnožinu jazyka C. Obsahuje všechny obvyklé části objektových jazyků jako jsou třídy (realizované pomocí interface a implementace), protokoly, dynamické typování nebo kategorie. Volání metody instancí třídy probíhá, podobně jako v jazyce SmallTalk, kterým se tvůrci inspirovali, zasíláním zpráv příjemci, což může být zpočátku matoucí.

Pro operační systém Mac OS X existuje nástavba Open GL zvaná GLUT. Protože jsem však nechtěl přijít o správu některých systémových událostí, implementoval jsem aplikaci, která je schopná pracovat v režimu celé obrazovky, bez použití knihovny GLUT pomocí prostředků dostupných v systému. Ve zkratce se jedná o následující postup:

- Vytvoření podtřídy třídy `NSOpenGLView` a přidání její instance do aplikace.
- Vytvoření dvou instancí třídy `NSWindow`, jedna se stará o zobrazení v okně, druhá v režimu celé obrazovky.
- Přepínání mezi těmito dvěma okny a nastavení obsahu okna na podtřídu třídy `NSOpenGLView` vytvořené v prvním kroku.

Poté je již možné vykreslovat do obsahu okna pomocí standardních funkcí Open GL nebo pomocí programovatelného grafického řetězce a následně překreslit obsah okna získáním jeho Open GL kontextu a "vyprázdněním"paměti takto: `[[self Open GLContext] flushBuffer];`.

5.2 Specifika programování pod iOS

Programování pod tímto operačním systémem má několik specifik. Proberu především způsob vykreslování, omezené systémové zdroje a ovládání systému.
Vykreslování na obrazovku probíhá podobně jako v operačním systému Mac OS X. Aplikace obsahuje instanci třídy `UIApplicationDelegate`, ve které se odchytávají hlavní události aplikace, například její zapnutí a vypnutí. Při její inicializaci je vytvořena a inicializována instance třídy `UIViewController`, která se stará o odchytávání dalších systémových událostí, jako je ovládání dotykem a jiné. V ní jsou také vykreslována data. Pro účely vykreslování je vytvořena instance třídy `UIView`, která obsahuje `renderbuffer` a `framebuffer`. Celé téma je obsáhlé a přesahuje záběr této práce, více informací je možné nalézt v příslušné literatuře [13].
Zjednodušeně řečeno vykreslování probíhá standardními funkcemi nad objektem, který je standardně přítomen v systému. Pro úplnost dodám, že jednotlivé typy zařízení mají různá rozlišení obrazovky, s čímž je potřeba počítat při inicializaci a vlastní implementaci.
Mobilní zařízení mají omezené systémové zdroje, se kterými je potřeba hospodárně nakládat. Je důležité vyvarovat se příliš častého vykreslování, zbytečných a opakovaných volání různých funkcí pro vykreslování a plýtvání výkonu procesoru zbytečnými výpočty.

5.2.1 Open GL ES

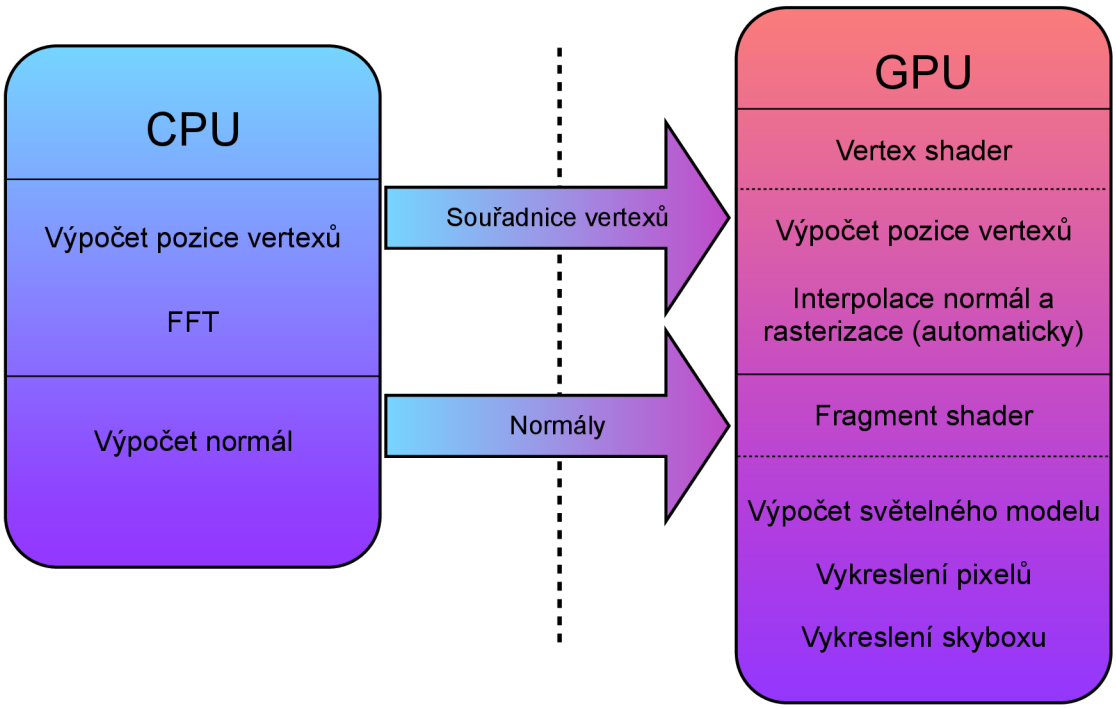
Open GL ES je zjednodušenou verzí grafické knihovny Open GL. Nepoužívá některé nadbytečné funkce, aby byla jednoduše použitelná v hardware mobilních zařízení.
Platforma iOS obsahuje standardní verzi knihovny Open GL ES verze 1.0 a verze 2.0.

5.3 Rozložení výpočtu

Při implementaci jsem byl nucen rozdělit výpočet mezi grafickou kartu a procesor. Dokud jsem nevyužil možností programovatelného grafického řetězce, bylo vykreslování příliš pomalé. Některé dílčí výpočty jsem přesunul do grafické karty také proto, že jsem k tomu byl donucen omezeními platformy Open GL ES 2.0. Ta, mimo jiné, vyžaduje použití programovatelného řetězce vykreslování, tedy použití `vertex` a `fragment shaderu`.
Rozdělení na dvě části se pozitivně projevilo na rychlosti výpočtu především u mobilních zařízení s platformou iOS.
Při inicializaci pošlu do grafické karty síť trojúhelníků, které představují klidnou hladinu. Všechny mají výšku, tedy *y-souřadnici* rovnou nule. Dále při inicializaci pošlu grafické kartě data s parametry osvětlení.
Při každém překreslení obrazovky se grafické kartě neposílá celá výšková mapa vln ale pouze y-souřadnice, které se připočtou ke klidné hladině. Dále posílám vypočtené normály, které se použijí pro výpočet osvětlení.
Na obrázku 5.1 je znázorněno rozdělení výpočtu mezi grafickou kartu a procesor.

5.3.1 CPU

V procesoru se snažím provádět pouze ty výpočty, které není možné provést na grafické kartě nebo by režie výpočtu na grafické kartě byla náročnější, než úspora výkonu získaná tímto řešením.



Obrázek 5.1: Rozložení výkonu mezi CPU a GPU

Jedná se o výpočet výškové mapy vlny, protože systém Mac OS X a iOS obsahují funkce pro výpočet rychlé Fourierovy transformace.
Výpočet normál jednotlivých vrcholů je také potřeba provést na procesoru. Ve `vertex shaderu` by to nebylo možné, protože provádí transformace pouze nad jedním vrcholem a není v něm možné přistupovat k ostatním vrcholům.

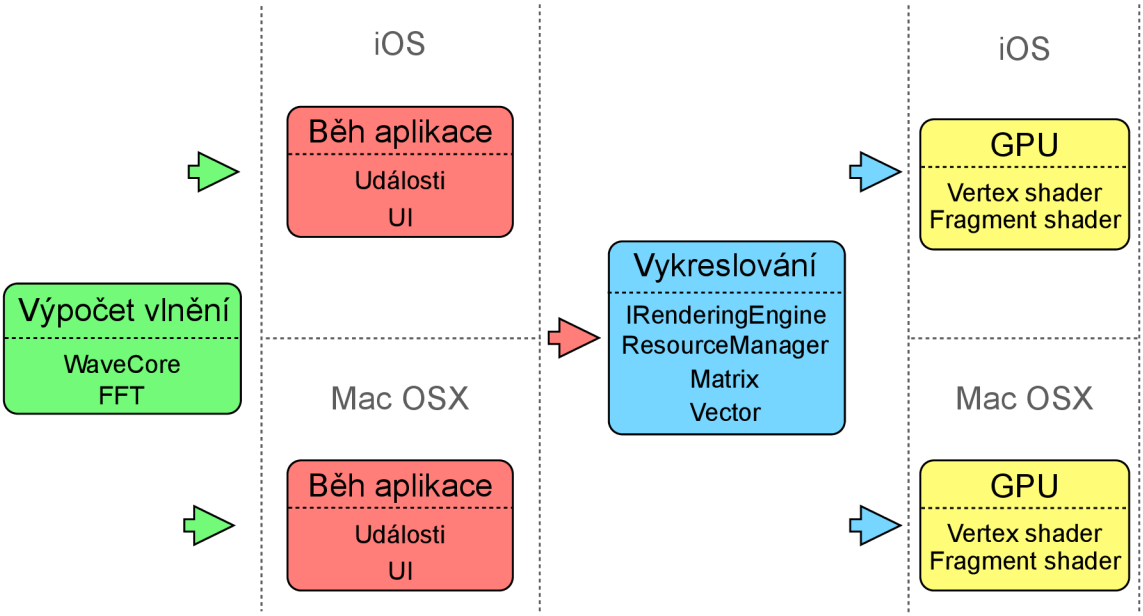
5.3.2 GPU

V grafické kartě se provádí vlastní vykreslování, kdy ve `vertex shaderu` vypočítám souřadnice jednotlivých vrcholů přičtením y-souřadnice k trojúhelníkové síti reprezentující klidnou hladinu. Tato trojúhelníková síť je poslána grafické kartě pouze při inicializaci aplikace. Y-souřadnice jsou posílány grafické kartě při každém překreslení.
Dále v grafické kartě počítám osvětlení za použití normál vypočtených v procesoru pomocí upraveného Phongova modelu osvětlení, kdy k němu přičítám i světlo odražené ze *skyboxu* reprezentujícího pozadí scény.

