FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Simulování vodního povrchu
Student:	Hong Son Ngo
Vedoucí:	Ing. Petr Pauš, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Počítačová grafika
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Voda a vodní hladina se vyskytuje ve spoustě grafických aplikací a hrách a jejich realistická real-time simulace je žádoucí. Cílem této práce je vytvořit simulátor vodního povrchu a jeho případných světelných efektů na tělesa pod vodou.

1. Analyzujte možnosti real-time simulace vodní hladiny.
2. Analyzujte vhodné nástroje pro její simulaci (např. OpenGL, Unity, atd.).
3. Na základě analýzy vyberte vhodný nástroj pro simulaci a navrhněte prototyp simulace.
4. Ve zvoleném nástroji implementujte.
5. Vytvořte testovací scénu, která simulaci demonstruje.

Elektronicky schválil/a Ing. Radek Richtr, Ph.D. dne 16. prosince 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Simulování vodního povrchu

Hong Son Ngo

Katedra softwarového inženýrství
Vedoucí práce: Ing. Petr Pauš, Ph.D.

1. května 2022

Poděkování

Rád bych poděkoval Ing. Petru Paušovi, Ph.D. za aktivní vedení, cenné rady, věcné připomínky a vstřícnost při konzultacích této bakalářské práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 1. května 2022 . . . . . .

Odkaz na tuto práci

Ngo, Hong Son. *Simulování vodního povrchu.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022. Dostupný také z WWW: ⟨`https://projects.fit.cvut.cz/theses/4032`⟩.

Abstrakt

Tato bakalářská práce se zabývá možnostmi *real-time* simulace vodního povrchu a jeho případných optických vlastností jak na hladině vody, tak i na objekty, které leží pod ní. V analýze jsou kromě aktuálních metod simulace dynamických a optických vlastností vodní hladiny probrány i technologie podporující *real-time* vizualizaci simulace. Praktická část je věnována návrhu simulátoru vodní hladiny využívající vlnové rovnice pro aproximaci jejího pohybu a aplikaci testovací scény a její následné implementaci. Optické vlastnosti vodní hladiny jako odrazy, refrakce a kaustiky jsou implementovány pomocí zjednodušené formy vrhání paprsků, které za pomoci podpůrných textur aproximují průsečíky paprsků v prostoru obrazu. Výsledná vodní hladina se po zásahu uživatele v reálném čase deformuje a následně opticky reaguje na okolní prostředí promítáním kaustik na podvodní objekty a výpočtem barvy vodního povrchu podle Fresnelových rovnic. Na závěr jsou uvedeny nedostatky zvolených algoritmů použité ve finální verzi aplikace, které spočívají v metodě výpočtu integrace vlnové rovnice nebo odhadu průsečíků světelných paprsků.

Klíčová slova simulace vodní hladiny, vlnová rovnice, optické vlastnosti vody, kaustiky, reflekce, refrakce, OpenGL, real-time vykreslování

Abstract

This bachelor thesis examines current options of real-time water surface simulation and its possible optic properties affecting the water surface itself or object laying underneath it. The contents of the analysis are not only the current methods of simulating dynamic and optic properties of a water surface but also technologies supporting its real-time rendering. The practical part is dedicated to the design of the water surface simulator, which uses wave equation as the main method for approximation of its movement, and the application of the testing scene and subsequently its implementation. Water surface optic properties such as reflection, refraction and caustics are then implemented with simplified method of ray-tracing which approximates light ray intersections in the screen space using auxiliary textures. Final water surface deforms after user interference and subsequently reacts optically on surrounding environment by casting caustics on underwater objects and by calculating water surface color from Fresnel equations. At the end deficiencies of used algorithms in the final version of the application, which lay in the method of integration of the wave equation and in the method of light ray intersection approximation, are stated.

Keywords water surface simulation, wave equation, water optic properties, caustics, refraction, reflection, OpenGL, real-time rendering

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Kvalita herních titulů za posledních let výrazně vzrostla. Jejich úspěch lze připsat nejen zajímavému příběhovému obsahu, ale také i jejich vizuálnímu zpracování. Právě v grafickém provedení můžeme vidět největší pokrok. Díky hardwarovým zlepšení v grafických kartách se obraz her čím dál více blíží fotorealistickým výsledkům a s příchodem zařízení pro virtuální a rozšířenou realitu je o realistický obraz ještě větší zájem.

Co rozlišuje scény napříč historií her, jsou speciální efekty. Na základě kvality jejich provedení je hráč hlouběji vnořen do virtuálního světa a následně i do jeho příběhu. Jedním z takových efektů jsou např. přírodní fenoména jako pohyb plamene, vlnění hladiny vody, proudění větru...

Fyzikálně korektní chování přírodních jevů však zůstává do dnešního dne mezi komplexnějšími problémy. Mezi těmi výpočetně nejobtížnějšími je považováno proudění tekutin, do kterých patří jak plynné, tak i kapalné skupenství. Pro vizualizaci se většina metod zabývá zejména kapalinami kvůli její povaze (jsou lidským okem viditelné), ale v některých případech lze postupy pro simulování proudění kapalin využít i pro zobrazení plynů jako např. kouře nebo plamene.

Kvůli vysoce dynamickému chování tekutin se však v současnosti nejrealističtějších výsledků dosáhne hlavně *off-line* metodami. *Real-time* aplikace využívají stejných principů, ale s podstatnými kompromisy jak paměťovými, tak i výpočetními. Vzhledem k tomu, že některé aplikace nevyužijí přesné a hlavně výpočetně drahé simulace, tak napodobují jen výsledné efekty, které se nijak neopírají o fyzikální zákony. Ačkoli *off-line* metody zobrazují věrohodně chování kapalin, resp. plynů, nejsou nijak zastoupeny v hrách, neboť je interakce s uživatelem a dynamické prostředí scén nedovolí využít.

Protože vývojáři her často ve svých produktech vodní hladiny a ostatní dynamické přírodní fenoména ručně animují a protože se jejich simulaci kvůli výpočetní náročnosti spíše vyhýbají, zkoumám ve své bakalářské práci možné metody simulace vodních ploch, které lze za pomocí současného hardwaru spo-

čítat v reálném čase. Obsah této bakalářské práce se zabývá kromě *real-time* simulace vodního povrchu, také jeho případnými světelnými efekty jak na hladině, tak i na tělesa pod ní, jejíž aplikace by bylo možné využít k realistickému zobrazení virtuálních scén v hrách.

Bakalářská práce je následovně členěna. Nejdříve je čtenář krátce seznámen s mechanikou tekutin, která je nezbytnou prerekvizitou pro pochopení simulace vodních útvarů. Následovně jsou probrány dynamické a optické vlastnosti vodních ploch, k jejímž hlavním rysům jsou uvedeny algoritmy, které je simulují. V další části je probrán návrh a implementace simulátoru vodní hladiny podle vybraných metod a technologií probraných v analýze. Na závěr je provedeno vizuální vyhodnocení použitých metod.

Kapitola 1

Cíl práce

Cílem teoretické části této bakalářské práce je prozkoumat možnosti *real-time* simulace vodního povrchu a jeho případných vlastností, které interagují se světelnými paprsky osvětlující vodní plochu. Na základě analýzy současných metod prozkoumat vhodné nástroje pro její zobrazení v grafických aplikací jako jsou počítačové hry.

Následně v praktické části je cílem vybrat vhodné simulační metody pro *real-time* grafické aplikace, navrhnout prototyp simulátoru, ve zvoleném nástroji ho implementovat a nakonec vytvořit testovací scénu, která simulaci demonstruje.

Kapitola 2

Základ teorie mechaniky tekutin

V této kapitole je shrnutí teorie mechaniky tekutin, která je nezbytnou součástí simulace fyzikálně korektního chování vody. Teorie je zde vyložena takovým způsobem, aby byl čtenář s ní seznámen a co nejrychleji pochopil principy chování tekutin, neobsahuje žádné rigorózní vysvětlení problematiky.

2.1 Navierovy–Stokesovy rovnice

Proud tekutin se v reálném světě řídí podle Navierových–Stokesových rovnic (NSE), soustavou nelineárních diferenciálních rovnic:

$$\nabla \cdot \mathbf{v} = 0, \tag{2.1}$$

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} + \frac{1}{\rho} \nabla p = \mathbf{g} + \nu \nabla^2 \mathbf{v}, \tag{2.2}$$

kde vektor $\mathbf{v} = (u, v, w)$ označuje rychlost tekutiny, ρ označuje hustotu tekutiny, p tlak, kterým působí tekutina na své okolí, $\mathbf{g} = (x, y, z)$ je gravitační zrychlení, ν je značení kinematické viskozity tekutiny. Symboly ∇, $\nabla\cdot$, ∇^2 označují diferenciální operátory nabla, divergence a Laplacův operátor. [1]

2.1.1 Vysvětlení Navierových–Stokesových rovnic

Před vysvětlením jednotlivých rovnic je dobré zmínit, že většina teorie mechaniky tekutin je založena na odvětví matematiky vektorové analýzy a pracují s vektorovými poli, resp. skalárními poli. NSE např. pracují s vektorovými poli, resp. se skalárními poli, které jednotlivým bodům prostoru přiřazují vektor určující rychlost proudu tekutiny, resp. hodnotu tlaku.

Na první pohled vypadají rovnice těžce pochopitelné, ale myšlenka za nimi je velmi jednoduchá. První rovnice 2.1 popisuje zákon o zachování hmotnosti, tím je myšleno, že není možné, aby hmota tekutiny na některém místě z ničeho vznikla nebo zanikla. [1]

Druhá rovnice 2.2 popisuje zákon o zachování hybnosti a je ve své podstatě Newtonův druhý zákon.

$$\mathbf{F} = m\mathbf{a} \tag{2.3}$$

Zrychlení $\mathbf{a}$ lze přepsat jako derivace rychlosti podle času.

$$\mathbf{F} = m\frac{D\mathbf{v}}{Dt}, \tag{2.4}$$

kde $\frac{D\mathbf{v}}{Dt}$ značí tzv. materiálovou derivaci. Pro odvození celé rovnice je třeba rozvést síly $\mathbf{F}$, které na tekutinu působí. [1] Jedna z nich je samozřejmě gravitace, jejíž hodnota je vyčíslena jako $m\mathbf{g}$.

Další síly vytváří tekutina sama na sobě. První z nich je síla způsobená rozdílem tlaků v tekutině. Tekutina se v oblastech s vyšším tlakem přesouvá do oblastí s nižším tlakem. Její hodnotu můžeme zapsat jako $-V\nabla p$ [1]. [1]

Další síla působící na tekutinu je ovlivněna její viskozitou. Viskozita působí při každém pohybu částic tekutiny a snaží se vyrovnat rychlost částice rychlostí svých sousedních částic. Její hodnotu můžeme vyjádřit jako $V\mu\nabla^2\mathbf{v}$ [2], kde μ označuje koeficient dynamické viskozity. [1]

$$m\mathbf{g} - V\nabla p + V\mu\nabla^2\mathbf{v} = m\frac{D\mathbf{v}}{Dt} \tag{2.5}$$

Rovnice 2.3, 2.4 a 2.5 předpokládají, že tekutinu lze rozložit na konečně mnoho malých částí, tímto způsobem je do vyčíslení představena výpočetní chyba. Řešením tohoto problému je celou rovnici 2.5 vydělit hodnotou V, aby zachytila pohyb nekonečně mnoho nekonečně malých částic tekutin. Dobré je připomenout, že hustota ρ lze vyjádřit jako $\frac{m}{V}$. [1]

$$\rho\mathbf{g} - \nabla p + \mu\nabla^2\mathbf{v} = \rho\frac{D\mathbf{v}}{D\mathbf{t}} \tag{2.6}$$

Následně po vydělení hustoty ρ, vyjádření derivace $\frac{D\mathbf{v}}{Dt}$ podle pravidla pro složené funkce a prohození sčítanců získáme druhou Navierovu–Stokesovu rovnici. Kinematická viskozita ν je rovna $\frac{\mu}{\rho}$. [1]

$$\frac{\partial\mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v} + \frac{1}{\rho}\nabla p = \mathbf{g} + \nu\nabla^2\mathbf{v} \tag{2.7}$$

[1] Operace ∇p vyjadřuje vektorové pole, ve kterém vektory směřují z jednotlivých bodů na své sousední body tak, aby jeho hodnota ve skalárním poli nejrychleji vzrostla.

[2] Operace $\nabla^2\mathbf{v}$ vyjadřuje míru odchýlení rychlosti částice od rychlosti svého okolí.

2.1.2 Využití Navierových–Stokesových rovnic

Výhodou NSE je, že je lze aplikovat na téměř jakékoliv tekutiny. V praxi se využívají pro předpověď počasí, modelování podnebí, proudění v oceánech, výpočtu aerodynamických vlastností vozidel. Přestože využití rovnic je nespočetné, mají zásadní problém, neboť do dnes nevíme, zdali mají pro náhodný vstup nějaké řešení. Pro zjednodušení výpočtu existují několik aproximací, které je umožňují aplikovat se zanedbatelnými chybami v modelovacích systémech. [2]

2.2 Popisy tekutin

V teorii mechaniky tekutin existují dva různé pohledy popisu tekutin. Na základě těchto popisů jsou následně založeny algoritmy pro simulaci kapalin, resp. plynů, které jich využívají pro diskretizaci NSE. [1]

2.2.1 Lagrangeův popis

Lagrangeův popis je jeden z popisů kontinua, který si většina nejspíše vybaví. Na tekutinu nahlíží jako na systém částic. Na jednotlivé body tekutiny nahlíží jako částice, které mohou být na základě potřeby různě velké, např. jako molekuly nebo části tekutiny o nějakém objemu. Ke každé z nich Lagrangeův popis přiřazuje pozici $\mathbf{x}$ a rychlost $\mathbf{v}$ a sleduje je, jak se tyto hodnoty v čase mění. [1]

2.2.2 Eulerův popis

Eulerův popis kontinua je na první pohled neintuitivní. Eulerův postup diskretizuje prostor tekutiny na pevně dané oblasti, ve kterých měří vlastnosti tekutiny, např. tlak, rychlost proudu, teplotu, a sleduje, jak se v čase mění.

Přestože tento postup vypadá omezující a složitý kvůli sledování veličin tekutin jen v pevně daných bodech, je v metodách pro simulování tekutin preferovaným popisem kontinua. Hlavní výhodou Eulerova pohledu je jednoduchost výpočtu prostorových derivací jako ∇p nebo $\nabla^2 \mathbf{v}$, které se lépe aproximují v pevné Eulerově mřížce oproti shlukům pohybujících se částic. [1]

Kapitola 3

Vlastnosti vodního povrchu a algoritmy, které je simulují

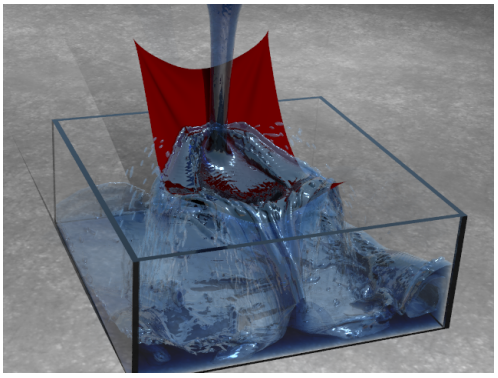
Tato kapitola obsahuje popis vizuálních vlastností vodní hladiny, které v reálném světě lze zpozorovat, a k jejím hlavním rysům jsou vypsané algoritmy, které je simulují. Z pohledu počítačové grafiky lze vlastnosti vody rozdělit do dvou kategorií: na dynamické vlastnosti, které popisují pohyb vodní hladiny, nebo na optické vlastnosti, které popisují interakce světelných paprsků s povrchem vody.

3.1 Dynamické vlastnosti

Dynamické vlastnosti zachycují, jak se vodní hladina pohybuje a jak reaguje na dynamické prostředí. V současnosti nejrealističtějších výsledků je dosaženo simulací Eulerovy vody (nahlíží na vodu Eulerovým popisem), jejíž prostor je rozdělen do alespoň 512^3 buněk. Takto řídké rozdělení obsahuje přes sto milionů neznámých pro vyřešení a s užitím globálních zobrazovacích metod jako *ray-tracing* pro realistickou vizualizaci odrazů, refrakcí a kaustik je výpočetně nemožné simulaci provést v reálném čase. [3]

Proto *real-time* simulace vody vhodné pro aplikace jako hry musí nutně splňovat tyto podmínky:

- být výpočetně rychlé – zlomek 15 ms, který je třeba pro vykreslení jednoho snímku,
- být paměťově nenáročné,
- být stabilní – korektně reagovat i na nefyzikálně pohybující se objekty. [3]



Obrázek 3.1: Ukázka Guendelmanovy *off-line* simulace Eulerovy tekutiny [4]

Jeden přístup, jak se nejvíce přiblížit *off-line* simulaci a stále dodržet podmínky pro *real-time* simulaci ve hrách, je zachovat stejný algoritmus, ale zmenšit rozlišení prostoru simulace. Tento postup ale spíše ubírá na realitě vodního povrchu, neboť se voda nefyzikálně shlukuje a detaily vody jsou z obrazu vynechány. [3]

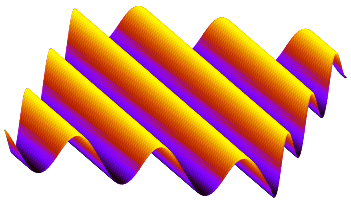
Další z možností, jak splnit výše zmíněné podmínky, je omezit míru interakce s vnějším prostředím, která tvoří největší výpočetní překážku. Na základě významnosti vodních útvarů ve scéně může simulace reagovat na všechny rigidní tělesa nebo u případů, ve kterých voda slouží jako pozadí, opominout jakoukoliv interakci. Podle tohoto principu můžeme dělit metody simulace vodní plochy na:

- procedulární,
- částicové,
- hybridní. [3]

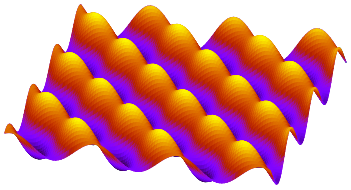
3.1.1 Procedulární metody

Procedulární metody simulují konečné efekty jako vlnění vodního povrchu, které nejsou vyvolány fyzikálními činiteli. Největší výhodou této metody je výpočetní rychlost a versatilita, na druhou stranu ale vodní útvary nereagují na dynamické prostředí.

