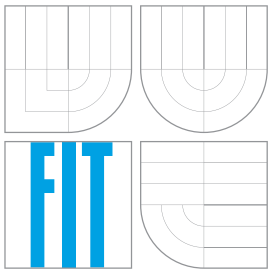
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SIMULACE A VIZUALIZACE VODNÍHO TOKU

SIMULATION AND VIZUALIZATION OF A WATER FLOW

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETR DRASTIL
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN NAVRÁTIL
SUPERVISOR

BRNO 2012

Abstrakt

Obsahem práce je návrh a implementace jednoduché demonstrační aplikace, v které bude probíhat simulace vodní hladiny nad nepravidelným terénem. Práce rozebírá základní stavební bloky, z nichž je simulace složena. Též navrhuje přístupy, které lze použít pro optimalizaci a/nebo rozšíření použitých metod.

Abstract

This work deals with design and implementation of simple demonstration aplication for simulation of a water flow on irregular terrain. The work examines essential building blocks of the simulation. It also suggests approaches that can be used to optimize and/or extend used method.

Klíčová slova

přírodní jev, animace založená na fyzice, simulace vody, výšková pole, přenos světla, Screen Space Ambient Occlusion

Keywords

natural phenomena, physically-based animation, water simulation, height-field, light transport, Screen Space Ambient Occlusion

Citace

Petr Drastil: Simulace a vizualizace vodního toku, diplomová práce, Brno, FIT VUT v Brně, 2012

Simulace a vizualizace vodního toku

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Jana Navrátila. Uvedl jsem všechny literální prameny a publikace, ze kterých jsem čerpal.

.....
Petr Drastil
14. května 2012

Poděkování

Tímto bych chtěl poděkovat Ing. Janu Navrátilovi za odbornou pomoc při tvorbě praktické a teoretické části mé práce.

Obsah

1 Úvod **3**

2 Přístupy používané pro simulaci vody **4**
2.1 Metody Eulerovského typu 5
2.2 Metody Lagrangeova typu 5
2.3 Metody využívající hydrostatickém tlaku 7

3 Simulace vody **8**
3.1 Fyzikální popis modelu 9
3.2 Optimalizace výpočtů 10
3.3 Paralelní návrh algoritmu 11
3.4 Přenos světla 13

4 Reprezentace demonstrační scény **16**
4.1 Testovací model 16
4.2 Osvětlení scény 17
4.3 Screen Space Ambient Occlusion 19

5 Implementace frameworku **22**
5.1 Implementační prostředky 22
5.2 Jádro frameworku 22
5.3 Manažeři pro správu zdrojů 24
5.4 Struktura HLSL shaderů 26

6 Implementace simulační aplikace **30**
6.1 Propojení s vyvinutým frameworkem 30
6.2 Chování vodní hladiny 31
6.3 Osvětlení scény 34
6.4 Screen Space Ambient Occlusion 36
6.5 Rozmazání obrazu 39
6.6 Vizualizace vodní hladiny 41

7 Vyhodnocení výsledků **44**
7.1 Hlavní přínos 44
7.2 Měření počtu vykreslených snímků za sekundu 46
7.3 Zhodnocení simulační metody 46

8 Závěr **48**

A Obsah DVD **53**

Kapitola 1

Úvod

Reprezentace vody byla vždy intenzivně zkoumána v oblasti počítačové grafiky vzhledem ke složitosti jejího chování a vizualizace. I když se výzkum v poslední době zaměřuje na efektivní metody, jak vyřešit výpočetně nákladné operace pro realistickou simulaci vody, tak tyto metody stále vyžadují minuty výpočetního času pro každý snímek animace. Interaktivní aplikace pro architektonický návrh krajiny, virtuální realitu nebo hry, které často potřebují trojrozměrnou simulaci vody, buď postrádají realisticky uspokojivé řešení, anebo musí pouze spoléhat na dvourozměrné zjednodušení vodního povrchu pomocí přístupů, které pracují s rovinami. Vzhledem ke složitosti chování vody neexistuje pouze jediná metoda, která by dokázala zachytit a simulovat veškeré jemné nuance, které ve vodě probíhají [14]. Z tohoto důvodu je nutné kombinovat několik metod k získání realistické animace. Přednostně by se mělo jednat o metody, které jsou založeny na fyzikálním modelu, aby jejich chování odpovídalo reálnému protějšku v přírodě a také proto, aby mohlo docházet k interakci mezi jednotlivými metodami.

Voda, která teče v přirozeném terénu vytváří několik přírodních jevů jako jsou řeky, vodopády, louže nebo jezera. Tok vody je hlavně ovlivňován gravitací a vodní hladina musí být téměř vodorovná vůči povrchu země [15]. Vzhledem k tomu, že terén bývá značně nepravidelný, dochází k nehomogennímu rozložení vody nad terénem. Z toho důvodu je vyžadována efektivní simulační metoda s dobrou prostorovou manipulací, při které zároveň nedochází ke ztrátě vizuálních detailů. Je také žádoucí, aby vizualizace byla poměrně jednoduchá a aby byla použitá metoda vhodná i pro interaktivní aplikace.

Cílem této práce je nalézt, implementovat a otestovat vhodnou metodu, která splňuje výše uvedené podmínky. Výsledná aplikace by měla obsahovat demonstrační scénu, která zahrnuje co nejvíce možných situací, které mohou při simulaci vodního toku nastat. Dále by měla simulace vody probíhat alespoň při 25 snímcích za sekundu a výsledná aplikace by měla mít dostatečně intuitivní ovládání.

Úvodní kapitola této práce obsahuje přehled a historii používaných metod pro simulaci vody. Kapitola 3 detailně popisuje zvolenou simulační metodu a kapitola 4 se zabývá úpravami simulační modelu s terénem a osvětlením demonstrační scény. V kapitole 5 je popsána implementace frameworku, nad kterým celá simulační aplikace běží a kapitola 6 se již věnuje implementaci vybrané simulační metody. Kapitola 7 se zaměřuje na zhodnocení dosažených výsledků a poslední kapitola 8 je závěrem práce.

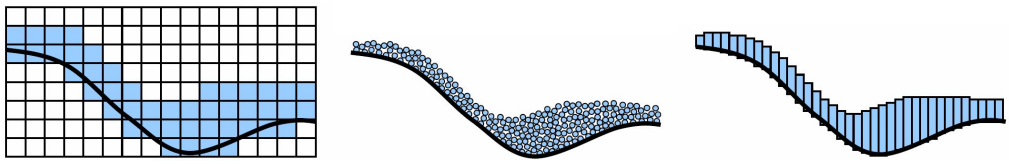
Kapitola 2

Přístupy používané pro simulaci vody

Mezi nejrozšířenější metodou pro simulaci kapalin patří bezesporu Navier-Stokesovy rovnice (dále jen NSE). Rovnice existují dvě: pro simulaci stlačitelných kapalin (plynů) a nestlačitelných kapalin (tekutin). Z hlediska této práce je zajímavá pouze část o nestlačitelných kapalinách, která je popsána pomocí rovnice 2.1 převzaté z [46]. V dané rovnici: $\mathbf{u}$ určuje rychlost kapaliny, P je tlak kapaliny, ρ je hustota kapaliny, v je kinematická viskozita a $\bigtriangledown$ je del operátor [43].

$$\frac{\delta \mathbf{u}}{\delta t} + \mathbf{u} \bullet \nabla \mathbf{u} = -\frac{\nabla P}{\rho} + v\nabla^2 \mathbf{u} \tag{2.1}$$

Výpočetní modely k vyřešení této rovnice potřebují značné množství systémových zdrojů ve smyslu uchování dat a výpočetního času [14]. Numerické řešení NSE [3] lze rozdělit na Eulerovský přístup (založen na síti buněk) a na Lagrangeův přístup (založen na částicích). První jmenovaný dělí prostor na pravidelnou síť buněk a pozoruje kapalinu, která prochází skrz tyto buňky. Druhý přístup sleduje pohyb disjunktních částic kapaliny v čase. Existuje ještě třetí přístup pro simulaci vodního toku, který však nevychází z NSE. Základní idea za tímto přístupem spočívá ve výpočtu hydrostatického tlaku v jednotlivých vodních sloupcích. Získaný hydrostatický tlak je následně využit pro přelití určitého objemu vody do jiného vodního sloupce.



(a) (b) (c)

Obrázek 2.1: Simulace vodního toku na terénu (černá křivka) s použitím různých metod: (a) Eulerovský přístup s dělením prostoru na síť buněk, které však mohou být prázdné během celé simulace; (b) Lagrangeův přístup založený na simulaci částic zvětší detail povrchu vodní hladiny, ale zároveň i náročnost výpočtu; (c) sloupce vody s variabilní výškou jsou dobrým kompromisem mezi počtem uložených buněk a detailem povrchu vodní hladiny. Obrázky převzaty z [24].

2.1 Metody Eulerovského typu

Jeden z prvních pokusů provést plně trojrozměrnou simulaci v počítačové grafice, která byla založena na NSE, byla práce Fostera a Metaxase [11]. Jejich přístup spočíval v rozdělení trojrozměrného prostoru na pravidelnou síť buněk a řešení NSE bylo provedeno pomocí diskretizace tlaku a rychlosti kapaliny vzhledem ke středu a orientaci této sítě. Pro sledování pohybu kapaliny na rozhraní se vzduchem byly použity speciální částice, které nesly barvu a/nebo výškovou informaci o kapalině.

Nejdůležitějším přínosem pro stabilitu NSE byla práce Stama [38]. Jeho metoda je bezpodmínečně stabilní použitím semi-Lagrangeovy metody pro část rovnice NSE, která počítá advekci ($\mathbf{u} \bullet \nabla$ v rovnici 2.1). Dvojrozměrná implementace na GPU byla představena v [12, 45] a třírozměrná byla představena v [21]. I když tyto simulace byly schopny běžet v reálném čase, tak neřešily problém, kdy kapalina neměla pevně definovanou hranici (například rozlévající se voda).

Tento problém byl adresován pomocí hybridní částicové a úrovňové množinové metody [37] publikováno v [9, 10, 15, 19]. Metoda využívá implicitní funkci, která se vyvíjí během simulace a má za cíl sledovat konturu okolo rozlévající se kapaliny. Tato kontura následně slouží jako rozhraní mezi kapalinou a druhým prostředím. Okolo rozhraní jsou v posledním kroku simulace použity částice uskupené v hrubě rozdělené mřížce tak, aby přesně upravily povrch kapaliny v závislosti na aktuálním stavu simulace.

Eulerův přístup má jednu závažnou chybu, kterou je značná prostorová neefektivita při simulování vodního toku v libovolném terénu. Vzhledem k tomu, že terén může být značně nepravidelný, tak síť buněk, na kterých se provádí simulace, zbytečně obsahuje buňky, které nikdy nemusí obsahovat tekutinu. Tento stav je demonstrován na obrázku 2.1a.

Lossasso a kolektiv [19] navrhovali použít adaptivní síť buněk k zmírnění výše jmenovaného problému. Výsledkem jejich práce je vyšší rozlišení simulační sítě pouze v místech, kde je potřeba vyšší míra vizuálních detailů. Toho je docíleno použitím datové struktury octree pro uchování dat a novou metodou pro diskretizaci tlaku a rychlosti kapaliny, která redukuje potřebný simulační čas. Nová rychlejší diskretizační metoda je navržena navíc tak, aby dávala srovnatelné výsledky s klasickým přístupem vzhledem k přesnosti výpočtů.

V roce 2006 Irving a kolektiv [15] navrhl hybridní metodu, která využívá dvourozměrnou mřížku složenou z výškových buněk s lineárním tlakovým profilem a třírozměrnou mřížku, která se nachází poblíž rozhraní kapaliny a jiného prostředí. Výpočet se provádí pomocí NSE na obou jmenovaných strukturách současně. Na trojrozměrné mřížce se navíc provádí výpočet pomocí částicové metody, který slouží ke sledování povrchu kapaliny. Dle tvrzení autorů dojde k zrychlení simulace, která je ovlivňována hlavně pomocí gravitačních sil. I přes toto zrychlení se, stejně jako u ostatních výše uvedených metod, pohybuje výpočet jednoho snímku v řádu minut.

Kvůli výše jmenovaným důvodům je tedy vhodné použít Eulerův přístup pouze pro aplikace, které potřebují využívat volumetrický simulační model. Avšak není již příliš vhodné používat tento přístup pro aplikace, které mají být interaktivní v reálném čase.

2.2 Metody Lagrangeova typu

Částicově založené metody reprezentují vodu podél terénu jen v takových oblastech, kde jsou jednotlivé částice potřeba. I přesto, že tyto metody mají lepší prostorovou distribuci, je často nutné použít pro simulaci menší časové kroky, aby částice nevylétávaly ven kvůli přitažlivým a odpudivým fyzikálním silám.

Jednu z prvních implementací částicového systému provedl Chiba a kolektiv [6]. Jeho tým navrhl kvazi-fyzikální metodu, ve které částice společně interagují pouze uvnitř jednoho voxelu. Tím dojde k omezení interakce částic s částicemi, které jsou již příliš vzdáleny na to, aby mohly mít na daný prostor nějaký vliv. Druhý důvod, který vedl tento tým k použití voxelů, byla správná detekce kolizí částic s překážkami v terénu. Pro rekonstrukci vodní hladiny byla použita implicitní funkce, která je závislá na aktuálním stavu částic i jejich počtu. Nevýhoda této metody spočívá v tom, že je nutné mít velké množství částic pro vizuálně celistvý povrch. Pokud je částic méně, vznikají na povrchu artefakty.

Müller a kolektiv [28] poprvé použili metodu Smoothed Particles Hydrodynamics (dále jen SPH) k simulaci tekutin pomocí interpolace fyzikálních veličin jako je viskozita a tlak. Tyto fyzikální veličiny byly následně uloženy v jednotlivých diskrétních částicích. Pro vykreslení povrchu tekutin byly použity metody point splatting a marching cubes. I přes vizuálně dobré výsledky zůstalo, dle tvrzení autorů, sledování povrchu kapaliny v interaktivních aplikacích nepoužitelné.

Kipfer a Westermann [18] vytvořili částicovou simulaci založenou na SPH, která poprvé využívala GPU akcelerace. Jejich metoda je založena na práci s třemi lineárně seřazenými seznamy, které slouží jako vyhledávací tabulky pro kolize částic. Dále je v metodě využita výšková mapa, která slouží, spolu s částicemi, pro reprezentaci vodní hladiny. Výhoda tohoto přístupu spočívá v redukci počtu částic, které jsou potřeba pro vytvoření povrchu kapaliny. Nevýhoda spočívá v nezachování objemu kapaliny. Detaily na povrchu kapaliny, jako jsou například vlny, záleží přímo na rozlišení výškové mapy, která musí být očividně malá, aby se zachoval interaktivní počet snímků animace.

Premžoe a kolektiv [32] vytvořili metodu Moving-Particle Semi-Implicit (dále jen MPS) k simulování kapalin s využitím úrovňové množinové metody, která slouží k rekonstrukci povrchu kapaliny. Úrovňová množinová metoda je numerická technika pro sledování tvarů objektů, které se vyvíjí v čase. Výhoda této metody spočívá v tom, že lze provádět numerické výpočty na křivkách i površích objektů v pevně dané kartézské souřadné soustavě bez nutnosti parametrizovat tyto objekty. MPS využívá úrovňové množinové metody pro simulaci zkoumané kapaliny v nízkém rozlišení, což poskytuje okamžitou zpětnou odezvu při interakci částic. Ve finálním kroku simulace, kdy je již známa oblast, která bude vizualizována, dojde k umělému navýšení počtu částic, pro které se dále spočítá jinou simulační metodou jejich interakce. MPS je metoda plně Lagrangeovského typu, takže částice jsou vždy prostorově rozmístěny jen na místech, kde jsou potřeba. Nevýhoda této metody spočívá v tom, že i jednoduchá polygonální scéna musí být převedena do částicové reprezentace, jinak by simulace nemohla probíhat.

Z textu výše lze říci, že metody využívající Lagrangeovův přístup obvykle potřebují značné množství částic k reprezentaci detailů na vodní hladině. Z tohoto důvodu se zvětšuje množství potřebného prostoru pro uložení dat a tím dochází i k navyšování potřebného výpočetního času pro jeden krok simulace. Jelikož částice neslouží výhradně k reprezentaci vodní hladiny, dochází v případě simulování vodního toku na nepravidelném terénu k celkem značnému navýšení počtu částic (viz obrázek 2.1b), které pro interaktivní aplikace, jako jsou například hry, nemají příliš velký význam. Rekonstrukce povrchu kapaliny pro částicové metody bývá též komplexní záležitostí, jelikož se spojitě mění topologie simulované tekutiny.

Některé práce [29, 33, 40, 41] se snaží oprostit od náročného výpočtu pro třírozměrnou simulaci vodního toku pomocí částicových metod a využívají tyto metody jen pro simulaci konkrétních jevů, které jsou často pozorovatelné hlavně v řekách a potocích jako je například chování vody při nárazu do překážky nebo její interakce s břehem řeky. Pro simulaci vodní hladiny, využívají tyto práce dvourozměrné diskretizované mřížky, na které se pro-

vádí výpočet rychlosti proudění vodního toku. Z vypočtených hodnot se následně odvodí místa, kde mají být zobrazeny vlnky a kam se mají namapovat animované bump mapy.

Z předchozího textu je zřejmé, že i když tyto metody poskytují vysoce realistickou simulaci kapalin, tak se pro interaktivní aplikace dají použít pouze a jen tehdy, pokud pracují v nízkém rozlišení. Částicové metody jsou spíš vhodné pro vizuální reprezentaci určitých efektů jako jsou vodopády nebo stříkání a tříštění vody. Z těchto důvodů jsou tyto metody často použity pouze ve filmovém průmyslu, kde záleží na kvalitě zobrazované vody, avšak již nezáleží na její interaktivitě.

2.3 Metody využívající hydrostatickém tlaku

Kass a Miller [17] poprvé navrhli provést simulaci vodního toku s předpokladem, že vodní hladina bude reprezentována pomocí výškové mapy a horizontální rychlost vodního toku bude konstantní skrz jednotlivé vodní sloupce. Jejich model používá zjednodušenou podmnožinu rovnic pro výpočet dynamiky tekutin ve dvou rozměrech. Jejich práce však nemodeluje interakci vody s externími objekty a efekty jako je tříštění nebo šplíchání vody.

Výše uvedenou metodu rozšířil a upravil O'Brien a Hodgins [30]. Princip jejich metody vychází z toho, že simulace probíhá nad vertikálními sloupci, které obsahují určitý objem vody. Tyto sloupce jsou následně uspořádány do pravoúhlé sítě a každý ze sloupců je spojen se svými sousedy pomocí virtuálního potrubí, které přelévá určitý objem vody do okolních sloupců v závislosti na fyzikálních zákonech o hydrostatickém tlaku. Díky těmto fyzikálním zákonům lze jejich model snadno rozšířit o interakci s vnějšími silami, které se na modelu projeví jako tlak působící na vodní hladinu. Vystříknutí vody lze detekovat pokud rychlost, kterou se přilévá voda do určitého vodního sloupce přesáhne stanovenou mez.