5.4 Struktura aplikace

Aplikaci jsem navrhoval tak, aby byla jednoduše přenositelná z platformy Mac OS X na platformu iOS. Mezi těmito platformami existuje několik rozdílů, proto jsem byl nucen vy-

tvořit dvě verze aplikace. Některé části jsou však shodné a jsou použity v obou verzích. Na obrázku 5.2 je zobrazeno blokové schéma činnosti aplikace rozdělené do čtyř částí. Je to **výpočet vlnění**, část starající se o **běh aplikace**, **vykreslování** a část pro **vykreslení v grafické kartě**.



Obrázek 5.2: Blokové schéma aplikace

Výpočet vlnění se skládá ze dvou částí, třídy pro výpočet rychlé Fourierovy transformace `FFT` a třídy pro výpočet vlastního vlnění. Obě třídy jsou napsány v jazyce Objective-C a využívají systémových funkcí, které se nacházejí na platformě iOS i Mac OS X. Při implementaci pro jinou platformu, než jsou tyto dvě, by bylo nutné celou tuto část přepsat do jiného programovacího jazyka, nejlépe C++.

Část starající se o **běh aplikace** je odlišná pro obě platformy a stejně tak by byla odlišná pro jakoukoliv jinou platformu, pro kterou by byla aplikace implementovaná. Stará se o zpracování událostí, například překreslení. Dále o zpracování uživatelského vstupu, v případě platformy Mac OS X se jedná o stisky kláves a pohyb myši, v případě iOS pak o obsloužení dotykové vrstvy displeje. A samozřejmě inicializaci celé aplikace a objektů pro vykreslení. V případě Mac OS X se jedná o objekt třídy `NSOpenGLView`, u platformy iOS je to objekt třídy `EAGLView`.

Část starající se o **vykreslování** obsahuje třídy pro reprezentaci a práci s vektory a maticemi a pro práci s texturami a třída starající se o vlastní vykreslení. Třídy pro reprezentaci matic a vektorů implementují nejčastější operace, jako skalární a vektorový součin, součin matic, transformace matic a jiné. Třída pro práci s texturami `ResourceManager` implementuje načítání textur v různých formátech, konkrétně **PNG** a **PVRTC**, formátu specifického pro grafické karty **PowerVR**, kterými jsou osazeny mobilní zařízení s platformou iOS.
Pro účely vykreslování jsem implementoval rozhraní `IRenderingEngine`. Jeho děděním vy-

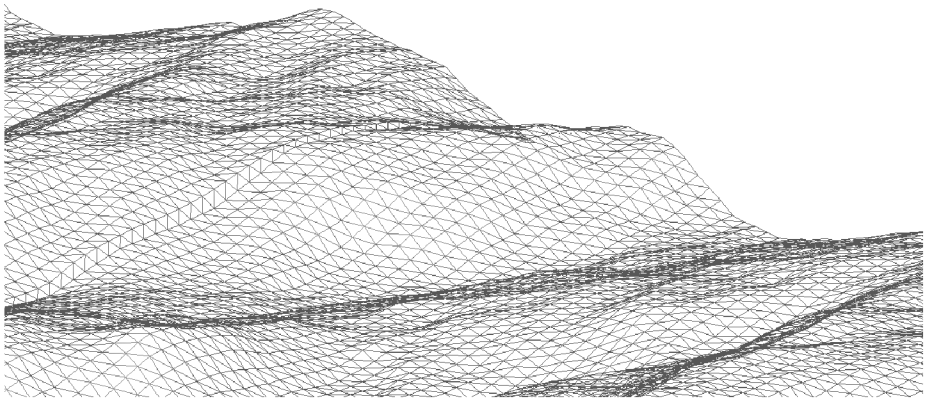
tvořím třídu, kterou je možné použít pro vykreslování. Mohl bych tedy takto implementovat například třídu pro vykreslování pomocí fixního grafického řetězce, a tím bych přenesl aplikaci na starší verze mobilních zařízení s platformou iOS. Nebo bych mohl implementovat třídu pro starší verze Open GL. Pokud bych chtěl přenést aplikaci na zařízení s operačním systémem Windows, mohl bych implementovat třídu využívající **Direct3D** místo knihovny Open GL. Jelikož je Open GL ES 2.0 podmnožinou knihovny Open GL, která se liší pouze v několika drobnostech, mé řešení obsahuje pouze jednu třídu. Ony drobné odlišnosti jsem ošetřil pomocí příkazu pro preprocesor `#ifdef`, `#else`, `#ifndef` a `#endif`.

Poslední logická část aplikace se stará o **vykreslení na grafické kartě**. Jedná se o programy pro vertex a fragment shader. Mezi shaderovacími jazyky GLSL, který je využíván na platformě Mac OS X, a GLSL ES existuje několik rozdílů, stručně popsaných v podkapitole 6.3, proto jsem implementoval dvě verze fragment a vertex shaderů.

5.5 Implementace výpočtu vlnění

Aplikaci jsem navrhoval tak, aby bylo možné některé vstupní parametry modifikovat. Metoda výpočtu vlnění, kterou jsem implementoval, však neumožňuje měnit parametry za běhu, protože se používají pro generování počátečních hodnot. Hodnoty, které je možné modifikovat, jsou rozměry hladiny ve dvou směrech, tento údaj musí být mocnina dvou kvůli výpočtu FFT, použil jsem velikost 128 x 128 hodnot, vektor udávající směr větru a parametr pro škálování výšky vlny.

Takto vypadá vygenerovaná vodní hladina. Velikost v pixelech, tedy jak velký je rozestup mezi dvěma hodnotami v ose `x` a `y`, je možné také škálovat.



Obrázek 5.3: Vodní hladina bez osvětlení

5.6 Implementace FFT

Fourierova transformace je vyjádření časově závislého signálu pomocí harmonických signálů. Slouží pro převod signálů z časové oblasti do oblasti frekvenční. Signál může být buď ve spojitém nebo diskrétním čase. Pokud zpracovávám naměřené hodnoty, tedy znám vzorky signálu či spektra z konečného intervalu, stojím před problémem, jak určit spektrum ze vzorků signálu nebo opačně. K tomuto účelu slouží *diskrétní Fourierova transformace* [1]:

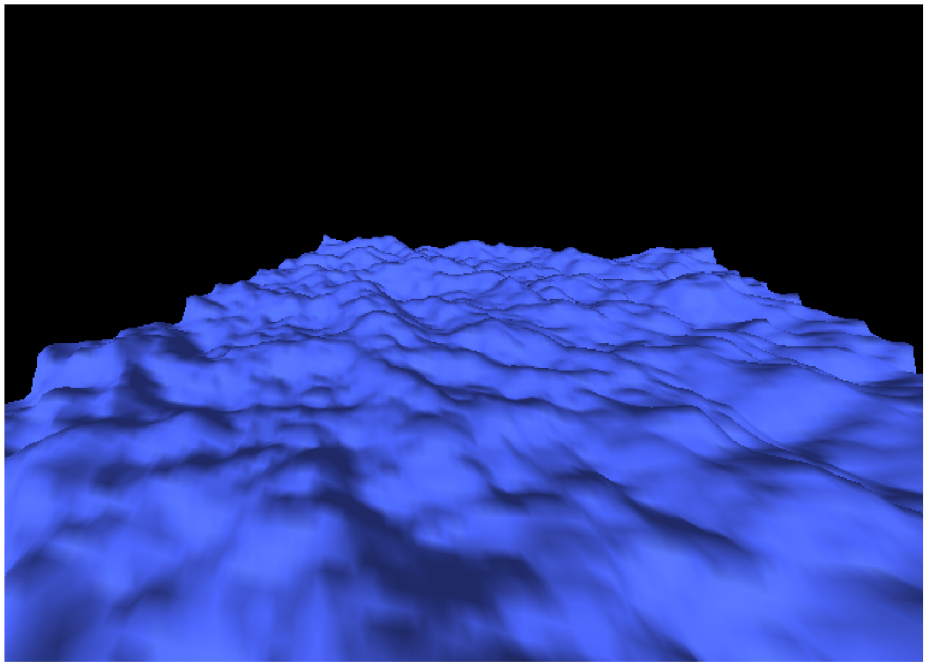
$$D(n) = \sum_{k=0}^{N-1} d(k)e^{-ink2\pi/N}, n = 0, ..., N-1 \tag{5.1}$$

Její inverzní podoba slouží k přesně opačnému účelu:

$$d(k) = \frac{1}{N} \sum_{n=0}^{N-1} D(n)e^{ink2\pi/N}, k = 0, ..., N-1 \tag{5.2}$$

Pro rychlý výpočet diskrétní Fourierovy transformace se používá *rychlá Fourierova transformace*, zkráceně *FFT*. Její princip spočívá v rozdělení výpočtu na menší části, které se postupně vypočítávají. Nejznámějším algoritmem pro výpočet FFT je Cooley-Tukey algoritmus, který popsali pánové James W. Cooley a John W. Tukey[8]. Pokud je potřeba provést FFT pro dvourozměrné pole, provádí se výpočet nejdříve pro řádky a poté pro sloupce.