Na základě těchto vlastností se procedulární metody používají pro vizualizaci rozsáhlých vodních ploch, které ve scéně hrají malou roli, např. jako oceán v pozadí scény.



Obrázek 3.2: Simulace vlnění pomocí jedné sinusoidy



Obrázek 3.3: Simulace vlnění pomocí sčítání sinusoid

3.1.1.1 Simulace podle sinusoid

Mezi prvními, kteří zkoumali procedulární metody pro zobrazení vodního povrchu, byl Nelson L. Max [5] ve své práci „*Vectorized Procedural Models for Natural Terrain: Wave and Islands in the Sunset*," ve které modeluje vlnění povrchu pomocí sinusoid. Na základě horizontální pozice a času manipuloval výškou vrcholů roviny. Výšku jednotlivých vrcholů roviny vyjádřil jako funkci

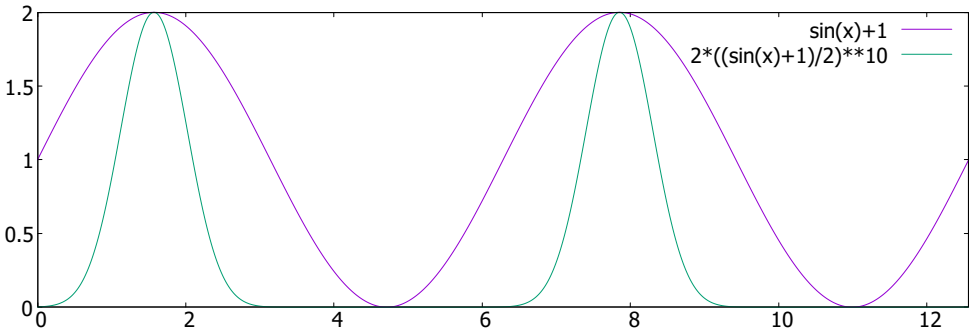
$$h(x, z, t) = -y_0 + A \sin(k_x x + k_z z - \omega t + \varphi), \tag{3.1}$$

kde (x, z) je horizontální pozice vrcholu roviny (v celé práci se bude uvažovat, že kladná osa y směřuje nahoru), t je čas, y_0 je výška hladiny vody v klidovém stavu (při nulovém výskytu vln), A reprezentuje amplitudu vlnění, $\mathbf{k} = (k_x, k_z)$ je vlnový vektor reprezentující směr a rychlost propagace vln, ω je úhlová frekvence a φ je fáze vlny.

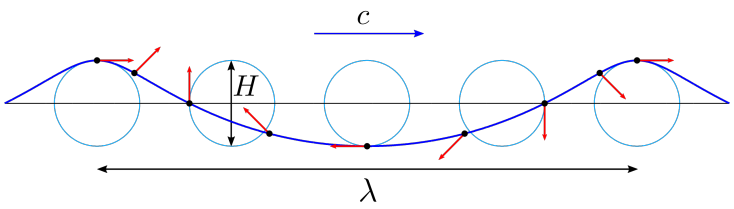
Tento model se ale vlní jen v jednom směru a výsledné vlny vypadají nerealisticky hladce. K dosažení větších detailů lze k funkci přičíst další sinusoidy s odlišnými parametry amplitudy, vlnového vektoru nebo úhlové frekvence:

$$h(x, z, t) = -y_0 + \sum_{i=1}^{N_w} A_i \sin(k_{x_i} x + k_{z_i} z - \omega_i t + \varphi_i), \tag{3.2}$$

kde N_w je celkový počet vln. [5]



Obrázek 3.4: Porovnání mezi vlnami funkcí 3.1 a 3.3



Obrázek 3.5: Gerstnerova vlna s fázovou rychlostí c, vlnovou délkou λ a rozdílem hřebenu a údolí H (CC BY-SA 4.0) [9]

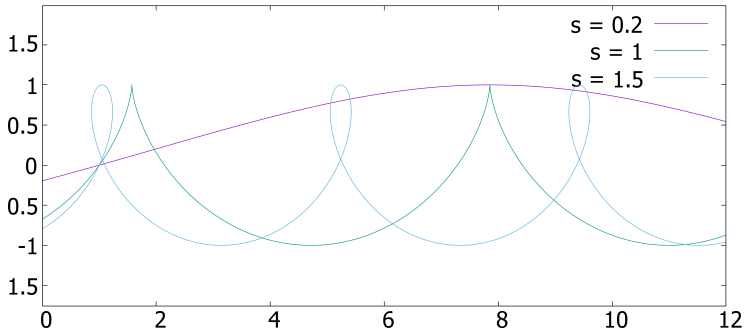
Realistického řešení však za pomocí jen obyčejných sinusoid nelze dosáhnout. Max [5] si všiml, že vlnění oceánu s vyššími amplitudami mají užší hřeben a mělčí údolí, zatímco vrchol a údolí sinusoidy jsou stejně oblé. Mark Finch [6] přišel na řešení tohoto problému, které stále využívá jednoduchých sinusoid. Kromě obyčejných funkcí sinusoid přičítává navíc funkce

$$f_i(x,z,t) = 2\left(\frac{\sin(k_{x_i}x + k_{z_i}z - \omega_i t + \varphi_i) + 1}{2}\right)^k, \tag{3.3}$$

kde $k \in \mathbb{R}^+$ určuje míru, jak má být hřeben úzký.

3.1.1.2 Simulace podle Gerstnerovy vlny

Limitující faktor metody založené na transformaci vrcholů pomocí sinusoid je, že manipuluje s jediným parametrem vrcholu, tj. výškou, a pro realističtější výsledky by bylo třeba manipulovat vrcholy i v horizontální rovině. Tento problém řeší cyklické křivky, tzv. trochoidy, podle kterých Franz Josef Gerstner, německý fyzik, modeloval vlnění hladiny v hlubokých vodách [7]. Trochoidou nazýváme trajektorii bodu pevně spojeného s „kutálející se“ kružnicí po nehybné přímce. [8]



Obrázek 3.6: Porovnání hodnot s ve dvourozměrném prostoru

Dnes v teorii dynamiky tekutin vlnění založené na trochoidních křivkách nazýváme Gerstnerovými vlnami. Mezi prvními, kteří se Gerstnerovými vlnami zabývali, byli Alain Fournier a William T. Reeves [10], kteří transformovali vrcholy roviny podle těchto parametrických rovnic:

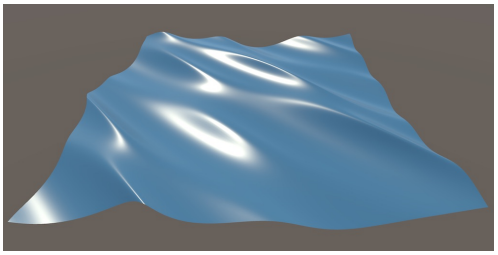
$$\begin{aligned} x &= x_0 + r\cos(k_x x_0 + k_z z_0 - \omega t), \\ y &= r\sin(k_x x_0 + k_z z_0 - \omega t), \\ z &= z_0 + r\cos(k_x x_0 + k_z z_0 - \omega t), \end{aligned} \tag{3.4}$$

kde vektor (x_0, z_0) reprezentuje klidovou pozici vrcholu roviny, r je vzdálenost opisujícího bodu od středu kružnice trochoidní křivky, vektor $\mathbf{k} = (k_x, k_z)$ určuje rychlost a směr propagace vlnění, ω označuje úhlovou frekvenci a t je čas.

Hodnota $s = r|\mathbf{k}|$ určuje, jak strmá bude vlna. Pro hodnotu $s = 0.2$ má vlna tvar jako sinusoida, pro $s = 1$ má tvar cykloidy a pro $s > 1$ dochází samoprotínání trochoidy, a proto by se hodnotám vyšší než jedna pro vizualizaci vodního povrchu mělo vyhýbat. [10] V obrázku 3.6 je porovnání hodnot s ve dvourozměrném prostoru.

Detailnějších výsledků lze získat podobně jako u sinusoid pomocí skládání Gerstnerových vln [6]:

$$\begin{aligned} x &= x_0 + \sum_{i=1}^{N_w} r_i \cos(k_{x_i} x_0 + k_{z_i} z_0 - \omega_i t), \\ y &= \sum_{i=1}^{N_w} r_i \sin(k_{x_i} x_0 + k_{z_i} z_0 - \omega_i t), \\ z &= z_0 + \sum_{i=1}^{N_w} r_i \cos(k_{x_i} x_0 + k_{z_i} z_0 - \omega_i t). \end{aligned} \tag{3.5}$$



Obrázek 3.7: Gerstnerovy vlny v simulaci vodního povrchu podle tutoriálu od Jaspera Flicka [11]

3.1.1.3 Simulace podle Fourierovy transformace

Předešlé dvě metody jsou kvalitními nástroji pro simulaci vodního povrchu a za jejich pomoci lze i s vysokým počtem vln N_w, který se pohybuje v řádech tisíců nebo více, získat fotorealistických výsledků. Vysoký počet N_w představuje však výpočetní překážku. Obvykle by výpočet posunutí podle rovnic 3.2 nebo 3.5 herní *enginy* provedly ve *vertex shaderech*, pro které je však výpočet funkcí sinus a cosinus náročný [12].

Johanson [13] řeší výpočetní náročnost adaptivní metodou, která na základě vzdálenosti kamery a vodní plochy vyřazuje některá vlnění. V případě, že je kamera dostatečně vzdálená od vodní plochy, vyřazuje Johanson vlnění s vysokými frekvencemi. Efektivnějšího výsledku dosáhl Lee [14], který před vykreslením vodní plochy navíc zahazuje vrcholy geometrie hladiny, které jsou mimo rozsah kamery.

V současnosti nejlepším řešením tohoto problému je užití rychlé Fourierovy transformace (FFT), resp. rychlé inverzní Fourierovy transformace (IFFT) podle Jerryho Tessendorfa [15]. Sčítání sinusoid v rovnici 3.2 je ve své podstatě inverzní Fourierova transformace, kde jednotlivé sinusoidy přispívají ke konečnému vlnění. Tessendorf podle statistických dat vlnění oceánů, např. ze satelitních snímků, modeluje výšku vrcholů hladiny jako inverzní Fourierovu transformaci funkce

$$h(\mathbf{x}, t) = \sum_{k} \widetilde{h}(\mathbf{k}, t) e^{i\mathbf{k}\cdot\mathbf{x}}, \tag{3.6}$$

kde $\mathbf{x}$ je pozice vrcholu roviny, funkce $\widetilde{h}$ závislá na čase t obsahuje informaci o amplitudě a fázi sinusoidy $e^{i\mathbf{k}\cdot\mathbf{x}}$ s vlnovým vektorem $\mathbf{k}$. Tessendorf následně podle oceánografických dat vhodně volí funkci $\widetilde{h}$, aby bylo výsledné vlnění nejvíce realistické.

Časová komplexita triviálního sčítání sinusoid v rovnici 3.2, tj. inverzní Fourierovy transformace, je $\mathcal{O}(n^2)$, kdežto užitím algoritmů pro IFFT se redukuje časová komplexita na $\mathcal{O}(n \log n)$ [16].

Obrázek 3.8: Ukázka simulace podle FFT od Asylum Darth [17]

3.1.2 Částicové metody

Procedulární metody jsou stavěny pro vizualizaci rozsáhlých vodních ploch jako oceány a moře. Protože jsou tyto vodní útvary rozlehlé, jakýkoliv zásah buď hráče, či prostředí by měl na výsledné zobrazení hladiny minimální efekt.

V některých případech se ale hladina vody nechová periodicky nebo reakce na dynamické prostředí je právě při zobrazování žádoucí jako u vody fontány nebo kaluží. Tyto detailní vlastnosti vody řeší částicové modely využívající Lagrangeova popisu kontinua. Částicové metody simulují vodní útvary jako systém částic, které reprezentují určitou hmotu kapaliny.

Poprvé, kdo využil částicových systémů pro simulaci tekutiny, byl v roce 1983 Reeves [18]. Reevesův přístup byl však velmi primitivní oproti dnešním variacím, pohyb částic nekorektně modeloval nezávislé na ostatních částicích. V realitě jeho simulace se více blížila k simulaci shlukům jemných částic, např. prachu.

Pro korektní chování je třeba, aby se jednotlivé částice mezi sebou přitahovaly a odpuzovaly. Obecně lze tuto sílu mezi dvěma částicemi vyčíslit jako

$$f(\mathbf{x}_i, \mathbf{x}_j) = F(|\mathbf{x}_i - \mathbf{x}_j|) \cdot \frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|}, \tag{3.7}$$

kde $\mathbf{x}_i$, resp. $\mathbf{x}_j$ je pozice částice i, resp. částice j, funkce F je velikost síly [1].

Fyzikálně korektního chování lze získat při volbě funkce F jako hodnotu Lennardovy–Jonesovy síly, která se používá v simulacích molekulární dynamiky [1]

$$f(\mathbf{x}_i, \mathbf{x}_j) = \left(\frac{k_1}{|\mathbf{x}_i - \mathbf{x}_j|^m} - \frac{k_2}{|\mathbf{x}_i - \mathbf{x}_j|^n} \right) \cdot \frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|}, \tag{3.8}$$

kde k_1, k_2, m, n jsou ovládací parametry. Obvyklou volbou je $k_1 = k_2 = k$, $m = 4$ a $n = 2$. Tuto sílu např. využili Miller a Pearce [19] pro modelování viskózní tekutiny.

Pro *real-time* simulace může být tento přístup pomalý. V případě, že vodní útvar obsahuje n částic, musí se funkce f evaluovat $\mathcal{O}(n^2)$. Existují ale optimalizace, které rozdělují částice do pravidelné mřížky, čímž redukují složitost na $\mathcal{O}(n)$. Přestože ji lze vypočítat poměrně rychle, simulace se vůbec neopírá o NSE. [1]

3.1.2.1 Simulace podle Smoothed Particle Hydrodynamics

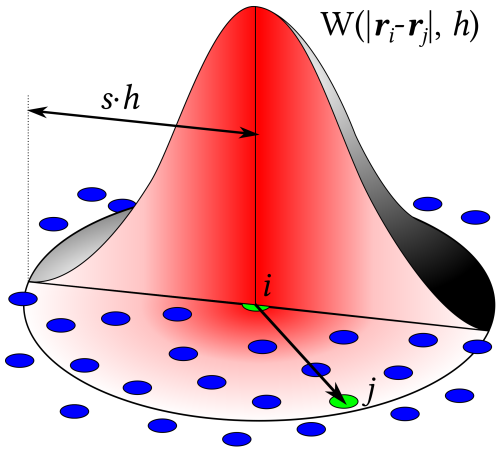
Aktuálně nejrozšířenější částicová metoda založená na výpočtu sil podle NSE vychází z Monaghanova článku „*Smoothed Particle Hydrodynamics*“ [20]. Monaghan vychází z interpolační metody *Smoothed Particle Hydrodynamics* (dále jako SPH), kterou zprvu Lucy vyvinul pro astrofyzické problémy [21]. Lucyho metoda je dostatečně obecná, aby ji bylo možné využít v simulaci tekutin pro výpočet skalárních polí hustoty nebo tlaku. Celý algoritmus simulace vody podle SPH je shrnut tímto pseudokódem [22]:

```
Nastavení počátečních hodnot částic.
Během celé doby simulace opakuj:
    Pro každou částici
        Najdi jeho sousedy.
        Pomocí SPH vypočti hustotu.
        Z hustoty aproximuj tlak.
        Pomocí SPH vypočti tlakovou sílu.
        Pomocí SPH vypočti sílu viskozity.
        Aplikuj na částici externí síly.
        Detekuj kolize.
    Integrace podle času.
    Vykreslení částicového systému.
```

SPH lze v počítačové grafice přirovnat ke konvoluci, kde místo barvy pixelů pracuje s vlastnostmi částic. SPH distribuuje obecnou veličinu pro danou částici jako sumu vážených hodnot veličin jejích sousedů. Hodnota obecné veličiny A pro částici i je podle SPH [23]

$$A_i(\mathbf{x}_i) = \sum_j m_j \frac{A_j}{\rho_j} W(|\mathbf{x}_i - \mathbf{x}_j|, h), \tag{3.9}$$

kde $\mathbf{x}_i$, resp. $\mathbf{x}_j$, je pozice částice i, resp. j, j iteruje přes všechny částice, m_j je hmotnost, A_j je hodnota veličiny A a ρ_j je hustota částice j. Funkce $W(|\mathbf{x}_i - \mathbf{x}_j|, h)$ se nazývá *smoothing kernel* s poloměrem h, což je analogií jádra u konvoluce. Obrázek 3.9 ilustruje, jak *smoothing kernely*, které mají obvykle tvar podobné Gaussovu rozdělení, váží hodnoty částic. Největší váhu W přiřadí částicím nejblíže středu, tj. částici i, a nejnižší částicím vzdálené od středu délkou h. Validní *smoothing kernely* musí navíc splnit normalizační podmínku [1, 23]:



Obrázek 3.9: Ilustrace vážení *smoothing kernelem* (CC BY-SA 4.0) [24]

$$\int W(r)dr = 1. \tag{3.10}$$

NSE mimo samotných hodnot veličin dále pracuje s gradientem (operace ∇), jehož aplikace na rovnici 3.9 ovlivní jen *smoothing kernel*

$$A_i(\mathbf{x}_i) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(|\mathbf{x}_i - \mathbf{x}_j|, h), \tag{3.11}$$

výpočet Laplace (operace ∇^2) obdobně jako u gradientu působí jen na jádro W

$$A_i(\mathbf{x}_i) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(|\mathbf{x}_i - \mathbf{x}_j|, h). \tag{3.12}$$

Obvyklá volba jádra W, které lze využít až na dvě výjimky téměř pro každou interpolaci veličin v simulaci tekutin, je *kernel poly6* [1, 23]

$$W_{poly6}(r,h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{jindy} \end{cases}. \tag{3.13}$$

Poly6 ale nekorektně aproximuje síly tlakového pole, v simulaci se částice pod vysokým tlakem nefyzikálně shlukují [23], neboť ∇W_{poly6}, jež je nutné pro výpočet síly způsobené rozdílem tlaků (rovnice 2.2), se ve středu jádra blíží k nule, což vede k zanedbání odpuzujících sil. Desburn řešil tento problém nahrazením jádra *poly6* za *kernel* [25]

$$W_{spiky}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & \text{jindy} \end{cases}. \tag{3.14}$$

Poslední nepřesnost jádra *poly6* se vyskytuje při výpočtu silové pole vytvořené viskozitou tekutiny. Výsledek jeho Laplacovy operace, od kterého se odvíjí síla vyvolaná viskozitou (rovnice 2.2), se u středu jádra pohybuje v záporných číslech, což může vést k nekorektnímu zvýšení rychlosti částice a následné nestabilitě simulace [23]. Na základě tohoto nedostatku Müller používá při výpočtu silového pole vyvolané viskozitou jádro

$$W_{viskozita}(r,h) = \frac{15}{2\pi h^3}\begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \leq r \leq h \\ 0 & \text{jindy} \end{cases}. \tag{3.15}$$

Pro konečnou simulaci vody je třeba vyřešit soustavu NSE za pomocí pole rychlosti, pole hustoty a pole tlaku, které SPH interpoluje z vlastností částic. Díky aplikaci Lagrangeova popisu kontinua se některé rovnice výrazně zjednoduší [23]. Rovnice o zachování hmoty 2.1 je automaticky splněna, neboť počet částic zůstane při simulaci konstantní. Dále lze zjednodušit v rovnici o zachování hybnosti 2.2 výraz $\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}$, protože se částice pohybují s tekutinou, může se z výrazu vynechat člen $\mathbf{v} \cdot \nabla \mathbf{v}$, který reprezentuje změnu rychlosti v případě, že se částice pohybuje *skrz* tekutinu. Konečné NSE pro SPH částicový systém zní takto

$$\rho \frac{\partial \mathbf{v}}{\partial t} = -\nabla p + \rho \mathbf{g} + \mu \nabla^2 \mathbf{v}. \tag{3.16}$$

Pohyb částice je následně určený silami vyvolané rozdílem tlaků $-\nabla p$, viskozitou tekutiny $\mu \nabla^2 \mathbf{v}$ a gravitací $\rho \mathbf{g}$ (na vodu působí i externí síly, ty lze zakomponovat s gravitací do $\mathbf{g}$).

