**Univerzita Hradec Králové**
**Fakulta informatiky a managementu**
**Katedra informatiky a kvantitativních metod**

# Renderování vodní hladiny v OpenGL

Bakalářská práce

Autor: Dennis Tschamler
Studijní obor: Aplikovaná informatika

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Hradec Králové duben 2020

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 25.4.2020 Dennis Tschamler

Poděkování:

Děkuji vedoucímu bakalářské práce prof. RNDr. PhDr. Antonín Slabý, CSc. za metodické vedení práce, trpělivost, cenné rady a vstřícnost při konzultacích.

# Anotace

Bakalářská práce se zabývá problematikou renderování vodní hladiny za použití grafické knihovny OpenGL. V práci jsou na začátku popsány optické vlastnosti vody z fyzikálního hlediska jako je například odraz a lom světla na vodní hladině. Poté je menší část práce věnována osvětlení scény a metodám stínování objektů. Dále je představen historický vývoj renderování vodní hladiny od konce 70. let a stručný popis jednotlivých technik. Druhá část práce je převážně věnována multiplatformní grafické knihovně OpenGL. Důraz je kladen zejména na základní principy fungování této knihovny. Pomocí OpenGL budeme implementovat algoritmus pro renderování vodní hladiny. V závěru je zhodnocen vizuální výsledek algoritmu a jeho možné další vylepšení.

# Annotation

## Title: Rendering water surface in OpenGL

The bachelor thesis deals with the issue of water surface rendering using the OpenGL graphics library. At the beginning, the optical properties of water from a physical point of view, such as the reflection and refraction of light on the water surface, are described. Then a smaller part of the work is dedicated to scene lighting and methods of shading objects. Furthermore, the historical development of water surface rendering since the late 70´s and a brief description of individual techniques are presented. The second part of the work is mainly devoted to the multiplatform graphics library OpenGL. Emphasis is placed especially on the basic principles of operation of this library. Using OpenGL, we will implement the algorithm for rendering the water surface. Finally, the visual result of the algorithm and possible further improvements are evaluated.

# Obsah

# Seznam obrázků

## Seznam tabulek

# 1 Úvod

Voda je fascinují kapalina, jejíž vizualizaci není jednoduché fyzikálně přesně reprezentovat pomocí počítače. Už koncem 70. let se začaly objevovat první metody pro renderování vodní hladiny. Od té doby vzniklo mnoho různých metod, kdy každá má své specifické užití. Pokud potřebujeme vykreslit scénu v reálném čase, často se přistupuje k fyzikálně nepřesným metodám, které se snaží vodu co nejlépe napodobit. Je to dáno tím, že fyzikálně přesné metody pro renderování vodní hladiny jsou pro počítač výpočetně náročné a tedy jsou nevhodné pro scénu v reálném čase. Ovšem s nástupem výkonnějších počítačů lze aplikovat více metod na tyto scény.

Samotné téma renderování vodní hladiny jsem si vybral proto, že mě vždy fascinovaly metody, kterými lze v počítačové grafice vykreslit přírodní jev jako je voda. Tyto metody se často snaží nějakým důmyslným algoritmem vodu co nejvíce napodobit.

V první kapitole si popíšeme optické vlastnosti vody z fyzikálního hlediska. Tyto znalosti využijeme v následné implementaci samotného algoritmu. Další část se věnuje osvětlení scény a způsobu, jak lze takové osvětlení vypočítat. Poté představíme historický vývoj hlavních dostupných metod pro renderování vodní hladiny. Další kapitola je věnována grafické knihovně OpenGL, kde si popíšeme její základní vlastnosti a funkce, které následně využijeme v poslední kapitole. Tato kapitola se věnuje implementaci algoritmu, který nám vytvoří realistické zobrazení vodní hladiny.

Cílem práce bude představit problematiku renderování vodní hladiny, představit grafickou knihovnu OpenGL. Na závěr pomocí OpenGL implementovat algoritmus pro renderování realisticky působící vodní hladiny.

# 2 Cíl práce

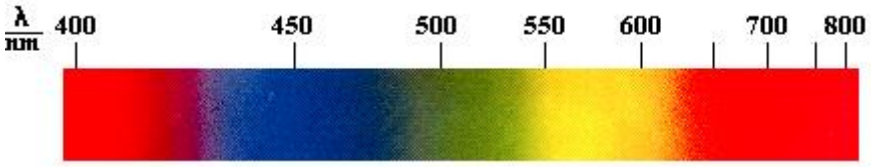
Cílem práce je uvést čtenáře do problematiky renderování vodní hladiny pomocí grafické knihovny OpenGL. V první části budou popsány optické vlastnosti vody, metody stínování a existující metody renderování vodní hladiny. V druhé části bude představena grafická knihovna OpenGL (včetně základních technik pro reprezentaci 3D scény) a pomocí této knihovny bude následně implementován algoritmus pro renderování vodní hladiny společně s okolní scénou (terén, ostatní objekty, skybox).

# 3 Optické vlastnosti vody

Optika zkoumá světlo a světelné jevy. Světlo je elektromagnetické vlnění a má svoji rychlost, kterou značíme písmenem c podle latinského výrazu *celeritas* [1]. Rychlost světla ve vakuu je důležitá fyzikální konstanta (viz rovnice 3.1), která se používá v mnoha výpočtech. Jedná o nejvyšší možnou rychlost, které lze ve vesmíru dosáhnout [2].

$$c = 299\,792\,458\,m \cdot s^{-1} \tag{3.1}$$

Světlo charakterizuje vlnová délka $\lambda = \frac{c}{f}$ , kde f je frekvence světla. Jednotlivé frekvence světla vnímáme vlastně jako barvy [2]. Světelný interval, který je viditelný lidským okem, je ohraničen fialovou barvou ($\lambda_f = 390nm$) a červenou barvou ($\lambda_{\check{c}} = 790nm$) viz **OBR. 1**.




**Obr. 1 Světelný interval**
Zdroj: Převzato z [2]

Voda dokáže velice dobře demonstrovat hned několik optických jevů. Odraz (nebo také reflexe) světla od vodní hladiny lze pozorovat např. u jezera, kde se objekt za jezerem zobrazí na hladině. Lom (nebo také refrakce) světla ve vodě můžeme pozorovat při vložení jakéhokoliv tělesa do vody. Dobře patrné je to například při ponoření pádla do vody.

## 3.1 Optické prostředí

Optické prostředí definujeme jako místo, v kterém se šíří světlo a dělíme ho do tří kategorií:

- Průhledné – světlo se nerozptyluje
- Průsvitné – část světla prochází prostředím, část světla se rozptyluje
- Neprůhledné – světlo zcela pohltí nebo se na rozhraní pouze odráží

Vodu řadíme mezi průhledná prostředí. Dále můžeme optická prostředí rozdělit podle rychlosti světla a jejího chování v určitých směrech. Izotropní prostředí má ve všech směrech stejnou rychlost světla a u anizotropního prostředí závisí rychlost světla na směru šíření vlnění [1].

## 3.2 Odraz světla na vodní hladině

Světlo se částečně odráží od vodní hladiny. Odraz je definován pomocí zákona odrazu, který platí pro všechny materiály.

### 3.2.1 Zákon odrazu

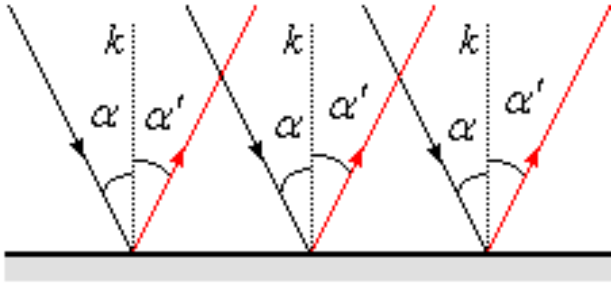
Zákon odrazu nám říká, že úhel odrazu je roven úhlu dopadu. Odražený paprsek zůstává v rovině dopadu (v rovině dané dopadajícím paprskem a kolmicí dopadu) a svírá s kolmicí dopadu úhel odrazu, který je stejně velký jako úhel dopadu viz rovnice 5.2.

$$\alpha' = \alpha \tag{3.2}$$

Úhel odrazu nezávisí na frekvenci dopadajícího světla, proto se paprsky světla různých barev (frekvencí) odrážejí stejně [2].

### 3.2.2 Pravidelný odraz

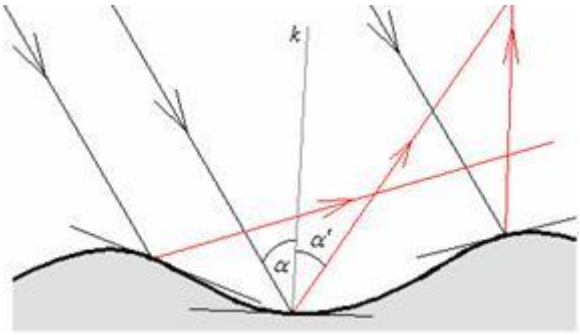
Pokud vyšleme několik rovnoběžných paprsků světla na klidnou rovnou hladinu pod úhlem alfa, tak se všechny odrazí rovnoběžné také pod úhlem alfa viz OBR. 2. Dochází k tzv. pravidelnému odrazu.




**Obr. 2 Pravidelný odraz**
Zdroj: Převzato z [2]

### 3.2.3 Difúzní odraz

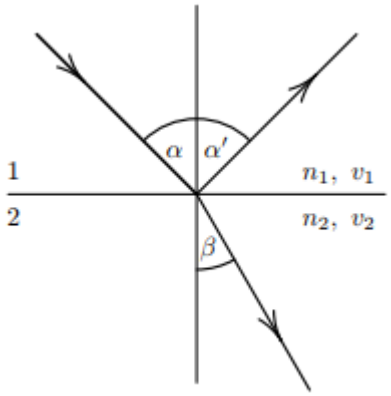
Pokud vyšleme několik rovnoběžných paprsků světla na rozbouřenou hladinu moře (nerovná plocha díky vlnám), tak se každý paprsek odrazí pod trochu jiným úhlem viz OBR. 3. Dochází k tzv. difúznímu odrazu.




**Obr. 3 Difúzní odraz**
Zdroj: Převzato z [2]

## 3.3 Lom světla

Světlo se láme, když přechází z jednoho optického prostředí do jiného. V případě vody se část světla odráží a část se láme viz OBR. 4.




**Obr. 4 Odraz a lom světla na vodní hladině**
Zdroj: Převzato z [1]

### 3.3.1 Index lomu

Index lomu charakterizuje optické prostředí z hlediska šíření světla. Rozlišujeme absolutní a relativní index lomu.

#### 3.3.1.1 Absolutní index lomu

Absolutní index lomu definujeme jako poměr rychlosti šíření světla ve vakuu $c$ a rychlosti $v$ v daném prostředí:

$$n = \frac{c}{v} \tag{3.3}$$

Tato hodnota je charakteristikou konkrétního prostředí (jedná se o materiálovou konstantu) a je vždy větší než 1 (světlo se ve vakuu šíří nejrychleji). Pro řadu pevných látek je index lomu definován ve fyzikálních tabulkách viz TABULKA 1.

**Tabulka 1 Index lomu vybraných látek**

| | |
|---|---|
| Vakuum | 1.0 |
| Led | 1.31 |
| Voda | 1.33 |
| Ethylalkohol | 1.36 |
| Plexisklo | 1.49 |
| Běžné sklo | 1.50 |
| Korunové sklo | 1.52 |
| Flintové sklo | 1.6 až 1.9 |
| Diamant | 2.42 |

Zdroj: Převzato z [1]

#### 3.3.1.2 Relativní index lomu

Relativní index lomu definujeme jako poměr rychlosti šíření světla ve dvou optických prostředích *v1* a *v2*:

$$n_{12} = \frac{v_1}{v_2} \tag{3.4}$$

Nebo lze také definovat jako poměr absolutního indexu druhého prostředí k prvnímu:

$$n_{12} = \frac{n_2}{n_1} \tag{3.5}$$

### 3.3.2 Zákon lomu (Snellův zákon)

Dopadá-li paprsek z prostředí s indexem lomu n1 do prostředí s indexem lomu n2, dochází k lomu paprsku. Lomený paprsek zůstává v rovině dopadu. Úhel dopadu značíme α, úhel lomu značíme β. Pro tyto úhly přitom platí:

$$\frac{\sin \alpha}{\sin \beta} = \frac{n_2}{n_1} \tag{3.6}$$

Lom ke kolmici nastává při přechodu světla z prostředí opticky řidšího do prostředí opticky hustšího, tzn. β < α (viz **OBR. 4**) a lom od kolmice nastává při přechodu světla z prostředí opticky hustšího do prostředí opticky řidšího, tzn. β > α [1].

### 3.3.3 Fresnelův efekt

Francouzský fyzik Augustin-Jean Fresnel pozoroval jev (viz např. **OBR. 5**), který v počítačové grafice známe pod pojmem Fresnelův efekt [3]. Množství odraženého světla závisí na úhlu mezi vektorem pozorování a normálovým vektorem vodní hladiny v daném bodě. Pokud bude tento úhel velký, odráží se skoro všechno světlo – hladina je téměř neprůhledná. Pokud je úhel malý, odráží se malá část světla – hladina je téměř průhledná.

**Obr. 5 Fresnelův efekt, boční okénka odrážejí více světla než přední okno vozidla**

Zdroj: Převzato z [3]

# 4 Osvětlení scény a stínování

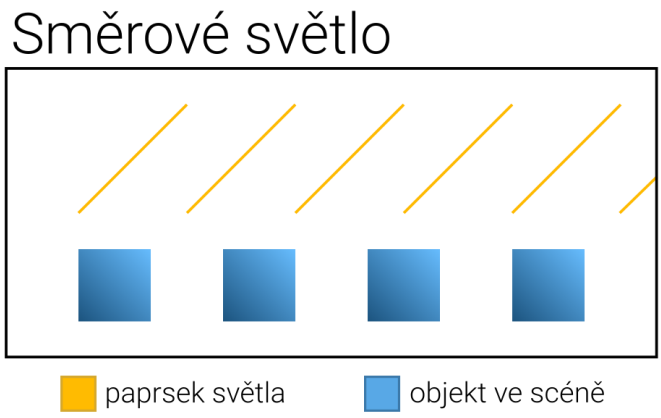
Osvětlení je nedílnou součástí každé scény, protože dodává objektům jistou dávku realističnosti. Pro osvětlení scény budeme potřebovat přidat zdroj světla, který bude vysílat paprsky světla do scény.

## *4.1 Zdroj světla*

Zdroj světla nám bude ve scéně reprezentovat světlo, které můžeme podle chování světelných paprsků rozdělit do 3 kategorií: směrové světlo, bodové světlo a světelný kužel.

### 4.1.1 Směrové světlo (Directional light)

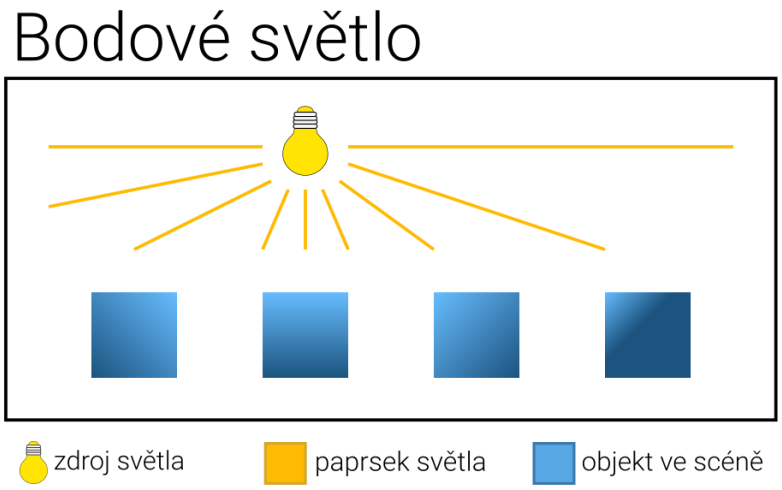
Směrové světlo si můžeme představit jako zdroj světla, který je nekonečně daleko (např. Slunce) [4]. Proto ho můžeme reprezentovat pouze pomocí jednoho směrového vektoru (světelný zdroj je natolik vzdálen, že můžeme počítat s tím, že všechny světelné paprsky jsou rovnoběžné) viz OBR. 6.




**Obr. 6 Směrové světlo**
Zdroj: vlastní zpracování

### 4.1.2 Bodové světlo (Point light)

Bodové světlo má určitou pozici ve scéně. Světlo z této pozice se šíří všemi směry, ale se vzdáleností postupně slábne [5] viz OBR. 7. Pro reprezentaci bodového světla v aplikaci nám bude stačit vektor pozice. Příkladem takového světla je například žárovka.




**Obr. 7 Bodové světlo**
Zdroj: vlastní zpracování

S rostoucí vzdáleností objektu od zdroje bodového světla dochází k útlumu světla:

$$F = \frac{1.0}{K_c + K_i \times d + K_q \times d^2} \tag{4.1}$$

Pro výpočet útlumu světla dle rovnice 4.1 potřebujeme znát vzdálenost d, konstantní hodnotu $K_c$, linearní hodnotu $K_i$ a kvadratickou hodnotu $K_q$. Pokud bychom chtěli světlem pokrýt vzdálenost 100, tak světlu nastavíme konstantní

hodnotu na 1.0, lineární hodnotu na 0.045 a kvadratickou hodnotu na 0.0075 viz TABULKA 2.