Mould a Yang [27] předcházející model dále rozšířili o možnost provádět simulaci na libovolné výškové mapě. Jejich další úpravou bylo snížení vertikální izotropie vodního sloupce jeho rozdělením na více buněk (viz obrázek 3.1), což má za následek přelévání většího objemu kapaliny mezi vodními sloupci. Do jejich modelu byl navíc přidán i nový částicový systém, který umožňoval simulovat stoupající bublinky uvnitř vody. Později tento model využil Holmberg a Wünsche [13] pro simulaci přirozeného pohybu řek a vodopádů.

Jednu z nejnovější prací, která je založena na simulaci pomocí hydrostatického tlaku vytvořil Maes a kolektiv [24], jejichž přínosem je hlavně optimalizace výše uvedených modelů. Jejich implementace vyniká v nízkých paměťových nárocích, čehož je dosaženo zrušením redundantního virtuálního potrubí, které spojuje sousední vodní sloupce. Navíc je jejich metoda přizpůsobena pro paralelní zpracování na GPU, čímž došlo ke snížení přenosu dat po sběrnici a tím pádem zrychlení celé simulace. Do simulačního modelu je také přidán nový systém pro přesný výpočet refrakce, přenosu světla a útlumu světla v závislosti na výšce vodního sloupce.

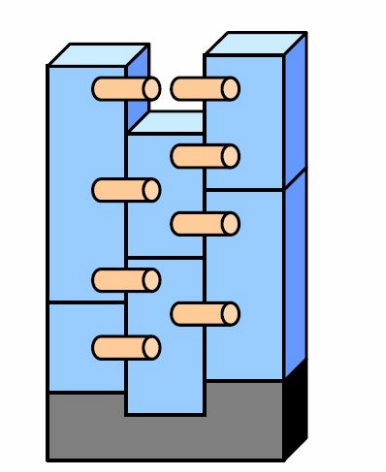
Z výše uvedeného textu je patrné, že simulace založena na hydrostatickém tlaku bude nejvhodnější pro interaktivní aplikace, jelikož má stejné výhody jako Lagrangeovy metody. Vzhledem k tomu, že každý vodní sloupec leží přímo na terénu, je výpočet simulačního kroku prováděn pouze v těch místech, kde je třeba (viz obrázek 2.1c). Výška vodních sloupců může být plně variabilní a vzorkování terénu je přímo spjato s diskretizací pravoúhlé sítě buněk, která se ukládá do výškové mapy. Z toho důvodu může být vodní povrch reprezentován jako výšková mapa nad terénem. Mezi další výhody patří nízká výpočetní náročnost hydrostatických rovnic a implicitní tvorba vln a jejich propagace na vodním povrchu. Mezi nevýhody této metody patří nemožnost simulovat cákání, pěnění nebo bublání vody. Pro tyto efekty je nutné použít jiný systém, který lze do této metody integrovat.

Kapitola 3

Simulace vody

Tato kapitola se zabývá použitým simulačním modelem, který byl převzat z práce Maese a kolektivu [24]. První podkapitola popisuje použité fyzikální rovnice, na kterých je simulace založena. Podkapitola 3.2 se zabývá způsoby jakými lze provést určité paměťové optimalizace pro simulační výpočty. Podkapitola 3.3 se zabývá uzpůsobením fyzikálních výpočtů pro GPU a poslední podkapitola 3.4 se zaměřuje na přenos světla ve vodě.

Simulační model vychází z následujících podmínek. Celý generický terén je převeden na výškovou mapu a nad touto výškovou mapou je uživatelem inicializován počáteční stav vodní hladiny. Každý texel výškové mapy reprezentuje jeden potencionální diskrétní vodní sloupec. Vodní sloupce, které slouží pro přítok a odtok vody ze systému, se modelují tak, že mají zachovanou svojí inicializační výšku během celé doby simulace. Vodní sloupec je následně rozdělen na více buněk, aby simulace probíhala rychleji. V místě překryvu buněk vodních sloupců je vytvořeno virtuální potrubí (viz obrázek 3.1). Žádná část potrubí není nikdy položena vertikálně mezi jednotlivými buňkami v jednom vodním sloupci. Průtok vody potrubím je po startu simulace nulový. Výška vodního sloupce se mění v závislosti na průtoku vody skrz potrubí. Pro šíření vody do oblastí, kde se ještě žádná voda nenachází, se používá simulační model pro přepad vody skrz vodní jez (viz obrázek 3.2a). Tento model je též použit pro přepad nestabilních vln na vodní hladině.



Obrázek 3.1: Vodní sloupec se dvěma buňkami. Virtuální potrubí je vytvořeno pro veškeré buňky, které se navzájem překrývají a také pro buňku, která sousedí se vzduchem. Obrázek převzat z [24].

3.1 Fyzikální popis modelu

Průtok vody ve virtuálním potrubí je určen pomocí fyzikální rovnice o hydrostatice. Tlak na jedné straně potrubí je definován jako

$$p = h\rho g + p_0 + p_e, \tag{3.1}$$

kde h je výška vodního sloupce nad daným bodem; ρ je hustota kapaliny; g je gravitační zrychlení; p_0 je atmosferický tlak a p_e je tlak zapříčiněný vnějšími silami.

Rychlost průtoku vody potrubím vychází z rozdílu tlaku mezi dvěma sousedními body, což lze popsat rovnicí

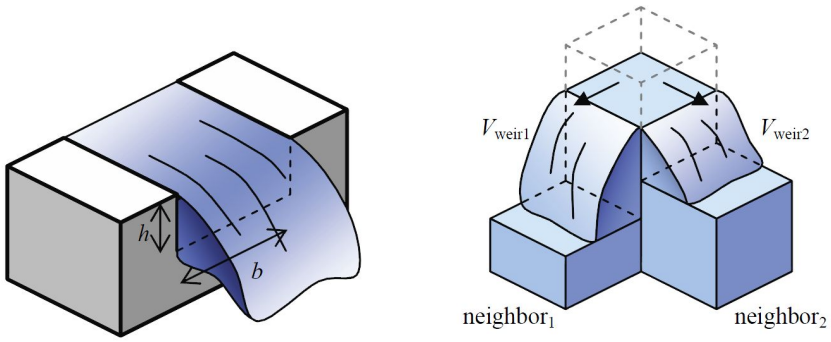
$$\eta = f\eta_0 + \Delta t \frac{(p_{head} - p_{tail})}{\rho l}, \tag{3.2}$$

kde f je nefyzikální koeficient tření, který byl použit v [27] k získání postupné ztráty energie; η_0 je rychlost proudění vody v předcházejícím kroku simulace; Δt je simulační časový krok; a l označuje délku potrubí. Při znalosti rychlosti průtoku vody potrubím je objem, který potrubím projde za jednotku času definován jako

$$V_{pipe} = \Delta t \eta c, \tag{3.3}$$

kde c určuje průřez potrubí. Průřez lze vyjádřit například plochou, kterou se sousední buňky ve vodním sloupci překrývají. Přenesený objem vody skrz potrubí lze převést na změnu výšky jednotlivých buněk vodního sloupce. Vzhledem k tomu, že hmota musí být zachována, je nutné provádět změnu měřítka odtékajícího objemu vody v případě, kdy suma všech objemů přenesených potrubím přesáhne objem vody, který buňka uchovává. V momentě, kdy se výška buňky dostane na určitou nejnižší možnou úroveň, je buňka považována za vyschlou a voda z ní dále neodtéká do okolních buněk.

Vzhledem k tomu, že rychlost toku potrubím závisí na předchozím časovém kroku, je potřeba tyto rychlosti uložit do paměti pro každou virtuální trubku. Jak se výška jednotlivých vodních sloupců mění během simulace, je nutné, aby doházelo k dynamickému vytváření a rušení virtuálních trubek na základě překrytí buněk sousedních vodních sloupců.



(a) (b)

Obrázek 3.2: (a) Tok vody přes vodní jez (b) aplikovaný na diskrétní model. Obrázky převzaty z [24].

Pro modelování jevů jako jsou cákance a vodopády používá Holmberg a Wünsche výpočet pro objem vody, který protéká vodním jezem (viz obrázek 3.2a). V jejich práci je tento

model použit v případě, kdy vlna putující po vodní hladině přejde do nestabilní fáze. Průtok vodním jezem je dán vztahem

$$\zeta = \frac{2}{3} b h^{\frac{3}{2}} \sqrt{2g}, \tag{3.4}$$

kde b určuje šířku vodního sloupce a h je výška přepadávající vody (viz obrázek 3.2a). Přenesený objem vody je pak dán rovnicí

$$V_{weir} = \zeta \Delta t. \tag{3.5}$$

Použití fyzikálního modelu pro průtok vody vodním jezem je velmi dobrá aproximace pro simulování volně padající vody, jelikož směr přepadu vody je dán jen vůči sousedním vodním sloupcům. Přepad vody nastává jen tehdy, když padající vodě v cestě nestojí nějaká překážka (viz šipky na obrázku 3.2b).

Pro získání nové výšky vodní hladiny je potřeba určit, jaký objem přitéká ze všech sousedních vodních sloupců a odečíst objem vody, který odtéká z aktuálně zpracovávaného vodního sloupce pomocí rovnic 3.3 a 3.5. Při tomto výpočtu může dojít k situaci, kdy objem odtékající vody přesáhne celkový objem uchované vody v daném vodním sloupci. V této situaci je nutné změnit měřítko odchozího objemu pomocí rovnice

$$scalefactor = \frac{V_{column}}{\sum_{i}^{out} V_{pipe} + V_{weir}}, \tag{3.6}$$

kde V_{column} je celkový objem uchované vody ve vodním sloupci; out je celkový počet sousedních vodních sloupců, do kterých odtéká voda volným pádem nebo pomocí virtuální trubky; V_{pipe} a V_{weir} jsou patřičné objemy, které odtékají do sousedního vodního sloupce i.

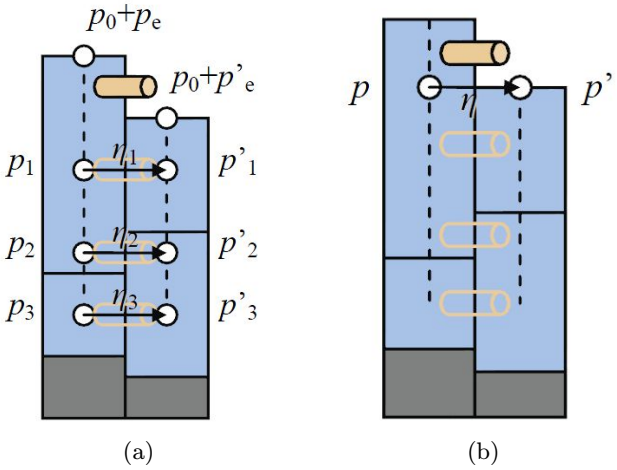
3.2 Optimalizace výpočtů

Je vhodné si uvědomit, že rozdíl tlaku mezi jakýmikoliv dvěma ponořenými body je stejný, pokud se tyto dva body nachází ve stejné výšce (viz obrázek 3.3a), kde platí vztah $(p1 - p'1) = (p2 - p'2) = (p3 - p'3)$. Výsledná rychlost průtoku vody každou trubkou, dle rovnice 3.2, bude tím pádem stejná ($\eta_1 = \eta_2 = \eta_3$). Z tohoto důvodu je možné snížit paměťové nároky tím, že se provede výpočet a uložení jen jedné hodnoty η pro jeden pár sousedních bodů; například pár s tlakovým rozdílem $(p - p')$, který je zobrazen na obrázku 3.3b.

K výpočtu přeneseného objemu vody je také nutné zjišťovat v každém simulačním kroku, zda se dvě buňky vzájemně překrývají či nepřekrývají. Tento proces neovlivňuje celkový výkon, jelikož obdobný proces probíhá i v původním algoritmu [30], kde se zjišťuje, zda má dojít k vytvoření nebo smazání určité trubky.

Výše popsaným způsobem lze redukovat maximální paměťové požadavky pro jeden pár sousedních vodních sloupců z $2 \times n_{cell}$, kde n_{cell} je počet buněk ve vodním sloupci, pouze na uložení dvou hodnot (viz obrázek 3.3b), jež obsahují rychlost proudění potrubím mezi sousedními vodními sloupci a objem volně padající vody na níže položený vodní sloupec. Tato optimalizace tedy zbavuje simulaci značného množství redundantních výpočtů, které nezávisí na počtu použitých buněk ve vodních sloupci.

Průtok volně padající vody vodním jezem, rovnice 3.4, nezáleží na objemu prošlé vody v předcházejícím časovém kroku. Této vlastnosti lze využít k redukci maximálního počtu



Obrázek 3.3: (a) Průtok vody mezi sousedními sloupci nastává při rozdílném hydrostatickém tlaku. (b) Paměťová redukce pro uložení průtoku vody. Obrázky převzaty z [24].

uložených hodnot na jednu jedinou hodnotu. Navíc je tímto způsobem přidán do simulace model pro volný pád vody.

Na dvourozměrné pravoúhlé síti má každý vodní sloupec 8 sousedů. Aby se zabránilo zbytečnému ukládání duplicitních virtuálních trubek, je možné uložit pouze 4 unikátní trubky, které spojují daný sloupec s okolními (viz obrázek 3.4b). Zbývající potrubí lze získat od patřičného souseda. Uložením jen jedné unikátní rychlosti pro proudění potrubím nedojde jen k zmenšení nároků na paměť, ale zároveň i k redukci přístupů do paměti. Například pro výškovou mapu s rozlišením 256×256, při uložení rychlosti toku jako 32-bitové hodnoty s plovoucí řádovou čárkou, je potřebná paměť pro uložení hodnot dána vztahem $256 \times 256 \times (2 \times n_{cell}) \times (4 \text{ floaty } \times 4 \text{ byty/float}) = 4\text{MB}$ pro $n_{cell} = 2$. Tato hodnota se sníží na 1MB ($\frac{1}{2 \times n_{cell}} = 25\%$) při použití této optimalizace. I když se zdá, že pro současný hardware je tato hodnota malá, tak počet přístupů pro změnu výšky vodních sloupců v jednom časovém kroku je $256 \times 256 \times (2 \times n_{cell}) = 262144$ pro $n_{cell} = 2$, zatímco s použitou optimalizací je to jen 65536 přístupů do paměti (25%). Tyto hodnoty jsou o hodně významnější při použití výškových map s vyšším rozlišením.

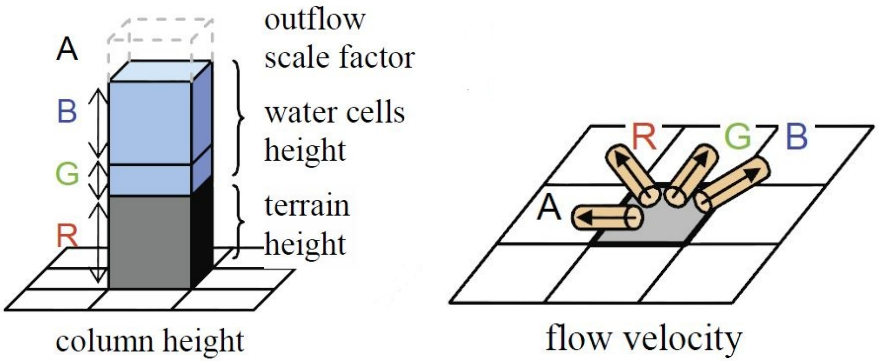
Poslední optimalizací, kterou lze ještě provést, je přidání dalšího vykreslovacího průchodu pro výpočet změny měřítka (rovnice 3.6). Tento výpočet lze provést i při aktualizaci výšky vodního sloupce, avšak separátní vykreslovací průchod zajistí přibližně 5×vyšší rychlost simulace. Toto zrychlení je způsobeno odstraněním stejných výpočtů pro změnu měřítka pro stejné sousední vodní sloupce.

3.3 Paralelní návrh algoritmu

V nedávné době začal být grafický hardware levný, programovatelný a je využíván jako výpočetní jednotka pro obecné paralelní výpočty, které se provádí pomocí vertex a fragment programů na stream procesorech [16]. Aktuální generace programovatelného grafického hardware je schopna získávat data pomocí paměťových operací, přičemž jakýkoliv stream procesor může přistoupit ke všem sdíleným vstupním datům. Výstupní hodnotu

vypočítanou stream procesorem lze však zapsat pouze do jedné jediné paměťové buňky o určité velikosti. Z tohoto důvodu je nutné přizpůsobit uložení simulačních dat pro výpočty na GPU.

Pro uložení simulačních dat jsou použity dvourozměrné textury; jedna pro rychlost průtoku vody potrubím a jedna pro výšky vodních sloupců (viz obrázek 3.4). Pro aktualizaci uložených hodnot ve fragment shaderech je nutné použít mapování 1:1 pro pixel na texel, jinak dochází k postupné degradaci hodnot kvůli automatické interpolaci při vykreslení textury.



(a) (b)

Obrázek 3.4: Uložení simulačních dat v texturách: (a) terén zabalený společně s dvěmi vodními buňkami pro jeden vodní sloupec. Celý vodní sloupec je uložen v jednom texelu; (b) rychlost průtoku skrz potrubí mezi sousedními vodními sloupci a standardní směr proudění vody potrubím. Obrázky převzaty z [24].

Vstupní terén je dán pomocí výškové mapy, která je uložena jako černobílý obrázek v textuře s plovoucí řádovou čárkou. Výška vodních sloupců musí být ve stejném měřítku jako výšková mapa terénu, aby nedocházelo k zaplavení celého terénu nebo zmizení celé simulované vodní hladiny. Pro redukci počtu přístupů k texturovací paměti jsou terén i vodní sloupce uloženy v jedné RGBA textuře, přičemž červená komponenta nese výšku terénu a zbylé komponenty uchovávají výšku jedné buňky vodního sloupce (viz obrázek 3.4a). Alfa kanál je použit pro uchování změny měřítka pro odtékající vodu pomocí rovnice 3.6.

Jeden texel může nést informace až o dvou buňkách ($n_{cell} = 2$) pro jeden vodní sloupec (viz obrázek 3.4a), což vede k minimálnímu počtu přístupů do texturovací paměti během simulační a vykreslovací etapy. Více buněk ($n_{cell} > 2$) lze uložit v dodatečných texturách, což však zřejmě zpomalí simulaci kvůli narůstajícímu počtu přístupů do paměti a výpočtům, které určují překrytí mezi jednotlivými buňkami (viz výpočet v závěru podkapitoly 3.2).