Částice si ale udržují informace jen o své pozici, rychlosti a hmotnosti, zbylé vlastnosti si musí simulátor spočítat sám. Každá z nich určuje nějaký objem $V_i = \frac{m_i}{\rho_i}$, zatímco hmotnost zůstává během celé simulace konstantní, ρ_i se musí evaluovat v každé vykreslovací smyčce (demonstrace různých hustot lze vidět v obrázku 3.10) [23]. Hustota ρ_i částice i na pozici $\mathbf{x}_i$ je pak podle SPH 3.9 určená jako

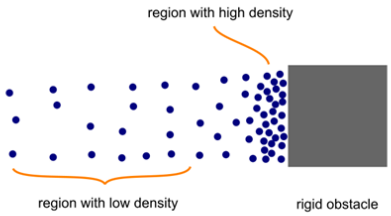
$$\rho(\mathbf{x}_i) = \sum_j m_j \frac{\rho_j}{\rho_j} W(|\mathbf{x}_i - \mathbf{x}_j|, h) = \sum_j m_j W(|\mathbf{x}_i - \mathbf{x}_j|, h). \tag{3.17}$$

Dále je třeba vypočítat tlak p_i na pozici $\mathbf{x}_i$, který lze aproximovat pomocí stavové rovnice ideálního plynu [23]

$$p_i = k\rho_i, \tag{3.18}$$

kde k je konstanta plynu závislá na teplotě a ρ_i je hustota částice. Desburn navrhl úpravu rovnice 3.18, která přispěla k větší stabilitě simulace [25]

$$p_i = k(\rho_i - \rho_0), \tag{3.19}$$



Obrázek 3.10: Demonstrace rozdílů v hustotě tekutiny [26]

kde ρ_0 označuje klidovou hustotu tekutiny. Protože síla vyvolaná rozdílem tlaků závisí na gradientu tlaku, je posun všech hodnot o konstantu stále validním řešením.

S výše uvedenými informacemi lze nakonec spočítat síly působící v tekutině $-\nabla p$ a $\mu\nabla^2\mathbf{v}$. Na základě SPH 3.9 je tlaková síla působící na částici i rovna

$$\mathbf{F}_i^{tlak} = -\nabla p(\mathbf{x}_i) = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(|\mathbf{x}_i - \mathbf{x}_j|, h). \tag{3.20}$$

Bohužel takto vypočítané síly nejsou symetrické (nesplňují Newtonův třetí zákon) [23], tento problém je nejvíce patrný, když simulace obsahuje jen dvě částice. Protože gradient *smoothing kernelu* je v středu, tj. na pozici částice i, roven nule, částice použije pro výpočet jen tlak částice j a naopak. Müller [23] řeší tento problém jednoduchým průměrováním hodnot p_i a p_j

$$\mathbf{F}_i^{tlak} = -\nabla p(\mathbf{x}_i) = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(|\mathbf{x}_i - \mathbf{x}_j|, h). \tag{3.21}$$

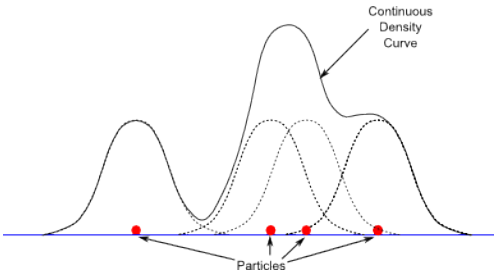
Podobný problém nastává při výpočtu sil způsobené viskozitou tekutiny

$$\mathbf{F}_i^{viskozita} = \mu\nabla^2\mathbf{v}(\mathbf{x}_i) = \mu\sum_j m_j \frac{\mathbf{v}_j}{\rho_j} \nabla^2 W(|\mathbf{x}_i - \mathbf{x}_j|, h), \tag{3.22}$$

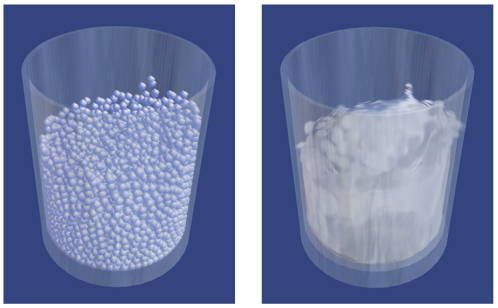
protože se rychlosti u každé částice liší. Symetrizace, kterou navrhl Müller [23], spočívá v tom, že síla způsobena viskozitou závisí pouze na rozdílech rychlostí částice od svých sousedů, což přirozeně vede k

$$\mathbf{F}_i^{viskozita} = \mu\nabla^2\mathbf{v}(\mathbf{x}_i) = \mu\sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(|\mathbf{x}_i - \mathbf{x}_j|, h). \tag{3.23}$$

Gravitace a externí síly jsou přímo aplikovány na samotné částice. Se všemi těmito informacemi lze vypočítat NSE, které lze formulovat Newtonovým druhým zákonem



Obrázek 3.11: Interpolace pole hustoty podle SPH [26]



Obrázek 3.12: Müllerova simulace vody založená na SPH [23]

$$\mathbf{a}_i = \frac{D\mathbf{v}_i}{Dt} = \frac{\mathbf{F}_i}{\rho_i}, \tag{3.24}$$

kde $\mathbf{a}_i$ je zrychlení, ρ_i je hustota a $\mathbf{F}_i$ součet sil působící na částici i. Pro získání rychlosti $\mathbf{v}_i$ stačí integrovat podle času zrychlení $\mathbf{a}_i$. Müllerův simulátor [23] pro integraci používal algoritmus Leap–Frog [27], který integruje podle konstantních časových rozestupů, pro lepší výsledky navrhl algoritmy založené na Courant–Friedrichs–Lewyho podmínce [25].

Posledním krokem je vizualizace částic. Bez žádných speciálních metod by částice měly pouhý tvar kuliček, které nejsou nijak spojeny. Řešením je tzv. *point splatting* [28], který na základě množiny bodů vytvoří jednolitý útvar.

3.1.3 Hybridní metody

Předešlé dvě metody představují extrémní případy pro simulaci vodní hladiny. Procedulární metody nereagují na okolní prostředí, ale jsou výpočetně rychlé. Částicové systémy na druhou stranu počítají s dynamickým prostředím, ale jsou výpočetně náročné. Hybridní metody jsou kombinací dvou, které zároveň zjednodušují problematiku vodní hladiny jako procedulární metody a zachovávají některé fyzikální vlastnosti.

Pro simulaci větších vodních ploch jsou částicové metody zbytečně náročné. Většina částic se bude nacházet pod hladinou a ve stojatých vodách

se jejich pozice nebude výrazně měnit. Proto by bylo pro ně zbytečné počítat NSE. Hybridní metody tuto výpočetní překážku řeší podobně jako procedulární, kdy problematiku vodní hladiny redukují z trojrozměrného prostoru do dvourozměrného [3] a transformují výšku vrcholů podle aproximací NSE.

Chentanezova hybridní metoda využívá *Shallow Water Equations* (SWE) [29], soustavu parciálních diferenciálních rovnic, které zjednodušují NSE do 2D výškových map. *Real-time* výpočet SWE je ale stále drahá operace. Bridson zjednodušil její integraci výměnou za stabilitu simulace [29].

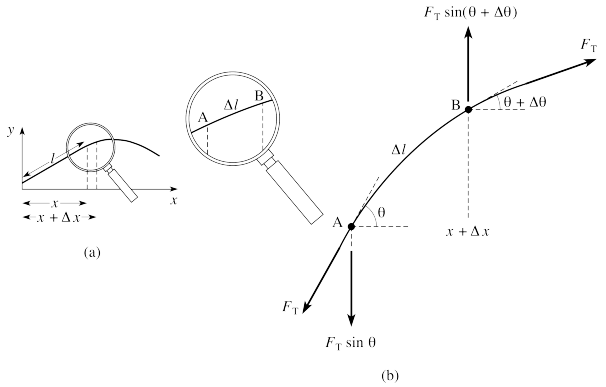
Další variantou je modelování hladiny podle vlnové rovnice [1], hyperbolické parciální diferenciální rovnice. Vlnová rovnice popisuje řadu vln od vlnění struny kytary až po vlnění vodní hladiny.

3.1.3.1 Simulace podle vlnové rovnice

Vlnová rovnice nahlíží na vodní hladinu jako na napnutou elastickou membránu, která nebere v potaz, co se pod ní děje. Protože problematiku redukuje do dvourozměrného prostoru, místo NSE využívá pro fyzikálně korektní pohyb pouhý Newtonův druhý zákon. Simulace je založena na transformaci výšky vrcholů roviny, pro kterou potřebuje dvě 2D pole `u[i,j]` a `v[i,j]`, které si uchovávají výšku a rychlost buňky v bodě `[i,j]`. Pro mřížku o velikosti `N*N` s rozestupy `h` pak algoritmus vypadá takto [3]

```
float u[N,N]
float v[N,N]
float unew[N,N]
Pro každé i, j proveď u[i, j] = u0[i, j] a v[i, j] = 0.
Během celé simulace opakuj
    Pro každé i, j proveď
      f = c^2*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]-4u[i,j])/h^2
      v[i,j] += f*dt;
      unew[i,j] = u[i,j] + v[i,j]*dt
    Pro každé i, j proveď
      u[i,j] = unew[i,j]
    Vykresli mřížku s výškovou mapou u.
```

Pole `u0` inicializuje výšku jednotlivých bodů roviny, `dt` značí časový rozdíl mezi vykreslovacími smyčkami.



Obrázek 3.13: Vlna v 1D: (a) vibrující struna, (b) zvětšení segmentu AB [31]

Vlnová rovnice a síla, která vyvolává pohyb, jsou v následující části vyvozené. Pro jednoduchost je problém redukován na 1D, kde vypadá vlna jako napnutá vibrující struna (viz. obrázek 3.13), postup lze analogicky převést do 2D. Jednodimenzionální vlnová rovnice předpokládá, že jsou splněny tyto podmínky:

1. hmotnost na jednotku délky je konstantní,
2. struna je perfektně elastická a neklade odpor ohybu,
3. gravitační síla je zanedbatelná (dominantní silou je ta, která strunu napíná),
4. výchylky struny v horizontální ose, ose x, jsou zanedbatelné (struna se hýbe jen ve vertikální ose y),
5. vertikální výchylky jsou malé a sklony struny (v obrázku 3.13 úhly θ a $\theta\Delta\theta$) od osy x jsou velmi malé. [30]

Nechť funkce $u(x,t)$ udává, jak je struna na pozici x v čase t vychýlena od osy x. V případě, že je struna v klidovém stavu, struna splývá s osou x. Dále nechť na segment struny AB, část struny od x do $x+\Delta x$, kde $\Delta x \to 0$, působí napínací síly F_{T_1} a F_{T_2}. Za předpokladu, že se segment AB málo vychyluje od osy x, lze jejich velikosti sil považovat za identické, $F_{T_1} = F_{T_2} = F_T$ [32]. Kvůli čtvrtému předpokladu se horizontální síly vyruší a stačí spočítat síly působící ve vertikální ose.

$$F_{vert} = -F_T \sin(\theta) + F_T \sin(\theta + \Delta\theta) \tag{3.25}$$

Síla $-F_T \sin(\theta)$ působí u bodu A směrem dolů a síla $F_T \sin(\theta + \Delta\theta)$ u bodu B opačným směrem. Díky pátému předpokladu lze aproximovat $\sin(\theta)$ a $\sin(\theta + \Delta\theta)$ jako velikosti jejich úhlů. [32]

$$F_{vert} = -F_T\theta + F_T(\theta + \Delta\theta) = F_T\Delta\theta \tag{3.26}$$

Po získání síly působící na segment AB lze aplikovat Newtonův druhý zákon.

$$F_{vert} = ma \tag{3.27}$$

$$F_T\Delta\theta = (\Delta x\mu)\frac{\partial^2 u}{\partial t^2} \tag{3.28}$$

μ značí hmotnost na jednotku délky. Zrychlení a lze přepsat jako druhou derivaci funkce u podle t. Dále pro $\Delta x \to 0$ platí, že

$$\mathrm{tg}(\theta) = \frac{\partial u}{\partial x} \tag{3.29}$$

a pro derivaci celé rovnice 3.29 podle x

$$\frac{1}{\cos^2(\theta)}\frac{d\theta}{dx} = \frac{\partial^2 u}{\partial x^2}. \tag{3.30}$$

Pro aproximaci hodnoty $\cos^2(\theta)$ se opět užije pátého předpokladu, že sklony θ a $\theta + \Delta\theta$ jsou velmi malé, podle kterého platí, že $\cos(\theta) \approx \cos(\theta + \Delta\theta) \approx 1$. Následně protože $\Delta x \to 0$, lze přepsat rovnici 3.30 jako

$$\frac{\Delta\theta}{\Delta x} = \frac{\partial^2 u}{\partial x^2}. \tag{3.31}$$

Po dosazení hodnoty $\Delta\theta$ z rovnice 3.31 do rovnice 3.28, vydělení hodnotou Δx a prohození členů získáme 1D vlnovou rovnici [32]

$$\frac{\partial^2 u}{\partial t^2} = \frac{F_T}{\mu}\frac{\partial^2 u}{\partial x^2}. \tag{3.32}$$

Pro zdůraznění pozitivní hodnoty $\frac{F_T}{\mu}$ se v literatuře uvádí vlnová rovnice jako

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2}. \tag{3.33}$$

Obecné řešení rovnice 3.33 lze zapsat jako

$$u(x,t) = f(x - ct) + g(x + ct), \tag{3.34}$$

kterou lze interpretovat tak, že řešením rovnice 3.33 jsou vlny, které se šíří doprava podle funkce f a doleva podle funkce g rychlostí c. Ve slově rovnice 3.33 popisuje, že síla působící na vlákno, příp. membránu, je přímo úměrná změně jejího sklonu. S rostoucím sklonem segmentu vlákna, případně membrány, roste i síla, která na něj působí a naopak. Při konstantním sklonu nedochází k žádné změně síly [1, 3]. Ve více rozměrném prostoru vlnová rovnice zní

$$\frac{\partial^2 u}{\partial t^2} = c^2\nabla^2 u. \tag{3.35}$$

Pro vyřešení rovnice 3.35 ji nejdříve Bridson a Müller [1, 3] rozkládají na dvě parciálně diferenciální rovnice prvního řádu

$$\begin{aligned} \frac{\partial u}{\partial t} &= v, \\ \frac{\partial v}{\partial t} &= c^2 \nabla^2 u. \end{aligned} \tag{3.36}$$

Následně 2D prostor diskretizují do mřížky s rozestupy o velikosti h, jejíž buňky si uchovávají informace o výšce hladiny u_t a rychlost v_t v čase t, a pomocí metody konečných diferencí (FDM) získají $\nabla^2 u$. Nakonec řešení diferenciální rovnice aproximují podle Eulerovy metody s explicitním krokem. Pak pro jednotlivé buňky v čase $t + 1$ platí [1]

$$\begin{aligned} \nabla^2 u_t[i,j] &= (u_t[i+1,j] + u_t[i-1,j] + u_t[i,j+1] + u_t[i,j-1] - 4u_t[i,j])/h^2, \\ v_{t+1}[i,j] &= v_t[i,j] + \Delta t c^2 \nabla^2 u_t[i,j], \\ u_{t+1}[i,j] &= u_t[i,j] + \Delta t v_{t+1}[i,j]. \end{aligned} \tag{3.37}$$

Parametry Δt, c a h diskretizované vlnové rovnice musí splňovat Courant–Friedrichs–Lewyho podmínku, $c \cdot \Delta t < h$, která říká, že se informace buňky mohou šířit do svého okolí pouze jednou buňkou za časový krok Δt [1, 3]. Všímavý čtenář si může všimnout, že

$$(u_t[i+1,j] + u_t[i-1,j] + u_t[i,j+1] + u_t[i,j-1] - 4u_t[i,j])/h^2 \tag{3.38}$$

je Laplacovské konvoluční jádro.

