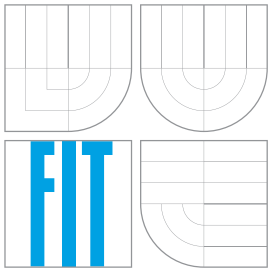
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GPU IMPLEMENTACE RESAMPLING ANTIALIASINGU

GPU IMPLEMENTATION OF RESAMPLING ANTIALIASING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ONDŘEJ SVOBODA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ STARKA

BRNO 2016

Zadání bakalářské práce/17945/2015/xsvobo0k

Vysoké učení technické v Brně - Fakulta informačních technologií

Výzkumné centrum informačních technologií Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Svoboda Ondřej**
Obor: Informační technologie
Téma: **GPU implementace Resampling Antialiasingu**
GPU Implementation of Resampling Antialiasing
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte metody Antialiasingu pro počítačovou grafiku. MSAA, některé morfologické metody a RSAA.
2. Seznamte se s OpenGL verze 4.3 nebo vyšší.
3. Implementujte metodu RSAA pomocí OpenGL a GLSL.
4. Implementujte demonstrační applikaci, která bude umět načíst model a ukázat rozdíl mezi žádným AA, MSAA a RSAA.
5. Volitelně vytvořte plakát či video k práci.

Literatura:

- Reshetov, A.: Reducing Aliasing Artifacts through Resampling

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

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

Vedoucí: **Starka Tomáš, Ing.**, VCIT FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Výzkumné centrum informačních technologií
Božetěchova 2, 612 66 Brno

prof. Ing. Tomáš Hruška, CSc.
vedoucí ústavu

Abstrakt

Předmětem této bakalářské práce bylo implementovat resampling antialiasing pro OpenGL. Základem bylo již existujícím řešení pro DirectX. Jsou zde představeny různé metody antialiasingu používaných v dnešních grafických aplikacích. Dále je zde popsána implementace aplikace i s rozdíly oproti originálnímu řešení. Námi implementovaný resampling antialiasing byl také porovnán s ostatními vyhlazovacími metodami i s originálním řešením. Hodnocena byla kvalita obrazu a rychlost metody.

Abstract

The aim of this thesis is the implementation of resampling antialiasing for OpenGL. This thesis is based on already existing solution for DirectX. We are introducing several antialiasing methods used in todays graphical aplications. Next we are describing application implementation with differences against the original solution. Our implementation of resampling antialiasing was compared with other antialiasing methods and with the original solution. Image quality and performace were evaluated.

Klíčová slova

Antialiasing, RSAA, MSAA, SADP, postprocessing

Keywords

Antialiasing, RSAA, MSAA, SADP, postprocessing

Citace

SVOBODA, Ondřej. *GPU implementace Resampling Antialiasingu*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Starka Tomáš.

GPU implementace Resampling Antialiasingu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Starky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Svoboda
17. května 2016

Poděkování

Chtěl bych poděkovat vedoucímu své bakalářské práce Ing. Tomáši Starkovi za podporu, ochotu a cenné rady při konzultacích.

Obsah

Kapitola 1

Úvod

S tím, jak se zvyšuje výkon počítačů, stoupá i touha lidí po stále dokonalejší a lépe vypadající počítačové grafice. Již od doby, kdy existují první monitory, se lidé snaží, aby výstupem počítačů byla grafika nejenom hezká, ale také co nejvíce realistická. Jedním z jevů, který jim v tom brání, je aliasing. Tento problém se vyskytuje prakticky u každé počítačové grafiky a nějakou dobu se nejspíše i vyskytovat bude. Na svět sice přicházejí monitory se stále lepším rozlišením, zároveň se ale objevují nové technologie, kde malé rozlišení je stále problém, jako například virtuální realita.

Řešení aliasingu se nazíva antialiasing. Tímto názvem se rozumí technika na odstranění nebo potlačení aliasingu. V dnešní době najdeme celou řadu antialiasingových metod, z nichž některé se zaměřují spíše na kvalitu, jiné na výkon a zbylé se snaží jít někde středem. Z historických metod tu máme supersampling, který, ač se jedná o jednu z nejstarších metod, stále podává nejlepší výsledky. Mezi moderní techniky antialiasingu patří celá řada morfologických metod, které obraz zpracovávají až po samotném vykreslení a snaží se v tomto obraze najít chyby a ty následně opravit.

Tato bakalářská práce se zabývá jednou z technik antialiasingu a tou je Resampling Antialiasing. Tato metoda patří mezi modernější a pracuje jak během samotného renderování, tak i zpracovává výsledný obraz. Cílem této metody je striktně se držet jednoho barevného vzorku na pixel, protože vzorkování je jednou z nejnáročnějších akcí v počítačové grafice. Teorie týkající se Resampling Antialiasingu a aliasingu obecně je popsána v druhé kapitole. Dále jsou zde popsány i další vyhlazovací metody jako je Multisampling Antialiasing, který s Resampling antialiasingem blízce souvisí a také některé moderní morfologické vyhlazovací metody.

V další kapitole se věnujeme návrhu samotné aplikace. Je zde nastíněno zakladní fungování aplikace a také zdůvodňujeme některé postupy v samotné implementaci. Dále jsou zde zmíněny všechny cizí knihovny a nástroje, které byly využity při tvorbě a implementaci této aplikace.

Čtvrtá kapitola se poté zabývá již konkrétní implementací. Je zde zmíněno z jakých částí se aplikace skládá a dále jak probíhá samotné vyhlazování pomocí resamplingu. Je zde také upozorněno na odlišnosti od původní implementace, protože tato práce vychází již z existujícího řešení.

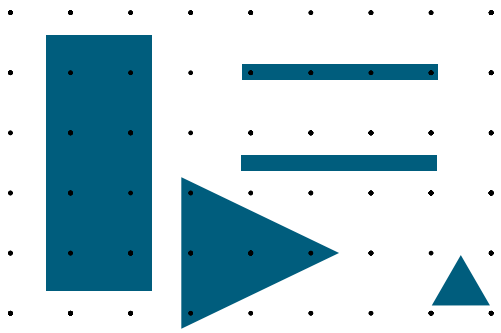
V poslední části jsou zhodnoceny výsledky a to jak po stránce rychlosti tak kvality. Námi implementovanou vyhlazovací metodu zde porovnáváme s jinými vyhlazovacími technikami.

Kapitola 2

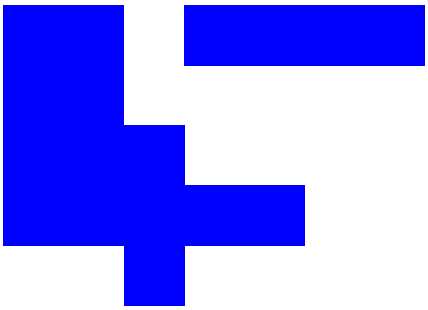
Teorie

Aliasing je jev, který nastává při převodu spojitého signálu na diskrétní. Projevuje se tím, že signál vzniklý rekonstrukcí ze vzorků, není shodný s originálním signálem. Podmínkou vzniku aliasingu je nesplnění Shannonova teorému popsaného v článku od C. E. Shannona [13]. Ten říká, že přesná rekonstrukce spojitého, frekvenčně omezeného signálu z jeho vzorků je možná jen tehdy, pokud byla vzorkovací frekvence vyšší, než dvojnásobek nejvyšší harmonické složky vzorkovaného signálu. Pokud tato podmínka není splněna, dochází ke změně frekvence oproti původnímu signálu.

V počítačové grafice se aliasing může projevovat při rasterizaci, při které se vektorově definované objekty převádějí do rastrové podoby, která již může být reálně zobrazena například na monitoru. Aliasing se zde nejvíce projevuje na hranách objektů, kde se hranice objektu pohybuje mezi několika řadami pixelů. Dalším problémem jsou malé objekty, které se nemusejí vykreslit vůbec, stejně jako ostré hrany objektů, které mohou měnit tvar. Tento jev je demonstován na obrázcích 2.1 a 2.2. Toto je popsáno v knize od M. K. Agostona [1]. Aliasing způsobuje problémy také u textur. Tomuto typu aliasingu se říká prostorový (spatial) aliasing. Typické pro něj je, že ho můžeme rozpoznat již z jediného snímku a není zapotřebí animace.



Obrázek 2.1: Vektorová reprezentace, černými tečkami jsou označeny středy pixelů pro rasterizaci.



Obrázek 2.2: Aliasing zapříčiněný rasterizací, můžeme vidět artefakty jako je mizení objektů atd.

Dalším typem aliasingu je tzv. temporal aliasing. V počítačové grafice je způsoben nízkou snímkovou frekvencí monitoru. Pokud se tedy v animaci pohybuje velkou rychlostí nějaký předmět, urazí mezi jednotlivými snímky velkou vzdálenost a obraz tak nepůsobí plynule. Řešením by bylo zvýšit snímkovou frekvenci monitoru, což přímo odpovídá Shan-

nonovu teorému. V praxi toto řešení ale není často proveditelné, proto se často zavádí technika tzv. Motion Blur. Ta místo toho, aby renderovala předmět zobrazený ostře, vyrenderuje předmět částečně rozmazaný. Díky tomu působí výsledný obraz mnohem více plynuleji. Další projev temporal alisasingu může být pozorován u rotujících předmětů, jako například vrtule letadla. Díky nízké snímkové frekvenci se může zdát, že se předmět točí na opačnou stranu nebo pomaleji než ve skutečnosti.

Řešením problému s aliasingem je antialiasing. V počítačové grafice se pod pojmem antialiasing rozumí metoda pro odstranění nebo zmírnění důsledků aliasingu, neboli artefaktů vzniklých rasterizací. Používá se k tomu několik přístupů. Nejpřirozenějším je zvýšení vzorkovací frekvence, neboli zvětšení rozlišení u výstupního obrazu. Tento princip přímo odpovídá Shannonovu vzorkovacímu teorému. V praxi bohužel není často možný. Další možností je pre-filtering. Zde se každý pixel bere jako plocha. Dle toho, jakou část této plochy zabírá objekt scény, je tomuto pixelu přiřazena intenzita, dle které se poté získá výsledná barva pixelu [8]. Dále můžeme použít tzv. post-filtering. Zde dochází ke zvýšení vzorkovací frekvence, ovšem rozlišení výstupního obrazu zůstává zachováno. Znamená to tedy, že pro každý výstupní pixel se generuje více vzorků a výsledná barva pixelu se získá zprůměrováním těchto vzorků [1]. Poslední možností je postprocessing. Při něm dochází k vyhlazení výstupních dat tím způsobem, že pro každý pixel pomocí nějakého filtru vypočítá jeho nová barva. Tento filtr bere v potaz i okolí pixelu. Znamená to tedy, že dochází k vyhlazení hran, ale i k rozmazání celého obrazu.

2.1 Supersampling Antialiasing

Jednu z nejzákladnějších metod antialiasingu popisuje William J. Leler [5]. Jmenuje se supersampling antialiasing (SSAA) a funguje na velmi jednoduchém principu, kdy pro každý pixel je generováno více samplů[1]. Pro každý tento sample je spuštěn fragment shader. Tedy například při renderování se čtyřmi samply je renderován obraz s čtyřikrát větším rozlišením, než je původní. Výsledná barva pixelu se poté získá zprůměrováním těchto samplů. To má za následek, že hrany a textury vypadají lépe. SSAA zvládá vyhlazování i u průhledných textur. V současné době se jedná o metodu s nejlepšími výsledky. Její největší nevýhodou je ovšem to, že je velmi výpočetně náročná, protože pro obraz s daným rozlišením se musí generovat několikanásobně více samplů. Z tohoto důvodu není tato metoda v současnosti příliš používána.

2.2 Multisampling Antialiasing

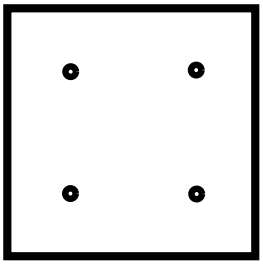
Podobně jako SSAA funguje multisampling antialiasing (MSAA), ovšem s tím rozdílem, že oproti SSAA je mnohem méně výpočetně náročný. Tuto metodu popsal ve svém článku Kurt Akeley [2]. MSAA funguje na principu, že pokud všechny samply v pixelu náleží ke stejnému trojúhelníku, tak tento pixel se nenachází na hraně, tudíž zde není nutné provádět vyhlazování. Naopak pokud různé samply v pixelu náleží k jiným trojúhelníkům, tak tento pixel leží na hraně a je nutné provést jeho vyhlazení. I v tomto případě je ovšem fragment shader spuštěn pouze jednou a barva se vzorkuje na středu pixelu a je poté přiřazena příslušným samplům. Samply se zde používají k určení pokrytí pixelu jednotlivými trojuhelníky. To přináší oproti SSAA značné zrychlení.

[1] jedná se o anglický ekvivalent pro české slovo vzorek.

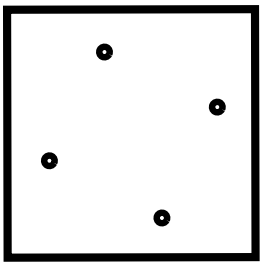
Speciální variantou MSAA je tzv. Deffered MSAA, popsána v [9]. Tato metoda využívá stejné samply jako MSAA, z nichž se navzorkuje hloubka. Tato informace se poté přenáší do postprocessing fáze, kde je hloubka každého samplu porovnána se středem pixelu. Tím se samply rozdělí na ty, které náleží stejnému trojúhelníku jako střed a na ty odlišné. Podle toho se poté určí pozice, ze které se provede bilineární interpolace čtyř nejbližších pixelů. Tím získáme barvu pixelu.

2.2.1 Způsob rozložení vzorků

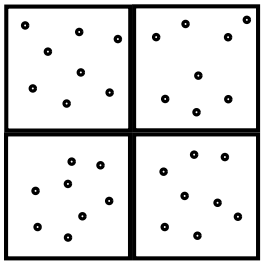
Způsob rozložení jednotlivých samplů v pixelu se může lišit dle verze MSAA. Tyto jednotlivé způsoby jsou popsány v [4]. Základní způsob rozložení samplů v pixelu je formou mřížky (obrázek 2.3). Tato metoda má výhodu jednoduché implementace. Její nevýhodou je to, že může docházet ke vzniku artefaktů, které jsou zapříčiněny pravidelným rozložením. Lidské oko je navíc citlivé k pravidelně se opakujícím chybám. Tyto chyby řeší stochastické rozložení samplů (obrázek 2.5). V tomto případě jsou samply v pixelu rozloženy náhodně, což má za následek vznik šumu. Ten ovšem lidské oko vnímá mnohem méně, než pravidelné artefakty. Emulace stochastického rozložení se nazývá jittering. Nevýhodou tohoto řešení je již samotné náhodné rozložení a dále je také výpočetně náročnější než jiná řešení. Na podobném principu funguje další rozložení, kde jsou samply uspořádány v mřížce a ta je následně pootočena tak, aby žádné dva samply v pixelu nebyly ve stejné vertikální nebo horizontální rovině (obrázek 2.4). Tato metoda řeší poměrně dobře artefakty vzniklé pravidelným vzorkováním a navíc není tak výpočetně náročná jako náhodné rozložení samplů.



Obrázek 2.3: Rozložení podle pravidelné mřížky



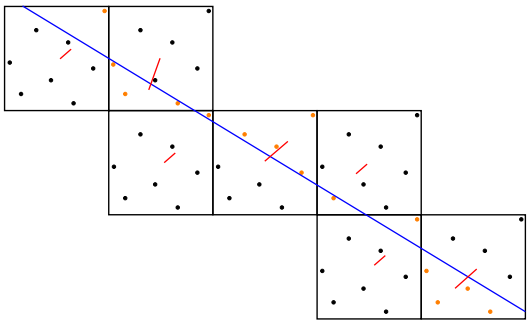
Obrázek 2.4: Rozložení podle pootočené mřížky



Obrázek 2.5: Rozložení stochastické

2.3 Resampling Antialiasing

Resampling antialiasing je metoda, jejímž autorem je Alexander Reshetov [12]. Využívá podobně jako supersampling nebo multisampling více samplů na pixel. V pre-processing fázi je pro každý pixel vytvořena bitová maska o velikosti odpovídající počtu samplů. V této masce každému samplu přísluší jeden bit. Pokud je tento bit nastaven na 1, znamená to, že sample náleží ke stejnému polygonu jako střed pixelu. Nulou jsou poté označeny ty samply, které náleží jiným polygonům nežli střed pixelu. Tato maska je poté využita v postprocessing fázi, kdy slouží jako index do připravené lookup tabulky. Z této tabulky lze získat offset, který udává pozici, ze které bude provedena bilineární interpolace mezi čtyřmi nejbližšími pixely. Ukázku těchto offsetů můžeme vydět na obrázku 2.6.



Obrázek 2.6: Ukázka offsetu v RSAA

2.3.1 Rozdělení vzorků do dvou clusterů

V preprocessing fázi je nutné rozdělit samply do dvou skupin (clusterů) podle toho, zda náleží stejnému polygonu jako střed pixelu či nikoliv. Pokud všechny samply náleží stejnému polygonu jako střed, tak pro tento pixel není nutné provádět antialiasing a výsledná barva pixelu bude dána podle barvy ve středu pixelu. Pokud alespoň jeden sample náleží jinému polygonu nežli střed, je nutné provádět antialiasing. V praxi se dále zavádí určitý práh, který říká, jak moc samply musejí být odlišné, aby bylo nutné provádět antialiasing. K rozdělení samplů do clusterů je možné využít několik metod [12].

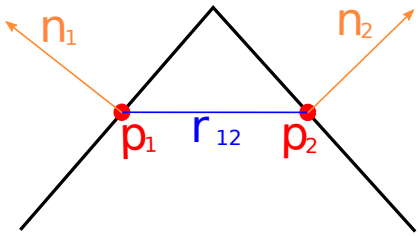
Jednou z používaných metod je porovnávání hloubky středu pixelu a daného samplu. Pokud se hloubky výrazně liší, tak je možno říci, že střed a daný sample náleží jiným polygonům. Tato metoda má ovšem nedostatky při detekování rohů, kdy dva odlišné polygony mohou být vydávány za jeden. Tuto metodu využívá například deferred multisampling [9].

Další metodou je využití normál samplu a středu pixelu s tím, že provedeme jejich skalární součin. Tato metoda dokáže detekovat rohy, ale má nedostatek v tom, že nedokáže odlišit dva polygony, které jsou rovnoběžné, ale mají rozdílnou hloubku. V praxi se také může využívat kombinace této metody s metodou porovnávání hloubky, nicméně problémy mohou stále nastat u povrchů, které jsou vodorovné a jejich hloubka je jen málo rozdílná [12].

Pro využití v RSAA představil pan Alexander Reshetov [12] novou metodu nazvanou *sum of the absolute dot product* (SADP). V této metodě jsou využity normály n_1 a n_2 samplů a také jejich pozice p_1 a p_2. Hodnotu SADP pro dva samply lze spočítat dle vzorce:

$$sadp(n_1, n_2, r_{12}) = \frac{|n_1 \cdot r_{12}| + |n_2 \cdot r_{12}|}{length(r_{12}} \tag{2.1}$$

kde r_{12} je vektor mezi pozicemi samplů p_1 a p_2 (obrázek 2.7). Ze vzorce vychází, že pokud se dva samply nacházejí na stejném polygonu, tak jejich hodnota SADP se rovná nule. Hodnota SADP se zvětšuje s tím, jak se zvětšuje úhel mezi normálami a normalizovaným vektorem, který spojuje pozice samplů. V samotném algoritmu tedy vypočítáme hodnotu SADP pro všechny samply spolu se středem pixelu. Pokud hodnota SADP pro nějaký sample vyjde více než zadaný práh, je nutné provést antialiasing. V takovém případě musíme rozdělit samply do dvou skupin. V jedné budou samply, které budeme považovat za ty, které náleží stejnému polygonu jako střed pixelu a naopak v druhé budou samply, které náleží jiným polygonům nežli střed. Pro rozdělení do těchto dvou skupin použijeme algoritmus 1 popsaný v [12]:



Obrázek 2.7: SADP

Algoritmus 1: Míra podobnosti → dva clustery

Input: $(SADP, threshold)$

```
maxsim = 0
for i = 0 to 7 do
    if SADP[i] > maxsim then
        maxsim = SADP[i]
    end if
end for
for j = 0 to 7 do
    if SADP[i] < (maxsim/2 + threshold/2) then
        // Sample náleží stejnému polygonu jako střed pixelu.
    else
        // Sample náleží jinému polygonu nežli střed pixelu.
    end if
end for
```

2.3.2 Získání offsetu pro resampling

Cílem je získat offset pro každou možnou bitovou masku takový, aby při provádění bilineární interpolace, měly na výslednou barvu větší vliv ty sousední pixely, které jsou blíže té části pixelu, kterou pokrývá jiný polygon než střed pixelu. V praxi to znamená, že je nutno najít 255 souřadnic (pro 8 samplů), z kterých se bude provádět bilineární interpolaci. Tyto souřadnice jsou zapsány do lookup tabulky, kde jsou poté dle dané bitové masky vyhledatelné. Bitová maska zde slouží jako index.

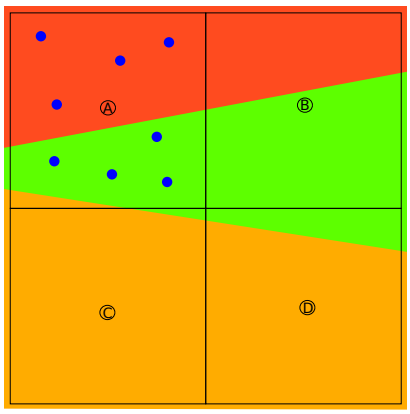
K nalezení těchto offsetů je využíváno podobných metod, jaké jsou používány v počítačovém vidění. Nejprve se připraví 10^6 případů, kdy pixel protíná jedna či dvě přímky. Samply, které se nacházejí ve stejném nově vniklém polygonu jako střed, jsou označeny jednou barvou, zbylé jsou označeny barvou jinou. Poté se pro každou bitovou masku vyberou takové případy, které tuto masku vytvoří. Pokud je takovou masku možné vytvořit pouze za použití jedné přímky, jsou pro tuto masku ponechány pouze tyto případy. Jinak jsou použity přímky dvě. Pokud ani jedna z těchto možností nestačí, tak se offset získá tak, že směr udává vektor od středu k nejvzdálenějšímu vzorku náležícího jinému polygonu a velikost je dána dle pokryté oblasti [12].

2.3.3 Metody k zvýšení přesnosti resampling offsetu

Při provádění resampling antialiasingu mohou pro některé situace nastat problémy, což má za následek vznik nežádoucích artefaktů. Problémy nastávají v případech, kdy polygony procházející pixelem jsou velmi malé, nebo vrcholy polygonů jsou velmi ostré. Tato situace je znázorněna na obrázku 2.8. V tomto případě se očekává, že výsledná barva pixelu A bude kombinací hlavně červené a zelené barvy. Offset pro tento pixel bude směrovat k pixelu C a D. Díky tomu bude výsledná barva kombinací hlavně červené a oranžové barvy. Tento problém má několik řešení.

Jedním z řešení je překontrolovat samply v clusteru, který nenáleží ke středu pixelu. Pokud je SADP hodnota mezi těmito samply a nejbližším venkovním samplem pro každý z nich větší, než SADP hodnota mezi těmito samply a středem pixelu, tak je tento sample přeřazen do clusteru příslušícího středu pixelu. Nevýhodou této metody je, že tento algoritmus budeme provádět i pro pixely, pro které nebude prováděn antialiasing [12].

V druhém řešení nejprve najdeme sample, který je v clusteru, který nepřísluší středu pixelu a zároveň je od středu pixelu nejvíce vzdálen. Poté vypočítáme SADP hodnotu mezi tímto samplem a středem pixelu. Dále vypočítáme SADP hodnotu mezi středem pixelu a středem sousedního pixelu, který je nejblíže k našemu samplu. Pokud je tato hodnota menší než první zmiňovaná SADP hodnota, tak resampling vektor zmenšíme na polovinu [12].



Obrázek 2.8: Problémy resampling antialiasingu

2.3.4 Optimalizace RSAA

Jedním ze způsobů optimalizace v resampling antialiasingu je neukládat bitové masky do postprocessing fáze, ale uchovávat SADP hodnoty pro všechny vzorky. Z těchto hodnot je poté možné v postprocessing fázi stanovit bitové masky pro přístup do tabulky offsetů. Dále je nutné SADP hodnoty uchovávat do postprocessing fáze kvůli ověření offsetu pro bilineární interpolaci.

Další možností optimalizace je vynechání těch pixelů, které jsou celé pokryté pouze jedním polygonem a není tedy nutné pro ně antialiasing vůbec řešit. V takovéto situaci by hodnota SADP pro všechny vzorky byla nula, tudíž by se výsledná barva pixelu získala podle středu pixelu [12].

2.4 Fast Aproximate Antialiasing

Fast aproximate antialiasing (FXAA) je morfologická metoda postprocessing antialiasingu popsaná v [6]. Na rozdíl od MSAA nebo RSAA, nezískává informace o obraze ze samotných 3D modelů, ale analýzou výsledného 2D obrazu. FXAA v tomto obraze detekuje hrany a poté vyhladí pixely podle toho, jakou mírou se nacházejí na hraně objektů. Díky tomu je FXAA méně výpočetně náročný než metody využívající vzorkování. Oproti tomu nevýhodou FXAA je to, že některé textury mohou být rozmazané, pokud je algoritmus vyhodnotí jako hrany. Další nevýhodou je to, že FXAA vyhlazuje například i text nebo různé ovládací prvky v obraze a ty se tím pádem nejeví natolik ostré. Řešením tohoto problému je generovat text a ovládací prvky do obrazu až po provedení vyhlazování pomocí FXAA.

FXAA nejprve analyzuje obraz a detekuje v něm hrany, které mají být vyhlazeny. Detekce hran se provádí pomocí porovnávání kontrastů. Pro každý pixel se najde nejmenší a největší hodnota jasu mezi tímto pixelem a jeho čtyřmi nejbližšími sousedy. Rozdíl těchto dvou hodnot se poté porovná s prahem, jehož hodnota se stanovuje podle maximální hodnoty jasu v těchto pixelech. Minimální hodnota tohoto prahu je nastavena globálně, protože není nutné provádět antialiasing ve velmi tmavých oblastech scény. Podle rozdílu kontrastu mezi pixelem a jeho okolím se také stanoví podíl přítomnosti pixelu na hraně a tato hodnota se později použije jako koeficient při vyhlazování. Získané hrany se následně rozdělí na horizontální a vertikální. Pro každý pixel na hraně se najde další pixel tak, aby kontrast mezi těmito dvěma pixely byl co největší. Hledání probíhá pouze v kolmém směru na hranu. Následně jsou nalezeny konce hran a to tak, že se hodnotí průměrný jas v těchto párech pixelů a pokud se tato hodnota výrazně změní, je nalezen konec hrany. Z těchto míst se poté stanoví offset pro hranu, který bude použit při vyhlazování [6].

2.5 Morphological Antialiasing

Morphological Antialiasing (MLAA) je jednou z dalších morfologických metod postprocessing antialiasingu. Je popsaný také panem Reshetovem v [11]. Stejně jako FXAA pracuje mimo vykreslovací řetězec a zpracovává až výsledný 2D obraz. V něm nejprve najde hrany objektů a ty následně vyhladí, přičemž rozpoznávání hran probíhá pouze podle porovnávání barev sousedních pixelů. Chování MLAA se liší pro černobílý obraz a pro barevný obraz. V praxi se jako černobílá varianta používá při vyhlazování stínů. Nevýhodou MLAA je, že pokud se ve scéně objevuje textura s velmi rozdílnými barvami, tak dojde k jejímu rozostření. Další nevýhodou jsou objekty o velikosti pixelu. U nich dojde k výraznému splynutí s okolím.

2.5.1 Popis chování MLAA pro černobílý obraz

V implementaci MLAA pro černobílý obraz algoritmus nejprve projde obraz řádek po řádku a následně sloupec po sloupci. Cílem této fáze je najít co nejdelší horizontální či vertikální hraniční linie. Tyto linie se hledají tím způsobem, že se vždy zpracovávají dva sousední řádky či sloupce zároveň a porovnává se barva dvou příslušných sousedních pixelů. Počítají se pouze ty dvojice, kde se barva liší. Těmto liniím se říká primární [11].

V druhém kroku jsou hledány sekundární linie. Ty jsou vždy kolmé na primární a nacházejí se pouze na jejich koncích. Ke každé primární linii tedy připadají dvě sekundární s výjimkou primárních linií procházejících krajem obrazu. Těm připadá sekundární linie jedna. Tímto způsobem se primární linie rozdělí do třech kategorií:

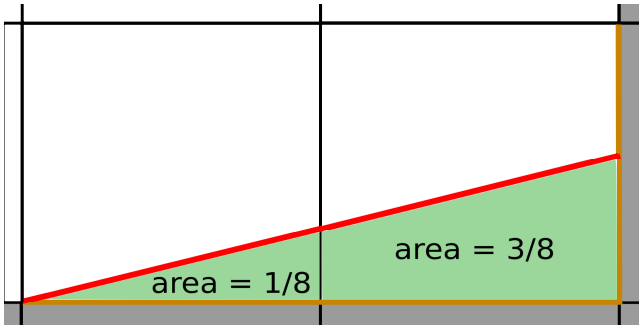
1. Linie ve tvaru Z. Tyto linie mají dvě sekundární, z nichž každá směřuje jiným směrem.
2. Linie ve tvaru U. Tyto linie mají dvě sekundární, z nichž obě směřují stejným směrem.
3. Linie ve tvaru L. Tyto linie mají pouze jednu sekundární linii. Mohou vznikat pouze na okrajích obrazu.

Linie ve tvaru Z a U mohou být transformovány na linie tvaru L tak, že se v polovině rozdělí. Tím tedy zbydou pouze útvary typu L. Ty jsou tvořeny primární linií o velikosti 0.5 pixelu až nekonečno, podle toho, jak se rozdělil útvar typu Z či U. Dále jsou tvořeny sekundární linií, která má vždy délku 1 [11].

Dalším krokem je nalezení koeficientů pro míchání barev pro každý pixel, který se nachází na hraně. Každý tento pixel přísluší nějakému L vzoru. U tohoto útvaru se propojí střed sekundární linie se vzdálenějším koncem linie primární. To rozdělí pixely uvnitř L vzoru na dvě části (obrázek 2.9). Následně je vypočítán poměr mezi částí příslušící k primární linii a zbytkem pixelu. Tato hodnota se poté použije jako koeficient pro míchání barev dle následujícího vzorce:

$$c_{new} = (1 - area) \cdot c_{old} + area \cdot c_{opposite} \tag{2.2}$$

kde c_{new} je výsledná barva pixelu, c_{old} je stará barva pixelu a $c_{opposite}$ je barva pixelu na opačné straně primární linie [11].



Obrázek 2.9: Výpočet koeficientu pro MLAA

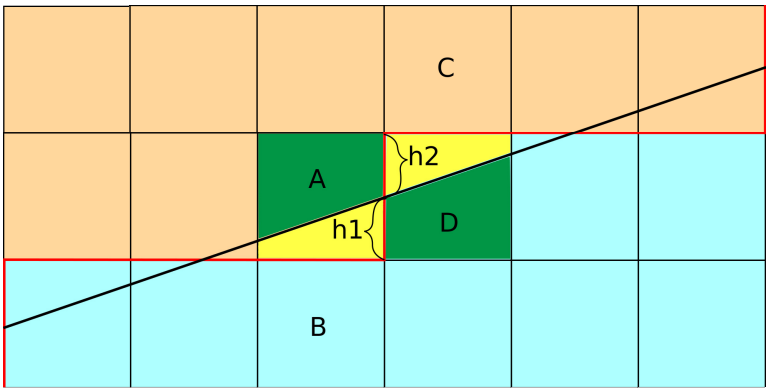
2.5.2 Popis chování MLAA pro barevný obraz

V implementaci MLAA pro barevný obraz se při hledání hran opět porovnává barva sousedících pixelů. Oproti implementaci pro černobílý obraz je nutné porovnávat rozdíl barev s určitým prahem, aby nedocházelo k vyhlazování mezi podobnými barvami. V MLAA se tento problém řeší tak, že při porovnávání jednotlivých barevných kanálů se berou v potaz pouze nejvýznamnější čtyři bity. Pokud jdou všechny tři barevné kanály vyhodnoceny jako rozdílné, tak barvy dvou příslušných pixelů jsou také brány jako rozdílné. Vyhledávání nejdelších primárních linií v horizontální a vertikální rovině probíhá stejně jako pro černobílý obraz [11].

Dalším krokem je nalezení příslušných vzorů pro vyhlazování. K tomu využijeme následující soustavu rovnic popisujících obrázek 2.10:

$$\begin{gathered}(h_1 - r)C_B + (1 - h_1 + r)C_A = (h_2 - r)C_C + (1 - h_2 + r)C_D \\ h_1 + h_2 = 1\end{gathered} \tag{2.3}$$

kde $r = h_2/(delkavzoru)$, neboli polovina rozdílu dvou základen žlutého lichoběžníku. Obsah žlutého lichoběžníku je poté $h_2 - r$, z čehož vyplývá, že obsah zeleného lichoběžníku je $1 - h_2 + r$. Hodnoty C_A, C_B, C_C a C_D jsou poté barvy příslušných pixelů. Příslušný vhodný vzor je poté nalezen tak, že se postupně pro každý možný z nich vypočítává soustava rovnic 2.3, dokud hodnoty h_1 a h_2 nevycházejí v intervalu $\langle 0, 1\rangle$. Je použit první vzor, vyhovující této podmínce. Pro výpočet hodnot h_1 a h_2 jsou sjednoceny jednotlivé barevné kanály do jedné hodnoty tak, že jsou sečteny. Po získání hodnot h_1 a h_2 je již možné vypočítat barvu pro pixely dle rovnice 2.2. Tato hodnota se již počítá pro každý barevný kanál zvlášť [11].



Obrázek 2.10: Určení vzoru pro MLAA

Kapitola 3

Návrh aplikace

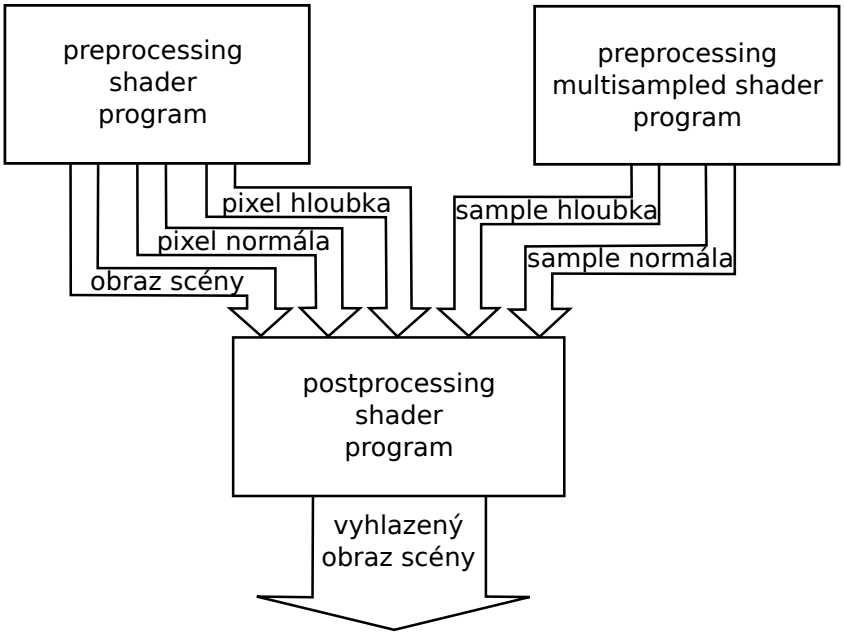
Cílem této práce je vytvořit demonstrační aplikaci, ve které bude možné porovnat obraz vyhlazený pomocí resampling antialiasingu nebo multisampling antialiasingu s původním nevyhlazeným obrazem. Cílem této aplikace nebude praktické použití, nýbrž pouze ukázka vlastností implementovaných metod. Aplikace by tedy měla umět zobrazit model a následně na něj aplikovat jednu ze zmíněných vyhlazovacích metod. Resampling antialiasing bude implementován explicitně jakožto cíl této práce, zatímco pro multisampling antialiasing bude použita implementace poskytovaná samotným OpenGL.

Implementace Resampling antialiasingu bude vycházet z návrhu pana Alexandera Reshetova [12]. Jádro samotného algorimu bude rozděleno do několika shader programů. V preprocessing fázi je nutné zjistit informace o scéně pro postprocessing. Model se musí transofmovat pomocí standartních matic a dále je nutné uložit data pro postprocessing. Tok dat mezi jednotlivými shader programy je znázorněn na obrázku 3.1. Mezi tyto data patří samotná vizualizace scény, tzn. to, co by se vykreslilo na monitor, pokud by nebyl použit antialiasing. Právě na těchto datech bude v postprocessing fázi prováděno vyhlazování. Dále je nutné přenášet world space[1] normály a to jak pro středy pixelů, tak pro jednotlivé samply v pixelech. Tyto normály jsou v postprocessing fázi nutné k určení hran jednotlivých trojúhelníků. K tomu je také zapotřebí world-space souřadnic středů pixelů i samplů. Tato informace se nicméně do postprocessing fáze nepřenáší přímo, ale world-space pozice je rekonstruována z hloubky, která se přenáší. O samotné rekostrukci bude podrobněji pojednáno v následující kapitole. Jako algoritmus pro detekci hran trojúhelníků bude použito již zmíněné *sum of the absolute dot product* (SADP). K samotné detekci hran by bylo možné použít i jiné algorimy, které využívají pouze například skalární součin normál nebo porovnávání hloubek. SADP bylo vybráno proto, že je použito v [12] a také proto že má oproti jiným metodám výhody zmíněné v kapitole 2.3.1.

Pro získání offsetů pro bilineární interpolaci budou použita stejná data, jaká použil pan Reshetov v [12]. Tato data byla vygenerování pro 8x MSAA v DirectX, nicméně v OpenGL je možné tato data také použít, protože samply jsou v pixelu rozloženy podle stejného vzoru jako v DirectX. Pozice těchto samplů jsou podrobně popsány v [7].

V postprocessing fázi budou také implementovány obě dvě metody pro ověřování resampling offsetu dle [12]. Tyto metody nicméně nejsou pro fungování resampling antialiasingu nutné, pouze zvyšují jeho kvalitu a to hlavně u scén s mnoha malými polygony. Aplikace tedy při RSAA může běžet v jednom ze tří módů. V prvním se offset neověřuje, ve zbylých dvou se offset ověřuje pomocí zmíněných metod.

[1] Jedná se o reprezentaci scény, kde každý objekt má svoji pozici a orientaci.



Obrázek 3.1: Shader programy pro RSAA

3.1 Použité nástroje a knihovny

Pro tvorbu této aplikace bude využit programovací jazyk C++ společně s OpenGL Shading Language (GLSL), což je jazyk určený na programování shaderů grafických karet. Jak již bylo zmíněno výše, pro komunikaci s grafickou kartou použijeme rozhraní OpenGL, které se běžně používá pro tvorbu aplikací s počítačovou grafikou. Jako nadstavba nad OpenGL bude využit GPUEngine, což je nástroj vyvíjený na Ústavu počítačové grafiky a multimédií FIT VUT. Tento nástroj usnadňuje práci s OpenGL a se soubory obsahujícími kód v GLSL. Dále bude použit framework Qt, který slouží ke tvorbě oken, uživatelského rozhraní, atd. Tento framework má i přímou podporu pro OpenGL, jak bude zmíněno v další kapitole. Další využívanou knihovnou je OpenGL Mathematics (GLM). Tato knihovna poskytuje podporu pro různé matematické prostředky využívané v počítačové grafice jako jsou matice, kvaterniony apod. K náčítání textur byla použita knihovna lodepng, která umožňuje načítat obrázky ve formátu png. Pro načítání samotných objektů byly využity kódy z tutoriálů k OpenGL[2].

[2] http://www.opengl-tutorial.org/

Kapitola 4

Implementace

K implementaci této demonstrační aplikace je použit programovací jazyk C++ společně s GLSL. Cílovou platformou je systém Windows. Díky C++ je aplikace implementována objektově, což souvisí i s použitým toolkitem Qt. Tento toolkit nám usnadňuje komunikaci s aplikací a také umožňuje jednoduché vytváření a správu oken.

Aplikace není, co se objektově orientovaného návrhu týče, vůbec složitá. Skládá se z několika tříd, z nichž první je třída `RsaaOpenGL`. Toto je základní třída aplikace, která obsahuje ukazatele na všechny zbylé třídy aplikace. Obsahuje dvě metody, z nichž první je metoda `init`, která se stará o vytvoření a inicializaci objektů nutných pro běh aplikace. Dále obsahuje metodu `run`, která započne běh aplikace.

S těmito dvěmi metodami souvisejí metody třídy `RsaaOpenGLWidget`. První z nich je `initializeGL` a jak již název napovídá, slouží k inicializaci vykreslovacího prostředí před samotným vykreslováním. `PaintGL` poté funguje jako vykreslovací smyčka, kdy je tato funkce volána před každým vykreslením snímku. Toto volání probíhá interně v Qt, což ovšem přináší jednu nevýhodu. Jedná se o to, že tato metoda není volána v co nejmenším intervalu, ale je volána pouze jako reakce na nějaký uživatelský vstup a nebo změnu na obrazovce. Z toho vyplývá, že pokud uživatel nebude s aplikací nijak interagovat, tak tato funkce nebude volána, tudíž ani výsledný obraz nebude překreslován. V této aplikaci je nicméně tato metoda volána i explicitně a to díky další metodě třídy `QOpenGLWidget` a tou je `update`. Díky tomu může aplikace běžět až do maximální obnovovací frekvence monitoru.

Nejdůležitější třídou v aplikace je `Renderer`. Je zde implementováno samotné vykreslování, práce se shadery a ovládání aplikace.

4.1 Implementace ovládání

Jak již bylo zmíněno výše, k implementaci přepínání mezi jednotlivými vyhlazovacími metodami slouží tlačítka `QPushButton`. Dále je ovšem nutné implementovat ovládání pohybu modelu. K tomu jsme využili opět metod třídy `QOpenGLWidget`, díky nimž dokážeme zpracovávat různé zprávy z myši či klávesnice. V těchto metodách upravujeme různé proměnné objektu `Renderer`, ve kterém je poté implementována samotná reakce aplikace na tyto zprávy.

Vzhledem k tomu, že v aplikaci pracujeme pouze s jedním modelem, není nutné implementovat pohyb kamery, ale stačí implementovat pouze otáčení objektu. Otáčení a přibližování objektu je implementováno ve třídě `Renderer` v její metodě `rotate_view_zoom`. Přibližování je zde implementováno pomocí funkce `lookAt` z knihovny GLM. Tato funkce

nám s příslušným parametrem přiblížení vrátí view matici, kterou poté můžeme použít. Co se týče otáčení objektu, tak tam je situace o něco složitější. Nejjednodušší by bylo využít opět matic, nicméně zde se setkáváme s problémem zvaným Gimbal Lock, který je blíže popsaný v [3]. V naší aplikaci tento jev způsobuje to, že pokud objekt nejprve otočíme podle osy y, tak se otáčí přirozeně. Pokud ale následně objekt otočíme podle osy x, tak zjišťujeme, že se objekt již nadále neotáčí podle osy y přirozeně, ale otáčí se spíše kolem osy z. Otáčení objektu je tudíž značně neintuitivní.

Řešením tohoto problému jsou kvaterniony. V rendereru tedy existuje kvaternion reprezentující natočení objektu v prostoru a pokud chceme objektem otočit, stačí tento kvaternion vynásobit jiným kvaternionem, který nam vrátí funkce `angleAxis` z balíku GLM. Tato funkce sestaví příslušný kvaternion z úhlu o který chceme objekt otočit a dále z osy, podle které chceme objekt otáčet.

4.2 Inicializace

Aby bylo možné renderer správně používat, je nutné nejdříve provést jeho inicializaci. To se provede spuštěním metody `init`. Zde je neprve nutné vytvořit všechny potřebné Vertex Buffer Objekty[1](VBO). V případě této aplikace se tedy jedná o vrcholy, barvu a normály jednotlivých trojúhelníků. Normály je ovšem nejprve nutné pro každý trojúhelník spočítat. Tento výpočet probíhá dle algoritmu 2. Využívá se zde vektorového součinu dvou vektorů, jehož výsledkem je vždy vektor kolmý na tyto dva vektory. V našem algoritmu tedy provádíme vektorový součin dvou stran trojúhelníku a dostaneme vektor, který je na tento trojúhelník kolmý.

Algoritmus 2: Výpočet normál

Input: $(p1, p2, p3)$
Output: $(normal)$

1: $v_1 = p_2 - p_1$
2: $v_2 = p_3 - p_1$
3: $normal = cross(v_1, v_2)$
4: $len = sqrt(normal.x^2 + normal.y^2 + normal.z^2)$
5: $normal = normal/len$

Dalším krokem je načíst všechny potřebné shader programy a provést jejich inicializaci. V této aplikaci využíváme celkem čtyři shader programy, přičemž tři již byly zmíněny v kapitole 3 v rámci návrhu implementace RSAA. Čtvrtý shader program slouží pro multisampling antialiasing, kde se scéna renderuje do multisamplovaného Framebuffer Objektu[2](FBO). Toto slouží pro porovnání s RSAA.

4.2.1 Inicializace nemultisamplovaného shader programu

Jak je vidět na obrázku 3.1, před samotným postprocessingem využíváme dva shader programy. Je to z toho důvodu, že framebuffer musí být buď normální a nebo multisamplovaný a není tedy možné zaráz do jednoho framebuffer renderovat data ze středů pixelů i ze vzorků.

[1] Jedná se o buffer objekt, přes který je možné nahrávat vertexy do grafické karty.
[2] Uživatelsky definovaný framebuffer, který funguje jako cíl pro renderování a poté je překreslen na obrazovku

V nemultisamplovaném shader programu si tedy připravíme všechna data ze středů pixelů pro postprocessing. V praxi to znamená, že výstupem tohoto shader programu budou tři textury, z nichž jedna bude obsahovat barvu, druhá world-space normálu a třetí hloubku. Pro texturu s barvou jsou zde vytvořeny dva samplery, které určují, jaká výsledná barva se z textury získá na základě zadaných souřadnic. Do postprocessing shader programu tudíž tato textura vstupuje dvakrát, přičemž poprvé je vzorkována jako `GL_NEAREST`, což zajistí to, že barva navzorkovaná z textury, se bude rovnat barvě nejbližšího texelu. Podruhé je textura vzorkována jako `GL_LINEAR`, tedy při vzorkování se barva získá bilineární interpolací čtyř nejbližších texelů. Co se textury s normálami týká, zde byl použit interní formát `GL_RGB10_A2`. V tomto formátu má každý barevný kanál 10 bitů a alfa složka má bity dva. To nám umožní ukládat normály s dostatečnou přesností. Nevýhodou tohoto formátu je to, že neumožňuje ukládat záporná čísla, normály je tedy před uložením nutné transformovat. Pro uložení hloubky, byl použit interní formát `GL_DEPTH_COMPONENT32` a to z toho důvodu, že rekonstruovaná world-space pozice musí být velmi přesná, protože world-space vzdálenosti mezi středem pixelu a vzorkem nebudou příliš velké.

4.2.2 Inicializace multisamplovaného shader programu

V tomto shader programu je situace o něco jednodušší, protože zde nemusíme řešit ukládání barvy, ale výstupem tohoto programu budou pouze textury s hloubkou a s normálami. Vzhledem k tomu, že zde chceme ukládat normály a hloubku u každého samplu zvlášť, tak obě dvě textury musejí být multisamplové, jinak mají parametry stejné, jako textury v nemultisamplovaném shader programu. Je také nutné nastavit, aby obě dvě textury měly fixní pozice samplů, protože celý algoritmus RSAA stojí na tom, že známe jejich pozice.

Kromě textur je zde ještě nutné připravit pole, které obsahuje pozice samplů v pixelu oproti středu. Tuto informaci využijeme v postprocessing shader programu, při rekostrukci world-space pozice. Tyto data nicméně nezjišťujeme v shader programu, ale pomocí OpenGL funkce `glGetMultisamplefv`. Tato funkce nám vrátí pozici samplu v pixelu, nicméně střed pixelu zde nemá souřadnice (0, 0), jak by se nám v postprocessingu hodilo, ale má souřadnice (0.5, 0.5). Toto napravíme jednoduchým odečtením 0.5 od obou hodnot souřadnic. Tyto data musejí být zjišťována na tomto místě v programu, protože funkce `glGetMultisamplefv` potřebuje ke své správné činnosti, aby byl nabindovaný multisamplovaný framebuffer.

4.2.3 Inicializace postprocessing shader programu

V tomto programu budeme zpracovávat data z předešlých dvou shader programů a výstupem by měl být vyhlazený obraz. Vstupem je pouze čtyřúhelník, který zaplní celý obraz, protože každý vertex shader musí mít na svém vstupu nějaké vertexy. Dále musíme programu zpřístupnit všechny vytvořené textury, pozice samplů v pixelu nebo i informaci o tom, jaký ověřovací mód u RSAA má být použit. Tyto data budou do grafické karty nahrána pomocí uniformních proměnných.

4.3 Preprocessing shader programy

Jak již bylo několikrát řečeno, v této aplikaci existují dva preprocessing shader programy. V programu, který není multisamplovaný, nejprve ve vertex shader provedeme standartní

maticové operace nad objektem. Dále předáme barvu a normálu do fragment shaderu. Důležité je vynásobit normálu normálovou maticí, protože tím, že otáčíme objektem a nikoliv kamerou, tak se bude měnit i hodnota normály. Ve fragment shaderu již pouze zapíšeme barvu a normálu do textury. Normálu ještě předtím musíme převést do intervalu $\langle 0, 1 \rangle$, protože formát `GL_RGB10_A2` nepodporuje záporná čísla. To provedeme dle následující rovnice:

$$normal = (normal + 1)/2 \tag{4.1}$$

O zapisování hloubky se nemusíme vůbec starat, protože se zapíše implicitně do té textury, kterou máme navázanou na `GL_DEPTH_ATTACHMENT`, což v našem případě je textura pro hloubku.

V multisamplovaném shader programu probíhá vše velmi podobně. Jediným rozdílem je, že nezapisujeme nikam barvu. Nabízí se otázka, proč rekonstruovat v postprocessing shader programu world-space pozici z hloubky, když preprocessing fázi ji máme přístupnou a tudíž by bylo možné ji zapsat do textury a v postprocessingu přímo použít. Problémem je samotné fungování multisamplingu v OpenGL. To totiž funguje na tom principu, že jediné, co se skutečně vzorkuje na každém samplu, je hloubka. Barva, tudíž v našem případě i world-space pozice, se vzorkuje jednou na středu a tato hodnota se poté přiřadí všech samplům stejného trojúhelníku. Tudíž pokud by jsme zapisovali world-space pozici do textury, tak by každý sample obsahoval stejné hodnoty. Toto chování se dá změnit a to sice povolením `GL_SAMPLE_SHADING` v kombinaci s `glMinSampleShading(1.0)`. Toto řešení je použito v [12], nicméně my jsme se rozhodli pozici rekostruovat z hloubky. Důvody jsou uvedeny v kapitole 4.4.4. Pokud porovnáme výsledné hodnoty z obou dvou metod tak zjistíme, že se liší v řádech statisíců, což je pro naši aplikaci dostačující.

4.4 Postprocessing shader program

Postprocessing shader program používá dříve získaná data k tomu, aby vyhladil výsledný obraz. Nejprve je nutné z textur načít vše pořebné, tzn. normály pro střed a samply a hloubku pro střed a samply. S normálami nemusíme provádět žádné operace, pouze je musíme převést zpět do intervalu $\langle -1, 1 \rangle$, podobně jako jsme je převáděli v kapitole 4.3. Pokud jsou všechny normály samplů rovny normále pozadí, nebudeme tento pixel dále vůbec řešit a přidělíme mu jeho originální barvu. World-space pozice získáme tak, že je rekonstrujeme z hloubky. Pokud se hloubka pixelu nebo samplu rovná 1, znamená to, že se nachází na pozadí a tudíž mu nastavíme souřadnice pozice na 0.

4.4.1 Rekonstrukce world-space pozice

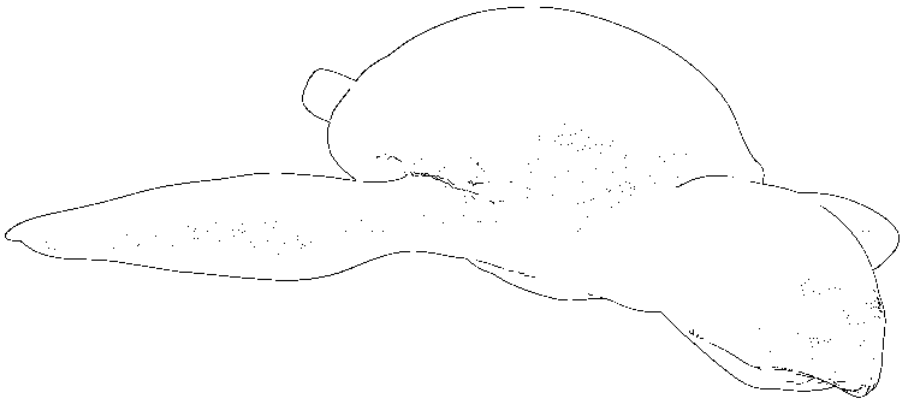
Algoritmus rekostrukce world-space pozice vychází z [10]. V tomto algoritmu nejprve použijeme pozici středu momentálně počítaného pixelu a převedeme ji do Normalized Device Coordinates[3] (NDC). Stejný převod použijeme i pro hloubku, kterou jsme navzorkovali z textury. Pro převod do world-space poté stačí tyto souřadnice vynásobit inverzní projekční a view maticí a dále vydělit w složkou. Tímto získáme pozici bodu ve world-space.

Pokud počítáme world-space pozici samplu v pixelu, tak je nutné nejprve ke středu pixelu přičíst hodnotu, kterou jsme získali díky `glGetMultisamplefv` podělenou velikostí viewportu.

[3] Jedná se o normalizovaný souřadný systém, který OpenGL pozdeji namapuje na konkrétní viewport.

4.4.2 Implementace RSAA

Nyní tedy máme všechny potřebné informace pro implementaci Resampling antialiasingu. Nejprve musíme spočítat míru podobnosti mezi středem pixelu a všemi jeho samply a to s pomocí SADP. SADP se ovšem dá počítat, pouze pokud střed pixelu i příslušný sample náleží nějakému polygonu. Pokud tedy střed pixelu nepřísluší žádnému polygonu, tak všechny samply, které nějakým polygonům náleží, jsou rozdílné. Stejně tak pokud sample nenáleží žádnému polygonu a střed náleží polygonu, tak jsou rozdílné. Pokud tedy sample i střed náleží nějakým polygonům, můžeme přistoupit k výpočtu SADP. To provádíme stejně, jak je popsáno v kapitole 2.3.1. Jediným rozdílem je to, že místo aby jsme pro získání normalizovaného vektoru výsledek podělili jeho délkou, tak využijeme vestavěné funkce glsl `normalize`, která by měla být rychlejší než dělení. Výsledek detekce hran pomocí SADP můžeme vidět na obrázku 4.1. Můžeme si všimnout, že k vyhlazení jsou vybrány pouze ty pixely, kde úhel mezi polygony je velký, tudíž i aliasing by tam byl lehce viditelný. Naopak na pixelech, kde je tento úhel malý (krunýř želvy), k vyhlazování docházet nebude.



Obrázek 4.1: Ukázka detekce hran pomocí SADP. Černé pixely budou vyhlazovány.

Pro rozdělení samplu do clusterů a tvorbu masky poté použijeme algoritmus popsaný v 2.3.1. Pro získání pozice pro bilineární interpolaci poté musíme přičíst ke středu pixelu offset, který získáme z tabulky offsetu díky masce. Tento offset musíme ještě převést do intervalu $\langle 0, 1 \rangle$ a to tak, že ho jednoduše podělíme velikostí viewportu. Poté již můžeme provést bilineární interpolaci a dostávámi konečnou barvu pixelu.

4.4.3 Implementace ověřovacích metod pro resampling offset

Tyto metody jsou implementovány podle kapitoly 2.3.3. Pokud používáme první ověřovací algoritmus, tak poté co získáme SADP hodnotu pro sample, spočítáme ještě jednu SADP hodnotu a to mezi tímto samplem a středem nejbližšího pixelu k tomuto samplu. Pokud je sample v rozdílném clusteru, musí tedy být „podobnější“ se středem nejbližšího pixelu, než se středem vlastního pixelu.

Pro druhou ověřovací metodu si během rozdělování samplů do clusterů najdeme takový sample, který je rozdílný a současně je nejdále od středu pixelu. Poté spočítáme hodnotu SADP mezi středem aktuálního pixelu a středem pixelu, který je nejblíže našemu nejvzdále-

nějšímu rozdílnému samplu. Pokud je tato hodnota dvakrát větší, než SADP tohoto samplu, tak resampling offset změnšíme na polovinu.

4.4.4 Rozdíly oproti originální implementaci

V originální implementaci RSAA v [12] jsou nejprve do multisampovaného shader programu dodány world-space pozice a normály. Dále se přímo v tomto shader programu vypočítá SADP pro každý sample a tato hodnota je poté zapsána do výstupní textury shaderu. Nevýhodou tohoto řešení je to, že aby jsme mohli do multisamplované textury zapisovat pro každý sample jiná data, musíme povolit tzv. sample shading. Pokud je tento režim povolen, tak fragment shader není spouštěn jednou za pixel, ale skutečně pro každý sample zvlášť. To pochopitelně přináší zpomalení. Naším řešením je tedy z multisamplovaného shader programu brát pouze normály a hloubky. Jak již bylo několikrát řečeno, hloubka je vzorkována pro každý sample samotným OpenGL. Normály vzorkovány pro každý sample zvlášť být nemusejí, protože všechny sample náležící jednomu trojúhelníku budou mít stejnou world-space normálu. SADP hodnoty poté stanovíme až v postprocessing shader programu.

Kapitola 5

Výsledky a měření

K měření byly použity celkem dvě počítačové sestavy, přičemž všechny testy se uskutečnily na operačním systému Microsoft Windows 10 s aktuálními ovladači grafických karet dle jejich výrobce. Grafické karty a taktéž procesory v sestavách byly ponechány na svých továrních taktech.

První sestava

- Intel Core i5-4440 @3.10 GHz
- 8.0 GB RAM
- NVIDIA GTX 770 @1215 MHz verze ovladače 365.19

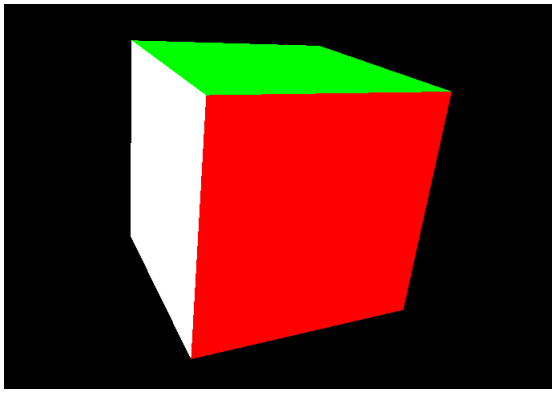
Druhá sestava

- Intel Core i5-4590 @3.30 GHz
- 8.00 GB RAM
- AMD Radeon R9 380 @980 MHz verze ovladače 16.15.2211

5.1 Kvalita obrazu

Měření kvality obrazu není jednoduchá záležitost a to z toho důvodu, že vnímání obrazové kvality je značně subjektivní. Z tohoto důvodu je velmi těžké hodnotit kvalitu obrazu, protože stroje nemusejí rozpoznat artefakty, které ale lidské oko vidí a působí rušivým dojmem. Naopak části obrazu, které jsou strojem označeny jako nekvalitní, nemusejí lidem vůbec vadit, protože na dané chyby nejsou citliví.

Kvalita výsledného obrazu naší aplikace byla měřena na jednoduché scéně s kostkou, jejíž stěny byly obarveny výraznými barvami, takže nevyhlazené přechody byly velmi výrazné. Ukázku scény můžeme vidět na obrázku 5.1.



Obrázek 5.1: Ukázka scény pro testování kvality.

Pro testování výsledné kvality obrazu naší aplikace byla vybrána metoda peak-signal-to-noise ratio (PSNR). Důvodem pro výběr této metodiky je to, že je vhodná na porovnávání výsledků podobných metod aplikovaných na obraz. Dalším důvodem je to, že byla použita v [12] a tudíž budeme moci porovnat výsledky implemetace v DirectX a OpenGL.

Tato metoda funguje na principu porovnávání poměrů maximální možné hodnoty barvy a rozptylu barvy v obraze. Při měření kvality antialiasingové metody postupujeme tak, že nejprve si zjistíme hodnotu PSNR mezi nevyhlazeným obrazem a referenčním vyhlazeným obrazem. V našem případě byl jako referenční antialiasing brán 32x supersampling, jehož výsledný obraz je považován za „správný“. Poté zjistíme hodnotu PSNR mezi našim antialiasingem a referenčním obrazem. Rozdíl mezi PSNR nevyhlazeného obrazu a PSNR našeho antialisingu je potom hodnota v decibelech, která vyjadřuje, o kolik náš antialiasing zvýšil kvalitu výsledného obrazu. Výpočty PSNR probíhaly pomocí nástroje Wolfram Mathematica.

Měření	**PSNR** [dB]						
	noAA	MSAA 4x	zisk	RSAA	zisk	MSAA 8x	zisk
1.	35.9003	43.2284	7.3281	44.7699	8.8696	45.0482	9.1479
2.	35.7285	41.9650	6.2365	43.5289	7.8004	44.4856	8.7571
3.	35.8880	42.6472	6.7592	44.9333	9.0453	45.5350	9.6470
4.	42.3268	47.8975	5.5707	48.1680	5.8412	49.8715	7.5447
5.	41.9490	47.9829	6.0339	48.0462	6.0972	50.3676	8.4186
6.	42.0447	47.3652	5.3205	48.2452	6.2005	50.0312	7.9865

Tabulka 5.1: Výsledky měření kvality obrazu metodou PSNR. Na jednotlivých řádcích vydíme opakované pokusy měření. Z tabulky je možno vidět, že RSAA se kvalitou řadí mezi MSAA 4x a MSAA 8x.

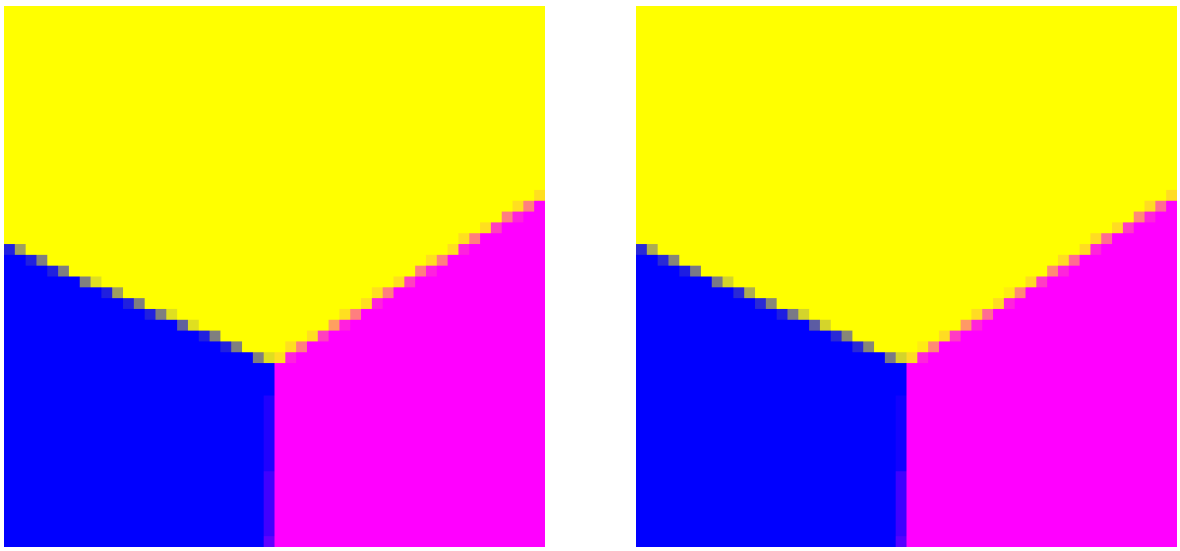
Výsledky metody PSNR pro naší aplikaci můžeme vidět v tabulce 5.1. Proběhlo celkem šest měření, přičemž model byl vždy v jiné vzdálenosti od kamery a jinak natočen. Při měřeních 1-3 byl model blízko ke kameře, pro měření 4-6 byla kamera oddálena. Z výsledků můžeme usuzovat, že námi implementovaná metoda RSAA je co se kvality týče, někde na pomezí mezi MSAA 4x a MSAA 8x, což se shoduje s výsledky v [12]. Z tabulky je také patrný rozdíl ve výsledcích pro měření s přiblíženým a oddáleným modelem. RSAA zde dosahuje viditelně horších výsledků, což je dáno tím, že při oddáleném modelu pracujeme s menšími polygony, tudíž se zde na jednom pixelu protínají více než dva polygony. Toto

patří mezi nevýhody RSAA popsaných v 2.3.3. V [12] se uvádí, že zlepšení obrazu po použití RSAA dosahuje až 10 dB. V naší aplikaci dosahujeme nejvýše ke zlepšení okolo 9 dB.

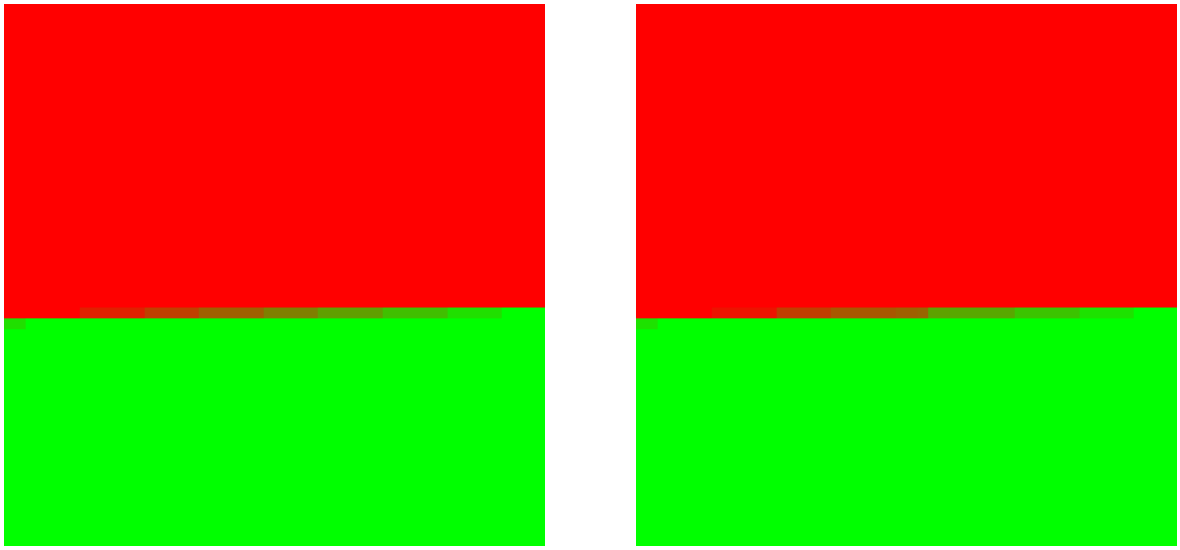
Pro kontrolu byla porovnána kvalita originální implementace z [12] s naší implementací. Závěr je takový, že výsledný obraz z těchto dvou metod je naprosto totožný.

Na obrázcích 5.2, 5.3 a 5.4 můžeme vidět přímé porovnání obrazu vyhlazeného MSAA 8x a RSAA 8x. Z obrázků je patrné, že obě dvě vyhlazovací metody zvládají vyhlazování diagonálních linií velmi podobně. Při přiblížení je možné na těchto liniích vidět určité odlišnosti, ale většinou se jedná pouze o velmi drobné odchylky. Větší rozdíly už jsou patrné na liniích, které jsou téměř horizontální či vertikální jako například na obrázku 5.3. Zde můžeme vidět, že zatímco u MSAA barva přechází ze zelené na červenou plynule, u RSAA tento přechod tak plynulý není. Tento jev je částečně pozorovatelný i u nepřiblíženého obrazu, nicméně vnímatelný je pouze při přímém porovnávání mezi MSAA a RSAA.

Na obrázku 5.5 můžeme vidět porovnání RSAA s kontrolou a bez kontroly resampling offsetu. Z obrázků je zřejmé, že bez použití kontroly se v místě, kde se setkávají bílý a žlutý polygon s černým pozadím, tvoří chyba. Pixely v tomto místě jsou mnohem více ovlivněny černou barvou pozadí, než by měly být. Naopak na obrázku s kontrolou resampling offsetu můžeme vidět, že na tyto pixely má mnohem větší vliv bílý polygon, což je správný výsledek.

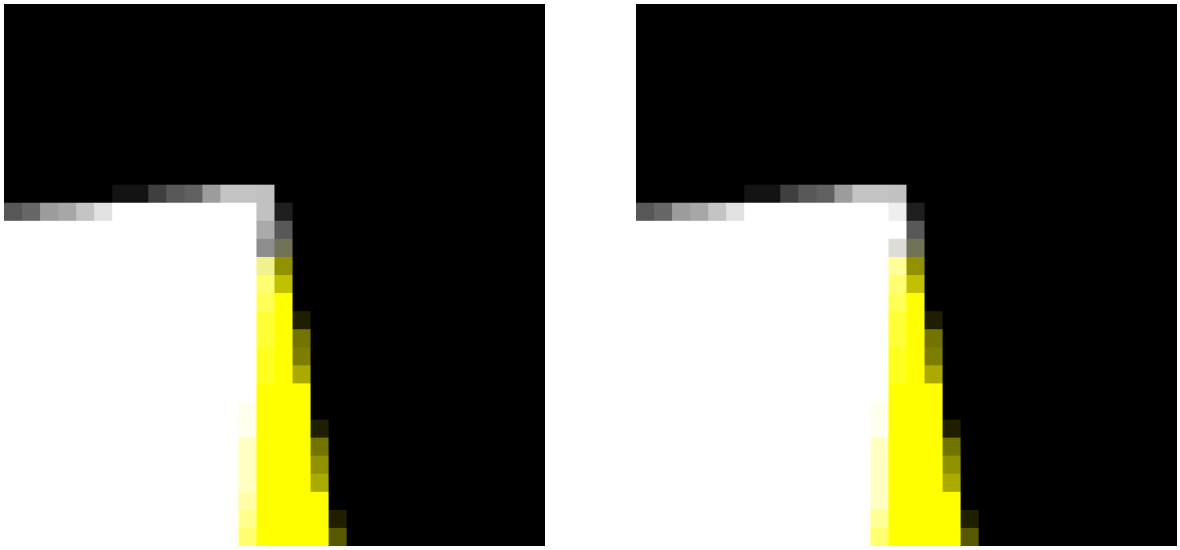


Obrázek 5.2: První porovnání MSAA (vlevo) a RSAA (vpravo). Můžeme si všimnout, že na diagonálních liniích dosahuje MSAA a RSAA velmi podobných výsledků.



Obrázek 5.3: Druhé porovnání MSAA (vlevo) a RSAA (vpravo). Na horizontálních nebo vertikálních liniích dosahuje RSAA o něco horších výsledků. Zde si můžeme všimnout, že barevný přechod není u RSAA natolik plynulý jako u MSAA.

Obrázek 5.4: Třetí porovnání MSAA (vlevo) a RSAA (vpravo). Opět jsou patrné rozdíly na vertikálních nebo horizontálních liniích.



Obrázek 5.5: Porovnání mezi RSAA bez kontroly resampling offsetu (vlevo) a RSAA s kontrolou resampling offsetu (vpravo). Na levém obrázku můžeme vidět, že v místech kde se setkává bílý a žlutý polygon s pozadím, jsou některé pixely ovlivněny příliš černou barvou pozadí, což je špatně. Naopak na pravém obrázku s kontrolou resampling offsetu má na tyto pixely větší vliv bílý polygon, což je správně.

5.2 Porovnání rychlosti

Měření rychlosti probíhala na modelu „turtle“ s 261 868 polygony, „giraffe“ s 263 292 polygony a „robot“ s 261 892 polygony. Program jsme nejprve spustili s danou vyhlazovací metodou a následně jsme počkali asi 10 sekund na ustálení stavu. Poté jsme provedli 5 měření v intervalu asi 5 sekund. Tyto výsledky byly poté zprůměrovány. Rychlost vykreslování snímku byla měřena pomocí Query Object.

Model	**Rychlost 1. sestava** [ms]		
	noAA	MSAA	RSAA
turtle	0.376592	0.451672	0.924372
giraffe	0.467421	0.500218	0.923651
robot	0.454428	0.484215	0.936652

Tabulka 5.2: Výsledky měření rychlosti první sestava

Model	**Rychlost 2. sestava** [ms]		
	noAA	MSAA	RSAA
turtle	0.505333	0.612743	12.21955
giraffe	0.491971	0.589259	12.163852
robot	0.521037	0.710963	12.139852

Tabulka 5.3: Výsledky měření rychlosti druhá sestava

V tabulkách 5.2 a 5.3 můžeme vidět výsledky měření na modelech. Nejmarkantnější je zde rozdíl v RSAA mezi oběma sestavami a tedy mezi grafickou kartou od společnosti

Nvidia a grafickou kartou od společnosti AMD. Z tabulek je zřejmé, že u druhé sestavy s kartou AMD probíhalo vykreslení snímku s RSAA několikanásobně déle, než s kartou Nvidia. Důvod tohoto jevu není autorovi této práce znám. Nejspíše se bude jednat o rozdíl v optimalizaci mezi kartami těchto dvou společností. Pokud se ovšem zaměříme pouze na výsledky z měření jedna, tak můžeme vidět, že RSAA neprodlužuje vykreslování snímku o více jak 0,7 ms. To bylo již avizováno v [12] a my se našimi výsledky přibližujeme výsledkům v tomto článku.

Kapitola 6

Závěr

Tato práce se zabývá vyhlazováním počítačové grafiky pomocí Resampling Antialiasingu. Je zde obsažena teoretická část popisující několik technik antialiasingu a dále studie samotného Resampling Antialiasingu. Na základě studia této metody byl poté vytvořen návrh aplikace, která má za cíl demontrovat Resampling Antialiasing oproti jiné vyhlazovací metodě.

K implementaci byl použit jazyk C++ společně s OpenGL. Pro programování grafických karet bylo využito jazyka glsl, ve kterém vzniklo v rámci aplikace několik shader programů. V těchto programech se odehrává samotné jádro vyhlazovací metody.

Po vytvoření aplikace byly změřeny výsledky jednotlivých implementovaných vyhlazovacích technik. K hodnocení kvality výsledného obrazu bylo využito metody peak-signal-to-noise ratio. Ač tato metoda nezaručuje, že obraz bude skutečně na člověka působit dobrým dojmem, umožňuje nám objektivně hodnotit kvalitu obrazu. Z měření touto metodou vyplynulo, že námi implementovaný Resampling Antialiasing splňuje do značné míry výstupní kvalitu obrazu, jakou dosahovalo originální řešení. Zlepšení díky našemu antialiasingu dosahuje až 9 dB.

Obrázek 6.1: Finální podoba objektu bez vyhlazování a s vyhlazováním

U měření rychlosti je situace složitější. Samotný postup měření byl sice oproti měření kvality jednodušší, za to težší je interpretace výsledků. Náš antialiasing totiž dosahuje dosti odlišných rychlostí pro karty od společnosti Nvidia a pro karty od společnosti AMD. Máme

za to, že důvod pro tento rozdíl bude v architektuře samotných karet, konkrétní důvod této odchylky se nicméně autorovi této práce nepodařilo zjistit.

Co se týká optimalizace, tak jistě jsou v naší aplikaci místa pro zlepšení. Jedním takovým je algoritmus pro rekostrukci world-space pozice z hloubky. V naší aplikaci je tento algoritmus velmi jednoduchý, což ovšem přináší problémy v podobě častého dělení a také v podobě častého násobení maticemi.

Literatura

[1] Agoston, M. K.: *Computer graphics and geometric modeling: implementation and algorithms*. London: Springer-Verlag, 2005, ISBN 1852338180.

[2] Akeley, K.: Reality Engine graphics. In *SIGGRAPH '93 Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, 1993, ISBN 0-89791-601-8, s. 109–116, doi:10.1145/166117.166131.
URL http://dl.acm.org/citation.cfm?id=166117.166131

[3] Barbic, J.: Quaternions and Rotations. [online].[cit. 2016-05-07].
URL http://run.usc.edu/cs520-s12/quaternions/quaternions-cs520.pdf

[4] Beets, K.: Super-sampling Anti-aliasing Analyzed - Page 3. Consumer Graphics, [online]. 2000-04-28 [cit. 2016-01-22].
URL https://www.beyond3d.com/content/articles/37/3

[5] Leler, W. J.: Human vision, anti-aliasing, and the cheap 4000 line display. In *SIGGRAPH '80 Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, ročník 14, 07 1980, ISBN 0-89791-021-4, s. 308–313, doi:10.1145/800250.807509.

[6] Lottes, T.: FXAA. NVIDIA white paper, [online]. 2009 [cit. 2016-01-22].
URL http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

[7] Microsoft: *D3D11_STANDARD_MULTISAMPLE_QUALITY_LEVELS enumeration*. [online].[cit. 2016-05-07].
URL https://msdn.microsoft.com/en-us/library/windows/desktop/ff476218(v=vs.85).aspx

[8] Owen, S.: Antialiasing strategies. [online]. 1999-10-04 [cit. 2016-05-08].
URL https://web.cs.wpi.edu/~matt/courses/cs563/talks/antialiasing/methods.html

[9] Pettineo, M.: Deferred MSAA. WordPress blog, [online]. 2010 [cit. 2016-01-22].
URL https://mynameismjp.wordpress.com/2010/08/16/deferred-msaa/

[10] Pettineo, M.: Recontructing position from depth. WordPress blog, [online]. 2009-03-10 [cit. 2016-05-08].
URL https://mynameismjp.wordpress.com/2009/03/10/reconstructing-position-from-depth/

[11] Reshetov, A.: Morphological antialiasing. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, The Eurographics Association, 08 2009, ISBN 9781605586038, s. 109–116, doi:10.1145/1572769.1572787.
URL http://dl.acm.org/citation.cfm?id=1572787&preflayout=flat

[12] Reshetov, A.: Reducing Aliasing Artifacts through Resampling. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, editace C. Dachsbacher; J. Munkberg; J. Pantaleoni, The Eurographics Association, 2012, ISBN 9783905674415, ISSN 20798679, doi:10.2312/EGGH/HPG12/077-086.
URL http://diglib.eg.org/handle/10.2312/EGGH.HPG12.077-086

[13] Shannon, C. E.: Communication in the Presence of Noise. In *Proceedings of the IRE*, ročník 37, 01 1949, ISSN 00968390, s. 10–21, doi:10.1109/JRPROC.1949.232969.

Přílohy

Seznam příloh

Příloha A

Obsah CD

- **appSrc** - adresář obsahující zdrojové soubory aplikace
- **textSrc** - adresář obsahující zdrojové soubory pro LaTeX
- **licenses** - licence k různým externím knihovnám
- **README.txt** - soubor obsahující informace o kompilaci a spouštení programu
- **xsvobo0k.pdf** - text této práce
- **poster.pdf** - demonstrační plakát k této bakalářské práci

Příloha B

Plakat