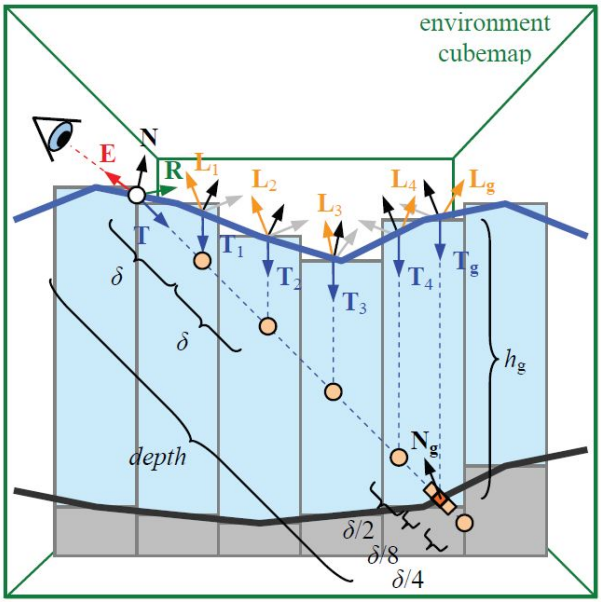
Dále je potřeba ukládat hodnotu pro rychlost průtoku trubkou mezi jedním párem sousedních vodních sloupců, což je mnohem lepší způsob než alokování a údržba všech virtuálních trubek mezi překrývajícími se buňkami. Na obrázku 3.4b je zobrazeno uspořádání těchto rychlostí pro jednotlivé sousední vodní sloupce. Dále je zde zobrazen standardní směr proudění vody potrubím. Pokud je uložená hodnota záporná, voda proudí trubkou v opačném směru.

3.4 Přenos světla

Pro realistickou vizualizaci povrchu vodní hladiny je potřeba pro každý vykreslený fragment spočítat přenos světla. Toho lze dosáhnout přístupem k textuře, která obsahuje hodnoty s výškami vodních sloupců. Přístup k hodnotám musí probíhat pomocí bilineární interpolace, aby nedocházelo k zobrazování vizuálních artefaktů v podobě nepěkných "zubů", které jsou dány diskretizací výškové mapy.

Základní fyzikální vztahy

Pro výpočet přenosu světla je nutné určit odraz světla **R** a refrakci světla **T** na povrchu vodní hladiny v závislosti na pozici pozorovatele **E** a normály vodní hladiny **N** (viz obrázek 3.5). Pro refrakci světla ze vzduchu do vody je použit Snellův zákon s indexem lomu 1 pro vzduch a indexem lomu 1,33 pro vodu.



Obrázek 3.5: Diagram znázorňující potřebné vektory pro přenos světla ve vodě. Obrázek převzat z [24].

Fresnelův faktor určuje poměr mezi odraženým a lomeným nepolarizovaným světlem na dielektrickém materiálu. Pro jeho výpočet je použita Schlickova aproximace [35], která je dána vztahem:

$$R = f_\lambda + (1 - f_\lambda)(1.0 - N \cdot E)^5, \tag{3.7}$$

kde f_λ je spektrální distribuce Fresnelova faktoru při dopadu incidenčního paprsku. Spektrální distribuce f_λ je závislá na vlnové délce světla λ, ale pro zjednodušení výpočtu je použita jen jedna jediná hodnota pro celé spektrum. Tato hodnota je $\approx 0{,}02$ pro vodu [36]. Přenosový koeficient T je doplněk k odrazivosti, který vyplývá ze zákonu o zachování energie. Tento koeficient je dán vztahem:

$$T = 1 - R. \tag{3.8}$$

Celková intenzita světla vyzářená směrem k pozorovateli **E** je pak dána vztahem

$$I_E = RI_R + TI_T, \tag{3.9}$$

kde I_R je barevná složka světla přicházejícího z odraženého paprsku $\mathbf{R}$, který je získán vzorkováním cubemapy za předpokladu, že okolní prostředí je příliš daleko od povrchu vodní hladiny. I_T je barevná složka světla získaná z refrakčního paprsku.

Výpočet refrakční barevné složky

Pro výpočet refrakční barevné složky I_T (viz rovnice 3.9) je potřeba určit průsečík refrakčního paprsku s terénem. Tento údaj lze zjistit pomocí lineárního vyhledávání s pevnou délkou přírůstku δ, jež má počátek na vodní hladině a v každém kroku se podle refrakčního vektoru $\mathbf{T}$ noří hlouběji do vodní hladiny, dokud se nedostane pod úroveň terénu. Jakmile se lineární vyhledávání dostane pod terén, přejde se k binárnímu vyhledávání [31], které pokračuje z poslední známé pozice nad terénem s velikostí přírůstku $\pm\delta/2^{level}$, kde $level$ určuje počet zanoření pod terén. Tímto způsobem lze velmi přesně určit průsečík refrakčního paprsku s terénem. Celý proces je pro snadnější pochopení zobrazen na obrázku 3.5.

Útlum světla v průhledných materiálech nesnižuje pouze intenzitu barvy, ale také prohlubuje sytost barvy a mění její odstín [39]. Vnitřní propustnost světla v roztocích kapalin je dán pomocí Beerova zákona:

$$T_{internal}(\lambda, l) = 10^{-a(\lambda)cl}, \tag{3.10}$$

kde λ je vlnová délka světla; $a(\lambda)$ je absorpční spektrum materiálu; c je koncentrace roztoku a l je délka světelné cesty. Absorpční spektrum čisté vody bylo pro barevný model RBG určeno koeficientem (0,648; 0,053; 0,036) m^{-1} [42]. Tato aproximace může způsobit značné nepřesnosti kvůli podvzorkování světla [39]. I přes tuto nepřesnost by měl být výsledný vzhled vody dostatečný.

Je předpokládáno, že nejvyšší příspěvek světla pod vodou přichází z vertikálního směru nad hladinou. Osvětlení dopadající na terén pod vodou je pak vypočítáno z intenzity světla, které přichází z vertikálního refrakčního paprsku $\mathbf{T_g}$ (viz obrázek 3.5). Fresnelův faktor je též použit pro modulaci intenzity světla přeneseného do vody. Difusní osvětlení na terénu je dáno skalárním součinem $\mathbf{N_g} \cdot (-\mathbf{T_g})$ a intenzita světla je utlumena vzhledem k výšce h.

Je též předpokládáno, že se ve vodě nacházejí nečistoty, které dále tlumí intenzitu světla, ale také přispívají k rozptylu světla pod hladinou. Pro útlum je použita jednoduchá exponenciální útlumová funkce dána rovnicí:

$$A(l) = e^{-kl}, \tag{3.11}$$

kde k je útlumový koeficient a l je délka cesty uraženého světla. Experiment s různými hodnotami k lze nalézt v příloze práce [24].

Pro rozptyl světla je použita jednoduchá metoda [7, 26], která je počítána pro každý lineární vyhledávací interval δ (viz hledání průsečíku s terénem výše). Tento přístup má nízkou výpočetní cenu, jelikož využívá dostupné informace o aktuální hloubce pod vodní hladinou, jež je získána při lineárním vyhledávání dna terénu. Pokud je interval δ příliš velký, může dojít při rozptylu světla k vizuálním artefaktům, které budou působit rušivě ve vzhledu vody.

Vertikálně přicházející světlo je modulováno pomocí $\mathbf{T} \cdot \mathbf{T_i}$. Intenzita světla získaná z refrakčního paprsku ve směru $\mathbf{T}$ je tedy dána osvětlením terénu a rozptylem světla ve vodě kvůli nečistotám pomocí následující rovnice:

$$\begin{aligned} I_T = {} & I_{\mathbf{L_g}} T_{\mathbf{T_g}}(\mathbf{N_g} \cdot (-\mathbf{T_g})) T_{internal}(\lambda, h_g + depth) A(h_g + depth) \\ & + \sum_{i=1}^{n} I_{\mathbf{L}_i} T_{\mathbf{L}_i}(\mathbf{T} \cdot \mathbf{T}_i) T_{internal}(\lambda, h_i + i \times \delta) A(h_i + i \times \delta), \end{aligned} \tag{3.12}$$

kde $I_{\mathbf{T}}$, $I_{\mathbf{L_g}}$, $I_{\mathbf{L_i}}$ jsou v tomto pořadí: intenzita světla na povrchu vodní hladiny; intenzita světla přicházejícího ze směru L_g a intenzita světla přicházejícího ze směru L_i. Hodnoty $T_{\mathbf{T_g}}$ a $T_{\mathbf{L_i}}$ jsou Fresnelovy přenosové faktory (rovnice 3.8) pro vektory $\mathbf{T_g}$ a $\mathbf{L_i}$ (viz obrázek 3.5). Hodnota n určuje počet kroků lineárního vyhledávání uvnitř vody, které se děje nad terénem; h_i je výška z aktuální pozice uvnitř kroku k vodní hladině a *depth* určuje vzdálenost od vodní hladiny k terénu.

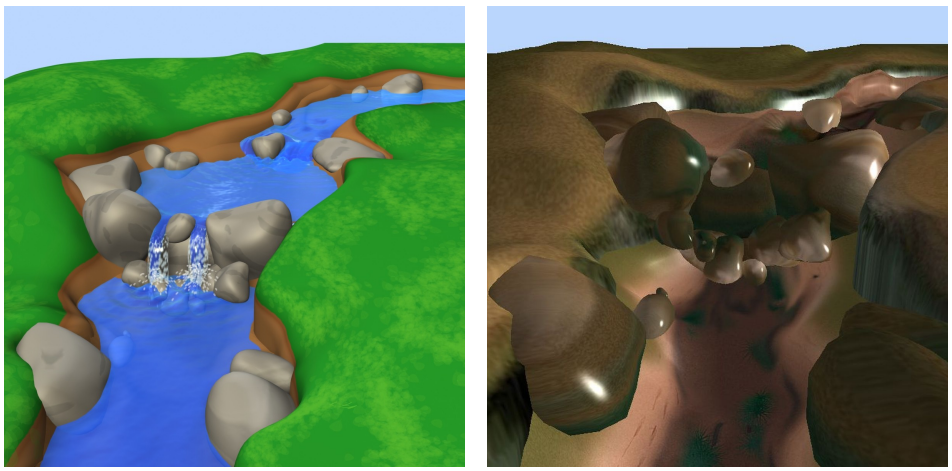
Kapitola 4

Reprezentace demonstrační scény

Úvodní část této kapitoly se zabývá volbou a úpravami testovacího modelu pro potřeby zvolené simulační metody v kapitole 3. V podkapitole 4.2 je popsán použitý osvětlovací model, který slouží k vizualizaci celé scény. Tento model je následně rozšířen o globální iluminaci pomocí metody Screen Space Ambient Occlusion v podkapitole 4.3.

4.1 Testovací model

Testovací model by měl postihnout co nejvíce případů, ke kterým může během simulace vody dojít. Měl by obsahovat alespoň koryto řeky, jezírko a vodopád. Z toho důvodu byl zakoupen animovaný model z [2], který je zobrazen na obrázku 4.1a.



(a) (b)

Obrázek 4.1: Použitý testovací model: (a) originální model získaný z [2]; (b) upravený model pro potřeby simulace.

Pro účely této práce musel být model mírně upraven. V prvním kroku byla z modelu odstraněna animovaná vodní hladina. Tím se však objevil problém se dnem koryta řeky, které nemělo v některých oblastech definován povrch. Tyto díry musely být zaplátovány, aby mohla simulace probíhat korektně. Druhý problém se projevil až při testování simulace. Model příliš neodpovídal reálné krajině, a tak se v některých oblastech voda vylévala

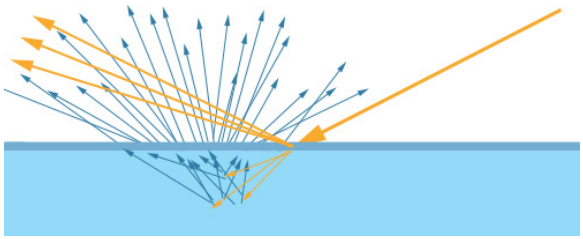
na místa, kde by intuitivně neměla být. Tento nedostatek byl opraven prohloubením koryta řeky. Provedené změny v geometrii měly za následek rozhození texturovacích koordinátů. Z toho důvodu byl model v závěru práce kompletně přetexturován. Výsledek práce je vidět na obrázku 4.1b.

Veškeré úpravy probíhaly v trial verzi programu 3D Studio Max, který byl použit kvůli proprietárnímu formátu .max originálního modelu. Upravený model byl vyexportován do formátu .obj, který je plně dokumentovaný a z toho důvodu i vhodnější pro použití ve vlastní 3D aplikaci.

4.2 Osvětlení scény

Pro osvětlení scény je použit Phongův model pro odraz světla a Phongův stínovací model. První věc, kterou je tedy nutné definovat, je rozdíl mezi těmito dvěma modely.

Model pro odraz světla (viz obrázek 4.2) je matematický model, který popisuje, jak se na daném povrchu chová dopadající paprsek světla (incidenční paprsek). Výstupem tohoto modelu je pak hodnota, která určuje jaký je celkový odraz světla v místě dopadu incidenčního paprsku.



Obrázek 4.2: Model pro odraz světla. Modré paprsky určují difuzní odraz, žluté zrcadlový odraz. Obrázek převzat z [4].

Stínovací model se zabývá tím, jak určit správnou barvu materiálu v závislosti na použitém světle. Tato barva může být určena pomocí dvou přístupů: pro jednotlivé vrcholy (per vertex) anebo pro jednotlivé fragmenty (per pixel). Per vertex přístup určí barvu na každém vrcholu modelu. Výsledná barva povrchu je pak dána lineární interpolací mezi jednotlivými barvami na sousedních vrcholech. Per pixel přístup určuje výslednou barvu pro každý pixel obrazu, což vede k získání mnohem většího detailu.

Rozdíl mezi těmito dvěma přístupy spočívá v počtu potřebných hodnot pro výpočet intenzity světla. Per vertex přístup je očividně mnohem rychlejší, jelikož je výpočet proveden pouze 3× pro jedno vykreslované primitivum, zatímco per pixel přístup je proveden tolikrát, kolik pixelů zabírá povrch daného primitiva.

Primární osvětlení scény má na starosti světelný zdroj. Existuje mnoho typů světelných zdrojů s různým chováním, avšak všechny typy sdílí jednu společnou vlastnost, kterou je jejich intenzita. Intenzitu světelného zdroje si lze představit jako jas. Hodnota jasu se pohybuje v intervalu $\langle 0; 1\rangle$ a je individuální pro každý kanál barevného modelu RGB. Intenzita světla tedy určuje barvu vyzařovaných paprsků světelného zdroje. Ve většině knih o grafice jsou termíny intenzita světla a barva světla volně zaměňovány, v dalším textu bude používán výhradně termín intenzita světla.

Nejběžněji používané typy světel pro osvětlení scény lze rozdělit na:

- **Směrová světla** - jsou definována směrem a intenzitou světla. Směr světla je stejný pro každý bod, na který dopadají světelné paprsky. Tento případ nastává v situaci,

kdy je světelný zdroj vzdálen nekonečně daleko (např. Slunce). Tento typ světla využívá tato práce k osvětlení demonstrační scény.

- **Bodová světla** - jdou definována pomocí intenzity a pozice světla. Intenzita tohoto typu světla klesá se vzdáleností od zdroje. Směr světla a útlum světla musí být zjištěn ještě před provedením osvětlovacích výpočtů.

- **Reflektorová světla** - jsou speciálním typem bodových světel. Světlo je vyzařováno pouze v kuželi, který určuje směr vyzařování.

V závislosti na povrchu materiálu se může incidenční paprsek odrazit dvěma různými způsoby:

- **Difuzní odraz** - nastává v případě, kdy světlo vstoupí do struktury materiálu, roztříští se, následně interaguje s částicemi uvnitř materiálu a nakonec je vyzářeno z materiálu ven (viz modré paprsky na obrázku 4.2). Vyzářené světelné paprsky mají jiný směr než vstupní incidenční paprsek kvůli náhodné interakci s částicemi materiálu. V počítačové grafice jsou veškeré povrchy považovány za perfektně difuzní, což znamená, že difuzní světlo je vyzářeno z materiálu ve všech směrech rovnoměrně. Tato vlastnost je nezbytná k aplikaci Lambertova zákona o odrazu [23].

- **Zrcadlový odraz** - nastává v případě, kdy světelný paprsek narazí na povrch materiálu. Část paprsku je následně pohlcena materiálem a zbytek je zrcadlově odražen (viz žluté paprsky na obrázku 4.2). Odražené světlo se zpět do prostoru šíří v oblouku, jehož šířka je dána hladkostí povrchu, na který světlo dopadá. Čím hladší je povrch, tím užší je odražený svazek světla. Je důležité zmínit, že zrcadlový odraz je závislý na pozici pozorovatele. Pozorovatel odraz vidí pouze a jen tehdy, pokud stojí v odraženém oblouku světla. Pokud je úhel mezi vektorem odrazu a vektorem pohledu pozorovatele velmi malý, intenzita světla bude velmi vysoká a pozorovatel uvidí malou kulatou oblast, která bude mít téměř 100%-ní intenzitu dopadajícího světla.

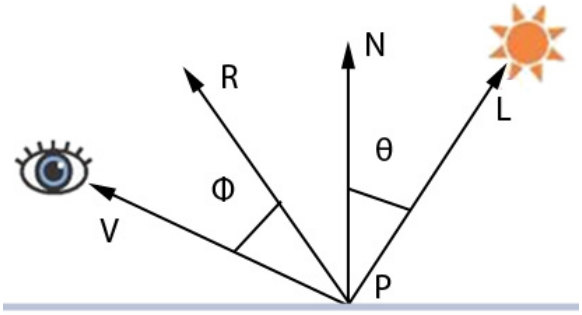
Je též důležité zmínit, že osvětlení, které je použito pro nasvícení scény, je pouze lokální v tom významu, že každý objekt na scéně je osvětlen zvlášť a neberou se v úvahu druhotné odrazy světla od ostatních objektů, překážek, atd. Z tohoto důvodu je nutné brát v potaz ještě jeden faktor, který ovlivňuje výslednou intenzitu dopadajícího světla na povrch materiálu.

- **Okolní odraz** - V reálném životě jsou objekty nasvíceny i odrazem světla od ostatních objektů. Vzhledem k tomu, že je výpočet všech odrazů od ostatních objektů velmi náročný, spočívá nejrychlejší metoda v přidání patřičného množství okolního světla ke všem objektům na scéně.

Povrch každého objektu je zhotoven z určitého materiálu a každý materiál odráží světlo jinak. Z toho důvodu je potřeba jednotlivé materiály parametrizovat, přičemž jednotlivé parametry ovlivňují jakým způsobem se od povrchu materiálu odráží světlo. Každý povrch má tři základní parametry pro odraz: K_a (pro okolní odraz), K_s (pro zrcadlový odraz), K_d (pro difuzní odraz) a čtvrtý parametr F, který určuje jak velká má být oblast pro odlesky ze zrcadlového odrazu.