Integrace Eulerovou metodou požaduje počáteční hodnoty pro u_t a v_t v čase $t = 0$, tj. u_0 a v_0 [1, 3]. Hodnota v_0 je pro všechny buňky rovna nule a hodnota u_0 je inicializovaná nějakou funkcí. Kdyby u_0, tj. počáteční výška hladiny, byla rovna nule, tak by to znamenalo, že hladina je v úplném klidu a k žádnému vlnění nedojde.

Bohužel volba explicitního kroku Eulerovy metody vede k podmíněné stabilitě simulace [1, 3]. Krok $\Delta t v_{t+1}$ nekorektně předpokládá, že během časového kroku Δt zůstává rychlost v_{t+1} konstantní. Navíc jsou všechny výše zmíněné výpočty aproximacemi. V případě, že se špatně odhadne rychlost, může výška hladiny nekonečně nárůst nebo klesnout. Müller řeší tento problém dvěma způsoby. Jedním je oříznout výšky podle vybraného maximálního sklonu nebo další možností je zmenšit velikost rychlosti v_{t+1} koeficientem $s < 1$

$$u_{t+1}[i,j] = u_t[i,j] + s \cdot \Delta t v_{t+1}[i,j]. \tag{3.39}$$

Mřížka má dále konečný rozměr $n \times n$, a proto je nutné definovat krajní podmínky (pro výpočet Laplace) [1, 3]. Jednou variantou je přiřadit hodnotám mimo rozsah jim nejbližší krajní hodnoty:

$$\begin{aligned} u[-1, j] &= u[0, j], \\ u[n, j] &= u[n-1, j], \\ u[i, -1] &= u[i, 0], \\ u[i, n] &= u[i, n-1]. \end{aligned} \tag{3.40}$$

Takto zvolené řešení má v simulaci efekt reflektování vln od hranic mřížky. Další možností je spojit levou hranu s pravou hranou a horní hranu s dolní hranou mřížky. Výsledný efekt je takový, že vlny vycházející z levého, resp. horního, kraje vstupují do mřížky zpět z pravého, resp dolního, kraje. Toto chování může být žádoucí pro skládání vodních povrchů, tzv. *tiling*. [1]

Přestože simulace podle vlnové rovnice vypadá jako univerzální metoda pro simulaci vodního povrchu, má tato metoda i nevýhody, které pramení z redukování prostoru vody z 3D na 2D. Protože si tato metoda udržuje u každého bodu roviny právě jednu hodnotu jeho výšky, nelze touto metodou zobrazit jevy jako lámání vln, které vzniká na břehu vodních útvarů. Další chybou je nepřesnost vlnění v mělkých vodách, kde je dalším důležitým faktorem vzdálenost hladiny od dna útvaru, která tato metoda opomíjí. [3]

3.2 Optické vlastnosti

U každého objektu ve světě způsob, jak je naše oko vnímá, zcela závisí na světle, jak interaguje s materiálem objektu. Světlo se skládá z fotonů, které se přímočaře šíří od svého emitoru všemi směry. Při dopadu fotonu na objekt může na základě jeho materiálu dojít ke třem možnostem: pohlcení fotonu, odražení nebo transmisi. Co lidské oko následně uvidí, závisí na světelných paprscích, které oko zasáhnou. [33]

Chování světla pro realistické vykreslení scén zachycují globální zobrazovací metody. Tyto metody jsou však výpočetně náročné, a proto je využívají hlavně *off-line* aplikace. Protože velká většina paprsků naše oko (kameru) vůbec nezasáhne, *real-time* aplikace světlo obvykle zjednodušují takovým způsobem, že světelné paprsky vrhají z pozice oka a na základě toho, kam po odrazech dopadnou předají oku barvu objektu [33]. Přestože současné grafické karty začínají podporovat *ray-tracing*, velká většina zařízení a softwaru stále využívá k zobrazování rasterizaci.

Kromě vrhání paprsků je další drahá operace výpočet průsečíků paprsků s objekty. Tato operace je obzvlášť náročně zakomponovatelná do renderovací *pipeliny* kvůli restrikcím omezující, jaké informace lze zaslat grafickým kartám. Proto většina algoritmů pro osvětlování scén je založena na aproxi-

maci průsečíků. Následující metody jsou proto zvolené tak, aby je bylo možné zakomponovat do renderovací *pipeliny* rasterizačních API.

3.2.1 Odrazy světla

Voda má reflektivní vlastnosti, a proto barva její hladiny závisí na tom, odkud světelný paprsek, který následně zasáhne přijímač, přijde. Paprsek světla, resp. paprsek vrženého z přijímače (oka, kamery), se od ideálně reflektivního povrchu odráží pod stejným úhlem jako, pod kterým parsek dopadá. [34]

Odrazy scény na vodní hladině lze levně simulovat pomocí tzv. *cubemap*. Jedná se o druh textur, které mají tvar krychle. Hlavní výhodou užití *cubemap* je její způsob vzorkování, které místo 2D souřadnic využívá 3D vektor. Jestliže se střed *cubemapy* (krychle) napozicuje na počátek souřadnic, vzorkovací 3D vektor od počátku směřuje na jednu jeho stranu a podle toho, kde vektor protne stranu, převezme fragment barvu textury. Pro dosažení efektu odrazu scény pak stačí jako vzorkovací vektor zvolit vektor odrazu. [35]

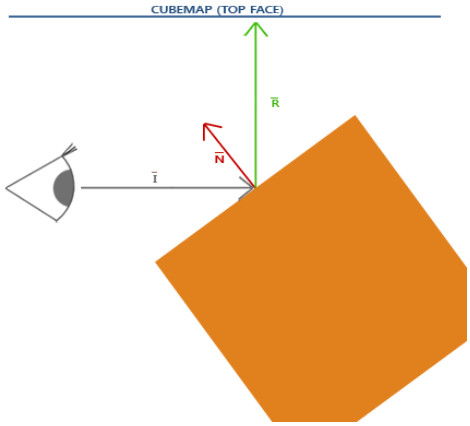
Cubemapy ale nedokáží reflektovat geometrie scény, neboť se skládá ze statických obrazů. Řešením může být např. *cubemapy* vykreslit dynamicky, tj. vykreslit šest stran krychle a namapovat je na *cubemapu*. Takto vytvořená textura by ale reflektovala objekty, jako by stály v dáli.

Pro geometrie, které jsou v blízkém okolí reflektujícího objektu, řeší Lettier [36] odrazy světla pomocí zjednodušené metody *ray-tracingu*, která počítá průsečíky v prostoru obrazu. Lettier nejdříve vykreslí scénu z pohledu kamery do tří pomocných textur: do první vykreslí scénu, jakoby by byla textura displej monitoru, do druhé místo barvy fragmentu uloží 3D souřadnice interpolovaných vrcholů a do další hodnoty jejich normál. Následně vyšle z každého pixelu, resp. texelu, prostoru obrazu, resp. textury, paprsek, a jestli se paprsek odrazí od reflektujícího objektu, prochází Lettier podél odraženého paprsku po malých inkrementech a kontroluje, zda neprotrl nějakou geometrii ve scéně. Při průniku si daný fragment, z kterého vyšel paprsek, zaznamená barvu zasažené geometrie uložené v textuře.

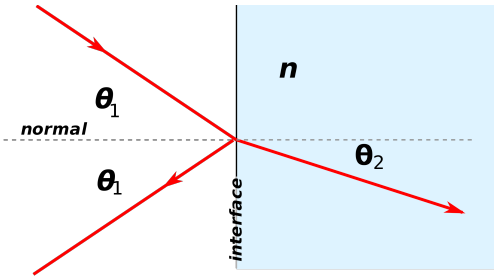
Limitováním prostoru na prostor obrazu přináší nějaké vady této metody. Tou největší je právě prostor, od kterého se paprsky odráží, neboť kamera vždy směřuje dopředu, nemůže metoda odrážet předměty za kamerou. Podobný problém může nastat i vepředu kamery, kdy odražený paprsek narazí na geometrii v nějakém rohu, který ale kamera nevidí.

3.2.2 Refrakce

Hladina vody ale nemá perfektně odrazivý povrch. Některé světelné paprsky prochází skrz ni, a proto ji vnímáme jako průsvitný materiál. Objekty pod hladinou ale vidíme zkresleně důsledkem tzv. refrakcí [34]. Světelné paprsky se při přechodu médií lámou, neboť prochází dvěma prostory, ve kterých má světlo



Obrázek 3.14: Vzorkování odrazů scény, kde odražený pohledový parsek směřuje na horní stranu *cubemapy* (CC BY 4.0) [35]



Obrázek 3.15: Odraz a refrakce paprsku (CC BY-SA 3.0) [39]

různé rychlosti. V hustších prostorech se světlo pohybuje pomaleji, zatímco v řidších rychleji. Lom světla lze popsat Snellovým zákonem

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}, \tag{3.41}$$

kde θ_1 značí úhel dopadu příchozího světla, θ_2 je úhel lomeného paprsku (viz obrázek 3.15), n_1 a n_2 jsou refraktivní indexy (IOR) příchozího a odchozího prostředí. [37]

Refrakce lze implementovat podobně jako reflekce podle Lettierovy metody [36, 38]. Místo ale odrazeného paprsku pracuje s refraktovaným. Další modifikace metody reflekcí je, že místo obyčejné scény, kterou si uloží do pomocné textury, si musí vykreslit scénu, ale bez refraktujícího objektu. Důvodem je, že textura slouží pro vzorkování barvy, u reflekcí odražený paprsek může znovu narazit na reflektující objekt, zatímco u refrakcí paprsek prochází skrz objekt, a proto by vzorkovaná barva měla být až ta, která je za refraktujícím objektem.

Obrázek 3.16: Optický efekt popsaný Fresnelovými rovnicemi [40]

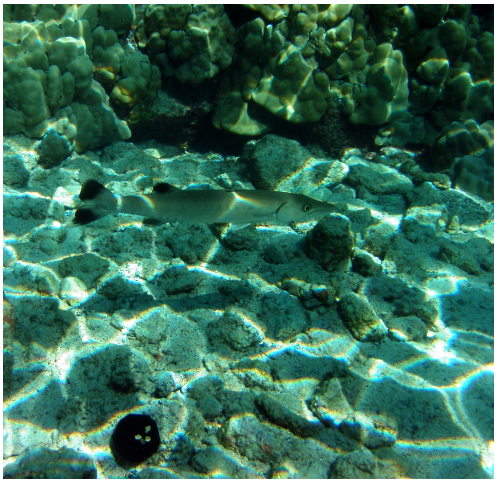
3.2.3 Fresnelovy rovnice

Protože povrch vody je zároveň odrazivý a refraktivní, je třeba zjistit podle kterých světelných paprsků ho obarvit. Tento problém řeší Fresnelovy rovnice. Fresnelovy rovnice podle úhlu pohledu popisují, jak moc refraktované světlo nebo odražené světlo přispívá finální barvě hladiny. Rovnice zní takto

$$\begin{aligned} R_s &= \left(\frac{n_1\cos(\theta_1) - n_2\cos(\theta_2)}{n_1\cos(\theta_1) + n_2\cos(\theta_2)}\right)^2, \\ R_p &= \left(\frac{n_1\cos(\theta_2) - n_2\cos(\theta_1)}{n_1\cos(\theta_2) + n_2\cos(\theta_1)}\right)^2, \\ R &= \frac{R_s + R_p}{2}, \\ T_s &= 1 - R_s, \\ T_p &= 1 - R_p, \\ T &= \frac{T_s + T_p}{2} = 1 - R. \end{aligned} \tag{3.42}$$

Koeficient R popisuje intenzitu odraženého světla, kdežto koeficient T říká intenzitu refraktovaného světla [34, 37]. Úhly θ_1 a θ_2 jsou úhly dopadu příchozího paprsku a lomeného paprsku. Konstanty n_1 a n_2 jsou IOR příchozího a odchozího prostředí. Protože koeficienty R a T závisí na polarizaci světla [37], které rozlišujeme na s-polarizaci a p-polarizaci, jsou hodnoty vyčísleny jako průměry hodnot R_s a R_p, resp. T_s a T_p.

V realitě lze tento optický fenomén nejvíce vidět u mělkých stojatých vod (viz obrázek 3.16). Místa, na která se díváme pod malým úhlem (θ_1 v obrázku 3.15), vidíme spíše průhledně, zatímco v místech, na která se díváme pod velkým úhlem, vidíme spíše odraz okolního prostředí.



Obrázek 3.17: Kaustika pod hladinou vody (CC BY-SA 3.0) [44]

3.2.4 Útlum světla

Potom, co světlo protne hladinu, jeho paprsek prochází ve vodě dvěma transformacemi, může dojít k rozptylu paprsku nebo k absorpci světla. Absorpci způsobuje nejen délka dráhy paprsku a látky obsažené ve vodě, ale také mikroorganismy žijící v ní [41]. Jev, který je predominantním důsledkem absorpce, je např. distorze barevného spektra [37]. Na základě obsahu vody jsou různé vlnové délky pohlceny. Např. v moři kvůli vysokému obsahu soli dochází k vysoké absorpci barev s dlouhými vlnovými délkami jako červená nebo u vody s vysokým obsahem chlorofylu je pohlcováno vše kromě zelených barev [42]. Protože mnoho faktorů hraje roli v absorpci, musí se chování značně zjednodušit, aby je počítače stihly vypočítat. Baboud a Décoret [43] modelují záření s vlnovou délkou λ odraženého z bodu ve vodě p_w do bodu na hladině p_s jako

$$L_\lambda(p_s, \omega) = \alpha_\lambda(d) L_\lambda(p_w, \omega) + (1 - \alpha_\lambda(d)) L_{d\lambda}, \tag{3.43}$$

kde ω značí směr od p_w do p_s, $L_\lambda(p_w, \omega)$ je odchozí záření z bodu p_w ve směru ω, d je vzdálenost mezi p_w a p_s, konstanta $L_{d\lambda}$ reprezentuje záření z rozptylu světla, $\alpha_\lambda(d)$ je koeficient exponenciálního útlumu závislý na d

$$\alpha_\lambda(d) = e^{-a_\lambda d}, \tag{3.44}$$

kde a_λ je útlumová konstanta závislá na vlastnostech vody. Protože obraz počítače pracuje v RGB barevném spektru, musí se rovnice 3.43 aplikovat pro jednotlivé kanály RGB

$$L_{RGB}(p_s, \omega) = (L_R(p_w, \omega), L_G(p_w, \omega), L_B(p_w, \omega)). \tag{3.45}$$

3.2.5 Kaustiky

Kaustika je optický jev, který vzniká díky reflektivním a refraktivním vlastnostem vody. Vytvářejí komplexní vzory světla způsobené konvergencí odražených nebo lomených paprsků světla do jednoho bodu [45]. Běžně je lze vidět např. při osvětlení sklenice nebo hladiny moře (viz obrázek 3.17).

Zobrazování kaustik je bohužel náročná operace, neboť pro její simulaci je třeba vypočítat průsečíky paprsků s objekty ve scénách. V případě, že objekty scény jsou jednoduché útvary jako kvádr nebo koule, lze takové průsečíky jednoduše analyticky vypočítat [46], ale tyto tvary se jen ojediněle vyskytují v hrách. Shah proto simuluje kaustiky skrz pomocnou mapu kaustik [45]. Všechny výpočty pro aproximaci průsečíku počítá v prostoru obrazu. Výhodou této metody je, že dokáže reagovat na různě složité objekty a také na dynamické prostředí. Shahův algoritmus lze rozdělit na tyto části:

1. **Získání souřadnic přijímacích objektů:** Shah nejdříve vykreslí přijímací objekty (nebo přijímače), tj. objekty, na kterých se kaustika může zobrazit, z pohledu světelného zdroje do textury, které místo barvy ukládá světové souřadnice.

2. **Získání souřadnic a normál refraktujícího objektu:** Obdobně jako u prvního bodu refraktující objekt, v tomto případě hladinu vody, vykreslí do jedné textury, ale místo barvy uloží světové souřadnice fragmentu. Kromě souřadnic se do další textury navíc vykreslí hodnoty normál povrchu refraktujícího objektu.

3. **Vytvoření mapy kaustik:** Jednotlivé fragmenty refraktujícího objektu, které lze vidět z pohledu světla, promítne po aplikování Snellova zákona na přijímací objekty a zaznamená, na jaké místa přijímacích objektů body dopadly.

4. **Konečné vykreslení:** Každý přijímací objekt promítne do souřadnicového systému světla, aby mohl vypočítat texturovací souřadnice. Nakonec aplikuje mapu kaustik na přijímací objekty za pomocí vypočítaných souřadnic a vykreslí konečnou scénu. [45]

Hlavní krok algoritmu je třetí bod, a proto je v této části rozveden do většího detailu. Nejpodstatnější část tvorby mapy kaustik se nachází ve *vertex shaderu*. Nejdříve Shah vykreslí body [45], které spolu vytváří mřížku o velikosti textury z druhého bodu, tj. o velikosti textury souřadnic a normál refraktujícího objektu. Účelem této velikosti je, aby pro každý takovýto bod náležel jeden texel textury, a v případě, že bod, resp. texel, náležel refraktujícímu objektu, promítne ho na přijímací objekty. Ve *vertex shaderu* by tento postup vypadal takto

```
Pro každý vrchol V proveď
    Jestli vrchol V patří refraktujícímu objektu:
        r = lomený směr příchozího světla
        P = OdhadPrůsečíku(pozice V, směr r, textura pozic přijímačů)
        Vrať P.
    Jinak
        Označ, že vrchol V je nevalidní.
        Vrať pozici V.
```

Funkce `OdhadPrůsečíku` provádí promítnutí vrcholu `V` ve směru `r` na přijímací objekty. Shah pro odhad průsečíku `P` vrcholu `V` a přijímacích objektů využívá vztahu [45]

$$P = V + d \cdot \mathbf{r}, \tag{3.46}$$

kde V je světová pozice vrcholu `V`, P je světová pozice `P`, $\mathbf{r}$ je směr `r` a d je délka vektoru $\mathbf{r}$. Úloha odhadu průsečíku je pak podle rovnice 3.46 přeformulována na odhad délky d, protože kdyby byla hodnota d rovna vzdálenosti P od V, pak by rovnice 3.46 popisovala reálný průsečík `P`.