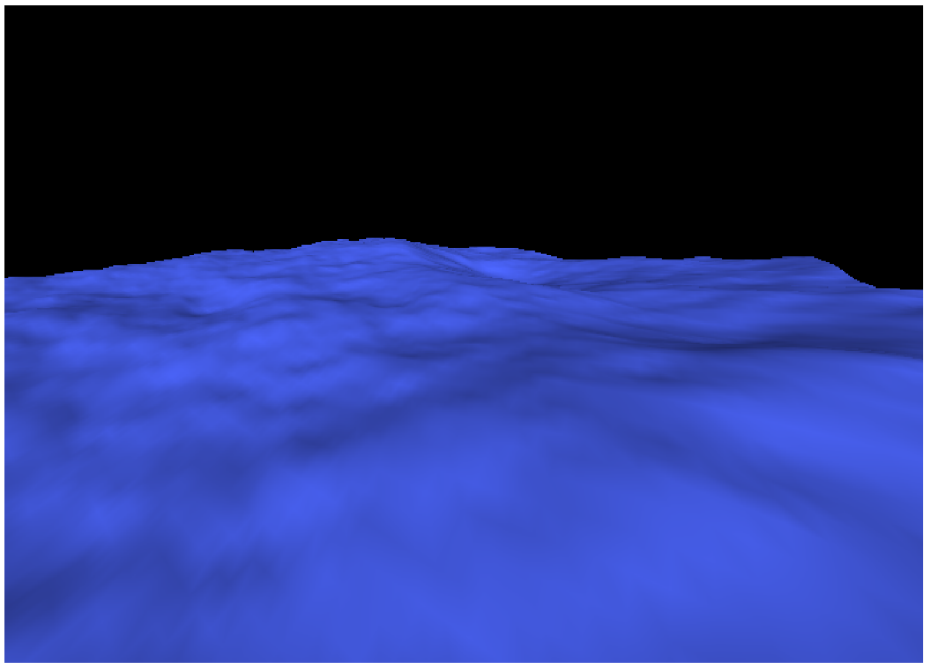
V aplikaci jsem se pokusil implementovat právě Cooley-Tukey algoritmus. Výsledky byly uspokojivé, aplikace dosahovala obnovovací frekvence okolo 30 snímků za sekundu.



Obrázek 5.4: Vlastní implementace FFT

Poté jsem implementoval výpočet FFT pomocí vestavěných funkcí operačního systému, což mělo za následek především zkrácení doby výpočtu na polovinu a dosažení obnovovací

frekvence téměř 60 snímků za sekundu a také jsem dostal jiné výsledky výpočtu. To může být způsobeno mou špatnou implementací algoritmu Cooley-Tukey. Na obrázku 5.5 je vidět implementace při použití systémových funkcí pro výpočet FFT:



Obrázek 5.5: Implementace za použití systémových funkcí pro výpočet FFT

Funkce potřebné pro výpočet FFT jsou obsaženy ve frameworku `Accelerate.framework`, konkrétně v hlavičkovém souboru `vecLib/vDSP.h`. Nejdůležitější jsou funkce `vDSP_create_fftsetup` pro vytvoření struktury s nastavením a `vDSP_fft2d_zip` pro provedení FFT dvourozměrného pole [3]. Je důležité, aby velikost pole byla mocnina dvou, aby se dalo jednoduše dělit na poloviny.

5.6.1 Implementace na iOS

Na platformě iOS je také možné použít framework Accelerate. Pro využití funkcí je však potřeba použít jiný hlavičkový soubor, a to `Accelerate/Accelerate.h`. Zbytek funkcionality je naprosto shodný s implementací na platformě iOS.
Jelikož je výkon mobilních zařízení omezen, je vhodné počítat FFT na menším objemu dat, aby nebyl procesor příliš vytížen. V implementaci pro mobilní zařízení jsem použil velikost hladiny 64 x 64 hodnot.

5.7 Vykreslování vodní hladiny

Vodní hladina je reprezentována jako výšková mapa, proto ji tak i vykresluji. Při inicializaci vytvořím VBO, tedy *Vertex Buffer Object*, do kterého uložím trojúhelníkovou síť, čímž docílím rychlejšího vykreslování. Trojúhelníková síť je uložena v rychlejší paměti a není při každém překreslení posílána grafické kartě. Všechny vrcholy sítě mají nastavenou *y-souřadnici* na nulovou hodnotu. Při každém překreslení vypočítám pouze y-souřadnice,

které následně přepošlu grafické kartě a zpracuji jednoduchým vertex shader programem:

```
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
attribute vec4 position;
attribute vec4 position_y;
vec4 vertexPosition;
void main()
{
    vertexPosition = position;
    vertexPosition.y += position_y.x;
    gl_Position = modelViewMatrix * projectionMatrix * vertexPosition;
}
```

Princip je jednoduchý, k vertexu s nulovou y-souřadnicí přičtu novou výšku a následně vertex transformuji podle pohledu a pozice kamery.
Při rasterizaci jsou jednotlivé primitiva interpolovány na fragmenty a pro každý z nich je spuštěn program fragment shaderu, ve kterém spočítám jejich barvu.

Reprezentovat rozlehlou, z pohledu kamery nekonečnou, vodní hladinu se dá dvěma způsoby. První spočívá ve vykreslení velké vodní plochy, což je velice náročné především na výpočet FFT a především pro mobilní zařízení nerealizovatelné. Druhým je vykreslení plochy malé velikosti a její opakování do libovolné vzdálenosti. Z definice FFT [1] vyplývá, že je periodická. Teoreticky by tedy mělo stačit poskládat více stejných dlaždic vedle sebe. Při implementaci mi bohužel jednotlivé dlaždice na sebe nenavazovaly a byl jsem proto nucen napojit jednotlivé dlaždice pomocí pruhu trojúhelníků. K těmto trojúhelníkům jsem dopočítal i normály. Obrázek 5.6 znázorňuje rozdíl mezi variantou bez (vlevo) a s dopočítanými trojúhelníky. Nepodařilo se mi nalézt chybu v implementaci výpočtu, proto jsem problém musel vyřešit tímto způsobem.
Opakování nekonečné vodní hladiny řeším poměrně jednoduše. V aplikaci je nastavena velikost hladiny v dlaždicích, kterou vykresluji od pozice kamery všemi směry. Při pohybu kamery kontroluji, jestli jsem se posunul o takový úsek, abych na jedné straně řadu dlaždic ubral a na druhé přidal.

5.8 Implementace osvětlení

Aby voda působila co nejrealističtějším dojmem, implementoval jsem upravený Phongův světelný model. Výpočet světelného modelu probíhá ve fragment shaderu na grafické kartě, jedná se o *per fragment osvětlení*, tedy osvětlení počítané zvlášť na každém fragmentu objektu.
Pro výpočet osvětlení je potřeba provést výpočet normál vrcholů. Ten probíhá pro každý trojúhelník. Zvolil jsem si tuto možnost, protože je rychlejší než výpočet pro každý vrchol zvlášť.
V dalších podkapitolách proberu postup implementace a ukázky z kódu shaderů.

Obrázek 5.6: Porovnání vlny bez (vlevo) a s dopočítanými trojúhelníky

5.8.1 Výpočet osvětlení

V podkapitole 4.2.2 jsem teoreticky rozebral Phongův světelný model, který je ve zkratce sumou vlastností světla spolu s vlastnostmi materiálu v závislosti na úhlu pohledu a normále, tedy naklonění, povrchu objektu.
Jednoduchý fragment shader, který počítá Phongovo osvětlení, vypadá následovně:

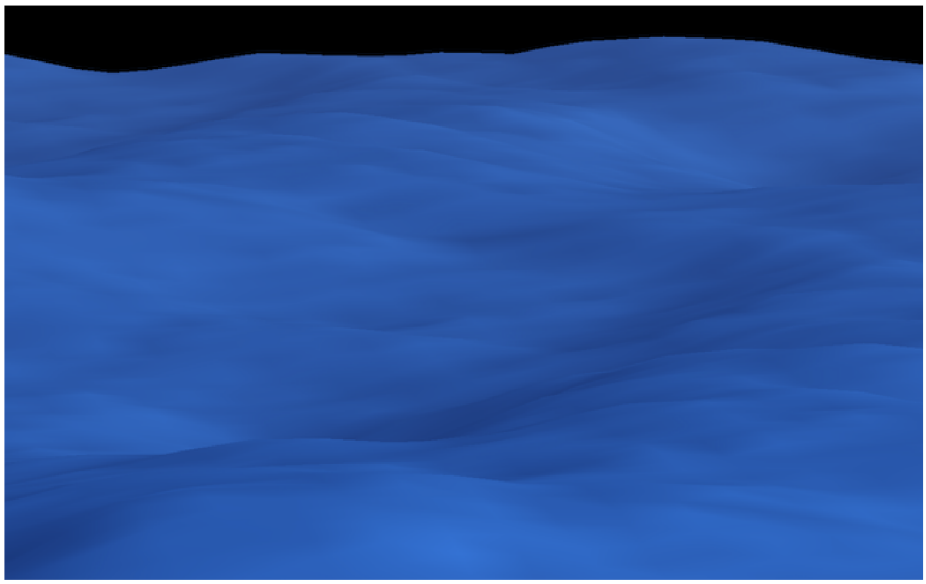
```
uniform vec3 DiffuseMaterial;
uniform vec3 LightPosition;
uniform vec3 AmbientMaterial;
uniform vec3 SpecularMaterial;
uniform float Shininess;
varying vec3 EyespaceNormal;

void main() {
    vec3 N = normalize(EyespaceNormal);
    vec3 L = normalize(LightPosition);
    vec3 E = vec3(0, 0, 1);
    vec3 H = normalize(L + E);
    float df = max(0.0, dot(N, L));
    float sf = max(0.0, dot(N, H));
    sf = pow(sf, Shininess);
    gl_FragColor = AmbientMaterial +
        (df * DiffuseMaterial + sf * SpecularMaterial);
}
```

Výpočet jsem pro mé účely zjednodušil, aby probíhal co nejrychleji. Počítám pouze s jedním zdrojem světla, kterému navíc nenastavuji žádnou vlastnost, jenom polohu. Počítám pouze s odrazivostí materiálu a jeho vlastnostmi. Parametry `AmbientMaterial`,

`DiffuseMaterial`, `SpecularMaterial` a `Shininess` udávají vlastnosti materiálu. Parametr `LightPosition` pozici světla a `EyespaceNormal` je normála povrchu objektu. Funkce `normalize` je vestavěná přímo v GLSL a slouží k normalizaci vektoru. Ve vertex shaderu je pak tato normála vypočítána jednoduše:

```
uniform mat3 normalMatrix;
attribute vec4 normal;
varying vec3 EyespaceNormal;

void main()
{
    EyespaceNormal = normalMatrix * normal.xyz;
}
```

Protože je normála předána fragment shaderu, je interpolována pro každý fragment. Je důležité vynásobit normálu pomocí normálové matice. Tím docílím, že bude normála správně natočena podle pohledu kamery. Pokud by normála nebyla vynásobena, neměnila by se scéna podle natočení kamery.
Normálovou matici získám inverzí transponované matice, kterou získám jako matici 3x3 z modelview matice odstraněním čtvrtého řádku a sloupce. Na obrázku 5.7 je uveden příklad výpočtu Phongova osvětlení.

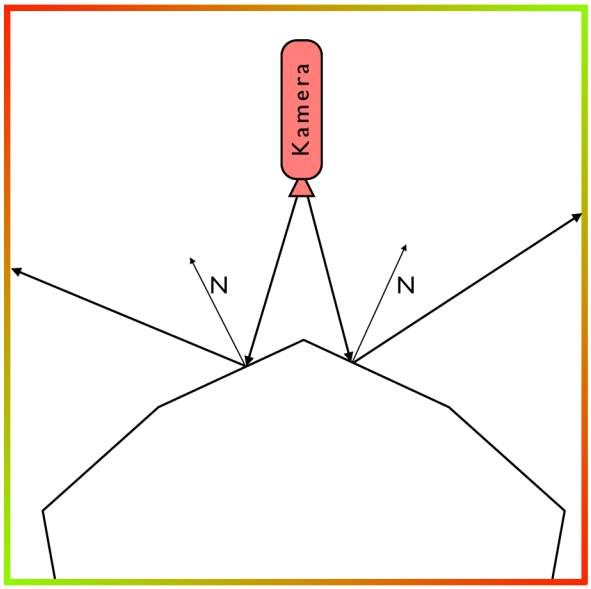


Obrázek 5.7: Osvětlení pomocí zjednodušeného Phongova světelného modelu

5.8.2 Výpočet odrazu světla

Pro vytvoření dojmu lesknoucí se hladiny vody jsem implementoval metodu *cube mapping*. Tato metoda spočívá v tom, že kolem scény vytvořím fiktivní kostku s texturou zvanou `cube map`. Pro každý fragment spočítám odraz vektoru, který vedu z kamery. Z odrazu

získám barvu pixelu textury umístěné na krychli a přičtu ji k barvě vypočítané pomocí Phongova světelného modelu. Pro názornost přikládám obrázek 5.8.



Obrázek 5.8: Výpočet osvětlení pomocí cube map

Pro výpočet odrazu potřebujeme znát pozici kamery, pozici vertexu a z ní je možné vypočítat odraz podle normály. Vertex shader je tedy opět velice jednoduchý:

```
uniform mat3 normalMatrix;
uniform mat3 model;
uniform vec3 EyePosition;
attribute vec4 normal;
varying vec3 ReflectDir;

void main
{
    vec3 EyespaceNormal = normalMatrix * normal.xyz;
    vec3 eyeDir = normalize(gl_Position.xyz - EyePosition);
    ReflectDir = model * reflect(eyeDir, EyespaceNormal);
}
```

Pro zjednodušení předpokládám, že pozice vertexu `gl_Postion` byla nastavena buď pomocí atributu, nebo například pomocí VBO. Na třech řádcích pro každý vertex spočítám odraz vektoru vedeného z kamery `EyePosition` k vertexu pomocí vestavěné funkce `reflect`. Odraz je počítán podle normály `EyespaceNormal` a transformován podle modelové matice `model` tak, aby byl korektně umístěn. Dále je předán ve vektoru `ReflectDir` fragment shaderu k dalšímu zpracování.
Fragment shader pouze namapuje vektor odrazu na texturu, která se má ve vodě lesknout.

```
uniform samplerCube Sampler;
```

```
varying vec3 ReflectDir;

void main() {
    vec4 color = textureCube(Sampler, ReflectDir);
    gl_FragColor = color;
}
```

Jak GLSL, tak GLSL ES obsahují vestavěný datový typ pro reprezentaci krychle s texturou `samplerCube`. Dále obsahují vestavěnou funkci `textureCube`, která získá z cube map textury barvu podle zadaného vektoru.
Pro cube map je potřeba vybrat správnou texturu, takovou, na které při umístění textury krychle nebudou patrné hrany. Výsledná textura by mohla vypadat podobně jako na obrázku 5.9. Pokud chci využít vestavěné funkce a datový typ pro cube map, musím texturu

Obrázek 5.9: Příklad cube map textury

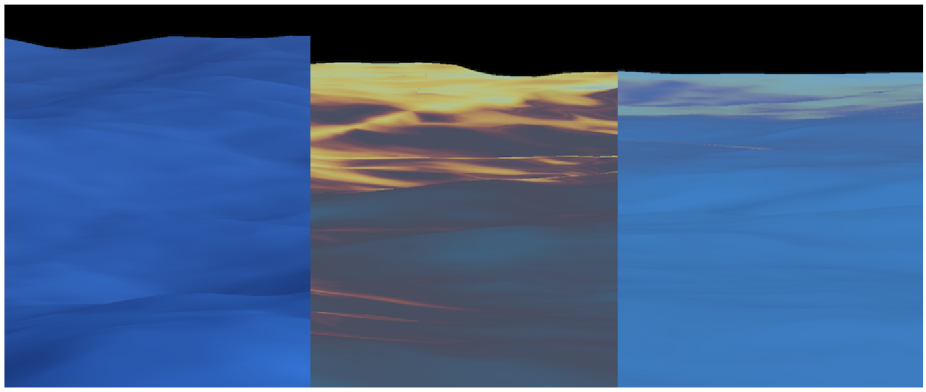
nahrát do paměti grafické karty specifickým způsobem. Rozdělím texturu na šest stěn, které umístím na výslednou krychli. Postup je ve stručnosti následující:

- Vytvořím funkcí `glGenTextures` texturu a funkcí `glBindTexture` jí nastavím typ pomocí parametru `GL_TEXTURE_CUBE_MAP`. Tato funkce je také informací pro Open GL, že se bude právě tato textura používat.
- Pomocí funkce `glTexImage2D` nahraji do paměti jednotlivé textury všech šesti stěn

krychle. Jedním z parametrů je typ textury, který může nabývat hodnot `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z` a `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`, pro levou a pravou stěnu, horní a dolní stěnu a nakonec pro přední a zadní.

- nakonec funkcí `glGenerateMipmap(GL_TEXTURE_CUBE_MAP)` vygeneruji *mipmap* a vytvořím tak cube map.

Tímto postupem vytvořím cube map, která se automaticky naváže na proměnnou typu `samplerCube` ve fragment shaderu, a to z toho důvodu, že používám pouze jednu jednotku pro texturování.
Po výpočtu odrazu přičtu výslednou barvu k hodnotě získané pomocí Phongova světelného modelu. Na obrázku 5.10 je porovnání vodní hladiny pouze s odrazem (uprostřed) a vodní hladiny s kombinací odrazu a Phongova osvětlení.



Obrázek 5.10: Phongův světelný model (vlevo), vodní hladina s odrazem (uprostřed) a vodní hladina s použitím odrazu a Phongova osvětlení (vpravo)

5.9 Skybox

Pro zobrazení okolí vykreslované scény používám *skybox*, což je krychle s texturou, která je vykreslena zdánlivě v nekonečnu. K tomuto účelu jsem použil již vytvořenou cube map 5.8.2. Bylo však potřeba si vytvořit zároveň krychli, na jejíž stěny umístím požadovanou texturu. Protože Open GL ES 2.0 neumožňuje vykreslování grafických primitiv, jako jsou polygon nebo čtyřúhelník, vytvořil jsem si pruh trojúhelníků, neboli *triangle strip*.
Souřadnice rohů jsou:

```
GLfloat vertices[24] = {
    -1.0f, -1.0f,  1.0f, 1.0f, -1.0f,  1.0f,
    -1.0f,  1.0f,  1.0f, 1.0f,  1.0f,  1.0f,
    -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f, 1.0f,  1.0f, -1.0f,
};
```

Jedná se o *jednotkovou krychli*, jejíž stěny vykreslím průchodem jednotlivými jejími rohy v pořadí:

```
GLubyte indices[14] = {0, 1, 2, 3, 7, 1, 5, 4, 7, 6, 2, 4, 0, 1};
```

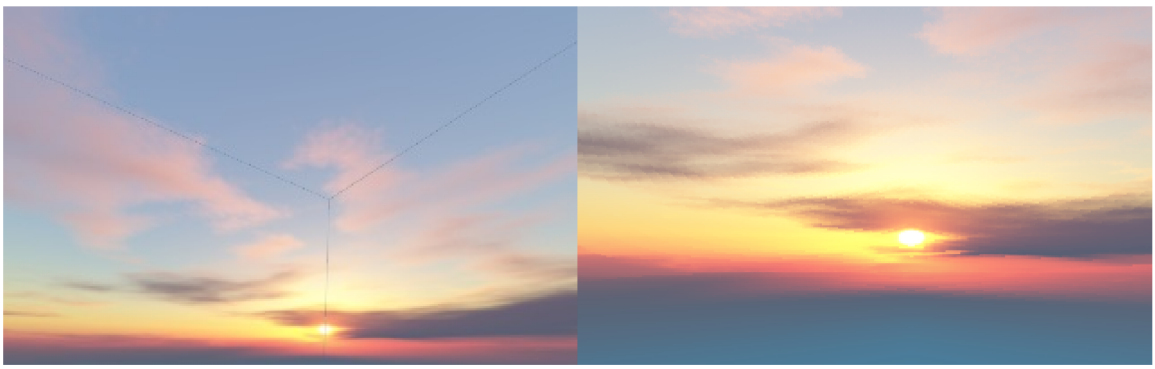
K předání dat grafické kartě použiji funkci `glDrawElements`, blíže popsanou v podkapitole 6.1. K vykreslení výsledného skyboxu používám jednoduchý vertex a fragment shader. Ve vertex shaderu podle pohledu kamery natočím krychli transformací jejích vrcholů a pomocí normálové matice spočítám souřadnice textury, které následně předám fragment shaderu pro texturování.

```
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat3 normalMatrix;
attribute vec4 position;
varying vec3 texCoord;

void main() {
    gl_Position = modelViewMatrix * projectionMatrix * position;
    texCoord = normalMatrix * (gl_Position.xyz);
}
```

Ve fragment shaderu použiji texturovací souřadnice pro vykreslení textury.

```
uniform samplerCube Sampler;
varying vec3 texCoord;

void main() {
    vec3 cube = vec3(textureCube(Sampler, texCoord));
    gl_FragColor = vec4(cube, 1.0);
}
```

Během implementace skyboxu jsem řešil problém, kdy mi jednotlivé stěny krychle opticky nedoléhaly jedna k druhé, jak je vidět na obrázku 5.11 vlevo. Běžně je toto způsobeno nepřesností souřadnic datového typu `float` a implicitní vlastností textur v Open GL, kdy se textura, pokud nevyplní celou plochu, opakuje.



Obrázek 5.11: Chyba ve vykreslování (vlevo) a korektní vykreslování (vpravo)

Toto chování je možné samozřejmě změnit nastavením textury. Při načítání jsem každé textuře, reprezentující jednotlivé strany krychle, nastavil její roztažení na celou šířku plochy

jak v horizontálním, tak ve vertikálním směru. Jinými slovy jsem se pokusil následujícími příkazy znemožnit opakování textury:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Toto nastavení se nesetkalo s úspěchem, bylo totiž nutné jej provést nikoliv pro každou z načtených textur, neboli stěn, ale pro celou cube map takto:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Na obrázku 5.11 vpravo je možné vidět výsledek.

5.10 Implementace pohledu kamery

Pro implementaci jsem musel vyřešit dvě věci: pohyb kamery po scéně a ovládání pohledu kamery, tedy její natočení.
Při natočení kamery počítám nový směr vektoru pohledu kamery, zatímco pozice kamery zůstává nezměněna. Využiji tedy goniometrických funkcí *sinus* a *cosinus*, princip je patrný z obrázku 5.12. Pro rotaci kolem svislé osy, tedy rotaci horizontální, platí následující vztahy:

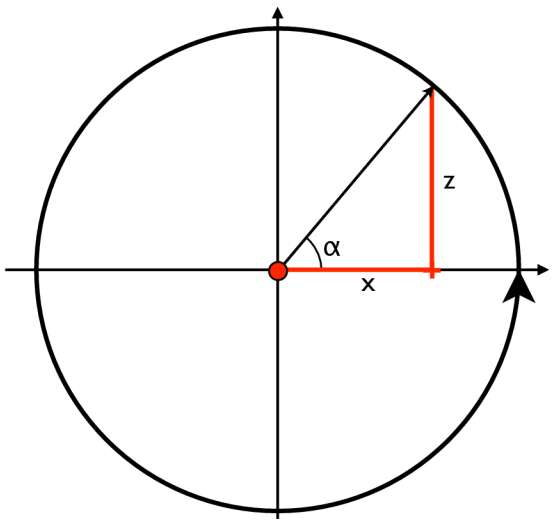
$$x = cos(\alpha) \tag{5.3}$$

$$z = sin(\alpha) \tag{5.4}$$

Pro výpočet vertikální rotace, tedy rotace kolem osy x platí:

$$y = -sin(\alpha) \tag{5.5}$$

Toto jsou nové hodnoty vektoru určujícího směr pohledu kamery. Pro korektní výpočet pohybu kamery jsou velice důležité.



Obrázek 5.12: Výpočet rotace kamery

Při výpočtu pohybu kamery je situace následovná:

- Při pohybu vpřed nebo vzad, tedy ve směru nebo proti směru pohledu, stačí provést součet vektoru značícího polohu kamery a vektoru směru pohledu, čímž získám novou polohu kamery.
- Při pohybu kamery vlevo přičtu vektor získaný vektorovým součinem vektoru směru pohybu a vektoru označujícím směr horní hrany kamery. Při pohybu vpravo provádím opačný vektorový součin. Vektorovým součinem získám vektor kolmý k těmto dvěma vektorům.

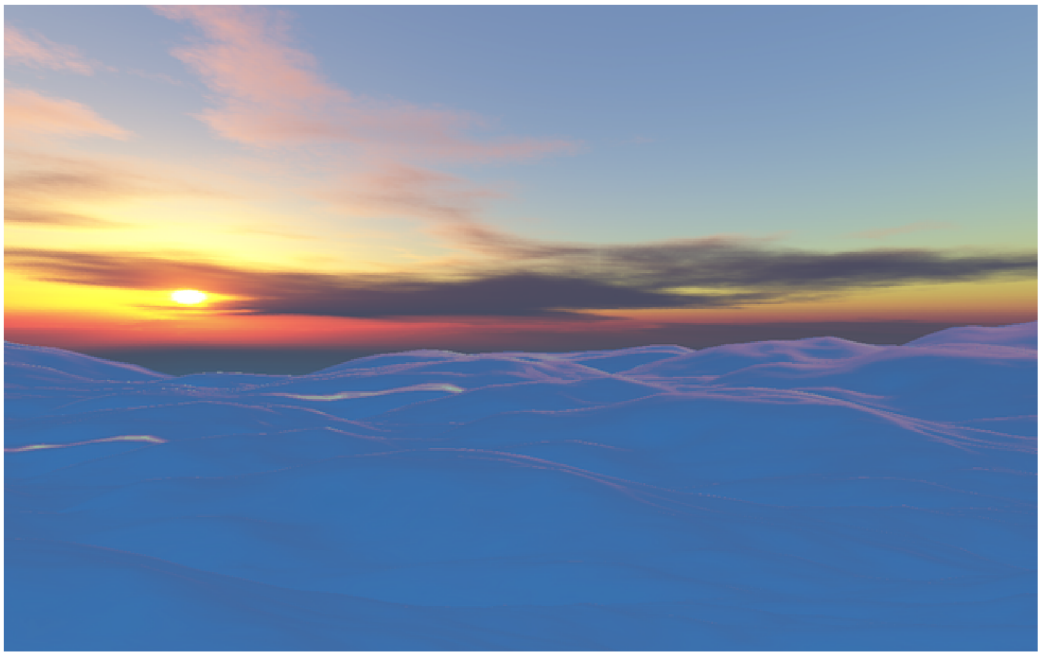
5.11 Přenositelnost aplikace

Od začátku vývoje jsem chtěl aplikaci přenést na mobilní zařízení s platformou iOS. Ve výsledku se to podařilo, i když jsem musel překonat několik problémů.
Nejdříve jsem se musel rozhodnout, kterou verzi Open GL ES budu používat. Verze 2.0 podporuje programovatelný grafický řetězec, bohužel však není zpětně kompatibilní z verzí 1.0, která používá grafický řetězec fixní. Použitím verze 2.0 jsem přišel o kompatibilitu se staršími zařízeními, podporována jsou pouze zařízení iPhone 3GS, iPhone 4, iPad a iPad 2 s operačním systémem iOS 4.3 a vyšším. Starší verze zařízení a operačního systému nejsou podporovány.
Každé z těchto mobilních zařízení má obrazovku s jiným rozlišením. Obrazovka iPhone verze 3GS má rozlišení 480x320 bodů, iPhone čtvrté generace 960x640 bodů a obě verze iPadu 1024x768. Protože používám textury pouze pro pozadí scény, nemusel jsem textury brát v potaz, do budoucna s tím ale musím počítat.
Až na pár drobných detailů je implementace pro mobilní zařízení a stolní počítače shodná. Většinu těchto rozdílů řeším pomocí příkazů pro preprocesor `#ifdef`, `#ifndef`, `#else` a `#endif`, spolu s nadefinovaným makrem `IPHONE`.
Rozhodl jsem se, že aplikace pro platformu iOS bude implementována tak, aby zobrazovala obsah v rozvržení na šířku. Musel jsem tedy ve všech případech prohodit osy `x` a `y`, což bylo někdy matoucí.
V shader programech pro mobilní zařízení je potřeba, podle specifikace jazyka GLSL ES, uvádět u definicí proměnných spolu s datovým typem i přesnost datového typu jedním z klíčových slov `lowp`, `mediump` a `highp`, proto používám dvě verze shader programů.
Pro vykreslování se u mobilních zařízení používá explicitně vytvořený *renderbuffer* a *framebuffer*. Na platformě Mac OS X je není potřeba vytvářet, jsou implicitně vytvořeny pomocí objektu `NSOpenGLView`.
Pro platformu Mac OS X jsem vyvíjel aplikaci ve verzi 10.6, nižší verze nejsou podporovány kvůli některým chybějícím funkcím knihovny Open GL. Neměl by být ale velký problém zpětnou kompatibilitu dořešit pomocí implementace rozhraní `IRenderingEngine`. Stejně tak by neměl být problém přenést aplikaci na jiné mobilní platformy podporující knihovnu Open GL nebo dokonce na platformu Windows s Direct3D.

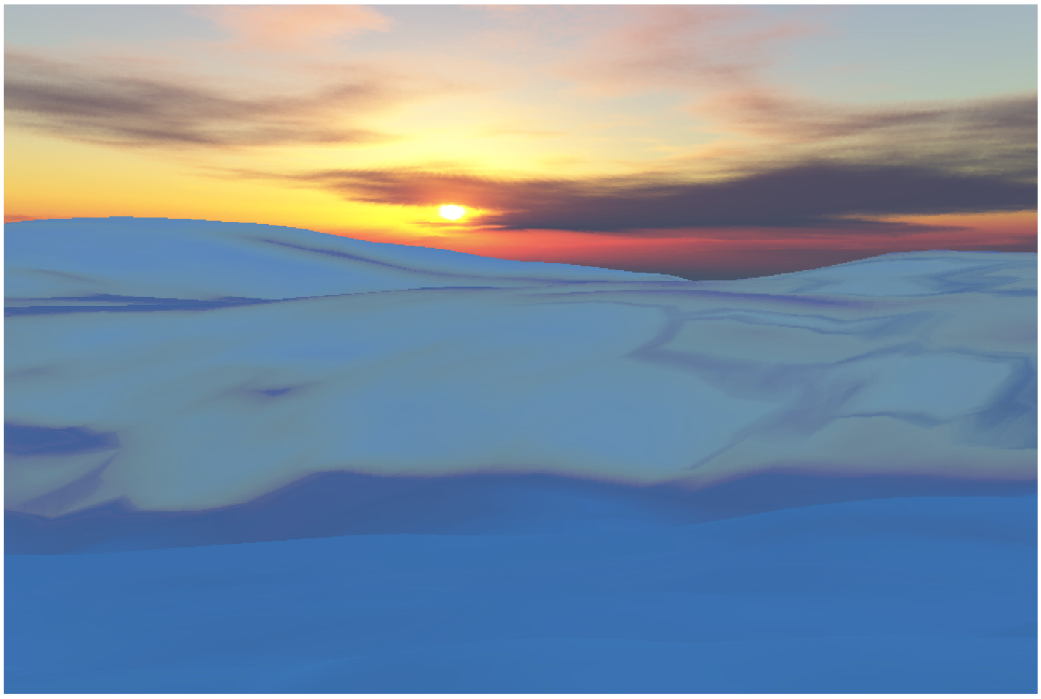
5.12 Výsledek implementace

Na následujících snímcích obrazovky je vidět výsledná implementace aplikace. Jsou použity všechny techniky popsané v této práci. Především se jedná o výpočet vlnění, osvětlení, odrazu světla a vykreslení vlastní hladiny vody spolu pozadím scény.

Obrázek 5.13 je z aplikace běžící na stolním počítači, obrázek 5.14 z mobilního zařízení s operačním systémem iOS.



Obrázek 5.13: Snímek z operačního systému Mac OS X



Obrázek 5.14: Snímek z mobilního zařízení s operačním systémem iOS

Kapitola 6

Optimalizace

Během implementace práce jsem postupně prováděl určité optimalizace, díky kterým jsem byl schopen spustit výslednou aplikaci na slabých strojích i mobilních zařízeních. V této kapitole popíšu několik z nich, a to Vertex Arrays, Vertex Buffer Object, použití programovatelného grafického řetězce a frustum culling.

6.1 Použití Vertex Arrays

Data pro vykreslení je možné posílat grafické kartě několika způsoby. Nejzákladnějším je odeslání dat vrcholů mezi voláním funkcí `glBegin` a `glEnd`. Další možností je použití `DisplayListu`. Já jsem se však rozhodl použít `Vertex Arrays`.
Nejdříve bylo potřeba vytvořit pole hodnot typu `GL_FLOAT`, které tvořilo souřadnice jednotlivých vrcholů. Dále obdobné pole s normálami jednotlivých vrcholů. Souřadnice i normály vrcholů jsou v poli uloženy v pořadí, v kterém je potřeba je vykreslovat. Dále je potřeba pomocí funkcí `glVertexPointer` a `glNormalPointer` informovat Open GL o ukazateli na tyto pole, jejich datovém typu a počtu hodnot na jeden element. Typicky jsou to tři hodnoty. Nakonec stačí funkcí `glDrawArrays` poslat data k vykreslení spolu s údajem o počtu vrcholů a způsobu jejich vykreslení. V mém případě se jednalo o `GL_TRIANGLE_STRIP`.
Tento způsob je možné modifikovat a dále optimalizovat použitím funkce `glDrawElements`. Pole souřadnic a normál nemusí obsahovat data v pořadí, v jakém jej chceme vykreslovat. Pořadí vykreslování se určí pomocí `indices`, což je pole číselných hodnot pozic souřadnic a normál vertexů. V tomto pořadí jsou pak vertexy vykresleny. Tím se zamezí posílání nadbytečných dat k vykreslení.

6.2 Použití Vertex Buffer Object

Použití `VBO` umožňuje uložit data v rychlé paměti grafické karty. Hodí se především pro statická data, která se příliš často nemění. Funkcí `glGenBuffers` je vytvořen nový VBO, následně se funkcí `glBindBuffer` určí jeho použití. VBO může být použit jako pole vertexů, normál případně texturovacích souřadnic při zadání parametru `GL_ARRAY_BUFFER`, případně jako pole indices při zadání parametru `GL_ELEMENT_ARRAY_BUFFER`. Inicializace je dokončena nahráním dat do bufferu funkcí `glBufferData`. Touto funkcí určíme i vlastnosti dat, například jak často se budou měnit.
Vykreslování probíhá obdobně jako v případě Vertex Arrays. Před voláním funkce `glVertexPointer` (které předám nulový ukazatel na data) zavoláme funkci `glBindBuffer`

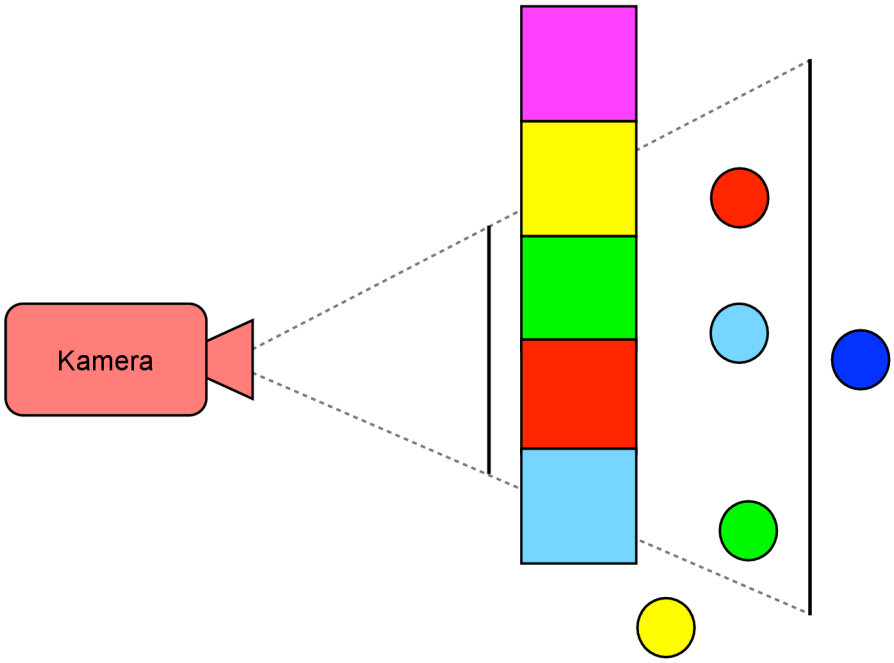
s ID požadovaného bufferu. Tím Open GL oznámím, že má použít data právě ze zadaného bufferu.

6.3 Programovatelný grafický řetězec

K použití programovatelného grafického řetězce mě dovedla absence fixního programovatelného řetězce v Open GL ES 2.0. Ten vyžaduje použití shaderovacích programů pro vykreslování. Urychlil jsem tím především zpracování vertexů.
Kombinuji předchozí dva přístupy, kdy při inicializaci uložím do VBO statická data reprezentující klidnou hladinu s nulovou výškou. Při každém překreslení jsou pak pomocí Vertex Array odeslána do vertex shaderu pouze normály a výškové souřadnice jednotlivých vertexů. Ve vertex shaderu je pak výška přičtena k nulové hladině.
V shaderovacích programech počítám i upravený Phongův světelný model. Nepředpokládám však, že by vlastní implementace byla rychlejší než systémová, která se používá při použití fixního grafického řetězce.

6.4 Frustum culling

Frustum culling je test, kterým se pro každý objekt ve scéně zjišťuje, jestli se nachází uvnitř zorného pole kamery. Zamezí se tím posílání zbytečných dat grafické kartě ke zpracování.



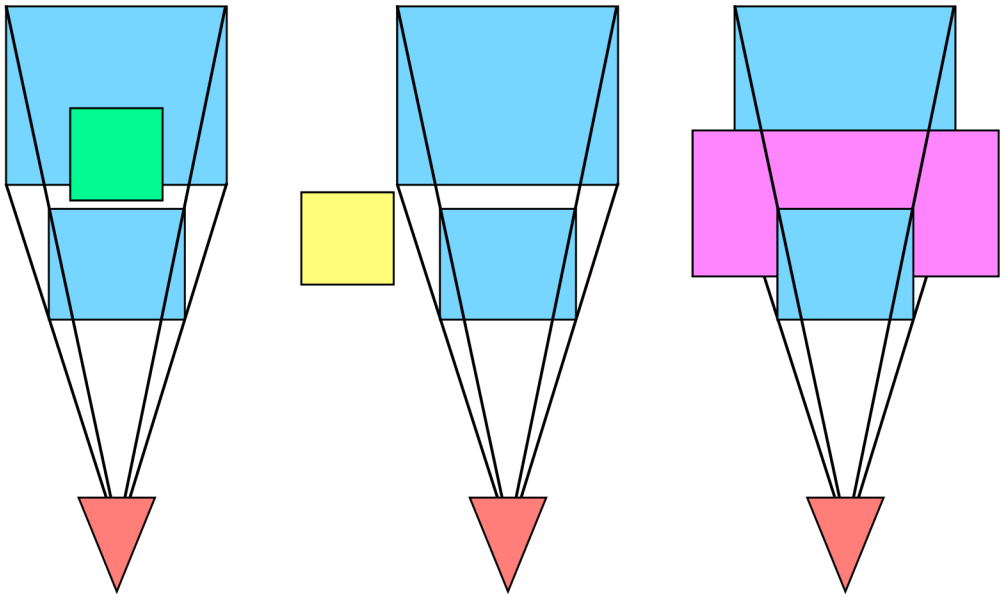
Obrázek 6.1: Frustum culling

V mém případě jsem testoval, jestli se některá z vykreslovaných dlaždic nachází v zorném poli kamery. Postupně jsem porovnal všechny čtyři její rohy a každý z nich testoval, jestli leží před nebo za plochami tvořícími zorné pole.
Řešil jsem i situaci, kdy je kamera příliš blízko dlaždici a všechny její rohy leží vně pohledu

ale každý vně jiné plochy. Stačí testovat, zda všechny čtyři rohy leží vně jedné z ploch zároveň.
Způsob, jakým jsem zjišťoval, na které straně ploch leží jednotlivé body vychází z obecné rovnice roviny:

$$ax + by + cz + d = 0 \tag{6.1}$$

Hodnoty a, b, c a d jsou souřadnice normály kolmé k rovině a udávají její orientaci. Hodnoty x, y a z jsou souřadnice bodu ležícího na rovině. Pokud tedy do této rovnice dosadím souřadnice bodu, o kterém chci zjistit, na které straně plochy leží, dostanu jeden ze tří možných výsledků. Pokud bude hodnota rovnice rovna nule, bod leží na dané rovině. Pokud bude hodnota větší než nula, bod leží mimo rovinu ve směru její normály, tedy pro mé účely může být viditelný. Pokud je však výsledná hodnota menší než nula, bod je neviditelný, protože leží vně roviny.
Při vyhodnocování viditelnosti mohou nastat tři situace, znázorněné na obrázku 6.2.



Obrázek 6.2: Tři situace při vyhodnocování viditelnosti

První z nich nastane, pokud je minimálně jeden, maximálně čtyři rohy vykreslované dlaždice hladiny uvnitř pohledu. V tom případě je dlaždice viditelná. Ve druhém případě jsou všechny čtyři rohy vně pohledu a dlaždice je neviditelná. Třetí případ nastane, pokud je kamera tak blízko dlaždice, že jsou všechny čtyři rohy vně pohledu ale dlaždice by měla být logicky viditelná.
Řešením je zneviditelnění dlaždice pouze v případě, že se všechny její čtyři rohy nacházejí vně jedné z ploch zároveň.

Kapitola 7

Výkon implementace

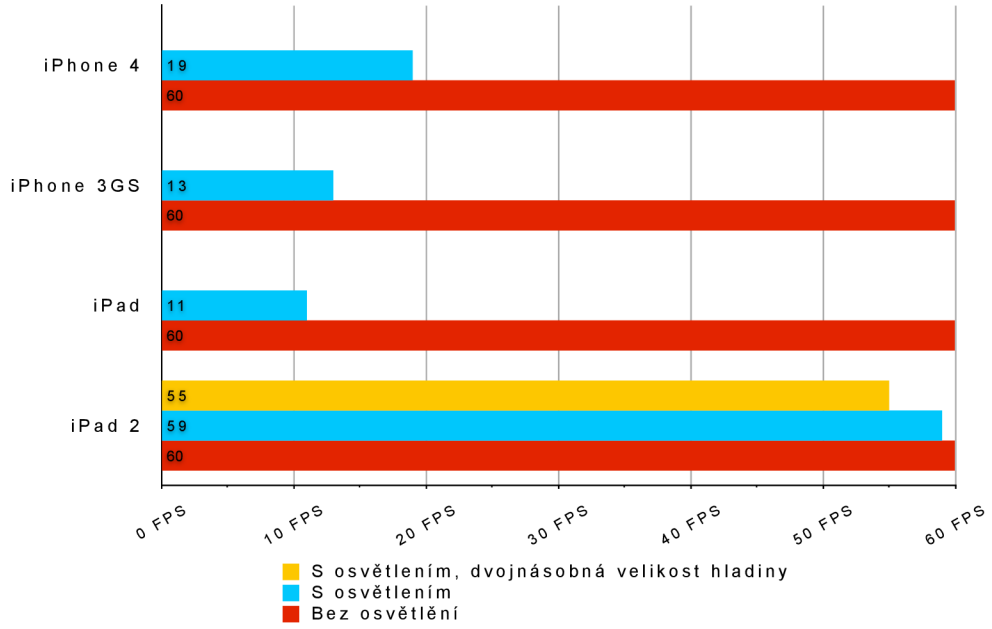
Aplikaci jsem otestoval na různých zařízeních. Snažil jsem se pokrýt široké spektrum hardware instalovaného do počítačů firmy Apple.
Ve stolních a přenosných zařízeních jsou zastoupeny grafické karty firem *AMD* a *nVidia*, několik generací procesorů *Intel* a jsou osazené různými kapacitami operační paměti. Největší vliv na rychlost aplikace bude mít procesor a grafická karta. Na druhou stranu kapacita operační paměti nebo rychlost disku se neprojeví. K otestování se mi podařilo sehnat jak nejnovější verze přenosných a stolních počítačů, tak i několik starších.
Mobilní zařízení s operačním systémem iOS, která jsem otestoval, jsou nejnovější verze mobilního telefonu iPhone 4, jeho předchůdce iPhone 3GS, první generace tabletu iPad a také jeho nejnovější verze iPad 2.
Naměřené hodnoty jsou pouze orientační a nemusí přesně odpovídat výkonu konfigurace.

7.1 Mobilní zařízení

Mobilní zařízení jsem otestoval na dvou verzích aplikace. Jedna obsahuje výpočet osvětlení a je konečnou verzí aplikace, druhá výpočet osvětlení neprovádí. Chtěl jsem porovnat, jak moc je výpočet osvětlení náročný.

Na obrázku 7.1 je vidět, jak jsou jednotlivá zařízení výkonná. Výpočet osvětlení je pro mobilní zařízení velice náročný. Další optimalizaci by si zřejmě vyžádal výpočet normál a zřejmě i výpočet Phongova světelného modelu.

Všechna zařízení dosahují při běhu bez výpočtu osvětlení stabilně rychlosti vykreslování 60 snímků za sekundu. Tato horní hranice je dána systémem aby Open GL aplikace příliš nezatěžovaly procesor a grafickou kartu. Velice dobrý je výsledek zařízení iPad 2, který dosahoval této hranice i při výpočtu osvětlení. Dokonce při dvojnásobné ploše vodní hladiny a tedy větší zátěži na procesor a grafickou kartu poskytoval nadprůměrný výkon. To je dáno výkonnějším procesorem a především výkonnou grafickou kartou.

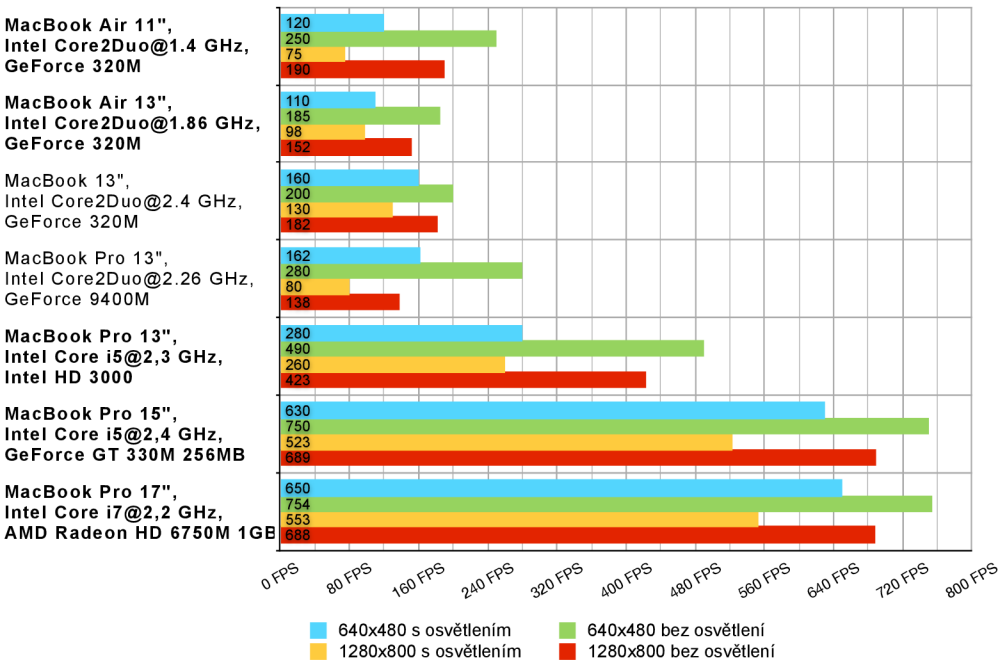


Obrázek 7.1: Porovnání výkonu mobilních zařízení

7.2 Přenosné počítače

Přenosné počítače jsem testoval ve dvou rozlišeních, 640x480 a 1280x800 obrazových bodů. Obě verze potom s vypnutým a zapnutým výpočtem osvětlení.
Podařilo se mi otestovat i nejnovější verze přenosných počítačů, ty jsou v obrázku 7.2 označeny tučným písmem.

Velice zajímavý výkon podávají především konfigurace s nejnovějšími verzemi procesorů firmy Intel osazené grafickými kartami AMD Radeon. Nejsilnější konfigurace jsou schopny překreslit přes 500 snímků za sekundu.
Je zajímavé, že i nejslabší konfigurace osazené úspornými procesory a integrovanými grafickými kartami dokáží soupeřit s dva roky starými počítači.

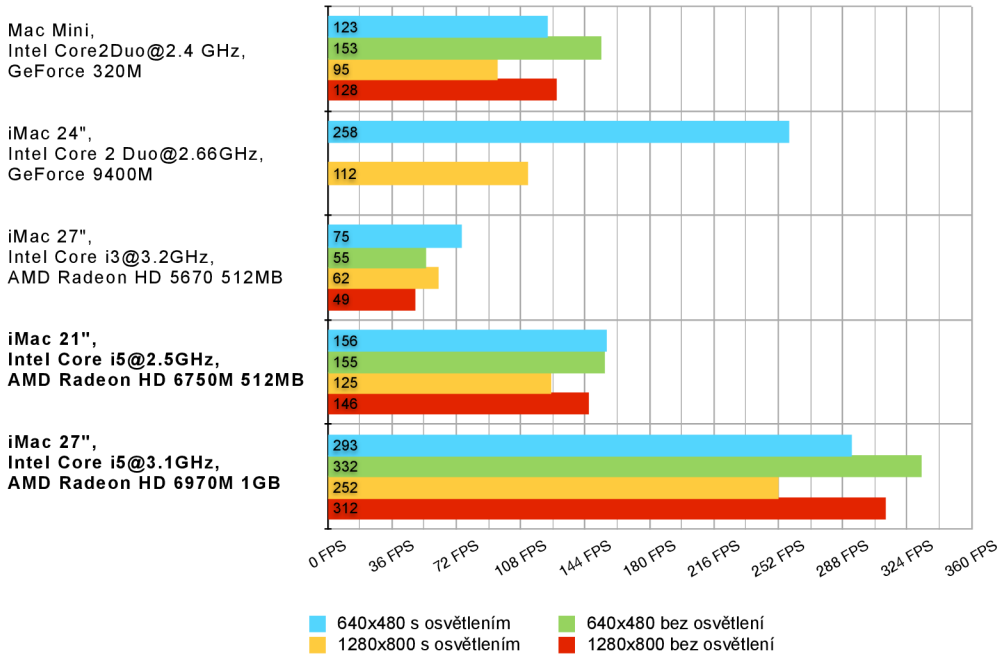


Obrázek 7.2: Porovnání výkonu přenosných počítačů

7.3 Stolní počítače

K otestování jsem sehnal pouze stolní počítače iMac, které jsou vybaveny mobilními procesory a grafickými kartami. Rozdíl ve výkonu tedy není příliš velký. Bohužel se mi nepodařilo sehnat ani jednu pracovní stanici Mac Pro.
Na obrázku 7.3 jsou nejnovější konfigurace vyznačeny tučným písmem.

Velice zajímavé je, že stolní počítače iMac podávají menší výkon než ty přenosné, přestože používají výkonnější komponenty. Může to být způsobeno nepřesností měření nebo nevyužitím veškerého výkonu. Je totiž možné, že kvůli úspoře energie běžely procesor a grafická karta na nižší výkon.



Obrázek 7.3: Porovnání výkonu stolních počítačů

7.4 Možnosti optimalizace

Podařilo se mi objevit několik slabých míst, které by bylo možné dále optimalizovat, a tím zvýšit výkon jak na stolních a přenosných počítačích se systémem Mac OS X, tak na mobilních zařízeních se systémem iOS.
Jsou to tyto:

- Výpočet normál - Výpočet normál by bylo potřeba optimalizovat, protože se provádí pro každý trojúhelník při každém překreslení obrazovky. Optimalizace by spočívala v lepší implementaci práce s vektory.
- Výpočet Phongova světelného modelu - optimalizace programu pro vertex a fragment shader.
- Optimalizace shader programů - VBO - spočívá především v odstranění nadbytečného posílání dat pro vertex shader. Posílání dat při každém překreslení má význam pouze u některých dat, jako jsou normály, které se neustále mění. Nejvhodnější je použít *VBO* neboli *Vertex Buffer Object* a související funkce *glGenBuffers*, *glBindBuffer* a *glBufferData*.
- Optimalizace shader programů - VAO - pro konfiguraci posílání pole trojúhelníků grafické kartě je možné použít *VAO* neboli *Vertex Array Object*.

- Pokročilé techniky Level-Of-Detail - pro omezení počtu dat posílaných grafické kartě a trojúhelníků, které vykresluje, by bylo vhodné implementovat jednu z pokročilých metod *Level-Of-Detail*.
- Výpočet na GPU - jednou z možností je provádět část výpočtů na grafické kartě, pokud by se režie výpočtu na výkonu neprojevilo spíše negativně.
- Posílání dat shaderovacím jednotkám prokládaně - to znamená posílat za sebou souřadnice vertexu, jeho normálový vektor a příslušnou texturovací souřadnici.
- Pro textury použít kompresi a případně multitexturování.

Kapitola 8

Závěr

Během práce na Diplomovém projektu jsem prostudoval metody pro popis vlnění vody. Některé z nich, jako například NSE, jsou vhodné pouze pro fyzikální simulace. Na druhou stranu ty, které jsou založené na statistické analýze a počítány pomocí FFT, jsou vhodné pro použití v počítačové grafice pro zobrazování v reálném čase. Práce Jerryho Tessendorfa, ze které jsem vycházel, popisuje jednu takovou metodu.
Implementoval jsem metodu založenou na Phillipsově spektru a využívající k výpočtu FFT. Celou práci jsem implementoval tak, aby byla jednoduše přenositelná na mobilní zařízení s platformou iOS a případně jiné platformy.
Během práce jsem otestoval hranice výkonnosti mobilních zařízení a získal spoustu zkušeností a znalostí v programování Open GL aplikací pro mobilní zařízení. Zároveň jsem tyto znalosti využil při optimalizaci Open GL aplikací pro stolní počítače.
Největším přínosem je pro mě implementace vybrané metody, její přenos na mobilní platformu, rozšíření znalostí v oblasti programování grafického řetězce.
Aplikaci bych chtěl v dalším vývoji dále optimalizovat použitím pokročilých metod Level-Of-Detail, vylepšit vizuální stránku použitím textur, lomu světla a částicových systémů použitých pro reprezentaci pěny na vodní hladině. Dalším vývojem bych chtěl z práce vytvořit jednoduchou počítačovou hru nebo multimediální aplikaci.

Literatura

[1] Alan V. Oppenheim, R. W. Schafer, J. R. Buck: *Discrete-time signal processing.* Prentice Hall, 1999, iSBN 0-13-754920-2.

[2] Apple Inc.: OpenGL Programming Guide for Mac OS X. http://developer.apple.com/library/mac/documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl_intro/opengl_intro.html (22.5.2011).

[3] Apple Inc.: vDSP Programming Guide. http://developer.apple.com/library/mac/documentation/Performance/Conceptual/vDSP_Programming_Guide/Introduction/Introduction.html (22.5.2011).

[4] B. T. Phong: *Illumination for computer generated pictures, Communications of ACM 18, no. 6, 311-317.* ACM, 1975.

[5] D. J. Acheson: *Elementary Fluid Dynamics.* Oxford University Press, 1990, iSBN 0198596790.

[6] Dave Shreiner: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition).* Addison-Wesley Professional, 2009, iSBN 0321552628.

[7] E. F. Carter: *The Generation and Application of Random Numbers, Forth Dimensions Vol XVI Nos 1, 2.* Forth Interest Group, 1994.

[8] James W. Cooley, John W. Tukey: *An algorithm for the machine calculation of complex Fourier series.* Math. Comput. 19: 297-301, 1965.

[9] Jerry Tessendorf: *Simulating Ocean Water.* SIGGRAPH 2001 Course Notes, 2001.

[10] Khronos Group: OpenGL Shading Language Documentation. http://www.opengl.org/documentation/glsl/ (22.5.2011).

[11] M. Finch: *Chapter 1. Effective Water Simulation from Physical Models. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics.* Addison-Wesley Professional, 2004, iSBN 0321228324.

[12] O. M. Phillips: *On the generation of waves by turbulent wind.* Journal of Fluid Mechanics. 2 (5): 417-445, 1957.

[13] Philip Rideout: *iPhone 3D Programming, Developing Graphical Applications with OpenGL ES.* O'Reilly Media, 2009, iSBN 9780596804824.

[14] Tamás Umenhoffer, Gustavo Patow, László Szirmay-Kalos: *Chapter 17. Robust Multiple Specular Reflections and Refractions. GPU Gems 3.* Addison-Wesley Professional, 2007, iSBN 0321515261.

Příloha A

Obsah CD

Přiložené CD obsahuje:

- Adresář `doc` obsahující zdrojové soubory pro vytvoření písemné zprávy pomocí prostředku LaTeX.
- Adresář `src` obsahující zdrojový kód aplikace.
  - Adresář `Common` obsahuje části společné pro mobilní verzi aplikace a verzi pro stolní počítače.
  - Adresář `Diplomka-iphone` obsahuje zdrojové soubory a projekt pro přeložení aplikace pro mobilní zařízení.
  - Adresář `Diplomka-mac` obsahuje zdrojové soubory a projekt pro přeložení aplikace pro stolní počítače.
  - Adresář `execs` obsahuje spustitelné soubory pro mobilní zařízení s iOS 4.3.x a stolní počítače s Mac OSX 10.6.x.
- Soubor `projekt.pdf` s elektronickou podobou této práce.
- Soubor `README` obsahující stručný popis práce s aplikací.
- Soubor `WATCHME.mov` s video ukázkou aplikace, ke shlédnutí také na adrese: http://vimeo.com/24084048.

Příloha B

Manuál

Pro překlad aplikace je potřeba mít nainstalované prostředí XCode, aplikace byla vyvíjena pod verzí 4.0 ale je přeložitelná i ve verzi 3.2.1. Aplikace vyžaduje operační systém Mac OS X 10.6.x. Aplikace byla testována na verzích 10.6.4, 10.6.5, 10.6.6 a 10.6.7. Na verzích operačních systémů 10.5.x a nižších mohou nastat problému s překladem z důvodu některých chybějících funkcí knihovny OpenGL.
Po přeložení je možné přepnout aplikaci do celoobrázkového režimu stiskem klávesy `F`. Stejnou klávesou je možné přepnout aplikaci zpět do režimu okna. Klávesou `Q` lze aplikaci ukončit. Klávesami `W, S, A, D` se lze po scéně pochybovat. Kamera se ovládá pohybem myši při stisknutém levém tlačítku.

Mobilní verze aplikace je spustitelná na operačním systému iOS verze 4.3.x a zařízeních iPhone 3GS, iPhone 4, iPad a iPad 2. Starší zařízení nejsou podporována, starší verze operačního systému iOS nejsou otestovány ale aplikace by na nich měla být spustitelná.
Po spuštění je možné tahem prstu po dotykovém displeji ovládat pohyb kamery a rozhlédnout se po scéně.