V předchozím textu bylo zmíněno, že se v počítačové grafice považují veškeré povrchy jako perfektně difuzní kvůli aplikaci Lambertova zákona o odrazu [23], jež říká, že intenzita difuzního odrazu je proporcionální vůči cosinu úhlu θ mezi příchozím incidenčním paprskem

L a normálovým vektorem povrchu N (viz obrázek 4.3). Vzhledem k tomu, že incidenční paprsek L a normálový vektor N jsou oba jednotkové vektory, lze cosinus úhlu mezi těmito dvěma vektory počítat pomocí skalárního součinu. Je absolutně nezbytné poznamenat, že skalární součin dvou vektorů může nabývat i negativních hodnot, avšak výstupní hodnoty pro intenzitu světla se pohybují v intervalu $\langle 0; 1 \rangle$. Z tohoto důvodu je nutné výsledek vždy oříznout, tak aby spadal do výše uvedeného intervalu.



Obrázek 4.3: Vektory nutné pro výpočet osvětlení. Obrázek převzat z [4].

Výpočet intenzity difuzního světla v daném bodě P je dán vztahem

$$I_d = K_d \times (N \cdot L) \times light\_intensity. \tag{4.1}$$

Zrcadlový odraz je závislý na pohledovém vektoru V a odrazovém vektoru R. Čím blíže jsou tyto dva vektory u sebe, tím silnější je výsledná intenzita odrazu. Cosinus úhlu Φ, který je svírán těmito dvěma vektory lze opět vyjádřit jako skalární součin. Pro zrcadlový odraz je nutné vzít v úvahu i hladkost povrchu. Čím hladší je povrch, tím užší bude odražený svazek světelných paprsků. Toto chování lze simulovat přidáním exponentu k intenzitě světla. Čím větší bude hodnota exponentu F, tím menší bude výsledná intenzita zrcadlově odraženého světla a tím je povrch hladší. Výsledná rovnice pro zrcadlový obraz je tedy definována jako

$$I_s = K_s \times (R \cdot V)^F \times light\_intensity. \tag{4.2}$$

Intenzita okolního světla je dána jako součin intenzity světelného zdroje a koeficientu pro okolní odraz. Výsledná rovnice je dána vztahem

$$I_a = K_a \times light\_intensity. \tag{4.3}$$

Finální intenzita světla v bodě P je pak suma všech předchozích intenzit, což lze popsat vztahem

$$I = I_a + I_s + I_d. \tag{4.4}$$

4.3 Screen Space Ambient Occlusion

Screen Space Ambient Occlusion (SSAO) je osvětlovací model, který aproximuje množství světla, které dopadá na difuzní povrch v závislosti na přímo viditelných překážkách. Tento model je používán pro přidání globálního osvětlení do vykreslované scény. Rozdíl mezi vykreslenou scénou bez SSAO a se SSAO je viditelný na obrázku 4.4. První použitá implementace [25] se vyskytla ve hře Crysis a veškeré další implementace na tuto práci navazují.

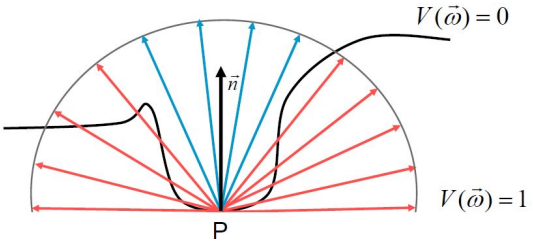
(a) (b)

Obrázek 4.4: Screen space ambient occlusion: (a) scéna s vypnutým SSAO; (b) scéna se zapnutým SSAO. Obrázky převzaty z [8].

Metoda SSAO vychází z následující rovnice pro Ambient Occlusion:

$$A_P(\vec{n}) = 1 - \frac{1}{\pi} \int_{\Omega} V_P(\vec{\omega})(\vec{n}.\vec{\omega})\, \mathrm{d}\omega, \tag{4.5}$$

kde A_P je výsledný koeficient pro útlum intenzity dopadajícího světla; $\vec{n}$ je normála bodu P; Ω určuje jednotkovou polokouli okolo tohoto bodu; $\vec{\omega}$ je vyšetřovaný směrový vektor a V_P je funkce, která rozhoduje, zda vektor $\vec{\omega}$ protíná překážku nacházející se v polokouli Ω. Celá situace je demonstrována na obrázku 4.5.



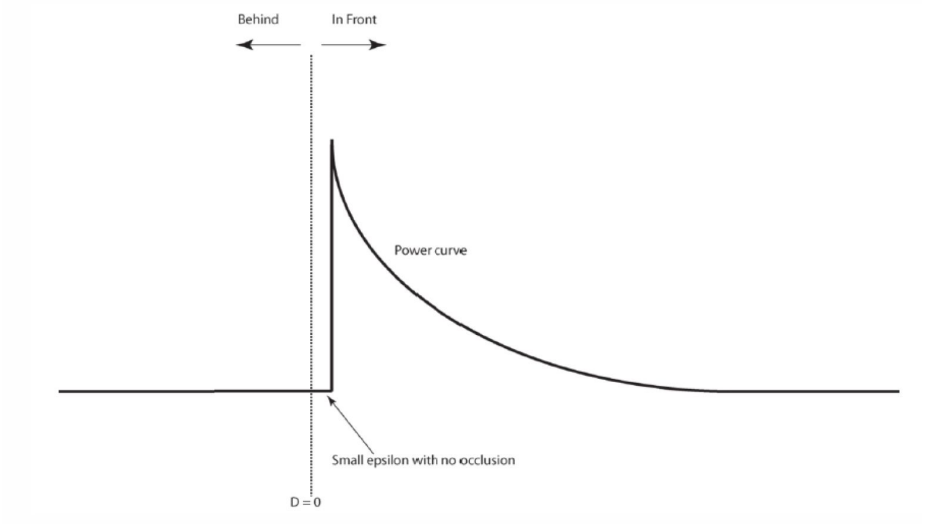
Obrázek 4.5: Ambient Occlusion: červené vektory protínají terén (černá čára) umístěný v jednotkové polokouli, modré terén neprotínají. Obrázek převzat z [34].

Popsanou rovnici lze snadno implementovat pomocí ray-tracingu, což však vede k offline renderování. Z toho důvode se provádí aproximace rovnice 4.5 ve screen space s využitím dodatečných textur pro uchování normál a hloubky scény. Výhoda tohoto přístupu spočívá v nulové závislosti na počtu vykreslovaných primitiv ve scéně. Zároveň je, jako součást normálního vykreslovacího procesu, k dispozici z-buffer s hloubkovou informací o scéně, čímž se lze zbavit dalších výpočtů.

Princip fungování metody SSAO spočívá ve vykreslení scény a uložení hodnot ze z-bufferu do hloubkové textury a normál do normálové textury. Následně je provedeno náhodné vzorkování hodnot z hloubkové textury v okolí pixelu P. Toto okolí je specifikované určitým poloměrem R. Získaná hloubka z náhodně vygenerované pozice je poté porovnána s hloubkou zkoumaného bodu P. Rozdíl těchto dvou hodnot určuje, zda je zkoumaný pixel zastíněný či nezastíněný.

Zastínění je určeno pomocí útlumové funkce (viz obrázek 4.6), která zároveň určí i intenzitu zastínění vzhledem ke vzdálenosti získaného náhodného bodu od zkoumaného bodu P.

Tato závislost je důležitá, aby nedocházelo k zastínění tělesem, které je již příliš daleko na to, aby mohlo mít na zkoumaný bod P výraznější vliv. Výstupní hodnota z útlumové funkce je akumulována a po prozkoumání všech náhodně vygenerovaných pozic určuje výsledný koeficient pro útlum intenzity osvětlení v daném bodě P. Získaný koeficient se následně zkombinuje s difuzní složkou osvětlení, čímž dojde k zastínění patřičných oblastí.



Obrázek 4.6: Útlumová funkce k určení hodnoty koeficientů pro zastínění. Obrázek převzat z [5].

Vzhledem k tomu, že metoda SSAO pouze aproximuje výpočet pomocí rovnice 4.5, dochází v ní při výpočtu ke vzniku vysokofrekvenčního šumu, který je viditelný obzvlášť na hranách obrazu, kde chybí další hodnoty pro výpočet. Z tohoto důvodu je potřeba výsledný obraz nesoucí SSAO koeficienty rozmazat pomocí nějakého filtru.

Druhá nevýhoda této metody spočívá v pomalosti výpočtů. Již pro rozlišení 1024x768 je vypočteno přibližně 12,5 milionů bodů pro jeden snímek, což značně zpomaluje chod aplikace. Z toho důvodu je vhodné provádět výpočet SSAO ve čtvrtinové velikosti a následně provést up-scaling následovaný rozmazáním obrazu. Výhoda tohoto přístupu spočívá ve výpočtu menšího počtu bodů a dodatečnému rozmazání, které je získáno zadarmo zvětšením textury.

Existuje více způsobů jak výše uvedený algoritmus implementovat. Hlavní rozdíly spočívají v určení průsečíku s okolním terénem, v nalezení náhodných pozic a použité útlumové funkci. Více informací lze nalézt ve [5, 8, 25, 34].

Kapitola 5

Implementace frameworku

Úvodní část této kapitoly se zabývá volbou implementačních prostředků. V podkapitole 5.2 jsou představeny základní stavební bloky navrženého frameworku a podkapitola 5.3 se zabývá popisem manažerů pro správu objektů, jež využívá výsledná aplikace během svého běhu. Poslední podkapitola 5.4 popisuje detailně třídy pro HLSL shadery, které masivně využívá simulační aplikace popsaná v kapitole 6.

5.1 Implementační prostředky

Aplikace využívá multimediální knihovnu Microsoft DirectX, z které jsou použity moduly DirectInput pro získání uživatelského vstupu a Direct3D 10.0 pro vizualizaci 3D grafiky. Vzhledem k tomu, že úvodní seznámení s touto knihovnou je značně náročné, byl zvolen wrapper SlimDX [1] pro jazyk C#, který poskytuje v rámci vývojového prostředí Microsoft Visual Studio 2010 neocenitelnou nápovědu pomocí technologie IntelliSense. Daň za tuto uživatelskou přívětivost je menší úbytek výkonu, jelikož je nutné provádět marshalling[2] mezi nativním kódem DirectX v jazyce C++ a spravovaným kódem vyvinuté aplikace v jazyce C#. Prerekvizita je též nainstalovaný Microsoft .NET framework 4, z kterého jsou využívány některé komponenty.

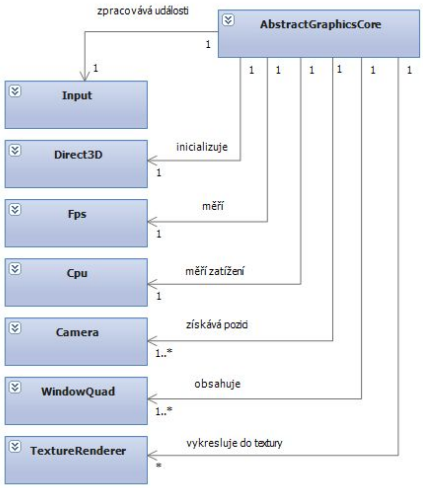
5.2 Jádro frameworku

Nejdůležitější abstraktní třída frameworku je jistě `AbstractGraphicsCore`, od které veškeré aplikace nad tímto frameworkem dědí své chování (viz obrázek 5.1). Třída obsahuje statické proměnné pro název aplikace, vzdálenost ořezávacích rovin, modifikátor pro zobrazení okna na celou obrazovku a modifikátor pro synchronizaci vykreslování s obnovovací frekvencí monitoru. Dále jsou v ní definovány dvě čistě virtuální metody pro inicializaci a destrukci standardně používaných objektů, které se musí pro správnou funkcionalitu přetížit. Třetí čistě virtuální metoda `Render()` je volána pro vykreslení všech objektů na obrazovku. Virtuální metoda `HandleInputs()` obsahuje standardní mapování kláves pro pohyb kamery ve scéně, kde klávesy W,A,S,D jsou použity pro pohyb a myš pro otáčení rozhlížení. Klávesa Escape standardně ukončí aplikaci.

[1] Dostupný na: http://slimdx.org/

[2] Proces transformace paměťové reprezentace objektu do formátu dat vhodných pro přenos mezi dvěma různými platformami nebo aplikacemi. Druhá strana provádí zpětnou transformaci, aby s daným objektem mohla dále pracovat.

Pro získávání vstupů z klávesnice a myši je implementována třída `Input`, která využívá knihovny DirectInput k zachycení zmáčknutých kláves na klávesnici a relativní pohyb myši, z kterého se dá určit jakou vzdálenost myš urazila od posledního známého bodu v rovině XY. K oběma periferiím je v rámci spuštěné aplikace získán exkluzivní přístup, což znamená, že jiné běžící aplikace nemohou tyto prostředky využít dokud pro ně nebudou uvolněny.



Obrázek 5.1: Diagram tříd základních objektů frameworku.

Další důležitou třídou je `Direct3D`, která při inicializaci zjistí na jakém hardwaru je aplikace spuštěna a jaké jsou podporované rozlišení grafického adaptéru. Následně dojde k inicializaci vykreslovací sběrnice, kde se musí vytvořit vykreslovací buffery, Z-buffer a *render states*, které uživateli nabízejí možnost zapnout nebo vypnout blending a Z-buffer. Třída též obsahuje implicitně vytvořené transformační matice pro souřadný systém modelu a projekce (perspektivní, ortogonální). Vzhledem k tomu, že instance této třídy je v aplikace potřeba jenom jedna a je využívána téměř všemi ostatními třídami k vytváření nebo umisťování objektů na vykreslovací sběrnici, bylo rozhodnuto, že implementace této třídy využije návrhový vzor *Singleton*.

Pro umístění pozorovatele ve scéně je využívána třída `Camera`, která dovoluje uživateli definovat pozici a natočení ve scéně. Z těchto dat je vytvářena transformační matice pro souřadný systém kamery. Standardně je ve frameworku při inicializaci vytvořen jeden objekt této třídy, do kterého jsou posílány události ze vstupních zařízení, avšak uživatel může těchto kamer umístit do scény více a dle potřeby mezi nimi přepínat.

Pro vykreslování 2D objektů byla vytvořena třída `WindowQuad`, která dokáže vygenerovat obdélník o určité výšce a šířce a umístit ho na určitou pozici na obrazovce tak, aby vždy směřoval ke kameře. Standardní systém koordinátů DirectX má pozici $[0; 0]$ pro 2D kreslení definovanou přímo ve středu obrazovky, zatímco objekt třídy `WindowQuad` má pozici $[0; 0]$ definovanou v levém horního rohu obrazovky.

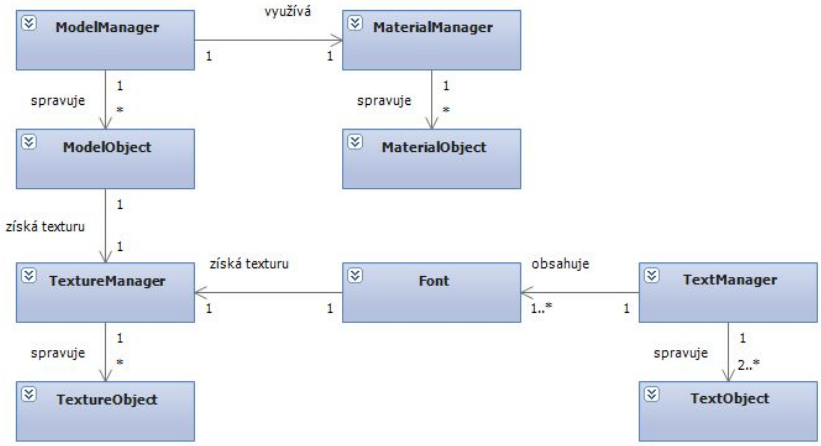
Pro vykreslování do textury byla vytvořena třída `TextureRenderer`, která supluje chybějící Framebuffer z knihovny OpenGL s následujícím omezením. Lze vytvořit libovolné množství textur, avšak textury budou vždy ve formátu *RGBA_Float32*. Jiné textury nejsou touto třídou podporovány.

Poslední dvě třídy v jádře frameworku jsou `FPS` a `CPU`. Jak již název napovídá jedná se o statistické nástroje pro měření vytížení procesoru a vypočítání průměrného počtu

vykreslených snímků za sekundu.

5.3 Manažeři pro správu zdrojů

Každá grafická aplikace využívá mnoho zdrojů jako jsou objekty s modely, textury a texty. Pro tyto účely byli navrhnuti příslušní manažeři (viz obrázek 5.2), kteří mají během chodu aplikace pouze jednu instanci. Z toho důvodu byl využit pro jednotlivé manažery návrhový vzor *Singleton*.



Obrázek 5.2: Diagram tříd pro správu zdrojů.

Třída `ModelManager` spravuje veškeré používané modely v aplikaci, jež jsou reprezentovány objekty třídy `ModelObject`. Načítání modelů lze provést dvěma způsoby. Buď předáním příslušného modelu uloženého v souboru ⟨`model_file`⟩`.mod` nebo pomocí soupisky, která obsahuje veškeré definice modelů a definice materiálů, které tyto modely využívají.

Manažer pro modely dokáže buď vrátit celou kolekci všech spravovaných modelů nebo kolekci obsahující modely v určité skupině a nebo jen jeden model dle názvu souboru s modelem. Struktura souboru s daty modelu je zobrazena níže.

```
<model>
  <texture cube="false" path="../Textures/txTerrain2.dds" />
  <vertices count="4026">
    <triangle>
      <vertex>
        <point x="-105.6704" y="33.0523" z="4.6557" />
        <texcoord u="0.1083" v="0.308" />
        <normal x="0.1064" y="0.1325" z="-0.9855" />
      </vertex>
      <vertex>
         ...
      </vertex>
      <vertex>
         ...
      </vertex>
    </triangle>
```

```
    <triangle>
       ...
    </triangle>
        .
        .
        .
  </vertices>
</model>
```

Ze struktury souboru s modelem je patrné, že každý model má přiřazenu právě jednu texturu, která je uložena v DDS formátu, což je standardní formát uchování textur v prostředí DirectX. Nástroj pro konverzi zdrojových obrázků do toho formátu je součástí DirectX SDK. Parametr `cube=false` označuje, že se jedná o standardní 2D texturu. Pokud je parametr roven true, jedná se o cubemapu složenou ze šesti textur. Struktura dále definuje počet všech vrcholů modelu, jednotlivé skupiny trojúhelníků a pro každý trojúhelník tři vrcholy nesoucí informace o pozici, texturovacích koordinátech a normále. Je vhodné poznamenat, že DirectX má jiný souřadný systém oproti OpenGL, kdy Z-ová osa nabývá kladných hodnot směrem do obrazovky a z toho důvodu je i vykreslování prováděno po směru hodinových ručiček, což musí být bráno v potaz při vytváření dat modelu.

Při načítání modelu je zavolán manažer pro správu textur implementovaný v třídě `TextureManager`. Ten spravuje objekty třídy `TextureObject`, které načítají texturu do paměti a obalují jí strukturami, jež jsou nutné pro mapování textury do shader programů. Výhoda tohoto přístupu spočívá v zamezení duplicit při načítání stejných textur pro různé modely, které se liší jen použitými texturovacími koordináty. Třída `TextureManager` si též udržuje pro každou načtenou texturu počet aktivních objektů, které ji využívají. Pokud tento počet klesne na nulu, je textura automaticky odstraněna z paměti.

V textu výše byl zmíněn ještě druhý způsob načítání modelů pomocí soupisky. Soubor se soupiskou má následující formát.

```
<model_list>
  <materials>
    <material id="riverbed">
      <ambient r="0.5882" g="0.5882" b="0.5882" a="1.000" />
      <diffuse r="0.5882" g="0.5882" b="0.5882" a="1.000" />
      <specular r="0.5490" g="0.5490" b="0.5490" a="1.000" />
      <specular_power factor="61.0000" />
      <optical_density factor="1.5" />
    </material>
        .
        .
        .
    <material id="rocks">
        ...
    </material>
  </materials>
  <models>
    <group id="skybox">
      <model path="Models/Skybox.mod" material="line" />
    </group>
```

```
    <group id="terrain">
      <model path="Models/Line01.mod" material="line" />
      <model path="Models/Line02.mod" material="line" />
        .
        .
        .
    </group>
  </models>
</model_list>
```

Zde je nejzajímavější částí sekce ⟨`materials`⟩, která definuje potřebné konstanty pro osvětlení a index lomu. Tyto hodnoty jsou uloženy v objektech třídy `MaterialObject`, které jsou spravovány pomocí instance třídy `MaterialManager`. Index lomu je využit pro refrakční koeficient $R(0)$ (viz kapitola 6.3), přičemž jako druhý index se bere index vzduchu.

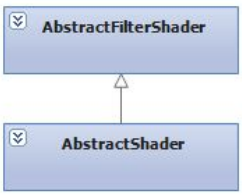
Posledním manažerem je `TextManager`, který spravuje textové řetězce, jež jsou vykreslovány na obrazovku. Data s textovým řetězcem jsou uloženy v objektech třídy `TextObject` a font potřebný pro vykreslení je implementován ve třídě `Font`. Tato třída využívá textury, jež má všechny vykreslitelné znaky položeny vedle sebe a souboru, který takový font popisuje. Každý znak v tomto souboru je popsán v následujícím formátu:

```
65 A 0.185547   0.194336    9
```

První položka je ASCII hodnota znaku, poté následuje samotný znak, třetí a čtvrtá hodnota jsou texturovací koordináty do příslušné textury a poslední parametr určuje šířku znaku v pixelech. Spolu s frameworkem je dodáván jeden druh fontu a při inicializaci jsou ve třídě `TextManager` obsaženy již dva řetězce pro vypsání informace o zatížení CPU a počtu vykreslených snímků za sekundu. Pokud jsou tyto informace nežádoucí, dají se snadno vypnout.

5.4 Struktura HLSL shaderů

V dnešních moderních multimediálních aplikacích je vykreslování modelů, osvětlení a dalších efektů řešeno pomocí shader programů, jež běží na GPU. Jelikož tyto programy mívají do jisté míry podobnou strukturu, byly navrženy dvě abstraktní třídy `AbstractShader` a `AbstractFilterShader` (viz obrázek 5.3), které slouží jako základní stavební bloky pro vytváření 2D a 3D HLSL shader programů.



Obrázek 5.3: Diagram tříd pro vytváření shaderů.

3D vykreslování

Třída `AbstractShader` umožňuje uživateli definovat chování jak vertex tak pixel shader programů. Pro správnou funkčnost je nutné dodefinovat čistě virtuální funkci `Initialize()`, v které proběhne vytvoření specifických zdrojů pro daný shader. Nejčastěji uživatelsky vytvářené zdroje jsou buffery nesoucí konstantní data. Pro jejich rychlé vytvoření je možné použít funkci `CreateConstantBuffer()`. Je však nutné vzít na vědomí, že velikost konstantního bufferu je dle specifikace DirectX vždy rovna násobku 16 bytů. Pokud velikost není rovna násobku 16 bytů, dojde k vyvolání příslušné výjimky. Na konci inicializace je nutné vždy zavolat funkci `InitializeBaseShader()` s jmény souborů, jež obsahují vertex a pixel shader programy.

Aby vertex shader program věděl, jak jsou uložena data ve vertex bufferu, který bude mít na svém vstupu, je nutné dodefinovat čistě virtuální funkci `GetInputElemets()`, která bude vracet pole struktur `InputElement`. Tato struktura vždy obsahuje definici pozice vstupních dat ve vertex bufferu, velikost těchto dat a sémantické označení (např. *POSITION, TEXCOORD, NORMAL,...*).

Pro aktualizaci dat v uživatelsky vytvořených strukturách slouží čistě virtuální funkce `SetShaderParameters()`. Vykreslení vstupních dat na obrazovku či do textury se provádí pomocí veřejné funkce `Render()`. Ta využívá výše jmenované čistě virtuální funkce k tomu, aby aktualizovala data ve všech vytvořených zdrojích, poté je namapovala na grafickou sběrnici a nakonec zavolala příkaz pro vykreslení.

Vertex shader program musí mít vzhledem k návrhu třídy `AbstractShader` také specifickou strukturu. Pro názornost bude tato struktura detailněji rozebrána na kódu uveřejněném níže.

```
//Globals
cbuffer MatrixBuffer {
  matrix worldMatrix;
  matrix viewMatrix;
  matrix projectionMatrix;
  matrix worldViewIT;
};

//Typedefs
struct VertexInputType {
  float4 position : SV_POSITION;
  float2 texCoords : TEXCOORD0;
  float3 normal : NORMAL;
};

struct PixelInputType {
  float4 position : SV_POSITION;
  float2 texCoords : TEXCOORD0;
  float3 normal : NORMAL;
};
```

```
//Vertex Shader
PixelInputType main(VertexInputType input)
{
  PixelInputType output = (PixelInputType)0;

  //Calculate the position
  output.position = mul(float4(input.position, 1.0f), worldMatrix);
  output.position = mul(output.position, viewMatrix);
  output.position = mul(output.position, projectionMatrix);

  //Pass normal and tex coord to pixel shader
  output.texCoords = input.texCoords;
  output.normal = input.normal;

  return output;
}
```

Třída `AbstractShader` standardně poskytuje buffer konstant s transformačními maticemi, který je mapován do prvního bufferu na grafické sběrnici s označením `MatrixBuffer`. Matici pro souřadný systém světa, kamery a projekce musí uživatel zadat sám, matice `worldViewIT` je dopočítána automaticky a slouží ke specifickým transformacím, jež budou popsány v kapitole 6.3.

Struktura `VertexInputType` musí přesně kopírovat uložení dat ve vertex bufferu, jež bylo definováno pomocí zmíněné virtuální funkce `GetInputElements()`. Poslední nutná podmínka je, že vstupní bod do vertex i pixel shader programu musí být vždy funkce označená jako `main()`.

2D vykreslování

Třída `AbstractFilterShader` je specializovaná třída určená jen pro 2D efekty. Vstupní geometrie je vždy poskytována třídou `WindowQuad`, jež byla zmíněna v podkapitole 5.2 a využívá předdefinovaný vertex shader program, který je velmi podobný příkladu, jež byl uveden výše. Z toho důvodu je uživateli umožněno definovat při inicializaci pouze soubor obsahující pixel shader program. Tento program má také určité specifické struktury a pravidla, která budou vysvětlena na příkladu níže.

```
//Globals
Texture2D shaderTexture;
SamplerState textureSampler;

cbuffer ScreenBuffer {
  float screenWidth;
  float screenHeight;
};
```

```
//Typedefs
struct PixelInputType {
  float4 position : SV_POSITION;
  float2 texCoords : TEXCOORD0;
};

//Pixel shader
float4 main(PixelInputType input) : SV_TARGET
{
  //Make offset to 1:1 pixel-texel mapping
  float2 texCoords;
  texCoords.x = input.position.x / screenWidth;
  texCoords.y = input.position.y / screenHeight;

  //Sample color from texture
  float4 textureColor = shaderTexture.Sample(textureSampler, texCoords);

  return textureColor;
}
```

Pro vzorkování textury je potřeba struktura `SamplerState`, která definuje jakým způsobem bude získán vzorek z textury. Standardně je třídou `AbstractShader` vytvořen jeden objekt tohoto typu s adresačním módem typu `WRAP` a filtrováním nastaveným na typ `MinMagMipLinear`. Tyto hodnoty lze při inicializaci shaderu změnit na další podporované hodnoty. Také lze vytvořit další `SamplerState` s jinými parametry pomocí dostupné funkce `CreateTextureSampler()`.

Struktura `PixelInputType` je vstupní struktura získaná z výstupu vertex shaderu, a proto musí přesně kopírovat výstupní strukturu vertex shader programu. Pro třídu `AbsractFilterShader`, jež je určena jen pro 2D zpracování obrazu, vždy platí, že tato struktura obsahuje pouze pozici vykreslovaného fragmentu a příslušné texturovací koordináty.

Při zpracování 2D obrazu často dochází k situaci, kdy je potřeba získat přesnou hodnotu daného texelu. Texturovací koordináty získané z výstupu vertex shaderu však určují pozici přesně mezi dvěma texely, čímž při vzorkování dojde k získání interpolované hodnoty. Pokud by došlo k použití více 2D filtrů za sebou, jež budou využívat interpolovaných hodnot, došlo by k postupné degradaci obrazu. Z toho důvodu třída `AbstractFilterShader` poskytuje buffer konstant `ScreenBuffer`, jež obsahuje výšku a šířku daného obdélníku z objektu třídy `WindowQuad`. Tato výška a šířka je poté použita pro určení nových texturovacích koordinátů (viz příklad), jež umožní přesné vzorkování hodnoty ze středu daného texelu. Tímto způsobem pak nedochází k interpolaci hodnot a postupné degradaci obrazu.

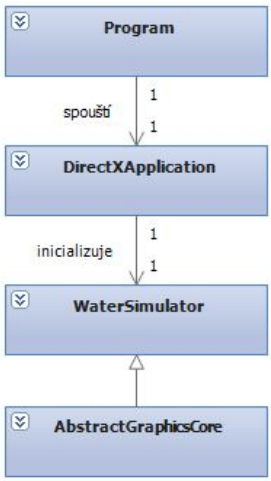
Kapitola 6

Implementace simulační aplikace

Úvodní část této kapitoly se věnuje navázání simulační aplikace na vyvinutý framework, který byl probírán v předcházející kapitole. Podkapitola 6.2 se zaměřuje na implementaci chování vodní hladiny, další podkapitola 6.3 řeší implementaci osvětlení. Podkapitola 6.4 se zaměřuje na implementaci efektu Screen Space Ambient Occlusion a poslední podkapitola 6.6 se zabývá implementací vizualizace vodní hladiny.

6.1 Propojení s vyvinutým frameworkem

V kapitole 5 byla probrána kostra vyvinutého frameworku, který pohání 90% simulační aplikace. Následující diagram na obrázku 6.1 zobrazuje potřebné třídy pro vytvoření spustitelné aplikace.



Obrázek 6.1: Diagram tříd pro inicializaci simulátoru.

Statická třída `Program` slouží jako vstupní bod do aplikace a slouží jen k vytvoření objektu třídy `DirectXApplication`. Tento objekt v prvním kroku vytvoří okno aplikace dle statických parametrů umístěných v třídě `WaterSimulator`. V druhém kroku je vytvořena konkrétní instance této třídy, která obsahuje implementaci pro zpracování událostí a na závěr je spuštěna obsluha událostí. Po tomto kroku je veškeré řízení aplikace předáno do vytvořeného objektu třídy `WaterSimulator`.

Třída `WaterSimulator` je potomkem abstraktní třídy `AbstractGraphicsCore` a obsahuje metody pro inicializaci a destrukci všech potřebných prostředků pro běh simulace. Též obsahuje přetíženou metodu `Render()`, která provádí simulační výpočty a vykresluje výslednou scénu na obrazovku. Celý postup pro jeden simulační krok lze popsat pomocí následujícího pseudokódu.

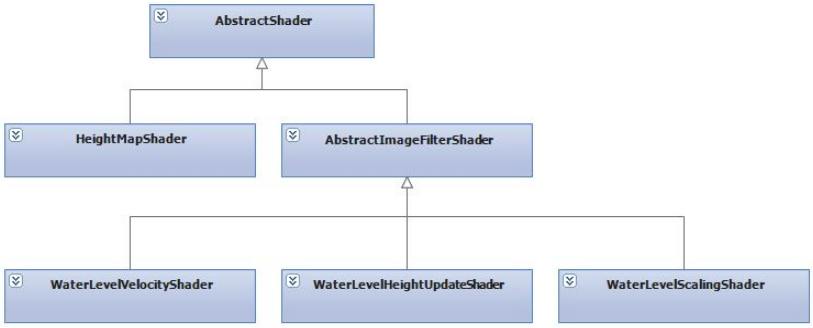
```
void Render()
{
  WaterSimulationStep();
  RenderScene();
  RenderSSAO();
  RenderSkybox();
  RenderTerrain();
  RenderText();
}
```

Metoda `WaterSimulationStep()` provede výpočet chování vody pro definované Δt. `RenderScene()` vykreslí geometrii scény spolu s osvětlením do samostatných textur pro každou složku osvětlení. Tyto textury jsou následně využity v metodě `RenderSSAO()`, která vypočítá koeficienty pro útlum osvětlení a zrekonstruuje z jednotlivých textur výsledný obraz scény. Další metoda přidá do scény skybox, který slouží pro vizualizaci velmi vzdáleného okolí scény. Poslední částí simulace je metoda `RenderTerrain()`, která provede zpětné mapování z 2D do 3D prostoru a vypočítá osvětlení na povrchu vodní hladiny. Metoda `RenderText()` slouží pouze k zobrazení statistických informací o vytížení procesoru a zobrazení informace o počtu vykreslených snímků za sekundu.

Pro výše jmenované metody byla vytvořena celá řada tříd, které zapouzdřují chování HLSL shaderů, jež řeší problémy nastíněné ve výše zmíněném pseudokódu. Detailní popis těchto tříd se bude vyskytovat v následujících kapitolách, které se zaměřují na jednotlivé složky simulace.

6.2 Chování vodní hladiny

Simulace chování vodní hladiny lze rozdělit do dvou kroků. V prvním kroku je simulační model inicializován na počáteční hodnoty. V druhém pak dochází k samotné simulaci. Implementace těchto kroků vychází z kapitoly 3. Pro účely algoritmu byly vytvořeny následující třídy: `HeightMapShader` pro inicializaci výškové mapy, nad kterou bude probíhat simulace, a třídy `WaterLevelVelocityShader`, `WaterLevelScalingShader` a `WaterLevelHeightUpdateShader` pro simulování chování vodní hladiny. Diagram tříd je zobrazen na obrázku 6.2.



Obrázek 6.2: Diagram tříd pro simulační shadery.

Inicializace simulačního modelu

Před započetím samotné simulace je potřeba vytvořit výškovou mapu modelu terénu. Toto chování je implementováno v příslušném vertex a pixel shaderu, jež zapouzdřuje třída `HeightMapShader`. Objekt této třídy je vytvořen a použit pouze jednou při inicializaci simulační metody.

Vertex shader při vykreslení zjistí výšku vrcholu z Y-ové složky pozice zpracovávaného vrcholu a spolu s daty o výsledné pozici, jež je vypočítána pomocí transformačních matic, je tato informace předána dále ke zpracování pixel shaderu. Ten provede normalizaci výšky do intervalu $\langle 0; 1 \rangle$ a následně vypočítá výšku uživatelsky zadaného vodního sloupce tak, aby tato výška byla také normalizována. Obě hodnoty jsou na závěr vykresleny do textury.

Pro připomenutí: formát dat uložených v textuře je zobrazen na obrázku 3.4a. Výsledek, který vznikne po zpracování je zobrazen na obrázku 6.3. Uveřejněný obrázek je pro účely dokumentace mírně upraven. V praxi je modrá oblast s počáteční výškou vody jen slabě viditelná (má menší hodnotu), aby nedošlo k příliš velkému přesahu vodních sloupců oproti korytu řeky. V případě zobrazeném na obrázku 6.3 by došlo k zaplavení celého modelu a obtížnému testování vlastností simulační metody.

Obrázek 6.3: Textura s výškovou mapou terénu (červená barva) a počátečním stavem vodní hladiny (světle modrá barva v pravé části koryta řeky).

Simulace chování vodní hladiny

Na začátku této kapitoly byly zmíněny tři další třídy, které již slouží k samotnému výpočtu simulační metody. První použitý objekt je `WaterLevelVelocityShader`, jež má na vstupu texturu s inicializovaným modelem. Tento objekt implementuje výpočet rychlosti průtoků potrubím (rovnice 3.2) v pixel shaderu. Z rovnice 3.1 nebyla implementována interakce s vnějšími silami ($p_e = 0$) a byl vzat do úvahu předpoklad, kdy atmosférický tlak p_0 je konstantní pro podobnou nadmořskou výšku. Velikost délky potrubí mezi dvěma buňkami je určena jedním pixelem, čímž lze napsat $l = 1$. Tyto úpravy celou rovnici 3.2 zjednoduší do tvaru

$$\eta = f\eta_0 + \Delta t g(h_1 - h_2), \tag{6.1}$$

kde $(h_1 - h2)$ určuje rozdíl výšek dvou sousedních vodních sloupců. Hodnota nefyzikálního koeficientu f pro tření byla zvolena 0,95. Výstupní data s rychlostí proudění vody potrubím jsou poté uložena do textury dle obrázku 3.4b.

Po výpočtu všech rychlostí průtoků potrubím je použit objekt třídy `WaterLevelScalingShader`, který vypočítá v pixel shaderu změnu měřítka dle rovnice 3.6. Na vstupu tohoto shaderu je textura s inicializovaným simulačním modelem a textura s vypočítanou rychlostí proudění vody v potrubí. V prvním kroku výpočtu dojde k načtení aktualizovaných hodnot rychlostí průtoku vody potrubím z pomocné textury. Následně se určí velikost překrývajících se oblasti mezi jednotlivými buňkami dvou aktuálně zpracovávaných vodních sloupců a z této hodnoty se určí objem, který proteče potrubím za jednotku času dle rovnice 3.3. Poté dojde k výpočtu objemu vody, který padá na níže položený vodní sloupec dle rovnic 3.4 a 3.5. Pokud suma těchto objemů překročí objem vody, který je uchovaný ve vodním sloupci, je patřičně upraveno měřítko pro odliv vody na hodnotu menší než 1.0f a tato hodnota je zapsána do alfa kanálu textury se simulačním modelem (viz obrázek 3.4a).

V posledním fázi simulačním kroku je použit objekt třídy `WaterLevelHeightUpdateShader`, který provádí téměř stejný výpočet pro objem odtékající vody jako objekt `WaterLevelScalingShader` s tím rozdílem, že je již brána v úvahu vypočtená změna měřítka z minulého kroku. Navíc je proveden obdobný výpočet i pro objem přitékající vody. Rozdíl těchto dvou vypočtených objemů je na závěr zpětně transformován na novou výšku vodního sloupce, čímž je uzavřen jeden simulační krok o délce Δt.

Hraniční podmínky simulace

Je nutné poznamenat, že originální dokumentace simulační metody neobsahuje důležité informace o hraničních podmínkách a o stabilitě použité metody. Bylo zjištěno, že pro všechny objekty, které provádějí výpočet simulace je nutné nastavit přístup *sampleru* k textuře na typ **BORDER** s hodnotami $(\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0})$, jinak při přístupu za hranice textury dojde k získání dat, které zanesou chybu do výpočtu a v krátkém časovém horizontu zapříčiní zmizení většiny vody ze simulace. Další problém, který musel být vyřešen je nestabilita metody. Empiricky bylo zjištěno, že délka časového kroku musí splňovat následující podmínku

$$\Delta t < \frac{1}{3} h_{max}, \tag{6.2}$$

kde h_{max} je v tomto případě normalizovaná výška v intervalu $\langle 0; 1 \rangle$. Při překročení této podmínky dojde nejdříve k zobrazení artefaktů v obraze a v dlouhodobějším časovém intervalu opět ke ztrátě veškeré informace o výšce vodní hladiny.

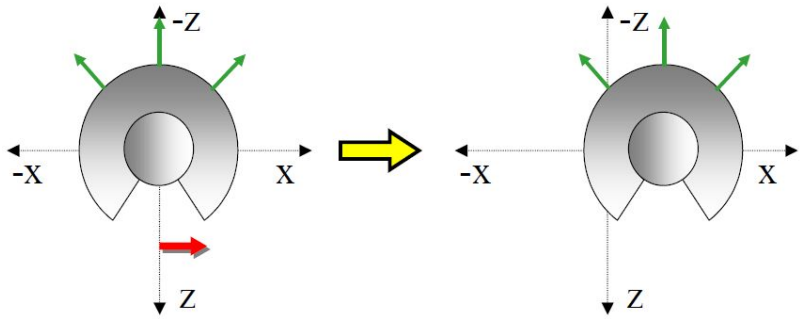
6.3 Osvětlení scény

Jak již bylo zmíněno v předcházejícím textu, osvětlení značně přispívá k reálnosti zobrazené scény. Z toho důvodu je implementováno osvětlení pro každý pixel obrazu, které vychází z kapitoly 4.2 a implementaci lze nalézt ve třídě `LightShader`, která je potomkem třídy `AbstractShader`, což znamená, že je potřeba definovat chování jak pro vertex shader, tak pro pixel shader.

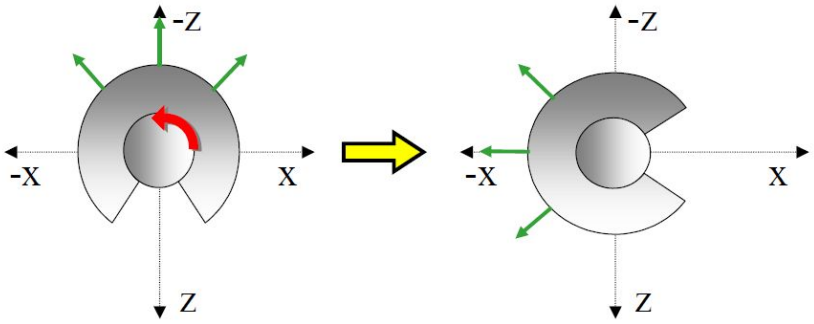
Vertex shader má na vstupu standardní parametry (pozice vrcholu, texturovací koordináty a normálu) a navíc obsahuje parametr určující pozici světla ve scéně. Výsledná pozice vrcholu je získána aplikací transformačních matic. Výpočet osvětlení se běžně provádí v souřadném systému kamery. Z toho důvodu je nutné transformovat pozici světla ve scéně a normály do tohoto souřadného systému. Pro pozici světla je transformace bezproblémová, avšak pro normály je transformaci provést jiným způsobem.

Transformace normál

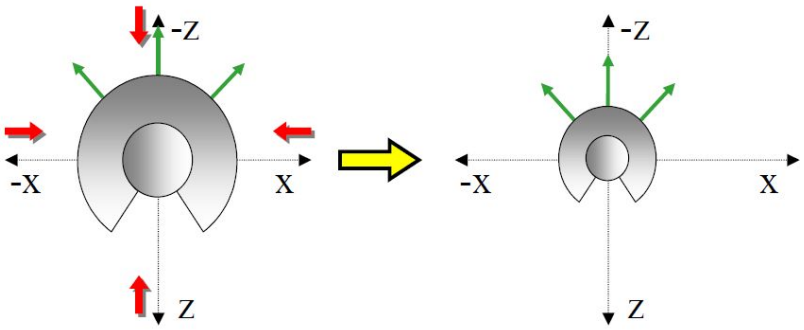
Důvodem je chování normál při jednotlivých transformacích. Na obrázcích 6.4, 6.5 a 6.6 je vidět, že pro translaci, rotaci a uniformní změnu měřítka je vše v pořádku. Problém však nastane při neuniformní změně měřítka, která je zobrazena na obrázku 6.7, kde se normály deformují místo toho, aby zůstaly orientované stejným směrem. Tento problém byl vyřešen pomocí matice, která je vůči provedeným operacím inverzní a transponovaná, čímž dojde k zachování správného směru normál.



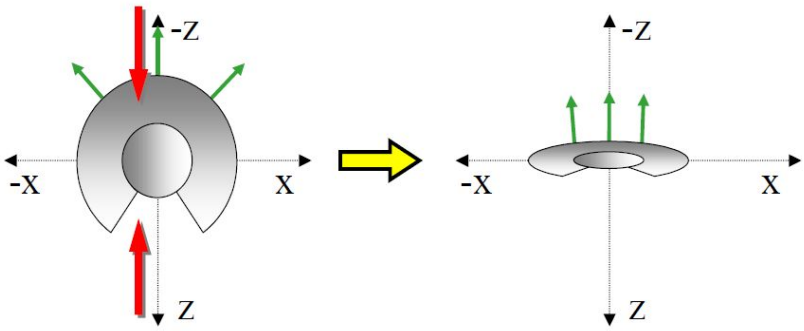
Obrázek 6.4: Výsledný směr normály při translaci.



Obrázek 6.5: Výsledný směr normály při rotaci.



Obrázek 6.6: Výsledný směr normály při uniformní změně měřítka.



Obrázek 6.7: Výsledný směr normály při neuniformní změně měřítka.

Po veškerých transformacích jsou data předána do pixel shaderu, kde již dochází k výpočtu osvětlení dle rovnic 4.1, 4.2 a 4.3.

Fresnelův faktor

Rovnice 4.2 pro výpočet odraženého světla byla navíc rozšířena o Fresnelův faktor, který snižuje míru odlesků v závislosti na materiálu. Efektivní implementace v HLSL shaderu byla převzata z [44] a vypadá následovně.

```
float fresnel(float3 light, float3 normal, float R0)
{
  const float cosAngle = 1-saturate(dot(light, normal));
  float result = cosAngle * cosAngle;
  result = result * result;
  result = result * cosAngle;
  result = saturate(mad(result, 1-saturate(R0), R0));
  return result;
}
```

Parametr *R0* je určen dle rovnice

$$R(0) = \frac{(1 - refactionIndexRatio)^2}{(1 + refactionIndexRatio)^2}, \tag{6.3}$$

přičemž tato hodnota je počítána na CPU z materiálových koeficientů a do shaderu je dodána jako konstanta.

Vstupní data pro SSAO efekt

Mimo výpočtu osvětlení se též pixel shader stará o vypočítání vstupních dat pro efekt SSAO, který na svém vstupu potřebuje normály a hloubku scény. Není vhodné předávat na výstup normály, s kterými se pracuje během výpočtu osvětlení, jelikož tyto hodnoty jsou pro každý vykreslený fragment interpolovány. Lepší je získat neinterpolovanou hodnotu pomocí integrovaných funkcí `ddx` a `ddy`, které slouží k získání derivátu aktuální pozice ve směru X a Y. Pokud je nad těmito deriváty proveden vektorový součin, je získána původní neinterpolovaná hodnota normály.

Hloubka scény je standardně v depth bufferu uložena takovým způsobem, že 90% hodnot obsahuje detailní informace o objektech, které leží nejblíže u pozorovatele. Zbylých 10% je určeno pro vzdálenější objekty. Efekt SSAO se ale nesoustředí na nejbližší objekty, ale analyzuje scénu jako celek, čili je vhodnější mít k dispozici lineární hloubku scény. Ta lze spočítat v souřadné soustavě kamery pomocí rovnice

$$depth = \frac{vertexPos.z - z_{near}}{z_{far} - z_{near}}, \tag{6.4}$$

kde $vertexPos.z$ je vzdálenost vrcholu objektu od kamery; z_{near} označuje ořezávací rovinu, která je nejblíže ke kameře a z_{far} označuje ořezávací rovinu, která je od kamery vzdálena.

Z důvodů, které budou vysvětleny v následujících kapitolách, jež se věnují implementaci efektu SSAO, bylo rozhodnuto, že pro vykreslení bude využit přístup *deferred rendering*, což znamená, že jednotlivé složky osvětlení (intenzita difuzního, zrcadlového a okolního světla spolu s texturou scény a daty pro výpočet efektu SSAO), budou během jednoho průchodu uloženy do separátních textur pro zpracování efektem SSAO.

6.4 Screen Space Ambient Occlusion

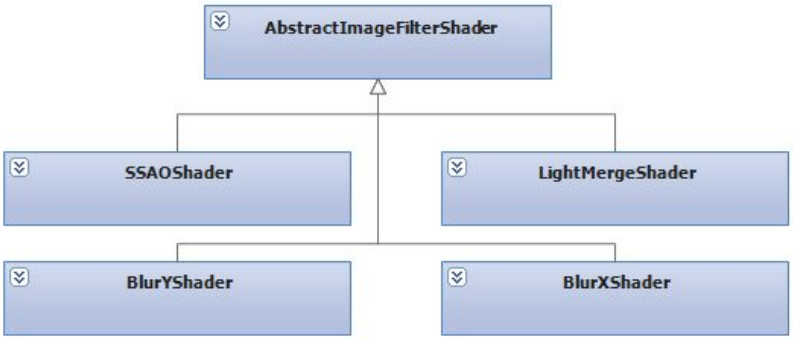
Efekt SSAO slouží k vytvoření stínů, které vznikají mezi dvěma objekty, jež se navzájem zastiňují. Implementace se dá rozdělit do čtyř kroků. V prvním kroku je potřeba vygenerovat vzorkovací vektory. Ve druhém je nutné tyto vektory modifikovat, aby vzorkování probíhalo správně a v třetím kroku se provádí samotný výpočet koeficientů pro zastínění. V posledním kroku dochází k rekonstrukci obrazu ze složek osvětlení a koeficientů pro zastínění.

Implementace efektu vychází z kapitoly 4.3 a je obsažena v třídě `SSAOShader`. Spojení koeficientů pro zastínění s osvětlovacími složkami z předcházející kapitoly je implementováno v třídě `LightMergeShader`. Diagram tříd je zobrazen na obrázku 6.8.

Během výpočtu efektu SSAO vzniká nepříjemný vysokofrekvenční šum, který se nepříjemně projevuje při pohybu kamery "zrněním"obrazu. Z toho důvodu je na konci výpočtu využit bilaterální filtr, který je implementován v třídách `BlurXShader` a `BlurYShader`. Implementace tohoto filtru je detailněji popsána v následující podkapitole 6.5.

Generování náhodných vektorů

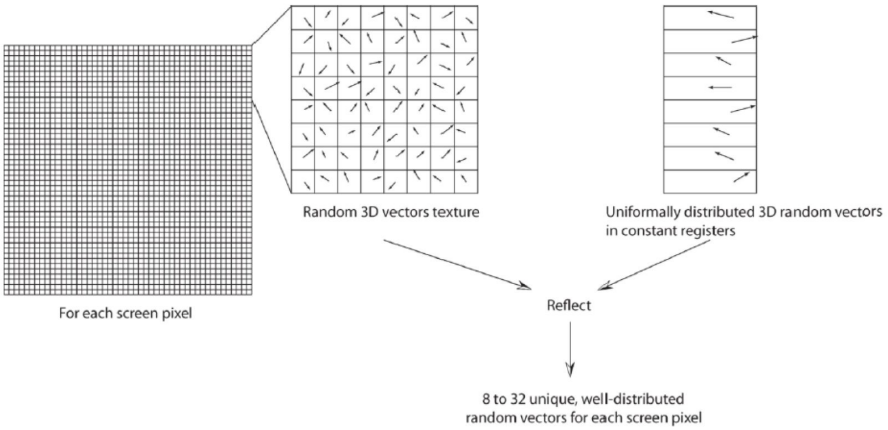
Při implementaci třídy `SSAOShader`, která provádí výpočet efektu SSAO vzniklo několik problémů. Prvním z nich bylo určit způsob, jakým se budou náhodně vzorkovat data z textury, jež obsahuje normály a hloubku scény. Nakonec byl zvolen přístup uveřejněný v [5]. Pro vzorkování okolí zpracovávaného fragmentu je využita malá pomocná textura, která obsahuje Perlinův šum. Dále jsou potřeba předem vypočítané uniformní vektory pro jednotkovou kouli.



Obrázek 6.8: Diagram tříd pro efekt SSAO.

Z pomocné textury je získána šumová hodnota, která představuje náhodnou normálu. Pokud je nad uniformním vektorem a náhodnou normálou provedena operace `Reflect()` (viz obrázek 6.9), lze vytvořit 8 až 32 unikátních, dobře distribuovaných vektorů pro každý pixel obrazovky.

Běžně je doporučováno mít předem vygenerovaných 16 uniformních vektorů, ale při testování bylo zjištěno, že bohatě stačí i 10 uniformních vektorů pro velmi slušně vypadající výsledek (viz obrázek 6.11), přičemž dojde k menšímu nárůstu výkonu v řádu maximálně 10-ti vykreslených snímků za sekundu.

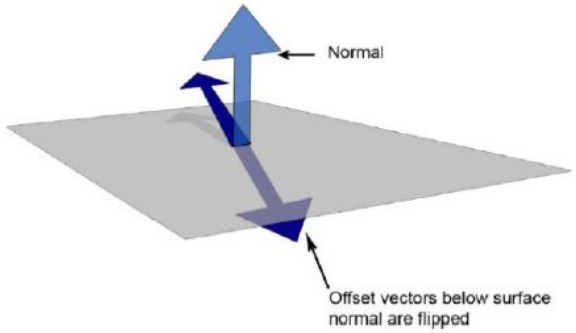


Obrázek 6.9: Vygenerování náhodného vektoru v shaderu. Obrázek převzat z [5].

Modifikace náhodného vektoru pro vzorkování

Po vytvoření náhodného vzorkovacího vektoru je potřeba tento vektor dále upravit takovým způsobem, aby nemohlo dojít ke vzorkování celé textury s normálami a hloubkami scény z libovolného zpracovávaného fragmentu. Toho je docíleno vynásobením vektoru hodnotou $scale = 1 - depth$, kde $depth$ je získaná hloubka aktuálně zpracovávaného fragmentu v intervalu $\langle 0; 1 \rangle$. Jednotlivé složky vzorkovacího vektoru se často po tomto kroku ještě násobí jednou desetinou šířky a výšky zpracovávané textury, aby byl zachován poměr stran zpracovávaného obrazu.

Před vzorkováním okolí aktuálně zpracovávaného fragmentu musí proběhnout kontrola, zda získaný vzorkovací vektor náhodou nesměřuje pod terén. Tuto informaci lze jednoduše získat pomocí skalárního součinu vzorkovacího vektoru a normály zpracovávaného fragmentu. Pokud je součin záporný je nutné vzorkovací vektor převrátit, což demonstruje obrázek 6.10.



Obrázek 6.10: Převrácení vzorkovacího vektoru v případě, kdy vektor směřuje pod terén. Obrázek převzat z [5].

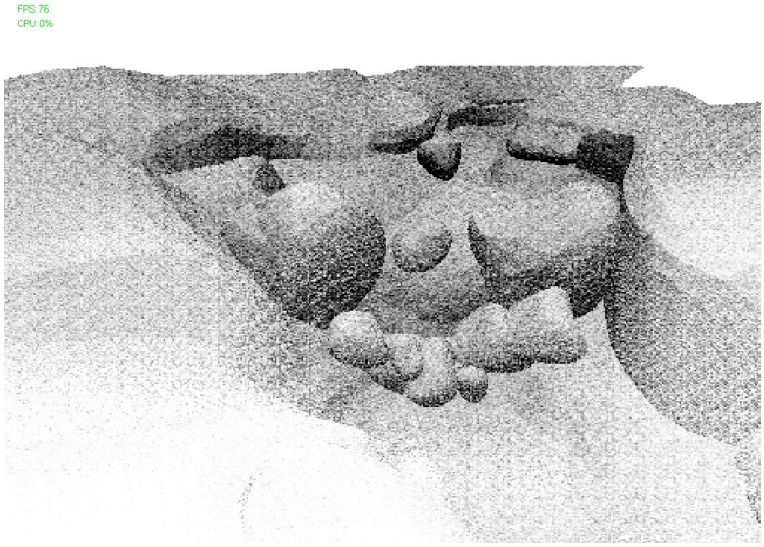
Výpočet koeficientů pro zastínění

Nyní již nic nebrání získání vzorků v okolí zpracovávaného fragmentu a porovnání jejich hloubek. Problém však může nastat v případě, kdy aktuálně zpracovávaný fragment a získaný vzorek leží v jedné rovině. Pokud budou jejich hloubky různé vzhledem k perspektivní projekci, dojde k zastínění zpracovávaného fragmentu i přes to, že se oba body nemohou navzájem nijak zastínit. Z toho důvodu je vhodné udělat test souběžnosti normál pomocí skalárního součinu obou normál a získanou hodnotu ze skalárního součinu odečíst od jedničky. Pokud bude výsledná hodnota větší než nula, tak se zastínění fragmentu může provést, jinak nikoliv.

Vypočtený rozdíl hloubek a výsledek testu na souběžnost normál je použit v útlumové funkci, která byla z obrázku 4.6 implementována v prostředí HLSL shaderu takto:

```
occlusionSum += step(falloff, depthDiff) * normDiff
                * (1.0f - smoothstep(falloff, maxDistance, depthDiff));
```

Funkce `step(falloff, depthDiff)` vrací hodnotu 1 pokud `depthDiff` $\geq$ `falloff`, jinak 0. Tato funkce vyjadřuje tu část útlumové funkce (viz obrázek 4.6), jež obsahuje malou hodnotu epsilon s nulovým zastíněním pro vzorky, které se nacházejí přibližně ve stejné vzdálenosti. Proměnná `falloff` vyjadřuje toto epsilon a její hodnota byla zvolena 0.00001f. Proměnná `normDiff` je výše popsaný test na souběžnost normál. Poslední část s funkcí `smoothstep` simuluje zbytek útlumové funkce pomocí vyhlazené Hermitovy interpolace [22] v intervalu $\langle 0; 1\rangle$. Proměnná `falloff` ve funkci `smoothstep` označuje minimální hranici, `maxDistance` maximální hranici a `depthDiff` aktuální hodnotu. Pokud se aktuální hodnota nachází mezi minimem a maximem dochází k interpolaci a mapování aktuální hodnoty do intervalu $\langle 0; 1\rangle$, jinak se při překročení minima vrací hodnota 0 a při překročení maxima hodnota 1. Hodnota proměnné `maxDistance` byla zvolena na vzdálenost 15.0f jednotek.



Obrázek 6.11: Koeficienty zastínění pro vykreslenou scénu.

Po získání všech vzorků je akumulovaný výsledek uložen v proměnné `occlusionSum`, která je po ukončení vzorkovacího cyklu zprůměrována počtem vzorků (v této implementaci 10) a výsledek je odečten od hodnoty jedna. Tímto způsobem dojde k získání koeficientu pro zastínění aktuálně zpracovávaného fragmentu. Výsledná textura s koeficienty je zobrazena na obrázku 6.11.

Sloučení osvětlení se SSAO koeficienty

Posledním krokem je sloučení výsledných koeficientů pro zastínění spolu s texturami obsahující složky osvětlení. Tuto funkcionalitu poskytuje objekt třídy `LightMergeShader`, který provede rekonstrukci obrazu scény dle rovnice 4.4.

V žádném dokumentu, který byl použit pro vytváření efektu SSAO nebylo řečeno na jakou složku osvětlení by se měly koeficienty pro zastínění použít. Z toho důvodu bylo osvětlení v kapitole 6.3 vykresleno přístupem *deferred rendering* tak, aby mohlo být snadno vyzkoušeno, na jaké složce osvětlení bude mít efekt nejvýraznější vliv.

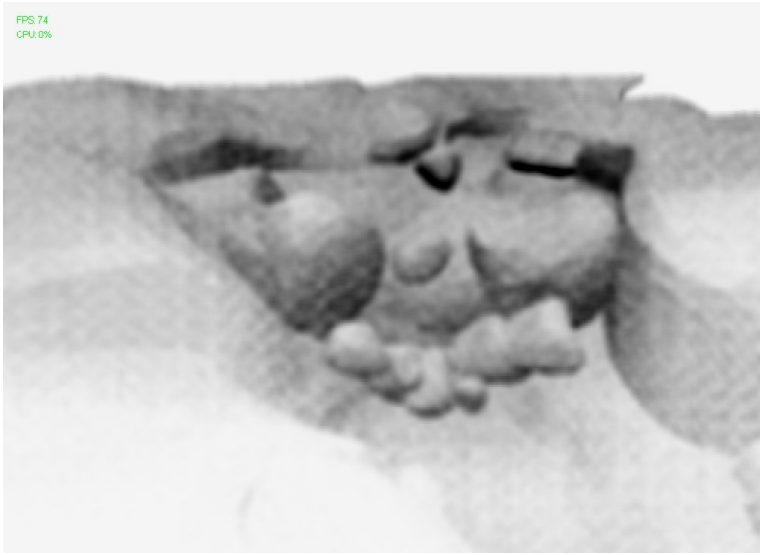
Bylo zjištěno, že vizuálně nejlepších výsledků je dosaženo, pokud se koeficienty pro zastínění aplikují na složku s okolním osvětlením. Pokud byly koeficienty aplikovány na difusní osvětlení, nebyl vliv zastínění téměř viditelný. V případě, kdy se koeficienty aplikovaly na celkový výsledek osvětlení s materiálovou texturou, došlo k výraznému potemnění celé scény.

Efekt SSAO je značně výpočetně náročný. Z toho důvodu byla provedena optimalizace, kdy zpracování probíhá jen ve čtvrtinové velikosti, čímž nedojde k tak velkému poklesu vykreslených snímků za sekundu jako u neoptimalizované verze.

6.5 Rozmazání obrazu

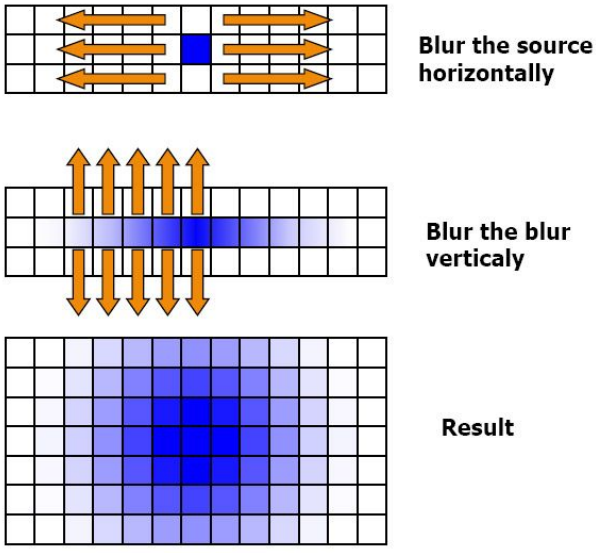
V předchozí podkapitole bylo zmíněno, že při výpočtu efektu SSAO vzniká nepříjemný vysokofrekvenční šum, který se nepříjemně projevuje při pohybu kamery "zrněním"obrazu, což je dobře viditelné na obrázku 6.11. Z toho důvodu se výsledný obraz s útlumovými

koeficienty filtruje pomocí bilaterálního filtru, tak aby byl získán výsledek, jež je zobrazen na obrázku 6.12.



Obrázek 6.12: Filtrovaný obraz pomocí bilaterálního filtru.

Bilaterální filtr je takový filtr, který při rozmazávání respektuje hrany obrazu. Jeho implementace lze nalézt v třídách `BlurXShader` a `BlurYShader`. Filtr vychází z klasického konvolučního filtru pro Gaussovské rozmazání o velikosti kernelu 9×9. Standardní implementace konvoluce je pro bilaterální filtr rozšířena o hloubkovou funkci, která pro získané vzorky vezme v úvahu i jejich vzdálenost. Pokud rozdíl vzdáleností překoná určitý práh, daný vzorek do konvoluční sumy nepřispívá žádnou hodnotou.



Obrázek 6.13: Implementace Gaussovského rozmazání na GPU. Obrázek převzat z [1].

Z textu výše je možné si všimnout, že filtr je implementovaný ve dvou třídách místo očekávané jedné třídy. To je dáno optimalizovaným návrhem filtru na GPU. Pokud by byl filtr implementovaný jen v jedné třídě, tak by mohl provést jen jednoprůchodové zpracování

obrazu. Pro konvoluční kernel o velikosti 9×9 je potřeba při jednoprůchodovém zpracování provést vzorkování 81 texelů na jeden zpracovávaný fragment, což je poměrně velká hodnota.

V případě separace filtru do dvou průchodů, kdy první průchod zpracovává data vodorovně a druhý horizontálně je potřeba provést vzorkování jen 9-ti texelů na průchod, což dává v celkovém součtu jen 18 potřebných vzorků na jeden zpracovaný fragment. Nevýhoda v použití dvou průchodů spočívá pouze v potřebě dočasné textury, do které bude uložen mezivýsledek po zpracování prvním průchodem. Celou situaci pro snadnější pochopení demonstruje obrázek 6.13.

6.6 Vizualizace vodní hladiny

Vizualizace vodní hladiny se skládá za tří kroků. V prvním je třeba provést rekonstrukci 3D modelu z 2D textury obsahující simulační data s aktualizovanými výškami vodních sloupců. Pro připomenutí, tato textura je získána pomocí objektu třídy `WaterLevelHeightUpdateShader` (viz podkapitola 6.2). V druhém kroku je nutné se zbavit nepotřebných dat (terén bez vodní hladiny), jež způsobují artefakty v obraze vlivem jevu, jež je známý pod označením *z-fighting*[1]. V posledním kroku je pro realistický vzhled potřeba určit optické vlastnosti, aby rekonstruovaný model připomínal reálnou vodní hladinu.

Implementaci vizualizace vodní hladiny lze nalézt ve třídě `TerrainShader`, která je potomkem třídy `AbstractShader`, což znamená, že je potřeba definovat chování jak pro vertex shader, tak pro pixel shader. Dále je třeba zmínit, že před vykreslením je potřeba obnovit hloubkovou mapu v *depth bufferu* získanou při osvětlení scény, aby bylo možné vykreslovat 3D objekty do obrazu, který byl zpracován 2D metodou SSAO.

Rekonstrukce 3D modelu

Pro rekonstrukci 3D modelu je zapotřebí geometrie z objektu třídy `TerrainGrid`, který vytvoří pravidelnou síť čtverců v rovině XZ. Počet čtverců je o jedna menší než výška a šířka simulační textury tak, aby se vrcholy této sítě daly namapovat přímo na střed každého texelu.

Umístění této sítě musí přesně lícovat s umístěním originálního otexturovaného modelu. Toto umístění je kritické proto, aby geometrie simulačního modelu kopírovala geometrii originální modelu, do kterého se bude přidávat vodní hladina.

Přesné umístění sítě ve scéně lze provést zpětnou projekcí z obrazovky do souřadného systému modelu pomocí standardní funkce `Vector3.Unproject()`. Výsledkem této funkce je translační vektor, který je použit jako jeden ze vstupů vertex shaderu třídy `TerrainShader`.

Mezi další vstupy patří standardní parametry (pozice vrcholu, texturovací koordináty a normála), proměnná nesoucí pozici pozorovatele a proměnné s konstantami udávajícími výšku a šířku simulační textury. Z těchto konstant jsou poté vypočítány texturovací koordináty pro vzorkování přesného středu potřebného texelu.

Zpracování vrcholu ve vertex shaderu se provádí následujícím způsobem. Ze simulační textury je načtena celková výška terénu s vodním sloupcem a tato výška je doplněna do Y-ové souřadnice zpracovávaného vrcholu. Následně je tento vrchol přesunut na výslednou pozici pomocí transformačních matic a předán na vstup pixel shaderu.

[1] Situace během vykreslování, při níž má obdobnou hodnotu v depth bufferu dvě nebo více grafických primitiv, čímž dochází k zobrazení primitiva, které nemá být vidět.

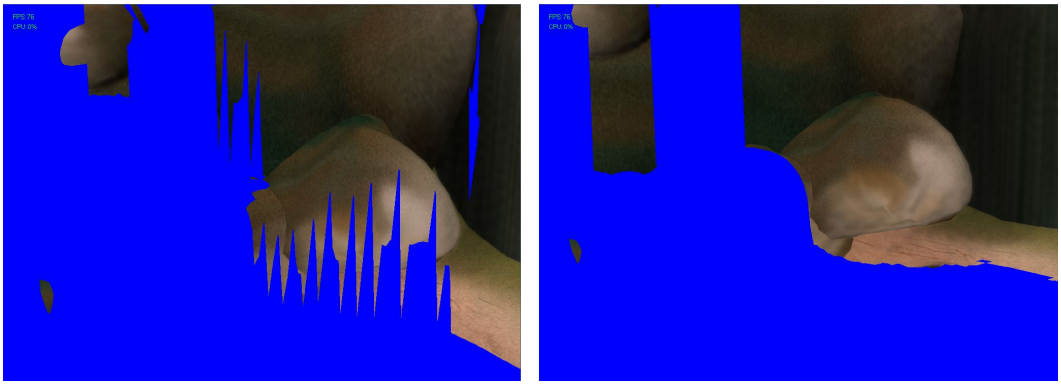
Vertex shader též musí vypočítat další parametry, které budou vstupem pixel shaderu. Konkrétně se jedná o pozici vrcholu v souřadné soustavě modelu a pozici vrcholu v souřadné soustavě kamery. Dále je vypočítán normalizovaný vektor, který udává směr z vrcholu k pozici pozorovatele. Tyto data budou využita pro výpočet optických vlastností vodní hladiny a jejich přesné využití bude rozebráno v textu níže.

Odstranění nepotřebných dat ze zpracování

Jak již bylo zmíněno v úvodu této podkapitoly během vykreslování dochází k jevu, jež se označuje jako *z-fighting*, kdy je zobrazeno grafické primitivum, jež by nemělo být správně vidět. Z toho důvodu musí pixel shader, který je součástí třídy `TerrainShader` odstranit takové fragmenty, jež neobsahují vodní hladinu a neměly by být správně vidět.

Tuto operaci řeší metoda `DiscardTest()`, která se v pixel shaderu provede jako první před jakýmkoliv dalším zpracováním. Fragmenty, které se mají zahodit jsou rozpoznány z textury obsahující simulační data. Pokud daný fragment, který koresponduje s daným texelem na této textuře, neobsahuje žádnou vodní hladinu, je rozhodnuto, že dojde k jeho vyloučení ze zpracování.

Uvedený způsob měl však jednu vážnou chybu. Vzhledem k diskretizaci hodnot na simulační textuře docházelo k zobrazení artefaktů v podobě nepříjemných špiček (viz obrázek 6.14a). Tyto špičky jsou nejčastěji pozorovatelné v místech, kde buď nastává velká změna výšky terénu, anebo vodní hladina skokově přechází na břeh.



(a) (b)

Obrázek 6.14: Odstranění nepotřebných fragmentů: a) Artefakty v podobě špiček; b) Obraz bez špiček.

K odstranění tohoto problému byl zvolen přístup, při kterém dochází i k vzorkování dalších osmi okolních hodnot ze simulační textury okolo aktuálně zpracovávaného fragmentu a pokud ani jeden vzorek neobsahuje vodní hladinu, tak je fragment vyloučen ze zpracování. Výsledek po použití tohoto přístupu je zobrazen na obrázku 6.14b.

Výpočet optických vlastností vody

Po odstranění nepotřebných fragmentů ze zpracování, musí pixel shader vypočítat normály pro vygenerovanou vodní hladinu, jež budou sloužit k výpočtu optických vlastností vody. Normály lze vypočítat z parametru nesoucí pozici vrcholu v souřadné soustavě modelu pomocí integrovaných funkcí `ddx` a `ddy`, které slouží k získání derivátu aktuální pozice

ve směru X a Y. Pokud je nad těmito deriváty proveden vektorový součin, je získána hodnota normály, jež je využita pro výpočet vektorů **R** a **T** (viz obrázek 3.5).

Vektor **R** slouží jako vyhledávací vektor pro texturu s cubemapou, z které je získán vzorek pro zrcadlový odraz vzdáleného okolí na vodní hladině. Vektor **T** je předán funkci `refraction()`, jež vypočítá barvu pro refrakční paprsek, který směřuje pod vodní hladinu.

V tomto místě bylo z části upuštěno od návrhu v kapitole 3.4, jelikož lineární vyhledávání kombinované s binárním vyhledáváním implementované v metodě `refraction()` velmi zpomalovalo chod aplikace. Místo toho bylo navrhnuto jiné řešení, které využívá texturu obsahující hloubkovou mapu pro výpočet efektu SSAO (viz sekce Výpočet vstupních dat pro SSAO v kapitole 6.3) a texturu s osvětlením scény sloučenou s koeficienty pro zastínění efektu SSAO (viz sekce Sloučení osvětlení se SSAO koeficienty v kapitole 6.4).

Metoda `refraction()` využívá pozici vrcholu v souřadné soustavě kamery k určení lineární vzdálenosti od kamery. Následně je získána lineární hloubka z hloubkové textury, která byla určena pro metodu SSAO. Pomocí těchto hloubek je určen počet kroků, které posouvá texturovací koordináty v textuře nesoucí celou rekonstruovanou scénu se SSAO efektem. Výsledný vzorek, který je z této textury získán je prohlášen za barvu terénu pod vodní hladinou.

V posledním kroku je vypočítán Fresnelův faktor pomocí stejného kódu, jehož implementace je uvedena v sekci Fresnelův faktor v kapitole 6.3. Pomocí rovnice 3.8 je určena výsledná intenzita barvy ze získaných vzorků pro odraz a refrakci. Výsledný obraz s optickými vlastnostmi vodní hladiny je zobrazen na obrázcích 7.1a a 7.1b, které se nachází v následující kapitole.

Kapitola 7

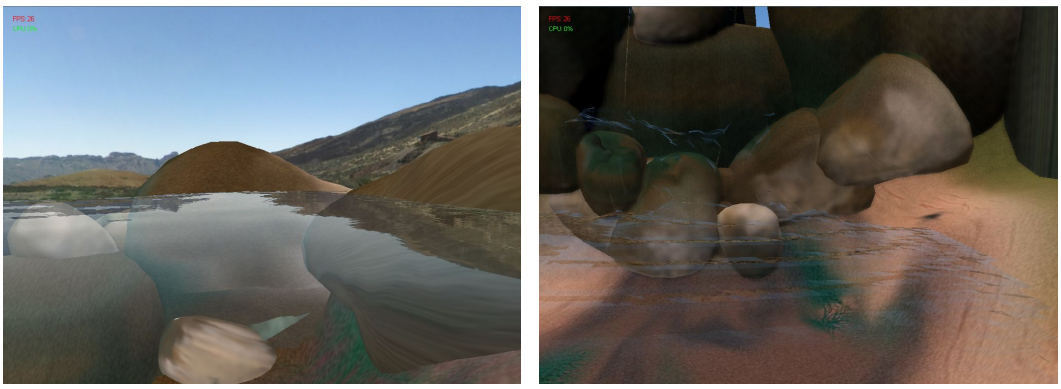
Vyhodnocení výsledků

Úvodní část této kapitoly se věnuje hlavnímu přínosu diplomové práce. Podkapitola 7.2 hodnotí použitelnost metody na základě počtu vykreslených snímků za sekundu a poslední podkapitola 7.3 se zaměřuje na použitelnost metody a příslušných omezení.

7.1 Hlavní přínos

V rámci implementace simulace vody byl hlavní přínos v dodefinování hraničních a simulačních podmínek, které nejsou v originálních materiálech uvedeny, a bez kterých simulace nebude fungovat (viz podkapitola 6.2).

Dále byla provedena úprava v algoritmu pro vizualizaci vodní hladiny, která využívá efekt SSAO k vykreslování zastínění pod vodní hladinou. Tato úprava není sice tak vizuálně přesná jako metoda popsaná v kapitole 3.4, avšak i přesto poskytuje velmi pěkné výsledky (viz obrázky 7.1a a 7.1b).

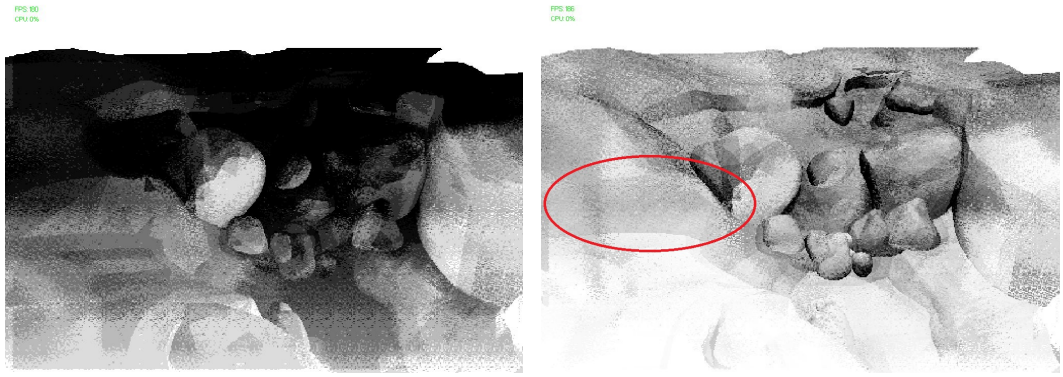


(a) (b)

Obrázek 7.1: Vizualizace vodní hladiny: a) Část vodní hladiny, kde je dobře pozorovatelný odraz z okolí; b) Část vodní hladiny, kde je dobře pozorovatelné implicitní vytváření vln.

V originálních materiálech též není zmíněno, že při rekonstrukci 3D modelu z diskretizované 2D textury vznikají nepříjemné artefakty v podobě špiček (viz obrázek 6.14a). Tento problém byl vyřešen rozhodovacím algoritmem, který vyloučí ze zpracování takové fragmenty, jež ve svém okolí neobsahují vodní hladinu. Výsledek zobrazený na obrázku 6.14b vypadá poté mnohem lépe.

Při implementaci metody SSAO bylo nalezeno také několik problémů, které nebyly nikde popsány. První problém spočíval v enormním zastínění ploch, které ležely přibližně v jedné rovině (viz obrázek 7.2a). Tento problém byl odstraněn přidáním testu na souběžnost normál, který se promítl do útlumové funkce. Výsledný obraz po tomto testu se výrazně zlepšil (viz obrázek 7.2b).



(a) (b)

Obrázek 7.2: Útlumové koeficienty SSAO: a) Koeficienty při nadměrném zastínění ploch ležících v jedné rovině; b) Koeficienty po aplikaci testu na souběžnost normál. Červeně zvýrazněná oblast označuje artefakt, který se při pohybu kamery posouvá v obraze.

V červeně zvýrazněné oblasti na obrázku 7.2b je též částečně znatelný druhý problém, který musel být vyřešen. Při pohybu kamery docházelo k posunu takovýchto pruhů se stíny ve scéně, což vizuálně nepůsobilo příliš dobře. Tento problém byl odstraněn zavedením měřítka, které ovlivnilo velikost vzorkovacího vektoru (viz sekce Modifikace náhodného vektoru v kapitole 6.4), čímž došlo k zamezení vzorkování celé textury.

(a) (b) (c)

Obrázek 7.3: Aplikace SSAO koeficientů na různé složky osvětlení. Červeně zvýrazněná oblast označuje dobře pozorovatelný vznik stínů. a) Koeficienty použity na složku s okolním osvětlením; b) Koeficienty použity na složku s difuzním osvětlením; c) Koeficienty požity na osvětlení složené ze všech osvětlovacích složek.

Poslední problém při implementaci SSAO efektu vyvstal při slučování útlumových koeficientů se složkami osvětlení. V žádném materiálu nebylo popsáno na jakou složku osvětlení se mají útlumové koeficienty aplikovat. Byly vyzkoušeny tři možnosti: aplikovat útlumové koeficienty na složku s okolním osvětlením (viz obrázek 7.3a), na složku s difuzním osvětlením (viz obrázek 7.3b) a na obraz, kde jsou již všechny složky osvětlení složeny dohromady

(viz obrázek 7.3c). Červeně zvýrazněná oblast na výše jmenovaných obrázcích určuje místo, kde je nejlépe pozorovatelný rozdíl při vzniku zastínění.

Bylo zjištěno, že na difuzní složce osvětlení nemají koeficienty téměř žádný vliv. Naopak největší útlum osvětlení nastal při použití útlumových koeficientů na obraz, pro který byly veškeré složky osvětlení složeny dohromady spolu s materiálovou texturou. Tato situace však nepůsobila vizuálně příliš dobře, jelikož došlo k rapidnímu potemnění celé scény. Nejlépe vizuálně vypadající varianta nastala při aplikování útlumových koeficientů na složku s okolním osvětlením, kdy byl efekt znatelný, avšak nezpůsoboval tak rapidní potemnění celé scény.

7.2 Měření počtu vykreslených snímků za sekundu

Se simulační aplikací byly provedeny experimenty, které se zaměřovaly na počet vykreslených snímků v závislosti na rozlišení simulačního gridu. Jako referenční testovací grafické karty byly použity GeForce GTX 590 a GeForce GTX 285M. Výsledky testů shrnuje tabulka 7.1.

	800×600	1024×768	1920×1080
GeForce GTX 590	135–140	80–100	35–40
GeForce GTX 285M	40–50	25–30	10–12

Tabulka 7.1: Počet vykreslených snímků za sekundu v závislosti na použitém rozlišení.

Z této tabulky je dobře znatelné, že simulace je dobře použitelná v nižších rozlišeních i na starších grafických kartách, přičemž byly použity i některé pokročilé techniky jako je SSAO, *enviromental mapping* a *deferred rendering*. Dále je z této tabulky znatelné, že metoda není ještě příliš vhodná pro nasazení do interaktivních multimediálních aplikací jako jsou hry, které nejčastěji používají rozlišení 1920×1080. Tato situace by možná mohla být jiná při použití nejnovějších grafických karet, které se právě dostávají na trh. U nich by průměrný počet vykreslených snímků za sekundu mohl sahat přibližně k hodnotě vyšší jak 50 i v tak velkém rozlišení.

7.3 Zhodnocení simulační metody

Implementovaná simulační metoda bude během krátkého časového horizontu zajímavá pro multimediální aplikace, které potřebují simulovat vodní hladinu v reálném čase. Při testování byly zjištěny i některé nevýhody a omezení dané simulační metody.

Největší problém vyplývá z převodu terénu na výškovou mapu. Pokud se pod vzorkovanou výškou nachází nějaká prohlubeň, vodní hladina nebude zasahovat do této prohlubně a v obrazu při perspektivní projekci vznikne nepříjemná díra, což omezuje použití této metody pouze na přirozené venkovní prostředí, kde je potřeba nejčastěji simulovat jezera nebo vodní toky.

Simulační metoda též zvládá simulovat chování vody i pro skokové výšky v terénu jako je například vodopád. Problém je však při vizualizaci, kde se provádí zpětná rekonstrukce simulace do 3D prostoru a při níž vznikají artefakty v podobě nepříjemných špiček (viz obrázek 6.14a), které nevypadají příliš dobře. Naštěstí se tento problém dá řešit pomocí vzorkování a vyhodnocení okolních fragmentů, jak bylo zmíněno v kapitole 6.6.

V poslední řadě při vizualizaci vzniká problém, kdy vodní hladina kvůli diskretizaci simulačních hodnot dosti špatně kopíruje břeh zakřiveného říčního koryta. Tento problém by se dal řešit pomocí částicových metod, které budou simulovat cákání vody nebo pěnění vody při přechodu vodní hladiny na břeh a tím tyto mezery skrýt pomocí billboardů.

Kapitola 8

Závěr

V úvodní kapitole diplomové práce byly nastudovány aktuálně využívané přístupy pro simulaci vodní hladiny. Z jejich výhod a nevýhod byl vybrán algoritmus pro simulaci říčního toku, který se jevil jako vhodný pro simulaci vody v reálném čase. Popis konkrétního algoritmu, jehož model vychází z fyzikálních zákonů o hydrostatickém tlaku, byl nastudován a popsán v kapitole 3 této práce.

Diplomová práce se v kapitole 4 dále zabývala reprezentací scény, jež je složena z upraveného testovacího modelu a osvětlení scény. Testovací model byl volen tak, aby pokrýval co nejvíce možných simulačních scénářů. Pro osvětlení byl vybrán Phongův osvětlovací model, který je popsán v kapitole 4.2 a jako rozšíření a náhrada globálního osvětlení je použita metoda Screen Space Ambient Occlusion popsaná v kapitole 4.3.

Práce se dále v kapitole 5 zabývala implementací snadno použitelného a rozšířitelného frameworku, nad kterým byla celá demonstrační aplikace postavena. V kapitole 6 bylo popsáno, jakým způsobem je nad tímto frameworkem postavena simulační aplikace. Poslední kapitola 7 vyhodnotila výsledky této práce.

Z výsledků je zřejmé, že simulační metoda je použitelná pro simulaci říčního toku v reálném čase nad generickým terénem, avšak při současné generaci grafických karet není použitelná pro interaktivní multimediální aplikace, které běží ve vysokém rozlišení. Jediná možnost jak tuto metodu použít pro tyto aplikace je snížit rozlišení simulačního gridu či použít tuto metodu pro rychlý výpočet dat, které budou poté vizualizovány rychlým offline přístupem.

V rámci rozšíření by bylo vhodné metodu rozšířit o částicové systémy jako je například cákání vody, pěnění vody či bublinky, které vznikají ve vodní hladině. Tento přístup by mohl dostatečně zamaskovat zmíněné nevýhody s prudkou změnou výšky a špatným kopírováním zakřiveného terénu. Jako nejvhodnější se v tomto směru jeví práce [20], jež by se dala snadným způsobem propojit se stávající simulační metodou. Dalším zajímavým rozšířením by mohlo být upravení simulační metody tak, aby podporovala LOD a mohla být použita i pro velké venkovní plochy.

Literatura

[1] Game Rendering » Gaussian Blur Filter Shader [online]. `http://www.gamerendering.com/2008/10/11/gaussian-blur-filter-shader/`, October 2008 [cit. 2012-04-18].

[2] Almaral, D.: 3D max river water stream [online]. `http://www.turbosquid.com/FullPreview/Index.cfm/ID/354010`, 2007 [cit. 2011-12-26].

[3] Anderson, J.: *Computational fluid dynamics: the basics with applications.* McGraw-Hill series in aeronautical and aerospace engineering, McGraw-Hill, 1995, ISBN 9780070016859.

[4] Anguelov, B.: DirectX10 Tutorial 8: Lighting Theory and HLSL [online]. `http://takinginitiative.net/2010/08/30/directx-10-tutorial-8-lighting-theory-and-hlsl/`, 2010 [cit. 2011-12-27].

[5] Chen, C.: Screen Space Ambient Occlusion in StarCraft II [online]. `http://www.cise.ufl.edu/~cchi/SSAO.pdf`, October 2010 [cit. 2011-12-28].

[6] Chiba, N.; Sanakanishi, S.; Yokoyama, K.; aj.: Visual simulation of water currents using a particle-based behavioural model. *The Journal of Visualization and Computer Animation*, ročník 6, č. 3, 1995: s. 155–171, ISSN 1099-1778.

[7] Dobashi, Y.; Kaneda, K.; Yamashita, H.; aj.: A simple, efficient method for realistic animation of clouds. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, ISBN 1-58113-208-5, s. 19–28.

[8] Drobot, M.: Practical use of Screen Space Ambient Occlusion. In *GCDC'08*, Reality Pump, 2008.

[9] Enright, D.; Marschner, S.; Fedkiw, R.: Animation and Rendering of Complex Water Surfaces. *ACM Transactions on Graphics*, 2002: s. 21, 3, 736–744.

[10] Foster, N.; Fedkiw, R.: Practical Animation of Liquids. In *Proceedings of ACM SIGGRAPH 2001*, 2001, s. 23–30.

[11] Foster, N.; Metaxas, D.: Realistic animation of liquids. *Graph. Models Image Process.*, ročník 58, September 1996: s. 471–483, ISSN 1077-3169.

[12] Harris, M. J.: Fast Fluid Simulation on the GPU. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Addison-Wesley Professional, April 1, 2004, ISBN 0321228324, str. 816.

[13] Holmberg, N.; Wünsche, B. C.: Efficient modeling and rendering of turbulent water over natural terrain. In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '04, New York, NY, USA: ACM, 2004, ISBN 1-58113-883-0, s. 15–22.

[14] Iglesias, A.: Computer graphics for water modeling and rendering: a survey. *Future Generation Computer Systems*, ročník 20, 2004: s. 1355–1374.

[15] Irving, G.; Guendelman, E.; Losasso, F.; aj.: Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Trans. Graph.*, ročník 25, July 2006: s. 805–811, ISSN 0730-0301.

[16] Kapasi, U. J.; Rixner, S.; Dally, W. J.; aj.: Programmable Stream Processors. *Computer*, ročník 36, č. 8, Srpen 2003: s. 54–62, ISSN 0018-9162.

[17] Kass, M.; Miller, G.: Rapid, stable fluid dynamics for computer graphics. *SIGGRAPH Comput. Graph.*, ročník 24, September 1990: s. 49–57, ISSN 0097-8930.

[18] Kipfer, P.; Westermann, R.: Realistic and interactive simulation of rivers. In *Proceedings of Graphics Interface 2006*, GI '06, Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2006, ISBN 1-56881-308-2, s. 41–48.

[19] Lasosso, F.; Gibou, F.; Fedkiw, R.: Simulating Water and Smoke with an Octree Data Structure. *ACM Transactions on Graphics*, 2004: s. 23, 3, 457–462.

[20] Lei, S.-I. E.; Chang, C.-F.: A Fast Rendering Method of Clouds Using Shadow-View Slices. In *Pacific Graphics 2008*, 2008.

[21] Liu, Y.; Liu, X.; Wu, E.: Real-Time three-dimensional Fluid Simulation on GPU with Complex Obstacles. In *Proceedings of Pacific Conference on Computer Graphics and Applications*, 2004, s. 247–256.

[22] Lorentz, R. A.: *Multivariate Birkhoff Interpolation*. Springer, první vydání, November 1992, ISBN 3540558705, 202 s.

[23] Luna, F. D.: *Introduction to 3D Game Programming with Direct 3D 10: A Shader Approach*. Plano, TX, USA: Wordware Publishing Inc., 2008, ISBN 1598220535, 9781598220537.

[24] Maes, M. M.; Fujimoto, T.; Chiba, N.: Low-Memory and Interactive-Rate Animation of Water-Column Based Flows. *The Journal of the Society for Art and Science*, ročník 7, č. 1, July 2008: s. 1–13, ISSN 1347-2267.

[25] Mittring, M.: Finding next gen: CryEngine 2. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA: ACM, 2007, s. 97–121.

[26] Miyazaki, R.; Dobashi, Y.; Nishita, T.: A Fast Rendering Method of Clouds Using Shadow-View Slices. In *Computer Graphics and Imaging 2004*, 2004, s. 93–98.

[27] Mould, D.; Yang, Y.-H.: Modeling water for computer graphics. *Computers & Graphics*, ročník 21, č. 6, 1997: s. 801–814.

[28] Müller, M.; Charypar, D.; Gross, M.: Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, ISBN 1-58113-659-5, s. 154–159.

[29] Neyret, F.; Praizelin, N.: Phenomenological simulation of brooks. In *Eurographics Workshop, Computer Animation and Simulation '01*, Springer, Eurographics, 2001, s. 53–64.

[30] O'Brien, J. F.; Hodgins, J. K.: Dynamic simulation of splashing fluids. In *Proceedings of the Computer Animation*, CA '95, Washington, DC, USA: IEEE Computer Society, 1995, ISBN 0-8186-7062-2, s. 198–.

[31] Policarpo, F.; Oliveira, M. M.; Comba, J. a. L. D.: Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph.*, ročník 24, č. 3, Červenec 2005: s. 935–935, ISSN 0730-0301.

[32] Premžoe, S.; Tasdizen, T.; Bigler, J.; aj.: Particle-Based Simulation of Fluids. *Computer Graphics Forum*, ročník 22, č. 3, 2003: s. 401–410, ISSN 1467-8659.

[33] Rochet, F.: *Simulation Réaliste de Ruisseaux en Temps Réel.* Diplomová práce, Université Joseph Fourier, 2005.

[34] Sainz, M.: Real-Time Depth Buffer Based Ambient Occlusion. In *Game Developers Conference 2008*, NVIDIA, 2008.

[35] Schlick, C.: An Inexpensive BRDF Model for Physically-Based Rendering. *Comput. Graph. Forum*, ročník 13, č. 3, 1994: s. 233–246.

[36] Schneider, J.; Westermann, R.: Towards Real-Time Visual Simulation of Water Surfaces. In *Proceedings of the Vision Modeling and Visualization Conference 2001*, VMV '01, Aka GmbH, 2001, ISBN 3-89838-028-9, s. 211–218.

[37] Sethian, J.: Level Set Methods: An initial value formulation [online]. `http://math.berkeley.edu/~sethian/2006/Explanations/level_set_explain.html`, 2010 [cit. 2011-12-26].

[38] Stam, J.: Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, ISBN 0-201-48560-5, s. 121–128.

[39] Sun, Y.; Fracchia, F. D.; Mark; aj.: Rendering the Phenomena of Volume Absorption in Homogeneous Transparent Materials. In *in the 2nd Annual IASTED International Conference on Computer Graphics and Imaging (CGIM'99*, 1999, s. 283–288.

[40] Thon, S.; Ghazanfarpour, D.: A Semi-Physical Model of Running Water. In *Eurographics UK 2001 Conference Proceedings*, 2001, s. 53–59.

[41] Thon, S.; Ghazanfarpour, D.: Real-Time Animation of Running Waters Based on Spectral Analysis of Navier-Stokes Equations. In *Computer Graphics Internationa*, 2002, s. 333–346.

[42] Webber, M. J.: *Handbook of Optical Materials*. CRC Press, první vydání, September 2002, ISBN 978-0849335129, 536 s.

[43] Weisstein, E. W.: "Del."From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/Del.html`, April 2012 [cit. 2012-05-5].

[44] Wloka, M.: Fresnel Reflection [online]. `http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/HLSL_FresnelReflection/docs/FresnelReflection.pdf`, July 2004 [cit. 2012-04-21].

[45] Wu, E.; Liu, Y.; Liu, X.: An Improved Study of Real-Time Fluid Simulation on GPU. *Journal of Computer Animation and Virtual World*, 2004: s. 15, 3–4, 139–146, ISSN 1546-4261.

[46] Zwillinger, D.: *Handbook of differential equations*. Academic Press, třetí vydání, 1998, ISBN 9780127843964, str. 15.

Příloha A

Obsah DVD

Přiložené DVD obsahuje:

- Spustitelný program (./bin/)
- Data s texturami a použitými modely (./data/)
- Krátké demonstrační video z běhu aplikace (./demo/)
- Text práce v elektronické podobě (./doc/)
- Prerekvizity pro kompilaci kódu (./install/)
- Shader programy (./resources/)
- Zdrojový kód programu (./src/)