Shahův konečný odhad průsečíku P_2 je vyčíslen dvěma iteracemi vztahu 3.46. Nejdříve za d dosadí hodnotu jedna. Poté vypočítá pozici $P_1 = V + 1 \cdot \mathbf{r}$, jehož souřadnice převede do prostoru světla a následně promítne na texturu přijímacích objektů. Promítnutím vrcholu P_1 zjistí jeho světovou souřadnici na přijímači. Jako poslední odhad d poté zvolí vzdálenost vrcholu P_1 od V. V pseudokódu vypadá funkce `OdhadPrůsečíku` takto [45]

```
OdhadPrůsečíku(float3 V, float3 r, texture2D receivers)
    P_1 = V + 1 * r
    // převod souřadnic P_1 do systému souřadnic pohledu světla
    lightP_1 = lightViewProjectionMatrix * float4(P_1, 1)
    // promítnutí bodu P_1 na přijímací objekty
    uvP_1 = 0.5 * (P_1.xy / P_1.w) + (0.5, 0.5)
    // zjistění světových souřadnic P_1
    worldP_1 = texture(receivers, uvP_1)

    P_2 = V + distance(V, worldP_1.xyz) * r
    lightP_2 = lightViewProjectionMatrix * float4(P_2, 1)
    uvP_2 = 0.5 * (P_2.xy / P_2.w) + (0.5, 0.5)
    worldP_2 = texture(receivers, uvP_2)
    Vrať worldP_2.
```

Kapitola 4

Technologie pro real-time simulaci

Technologie je rozhodujícím faktorem pro výběr algoritmu pro *real-time* simulaci vody. Podle Müllera by její výpočet měl v herních *enginech* trvat jen malý zlomek 15 ms, které je třeba pro vykreslení jednoho snímku scény [3]. Simulace lze do jisté míry provádět na procesoru počítače, ale pro větší vodní útvary je přínosné provádět kalkulace paralelně na grafické kartě.

4.1 Renderovací knihovny

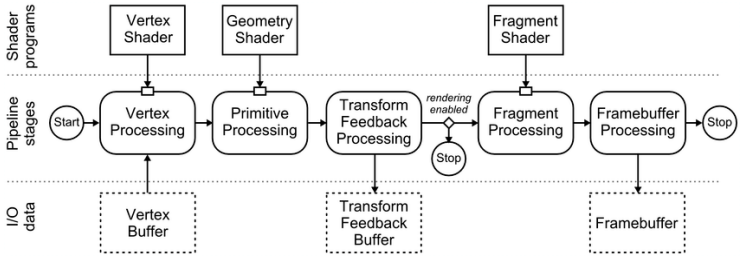
Aby bylo možné předat data o vodní ploše a informace, jak ji vykreslit na obrazovku, grafickým kartám, se používá v programech tzv. grafických API. Grafické API nabízí rozhraní, které umožňuje uživateli manipulovat obrazem skrz grafické karty.

Velká většina metod pro simulaci vody je založena na transformaci geometrie. Tato operace je běžná pro grafické karty, a proto má během vykreslování vyhrazenou fázi, při které lze s pozicemi vrcholů geometrie manipulovat skrz speciální program běžící na grafické kartě *vertex shader*.

Podobně lze jednoduše zakomponovat i simulaci optických vlastností do procesu vykreslování. Kromě pozic grafické karty pracují i s barvami geometrií. Pro její manipulaci je ve vykreslovací *pipelině* vyhrazena část nazývaná *fragment processing*.

4.1.1 OpenGL

OpenGL je víceplatformové grafické API, pro přesnost se jedná o specifikaci grafického API. OpenGL specifikuje, jak a co jednotlivé funkce mají provádět, na základě těchto požadavků si výrobci grafických karet a vývojáři operačních systému implementují vlastní řešení daných funkcí. [47]



Obrázek 4.1: Diagram zjednodušeného průběhu renderovací *pipeliny* OpenGL [48]

Dřívější verze OpenGL, tj. před verzí 3.2, využívaly tzv. fixní *pipelinu*, pro vykreslování obrazu. Její funkce byly téměř omezené jen na renderování jednoduchých primitiv jako čáry, body nebo trojúhelníky a nedávaly programátorům žádnou volnost pro jejich modifikaci. [47]

Od verze 3.2 a výše OpenGL změnilo přístup vykreslování obrazu. Všechny fixní funkce *pipeliny* nahradilo programovatelnými verzemi, které umožnily vývojářům mít větší kontrolu nad výsledným obrazem. Toho bylo docíleno pomocí tzv. vlastních *shader* programů. [47]

Proces renderování skrz OpenGL lze rozdělit do dvou hlavních fází: transformace 3D geometrie do 2D prostoru obrazu a vybarvení jednotlivých pixelů obrazu. [47] Zjednodušený průběh je znázorněn v diagramu 4.1.

Nejdříve se z aplikační části programu zašlou data o geometrii skrz tzv. *vertex buffer* (VBO) grafické kartě. Data minimálně obsahují informace o pozicích vrcholů objektu a volitelně i vlastní dodatečné informace jako hodnoty jejich normál nebo jejich barvy.

Od této chvíle všechny operace probíhají paralelně na grafické kartě. Nad každým vrcholem z VBO proběhne povinná část *shader* programu tzv. *vertex shader*. *Vertex shader* může pracovat se všemi informacemi jednoho vrcholu, tj. nemá přístup k datům okolních vrcholů. Ve *vertex shaderu* se provedou nad vrcholem vlastně definované operace, ale povinně musí předat grafické kartě jeho finální pozici.

V další fázi dochází z předaných vrcholů k sestavení primitiv (body, úsečky, trojúhelníky). Poté lze i s primitivy manipulovat skrz *geometry shader*, volitelné části *shader* programu. Vygenerovaná primitiva lze dále uložit do *transform feedback bufferu* [49].

Poté dojde k rasterizaci primitiv z předešlé fáze, která diskretizuje primitiva na fragmenty (potenciální pixely obrazu). Nad fragmenty proběhne povinná část *shader* programu, tzv. *fragment shader*. *Fragment shadery* manipulují zejména s barvami fragmentů, které na konci musí předat grafické kartě.

Nakonec po operacích viditelnosti fragmentů jako testování hloubky je výsledný obraz vykreslen do *framebufferu*.

4.1.2 Vulkan

Vulkan je nízkoúrovňové víceplatformové grafické API od vývojářů OpenGL. Vulkan oproti OpenGL dává ještě větší kontrolu nad procesy při vykreslování obrazu za účelem optimalizace běhu programu. Procesy jako řízení vláken a paměti, které by pro OpenGL prováděly *drivery* grafických karet, má u Vulkanu zodpovědnost aplikační část programu. [50]

4.1.3 DirectX

DirectX je alternativa OpenGL od společnosti Microsoft. Oproti OpenGL je DirectX uzavřený na platformy s operačními systémy od společnosti Microsoft, dále kromě vykreslovacích funkcionalit obsahuje i rozhraní pro práci se zvukem, sítí a vstupními daty od uživatele. Princip vykreslování je stejně jako u OpenGL založený na grafické *pipelině* [51]. Do verze DirectX 11 se principově podobal OpenGL, ale od verze DirectX 12 změnil svoji filozofii podobně jako Vulkan, který dává uživateli větší kontrolu nad hardwarem [50].

4.2 Herní enginy

Jestli je hlavním cílem vyprodukovat hru nebo *real-time* aplikaci, programování vlastního herního *enginu* může být časově neefektivní. OpenGL, DirectX a Vulkan jsou jen grafické API, na kterých jsou založeny renderovací *enginy*, což tvoří jen malou část herních *enginů*. Kromě nich obsahuje herní *engine* AI, fyzikální, kolizový *engine* a nástroje pro práci se vstupy od uživatele aplikace, a proto by bylo časově výhodné použít z některých komerčních herních *enginů*. Lze je využít nejen pro hry, ale obecně i pro ostatní *real-time* aplikace a v současnosti i ve filmovém průmyslu. [52]

4.2.1 Unity

Unity je víceplatformový herní *engine* s hlavním skriptovacím jazykem C#. Největší výhodou Unity je počet platforem, pro které lze vyvíjet. Lze jej využít jak ve webových, tak i v mobilních aplikací nebo pro vývoj konzolových her a dalších. V současnosti je populárním nástrojem pro mobilní platformy nebo pro VR/AR aplikace [53].

Protože herní *engine* Unity podporuje různou škálu platforem, obsahuje abstraktní vrstvu nad grafickými API, která obsahuje OpenGL, DirectX nebo další jako Metal. Unity pro programování svých *shader* programů používá zejména jazyk HLSL, proprietární jazyk DirectX, ale lze využít i jazyka určeného pro OpenGL a Vulkan GLSL. Výhodou HLSL je jeho univerzálnost, protože Unity nabízí jeho překlad skrz svůj kompilátor do ostatních jazyků. [54]

4.2.2 Unreal Engine

Unreal Engine je v oblasti herních *enginů* hlavní konkurentem Unity. Oproti Unity využívá pro práci programovací jazyka C++. Cílové skupiny pro Unreal Engine jsou díky jeho rychlosti a nástrojům pro fotorealistické zobrazení obrazu spíše aplikace jako AAA hry, které jsou graficky náročné. [53]

Podobně jako Unity používá Unreal Engine více grafických API a ve svých *shader* programech používá HLSL, které volitelně může přeložit skrz svůj *HLSL Cross Compiler* do optimalizovaného GLSL programu. [55]

Kapitola 5

Shrnutí analýzy

Tato kapitola je věnována shrnutí metod pro simulaci vody a nástrojů, pomocí kterých je lze implementovat. Na základě předešlé analýzy jsou následně zvolené algoritmy a nástroje pro implementační část bakalářské práce. Na závěr jsou zde uvedeny funkční a nefunkční požadavky kladené na prototyp simulace a aplikace testovací scény.

5.1 Volba algoritmu

Procedulární metody jsou určené pro vykreslení rozsáhlých vodních ploch, některé jeho implementace, zejména ty založené na IFFT, mají výsledné vodní hladiny nerozpoznatelné od těch reálných. Na druhou stranu nedovolují stejnou míru interakce jako ostatní.

Metody založené na částicových systémech výměnou za výpočetní náročnost se snaží řídit co nejvěrohodněji fyzikálním zákonům. Dokážou reagovat na externí síly a při vysokém počtu částic lze dosáhnout fotorealistických výsledků. Bohužel kvůli jejich výpočetní náročnosti se je vyplatí využít jen pro malé vodní útvary. Při volbě této metody pro útvary jako moře nebo rybníky nemusí být simulace dostatečně rychlá pro *real-time* zobrazování.

Hybridní metody jsou kompromisem mezi částicovými systémy a procedulárními metodami. Problematiku redukují do 2D prostoru jako procedulární, ale zachovávají jistou míru interakce s dynamickým světem. Simulace pomocí vlnové rovnice se navíc řídí i fyzikálními zákony, i když bere v potaz jen hladinu vody, výsledné vizuální provedení je dobrou aproximací vodní plochy. Oproti částicovým systémům se její výpočet tolik neškáluje s větší rozlohou vodního povrchu, a proto je lze využít pro simulaci jak menší, tak i větších vodních útvarů. Tyto důvody mne přesvědčily pro volbu hybridní metody založené na výpočtu vlnové rovnice pro implementaci do mého prototypu.

Většina fyzikálně korektních algoritmů pro simulaci optických vlastností je založena na metodě vrhání paprsků, které bohužel většina současného hard-

waru nepodporuje. Proto byly zvoleny pro simulaci optických vlastností metody aproximující *ray-tracing* pomocí výpočtu průsečíků v prostoru obrazu jako Shahovo mapování kaustik. Tyto metody využívají pro simulaci pomocné textury, v kterých jsou uložené podpůrné data pro výpočet průsečíků.

5.2 Volba technologií

Grafické API nabízí rozhraní pro práci s obrazem skrz grafické karty. Poskytují *shader* programy, které dovolují svým uživatelům manipulovat nejenom barvami objektů, ale také i jeho geometrií. Zatímco herní jádra budují nad grafickými API další podpůrné nástroje vhodné pro vývoj her jako AI nebo fyzikální *enginy*.

Pro implementaci prototypu simulátoru a aplikace testovací scény bylo vybráno grafického API OpenGL a to z následujících důvodů. Prvním důvodem je, že velká většina algoritmů (procedulární, hybridní metody) je založena na transformaci geometrie a jeho následné optické vlastnosti spočívají ve výpočtu barvy povrchu vody podle pozic kamery (oka) a pozorovaného bodu hladiny. Tyto operace jsou stěžejními částmi v simulaci, které lze jednoduše paralelizovat pomocí nabízených nástrojů API *shader* programů. Dalším důvodem pro volbu grafického API oproti hernímu *enginu* je, že výše zmíněné algoritmy potřebují ke svému fungování jen renderovací *pipelinu* grafických API a dalších volitelných nástrojů herního jádra by se v prototypu téměř vůbec nevyužilo. Posledním důvodem je, že i při volbě herního *enginu* by stejně výsledná implementace byla přeložena do grafického API. S použitím samostatného OpenGL má vývojář všechnu kontrolu nad vykreslováním a to bez žádného prostředníka.

Volba mezi grafickými API jako DirectX nebo Vulkan oproti OpenGL je poté jenom z mé preference. OpenGL a DirectX jsou téměř svými ekvivalenty. U grafického API Vulkan je oproti ostatním na druhou stranu příliš nízkoúrovňovým rozhraním a následná implementace v něm by byla příliš náročná a zbytečně zdlouhavá.

5.3 Model požadavků

Tato kapitola obsahuje model požadavků kladený na prototyp simulátoru vodního útvaru a aplikaci testovací scény. Model požadavků je rozdělen do dvou tabulek na funkční a nefunkční požadavky. Tabulka funkčních požadavků je dále rozčleněna na požadavky určené pro samotný simulátor vodní hladiny a aplikační část programu. Jednotlivé záznamy požadavků jsou v tabulkách rozvedeny a je jim přiřazena priorita podle metody *MoSCoW* [56].

Tabulka 5.1: Funkční požadavky simulátoru a aplikace testovací scény

Požadavek	Pohyb vodní hladiny
Popis	Simulátor bude vykreslovat vodní hladinu podle vlnové rovnice.
Priorita	Musí
Požadavek	Interakce s vodní hladinou
Popis	Simulátor umožní svému uživateli měnit tvar vodní hladiny. Tvar vodního povrchu může dále působit na objekty, které na ní leží.
Priorita	Mělo by
Požadavek	Vykreslení kaustik
Popis	Simulátor bude vypočítávat kaustiky formované nerovností vodní hladiny a následně je vykreslí na objekty pod ní.
Priorita	Musí
Požadavek	Reflekce
Popis	Simulátor bude vypočítávat odrazy okolního prostředí a zobrazovat je na povrchu vodní hladiny na základě Fresnelových rovnic.
Priorita	Mělo by
Požadavek	Refrakce
Popis	Simulátor bude počítat zkreslený obraz pod vodní hladinou a na základě Fresnelových rovnic je zobrazí.
Priorita	Mohlo by
Požadavek	Útlum světla
Popis	Simulátor bude regulovat osvětlení objektů na základě jejich vzdálenosti od hladiny vody.
Priorita	Mělo by
Požadavek	Změna parametrů simulátoru
Popis	Za běhu umožní simulátor změnu parametrů jednotlivých vlastností vodní plochy.
Priorita	Mohlo by
Požadavek	Ovládání pomocí klávesnice a myši
Popis	Aplikace umožní uživateli ovládat kameru pomocí myši a klávesnice.
Priorita	Musí
Požadavek	Grafické rozhraní pro změnu parametrů
Popis	Aplikace zobrazí grafické rozhraní pro změnu parametrů simulace a vizualizuje podpůrné data simulace.
Priorita	Mohlo by

Tabulka 5.2: Nefunkční požadavky simulátoru a aplikace testovací scény

Požadavek	Použití grafického API OpenGL
Popis	Pro akceleraci výpočtu simulace a zobrazování testovací scény bude aplikace využívat grafického API OpenGL.
Priorita	Musí
Požadavek	Použití programovacího jazyka C++
Popis	Aplikační část programu bude napsaná v jazyce C++.
Priorita	Musí
Požadavek	Aplikace určena pro platformu Windows
Popis	Cílová platforma pro aplikaci bude Windows 10 a Windows 11.
Priorita	Musí

Kapitola 6

Realizace aplikace

6.1 Návrh aplikace

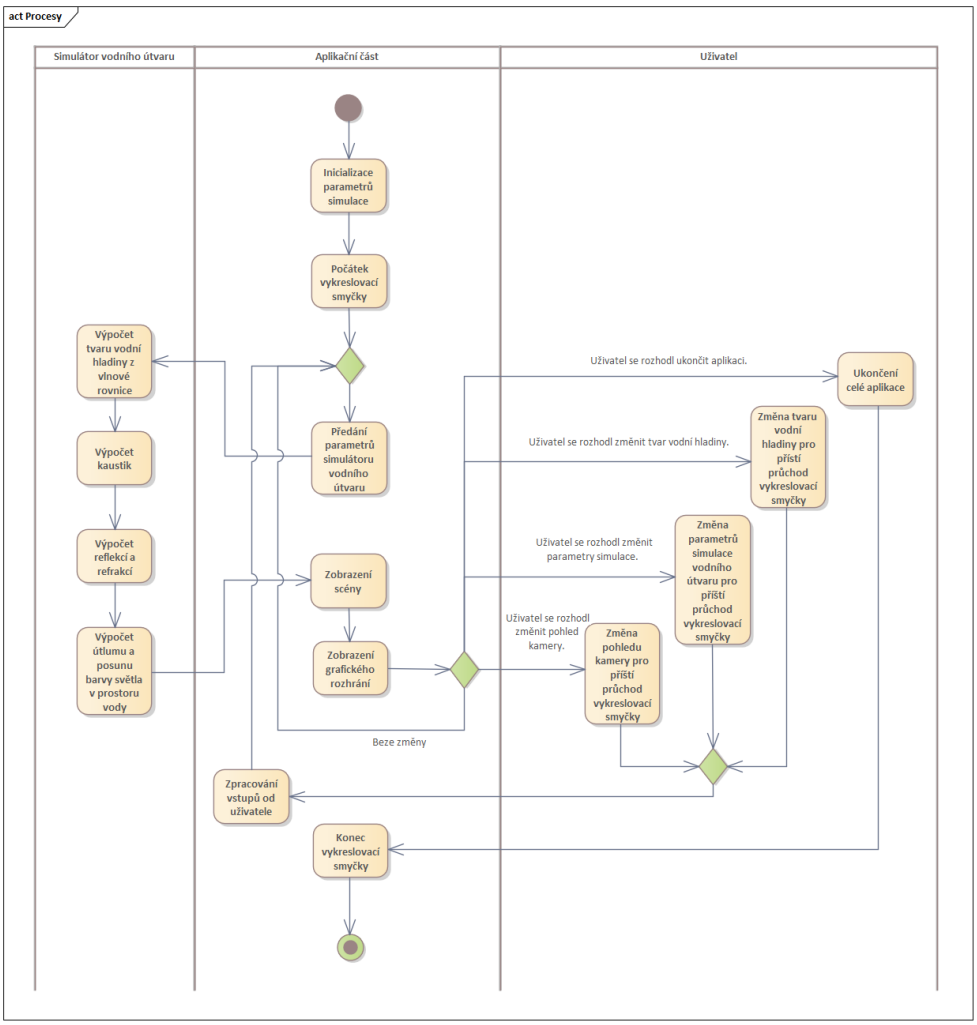
Tato kapitola se věnuje kromě návrhu simulátoru vodního povrchu a jeho dalších optických vlastností, také i struktuře aplikace testovací scény. Nejdříve je zmíněn celkový pohled na aplikaci, ve kterém jsou zohledněné procesy zvolených algoritmů pro zobrazení scény, z kterých následně vychází struktura simulátoru a aplikace.