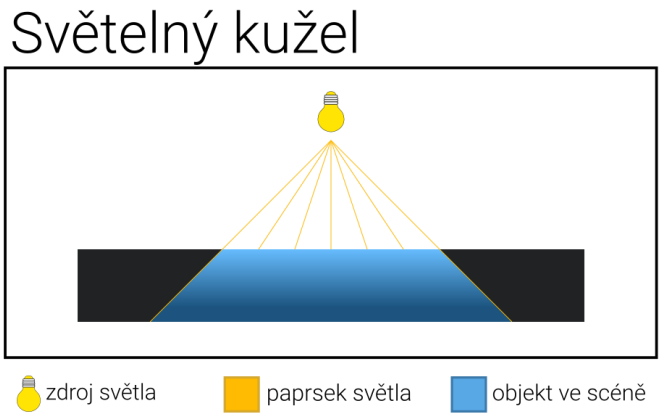
**Tabulka 2 Příklad hodnot pro výpočet útlumu světla**

| Vzdálenost | Konstantní hodnota | Lineární hodnota | Kvadratická hodnota |
|---|---|---|---|
| 7 | 1.0 | 0.7 | 1.8 |
| 13 | 1.0 | 0.35 | 0.44 |
| 20 | 1.0 | 0.22 | 0.20 |
| 32 | 1.0 | 0.14 | 0.07 |
| 50 | 1.0 | 0.09 | 0.032 |
| 65 | 1.0 | 0.07 | 0.017 |
| 100 | 1.0 | 0.045 | 0.0075 |
| 160 | 1.0 | 0.027 | 0.0028 |
| 200 | 1.0 | 0.022 | 0.0019 |
| 325 | 1.0 | 0.014 | 0.0007 |
| 600 | 1.0 | 0.007 | 0.0002 |
| 3250 | 1.0 | 0.0014 | 0.000007 |

Zdroj: Převzato z [6]

### 4.1.3 Světelný kužel (Spotlight)

Světelný kužel vysílá paprsky ve formě kuželu viz OBR. 8. Tedy dochází k osvětlení pouze objektů, které se nachází uvnitř tohoto kuželu. Směrem k okraji kužele intenzita osvětlení exponenciálně klesá [5]. Příkladem může být lampa nebo baterka.



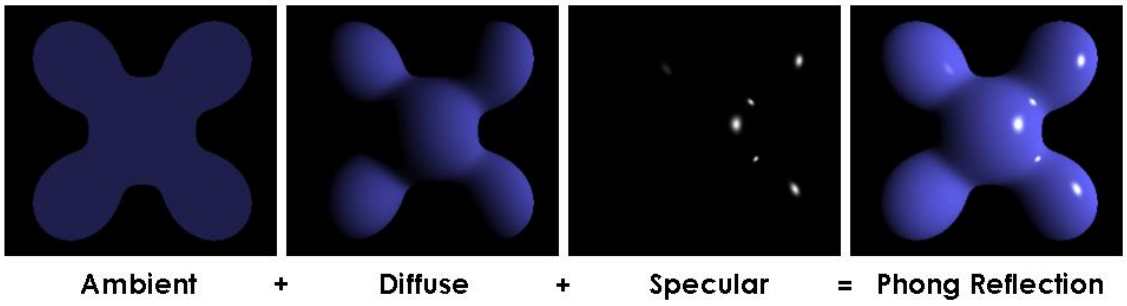

**Obr. 8 Světelný kužel**
Zdroj: vlastní zpracování

## *4.2 Osvětlovací model*

Osvětlovací modely se používají pro výpočet odraženého světla z povrchu objektu. Některé modely mohou být založené na fyzikálních zákonech, ale s tím bohužel přichází vyšší výpočetní náročnost. Častěji se setkáme s osvětlovacími modely, které se snaží fyzikální přesnost pouze aproximovat. Nejčastěji se v počítačové grafice používá Phongův osvětlovací model a vylepšený Blinn-Phongův osvětlovací model.

### 4.2.1 Phongův osvětlovací model

Phongův osvětlovací model představil Bui Tuong Phong v roce 1975 [7]. Tento osvětlovací model popisuje, jak povrch odráží světlo, pomocí 3 složek (viz OBR. 9): ambientní, difúzní a lesklá složka. Ambientní složka popisuje světlo, které je rovnoměrně přítomné v celé scéně. Difúzní složka určuje intenzitu světla, které se od povrchu tělesa rovnoměrně odráží do všech směrů. Nakonec lesklá složka (specular) popisuje intenzitu světla, které se odráží od povrchu dle zákona odrazu.




**Obr. 9 Phongův osvětlovací model**
Zdroj: Brad Smith
https://en.wikipedia.org/wiki/Phong_reflection_model#/media/File:Phong_components_version_4.png

### 4.2.2 Blinn-Phongův osvětlovací model

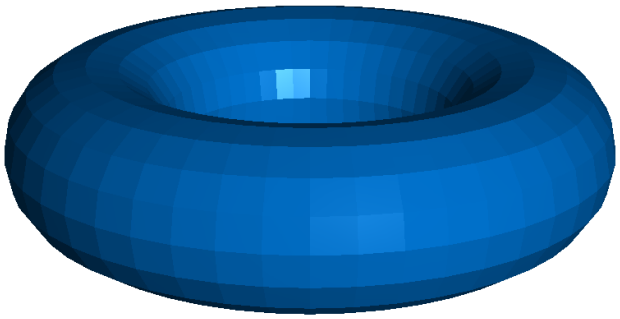
Jim Blinn v roce 1977 popsal aproximaci Phongova osvětlovacího modelu [8]. Při výpočtu Phongova stínování je potřeba stále přepočítávat skalární součin pohledového vektoru a vektoru odraženého světla. Jim Blinn zjistil, že stačí místo toho použít vektor, který se rovná skalárnímu součinu pohledového vektoru a vektoru světla. Díky této aproximaci je algoritmus mnohem rychlejší, pokud počítáme s pozorovatelem, který je nekonečně daleko (reprezentován vektorem pohledu) a směrovým světlem, protože tento vektor stačí vypočítat pouze jednou za renderovací cyklus pro každý zdroj světla.

## *4.3 Stínování*

Stínování, jako u klasické ruční kresby, dodá objektům v počítačové grafice větší hloubku a větší percepci trojrozměrnosti objektu. V počítačové grafice dokážeme toto stínování vypočítat pomocí několika metod.

### 4.3.1 Konstantní stínování (Flat shading)

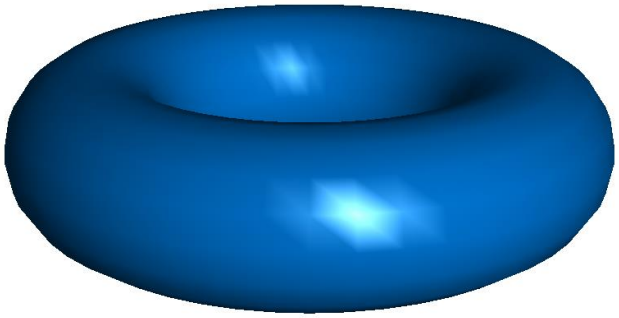
Konstantní stínování je metoda, která je velice rychlá a jednoduchá. Používá se pro stínování rovinných ploch, kde každá tato plocha má jeden normálový vektor [5]. Pro tuto normálu vypočteme barvu, kterou aplikujeme na každý pixel této plochy. Jak můžeme vidět na **Obr. 10**, tak objekt po aplikaci konstantního stínování působí hranatě a nepříliš realisticky.



**Obr. 10 Konstantní stínování**
Zdroj: Maarten Everts
https://en.wikipedia.org/wiki/Shading#/media/File:Flatshading01.png

### 4.3.2 Gouraudovo stínování (Gouraud shading)

Metodu Gouraudovo stínování navrhl Henri Gouraud v roce 1971 [9]. Pro výpočet Gouraudova stínování potřebujeme znát normálový vektor každého vrcholu. Pro každý vrchol vypočteme barevnou hodnotu a ta je následně mezi vrcholy interpolována viz **Obr. 11**.

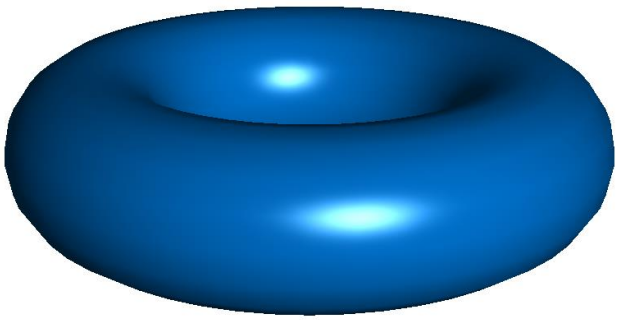


**Obr. 11 Gouraudovo stínování**
Zdroj: Maarten Everts

https://en.wikipedia.org/wiki/Shading#/media/File:Gouraudshading01.png

### 4.3.3 Phongovo stínování (Phong shading)

Společně s Phongovým osvětlovacím modelem představil Bui Tuong Phong ve své práci z roku 1975 také Phongovo stínování [7]. Phongovo stínování je vylepšením Gouraudova stínování a na rozdíl od něj probíhají všechny výpočty v jednotlivých bodech (fragmentech) geometrického primitiva. Pro každý jednotlivý bod je vypočtena barevná hodnota na základě interpolovaného normálového vektoru všech vrcholů viz **OBR. 12** [5].



**Obr. 12 Phongovo stínování**
Zdroj: Maarten Everts

https://en.wikipedia.org/wiki/Shading#/media/File:Phongshading01.png

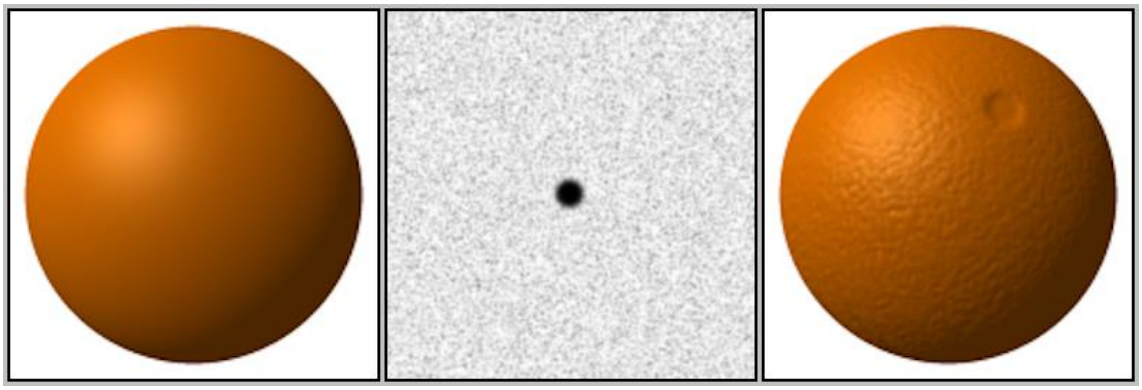
# 5 Historie vývoje renderování vody v počítačové grafice

Počátkem 80. let začínal vývoj v oblasti renderování přírodních jevů v počítačové grafice včetně vody [10]. Ze začátku se renderování vody zkoumalo zejména jako renderování jednolitého celku – oceánu.

## *5.1 Počátky renderování vody 1978 - 1986*

Z počátku se hlavně vycházelo z optických vlastností vody a jak tyto vlastnosti efektivně zobrazit pomocí tehdejších počítačů.

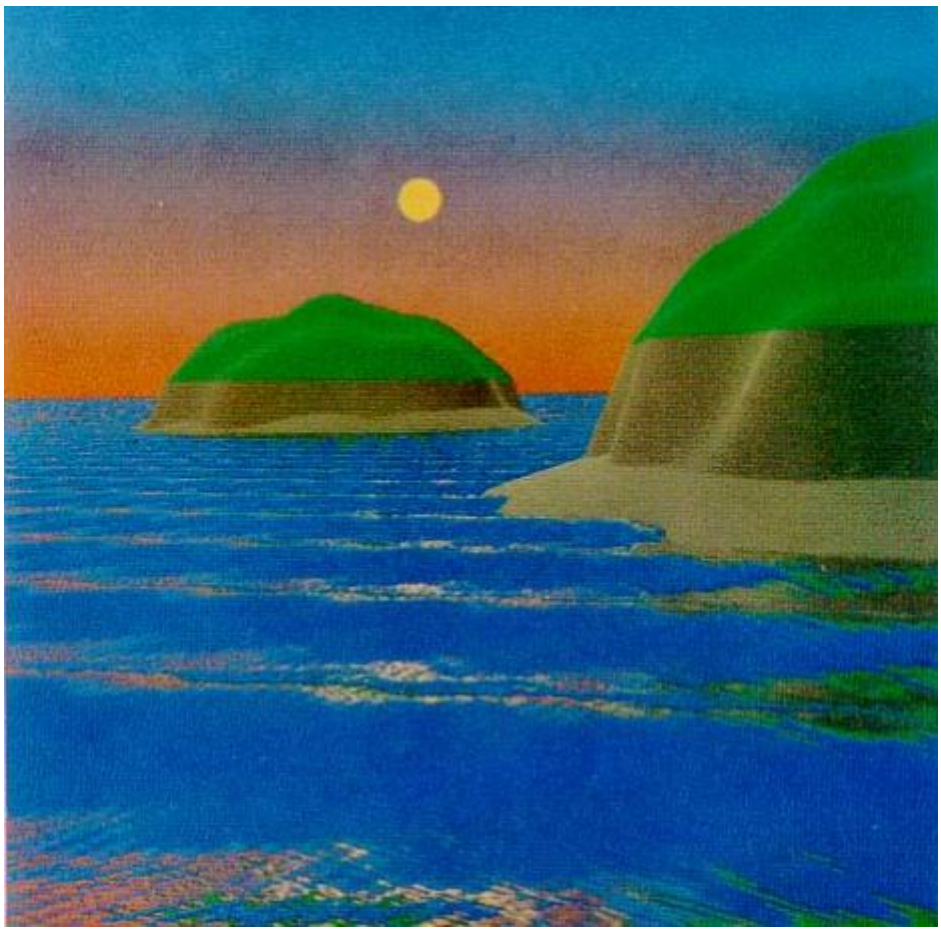
V roce 1978 představil James Blinn metodu bump mappingu, pomocí které se daly renderovat povrchy těles mnohem více realisticky [11]. Pomocí úpravy normál zcela hladkého objektu se změnil výpočet osvětlení a tím vznikla iluze vrásčitého povrchu viz **Obr. 13** (nedochází k žádné změně povrchu tělesa). Na základě jeho práce začaly vznikat první metody pro renderování vln na vodní hladině, které používaly různé funkce pro úpravu normál.



**Obr. 13 Ukázka bump mappingu**
Zdroj: Wikimedia Commons https://commons.wikimedia.org/wiki/File:Bump-map-demo-full.png

Turner Whitted vytvořil v roce 1978 krátkou animaci The Compleat Angler, v které použil bump mapping společně s ray tracingem, díky kterému mohl zobrazit průhlednost, odraz a lom světla [12].

Nelson Max použil pro renderování vln ray tracing v jeho krátkém filmu "Carla´s Island" viz **Obr. 14** [13]. Vlny určil pomocí nelineárních parciálních diferenciálních rovnic, které řešil pomocí aproximace. Tato aproximace byla validní pouze pro vlny s malou amplitudou. Samotný ray tracing využil pro stínování a odraz vodní hladiny.



**Obr. 14 Ukázka z filmu Carla´s Island**
Zdroj: Převzato z [13]

Alain Fournier se společně s William T. Reeves zabývali renderováním vln v oceánu a mimo jiné řešili ve svém modelu lamání vln u břehu [14]. Ten byl založen na Gerstner-Rankine modelu, kde částice vody popisují kruhovité nebo eliptické

stacionární oběžné dráhy. Díky různým parametrům těchto oběžných drah lze produkovat realistické vlny různých tvarů a velikostí. Samotnou plochu oceánu modeloval jako parametrickou plochu. Díky tomu mohly být použity tradiční renderovací metody jako ray tracing a adaptivní dělení. Pomocí částicového systému reprezentoval rozprášení vody a pěnu na vodní hladině při lámání viz OBR. 15.

**Obr. 15 Ukázka vln s rozprášenou vodou a pěnou**
Zdroj: Převzato z [14]

James Arvo ve své práci zkoumal zpětný ray tracing [15]. V klasickém ray tracingu se vysílá paprsek od pozorovatele do prostředí. Poté při každém střetu s překážkou se z tohoto bodu vysílá další paprsek ke každému zdroji světla a na základě toho se vypočte stínování. Pokud bychom chtěli zobrazit odraz a lom světla, tak narážíme na problém, kdy nelze přesně určit nepřímé osvětlení. Proto James Arvo použil zpětný ray tracing, tedy vyslání paprsku od zdroje světla do prostředí. Každému paprsku přidělil určitou část energie, která byla postupně předávána do každého povrchu, od kterého se paprsek odrazil.

Darwyn R. Peachey představil model pro renderování a animaci vln včetně lámání [16]. Pomocí částicových systémů dokázal modelovat rozprášení vody při střetu s překážkami.

## 5.2 Rozmach nových algoritmů 1987 – 1995

V tomto období bylo vymyšleno několik nových algoritmů, které k problému renderování vody přistupovaly jiným způsobem.

Pauline Y. Ts´o a Brian A. Barsky ve své práci prezentovali nové algoritmy pro modelování a renderování vody [17]. Renderovací algoritmy byly založeny na mapování textury odrazu, refrakce a aplikace Fresnelova efektu. Aplikací Fresnelova efektu (viz kapitola **3.3.3**) přiřadil každému bodu na vodní hladině část odražené a lomené barvy na základě vektoru pozorování a normálového vektoru.

Gavin Miller a Andrew Pearce prozkoumávali částicové systémy a jejich využití pro animaci viskózních kapalin [18]. Jeho metoda simulovala síly částic, které interagovaly mezi sebou. Proto může tato metoda detekovat kolize mezi částicemi a ostatními objekty ve scéně.

Michael Kass a Gavin Miller vytvořili novou metodu pro animaci vody na základě jednoduchého, rychlého a stabilního řešení řady parciálních diferenciálních rovnic, které vyplývají z aproximace rovnic mělké vody [19]. Tato nová metoda také dokáže generovat efekty lámání vln u pobřeží, odraz světla a přesun vody. Díky tomu je tento model velice vhodný pro animaci tekoucích řek, vln v akváriu nebo vln na pláži.

Karl Sims zkoumal především částicové systémy, pomocí kterých modeloval různé dynamické jevy jako vítr, sníh a vodu [20]. Ve své animaci Particle Dreams mimo jiné simuloval vodopád (viz **Obr. 16**), který se skládal z modrých částic, na které byla aplikována gravitace. Když se částice dostaly na spodní hranu vodopádu, tak byly recyklovány a vráceny zpět na vrchol vodopádu. Na tuto animaci bylo potřeba zhruba 60 000 částic.

**Obr. 16 Ukázka z animace Particle Dreams**
Zdroj: Převzato z [20]

Mark Watt objevil nový přístup pro zobrazení optických jevů, který byl založen na zpětném ray tracing – zpětný beam tracing [21].

David Tonnesen prezentoval novou metodu pro modelování pevných látek a kapalin, která vychází z molekulární a Newtonské dynamiky [22]. Model popisuje změny v geometrii a pohybu elementárních objemů jako důsledek termální energie a externích sil. Tyto objemy jsou reprezentovány částicovými systémy.

## 5.3 Dosahování působivé realističnosti 1996 – 2004

Lidé se více zaměřovali na dynamický aspekt vody a její realizaci pomocí počítačové grafiky. Proto voda působila stále více realističtěji.

Jean-Christophe Gonzato popsal algoritmus pro renderování pobřežních scén pomocí speciálního osvětlovacího modelu pro scénu a techniky mapování textury posunu pro práci s řadami po sobě jdoucích vln [23]. Ve své práci také představil algoritmus Dynamic Wave Tracing (DWT), který poskytuje geometrické definice pro pobřežní scénu.

Manuel N. Gamito a F. Kenton Musgrave zkoumali lámání vln u pobřeží a představili novou metodu pro výpočet lámání vlny v mělké vodě [24]. Tato metoda je přesnější než předchozí dostupné metody a zobrazuje realistické efekty.

Stefan Jeschke prezentoval procedurální model pro lámání vln [25]. Samotný pohyb a vizualizace vln je modelována řadou funkcí v závislosti na čase a prostoru. Díky tomu lze vypočítat všechny vlastnosti v jakémkoliv bodě povrchu oceánu bez znalosti předchozích časových kroků.

Nathan Holmberg a Burkhard C. Wünsche představili metodu pro modelování turbulentní vody jako jsou rychle tekoucí řeky a vodopády [26]. Ve své metodě využili 2D výškové pole a částicový systém pro modelování hlavního objemu a rozprášení vody. Navíc se toto řešení dynamicky přizpůsobuje jakémukoliv terénu.

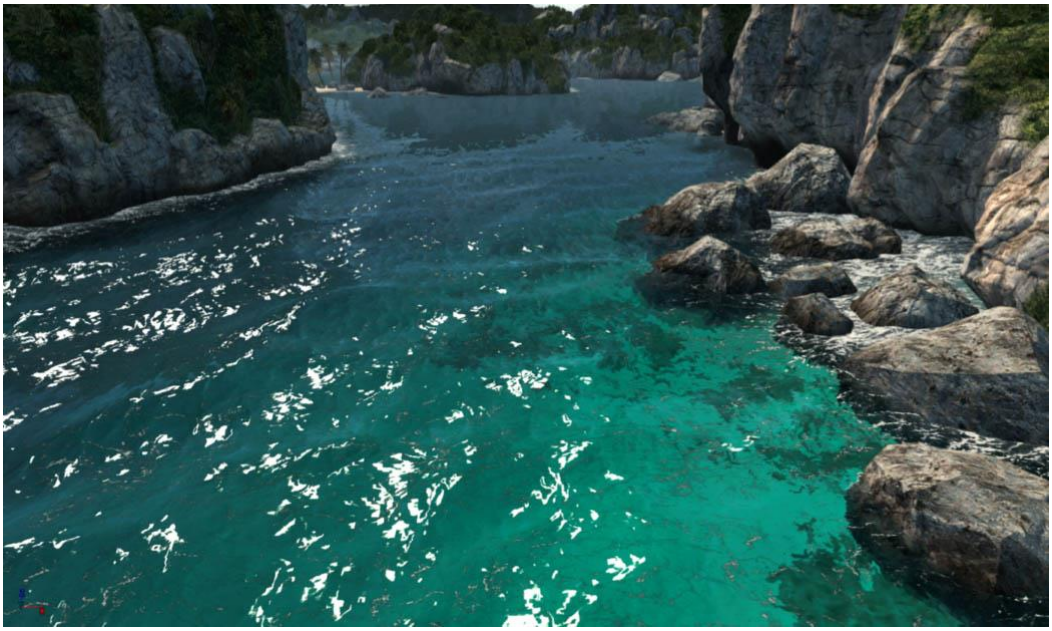
## 5.4 Počítačové hry 2005 až současnost

S dostupností výkonnějších počítačů přišel nástup počítačových her, které potřebovali zobrazit realistickou vodu pro fungování herních mechanik.

Yuri Kryachko se zabýval technikami pro renderování realistického oceánu pro počítačovou hru Pacific Fighters (2005) viz **OBR. 17** [27]. Pro osvětlení použil normálové mapy, pro samotné modelování vody využil výškové mapy.

**Obr. 17 Ukázka ze hry Pacific Fighters**
Zdroj: Převzato z [27]

Počítačová hra Assassins Creed III (2012) (viz **OBR. 18**) se odehrává převážně v USA. V této hře bylo potřeba renderovat oceán, který by dokázal reagovat na vnější podněty jako je například vítr [28]. Proto vývojáři vyvinuli systém pro rozbouření oceánu, který je rozdělen na určité úrovně [28]. Jednotlivé úrovně popisují sílu větru a vlny a každé úrovni přidělili určitá nastavení chování. Pro reprezentaci vln využili statistický model, který vycházel z práce Jerryho Tessendorfa [29].



**Obr. 18 Ukázka z počítačové hry Assassin´s Creed III**
Zdroj: Převzato z [28]

Počítačová hra Far Cry 5 (2018) se odehrává ve státě Montana, kde se vyskytuje mnoho jezer a řek viz **Obr. 19**. Proto bylo velice důležité vytvořit efektivní renderování vodní hladiny a toků, které bude reagovat na okolní prostředí. Algoritmus, který použili, se skládá z několika kroků [30]:

- Zjištění viditelnosti vody (pokud není z pohledu kamery vidět, tak ji vůbec nerenderovat)
- Vodní hladina se skládá z několika menších ploch, uchovávají se normálové vektory a informace o hloubce
- Aplikace rozostření pomocí šumu
- Ořezání pixelů, které nemají vodní hladinu
- Teselace
- Generování normálové mapy a mapy pro vyhlazení
- Zachycení pěny vodní hladiny pomocí textury šumu
- Flow mapa, která byla generována na základě okolního terénu a výšky hladiny
- Nakonec byla vodní hladina osvětlena

**Obr. 19 Ukázka z počítačové hry Far Cry 5**
Zdroj: Převzato z [30]

Pro tuto hru vyvinuli několik užitečných nástrojů, aby mohl grafický designér jednoduše přidávat vodní plochy do herního světa.

# 6 OpenGL

OpenGL (Open Graphics Library) je grafická knihovna pro práci s 2D a 3D grafikou. Vývoj začal v roce 1989 firmou Silicon Graphics Inc. Na přelomu 1990 a 1991 navázala firma Silicon Graphics Inc. spolupráci s firmou Microsoft a roku 1992 byla vydána první verze 1.0 [31].

OpenGL specifikace definuje abstraktní API pro vykreslování 2D a 3D grafiky pomocí několika funkcí a konstant. Uvnitř funguje jako konečný automat, který je definován několika stavy, mezi kterými lze přepínat pomocí funkcí.

OpenGL funkce lze rozdělit do 3 kategorií:

- Funkce, které mění stav
- Funkce, které získávají stav
- Funkce, které renderují podle aktuálního stavu

Rozhraní OpenGL je založeno na architektuře klient-server, kde program (klient) vydává příkazy a grafický adaptér (server) je vykonává [32]. Díky této architektuře je možné, aby program fyzicky běžel na jiném počítači a příkazy se předávaly prostřednictvím počítačové sítě.

Mezi alternativy k OpenGL patří Direct3D a Vulkan. Direct3D je rozhraní pro práci s 3D grafikou, které je plaformně závislé na operačním systému Microsoft Windows. Vulkan je poměrně nové multiplatformní rozhraní pro práci s 3D grafikou. Vstoupilo na trh v roce 2016 a hlavně je zaměřeno na poskytnutí vyššího výkonu a balancovanějšího využití procesoru a grafické karty [33]. Vulkan je stejně jako OpenGL vyvíjen společností Khronos Group a profiluje se jako plnohodnotný nástupce OpenGL.

## 6.1 Reprezentace 3D modelu

Aby mohlo OpenGL správně vykreslovat objekty ve scéně, musíme mu předat informace o samotných objektech. Každý objekt se skládá z vrcholů. Vrcholy můžeme reprezentovat jako vektor o 3 složkách např. $v_0 = (0, 1, 0)$. Na jednotlivé vrcholy můžeme návazat další informace jako jsou normálové vektory $vn_0 = (1,0,0)$ nebo souřadnice textury $vt_0 = (0.25, 0.5)$. Abychom mohli tyto informace použít v OpenGL, vytvoříme si Vertex buffer object.

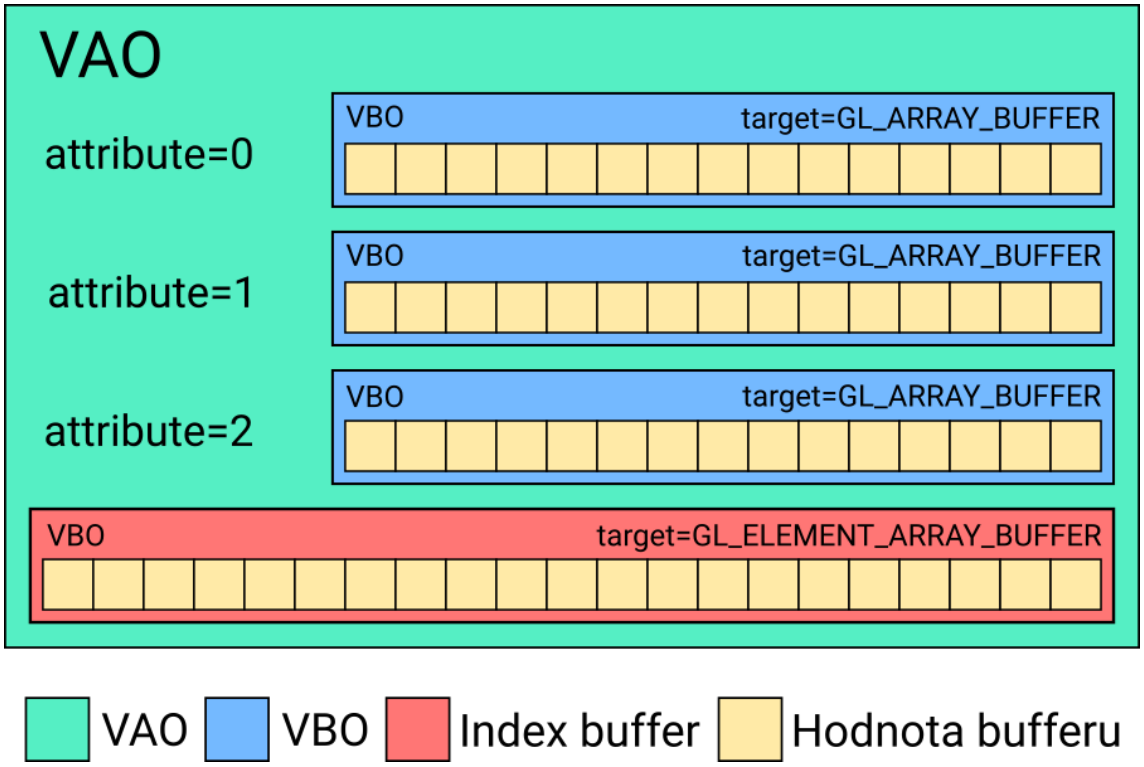
### 6.1.1 Vertex buffer object

Vertex buffer object (VBO) je paměťový buffer, který je uložen na grafické kartě a specificky se používá pro data o vrcholech [34]. Tento buffer můžeme použít na pozici vrcholů, normálové vektory, souřadnice textury a další. Abychom mohli takový buffer vytvořit, je potřeba zavolat metodu glGenBuffers. Pro uložení dat do bufferu použijeme metodu glBufferData, Jako cíl předáme GL_ARRAY_BUFFER (data v bufferu souvisí s vrcholy), velikost dat v bytech, samotná data a na závěr musíme specifikovat použití GL_STATIC_DRAW (data do bufferu nahrajeme pouze jednou a budeme je opakovaně používat, nebudeme je dále měnit). Pro předání indexů vrcholů musíme vytvořit VBO, který bude mít cíl dat nastaven jako GL_ELEMENT_ARRAY_BUFFER. V aplikaci jsem si pro tyto účely vytvořil třídu cz.dennistschamler.opengl.VertexBufferObject, která zapouzdřuje potřebnou funkcionalitu.

### 6.1.2 Vertex array object

Po vytvoření jednotlivých VBO musíme někde definovat, že jednotlivé buffery patří k jednomu objektu a k tomu nám právě poslouží Vertex array object (VAO) [34]. Pro vytvoření VAO použijeme metodu glGenVertexArrays (je možné vytvořit rovnou více VAO najednou) a následně napojíme metodou glBindVertexArray, které

předáme identifikátor VAO. Pro propojení VAO a VBO použijeme atributy, které jsou číslovány od 0 do GL_MAX_VERTEX_ATTRIBS – 1. Do každého atributu lze napojit jeden VBO viz Obr. 20. Abychom mohli začít atribut používat, je nutné jej nejdříve povolit pomocí metody glEnableVertexAttribArray (ve výchozím stavu jsou všechny atributy zakázány). V aplikaci jsem si pro tyto účely vytvořil třídu cz.dennistschamler.opengl.VertexArrayObject, která zapouzdřuje potřebnou funkcionalitu.




**Obr. 20 Vertex array object**
Zdroj: vlastní zpracování

### 6.1.3 OBJ souborový formát pro 3D modely

Pokud chceme používat složitější 3D modely v aplikaci, je nereálné, abychom modely vytvářeli přímo v aplikaci definováním jednotlivých vrcholů. Na internetu lze najít několik 3D modelů, které bychom v aplikaci mohli použít. Pro načtení modelu do aplikace je možné použít několik souborových formátů. Mezi

nejpopulárnější formáty patří Wavefront obj, který byl vytvořen společností Wavefront Technologies [35]. Je to jednoduchý formát pro uchování informací o 3D modelu. Každý řádek začíná specifickým textem, který určuje, jaká informace následuje. Dle specifikace lze definovat mnoho dalších informací, ale pro naše účely si vystačíme pouze s určitou množinou.

#### 6.1.3.1 Vrcholy

Řádek s vrcholem začíná písmenem *v*. Po mezeře následují 3 čísla, která určují souřadnice x, y a z.

*v* 1.000000 1.000000 -1.000000

#### 6.1.3.2 Souřadnice textury

Řádek se souřadnicemi textury začíná písmenem *vt*. Po mezeře následují 2 čísla, která definují souřadnice x a y.

*vt* 0.333333 1.000000

#### 6.1.3.3 Normálový vektor

Řádek s normálovým vektorem začíná písmenem *vn*. Po mezeře následují 3 čísla, která definují souřadnice x, y a z normálového vektoru.

*vn* 0.0000 1.0000 0.0000

#### 6.1.3.4 Stěna trojúhelníků

Na závěr je potřeba všechny informace spojit dohromady a složit objekt z trojúhelníků. K tomu nám slouží řádek, který začíná písmenem *f*. Tento řádek má následující formát:

*f* v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3

Každá skupina reprezentuje jeden vrchol, tedy např. první skupina obsahuje *v1* (index vrcholu), *vt1* (index souřadnice textury) a *vn1* (index normálového vektoru). To nám stačí, abychom mohli pomocí těchto informací vytvořit VAO. Pro každou jednotlivou informaci vytvoříme VBO a ten navážeme na atribut VAO. Momentálně máme ve VAO uložený celý 3D objekt. Abychom tento objekt mohli správně vykreslit na obrazovce, musíme ho transformovat z 3D prostoru na 2D obrazovku.

## 6.2 Zobrazení 3D objektu na 2D obrazovce

Každý 3D objekt je definován souřadnicemi, které nazýváme lokální. Abychom mohli tento objekt na obrazovce vykreslit, je nutné jej převést z lokálních souřadnic na souřadnice obrazovky a toho docílíme aplikací několika postupných transformací.

Transformace pro 3D objekty musíme reprezentovat $4 \times 4$ maticemi (více o maticích v [36]). Prvním krokem je umístění objektu do souřadnic světa pomocí modelové transformace.

### 6.2.1 Modelová transformace

Většina 3D modelů po exportu z 3D programu má svůj střed nastaven na souřadnice (0,0,0). Pokud takový objekt použijeme ve scéně, bude se nacházet na této pozici. To je ovšem nežádoucí, protože pozici objektu si potřebujeme určit sami. Proto musíme použít modelovou transformaci, díky které můžeme objekt škálovat, rotovat ho

kolem všech os a přesunout na určitou pozici ve světě. Pro reprezentaci modelové transformace je potřeba vytvořit následující matice: translační, rotační a škálovací. Vynásobením těchto tří matic (násobení matic je komutativní – záleží na pořadí [36]) dostaneme modelovou transformaci. Pro tyto účely má knihovna JOML velmi užitečné funkce, které nám pomůžou ve vytváření těchto matic.

V samotné aplikaci je vytvořena třída cz.dennistschamler.entity.component.Transform, která reprezentuje tuto modelovou transformaci. Třída obsahuje 3 atributy: Vector3f position, Vector3f rotation a Vector3f scale a metodu getTransformationMatrix, která vytváří modelovou matici násobením translační, škálovací a rotační matice.

#### 6.2.1.1 Translační matice

Translační matice zajišťuje změnu všech pozic vrcholů objektu stejným směrem a stejnou vzdáleností viz rovnice 6.1 [5].

$$T(t) = T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.1}$$

V knihovně JOML má třída Matrix4f metodu translate, která nám translační matici vytvoří.

#### 6.2.1.2 Rotační matice

Pomocí rotační matice můžeme rotovat objekt o určitý úhel v radiánech kolem os x, y a z viz rovnice 6.2, 6.3 a 6.4 [5]. Násobením matice $R_x$, $R_y$ a $R_z$ získáme rotační matici kolem všech os (nejdříve rotace podle osy x, poté osy y a nakonec osy z).

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.2}$$

$$R_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.3}$$

$$R_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.4}$$

V knihovně JOML má třída Matrix4f metodu rotateXYZ, která provede rovnou rotaci kolem všech os a získáme rotační matici.

#### 6.2.1.3 Škálovací matice

Tato matice nám zajistí zmenšení nebo zvětšení objektu viz rovnice 6.5 [5]. Pokud použijeme vektor škály (1.0, 1.0, 1.0), tak k žádnému škálování nedochází.

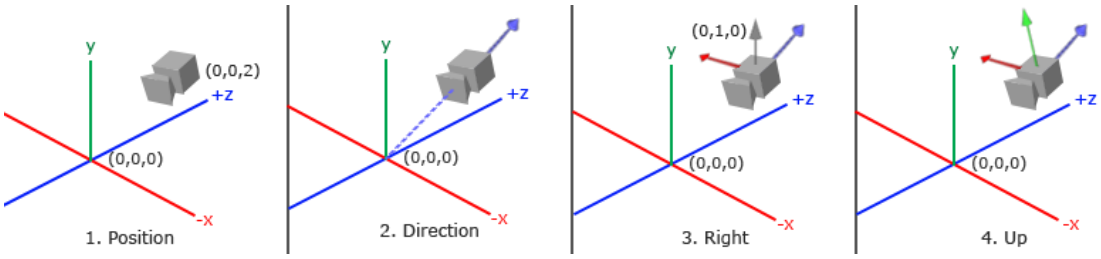
$$S(s) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.5}$$

V knihovně JOML má třída Matrix4f metodu scale, která nám škálovací matici vytvoří.

### 6.2.2 Pohledová transformace

Pohledovou transformaci lze reprezentovat pomocí matice, která zajišťuje převedení ze souřadnic světa na souřadnice pohledu. Tedy všechny souřadnice ve

světě jsou nyní v souřadnicích z pohledu kamery. Abychom mohli pohledovou matici vytvořit, potřebujeme znát následující vektory: vektor pozice, vektor cíle, vektor, který míří nahoru a vektor, který míří vpravo [5]. V samotné aplikaci budeme tyto vektory reprezentovat virtuální kamerou viz OBR. 21. V knihovně JOML zajišťuje metoda lookAt třídy Matrix4f vytvoření pohledové matice.




**Obr. 21 Kamera s potřebnými vektory**
Zdroj: Převzato z [6]

### 6.2.3 Promítání

Proces promítání transformuje 3D objekt na 2D reprezentaci pro vykreslení na obrazovce [5]. Transformací na 2D reprezentaci ztrácíme prostorovou informaci 3D objektu. Díky této transformaci se dostaneme z pohledových souřadnic na souřadnice ořezávání. Promítání můžeme reprezentovat pomocí rovnoběžného nebo středového promítání.

#### 6.2.3.1 Rovnoběžné promítání

Rovnoběžné promítání je jedním z nejjednodušších promítání [5]. Objekty mají stejnou velikost bez ohledu na vzdálenost od kamery [4]. Je velice vhodné pro scény, které vyžadují zachycení skutečných rozměrů objektů. Pro reprezentaci této projekce lze vytvořit ortografickou matici v knihovně JOML pomocí metody ortho třídy Matrix4f, které předáme rozměry pravoúhlého hranolu, který bude reprezentovat pohledový objem.

#### 6.2.3.2 Středové promítání

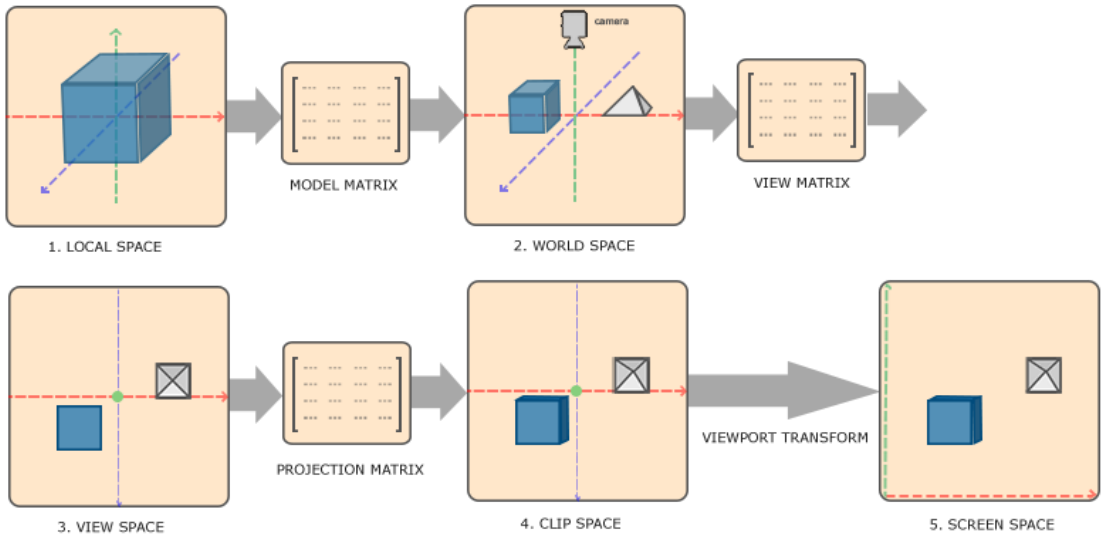
Středové promítání zachycuje objekty více realisticky, tedy s rostoucí vzdáleností od pozorovatele se objekty zmenšují [4]. Toto promítání není vhodné pro objekty, u kterých potřebujeme zachovat původní rozměry. Pro reprezentaci tohoto promítání nám poslouží perspektivní matice, kterou lze v knihovně JOML vytvořit pomocí metody perspective třídy Matrix4f. Této metodě předáme hodnotu zorného pole v radiánech (tato hodnota musí být větší než nula a menší než π), poměr stran plátna, vzdálenost od bližší roviny řezu a vzdálenější roviny řezu (reprezentováno komolým kuželem). Zpravidla se používá pro renderování realistické 3D scény.

### 6.2.4 Aplikace transformací

Pro aplikaci transformací budeme muset matice mezi sebou násobit. Velice zde záleží na pořadí násobení matic, protože násobení matic je komutativní [36]. Pro zobrazení 3D objektu na 2D obrazovce (viz **OBR. 22**) je potřeba provést následující transformace:

1. Převést objekt z lokálních souřadnic na souřadnice světa pomocí modelové transformace
2. Pomocí pohledové transformace získat souřadnice pohledu
3. Vhodným promítáním promítnout objekty do souřadnic ořezávání. Tyto souřadnice jsou dále zpracovány v ořezávacím kroku zobrazovacího řetězce a jsou v rozsahu -1.0 a 1.0 viz kapitola **6.3.4.2**.
4. Převedení ze souřadnic ořezávání na souřadnice obrazovky. Je potřeba se dostat z rozsahu -1.0 a 1.0 na rozměry, které jsou definovány metodou glViewport.
5. Nyní máme souřadnice obrazovky, které jsou následně odeslány k rasterizaci viz kapitola **6.3.6**.

Aplikace modelové, pohledové a projekční transformace probíhá zpravidla ve vertex shaderu viz kapitola 6.3.3.1.

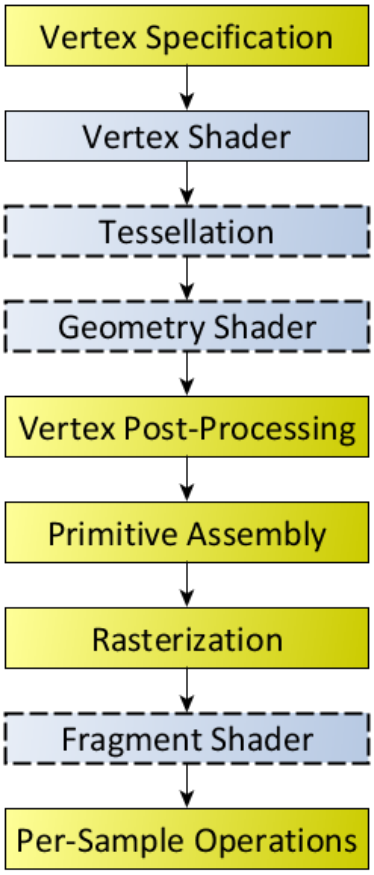



**Obr. 22 Transformace z lokálních souřadnic na souřadnice obrazovky**
Zdroj: Převzato z [6]

## 6.3 Zobrazovací řetězec

Popisuje kroky, které je potřeba provést, aby grafický systém mohl vyrenderovat 3D scénu na 2D obrazovku. Každá grafická knihovna má svůj vlastní zobrazovací řetězec, který se může lišit v určitých krocích. Jednotlivé kroky jsou velice závislé na softwaru a hardwaru, proto nemůže existovat žádný univerzální zobrazovací řetězec, který by obsáhl všechny možné situace.

OpenGL zobrazovací řetězec se skládá z několika kroků viz Obr. 23. Ke spuštění zobrazovacího řetězce dochází zahájením OpenGL renderování například metodou glDrawElements. Určité části zobrazovacího řetězce jsou programovatelné, tedy programátor může definovat chování v daných částech. Tyto programovatelné kroky jsou reprezentovány tzv. shader programy.




**Obr. 23 OpenGL zobrazovací řetězec, modré obdélníky jsou programovatelné shadery**
Zdroj: Převzato z [34]

### 6.3.1 Shader programy

Shader programy jsou izolované programy, které jsou spouštěny na grafické kartě [28]. Shadery mezi sebou nemůžou přímo komunikovat, ale shader může do vstupní proměnné následujícího shaderu poslat proměnnou pomocí výstupní proměnné viz 6.4.2 a 6.4.3.

V knihovně JOGL je pomocná třída ShaderProgram, která poskytuje základní práci s shader programem včetně přidávání shaderů, linkování programu, validace programu atd. Samotné vytváření a parsování shaderu probíhá pomocí statické metody create třídy ShaderCode. V aplikaci jsem funkcionalitu zapouzdřil do třídy

cz.dennistschamler.opengl.shader.MyShaderProgram, která mimo jiné umožňuje načítání uniform proměnných (viz kapitola 6.4.4) datového typu vektor, matice a dalších.

### 6.3.2 Upřesnění vrcholů

Pro předání informací o objektech do OpenGL je nejdříve nutné vytvořit VAO s VBO bufferem na pozice vrcholů a index bufferem pro specifikaci indexů viz 6.1. Před začátkem renderování stačí tento VAO objekt napojit a správně OpenGL informovat o typu geometrických primitiv ve VAO. O jaký typ geometrických primitiv se jedná, můžeme definovat pomocí metody glDrawElements, které předáme mód renderování (většinou GL_TRIANGLES – každé 3 vrcholy tvoří 1 trojúhelník). V tento moment je každý jednotlivý vrchol posílán dál do vertex shaderu k dalšímu zpracování.

### 6.3.3 Zpracování vrcholu

Ve fázi zpracování vrcholu může docházet k manipulaci vrcholů (změna pozice, přidávání vrcholů, odebírání vrcholů atd.). Pro zpracování musejí být vrcholy načteny z VAO (viz kapitola 6.1.2) a následně jsou odeslány do vertex shaderu.

#### 6.3.3.1 Vertex shader

Vertex shader je shader, který zpracovává každý jednotlivý vrchol. Je to první shader, který je přímo pod kontrolou programátora [4]. Spouští se pro každý jeden vrchol a nemá informace o tom, k jakému geometrickému primitivu vrchol patří. Každý vrchol může nést dodatečné informace jako je pozice, souřadnice textury, normály a další. Ve vertex shaderu většinou dochází k transformaci vrcholu do souřadnic ořezávání pomocí modelové matice, pohledové matice a projekční matice

viz kapitola 6.2. Výstupem vertex shaderu je upravený vrchol, který předáme do výstupní OpenGL proměnné gl_Position. Níže je uveden příklad vertex shaderu.

```
#version 400 core

in vec3 in_position;
in vec2 in_textureCoords;
in vec3 in_normal;

out vec2 pass_textureCoords;

uniform mat4 u_transformationMatrix;
uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;

void main(void) {
    gl_Position = u_projectionMatrix * u_viewMatrix *
u_transformationMatrix * vec4(in_position, 1.0);
    pass_textureCoords = in_textureCoords;
}
```

#### 6.3.3.2 Teselace

Tato fáze následuje hned po vertex shaderu a můžeme ji dále rozdělit na 3 fáze: Tesselation control shader, Tesselation primitive generator a Tesselation evaluation shader [34]. Teselace dokáže zvýšit detail objektu díky změně geometrie objektu. Například pokud je objekt blízko kamery, můžeme pro objekt použít větší množství trojúhelníků pro vykreslení a naopak, pokud je objekt daleko od kamery (není potřeba detailní model), tak stačí menší množství trojúhelníků. Navíc lze pomocí teselace celkově optimalizovat scénu a na méně výkonném hardwaru vykreslovat méně trojúhelníků.

#### 6.3.3.3 Geometry shader

Geometry shader dokáže měnit geometrii objektu pomocí přidávání nebo ubírání vrcholů. Tento shader je podporován od verze OpenGL 3.2 [34]. Do geometry shaderu vstupuje geometrické primitivum a výstupem je žádné nebo více

geometrických primitiv. Tedy můžeme zde vrcholy přidávat, ubírat, na vstup dostat jedno geometrické primitivum a na výstup odeslat zcela jiné.

### 6.3.4 Následné zpracování vrcholu

Na vrcholy z kroku zpracování vrcholu jsou dále aplikovány další operace.

#### 6.3.4.1 Transform feedback

V tomto kroku lze zachytit geometrická primitiva, které dostaneme z předešlého kroku. Ty můžeme uložit do bufferů a data z bufferů znovu používat bez potřeby provádět předešlé kroky stále dokola. Pro použití potřebujeme tzv. Feedback object a speciální nastavení shaderu [34]. Poté lze pomocí metody glDrawTransformFeedback renderovat přímo z tohoto bufferu.

#### 6.3.4.2 Ořezávání

Ve fázi ořezávání se pracuje s pozicí vrcholu, která je v souřadnicích ořezávání. Zde dochází k ořezávání objektu. Pokud se objekt nenachází v pohledovém objemu, je ořezán a není dále zpracováván. Pokud se objekt nachází na hranici pohledového objemu, je objekt rozdělen na části podle pohledového objemu. Části, které jsou mimo pohledový objem nejsou dále zpracovávány.

Uživatel si může definovat vlastní ořezávání pomocí float pole gl_ClipDistance, které je dostupné ve vertex shaderu jako výstupní proměnná [34]. Před použitím každého indexu je nutné jej nejprve vysloveně aktivovat pomocí metody glEnable a předat příslušnou konstantu GL_CLIP_DISTANCE*i*, kde *i* nabývá hodnot 0 až GL_MAX_CLIP_DISTANCES - 1. Během ořezávací fáze se projde každá hodnota v poli gl_ClipDistance. Pokud je hodnota pozitivní, tak se geometrické primitivum nachází v ořezávacím prostoru, pokud negativní, tak nikoliv.

### 6.3.5 Sestavení geometrických primitiv

V tomto kroku je potřeba z vrcholů sestavit řadu geometrických primitiv. Výstupem tohoto kroku je seřazená řada geometrických primitiv např. čáry, body nebo trojúhelníky. Následně jsou odeslány k rasterizaci.

Během tohoto procesu mimo jiné dochází i k tzv. face cullingu (odebrání nežádoucí stěny, kterou by pozorovatel stejně nemohl vidět) [34]. U trojúhelníků záleží na pořadí definování 3 vrcholů (ve směru hodinových ručiček nebo proti směru hodinových ručiček) a toto pořadí poté rozhoduje o tom, jestli stěna míří k pozorovateli nebo ne. Toto chování lze definovat pomocí OpenGL metody glFrontFace. Poté můžeme pomocí metody glCullFace nastavit, jak se ke stěnám trojúhelníku zachovat. Ve výchozím stavu je nastaven culling na zadní stěnu, tedy pokud stěna nemíří k pozorovateli, je v tomto kroku odstraněna z dalšího zpracování.

### 6.3.6 Rasterizace

V této fázi dochází k rasterizaci geometrického primitiva tj. nalezení všech pixelů, které se nacházejí uvnitř geometrického primitiva [4]. Výstupem tohoto kroku jsou poté jednotlivé pixely neboli fragmenty. Po rasterizaci jsou tyto pixely odeslány dále do fragment shaderu.

### 6.3.7 Fragment shader

Fragment shader získá z kroku rasterizace pixel. Každé geometrické primitivum je zpracováno a zjišťuje se, jaké pixely to které primitivum pokrývá [4]. Část geometrického primitiva, která překrývá pixel, se nazývá fragment.

Povinnou výstupní proměnnou fragment shaderu je datového typu vec4, která reprezentuje výslednou barvu fragmentu v RGBA formátu. Tuto barvu často reprezentuje barva texelu (pixel textury) viz kapitola 6.5.7. Fragment shader je volitelný. Pokud nebude žádný fragment shader použit, bude barva fragmentu nedefinovaná. Níže je příklad fragment shaderu, který má nastavenou výstupní proměnnou out_Color na červenou barvu.

```
#version 400 core

out vec4 out_Color;

void main(void) {
    out_Color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

### 6.3.8 Per-sample processing

V této fázi máme dostupný fragment, který můžeme podrobit několika operacím. Lze zapnout tzv. testy, které při selhání fragment vyřadí. Tyto testy můžeme dokonce aktivovat ještě před samotným fragment shaderem (tedy nebude zpracováván zbytečně fragment ve fragment shaderu, který bychom následně na základě testu stejně vyřadili) [34].

#### 6.3.8.1 Pixel ownership test

V tomto testu se testuje vlastnictví pixel OpenGL. Pokud například jiné okno aplikace zakrývá naši aplikaci, tak pixely, které překrývají naši aplikaci, již nepatří OpenGL a

tedy fragmenty, které jsou na této pozici mohou být vyřazeny. Tento test se týká pouze výchozího frame bufferu v (v FBO všechny fragmenty tímto testem projdou) viz kapitola 6.6.1.

#### 6.3.8.2 Scissor test

Pomocí OpenGL metody glScissor lze definovat obdélník v souřadnicích okna. Pokud se fragment nachází mimo tento obdélník, je vyřazen.

## 6.4 GLSL

GLSL (OpenGL Shading Language) je programovací jazyk určený pro psaní shaderů, který je velice podobný jazyku C. Je určený přímo pro práci s grafikou a obsahuje užitečné funkce pro manipulaci s vektory a maticemi. V OpenGL je tento programovací jazyk dostupný od verze 2.0 [34]. Aktuálně nejnovější verze GLSL je 4.60.7 [37]. Mezi hlavní výhody určitě patří kompatibilita napříč operačními systémy konkrétně GNU/Linux, macOS a Windows.

Každý shader má určité náležitosti co se týče struktury programu. Na začátku samotného souboru je nutné specifikovat použitou verzi GLSL např. #version 460 (použití verze GLSL 4.60.5) [32]. Pokud verzi v shaderu vůbec neuvedeme, bude nastavena na hodnotu 110 (použití verze GLSL 1.10.59) [34]. Po uvedení verze následuje výčet vstupních, výstupních a uniform proměnných. Poté mohou následovat funkce a nakonec nejdůležitější funkce main, která je hlavním vstupním bodem shaderu.

Soubor, který obsahuje kód shaderu, nemá zpravidla jasně danou příponu. Zpravidla se pro vertex shader používá koncovka vert a pro fragment shader koncovka frag. Ale více méně na tom nezáleží, našemu program bude z tohoto souboru pouze načítat veškeré informace jako text.

### 6.4.1 Datové typy

GLSL poskytuje několik základních datových typů, které najdeme téměř v každém programovacím jazyku. Navíc má definovány datové typy pro vektory a matice, které jsou velice důležité pro prácí s 3D grafikou. Poté ještě existují datové typy pro práci s texturami a obrázky.

**Tabulka 3 GLSL skalární datové typy**

| | |
|---|---|
| void | Pro funkce, které nevrací žádnou hodnotu |
| bool | Pravdivostní hodnota true nebo false |
| int | 32-bit celé číslo se znaménkem |
| uint | 32-bit celé číslo bez znaménka |
| float | 32-bit celé číslo s plovoucí desetinnou čárkou |
| double | 64-bit číslo s plovoucí desetinnou čárkou |

Zdroj: Převzato z [37]

**Tabulka 4 GLSL vektorové datové typy**

| Číslo *n* na konci datového typu může nabývat hodnot 2, 3 nebo 4. Toto číslo definuje, kolik složek vektor obsahuje. | |
|---|---|
| bvec*n* | Vektor datového typu bool |
| ivec*n* | Vektor datového typu int |
| uvec*n* | Vektor datového typu uint |
| vec*n* | Vektor datového typu float |
| dvec*n* | Vektor datového typu double |

Zdroj: Převzato z [37]

**Tabulka 5 GLSL maticové datové typy**

| | |
|---|---|
| mat*n*x*m* | Matice o *n* sloupcích a *m* řádcích. Číslo n a m může nabývat hodnot 2, 3 nebo 4. |
| mat*n* | Matice o *n* sloupcích a *n* řádcích. Číslo *n* může nabývat hodnot 2, 3 nebo 4. Je to zkrácený zápis pro mat*n*x*n*. |

Zdroj: Převzato z [37]

### 6.4.2 Vstupní proměnné

Vstupní proměnnou definujeme pomocí klíčového slova in, poté následuje typ proměnné a nakonec název. Název proměnné se musí shodovat s názvem výstupní proměnné předchozího shaderu.

```
in vec3 in_position;
```

Pokud potřebujeme získat atribut z VAO pro vertex shader, musíme použít metodu glBindAttribLocation, které předáme id shader programu, číslo atributu a název vstupní proměnné v shaderu. Tato metoda zajistí propojení proměnné z aplikace do shaderu a následně budeme mít tuto proměnnou dostupnou v shaderu.

### 6.4.3 Výstupní proměnné

Výstupní proměnnou definujeme pomocí klíčového slova out, poté následuje typ proměnné a nakonec název. Název proměnné se musí shodovat s názvem vstupní proměnné nadcházejícího shaderu. Fragment shader má povinnou výstupní proměnnou typu vec4, která reprezentuje konečnou barvu fragmentu.

```
out vec4 out_color;
```

### 6.4.4 Uniform proměnné

Uniform proměnnou definujeme pomocí klíčového slova uniform, poté následuje typ proměnné a nakonec název. Jedná se o proměnnou, která je určená pouze pro čtení, proto její hodnota už nemůžu být v shaderu změněna. Pro zapsání uniform proměnných z aplikace do shaderu slouží OpenGL funkce glUniform, která má různé varianty a přetížení pro datové typy.

```
uniform mat4 u_projectionMatrix;
```

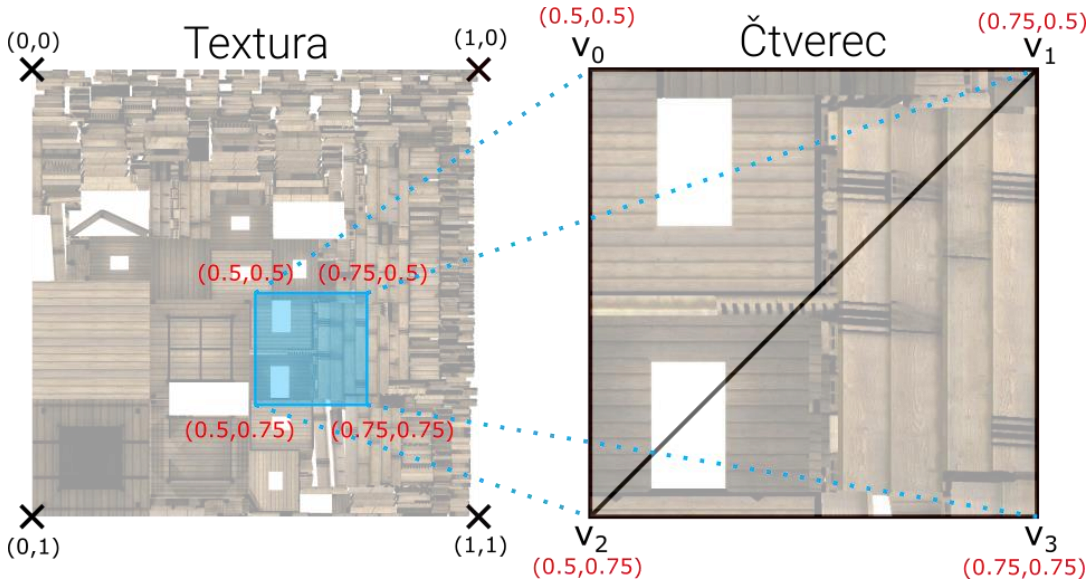
## *6.5 Textury*

Textura je 2D obrázek, který můžeme aplikovat na jakýkoliv 3D model. Je doporučováno používat textury, které mají následující rozlišení v pixelech $2^n \times 2^m$, $n, m \in \mathbb{N}$. Textura je tvořena pixely, které budeme v kontextu textury nazývat texely. Pro vytvoření textury použijeme metodu glGenTextures (může vytvořit rovnou více textur najednou). Po napojení vytvořené textury pomocí glBindTexture předáme data o textuře pomocí metody glTexImage2D. Knihovna JOGL nám práci s texturami výrazně zjednodušuje pomocí utility třídy TextureIO, která zapouzdřuje tyto OpenGL funkce. Díky statické metodě newTexture si načteme texturu do OpenGL a získáme instanci třídy Texture, která je připravena pro použití v aplikaci. Abychom tuto texturu mohli správně nanést na objekt, je nutné správně nadefinovat souřadnice textury.

### 6.5.1 Souřadnice textury

Souřadnice textury jsou standardně definovány v rozmezí (0,0) až (1,1), kde (0,0) je levý horní roh textury a (1,1) pravý spodní roh textury. Pro správné mapování textury je potřeba každému vrcholu 3D modelu přidělit texturu souřadnice.

Fragment shader poté zajistí interpolaci těchto souřadnic pro každý fragment a získáváme otexturovaný objekt viz OBR. 24.




**Obr. 24 Souřadnice textury**
Zdroj: vlastní zpracování, použitá textura Dennis Haupt https://free3d.com/3d-model/watch-tower-made-of-wood-94934.html

### 6.5.2 Formát obrázku

Formát obrázku popisuje způsob, jak obrázky uchovávají svá data v texturách nebo render bufferech [34]. Existují tří základní typy: color, depth a depth/stencil formát obrázku viz TABULKA 6.

.

**Tabulka 6 Základní interní formáty obrázku**

| Základní interní formát | RGBA, depth a stencil hodnoty | Interní komponenty |
|---|---|---|
| GL_DEPTH_COMPONENT | Depth | D |
| GL_DEPTH_STENCIL | Depth, Stencil | D, S |
| GL_RED | Red | R |
| GL_RG | Red, Green | R, G |
| GL_RGB | Red, Green, Blue | R, G, B |
| GL_RGBA | Red, Green, Blue, Alpha | R, G, B, A |

Zdroj: Převzato z [32]

### 6.5.3 Texture wrapping

Souřadnice textury jsou standardně definovány v rozmezí (0,0) až (1,1). Pokud se dostanou mimo tyto hranice, tak OpenGL ve výchozím nastavení texturu opakuje.

Pro jiné nastavení je možné použít OpenGL funkci glTexParameteri, která má 3 parametry: cíl (ve většině případech GL_TEXTURE_2D), typ parametru (pro wrapping lze použít GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_WRAP_R) a nakonec hodnota viz TABULKA 7.

**Tabulka 7 Hodnoty pro texture wrapping**

| | |
|---|---|
| GL_CLAMP_TO_EDGE | Pokud je souřadnice mimo rozsah, je nastavena na nejbližší hraniční hodnotu. Tedy dochází k roztažení hrany textury. |
| GL_CLAMP_TO_BORDER | Pokud je souřadnice mimo rozsah, tak na těchto souřadnicích lze pomocí GL_TEXTURE_BORDER_COLOR nastavit barvu. |
| GL_REPEAT | Výchozí. Textura se opakuje. |
| GL_MIRRORED_REPEAT | Textura se opakuje, po každém opakování se textura zrcadlí. |

Zdroj: vlastní zpracování

### 6.5.4 Filtrování textury

Při texturování objektu se často stane, že máme velký objekt, ale malé rozlišení textury. OpenGL musí každou souřadnici textury správně namapovat na pixel textury. Pro tyto účely se používá filtrování textury, které definuje, jak správně vybrat pixel textury podle souřadnice [6]. V OpenGL je implementováno několik algoritmů pro filtrování textury viz TABULKA 8.

**Tabulka 8 OpenGL algoritmy pro filtrování textury**

| | |
|---|---|
| GL_NEAREST | Je vybrán texel se středem, který je nejblíže souřadnici textury |
| GL_LINEAR | Interpolovaná hodnota sousedících texelů |

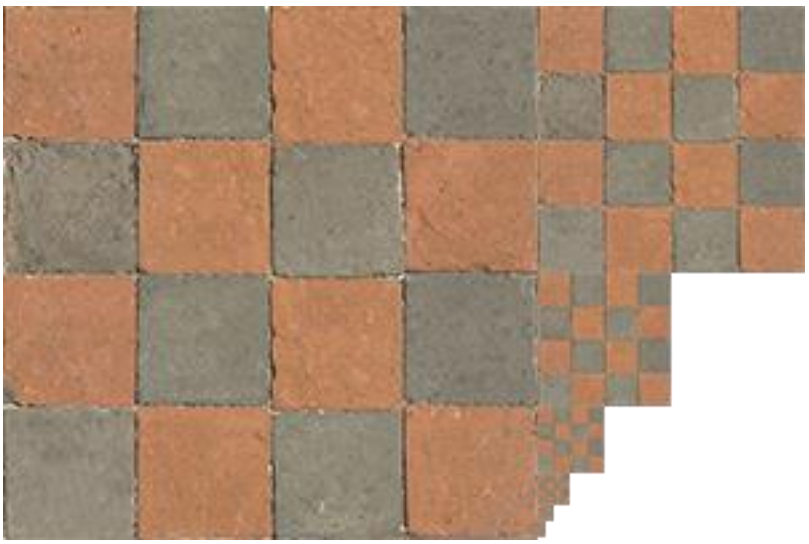
Zdroj: vlastní zpracování

Filtrování textury můžeme rozdělit do 2 kategorií: minifikace a magnifikace. Pokud je texturovaný objekt blízko pozorovateli, jeden texel je větší než pixel obrazovky – je nutné provést filtrování textury pro magnifikaci. Naopak, pokud je objekt daleko od pozorovatele, jeden texel je menší než pixel obrazovky – je nutné provést filtrování textury pro minifikaci. Filtrování textury pro minifikaci lze vyřešit pomocí mipmappingu.

### 6.5.5 Mipmapping

Mipmapping je velice užitečná technika pro optimalizaci vykreslování. Pokud u textury zapneme generování mipmapy pomocí glGenerateMipmap, tak OpenGL vytvoří různá rozlišení dané textury [6]. Získáváme tím naši původní texturu a poté několik textur menšího rozlišení. OpenGL tyto menší textury vytváří čtvrcením předešlé textury, dokud nedosáhne velikosti $1px \times 1px$ viz **Obr. 25**.

$$128 \times 128 \rightarrow 64 \times 64 \rightarrow 32 \times 32 \rightarrow 16 \times 16 \rightarrow 8 \times 8 \rightarrow 4 \times 4 \rightarrow 2 \times 2 \rightarrow 1 \times 1$$



**Obr. 25 Textura, která má vygenerové mipmapové textury**
Zdroj: Převzato z [6]

Poté můžeme na základě vzdálenosti od kamery použít adekvátní texturu při aplikaci filtrování textury:

- Textura se nachází blízko kamery, použije se vysoké rozlišení textury
- Textura se nachází daleko od kamery, použije se nízké rozlišení textury

Bez použití mipmapy může docházet k vizuálním artefaktům, pokud je textura s vysokým rozlišením daleko od kamery (texel je menší než pixel, nelze přesně určit barvu texelu, protože objekt, který je texturován, je velice malý).

Pro texturu, která má vygenerovanou mipmapu, můžeme definovat chování výběru mipmapy (viz **TABULKA 9**) pro situace, kdy je pixel větší než jeden texel nebo menší než jeden texel.

**Tabulka 9 Algoritmy pro výběr správné mipmapy textury**

| | |
|---|---|
| GL_NEAREST_MIPMAP_NEAREST | Vybere mipmapu, u které se nejvíce shoduje velikost s velikostí pixelu, který bude texturován a použije algoritmus GL_NEAREST pro výslednou hodnotu |
| GL_LINEAR_MIPMAP_NEAREST | Vybere mipmapu, u které se nejvíce shoduje velikost s velikostí pixelu, který bude texturován + chování jako GL_LINEAR |
| GL_NEAREST_MIPMAP_LINEAR | Vybere 2 mipmapy, u kterých se nejvíce shoduje velikost s velikostí pixelu, který bude texturován a použije algoritmus GL_NEAREST pro hodnotu každé mipmapy, výsledná hodnota bude vážený průměr těchto 2 hodnot |
| GL_LINEAR_MIPMAP_LINEAR | Vybere 2 mipmapy, u kterých se nejvíce shoduje velikost s velikostí pixelu, který bude texturován a použije algoritmus GL_LINEAR pro hodnotu každé mipmapy, výsledná hodnota bude vážený průměr těchto 2 hodnot |

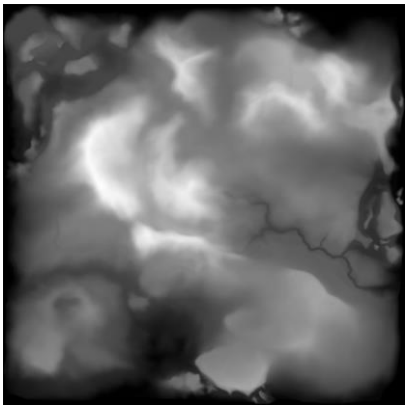
Zdroj: vlastní zpracování

### 6.5.6 Speciální typy textur

Texel u klasického obrázku reprezentuje pouze barvu obrázku v daném bodě. Nicméně texturu můžeme využít na uchování dalších informací, které můžeme uložit do barvy texelu.

#### 6.5.6.1 Výšková mapa

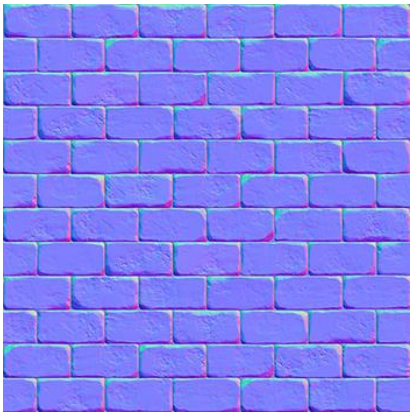
Výšková mapa je zpravidla černobílá textura, kde každý texel reprezentuje výšku v daném bodě. Bílá barva reprezentuje nejvyšší bod a černá barva reprezentuje nejnižší bod. Výška mezi těmito hraničními body je interpolována. Nejčastěji se používá pro reprezentaci terénu viz OBR. 26. Tvorba výškové mapy je velice snadná a lze ji připravit v jakémkoliv grafickém programu.



**Obr. 26 Ukázka výškové mapy terénu**
Zdroj: Drummyfish
https://commons.wikimedia.org/wiki/File:Hand_made_terrain_heightmap.png

#### 6.5.6.2 Normálová mapa

Normálová mapa je textura, která se používá na reprezentaci normál [4]. Kanál červené barvy reprezentuje souřadnici X, kanál zelené barvy souřadnici Y a nakonec kanál modré barvy souřadnici Z. Pro reprezentaci normál je potřeba převést hodnoty barev, které jsou v rozsahu 0 a 1 na rozsah -1 a 1. Toho docílíme vynásobením každé barvy číslem dva a následným odečtením čísla 1. Normálové mapy jsou zabarvené do modra (viz OBR. 27), protože všechny normály mají hodnotu souřadnice Z blízkou hodnotě 1.



**Obr. 27 Normálová mapa cihlové zdi**
Zdroj: Převzato z [6]

#### 6.5.6.3 Flow mapa

Flow mapa je speciální textura, která slouží k rozostření jiné textury viz **Obr. 36**. Používá se pro posun posun souřadnic jedné textury o souřadnice flow mapy. Tato informace je uchována ve 2 kanálech RGB: červený pro souřadnici x a zelený pro souřadnici y. Pokud je pixel černý, tak je informace zcela ignorována a dále se s ní nepracuje. Velice snadno lze flow mapu vytvořit z existující normálové mapy. Nejčastěji se používá pro realistický efekt vlnění vody nebo vlnění scény nad plamenem.

### 6.5.7 Použití v shaderu

Pro použití textury v shaderu je nejdříve potřeba definovat aktivní jednotku textury pomocí metody glActiveTexture, které předáme číslo jednotky. Tuto jednotku textury reprezentuje konstanta GL_TEXTURE*i*, kde *i* je nabývá hodnot z intervalu 0 až GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS – 1. Poté stačí texture napojit do OpenGL pomocí metody glBindTexture, které předáme cíl a id textury. V shaderu vytvoříme uniform proměnnou type sampler2D, která je ve výchozím stavu propojená s jednotkou textury 0. Nakonec stačí použít glsl metodu texture, které jako argumenty předáme sampler2D a vec2 souřadnice textury. Tato metoda nám vrátí výslednou barvu typu vec4.

```
#version 400 core

in vec2 textureCoordinates;

out vec4 color;

uniform sampler2D textureSampler;

void main(void) {
    color = texture(textureSampler, textureCoordinates);
}
```

## 6.6 Renderování do textury

Renderování do textury je velice užitečná metoda. Celou scénu si můžeme vyrenderovat do textury a následně na tuto texturu aplikovat různé efekty a modifikace. V OpenGL probíhá renderování do tzv. frame bufferu, který je uložen na grafické kartě [32]. Výchozí frame buffer obsahuje několik obrázků a je vytvořen společně s OpenGL kontextem. V knihovně JOGL je navázán na okno aplikace. OpenGL nám umožnuje definovat vlastní frame buffer, který budeme používat místo výchozího tzv. Frame Buffer Object.

### 6.6.1 Frame Buffer Object

Frame Buffer Object (FBO) je uživatelem definovaný frame buffer, který můžeme použit jako cíl renderování [38]. Na tento FBO se může aplikace kdykoliv přepnout.

V OpenGL FBO vytvoříme metodou glGenFrameBuffers a následně připojíme do OpenGL pomocí metody glBindFrameBuffeer, které musíme předat cíl: GL_DRAW_FRAMEBUFFER, GL_READ_FRAMEBUFFER nebo GL_FRAMEBUFFER. GL_DRAW_FRAMEBUFFER zajišťuje renderování a další zapisovací operace, GL_READ_FRAMEBUFFER pro čtecí operace a GL_FRAMEBUFFER umožňuje obojí.

Abychom mohli frame buffer používat, je nutné na něj navázat obrázek na speciálně určenou lokaci:

- GL_COLOR_ATTACHMENT lze napojit pouze obrázky s color formátem
- GL_DEPTH_ATTACHMENT lze napojit pouze obrázky s depth formátem
- GL_STENCIL_ATTACHMENT lze napojit pouze obrázky se stencil formátem
- GL_DEPTH_STENCIL_ATTACHMENT lze napojit pouze obrázky s depth-stencil formátem

Do takové lokace můžeme například připojit texturu, do které můžeme celou scénu vyrenderovat. Následně dostaneme texturu podle zvoleného formátu obrázku. S touto texturou můžeme dále pracovat a aplikovat různé efekty.

# 7 Implementace algoritmu pro renderování vodní hladiny

Pro implementaci algoritmu byl vybrán programovací jazyk Java a pro uživatelské rozhraní použita Java knihovna Swing. Pro práci s 3D grafikou byla vybrána grafická knihovna OpenGL. Pro vývoj aplikace bylo potřeba použít několik knihoven.

## *7.1 Použité knihovny*

Pro potřeby aplikace bylo potřeba použít 3 následující knihovny pro zpříjemnění a zrychlení vývoje. Protože jsem aplikaci programoval v Javě, bylo potřeba použít knihovnu JOGL, která zapouzdřuje API OpenGL.

### 7.1.1 JOGL

JOGL (Java binding for OpenGL) je open source knihovna pro práci s OpenGL v Javě. Původně byla vytvořena dvěma studenty Kenneth Bradley Russell a Christopher John Kline, poté byla spravována firmou Sun Microsystems Game Technology Group a od roku se o JOGL stará společnost Java on Graphics Audio and Processing (JOGAMP). Propojení s OpenGL je realizováno pomocí Java Native Interface, které používá přímo funkce napsané v jazyku C.

### 7.1.2 JOML

JOML (Java OpenGL Math Library) je open source matematická knihovna, která zpřístupňuje matematické operace potřebné pro práci s 3D grafikou. Především poskytuje operace s vektory, maticemi, vytváření projekčních matic, vytváření pohledové matice a další.

### 7.1.3 Toml4j

Toml4j je parser pro konfigurační formát TOML. V aplikaci je použito pro parsování konfigurace scény a nastavování určitých parametrů např. pozice entit ve světě. Pro vývoj je v aplikaci implementováno sledování tohoto konfiguračního souboru a při jakékoliv změně se scéna vytvoří znovu s novými parametry. Díky tomu lze jednoduše umístit jednotlivé objekty do scény s absolutní přesností bez opakovaného sestavení a spuštění aplikace ve vývojovém prostředí. Lze tedy i za běhu aplikace ve vývojovém prostředí entity odebírat nebo naopak přidávat.

## *7.2 Entity ve scéně*

Pro dokreslení scény kolem vodní hladiny bylo použito několik dalších entit. Mezi hlavní entitu, která definuje odraz/refrakci vodní hladiny, patří terén.

### 7.2.1 Terén

Terén je tvořen plochou, která se skládá z menších trojúhelníků. Každému vrcholu potřebujeme přiřadit správnou výšku, proto byla použita výšková mapa pro načtení této informace viz kapitola **6.5.6.1**. Bílá barva pixelu reprezentuje nejvyšší bod a černá barva pixelu nejnižší bod. Pomocí BufferedImage byl načten obrázek výškové mapy do paměti. Třída BufferedImage má metodu getRGB, která vrací integer barevnou hodnotu pixelu na souřadnicích x a y. Tato hodnota je vždy záporná a proto dostáváme interval, v kterém tato hodnota může být:

$$\langle -16777216, 0 \rangle$$

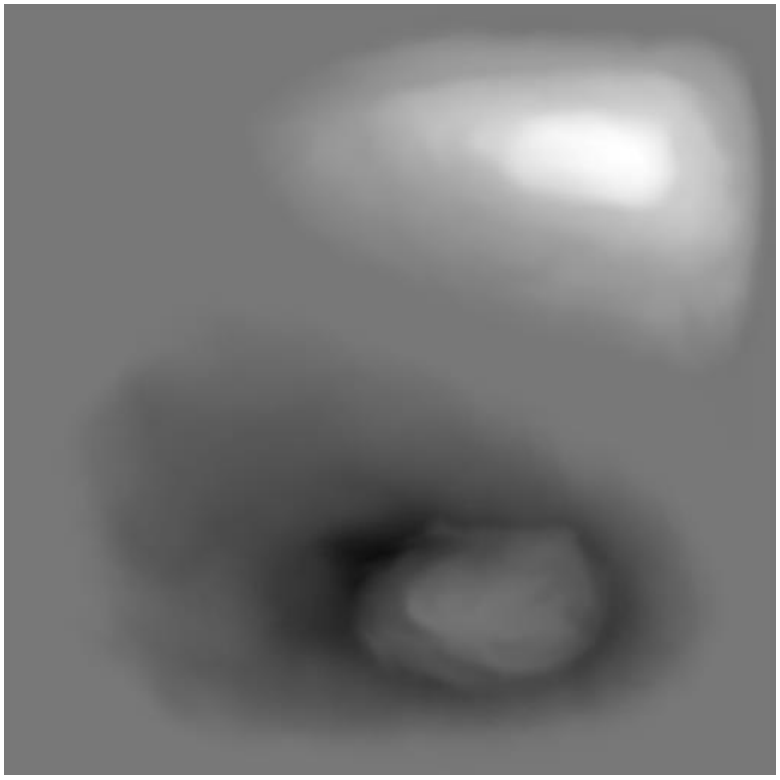
Číslo 16777216 odpovídá maximální hodnotě RGB barvy (každý kanál má 8-bit, celkově 256 hodnot), tedy $256 \times 256 \times 256$. Přičtením tohoto čísla získáme interval $\langle 0{,}16777216 \rangle$. Teď stačí interval vydělit tímto číslem, dostáváme interval $\langle 0{,}1 \rangle$ a

následně vynásobit žádanou výškou terénu, která je v kódu definována jako konstanta MAX_HEIGHT. Těmito operacemi dostáváme konečný interval:

$$\langle 0, MAX\_HEIGHT \rangle$$

Nyní bude každá hodnota pixelu v tomto intervalu. Teď stačí provést iteraci nad všemi pixely, pro daný pixel vytvořit vrchol a přiřadit výšku podle hodnoty pixelu.

Výšková mapa, která byla použita pro terén ve scéně, byla vytvořena v programu Gimp viz **Obr. 28**. Odstíny bílé barvy reprezentuje horu ve scéně a odstíny černé barvy jezero.



**Obr. 28 Výšková mapa terénu v aplikaci**
Zdroj: vlastní zpracování

### 7.2.2 Skybox

Skybox je krychle, která má aplikovanou texturu na každé stěně. Tyto textury na sebe navzájem bezešvě navazují a dohromady evokují okolí scény viz OBR. 29. Na takový případ je OpenGL připravené a skybox lze implementovat pomocí pár kroků.

**Obr. 29 Skybox, který byl použit v aplikaci**
Zdroj: Emil Persson http://www.humus.name/index.php?page=Textures&ID=83

Nejdříve vytvoříme texturu pomocí statické metody TextureIO.newTexture a jako cíl předáme konstantu GL_TEXTURE_CUBE_MAP. Dále postupně načteme všechny textury, pro každou texturu vytvoříme pomocí statické metody TextureIO.newTextureData instanci třídy TextureData. Poté už stačí akorát aktualizovat cubemap texturu pomocí metody updateImage, které předáme instanci TextureData a informaci o pozici textury. Pro tyto pozice jsou definovány konstanty viz TABULKA 10.

**Tabulka 10 Pozice cubemap textury**

| | |
|---|---|
| GL_TEXTURE_CUBE_MAP_POSITIVE_X | Pravá stěna |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_X | Levá stěna |
| GL_TEXTURE_CUBE_MAP_POSITIVE_Y | Horní stěna |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_Y | Dolní stěna |
| GL_TEXTURE_CUBE_MAP_POSITIVE_Z | Přední stěna |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_Z | Zadní stěna |

Zdroj: vlastní zpracování

### 7.2.3 Ostatní entity

Ostatní entity jsou pouze statické entity, které doplňují scénu vizuálně. Všechny tyto entity jsou načítány ze souborů formátu obj viz kapitola 6.1.3 a mají přidělenou svoji texturu.

## 7.3 Osvětlení scény

Pro osvětlení scény je použito pouze jedno směrové světlo viz kapitola 4.1.1. Každá entita počítá pouze s tímto světlem. Ve fragment shaderech se používá Phongův osvětlovací model viz kapitola 4.2.1.

## 7.4 Algoritmus pro renderování vodní hladiny

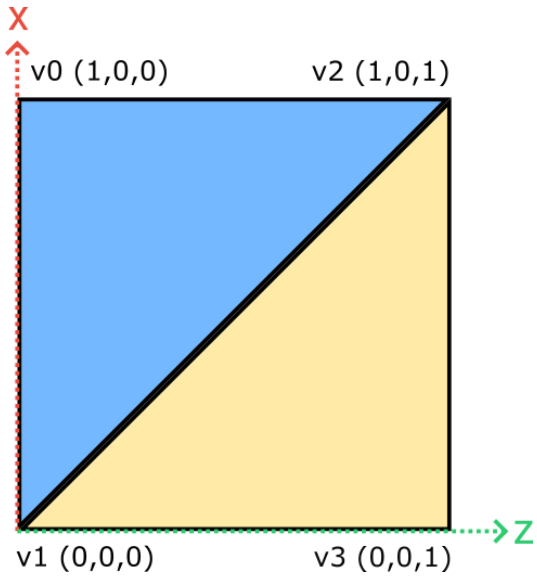
Algoritmus pro renderování vodní hladiny bude částečně vycházet z algoritmů představených v [14, 39]. Téměř všechny výpočty pro tento algoritmus budou probíhat ve fragment shaderu. Pro reprezentaci vodní hladiny nám postačí obyčejná plocha. Na tuto plochu budeme postupně aplikovat několik textur (projekce textury odrazu a refrakce, normálová mapa a flow mapa), aplikujeme Fresnelův efekt, vypočteme osvětlení a tím docílíme realistické vodní hladiny.

### 7.4.1 Plocha

Naši plochu, která se skládá ze 4 vrcholů, si nahrajeme do VAO (viz kapitola 6.1.2). Budeme potřebovat vytvořit 2 VBO (viz kapitola 6.1.1) – jeden pro vrcholy, který navážeme na atribut číslo 0 a druhý pro indexy (cíl dat nastavíme na GL_ELEMENT_ARRAY_BUFFER). Do prvního VBO nahrajeme následující vrcholy jako pole float hodnot (celkově 12 hodnot):

$$v_0 = (1,0,0), \quad v_1 = (0,0,0), \quad v_2 = (1,0,1), \quad v_3 = (0,0,1)$$

Do druhého VBO nahrajeme indexy jako pole celočíselných hodnot, kde každá trojice indexů nám bude tvořit trojúhelník. Dostáváme 2 trojúhelníky, které nám tvoří námi žádanou plochu viz Obr. 30.




**Obr. 30 Plocha, která představuje vodní hladinu**
Zdroj vlastní zpracování

Následně si můžeme plochu pomocí translační matice přesunout na požadovanou pozici ve scéně a pomocí škálovací matice roztáhnout na požadované rozměry viz kapitola 6.2.1. Na Obr. 31 je vidět plocha ve scéně s aplikovanou modrou barvu ve fragment shaderu.

**Obr. 31 Modrá plocha hladiny**
Zdroj: ukázka z aplikace

### 7.4.2 Textura pro odraz/refrakci vodní hladiny

Abychom mohli získat správné textury pro odraz/refrakci, budeme potřebovat rovinu řezu. Tuto rovinu umístíme tak, aby byla ve stejné výšce jako je vodní hladina a byla s ní rovnoběžná. Rovina nám bude velice nápomocná, protože díky ní můžeme oříznout vše, co se nachází za ní. Pro použití budeme muset v OpenGL nejdříve aktivovat ořezávací vzdálenost GL_CLIP_DISTANCE0 pomocí metody glEnable viz kapitola 6.3.4.2.

#### 7.4.2.1 Rovina řezu

Rovina je definováno obecnou rovnicí 7.1, kde A, B a C jsou složky normálového vektoru a D je vzdálenost od počátku souřadnicového systému [40].

$$Ax + By + Cz + D = 0 \tag{7.1}$$

Vzdálenost D se bude rovnat souřadnici y vodní hladiny (negativní pro rovinu mířící vzhůru).

My budeme potřebovat horizontální rovinu (rovnoběžná s vodní hladinou), která bude mít normálový vektor mířící nahoru $(A, B, C) = (0,1,0)$ nebo dolů $(A, B, C) = (0, -1,0)$.

#### 7.4.2.2 Textura odrazu vodní hladiny

Z kapitoly 3.2.1 víme, že úhel dopadu světelného paprsku se rovná úhlu odrazu. Abychom získali správný úhel z pohledu kamery a zachytili potřebnou část scény, musíme provést následující kroky:

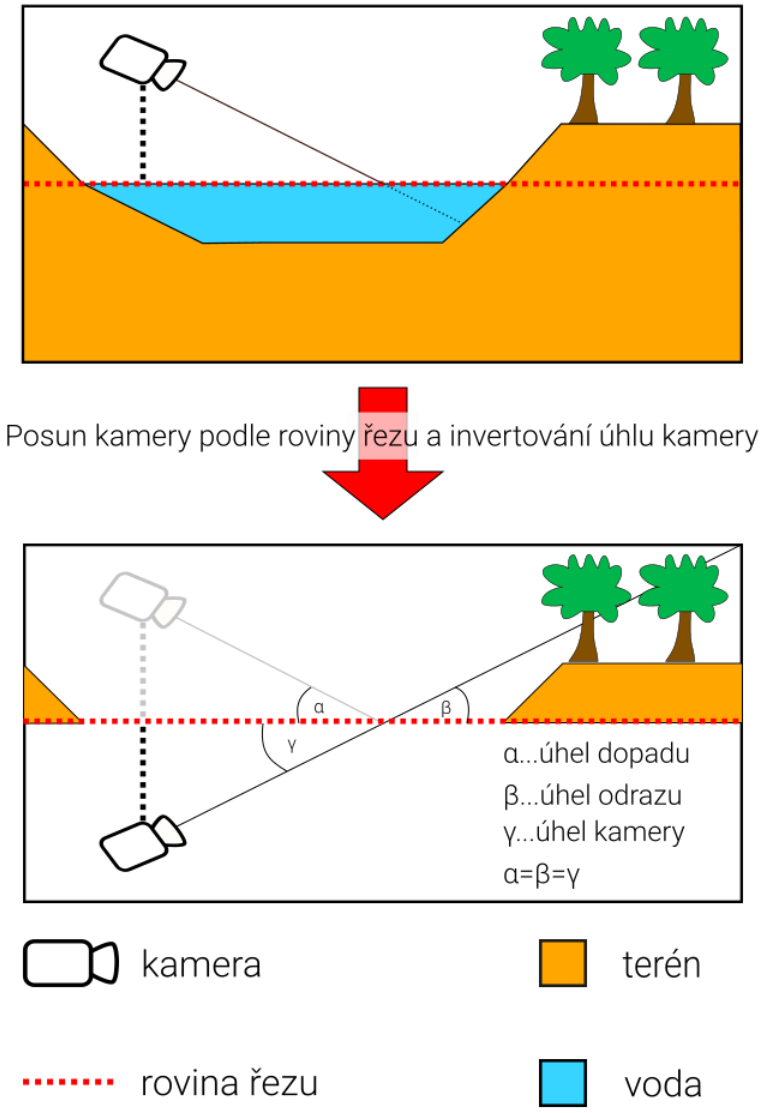
1. Připravit si FBO, na který napojíme texturu ve formátu RGB do lokace GL_COLOR_ATTACHMENT0
2. Napojit FBO do OpenGL
3. Přemístit kameru podle roviny řezu po ose y a invertovat úhel kamery
4. Aktivovat rovinu řezu v OpenGL
5. Vyrenderovat celou scénu
6. Odpojit FBO
7. Deaktivovat rovinu řezu
8. Ve fragment shaderu správně nastavit souřadnice textury pomocí projekce textury na plochu

Nejdříve si musíme vytvořit FBO (viz kapitola 6.6.1), do kterého připojíme texturu. Dále je potřeba si nadefinovat rovinu řezu, kterou umístíme na vodní hladinu s parametry $(A, B, C) = (0,1,0)$ a vzdáleností D, která se rovná negativní souřadnici y plochy jezera. V OpenGL aktivovat pomocí metody glEnable GL_CLIP_DISTANCE0, která nám umožní použít první index. Tuto rovinu si předáme pomocí uniform

proměnné (viz kapitola 6.4.4) do každého vertex shaderu, který pracuje s entitami ve scéně. V shaderu musíme aktivovat clip distance na indexu 0. Skalárním součinem pozice vrcholu ve světě a roviny získáme vzdálenost vrcholu od této roviny:

```
in vec3 position;
uniform mat4 transformationMatrix;
void main(void) {
    vec4 worldPosition = transformationMatrix * vec4(position, 1.0);
    gl_ClipDistance[0] = dot(worldPosition, plane);
}
```

Pokud je vzdálenost negativní, vrchol není v prostoru roviny (ve směru normálového vektoru) a je ořezán. Před renderováním scény se musíme přepnout na námi vytvořený FBO, změnit pozici kamery podle roviny řezu a invertovat úhel kamery viz OBR. 32.




**Obr. 32 Proces získání textury odrazu**
Zdroj: vlastní zpracování

Poté aktivovat rovinu řezu a scénu vykreslit. Do textury v FBO se nám uloží obrázek scény z pohledu kamery, kde budou vidět všechny entity nad rovinou řezu.

Tuto texturu budeme potřebovat správně namapovat pomocí projekce [41] na naši plochu, abychom správně viděli odrazy okolních objektů . Z vertex shaderu si pomocí výstupní proměnné předáme ořezávací souřadnice vrcholu

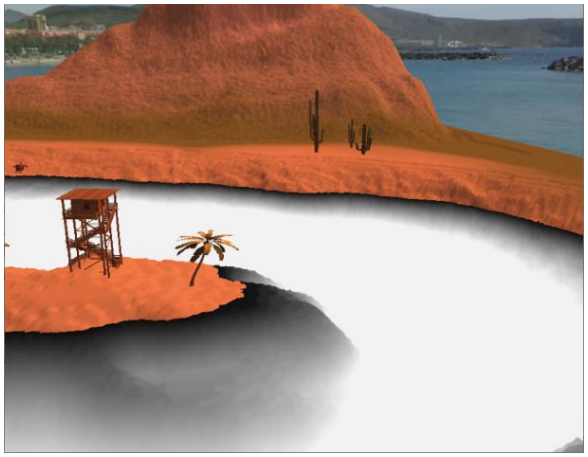
$(projectionMatrix \times viewMatrix \times transformationMatrix \times position)$ do fragment shaderu. Abychom tyto souřadnice mohli použít pro souřadnice textury, musíme je nejdříve normalizovat. Po normalizaci získáme rozsah -1.0 a 1.0 a ten si převedeme na rozsah 0.0 a 1.0. Tyto souřadnice již odpovídají souřadnicím textury. Pro správnou orientaci textury pouze musíme invertovat souřadnici y a získáme správně orientovanou texturu viz OBR. 33.

**Obr. 33 Aplikace textury odrazu na plochu**
Zdroj: Ukázka z vlastní aplikace

#### 7.4.2.3 Textura refrakce vodní hladiny

Pro texturu refrakce vodní hladiny budeme také potřebovat FBO s RGB texturou a navíc i depth buffer. Opět budeme potřebovat rovinu řezu, ale tentokrát s opačným normálovým vektorem $(A, B, C) = (0, -1, 0)$ a vzdáleností D, která se rovná pozitivní souřadnici y plochy jezera. Tentokrát kameru necháme ve své původní pozici. Napojíme si naše FBO, vyrenderujeme scénu, zrušíme GL_CLIP_DISTANCE0 a odpojíme FBO. Nyní jsme získali 2 textury: jednu klasickou texturu s barvami, která zachycuje všechny entity pod úrovní vodní hladiny a depth texturu, která zachycuje informace o hloubce scény. Tuto texturu budeme opět potřebovat správně namapovat jako texturu odrazu vodní hladiny. Provedeme stejné kroky, ale nebudeme invertovat souřadnici y.

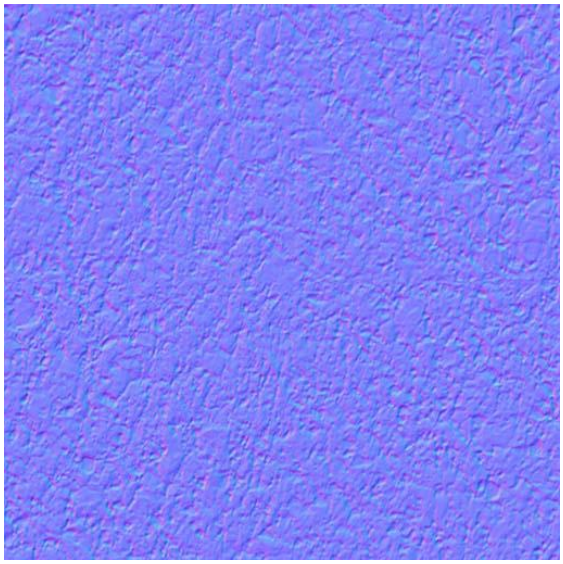
V textuře, která nám zachycuje hloubku, máme nyní uloženou informaci o hloubce v kanálu červené barvy. Stejné souřadnice textury poté použijeme i pro texturu, která nám zachycuje hloubku refrakce viz OBR. 34. Tuto informaci o hloubce můžeme využít pro vyhlazení okrajů vodní hladiny. Toho docílíme použitím alfa kanálu barvy – vodní hladina s menší hloubkou bude mít nastavenou větší průhlednost.



**Obr. 34 Textura s informací o hloubce**
Zdroj: Ukázka z vlastní aplikace

### 7.4.3 Normálová mapa

Pro výpočet osvětlení vodní hladiny použijeme normálovou mapu (viz kapitola 6.5.6.2), která nám bude simulovat osvětlení zvlněné vodní hladiny viz OBR. 35.



**Obr. 35 Normálová mapa pro vodní hladinu**
Zdroj: CAD Hatch http://www.cadhatch.com/seamless-water-textures/4588167784

Jelikož nám vodní hladina míří směrem vzhůru ve směru osy y, musíme prohodit souřadnici y se souřadnicí z. Nyní nám bude modrá barva reprezentovat souřadnici y. U modré barvy nebudeme měnit rozsah na -1.0 a 1.0, ale místo toho ji vynásobíme číslem 3 a o to více zdůrazníme normálový vektor ve směru osy y. Výsledný proces ve fragment shaderu vypadá následovně:

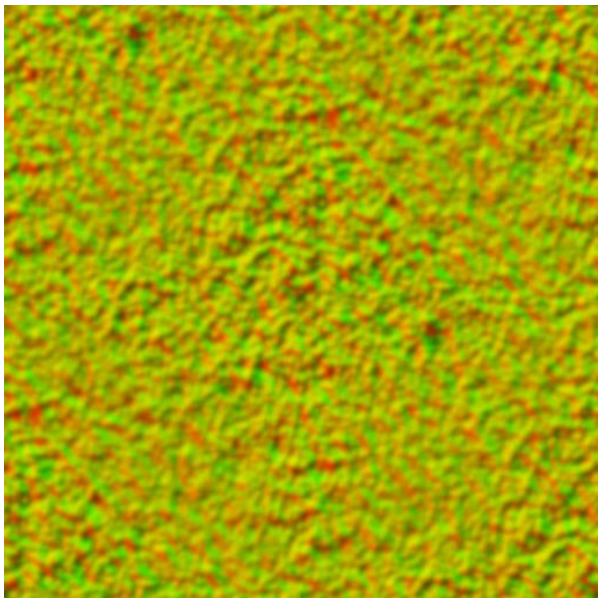
```
// Získání barvy textury na dané souřadnici
vec4 normalMapColor = texture(normalMap, textureCoords);

// Vytvoření normálového vektoru na základě barevných kanálů barvy + použití modré barvy pro y souřadnici + normalizace barev
vec3 normal = vec3(normalMapColor.r * 2.0 - 1.0, normalMapColor.b * 3.0, normalMapColor.g * 2.0 - 1.0);
```

### 7.4.4 Flow mapa

Odraz na vodní hladině je v reálném světě málokdy zcela nehybný a ostrý. Často je hladina zvlněná, a proto je odraz okolního světa rozostřený. Rozostřením budeme simulovat vlnění vodní hladiny a také aproximovat lámání světla na vodní hladině viz kapitola 3.3. Pro takový efekt je vhodné použít flow mapu viz kapitola 6.5.6.3.

Flow mapa byla vytvořena v programu Gimp z normálové mapy pomocí následujících kroků:

1. Importujeme si normálovou mapu
2. Invertujeme barvy
3. Zvýšíme jas a kontrast na maximální hodnotu
4. Získáme flow mapu viz **OBR. 36**



**Obr. 36 Flow mapa pro vodní hladinu**
Zdroj: vlastní zpracování pomocí normálové mapy vodní hladiny Obr. 35

Ve fragment shaderu použijeme R a G kanál textury pro aplikaci rozostření textury (změna souřadnice textury). Navíc můžeme texturu posouvat jedním směrem pomocí změny souřadnice textury x a y. K tomu nám poslouží proměnná float rovna 0, která se bude každý renderovací cyklus zvyšovat o určitou hodnotu a bude udržována v rozmezí 0 a 1. Tato dynamická hodnota se bude přidávat k souřadnicím textury. Toto rozostření textury nám poskytne pěkný realistický efekt vlnění vodní hladiny viz **OBR. 37**.

**Obr. 37 Rozostření vodní hladiny**
Zdroj: Ukázka z vlastní aplikace

### 7.4.5 Fresnelův efekt

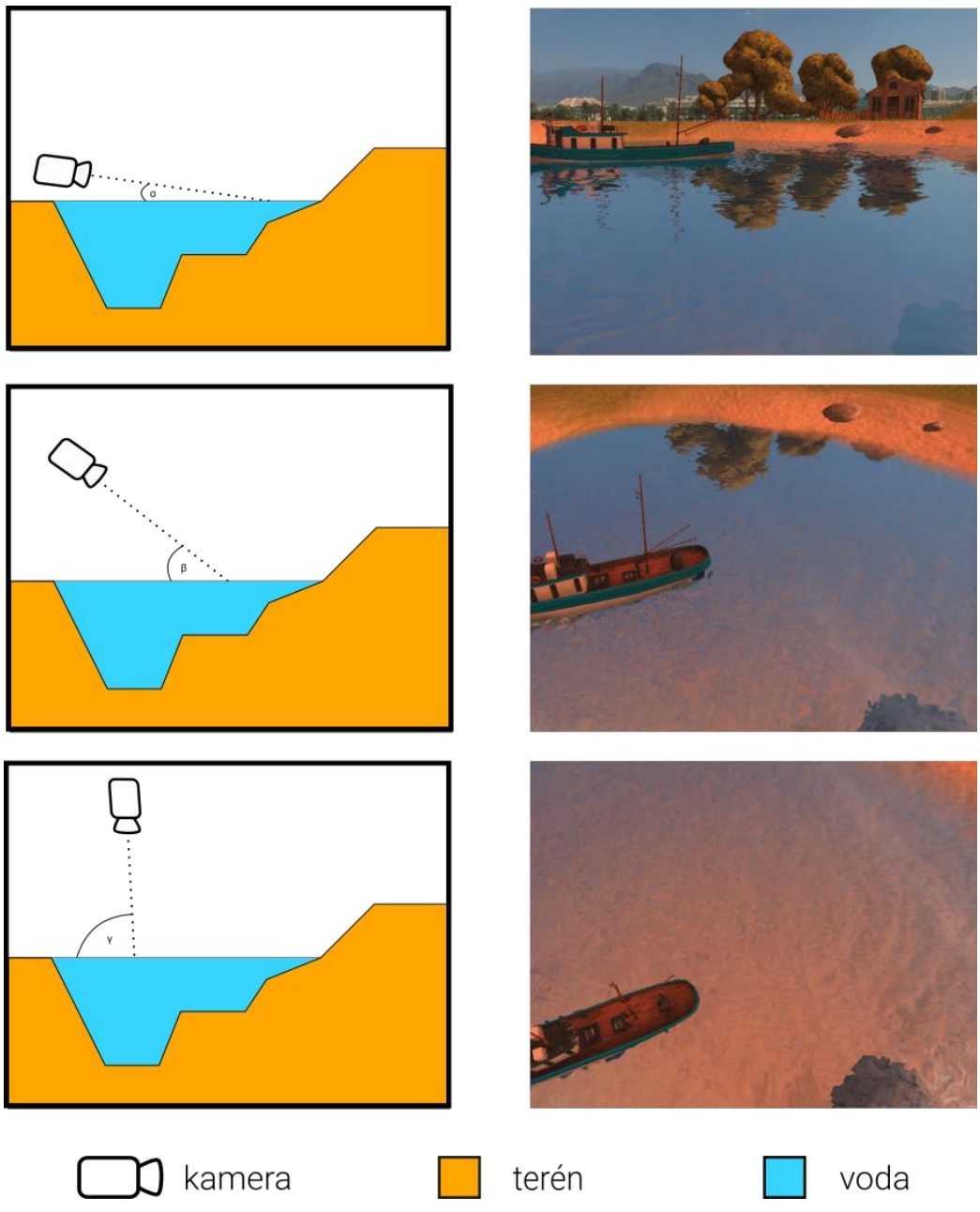
Fresnelův efekt (viz kapitola 3.3.3) dokáže dále evokovat ještě více realistickou vodní hladinu. Pokud vektor pozorování svírá s vodní hladinou malý úhel, je více vidět odražená hladina a naopak, pokud svírá velký úhel, je voda průzračná a není skoro vidět odraz hladiny. Abychom mohli implementovat Fresnelův efekt, potřebujeme znát 2 vektory: normálový vektor a vektor, který směruje do kamery. Výpočet Fresnelova efektu bude probíhat ve fragment shaderu. Skalárním součinem vektoru do kamery a normálového vektoru dostaneme číslo, z kterého můžeme vypočítat i úhel mezi vektory [36]:

$$\vec{u} \times \vec{v} = |\vec{u}| \times |\vec{v}| \times \cos \alpha \tag{8.2}$$

Pokud se toto číslo rovná nule, úhel mezi vektory se rovná 0. Toto číslo pak můžeme použít při aplikaci lineární interpolace textury odrazu a refrakce metodou mix.

```
// Skalární součin vektoru mířícího do kamery a normálového vektoru,
oba vektory musí být normalizované
float refractiveFactor = dot(toCameraVector, normal);

// lineární interpolace barvy odražené textury a refraktované textury
vec4 color = mix(reflectColor, refractionColor, refractiveFactor);
```

Výsledkem je Obr. 38, kde se vodní hladina mění v závislosti na pozici a úhlu kamery.




**Obr. 38 Fresnelův efekt**
Zdroj: Ukázka z vlastní aplikace

### 7.4.6 Lesk vodní hladiny

Pro zachycení lesku odrazu vodní hladiny byla použita pouze lesklá složka světla Phongova osvětlovacího modelu (viz kapitola 4.2.1). Nejdříve bylo potřeba získat směr odraženého paprsku světla pomocí glsl funkce reflect, které předáme normalizovaný směr světla a normálový vektor. Jak moc se vodní hladina leskne určíme skalárním součinem vektoru mířícího z kamery do scény a tohoto vektoru odraženého světla. Na základě této hodnoty můžeme vypočítat lesklou složku a následně ji přidat k výsledné barvě ve fragment shaderu.

### 7.4.7 Souhrn a výsledky algoritmu

Na závěr algoritmu byla do barvy každého fragmentu přimíchána modrá barva. Pro aplikaci algoritmu bylo potřeba provést následující kroky:

1) Vytvořit plochu pomocí 4 vrcholů a tu předat do VAO
2) Barevná textura odrazu vodní hladiny
   a) Definovat rovinu řezu na vodní hladině, která bude ořezávat všechny objekty pod vodní hladinou
   b) Vytvořit FBO, do kterého připojíme texturu formátu RGB
   c) Aktivovat rovinu řezu
   d) Přesunout kameru podle vodní hladiny po ose y
   e) Invertovat úhel kamery
   f) Přepnout se na FBO
   g) Vyrenderovat scénu do FBO
   h) Kameru přesunout zpět do původní pozice
3) Barevná/depth textura refrakce vodní hladiny
   a) Definovat rovinu řezu na vodní hladině, která bude ořezávat všechny objekty nad vodní hladinou
   b) Vytvořit FBO, do kterého připojíme texturu formátu RGB a depth texturu
   c) Aktivovat rovinu řezu
   d) Přepnout se na FBO
   e) Vyrenderovat scénu do FBO
4) Aplikace normálové mapy pro reprezentaci zvlněné vodní hladiny
5) Aplikace flow mapy pro rozostření vodní hladiny
6) Fresnelův efekt
   a) Na základě vektoru pozorování a normálového vektoru v daném bodě interpolovat barvu textury odrazu a refrakce vodní hladiny
7) Aplikace lesklé části Phongova osvětlovacího modelu
8) Přimíchání modré barvy pro každý fragment
9) Výsledná plocha s aplikovanými texturami a osvětlením viz **OBR. 39**

**Obr. 39 Ukázka konečného výsledku algoritmu pro vodní hladinu**
Zdroj: Ukázka z aplikace

# 8 Závěry a doporučení

Práce měla za cíl představit různé metody pro renderování vodní hladiny, popsat grafickou knihovnu OpenGL a následně pomocí OpenGL implementovat algoritmus pro reprezentaci vodní hladiny.

Použitý algoritmus se ukázal jako velice vhodný pro reprezentaci vodní hladiny. Výpočetně je velice nenáročný a výsledná scéna působí velice realisticky. Možným vylepšením tohoto algoritmu by byla skutečná 3D reprezentace vln a tvorba pěny u břehu pomocí částicového systému.

Určitě by bylo dále zajímavé prozkoumat implementaci algoritmu, který je založen na fyzikálních výpočtech a pracuje s vodou jako celkovým objemem.

Těmto problémům se zamýšlím věnovat v rámci navazujícího magisterského studia.

# 9 Seznam použité literatury


[1] LEPIL, Oldřich. *Fyzika pro gymnázia. Optika*. 5., přepracované vydání. Praha: Prometheus, 2015. ISBN 978-80-7196-444-5.

[2] REICHL, Jaroslav a Martin VŠETIČKA. *Encyklopedie fyziky* [online]. 2020 [vid. 2020-02-29]. Dostupné z: http://fyzika.jreichl.com/

[3] BIRN, Jeremy. *[Digital] lighting and rendering*. 3. vyd. B.m.: New Riders, 2014. ISBN 978-0-321-92898-6.

[4] AKENINE-MÖLLER, Tomas. *Real-Time Rendering, Fourth Edition* [online]. B.m.: A K Peters/CRC Press, 2018. ISBN 978-1-138-62700-0. Dostupné z: https://www.xarg.org/ref/a/1138627003/

[5] ŽÁRA, Jiří. *Moderní počítačová grafika*. Brno: Computer Press, 2004. ISBN 80-251-0454-0.

[6] VRIES, Joey de. *Learn OpenGL, extensive tutorial resource for learning Modern OpenGL* [online]. 2014 [vid. 2020-04-13]. Dostupné z: https://learnopengl.com/

[7] PHONG, Bui Tuong. Illumination for Computer Generated Pictures. 1975, **18**(6), 7.

[8] BLINN, James F. Models of Light Reflection for Computer Synthesized Pictures. In: *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques* [online]. New York, NY, USA: Association for Computing Machinery, 1977, s. 192–198. SIGGRAPH '77. ISBN 978-1-4503-7355-5. Dostupné z: doi:10.1145/563858.563893

[9] GOURAUD, H. Continuous Shading of Curved Surfaces. *IEEE Transactions on Computers*. 1971, **C–20**(6), 623–629.

[10] IGLESIAS, A. Computer graphics for water modeling and rendering: a survey. *Future Generation Computer Systems* [online]. 2004, **20**(8), 1355–1374. ISSN 0167-739X. Dostupné z: doi:https://doi.org/10.1016/j.future.2004.05.026

[11] BLINN, James F. Simulation of Wrinkled Surfaces. *SIGGRAPH Comput. Graph.* [online]. 1978, **12**(3), 286–292. ISSN 0097-8930. Dostupné z: doi:10.1145/965139.507101

[12] WHITTED, Turner. An Improved Illumination Model for Shaded Display. In: *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques* [online]. New York, NY, USA: Association for Computing Machinery, 1979, s. 14. SIGGRAPH '79. ISBN 0-89791-004-4. Dostupné z: doi:10.1145/800249.807419

[13] MAX, Nelson L. Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset. In: *Proceedings of the 8th Annual Conference on Computer*

*Graphics and Interactive Techniques* [online]. New York, NY, USA: Association for Computing Machinery, 1981, s. 317–324. SIGGRAPH '81. ISBN 0-89791-045-1. Dostupné z: doi:10.1145/800224.806820

[14] FOURNIER, Alain a William T. REEVES. A Simple Model of Ocean Waves. *SIGGRAPH Comput. Graph.* [online]. 1986, **20**(4), 75–84. ISSN 0097-8930. Dostupné z: doi:10.1145/15886.15894

[15] ARVO, James. Backward ray tracing. In: *Developments in Ray Tracing, Computer Graphics, Proc. of ACM SIGGRAPH 86 Course Notes*. 1986, s. 259–263.

[16] PEACHEY, Darwyn R. Modeling Waves and Surf. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* [online]. New York, NY, USA: Association for Computing Machinery, 1986, s. 65–74. SIGGRAPH '86. ISBN 0-89791-196-2. Dostupné z: doi:10.1145/15922.15893

[17] TS'O, Pauline Y. a Brian A. BARSKY. Modeling and Rendering Waves: Wave-Tracing Using Beta-Splines and Reflective and Refractive Texture Mapping. *ACM Trans. Graph.* [online]. 1987, **6**(3), 191–214. ISSN 0730-0301. Dostupné z: doi:10.1145/35068.35070

[18] MILLER, Gavin a Andrew PEARCE. Globular dynamics: A connected particle system for animating viscous fluids. *Computers & Graphics* [online]. 1989, **13**(3), 305–309. ISSN 0097-8493. Dostupné z: doi:https://doi.org/10.1016/0097-8493(89)90078-2

[19] KASS, Michael a Gavin MILLER. Rapid, Stable Fluid Dynamics for Computer Graphics. *SIGGRAPH Comput. Graph.* [online]. 1990, **24**(4), 49–57. ISSN 0097-8930. Dostupné z: doi:10.1145/97880.97884

[20] SIMS, Karl. Particle Animation and Rendering Using Data Parallel Computation. *SIGGRAPH Comput. Graph.* [online]. 1990, **24**(4), 405–413. ISSN 0097-8930. Dostupné z: doi:10.1145/97880.97923

[21] WATT, Mark. Light-Water Interaction Using Backward Beam Tracing. *SIGGRAPH Comput. Graph.* [online]. 1990, **24**(4), 377–385. ISSN 0097-8930. Dostupné z: doi:10.1145/97880.97920

[22] TONNESEN, David. Modeling liquids and solids using thermal particles. In: *Graphics Interface*. 1991.

[23] GONZATO, Jean-CHristophe a Le B. On Modeling and Rendering Ocean Scenes. *The Journal of Visualization and Computer Animation* [online]. 2000, **11**, 27–37. Dostupné z: doi:10.1002/(SICI)1099-1778(200002)11:1<27::AID-VIS214>3.0.CO;2-5

[24] GAMITO, Manuel N. a F. Kenton MUSGRAVE. An accurate model of wave refraction over shallow water. *Computers & Graphics* [online]. 2002, **26**(2), 291–307. ISSN 0097-8493. Dostupné z: doi:https://doi.org/10.1016/S0097-8493(01)00181-9

[25] JESCHKE, Stefan, Hermann BIRKHOLZ a H. SCHUMANN. A Procedural Model for Interactive Animation of Breaking Ocean Waves. In: *WSCG*. 2003.

[26] HOLMBERG, Nathan a Burkhard C. WÜNSCHE. Efficient Modeling and Rendering of Turbulent Water over Natural Terrain. In: *Proceedings of the 2nd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia* [online]. New York, NY, USA: Association for Computing Machinery, 2004, s. 15–22. GRAPHITE '04. ISBN 1-58113-883-0. Dostupné z: doi:10.1145/988834.988837

[27] KRYACHKO, Yuri. Using Vertex Texture Displacement for Realistic Water Rendering. In: *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*. 2005. ISBN 0-321-33559-7.

[28] MONTGOMERY, Jogn. Assassin's Creed III: The tech behind (or beneath) the action. *fxguide* [online]. 11. prosinec 2012 [vid. 2020-04-22]. Dostupné z: https://www.fxguide.com/fxfeatured/assassins-creed-iii-the-tech-behind-or-beneath-the-action/

[29] TESSENDORF, Jerry. Simulating Ocean Water. 2001.

[30] GRUJIC, Branislav. Water Rendering in FarCry 5. In: *Game Developers Conference*. 2018.

[31] PEDDIE, Jon. *The History of Visual Magic in Computers: How Beautiful Images are Made in CAD, 3D, VR and AR* [online]. London: Springer-Verlag, 2013 [vid. 2020-02-29]. ISBN 978-1-4471-4931-6. Dostupné z: doi:10.1007/978-1-4471-4932-3

[32] SEGAL, Mark a Kurt AKELEY. *The OpenGL Graphics System: A Specification (Version 4.6 (Core Profile) - October 22, 2019)* [online]. 2019. Dostupné z: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf

[33] Khronos Releases Vulkan 1.0 Specification. *The Khronos Group* [online]. 16. únor 2016 [vid. 2020-04-18]. Dostupné z: https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification

[34] WIKI, OpenGL. *Main Page — OpenGL Wiki*, [online]. 2018. Dostupné z: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Main_Page&oldid=14430

[35] MCHENRY, Kenton a Peter BAJCSY. An overview of 3D data content, file formats and viewers. 2008.

[36] BEČVÁŘ, Jindřich. *Lineární algebra*. Praha: Matfyzpress, 2005. ISBN 978-80-86732-57-2.

[37] KESSENICH, John. *The OpenGL® Shading Language, Version 4.60.7* [online]. 2019. Dostupné z: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf

[38] GREEN, Simon. The OpenGL framebuffer object extension. In: *Game developers conference*. 2005.

[39] LOMBARD, Yann. Realistic Natural Effect Rendering: Water I. *GameDev.net* [online]. září 2004 [vid. 2020-04-22]. Dostupné z: https://gamedev.net/tutorials/programming/graphics/realistic-natural-effect-rendering-water-i-r2138

[40] POLÁK, Josef. *Přehled středoškolské matematiky*. 6. vyd. Praha: Prometheus, 1995. ISBN 80-85849-78-X.

[41] MCREYNOLDS, Tom a David BLYTHE. *Advanced Graphics Programming Using OpenGL*. San Francisco, CA: Elsevier Morgan Kaufmann Publishers, 2005. ISBN 1-55860-659-9.

# 10 Přílohy

1) CD se zdrojovými kódy aplikace a spustitelnou aplikací

Univerzita Hradec Králové
Fakulta informatiky a managementu

# Zadání bakalářské práce

**Autor:** **Dennis Tschamler**

Studium: I1600623

Studijní program: B1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

**Název bakalářské práce:** **Dynamické systémy ve virtuální realitě**

Název bakalářské práce AJ: Dynamic systems in virtual reality

**Cíl, metody, literatura, předpoklady:**

Cíl: Seznámit se s metodami renderování vody ve virtuální realitě, implementace vybrané metody v OpenGL pomocí shaderu a porovnat schopnosti herních enginů v renderování vody.

Osnova:

1. Úvod
2. Historie vývoje renderování vody v počítačové grafice
3. Dostupné metody pro renderování vody v počítačové grafice
4. Realistické zobrazení fyzikálních vlastností vody v počítačové grafice
5. Implementace renderování vodní hladiny v OpenGL
6. Porovnání schopností herních enginů pro renderování vody

- HEARN, Donald a M. Pauline BAKER. Computer graphics with OpenGL. 3rd ed. Upper Saddle River, NJ: Pearson Prentice Hall, c2004. ISBN 0-13-015390-7.
- JOHANSON, Claes; LEJDFORS, Calle. Real-time water rendering. Lund University, 2004.
- MARSCHNER, Steve; SHIRLEY, Peter. Fundamentals of computer graphics. CRC Press, 2015.

Garantující pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Datum zadání závěrečné práce: 14.1.2018