6.1.1 Procesy simulátoru a aplikace testovací scény

Hlavním algoritmem simulace bude Müllerova metoda [3] simulace vodního povrchu podle vlnové rovnice. Metoda je založena na redukci 3D prostoru vody na 2D prostor. K její implementaci bude třeba předávat dvě 2D pole, které drží informace o výšce hladiny a rychlosti vlnění, mezi pamětí aplikační částí a pamětí grafické karty. Pole s výškami hladiny v jednotlivých bodech vodní roviny bude dále nutným vstupem pro aplikaci optických vlastností vody, konkrétněji k výpočtu povrchových normál. Obyčejná datová struktura pole ale nelze mezi pamětí aplikace a pamětí grafiky jednoduše předávat, řešením tohoto problému bude předávání textur, které mohou mít v grafických kartách podobu polí.

Po provedeném výpočtu výšek hladiny z vlnové rovnice lze dále provést simulaci optických vlastností. Před vykreslením refrakcí bude třeba promítnout kaustiky na přijímací objekty, tj. objekty pod hladinou vody, neboť refrakce zkreslují obraz pod vodou a to včetně kaustik. Shahova metoda formace kaustik [45] vypočítává mapu kaustik pomocí textury pozic a normál jednotlivých vrcholů vodního povrchu a textury pozic vrcholů objektů, na které se kaustiky mohou promítnout. Mapa bude obdobně ve formátu textury.

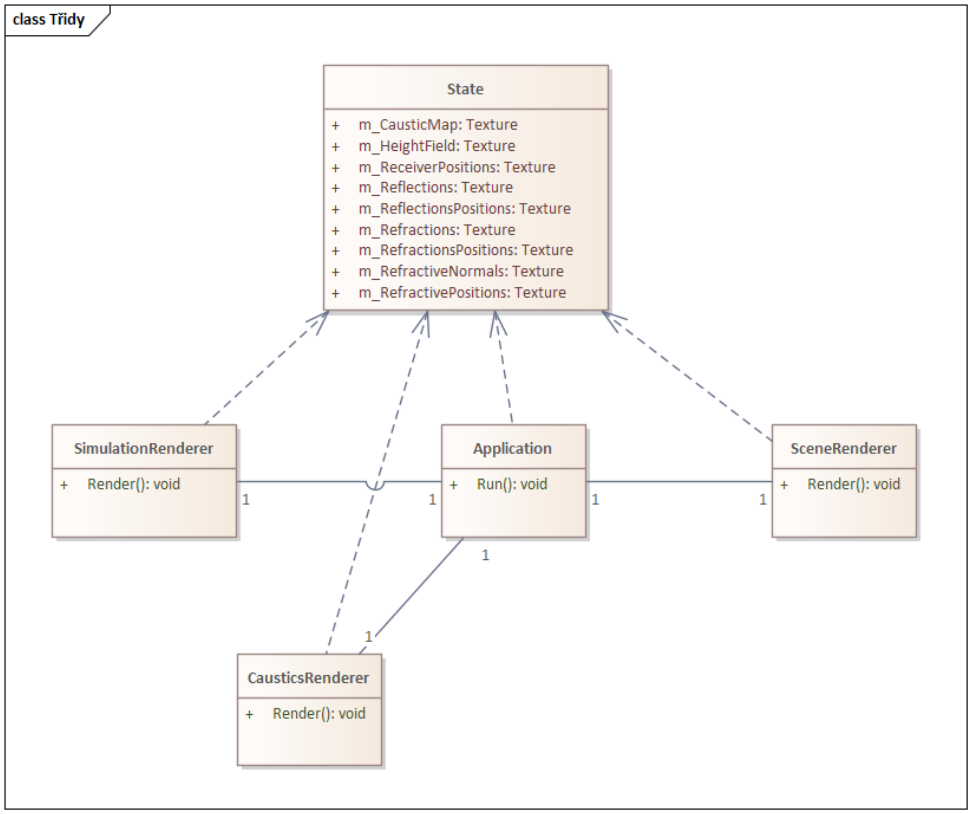
Reflekce a refrakce lze vykreslit stejným algoritmem, neboť principiálně fungují podobně. Jedna varianta využívá pro vzorkování barvy a světla odražený paprsek od povrchu vody, zatímco druhá varianta počítá s lomeným



Obrázek 6.1: Diagram procesů simulátoru a aplikace testovací scény

paprskem. Odhad průsečíků pohledových paprsků se scénou bude implementován stejně jako u Shahovy metody mapování kaustik [45], která přijímá na vstupu texturu s pozicemi vrcholů objektů, vůči kterým se mají průsečíky počítat.

Všechny informace v jednotlivých fázích simulace vodního povrchu se následně předají konečnému programu, který data zpracuje pro vykreslení finální podoby scény. Celá vykreslovací smyčka je znázorněna v diagramu 6.1.



Obrázek 6.2: Diagram tříd simulátoru a aplikace testovací scény

6.1.2 Struktura simulátoru a aplikace testovací scény

Z diagramu procesů 6.1 je patrná i struktura programu. Simulátor vodního útvaru může být rozdělen na dílčí části. Aplikační část programu bude provádět jen finální vykreslení a zpracování vstupu od uživatele programu. Tyto procesy simulátoru a aplikace zachycují následující třídy znázorněné v UML diagramu tříd 6.2.

6.1.2.1 Třída Application

Třída `Application` po zavolání funkce `Run` zahájí inicializaci ostatních tříd a začne provádět vykreslovací smyčku. Ve smyčce bude volat jednotlivé fáze simulace jako simulace vodního povrchu pomocí vlnové rovnice, která bude reprezentována třídou `SimulationRenderer`, výpočet mapy kaustik reprezentována třídou `CausticsRenderer` a vykreslení reflekcí, refrakcí nebo finální scény třídou `SceneRenderer`.

6.1.2.2 Třída State

Třída `State` bude zprostředkovávat všechny informace mezi třídami provádějící simulaci a vykreslování testovací scény `CausticsRenderer`, `SceneRenderer`, `SimulationRenderer` a `Application`. Mezivýsledky simulace, které budou ve formátu textur, se ve třídě `State` budou ukládat a budou následně přístupné všem ostatním třídám k použití.

6.1.2.3 Třída SimulationRenderer

`SimulationRenderer` bude provádět po zavolání funkce `Render` výpočet vlnové rovnice [3]. Následně bude ukládat výšku a rychlost vlnění vodní hladiny do textury `m_HeightField` ve třídě `State`, aby ji v dalších fázích mohly přijmout jako vstupní parametr, např. třída `CausticsRenderer` pro výpočet kaustik.

6.1.2.4 Třída CausticsRenderer

Třída `CausticsRenderer` pomocí výškové mapy `m_HeightField` bude vykreslovat podpůrné textury podle Shahovy metody [45]. Nejdříve vykreslí texturu pozic a normál vrcholů refraktujícího objektu, tj. vodní hladiny. Poté vykreslí pozice vrcholů objektů, na které se kaustiky mohou promítnout. S těmito podpůrnými daty bude následně možné vypočítat mapu kaustik. Pro další fázi simulace bude mapa kaustik uložena ve třídě `State` jako `m_CausticMap`.

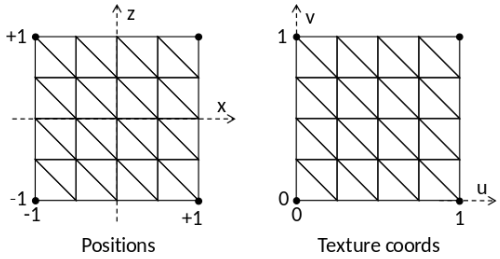
6.1.2.5 Třída SceneRenderer

Konečná fáze simulace bude vykreslení finální scény. Reflekce a refrakce budou zde též implementované, neboť pro jejich vykreslení bude třeba stejných algoritmů jako pro vykreslení finální scény.

Refrakce a reflekce jsem se rozhodl implementovat pomocí Shahova odhadu průsečíků [45]. Po vypočtení průsečíku lomeného, resp. odrazeného, paprsku s geometrií scény v prostoru obrazu převedu souřadnice průsečíku do souřadnicového systému textur. V případě, že převedené souřadnice textury nebudou ležet v prostoru obrazovky, budu vzorkovat barvu pro daný paprsek podle textury prostředí uložené ve formátu *cubemapy*. V případě, že bude ležet v prostoru obrazovky, budu vzorkovat barvu z obrazu scény.

Útlum světla pod vodní hladinou budu implementovat podle vzdálenosti daného vrcholu od klidové hladiny vody. Podle vzdálenosti budu měnit barvu světla podle předem nastavených barev. Blízko hladiny nebude barva světla téměř modifikována, zatímco v hlubších prostorech vody bude barva postupně přecházet na tmavě modrou.

Nakonec lze vykreslit finální scénu. Všechny předešlé fáze a operace se následně budou neustále opakovat ve vykreslovací smyčce.



Obrázek 6.3: Geometrie vygenerované roviny [57]

6.2 Implementace aplikace

Tato kapitola se zaobírá implementací simulátoru vodního útvaru. Protože aplikační část řídí jen nepodstatné části celého programu (zobrazení finální scény, zpracování vstupu nebo předávání instrukcí grafické kartě), bude hlavním obsahem implementace *shader* programů, které v realitě provádí všechny podstatné operace algoritmů simulace.

6.2.1 Simulace hladiny podle vlnové rovnice

Simulace hladiny podle vlnové rovnice spočívá v transformaci výšek jednotlivých bodů roviny. Samozřejmě je možné simulátoru rovinu dodat jako model (ve formátu `.obj`), lze ale také její geometrii procedulárně vygenerovat, což umožňuje větší kontrolu nad detailností vodní plochy zvyšováním rozlišení roviny (počtu vrcholů podél strany roviny). Třída `PlaneGenerator` takovou rovinu generuje, geometrie roviny je generátorem normalizovaná, tj. souřadnice jeho vrcholů jsou škálovány do intervalu $[-1, 1]$, dále vypočítá vrcholům jejich příslušnou texturovací souřadnici, které budou v simulaci důležitým vstupem.

```
void PlaneGenerator::Generate(const int & res,
    std::vector<Vertex> & vert, std::vector<unsigned int> & ind)
```

Metoda `Generate` podle rozlišení `res`, tj. počtu vrcholů podél strany roviny, generuje vrcholy roviny, které uloží do pole `vert`, indexy vrcholů tvořící trojúhelníky, které jsou potřebné pro vykreslení roviny grafickým API OpenGL, jsou dále uloženy v poli `ind`. Výsledná rovina je znázorněna v obrázku 6.3.

Jak bylo v předešlých kapitolách zmíněno, pro implementaci výpočtu simulace hladiny podle vlnové rovnice je třeba v polích `u[i,j]` a `v[i,j]` průběžně ukládat výšky roviny a rychlosti změn výšek v daných bodech roviny. Bohužel pole nelze jednoduše předávat mezi pamětí aplikace a pamětí grafické karty. Řešením je použití textur. Jednotlivé pixely textury následně reprezentují jednu buňku pole. Protože textury pracují v kanálech RGB, lze za-

komponovat pole `u[i,j]` a `v[i,j]` do jedné textury. Výšky budou uloženy v kanálu červené barvy, zatímco rychlost bude uložena v kanálu zelené barvy. Dále je dobré zmínit, že textury obvykle pracují s nezápornými desetinnými hodnotami. Výpočty vlnové rovnice se ale mohou pohybovat v záporných hodnotách. Pro nastavení použití i těchto hodnot je třeba vygenerovat texturu s parametrem `RGBA32F` místo `RGBA`,

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, m_Width, m_Height, 0,
             GL_RGB, GL_FLOAT, 0).
```

Vzorkování mimo rozsah textury je nastaveno, jak Müller navrhoval, opakování hraničních hodnot nastavením parametru `GL_CLAMP_TO_EDGE`

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

Další překážkou je, že do textur nelze přistupovat a zároveň zapisovat, a proto je třeba vytvořit dvě textury simulace hladiny a během simulace mezi nimi prohazovat tzv. metodou ping-pongování. Nechť první textura je označena `T_1` a druhá `T_2`. Ping-pongování textury funguje následujícím způsobem, při prvním průchod `i=0` vykreslovací smyčky bude simulátoru hladiny přiřazena textura `T_1` jako vstupní parametr, z kterých vypočte stav vodní hladiny do dalšího průchodu smyčky, výsledek výpočtu následně uloží do textury `T_2`. V dalším průchodu smyčky `i=1` se prohodí textury `T_1` a `T_2`. Nyní bude textura `T_2` vstupem simulátoru a `T_1` odchozí texturou. Tento proces se neustále opakuje.

Výpočet vlnové rovnice je znázorněn tímto kódem *fragment shaderu*

```
void main()
{
    if (abort) {
        FragColor = vec4(vec3(0.0), 1.0);
        return;
    }

    vec4 old_info = texture(heightField, fTexCoord);

    ...

    float vpdx = texture(heightField, pdx).r + AddDrop(pdx);
    float vndx = texture(heightField, ndx).r + AddDrop(ndx);
    float vpdy = texture(heightField, pdy).r + AddDrop(pdy);
    float vndy = texture(heightField, ndy).r + AddDrop(ndy);

    float nsum = vpdx + vndx + vpdy + vndy;
    float average =  (vpdx + vndx + vpdy + vndy) / 4.0;
    float h = texelSize;
    float old_u = old_info.r + AddDrop(fTexCoord);
    float old_v = old_info.g;

    float offset = average - old_u;
    float maxslope = waveSlope;
    float maxoffset = maxslope * h;

    if (offset > maxoffset)
        old_u = old_u + offset - maxoffset;
    if (offset < -maxoffset)
        old_u = old_u + offset + maxoffset;

    float c = max(waveSpeed, 0.001f);
    float f = c * c * (nsum - 4 * old_u) / (h * h);
    float new_v = old_v + f * deltaTime;
    new_v = new_v - old_u;
    new_v = max((1 - waveDamping), 0.001f) * new_v;
    float new_u = old_u + new_v * deltaTime;
    new_u = new_u;

    vec4 new_info = vec4(new_u , new_v, 0, 1);
    FragColor = new_info;
}.
```

Hodnota `abort` je pravdivá v případě, že bylo nařízeno vynulovat textury výškové mapy, a tím pádem i resetovat hladinu do klidového stavu. Funkce `AddDrop` provádí deformaci hladiny uživatelem, která připíše podle dané pozice výšku okolním bodům podle tvaru sinusoidy. Funkce mění výšku také podle zadané amplitudy `amplitude`, která udává maximální přidanou výšku, a poloměru rozsahu působení `radius`.

```
float AddDrop(vec2 pos)
{
    if (!drop)
        return 0.0;
    float r = max(radius, 0.001f)  * texelSize;
    float val = max(0.0, 1.0 - length(dropPos - pos) / r);
    val = 0.5 - cos(val * PI) * 0.5;
    return amplitude * val;
}
```

Müllerův uvedený algoritmus [3], ale posouvá výšku klidové roviny ve směru deformace. Jestli se za hladinu „zatáhne“ nahoru, posune se celá hladina nahoru, což ale v realitě není možné. Wallace [46] řeší tento problém úpravou rychlosti výškou tak, aby změněná výška konvergovala k výšce klidové hladiny

```
new_v = new_v - old_u.
```

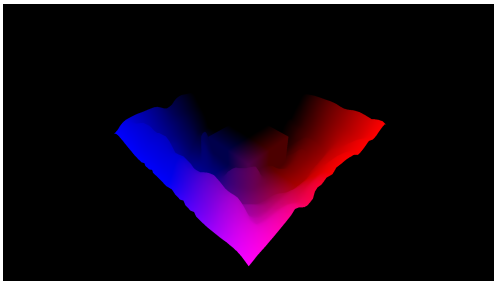
Zbylý program je dále podle Müllerova algoritmu [3] zmíněný v analýze metody podle vlnové rovnice. Deformovaná rovina, tj. vodní hladina, je následně zobrazena podle *vertex shaderu*, který na vstupu přijímá texturu výškové mapy a podle texturovacích souřadnic vrcholu přičte k výšce vrcholu hodnotu textury uložené v červeném kanálu

```
float height = texture(heightField, aTexCoord).r;
vec3 offset = vec3(0.0f, height, 0.0f);

vec4 worldCoord = model * vec4(aPosition + offset, 1.0f).
```

6.2.2 Simulace kaustik

Shahova formace kaustik [45] předpokládá na vstupu tři podpůrné textury. Jedna obsahuje souřadnice a druhá normály interpolovaných vrcholů vodního povrchu ve světovém souřadnicovém systému z pohledu světla, tzn. RGB barva objektu v textuře bude XYZ souřadnice pozice nebo normály.

Normály hladiny, ale nemáme ihned k dispozici, neboť máme jen informace o výšce jednotlivých bodů roviny uložené ve výškové mapě. Naštěstí lze její



Obrázek 6.4: Textura souřadnic interpolovaných vrcholů modelů scény z pohledu kamery

hodnotu z výškové mapy vypočítat jako vektorový součin horizontálních a vertikálních rozdílů výšek. Nechť M je vrchol, pro který počítáme normálu, a t, b, l a r jsou výšky horního, dolního, levého a pravého sousedního vrcholu M. Normála vrcholu M je poté rovna hodnotě n [58]

$$\begin{aligned} v &= (2, r - l, 0), \\ h &= (0, t - b, 2), \\ n &= v \times h. \end{aligned} \tag{6.1}$$

Dále bude třeba vyrenderovat pozice objektů ve světovém souřadnicovém systému, na které se kaustiky mohou promítnout.

Kromě textur je třeba vygenerovat mřížku podobné v obrázku 6.3. Na rozdíl od mřížky používané v simulaci vodní hladiny je tato mřížka bez jakýkoliv hran, které vrcholy spojují. Ve výsledku má mřížka podobu rovnoměrně rozložených vrcholů. Třída `PlaneGenerator` pro výpočet kaustik umí vygenerovat požadovanou mřížku ve zvoleném rozlišení.

Vertex shader, který provádí většinu výpočtu kaustik, poté přijímá podpůrné textury zmíněné výše a mřížku, která má stejné rozlišení jako podpůrné textury, aby jednotlivým bodům mřížky odpovídal právě jeden texel textury. Celý výpočet se řídí podle tohoto kódu

```
void main()
{
    vec4 refractivePosition = texture(refractivePositions, aTexCoord);
    vec4 refractiveNormal = texture(refractiveNormals, aTexCoord);
    vec3 incidentLight = normalize(light.dir);
    vec3 refractedLight  = normalize(refract(incidentLight,
                                    normalize(refractiveNormal.xyz), ETA));
    vec4 p = EstimateIntersection(refractivePosition.xyz, refractedLight);
    if (refractivePosition.a <= 0.001f ||
        refractiveNormal.a <= 0.001f ||
```

```
        p.a <= 0.001f ||
        p.y > refractivePosition.y ||
        dot(p.xyz - refractivePosition.xyz, refractedLight) < 0.01f)
        fValid = 0;
    else
        fValid = 1;

    fDistance = distance(p, refractivePosition);
    fPhi = dot(-incidentLight, normalize(refractiveNormal.xyz));
    gl_Position = projection * view * model * vec4(p.xyz, 1.0);
}.
```

Algoritmus funguje takovým způsobem, že prochází všemi body mřížky, a jestli danému bodu odpovídá nějaký validní texel z textury pozic refraktujícího objektu, tj. vodní hladiny, odhadne podle funkce `EstimateIntersection` používající postup navržený Shahem souřadnice průsečíku pomocí textury pozic přijímacích objektů.

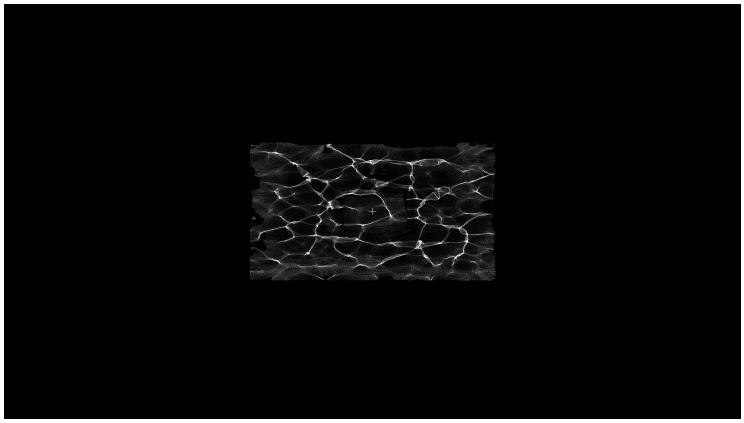
Následuje kontrola validity průsečíku. Nejprve se kontroluje, jestli vrcholu mřížky odpovídá vůbec nějaký vrchol vodní hladiny, dále jestli odhadovaný průsečík vůbec existuje a nakonec jestli neleží odhadovaný průsečík nad hladinou vody.

Výraznost kaustiky je pak určena podle počtu vrcholů, které se promítnou na stejné místo. OpenGL ale standardně překrývá fragmenty na stejných pozicích, a proto by se intenzita kaustik nezvyšovala s vyšší počtem vrcholů se stejným průsečíkem. Pro dosažení tohoto efektu je třeba povolit OpenGL míchání barev funkcemi

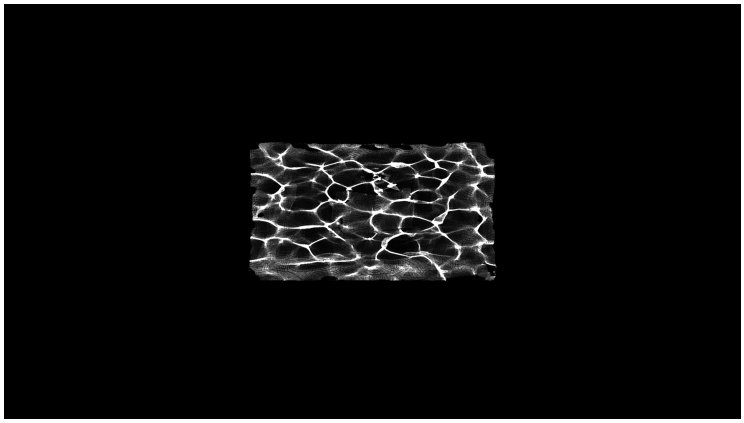
```
            glEnable(GL_BLEND);
            glBlendFunc(GL_ONE, GL_ONE).
```

Funkce `glBlendFund(GL_ONE, GL_ONE)` nastavuje způsob míchání barev tak, aby se po vyslání fragmentu pro vykreslení barva sečetla s barvou, která leží na stejné pozici jako vyslaný fragment. Intenzita kaustik je dále ve *fragment shaderu* tlumena vzdáleností `fDistance`, kterou promítnutý vrchol cestoval, a úhlem dopadu světla `fPhi`.

Objekty nakonec vzorkují mapu kaustik texturovacími souřadnicemi spočítané transformací světových souřadnic vrcholů do souřadnicové systému obrazu světla



Obrázek 6.5: Nefiltrovaná mapa kaustik



Obrázek 6.6: Mapa kaustik filtrovaná Gaussovým rozmazáním

```
vec2 calculateCausticsTexCoord()
{
    vec4 lightClip = orthogonal *
                     lightView *
                     model * vec4(aPosition, 1.0);
    vec2 texC = 0.5 * (lightClip.xy/lightClip.w) + 0.5;
    return texC;
}.
```

Shah dále doporučuje rozmazání mapy kaustik [45], aby nešly vidět artefakty způsobené promítáním konečného počtu vrcholů. Použití obyčejného rozmazání se z mapy kaustik ztrácí vysoké frekvence, které jsou nejvíce vizuálně atraktivní, a proto provádím rozmazání konvolucí s Gaussovým jádrem.

6.2.3 Simulace reflekcí a refrakcí

Reflekce a refrakce jsou implementované obdobně jako formace kaustik. Po vyslání pohledového paprsku a jeho transformaci na lomený, resp. odražený, paprsek se spočítá jeho odhadovaný průsečík pomocí textury pozic interpolovaných vrcholů objektů, od kterých se může obraz refraktovat, resp. odrazit. Odhad je stejně implementován jako u kaustik s výjimkou délky prvního odhadu, který je škálován podle úhlu pohledu

```
vec2 EstimateIntersection(vec3 v,
                          vec3 r,
                          vec3 normal,
                          sampler2D positions)
{
    vec3 p1 = v + firstGuess * (1 - dot(normal, normalize(toCameraDir))) * r;
    vec4 texPt = projection * view * vec4(p1, 1.0);
    vec2 texC = 0.5 * (texPt.xy/texPt.w) + 0.5;
    vec4 recPos = texture(positions, texC);
    if (recPos.a <= 0.0f)
        return vec2(-1.0f);
    float d = distance(v, recPos.xyz);
    vec3 p2 = v + d * r;
    texPt = projection * view * vec4(p2, 1.0);
    texC = 0.5 * (texPt.xy/texPt.w) + 0.5;
    if (texC.x < 0 || texC.x > 1 || texC.y < 0 || texC.y > 1)
        return vec2(-1.0f);
    return texC;
}.
```

Hodnota `v` zde značí bod, od kterého se lomený, resp. odražený, paprsek `r` pohybuje dále.

Jestli hledaný průsečík neexistuje nebo texturovací souřadnice jsou mimo obraz displeje, vzorkují se reflekce a refrakce podle textury prostředí uložené v *cubemapě*.

Po získání texturovacích souřadnic se vzorkuje barva reflekcí a refrakcí podle pomocných textur, ve kterých je uložena scéna. Pro refrakce pomocná textura obsahuje vykreslené objekty, které leží pod hladinou vody, neboť refrakce zkreslují jen, co je pod její hladinou. U reflekcí jsou to naopak jen objekty, které leží nad hladinou.

6.3 Simulace útlumu světla

Útlum světla je naivně implementován podle vzdálenosti objektů od hladiny vody. Podle vzdálenosti dále postupně přechází barva světla podle předem definovaných barev

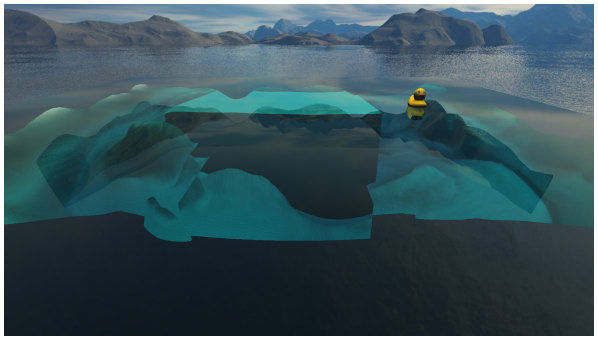
```
vec3 n = vec3(1.0f);
vec3 f = firstStageColor;
vec3 s = secondStageColor;
vec3 t = finalStageColor;

if (diff <= 0)
    lightColor = n;
else if (diff <= firstStage)
    lightColor =
      mix(n, f, (diff - 0.0f)/(firstStage - 0.0f));
else if (diff <= secondStage)
    lightColor =
      mix(f, s, (diff - firstStage)/(secondStage - firstStage));
else
    lightColor =
      mix(s, t, (diff - secondStage)/(3.0f - secondStage));

...

float a = attenuation;
float I = exp(-a * clamp(diff, 0, diff)).
```

6.4 Výsledky implementace

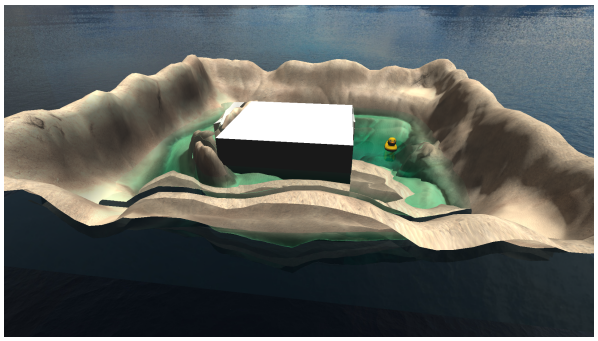
Následující kapitola je zaměřena na ukázku konečných vizuálních výsledků implementace. Mimo vykreslených scén jsou zde probrány nedostatky zvolených algoritmů.

Nejvýraznějších vizuálních chyb vzniká při aproximací velmi citlivých výpočtů jako refrakce a reflekce viditelné v obrázcích 6.7, 6.8, 6.9, 6.10.

V obrázku 6.7 dochází k tomu, že odhadovaný průsečík vychází z prostoru obrazu a následně místo textury scény vzorkuje z textury prostředí v *cube-mapě*. V obrázku 6.8 dochází k tomu, že vyslaný lomený prvek protne scénu ale ve špatném místě, protože reálný průsečík je z pohledu kamery schovaný za nějakou geometrií, která leží před ním. V 6.9 lze vidět chybu míchání barev odražené a refraktované scény podle Fresnelových rovnic způsobenou nepřesností aproximace průsečíků. Odraz kachničky v obrázku 6.10 má siluetu samotné



Obrázek 6.7: Chyba refrakcí způsobena špatným odhadem průsečíku a restrikcí odhadu na prostor obrazu



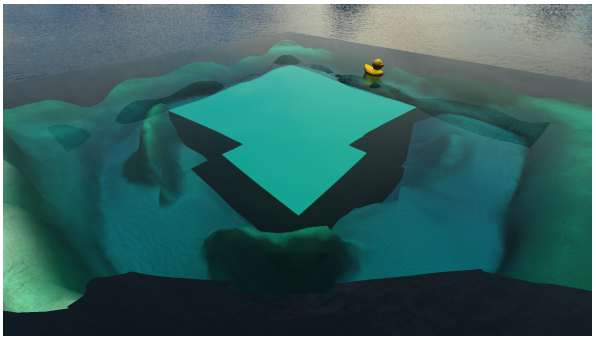
Obrázek 6.8: Chyba refrakcí způsobena schovaným průsečíkem za cizí geometrií

kachničky kvůli způsobu odhadu průsečíku, který ověřuje jeho správnost pomocí textury pozic reflektovaných objektů (zde kachničky).

Tyto nedostatky jsou nejvíce výrazné při pohledu zblízka a rovné vodní hladině. Naštěstí se v realitě vodní hladina neustále hýbe a díky zkreslení obrazu refrakcemi jsou tyto chyby zamaskovány. Korektnost výpočtu průsečíku spočívá ve správném odhadu délky lomeného, resp. odraženého, paprsku. V aplikaci testovací scény lze tento parametr měnit v sekci *Reflections/refractions settings* parameter *First guess*.

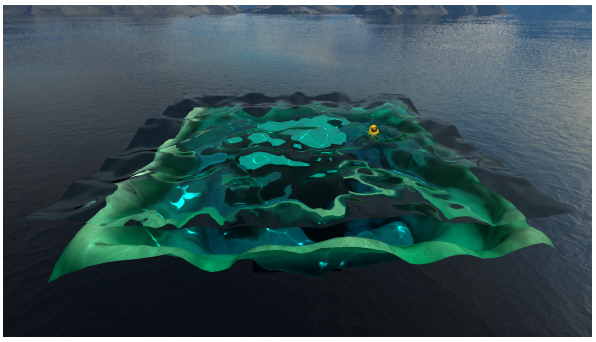
Další vizuální artefakty mohou nastat při volbě silné intenzity kaustik (viz obrázek 6.14), které jsou způsobené Shahovou metodou [45], která promítá jen konečný počet paprsků (vrcholů) skrz vodní hladinu.

Kromě optických chyb jsem narazil i na podstatný nedostatek v integraci vlnové rovnice. Müllerova metoda předpokládá, že časový krok integrace Δt je konstantní v průběhu celé simulace [3]. V případě že, rychlost výpočtu vlnové rovnice je nad ∼100 FPS, simulace zůstává stabilní, ale při ∼40 FPS může jakýkoliv výkyv rychlosti porušit stabilitu simulace a přivést chybu do výpočtu, která se může později projevit.

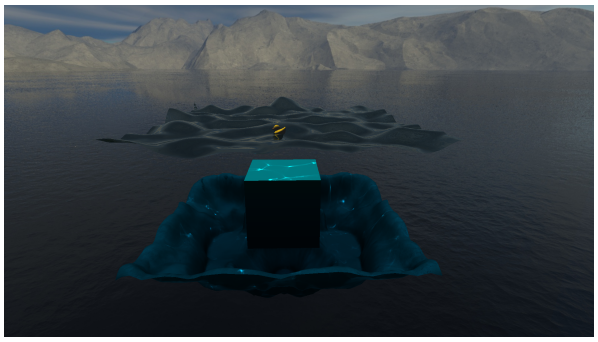


Obrázek 6.9: Chyba refrakcí a reflekcí způsobena nepřesností odhadu průsečíku

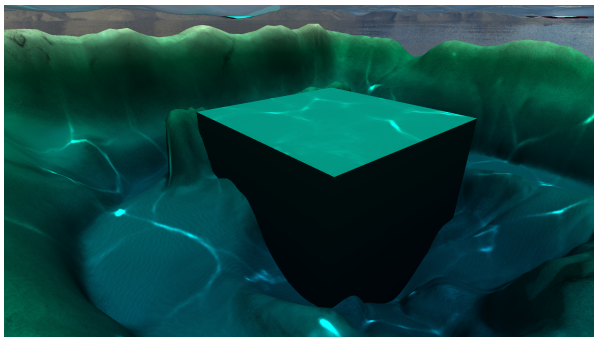
Obrázek 6.10: Odraz kachničky a chyba způsobena texturovací souřadnicí průsečíku ležící mimo obraz scény



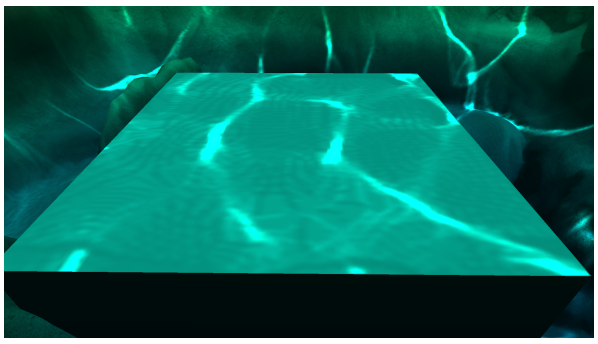
Obrázek 6.11: Testovací scéna se simulátorem vodního útvaru



Obrázek 6.12: Testovací scéna s viditelnými kaustikami



Obrázek 6.13: Formace kaustik na podvodních objektech



Obrázek 6.14: Artefakty mapy kaustik způsobené promítáním konečného počtu vrcholů (paprsků)

Závěr

Jedním z cílů práce bylo provést analýzu současných možností pro simulaci vodního povrchu. Rozbor obsahuje několik metod založených na různých přístupech k simulaci kapalin. Na základě typu scén her a míry interakce vodní plochy s prostředím jsou jednotlivé metody různě vhodné.

Kromě analýzy samotných metod simulace je proveden i rozbor aktuálních technologií pro *real-time* renderované aplikace. Jsou v něm probrány jak grafické API, tak i herní *enginy*.

Dalším cílem bylo vytvořit testovací scénu s vybraným algoritmem z analýzy pro simulaci vodní plochy. Mou volbou byla metoda výpočtu vlnové rovnice a to z důvodu, že zahrnuje kladné vlastnosti jak procedulárních, tak i částicových metod simulace. Simulace podle vlnové rovnice zachovává výpočetní rychlost procedulárních metod a fyzikální korektnost částicových systémů. Optické vlastnosti vodní plochy jako odrazy, refrakce a kaustiky jsou implementovány pomocí zjednodušené metody vrhání paprsků, které za pomoci podpůrných textur aproximují průsečík světelných paprsků v prostoru obrazu. Barva vodní hladiny je dále spočítána podle Fresnelových rovnic, které míchají barvy odrazeného a refraktovaného světla.

Testovací scéna je implementována za pomocí grafického API OpenGL, ale principy simulace jsou lehce přenositelné na ostatní nástroje. Vodní plocha v aplikaci reaguje opticky na prostředí testovací scény odrážením okolní scény, refraktováním obrazu nebo vrháním kaustik na podvodní objekty. Zachovává také jistou míru interakce, která umožňuje uživateli aplikace rozpohybovat vodní hladinu.

Aktuálně simulátor vodní hladiny dokáže vykreslovat vodní plochu, jejíž tvar má obdélníkovou podobu. Protože tyto tvary lze vidět jen u umělých vodních útvarů, bylo by pro další iterace vývoje přínosné, aby simulátor dokázal simulovat i jiné tvary, čehož by nejspíše šlo dosáhnout pomocí masky nad výškovou mapou. Pro realističtější zobrazení hladiny by dále bylo třeba vylepšit odhad průsečíků světelných paprsků zejména pro výpočet reflekcí a refrakcí.

Bibliografie

1. BRIDSON, Robert; MÜLLER-FISCHER, Matthias. Fluid Simulation: SIGGRAPH 2007 Course Notes Video Files Associated with This Course Are Available from the Citation Page. In: *ACM SIGGRAPH 2007 Courses*. San Diego, California: Association for Computing Machinery, 2007, s. 1–81. SIGGRAPH '07. ISBN 9781450318235. Dostupné z DOI: 10.1145/1281500.1281681.
2. NUMBERPHILE. *Navier-Stokes Equations - Numberphile* [online]. 2019 [cit. 2022-04-15]. Dostupné z: https://www.youtube.com/watch?v=ERBVFcutl3M.
3. MÜLLER-FISCHER, Matthias. *Fast Water Simulation for Games Using Height Fields* [online]. London, England, 2008 [cit. 2022-04-15]. Dostupné z: https://www.gdcvault.com/play/203/Fast-Water-Simulation-for-Games. https://archive.org/details/GDC2008Fischer.
4. GUENDELMAN, Eran; SELLE, Andrew; LOSASSO, Frank; FEDKIW, Ronald. Coupling Water and Smoke to Thin Deformable and Rigid Shells. In: *ACM SIGGRAPH 2005 Papers*. Los Angeles, California: Association for Computing Machinery, 2005, s. 973–981. SIGGRAPH '05. ISBN 9781450378253. Dostupné z DOI: 10.1145/1186822.1073299.
5. MAX, Nelson L. Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset. *SIGGRAPH Comput. Graph.* 1981, roč. 15, č. 3, s. 317–324. ISSN 0097-8930. Dostupné z DOI: 10.1145/965161.806820.
6. FERNANDO, R. *GPU gems*. Boston: Addison-Wesley, 2004. ISBN 03-212-2832-4.
7. GERSTNER, František Josef. *Theorie der wellen: samt daraus abgeleiteten theorie der deichprofile*. Gottlieb Haase, 1804.

8. FAKULTA ARCHITEKTURY ČVUT V PRAZE. *Deskriptivní geometrie*. Cyklické křivky [online]. Praha, 2022 [cit. 2022-04-15]. Dostupné z: https://www.fa.cvut.cz/studium/predmety/deskriptivni-geometrie-i-ii/dg_elskripta/krivky/cyklicke_krivky_1.pdf.
9. KRAAIENNEST. *Trochoidal wave* [online]. 2015 [cit. 2022-04-15]. Dostupné z: https://commons.wikimedia.org/wiki/File:Trochoidal_wave.svg.
10. FOURNIER, Alain; REEVES, William T. A Simple Model of Ocean Waves. *SIGGRAPH Comput. Graph.* 1986, roč. 20, č. 4, s. 75–84. ISSN 0097-8930. Dostupné z DOI: 10.1145/15886.15894.
11. FLICK, Jasper. *Waves: Moving Vertices* [online]. 2018 [cit. 2022-04-15]. Dostupné z: https://catlikecoding.com/unity/tutorials/flow/waves/.
12. *On Vertex Shader Performance* [online]. San Francisco, 2008 [cit. 2022-04-15]. Dostupné z: https://paroj.github.io/gltut/Positioning/Tut03%20On%20Vertex%20Shader%20Performance.html.
13. JOHANSON, Claes. *Real-time water rendering: Introducing the projected grid concept*. Lund, 2004. Diplomová práce. Lund University.
14. LEE, Ho-Min; GO, Christian; HYUNG, Won. An Efficient Algorithm for Rendering Large Bodies of Water. In: 2006, sv. 4161, s. 302–305. ISBN 978-3-540-45259-1. Dostupné z DOI: 10.1007/11872320_37.
15. TESSENDORF, Jerry et al. Simulating ocean water. *Simulating nature: realistic and interactive techniques. SIGGRAPH*. 2001, roč. 1, č. 2, s. 5. Dostupné také z: https://www.researchgate.net/publication/264839743_Simulating_Ocean_Water.
16. *Complexity of FFT algorithms (Cooley-Tukey, Bluestein, Prime-factor)* [online]. 2016 [cit. 2022-04-15]. Dostupné z: https://math.stackexchange.com/questions/1704788/complexity-of-fft-algorithms-cooley-tukey-bluestein-prime-factor.
17. ASYLUM DARTH. *OpenGL Ocean Rendering (fast Fourier transform on GPU)* [online]. 2018 [cit. 2022-04-15]. Dostupné z: https://www.youtube.com/watch?v=CeJCNmI-B7s.
18. REEVES, William T. Particle Systems—a Technique for Modeling a Class of Fuzzy Objects. *SIGGRAPH Comput. Graph.* 1983, roč. 17, č. 3, s. 359–375. ISSN 0097-8930. Dostupné z DOI: 10.1145/964967.801167.
19. MILLER, Gavin; PEARCE, Andrew. Globular dynamics: A connected particle system for animating viscous fluids. *Computers & Graphics*. 1989, roč. 13, č. 3, s. 305–309. ISSN 0097-8493. Dostupné z DOI: https://doi.org/10.1016/0097-8493(89)90078-2.

20. MONAGHAN, J J. Smoothed particle hydrodynamics. *Reports on Progress in Physics*. 2005, roč. 68, č. 8, s. 1703–1759. Dostupné z DOI: 10.1088/0034-4885/68/8/r01.

21. LUCY, L B. Numerical approach to the testing of the fission hypothesis. *Astron. J.; (United States)*. 1977, roč. 82:12. Dostupné z DOI: 10.1086/112164.

22. MCCRACKEN, Tom. *SPH Fluid Simulation* [online]. San Francisco, 2008 [cit. 2022-04-15]. Dostupné z: https://github.com/tommccracken/sph-fluid-simulation.

23. MÜLLER, Matthias; CHARYPAR, David; GROSS, Markus. Particle-Based Fluid Simulation for Interactive Applications. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. San Diego, California: Eurographics Association, 2003, s. 154–159. SCA '03. ISBN 1581136595.

24. JLCERCOS. *SPHInterpolationColorsVerbose* [online]. 2018 [cit. 2022-04-15]. Dostupné z: https://commons.wikimedia.org/wiki/File:SPHInterpolationColorsVerbose.svg.

25. DESBRUN, Mathieu; GASCUEL, Marie-Paule. Smoothed Particles: A New Paradigm for Animating Highly Deformable Bodies. In: *Proceedings of the Eurographics Workshop on Computer Animation and Simulation '96*. Poitiers, France: Springer-Verlag, 1996, s. 61–76. ISBN 3211828850.

26. GREEN, Simon; TONGE, Richard; SAINZ, Miguel; JOHNSTON, Dane; SCHOEMEHL, David. *Fluid Simulation in Alice: Madness Returns* [online]. 2011 [cit. 2022-04-15]. Dostupné z: https://developer.nvidia.com/content/fluid-simulation-alice-madness-returns.

27. POZRIKIDIS, C. *Numerical Computation in Science and Engineering*. 2nd ed. New York: Oxford University Press, 2008. ISBN 978-0-19-537611-1.

28. ZWICKER, Matthias; PFISTER, Hanspeter; BAAR, Jeroen van; GROSS, Markus. Surface Splatting. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 2001, s. 371–378. SIGGRAPH '01. ISBN 158113374X. Dostupné z DOI: 10.1145/383259.383300.

29. CHENTANEZ, Nuttapong; MÜLLER, Matthias. Real-time Simulation of Large Bodies of Water with Small Scale Details. In: 2010, s. 197–206. Dostupné z DOI: 10.2312/SCA/SCA10/197-206.

30. LUM, Christopher. *Derivation of the 1D Wave Equation* [online]. 2018 [cit. 2022-04-15]. Dostupné z: https://www.youtube.com/watch?v=IAut5Y-Ns7g.

31. THE UNIVERSITY OF READING. *Deriving the wave equation* [online]. Reading, 2004 [cit. 2022-04-15]. Dostupné z: http://www.met.reading.ac.uk/pplato2/h-flap/math6_4.html#top.

32. LEWIN, Walter. *(2:3) The Wave Equation: Derivation (Walter Lewin, MIT)* [online]. 2008 [cit. 2022-04-15]. Dostupné z: https://www.youtube.com/watch?v=r2GIY2ZmXPY.

33. *Introduction to Ray Tracing: a Simple Method for Creating 3D Images* [online]. 2009 [cit. 2022-04-15]. Dostupné z: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/how-does-it-work.

34. *Introduction to Shading* [online]. 2009 [cit. 2022-04-15]. Dostupné z: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel.

35. VRIES, Joey de. *Cubemaps* [online]. 2014 [cit. 2022-04-15]. Dostupné z: https://learnopengl.com/Advanced-OpenGL/Cubemaps.

36. LETTIER, David. *3D Game Shaders For Beginners: Screen Space Reflection (SSR)* [online]. San Francisco, 2008 [cit. 2022-04-15]. Dostupné z: https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-reflection.html.

37. FLECK, Bernhard. Real-Time Rendering of Water in Computer Graphics. In: Wien, 2007. Dostupné také z: https://www.cg.tuwien.ac.at/courses/Seminar/WS2007/arbeit_fleck.pdf.

38. LETTIER, David. *3D Game Shaders For Beginners: Screen Space Refraction (SSR)* [online]. San Francisco, 2008 [cit. 2022-04-15]. Dostupné z: https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-refraction.html.

39. EPZCAW. *Reflection and refraction* [online]. 2011 [cit. 2022-04-15]. Dostupné z: https://commons.wikimedia.org/wiki/File:Reflection_and_refraction.svg.

40. *Lake, Mountain, Nature* [online]. 2017 [cit. 2022-04-15]. Dostupné z: https://mocah.org/353879-4k-wallpaper.html.

41. AKKAYNAK, Derya; TREIBITZ, Tali; SHLESINGER, Tom; TAMIR, Raz; LOYA, Yossi; ILUZ, David. What Is the Space of Attenuation Coefficients in Underwater Computer Vision? In: 2017. Dostupné z DOI: 10.1109/CVPR.2017.68.

42. CEREZO, Eva; SERÓN, Francisco. Rendering Natural Waters: Merging Computer Graphics with Physics and Biology. 2002. Dostupné z DOI: 10.1007/978-1-4471-0103-1_31.

43. BABOUD, Lionel; DÉCORET, Xavier. Realistic Water Volumes in Real-Time. In: *Eurographics Workshop on Natural Phenomena*. Vienne, Austria, 2006. Dostupné také z: https://hal.inria.fr/inria-00510227.

44. BROCKEN INAGLORY. *Great Barracuda, corals, sea urchin and Caustic (optics) in Kona, Hawaii 2009* [online]. 2004 [cit. 2022-04-15]. Dostupné z: https://commons.wikimedia.org/wiki/File:Great_Barracuda,_corals,_sea_urchin_and_Caustic_(optics)_in_Kona,_Hawaii_2009.jpg.

45. SHAH, Musawir; KONTTINEN, Jaakko; PATTANAIK, Sumanta. Caustics Mapping: An Image-Space Technique for Real-Time Caustics. *IEEE transactions on visualization and computer graphics*. 2007, roč. 13, s. 272–280. Dostupné z DOI: 10.1109/TVCG.2007.32.

46. WALLACE, Evan. *WebGL Water Demo* [online]. San Francisco, 2008 [cit. 2022-04-15]. Dostupné z: https://github.com/evanw/webgl-water.

47. VRIES, Joey de. *OpenGL* [online]. 2014 [cit. 2022-04-15]. Dostupné z: https://learnopengl.com/Getting-started/OpenGL.

48. KWOLEK, Bogdan; RYMUT, Boguslaw. Reconstruction of 3D Human Motion in Real-Time Using Particle Swarm Optimization with GPU-Accelerated Fitness Function. *J. Real-Time Image Process.* 2020, roč. 17, č. 4, s. 821–838. ISSN 1861-8200. Dostupné z DOI: 10.1007/s11554-018-0825-5.

49. *Transform Feedback* [online]. 2021 [cit. 2022-04-15]. Dostupné z: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Transform_Feedback&oldid=14861.

50. *What is Vulkan and how does it differ from OpenGL?* [Online]. 2015 [cit. 2022-04-15]. Dostupné z: https://gamedev.stackexchange.com/questions/96014/what-is-vulkan-and-how-does-it-differ-from-opengl.

51. ALLAIN, Alex. *OpenGL vs. DirectX: A Comparison* [online]. 2019 [cit. 2022-04-15]. Dostupné z: https://www.cprogramming.com/tutorial/openglvsdirectx.html#:~:text=DirectX%20supports%20sound%2C%20music%2C%20input,only%20on%20Windows%20and%20XBox..

52. *OpenGL vs Commercial Engines?* [Online]. 2008 [cit. 2022-04-15]. Dostupné z: https://www.reddit.com/r/gamedev/comments/7s2l7z/opengl_vs_commercial_engines/.

53. GAJSEK, Dejan. *Unity vs Unreal Engine for XR Development: Which One Is Better? [2021 Updated]* [online]. 2022 [cit. 2022-04-15]. Dostupné z: https://circuitstream.com/blog/unity-vs-unreal/.

54. *GLSL in Unity* [online]. 2022 [cit. 2022-04-15]. Dostupné z: https://docs.unity3d.com/Manual/SL-GLSLShaderPrograms.html.

55. *HLSL Cross Compiler* [online]. 2004 [cit. 2022-04-15]. Dostupné z: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Rendering/ShaderDevelopment/HLSLCrossCompiler/.

56. *10 MOSCOW PRIORITISATION* [online]. Kent, 2022 [cit. 2022-04-15]. Dostupné z: https://www.agilebusiness.org/page/ProjectFramework_10_MoSCoWPrioritisation.

57. *Heightfields: Render a large terrain (and then some fractal water) by displacing vertices from height-map data.* [Online]. 2022 [cit. 2022-04-15]. Dostupné z: http://www.cse.chalmers.se/edu/course/TDA362/tutorials/heightfield.html.

58. *Calculating normals for a height map* [online]. 2015 [cit. 2022-04-15]. Dostupné z: https://stackoverflow.com/questions/33736199/calculating-normals-for-a-height-map.

Příloha **A**

Seznam použitých zkratek

1D Jednodimenzionální

2D Dvoudimenzionální

3D Trojdimenzionální

API Application programming interface

AR Artificial reality

FDM Finite difference method

FFT Fast Fourier transform

FPS Frame per second

GLSL OpenGL Shading Language

HLSL High-Level Shader Language

IFFT Inverse fast Fourier transform

IOR Index of refraction

NSE Navier–Stokes equations

RGB Red green blue

SPH Smoothed particle hydrodynamics

SWE Shallow water equations

UML Unified modeling language

VBO Vertex buffer object

VR Virtual reality

Příloha B

Obsah přiložené SD karty

- `README.md` ..... stručný popis obsahu SD karty
- `exe` ..... adresář se spustitelnou formou implementace
  - `opengl-water.exe` ..... spustitelný binární soubor implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF