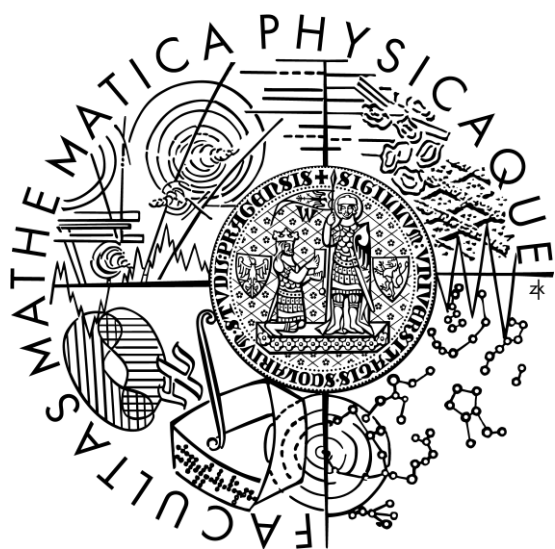


Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jana Skotáková

Bodová primitiva na grafických akcelerátorech

Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Josef Pelikán
Studijní program: Informatika

Ráda bych poděkovala RNDr. Josefu Pelikánovi za počáteční nasměrování a cenné připomínky k řešení problémů.

Prohlašuji, že jsem svou diplomovou práci napsala samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 10. 12. 2009

Jana Skotáková

Obsah

| | | |
|-------|---|----|
| 1 | Úvod | 7 |
| 1.1 | Cíl..... | 7 |
| 1.2 | Struktura textu..... | 7 |
| 2 | Modely reprezentované body | 9 |
| 2.1 | Získání dat pro reprezentaci modelu pomocí bodů | 9 |
| 2.1.1 | Skenování | 9 |
| 2.1.2 | Typy skenování..... | 10 |
| 2.2 | Post-processing | 12 |
| 2.3 | Rekonstrukce povrchu | 13 |
| 2.3.1 | Metody založené na implicitních funkcích..... | 13 |
| 2.3.2 | Metody založené na Voroného diagramu | 13 |
| 2.3.3 | Metody založené na evoluci povrchu | 13 |
| 2.4 | Převzorkování bodových modelů..... | 14 |
| 2.4.1 | Downsampling | 14 |
| 2.4.2 | Upsampling..... | 16 |
| 2.5 | Editace a modelování | 17 |
| 2.5.1 | Editace modelu | 17 |
| 2.5.2 | Modelování..... | 18 |
| 2.6 | Animace | 18 |
| 2.6.1 | Deformace elastických a plastických těles | 18 |
| 2.6.2 | Rozpad objektu | 20 |
| 2.6.3 | Simulace tekutin | 21 |
| 3 | Level Of Detail | 22 |
| 3.1 | Typy LOD | 22 |
| 3.1.1 | Diskrétní LOD | 22 |

| | | |
|-------|--|----|
| 3.1.2 | Spojité LOD | 23 |
| 3.1.3 | Pohledově závislý LOD | 23 |
| 3.2 | Generování LOD | 24 |
| 3.3 | Kritéria pro výběr diskrétního LOD | 24 |
| 3.3.1 | Vzdálenost | 25 |
| 3.3.2 | Plocha průmětu | 25 |
| 3.3.3 | Priorita | 25 |
| 3.4 | Kritéria pro pohledově závislý LOD | 25 |
| 3.4.1 | Plocha průmětu | 26 |
| 3.4.2 | Silueta | 26 |
| 3.5 | Vlivy lidského vnímání | 27 |
| 3.5.1 | Excentricita | 27 |
| 3.5.2 | Rychlost | 27 |
| 3.5.3 | Hloubka | 28 |
| 3.6 | Další techniky | 28 |
| 3.6.1 | Hystereze | 28 |
| 3.6.2 | Alpha blending | 28 |
| 3.7 | Budget-based simplification | 29 |
| 3.8 | Globální zjednodušování | 29 |
| 3.9 | Konstantní snímková frekvence | 29 |
| 3.9.1 | Reaktivní plánování | 29 |
| 3.9.2 | Prediktivní plánování | 30 |
| 4 | DirectX 10 | 31 |
| 4.1 | Grafická pipeline | 31 |
| 4.1.1 | Input Assembler | 32 |
| 4.1.2 | Vertex Shader | 32 |

| | | |
|-------|---|----|
| 4.1.3 | Geometry Shader | 32 |
| 4.1.4 | Stream Output..... | 33 |
| 4.1.5 | Rasterization Stage | 33 |
| 4.1.6 | Pixel Shader | 33 |
| 4.1.7 | Output Merger | 34 |
| 4.2 | Shrnutí změn | 34 |
| 5 | Progressive Splatting | 35 |
| 5.1 | Algoritmus | 35 |
| 5.1.1 | Inicializace..... | 35 |
| 5.1.2 | Operátor | 36 |
| 5.1.3 | Chybové metriky | 36 |
| 5.2 | Implementace | 38 |
| 5.2.1 | Inicializace..... | 38 |
| 5.2.2 | Aplikace operátoru..... | 39 |
| 5.2.3 | Algoritmy pro zpracování hran..... | 40 |
| 5.2.4 | Výstup..... | 41 |
| 5.2.5 | Přenos dat z GPU na CPU | 41 |
| 5.2.6 | Měření..... | 42 |
| 5.2.7 | Možná vylepšení..... | 48 |
| | Závěr | 50 |
| | Literatura..... | 51 |
| | Přílohy | 54 |
| | Grafy a tabulky | 54 |
| | Uživatelský manuál – program Generate Splats..... | 61 |
| | Uživatelský manuál – program Render Splats..... | 63 |
| | Obsah CD..... | 65 |

Název práce: Bodová primitiva na grafických akcelerátorech

Autor: Jana Skotáková

Katedra (ústav): Kabinet software a výuky informatiky

Vedoucí diplomové práce: RNDr. Josef Pelikán

e-mail vedoucího: josef.pelikan@mff.cuni.cz

Abstrakt: V posledních letech je bodům jako způsobu reprezentace objektů věnováno více pozornosti. V této práci jsou představeny možnosti zpracování modelů reprezentovaných body. Je zde uveden přehled algoritmů, které lze využít při řešení úloh jako je deformace nebo animace objektů reprezentovaných body. Také je popsáno, jak je možné takové modely získat. Je představen Level Of Detail objektů s ohledem na bodově reprezentované modely. Jsou zde uvedeny základní metody a kritéria pro výběr LODu objektů.

Dále je navrženo a implementováno přesunutí výpočtů zvoleného algoritmu pro generování LOD z CPU na GPU. Tím se CPU uvolní pro jiné části algoritmu nebo i pro jiné úlohy. K programování GPU je využit DirectX 10. Nový model grafické pipeline, kterou DirectX 10 používá, umožňuje přenést na GPU více částí algoritmu.

Klíčová slova: Bodová primitiva. LOD. Geometry shader.

Title: Point primitives on graphic accelerators

Author: Jana Skotáková

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán

Supervisor's e-mail address: josef.pelikan@mff.cuni.cz

Abstract: In recent years representation of objects based on point primitives receives more interest. This thesis presents possibilities of processing of point-sampled models. It gives an overview of algorithms that can be used in solving problems like deformation or animation of point-sampled models. There is also described how such models can be obtained. In the following part Level Of Detail is presented with regard to point-sampled models. There are given basic methods and factors used for the LOD selection.

Furthermore, there is proposed and implemented transfer of parts of the selected LOD generating algorithm to GPU. This keeps CPU available for other parts of the algorithm or even other tasks. DirectX 10 is used for the GPU programming. DirectX 10 uses the new model of graphic pipeline that enables the transfer of more parts of algorithm to GPU.

Keywords: Point-sampled model. LOD. Geometry shader.

1 Úvod

Body jsou nejjednodušší grafická primitiva, kromě 2D nebo 3D souřadnic mohou mít i další vlastnosti jako je barva nebo normála. Body nenesou žádnou informaci o vzájemné konektivitě, nemají žádnou informaci o topologii objektu, který reprezentují. Pokud algoritmus potřebuje znát konektivitu a topologii, zjišťuje se například pomocí k nejbližších sousedů. Tato vlastnost je pro některé metody výhodná, například pro rozsáhlé deformace objektů se topologie velmi jednoduše dynamicky mění, na rozdíl od trojúhelníkových sítí, zároveň ovšem opakované zjišťování sousedních bodů zabírá čas a paměť.

Body jsou pro reprezentaci flexibilnější než trojúhelníkové sítě, nemají ale takovou podporu pro zobrazování v grafickém hardware a API, zobrazení objektu reprezentovaného body je složitější než zobrazení objektu reprezentovaného trojúhelníkovou sítí. Při vykreslení objektu čistě pomocí bodů vznikají v objektu díry na místech, kde body chybí, je tedy nutné místo bodů zobrazovat alespoň kruhy, které se vzájemně překrývají.

1.1 Cíl

Cílem této práce je přenést zvolený algoritmus pro vytváření Level Of Detail objektů reprezentovaných body z procesoru (dále CPU) na grafický procesor (dále GPU). Není samozřejmě možné převést všechny části algoritmu, GPU není tak obecný jako CPU, udržování grafu popisujícího konektivitu bodů tak probíhá na CPU, ale některé části výpočtu na GPU přenést lze. Využitím CPU i GPU by výpočet měl probíhat paralelně a ve výsledku rychleji. Ideální by bylo plné využití CPU i GPU. K programování GPU je využit Microsoft DirectX 10, který používá Shader Model 4.0. Do grafické pipeline je v tomto modelu přidána nová část – geometry shader, který umí pracovat s celými primitivy, díky tomu lze v geometry shaderu například počítat ohodnocení hran grafu nebo provádět výpočet nového bodu ze dvou původních.

1.2 Struktura textu

V této práci je představeno zpracování modelů reprezentovaných body. Tuto reprezentaci lze využít ve všech fázích práce s objektem – modelování a deformace, animace atd. V následující kapitole jsou stručně popsány principy algoritmů, které se k tomu používají. Konkrétní vzorce a výpočty zde uváděny nebudou, v jednom algoritmu se často využívá vzorců několik, pro lepší pochopení je vhodné uvádět také jejich odvození. Tyto detaily lze dohledat v literatuře uváděné u jednotlivých algoritmů, mnoho z nich je podrobně popsáno v [9].

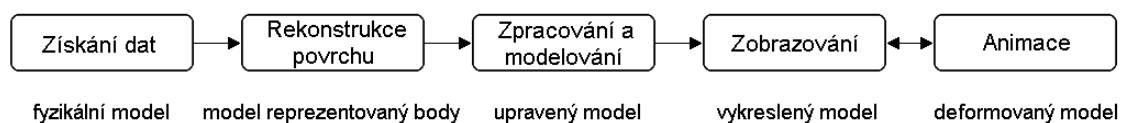
Vzhledem k zaměření praktické části této práce je ve třetí kapitole popsán Level Of Detail (dále LOD). Jsou uvedeny základní typy LODů a kritéria pro výběr úrovně pro zobrazení objektu (v této práci je používán termín verze). Většina algoritmů a kritérií pro LOD je prakticky shodná pro body i trojúhelníkové sítě, liší se především způsob vytváření LODů. Bodová reprezentace používá jiné operátory a metriky pro měření chyb než trojúhelníková, většinou jsou to ale zobecněné a upravené operátory a metriky pro trojúhelníkové sítě. Detailní informace o LOD lze najít v [18], která se zabývá trojúhelníkovými modely, generování bodového LOD a jeho zobrazování je popsáno v [9].

Ve čtvrté kapitole je představena grafická pipeline a DirectX 10, který je vyžit při implementaci zvoleného algoritmu.

Poslední kapitola je věnována samotnému vybranému algoritmu „*Progressive Splatting*“, představeného v článku [34], a jeho implementaci.

2 Modely reprezentované body

V této kapitole jsou stručně představeny základní způsoby práce s modely reprezentovanými pomocí bodů (viz Obrázek 2.1). Je zde popsáno, jak získat data pro reprezentaci modelu skenováním reálných objektů a následnou rekonstrukcí povrchu, jak se modely editují, jak zobrazit model a jak lze provádět animaci modelu, kdy se tvar modelu a jeho atributy mění v čase. Podrobný popis postupů, algoritmů a datových struktur lze nalézt v [9].



Obrázek 2.1 Postup práce s modelem reprezentovaným body. Vytvořeno podle obrázku v [9].

2.1 Získání dat pro reprezentaci modelu pomocí bodů

Pro práci s modelem je třeba nejprve získat data – tedy množinu bodů, které reprezentují nějaký reálný model. Data lze získat skenováním reálného modelu pomocí různých typů skenerů. Takováto data se získávají snadno a poskytují detailní informace o objektu.

Nevýhodou takovýchto dat je především jejich velké množství, což vyžaduje efektivní algoritmy pro jejich zpracování. Další vlastností, se kterou je třeba počítat, je šum a ne vždy dostačující počet vzorků v některých částech modelu.

2.1.1 Skenování

Jeden sken většinou nezachytí celý reálný model, ale jen jednu jeho část. Proto je nutné pořídit skenů více a z nich teprve vytvořit trojrozměrný model. Tento proces se dá rozdělit na několik základních fází, které jsou přítomné při zpracovávání dat získaných použitím libovolného typu skeneru:

- Skenování
- Registrace
- Sloučení dat
- Plánování pozice nového skenu
- Post-processing

Skenování znamená pořízení nového skenu a tedy získání dalších dat.

Registrace představuje začlenění nového skenu do stávajících a může zahrnovat zjišťování pozice nového skenu, vyhledávání a porovnávání různých příznaků v datech a další metody.

Ke *sloučení dat* dochází, protože mnoho „sousedních“ skenů se z části překrývá. Sloučením dat dochází ke zmenšení potřebné paměti a z části i k redukci šumu. Algoritmů, které provádějí toto slučování, je mnoho. Mezi ně patří například algoritmy založené na implicitních funkcích, „moving least squares“ a další.

Pro další sken je potřeba *naplánovat jeho pozici* – tedy jakou další část reálného modelu je potřeba snímat. K tomu je potřeba zjistit, podle dat získaných z předchozích skenů, ve které části modelu data chybí – v modelu je díra, nebo je jich příliš málo – část modelu je podvzorkována. Určení nové pozice je většinou ponecháno na uživateli, k tomu je nutné umět tyto částečné výsledky zobrazit.

Post-processing zahrnuje, podle způsobu využití modelu, redukci šumu, rozpoznání a odstranění bodů ležících mimo povrch modelu („outliers“), vyplnění děr nebo převzorkování podle křivosti a další metody.

Formát dat

Pro předávání dat mezi fázemi skenování je možné použít data ve formátu množiny bodů („unorganized point clouds“). Data jsou reprezentována jako jednoduchý seznam bodů a jejich vlastností. Tato reprezentace může být použita mezi libovolnými fázemi.

Druhou možností je předávat data ve formátu 2 ½ D obrazu (nebo také „ranged image“, „depth image“). Data jsou podobná normálnímu obrazu, ale místo barev se ukládá hloubka. Tento formát je často výstupem skenerů, snadno se v něm určují prázdná místa a dá se v něm lépe zjišťovat sousednost či spojitost.

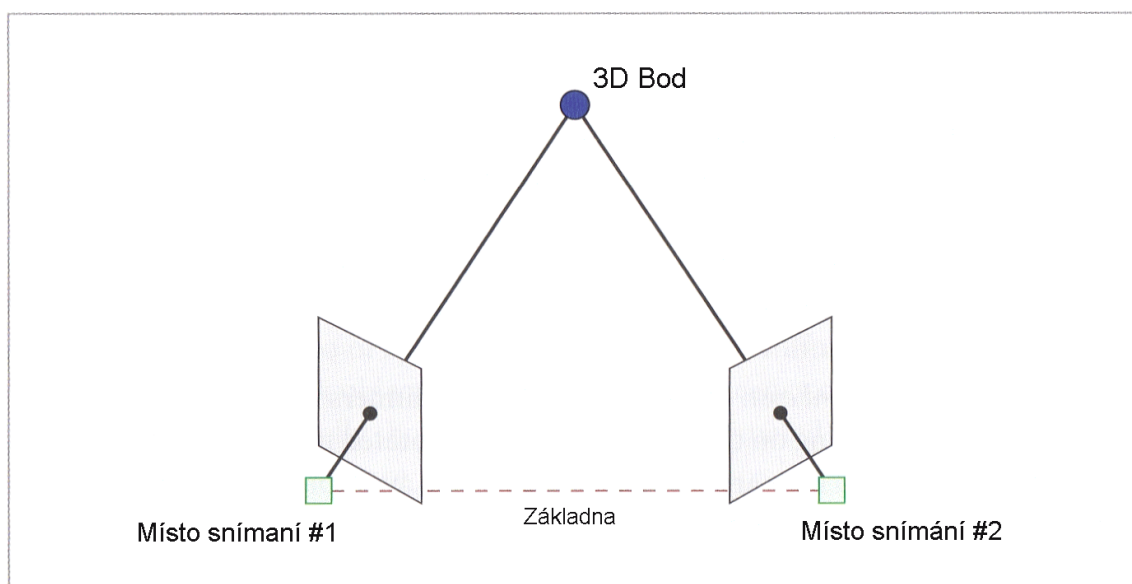
2.1.2 Typy skenování

Způsobů, jak skenovat reálný objekt a převést tak jeho reprezentaci do počítače pro další práci, je mnoho. Mezi ně patří skenování založené na triangulaci, skenování založené na čase letu paprsku a další.

2.1.2.1 Skenování založené na triangulaci

Při tomto typu skenování je model snímán ze dvou (nebo i více) míst současně. U vzniklých obrazů se pak musí určovat jejich korespondence.

Místa, ze kterých je objekt snímán, a samotný objekt vytváří trojúhelník, strana mezi místy snímání se nazývá základna (viz Obrázek 2.2). Při snímání vzniknou dva velmi podobné obrazy téže části objektu, na kterých se provádí korespondence. Vzdálenost bodů modelu se určuje spočítáním průsečíků paprsků z míst snímání objektu. Je také potřeba provádět kalibraci – mapování pixelů ze dvou paprsků v prostoru [10].



Obrázek 2.2 Skener založený na triangulaci hledá pozici bodů na povrchu objektu spočítáním korespondujících pixelů v obrazech získaných ze dvou míst. Korespondence definuje dva paprsky v prostoru, průsečík těchto paprsků určuje bod v 3D prostoru. Vytvořeno podle obrázku v [9].

Výhodou tohoto skenování je flexibilní umístění modelu vzhledem ke kamerám – toto umístění je dáno především vzájemnou vzdáleností kamer. Lze tedy jednoduše skenovat jak malé tak i velké objekty. Pro snímání velkých scén je ovšem potřeba velká základna snímání a kalibrace je tedy složitá.

Problémem při korespondenci jsou zakryté části modelu, kdy část modelu viditelná z jednoho místa je při pohledu z druhého místa zakrytá. Může dokonce dojít k situaci, že nějakou část modelu nelze sejmout ze dvou míst bez změny základny a úhlu snímání. V případě bodů pozorovatelných jen z jednoho místa je nutné použít algoritmy pracující nad celým modelem, aby nebyly při korespondenci označeny jako chybné („outliers“).

Metody založené na triangulaci předpokládají snímání opakního a matného objektu. Při skenování lesklých nebo průhledných objektů dochází k chybné korespondenci.

Metody lze dále rozdělit podle typu objektů v místech snímání:

- *Pasivní stereo* – na místech snímání jsou kamery, ve scéně není žádný kontrolovaný zdroj světla. Určení korespondence může být složitý problém, v datech je hodně šumu a jsou často řídká (model je podvzorkován).
- *Aktivní stereo* – na místech snímání jsou také kamery, pro snazší určení korespondence se ale provádí projekce měnícího se vzorku na snímáný model.

- „*Structured light system*“ – místo jedné kamery je umístěn zdroj světla. Speciálně jednou z možných variant je „single-light stripe“ systém. Data získaná tímto způsobem mají dobrou kvalitu, jsou dostatečně hustá. Nevýhodou je, že jejich získání dlouho trvá - pruh světla musí postupně projít přes celou scénu („sweeping“). Světlo také musí být detekovatelné v celé scéně, takže skenování venku za denního světla nemusí být proveditelné.

2.1.2.2 Skenování založené na čase letu paprsku

Jedním z těchto systému je *LIDAR* („Light detection and ranging“), který funguje podobně jako radar nebo sonar, ale vysílá pulzy světelné. Vzdálenost modelu se určuje pomocí času od vyslání paprsku ke vrácení odraženého.

Vzhledem k tomu, že se model snímá pouze z jednoho místa, není potřeba, na rozdíl od triangulace, žádná kalibrace. Systém je také schopen získat geometrii, kterou triangulační metody neovládají. Nevýhodou je dlouhý čas potřebný ke snímání objektu.

Jiný systém je založeným na *modulaci intenzity* světla. Vysílá se spojitý paprsek světla, který mění intenzitu. Vzdálenost modelu se počítá podle změny fáze odraženého paprsku. Volba modulace je důležitá, vzhledem k cyklickému opakování fáze. Model se často snímá při několika různých modulacích pro získání lepšího výsledku.

2.1.2.3 Určení tvaru objektu ze siluety

Další možné systémy jsou založené na rozpoznávání vnější siluety objektu v jednotlivých skenech [21]. Výhodou tohoto způsobu zpracování je získání „těsného“ povrchu bez děr („watertight surface“) – na rozdíl od jiných metod, u kterých je na závěr zpracování potřeba vyplnit díry v modelu apod. Problémem je rozpoznávání konkávních objektů.

2.2 Post-processing

Data získaná skenováním reálného modelu obsahují často různé artefakty. Výskyt artefaktů v datech je závislý na typu skenování a použitém zařízení. Mezi typické artefakty patří například:

- Šum – je zapříčiněný fyzikálními limity senzorů, vzniká i při kvantizaci nebo při pohybu snímaného objektu (například při skenování zvířat nebo lidí).
- Díry, podvzorkování některých částí objektu – jsou způsobeny zakrytím částí objektu, vlastnostmi povrchu (jeho odrazivost a především vysoký lesk) nebo také omezeným rozlišením senzorů skeneru.
- Textury na povrchu objektu mohou „zmást“ skener a ten vytváří chybnou geometrii.

Rozpoznání artefaktů je obtížné a většinou vyžaduje spolupráci uživatele.

Post-processing je vhodné provádět přímo na získané množině bodů, ještě před aplikací algoritmů pro rekonstrukci povrchu nebo dalším modelováním.

2.3 Rekonstrukce povrchu

Rekonstrukce povrchu znamená převést objekt reprezentovaný množinou bodů („point cloud“) například na trojúhelníkovou síť („triangle mesh“) nebo implicitní funkci.

Výběr metody pro rekonstrukci závisí na vlastnostech vstupu, například přítomnosti šumu, a požadovaném typu výstupu – některé algoritmy pro další zpracování modelu vyžadují watertight surface.

Mezi metody pro rekonstrukci povrchu modelu patří metody využívající implicitní funkce, metody založené na Voroného diagramu nebo metody evoluce povrchu.

Některé metody pro svůj běh vyžadují už spočtené normálové vektory vstupních bodů, jiné je zjišťují v průběhu práce. U některých algoritmů je výpočet normál jejich prvním krokem. Pokud už vstupní data obsahují normály, je rekonstrukce povrchu snazší.

Odhad normálových vektorů

Normála v daném bodě p se počítá pomocí k nejbližších bodů k bodu p [13], nebo pomocí všech bodů ležících ve vzdálenosti r od bodu p , tuto vzdálenost lze adaptivně měnit [22]. Pomocí těchto bodů se určí rovina metodou nejmenších čtverců („total least square plane“) a její normálový vektor se použije jako normála v bodě p .

2.3.1 Metody založené na implicitních funkcích

Tyto metody použijí vstupní body k vytvoření funkce $f()$ na 3D prostoru takové, že uvnitř objektu je $f()$ záporná a vně objektu je kladná. Povrch tedy představují hodnoty $f() = 0$. $f()$ mohou být různé funkce s takovou vlastností zvolené podle konkrétní metody. Výhodou těchto metod je, že jejich výsledkem je watertight surface.

2.3.2 Metody založené na Voroného diagramu

Metody využívající Voroného diagramu a Delaunayovy triangulace vyžadují dostatečně husté vzorkování objektu, pro ostré hrany dokonce vychází, že potřebné vzorkování je nekonečně husté. Metody proto pracují jen na hladkém povrchu. Dalším důležitým předpokladem je, že vstupní data neobsahují žádný šum. Naopak nepotřebují na vstupu dostávat normálové vektory, ty jsou spočítány během práce algoritmu.

2.3.3 Metody založené na evoluci povrchu

Metody jsou založené na postupné deformaci jednoduchého povrchu podle daných pravidel. Pravidla zachovávají strukturu povrchu a zároveň postupně přibližují povrch vstupním datům.

2.4 Převzorkování bodových modelů

Model je reprezentován určitým počtem bodů (vzorků) a jejich vlastnostmi – souřadnice bodu, hloubka, barva, případně i normála a další atributy. Zachycení důležitých detailů objektu závisí na hustotě vzorkování.

Pokud je model změněn, například deformací, může se stát, že vzorkování (počet a rozložení bodů) v části modelu bude nedostačující a bude nutné přidat další body – *upsampling*.

Naopak při velkém počtu bodů je možné jejich počet snížit – *downsampling*, další zpracování pak bude probíhat rychleji.

2.4.1 Downsampling

Při laserovém skenování vzniká velké množství bodů, je potřeba jejich počet zredukovat – provést downsampling. Pro práci s modelem je možné použít jak body, tak reprezentaci pomocí „splats“, kdy je povrch lokálně aproximován kruhem nebo elipsou.

Body netvoří souvislý povrch a nenesou žádnou informaci o vzájemné konektivitě. Při reprezentaci modelu body je nutné pro zmenšení chyby dvakrát zvětšit počet bodů čtyřikrát.

Při použití splatů je pro snížení chyby dvakrát nutné zvětšit počet splatů na dvojnásobek – což je stejné jako při reprezentaci modelu trojúhelníkovou sítí. Reprezentace pomocí splatů ale netvoří souvislý povrch a zachovává se jejich vzájemná nezávislost (stejně jako u bodů).

Při provádění downsamplingu modelu je možné počítat pouze se středy splatů (tedy body), což je rychlejší, nebo i s jejich geometrií, což dává lepší aproximaci povrchu objektu.

2.4.1.1 Metody založené na zjednodušování pomocí bodů

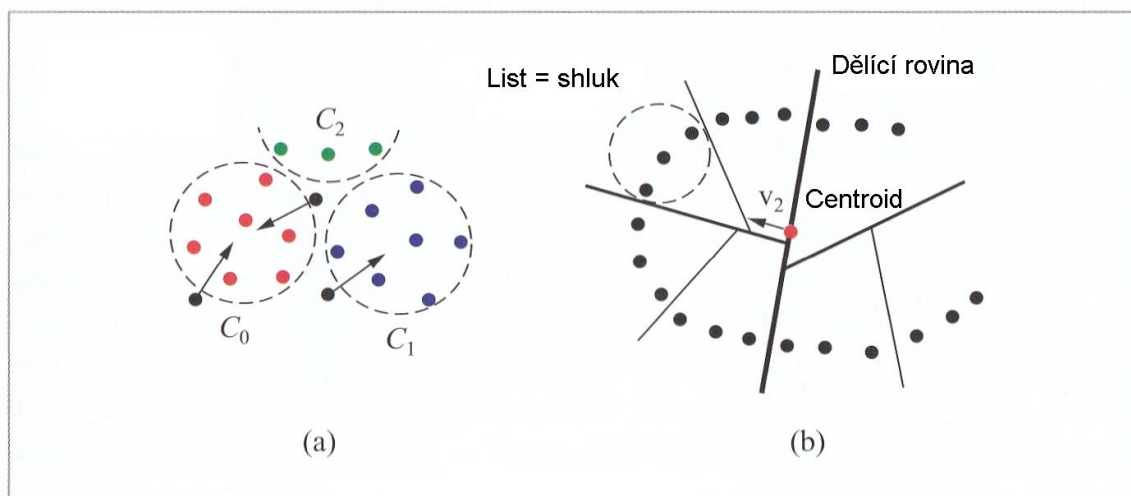
Metody zjednodušování pomocí bodů jsou stejné jako metody pro zjednodušování trojúhelníkových sítí („mesh simplification“), pouze jsou adaptované pro body místo pro polygony. Tyto metody se nazývají „point simplification“.

Shlukování

Při této metodě se vytváří shluky („clusters“), jejichž maximální velikost je dána zvolenými kritérii, a body obsažené v shluku se nahradí jedním vzorkem – centroidem shluku (viz Obrázek 2.3).

Shluky je možné stavět odspodu – „*region growing*“. Do shluku se přiřadí nejbližší dosud nezařazené body od zvoleného bodu. Některé shluky ovšem mohou zůstat příliš malé, body v takovýchto shlucích jsou distribuovány do vedlejších a shluk je zrušen.

Opačný způsob vzniku shluků se nazývá hierarchické shlukování („*hierarchical clustering*“) [2, 30]. Shluky vznikají rozdělením příliš velkého shluku na dva menší a vzniká tak binární strom shluků. Shluk se dělí podél osy největší korepondence bodů.



Obrázek 2.3 (a) Shlukování pomocí „region growing“, dosud nezařazené body se připojí ke shluku s nejbližším centroidem. (b) Hierarchické shlukování, tloušťka čáry dělicí roviny indikuje hloubku v BSP stromě (pro ilustraci je 2D). Vytvořeno podle obrázku v [9].

Iterativní point-simplification

Při iterativním point-simplification algoritmu [28] se počet bodů redukuje opakovaným použitím definovaných operátorů – podobně jako u zjednodušování trojúhelníkových sítí [12]. Operátory se aplikují podle chybové metriky, ty s menší chybou dříve. Operátorem je kontrakce dvou bodů do jednoho – „point-pair contraction“ (analogický operátor ke kontrakci hrany).

Dvojice bodů pro aplikaci operátoru se vytváří pomocí k nejbližších sousedů, bod a jeho sousedi tvoří dvojice. Nový bod vzniklý po použití operátoru získá jako sousedy všechny sousedy obou původních bodů.

Simulace částic

Simulace částic [28] se provádí pomocí adaptovaného point-repulsion algoritmu pro polygony [31]. Požadovaný počet částic je náhodně rozmístěn po povrchu modelu a jejich pozice se mění použitím point-repulsion algoritmu. Při inicializaci simulace se přidá více vzorků do těch částí objektu, kde je nižší hustota vzorků. Pro změnu pozice částic se používá lineární funkce jako „repulsion force“, tato funkce má konečný dosah – částice proto působí pouze na své sousedy, ne na všechny částice v modelu.

2.4.1.2 Metody založené na zjednodušování pomocí splatů

Metody založené na zjednodušování pomocí splatů se nazývají splat-decimation metody. Jsou velmi podobné point-simplification metodám, na rozdíl od nich však pracují s celou geometrií splatu.

Metody založené na hierarchickém shlukování

Vstupní body jsou zařazeny do hierarchické struktury jako je například octree [27], splat je vytvořen pro každý uzel. Algoritmus je rychlý a jednoduchý, nevýhodou je redundance vzniklých splatů.

Iterativní hladový algoritmus

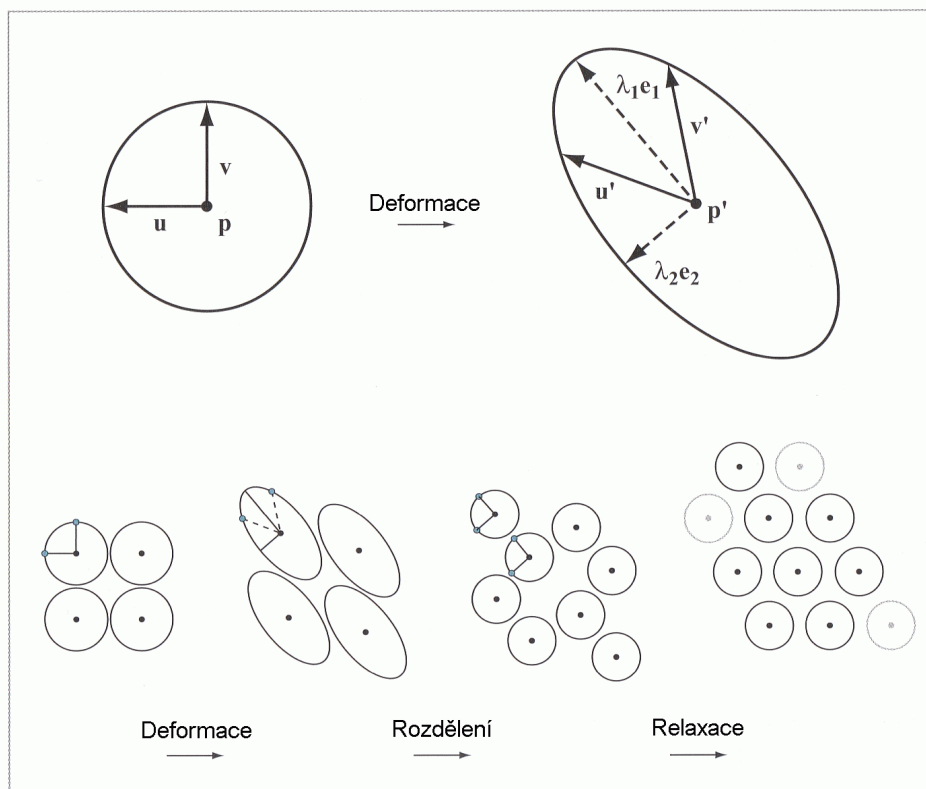
Iterativní hladový algoritmus [34] probíhá podobně jako iterativní point-simplification algoritmus. Na začátku se pro každý bod vytvoří iniciální splat. Počet splatů se redukuje pomocí „splat-merge“ operátoru, kdy se vybraná dvojice splatů nahradí jedním větším splatem. Aplikuje se operátor s nejmenší chybou podle zvolené chybové metriky. Dvojice splatů, které se mohou spojit, se určují stejně jako u iterativního point-simplification algoritmu pomocí k nejbližších bodů.

Optimalizovaný splat-decimation algoritmus

Tento algoritmus pomocí globální optimalizace spočítá aproximaci nejmenší množiny splatů, která je potřebná pro pokrytí celého povrchu modelu, a přitom chyba zůstane pod danou maximální tolerancí [33]. Výsledný počet splatů získaných touto metodou je výrazně nižší než u jiných metod, algoritmus je ovšem výpočetně náročnější.

2.4.2 Upsampling

Po deformaci modelu se může stát, že vzorkování v deformované oblasti nebude dostačující a bude nutné přidat další body [29]. Nejdříve se změří napětí povrchu („surface stretch“) a poté se příliš zdeformované splaty rozdělí na nové – tím vzniknou nové vzorky (viz Obrázek 2.4). K určení nové polohy se použije lineární relaxační filtr, který je podobný „repulsion force“ u simulace částic. K určení ostatních vlastností se použijí další interpolační filtry.



Obrázek 2.4 2D ilustrace lokálních změn po deformaci modelu a dynamické převzorkování. Vytvořeno podle obrázku v [9].

2.5 Editace a modelování

Model objektu získaný skenování a následnou rekonstrukcí povrchu je možné dále upravovat, měnit jeho tvar nebo barvu.

2.5.1 Editace modelu

Editace modelu se provádí pomocí úprav používaných u 2D obrazů, zobecněných pro 3D modely, které jsou reprezentované body.

Při editaci modelu uživatel vybere část, která se má upravit, a vybere „štětec“, který se má použít. Štětec je představován obrazem, který se má nanést na editovanou část modelu. Samotná editace modelu probíhá v několika fázích. První fází je parametrizace souřadnic té části modelu, která se má změnit (vznikne tzv. „patch“). V další fázi je potřeba vytvořit vzorkovací síť pro editovaný patch a štětec takovou, že mezi vzorky obrazu a štětce je jednoznačná korespondence. Toto většinou vyžaduje převzorkování patch. Nakonec se oba výsledky – štětec a editovaný patch, zkombinují dohromady.

2.5.2 Modelování

Změna tvaru modelu se dá provádět pomocí různých technik, například použitím booleovských operátorů nebo pomocí free-form deformace. Booleovské operátory se snadno definují na implicitním povrchu, naopak free-form deformace pro explicitní reprezentaci. Každou technikou lze dosáhnout jiných výsledných tvarů objektu.

2.5.2.1 Booleovské operátory

Složitý tvar modelu lze vytvořit kombinací jednoduchých tvarů pomocí booleovských operátorů.

Výsledný model vytvořený pomocí booleovské operace bude obsahovat podmnožinu bodů obou původních objektů a nové body, které budou reprezentovat průsečnice těchto objektů. Pro zjištění podmnožin je nutné provést klasifikaci – které body z jednoho modelu jsou uvnitř či vně druhého objektu. Dále je potřeba provést výpočet bodů ležících na průsečnicích objektů. Při použití pouze bodů z původních modelů by nebyla tato část povrchu dostatečně navzorkována.

2.5.2.2 Free-form deformace

Umožňuje ohýbání modelu, jeho různá zkroucení, natahování nebo smršťování a další deformace.

Uživatel označí na povrchu modelu oblast, která se má deformovat a vybere řídicí body. Povrch se pak modifikuje posuny řídicích bodů.

Často je nutné deformovanou oblast převzorkovat, původní splaty už dobře nezachycují novou geometrii této části objektu.

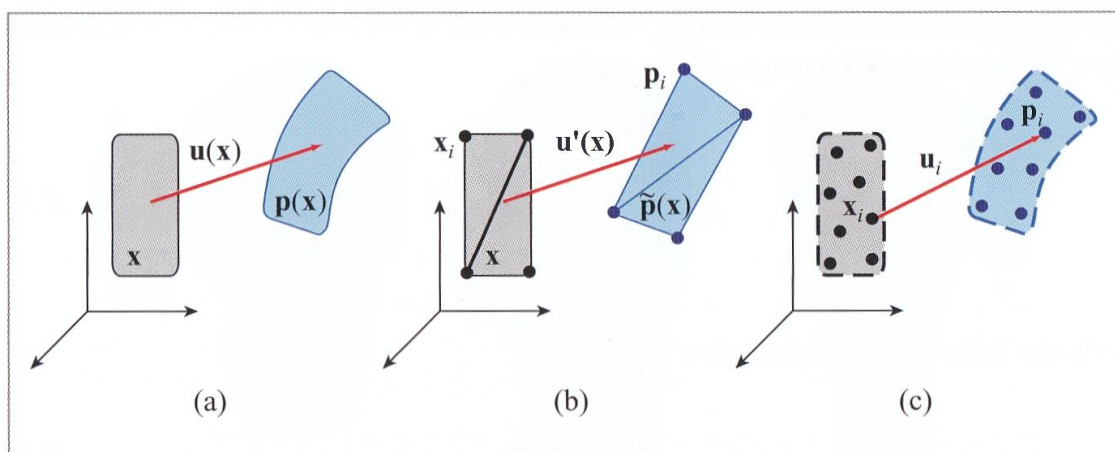
2.6 Animace

Pod animaci modelu můžeme zařadit jeho deformaci nebo rozpad na několik částí („fracturing materials“). Také pohyb tekutin lze animovat pomocí bodů.

2.6.1 Deformace elastických a plastických těles

Pro práci s objekty, jako je modelování a renderování, většinou stačí reprezentovat jejich povrch. Ovšem pro spočítání a animaci deformace tělesa pomocí bodů je potřeba reprezentovat a modelovat i jeho vnitřek.

Pro reprezentaci objemu tělesa a jeho elastických vlastností se použijí body - phyxely („physical element“), na kterých se bude počítat deformace. Pro reprezentaci povrchu se využijí další body – surfely. Surfely bude více než phyxely pro lepší výsledky při zobrazování objektu.



Obrázek 2.5 (a) Deformace tělesa je matematicky definována jako spojité vektorové pole $\mathbf{u}(\mathbf{x})$, které popisuje změnu polohy u každého bodu tělesa. Bod na pozici \mathbf{x} se po deformaci posune na místo $\mathbf{p}(\mathbf{x}) = \mathbf{x} + \mathbf{u}(\mathbf{x})$. (b) U metod pro trojúhelníkové sítě je $\mathbf{u}(\mathbf{x})$ aproximováno polem $\mathbf{u}'(\mathbf{x})$, které interpoluje posunutí v rozích elementu. (c) Při použití bodů je pole $\mathbf{u}(\mathbf{x})$ reprezentováno množinou diskrétních vzorků \mathbf{u}_i definovaných v bodech \mathbf{x}_i . Hodnoty posunutí mezi těmito body jsou interpolovány. Převzato z [9].

2.6.1.1 Metoda meshless finite elements

Metoda „meshless finite elements“ [24] počítá deformaci objektu pomocí rovnic teorie elasticity, která popisuje chování trojrozměrných těles při působení síly. Mezi důležité veličiny patří tuhost tělesa, posunutí, napětí a tlak.

Objem tělesa je reprezentován konečným počtem bodů – phyxelů, bez informace o jejich vzájemné konektivitě. Tyto phyxely určují původní nedeformovaný tvar tělesa včetně jeho materiálových vlastností – tj. jak se těleso chová při působení vnější síly, jakým způsobem se deformuje.

Při deformaci elastického tělesa se spočítá, jaké je napětí a tlak v jednotlivých phyxelech při působení vnější síly a jak se jejich vlivem ve výsledku změní souřadnice phyxelů. Do výpočtu se zahrnuje i vliv sousedních phyxelů.

Po ukončení působení sil se těleso vrátí do původního nedeformovaného stavu.

2.6.1.2 Deformace plastických objektů

Plastické těleso zůstává deformované i po skončení působení sil. Pro simulaci plastického materiálu se pro každý phyxel uchovává část spočetné deformace („strain state“) [25], takže se phyxely nevrací hned do původní pozice.

2.6.1.3 Zobrazení objektu

Pro zobrazení deformovaného objektu se pomocí váženého průměru spočítá posunutí surfelu podle posunutí sousedních phixelů. Pokud se počítá změna i pro tečny surfelů, může být dopočítána i deformace surfelů a normály a lze provádět převzorkování surfelů.

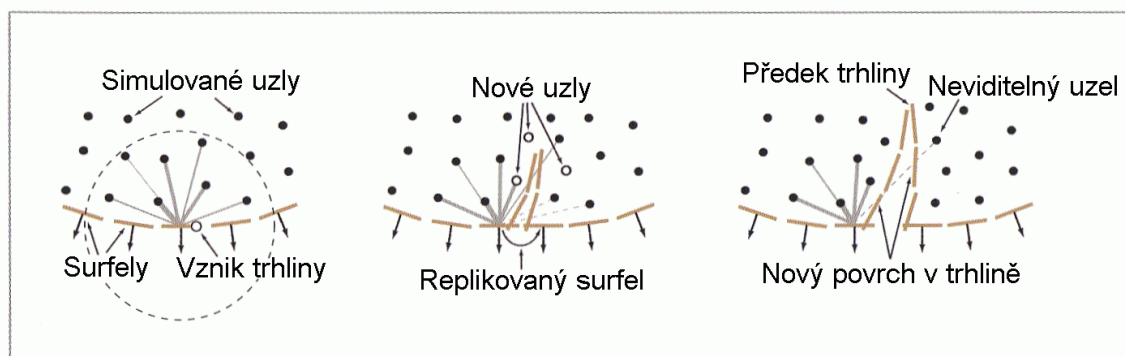
2.6.2 Rozpad objektu

Způsob rozpadu tělesa závisí na typu jeho materiálu – objekt z křehkého materiálu se rozpadne na kousky, objekt z tažného materiálu před rozpadem často projde plastickou deformací.

Meshless metody mají při simulaci rozpadu tělesa několik výhod proti mesh metodám. Díky absenci topologie se neprovádí složitý remeshing objektů. Jsou výhodné i pro velké deformace, protože převzorkování je pouze lokální. Nevýhodou je, že i malé fragmenty musí být dostatečně hustě vzorkovány, což zvětšuje množství phixelů, na kterých se provádí výpočty, a výpočet se zpomaluje.

Objekt je opět reprezentován phixely. Sousedící phixely se navzájem ovlivňují, pokud od sebe nejsou odděleny povrchem – tedy pokud mezi nimi nevede prasklina [1]. Prasklina je představována dvěma povrchy a předkem („crack front“), kde se tyto dva povrchy stýkají.

Praskliny ovšem vedou k nežádoucím diskontinuitám při výpočtu, tomu se dá zabránit například zavedením „průhlednosti“ trhliny [26] - trhlina je vpředu více průhledná. Phixely oddělené trhlinou spolu dále interagují, ale trhlina je „oddaluje“ a jejich vzájemný vliv se snižuje, až se, při dalším zvětšování trhliny, ovlivňovat přestanou (viz Obrázek 2.6, kde je tato vlastnost ilustrována na surfelích).



Obrázek 2.6 Váhy podle průhlednosti trhliny pro surfely, tloušťka čáry naznačuje vliv phixelu na posunutí surfelu. Při zvětšování trhliny jsou vytvářeny nové phixely a surfely pomocí dynamického převzorkování. Vytvořeno podle obrázku v [9].

Vzorkování phixelů a surfelů okolo praskliny a samotné praskliny - surfelů reprezentujících povrchy praskliny a uzlů reprezentujících předeek trhliny, je nutné dynamicky přizpůsobovat – probíhá jak upsampling tak i downsampling.

Při simulaci rozpadu tělesa trhliny vznikají, zanikají a také se mohou spojovat.

2.6.3 Simulace tekutin

Simulace tekutin („fluid simulation“) zahrnuje například simulaci kouře, ohně, mraků, vody nebo lávy. Pro simulaci se používají dva odlišné typy metod – Eulerovské metody a Lagrangeovské metody.

2.6.3.1 Eulerovské metody

Eulerovské metody diskretizují prostor, ve kterém simulace probíhá, na elementy – materiál se pohybuje skrz tyto základní elementy. Zjišťování povrchu a měnících se hranic látky je složité. Může také dojít ke ztrátě objemu simulované látky kvůli matematickým chybám.

Všechny tyto metody jsou díky svému principu mesh-based.

2.6.3.2 Lagrangeovské metody

Lagrangeovské metody naopak diskretizují materiál a nikoli prostor, základní elementy se pohybují s látkou. Zjišťování měnících se hranic látky a povrchu je tedy snazší. Protože každý element představuje určité množství simulované látky, nemůže, na rozdíl od Eulerovských metod, dojít ke ztrátě objemu látky.

Simulace částic

Pro simulaci tekutin pomocí částic se používá metoda „Smoothed particle hydrodynamics“ [23, 16]. Materiál je reprezentován částicemi a jejich vlastnostmi – objem látky, který každá částice reprezentuje, rychlost pohybu apod., a funkcí určující vliv částice na okolí. Určení pohybu částice závisí především na viskozitě materiálu, tlaku a okolních částicích.

2.6.3.3 Zobrazování

Pro zobrazení plynných látek se používá volume rendering – zobrazují se poloprůhledné koule s objemovou texturou. Pro kapaliny je nutné spočítat a reprezentovat jejich povrch - například pomocí surfelů. Přidáním dalších technik (např. normal displacement) lze zobrazovat povrch detailněji nebo i zjednodušit samotnou simulaci.

3 Level Of Detail

Zobrazování velkého počtu grafických primitiv, ať už bodů nebo trojúhelníků, je časově a paměťově náročné. Pokud by se každý objekt ve scéně vykresloval v plném rozlišení, při větším počtu objektů by renderování jednoho snímku trvalo příliš dlouho. Vzdálenější objekty navíc na displeji často zabírají jen několik pixelů a jejich vykreslení použitím všech původních primitiv je zbytečné.

Vzdálené objekty ve scéně se tedy nemusí zobrazovat v plném rozlišení, stačí je zobrazit méně detailně pomocí menšího počtu bodů či trojúhelníků a výsledný dojem pro pozorovatele bude stejný nebo alespoň dostatečný pro rozpoznání objektu. Ušetřená grafická primitiva – ať už jsou to body nebo trojúhelníky, lze využít pro jiné objekty ve scéně.

Jeden objekt tak lze reprezentovat pomocí různých verzí – od originálního plného rozlišení pro případ, kdy je objekt blízko pozorovateli, přes verzi obsahující méně bodů až po nejmenší rozlišení, kdy je objekt daleko a zabírá jen několik málo pixelů. Techniky pracující na tomto principu se nazývají Level Of Detail (LOD) – objekt se podle určitých kritérií zobrazuje v různě detailních verzích.

Jednou z technik, kde se využívá LOD, je udržení konstantní snímkové frekvence: fixed frame rate schedulers.

Podrobný popis LODů, kritérií pro výběr apod. lze nalézt například v [18]. Tam je popsán LOD pro modely reprezentované pomocí trojúhelníkových sítí, většina principů je stejná i pro modely reprezentované body.

3.1 Typy LOD

LOD lze rozdělit na tři základní typy:

- diskrétní LOD – objekt je reprezentován pomocí několika málo úrovní detailu,
- spojitý LOD – úroveň detailu objektu se mění v malých krocích,
- pohledově závislý LOD – dynamicky se mění přímo rozlišení jednotlivých částí objektu.

3.1.1 Diskrétní LOD

Při používání diskrétního LODu se předem připraví několik verzí téhož objektu a poté se podle daných kritérií zobrazuje vybraná verze.

Úrovní detailu bývá několik – od nejdetailnější až po nejméně detailní, poslední „úroveň“ je obvykle prázdná množina (bodů či jiných grafických primitiv), kdy se objekt vůbec nezobrazuje.

Verze objektu se připravují předem offline, generování tedy může trvat tak dlouho, jak je potřeba. Vzhledem k tomu, že se při offline generování nedá předvídat, ze které strany bude objekt pozorován, zjednodušování objektu probíhá uniformně, proto se diskrétní LOD nazývá také isotropní nebo pohledově nezávislý.

3.1.2 Spojitý LOD

Spojitý nebo také progresivní LOD mění úroveň detailu v malých krocích. Při generování LODu se tyto kroky ukládají do datové struktury, ze které se pak za běhu vybírá verze objektu. Vytváření této struktury probíhá při předzpracování offline.

Výhodou této metody je lepší využití zdrojů, daný objekt se zobrazí pomocí jen tolika bodů nebo polygonů, kolik jich je nutné použít. Pro jiné důležitější objekty se pak může využít vyšší počet grafických primitiv.

Spojitý LOD se také dá využít pro zobrazování objektu načítaného například přes internet, kdy se po přenesení části souboru zobrazí objekt v nejhorším rozlišení a to se pak postupně zlepšuje, jak se přenáší další části souboru.

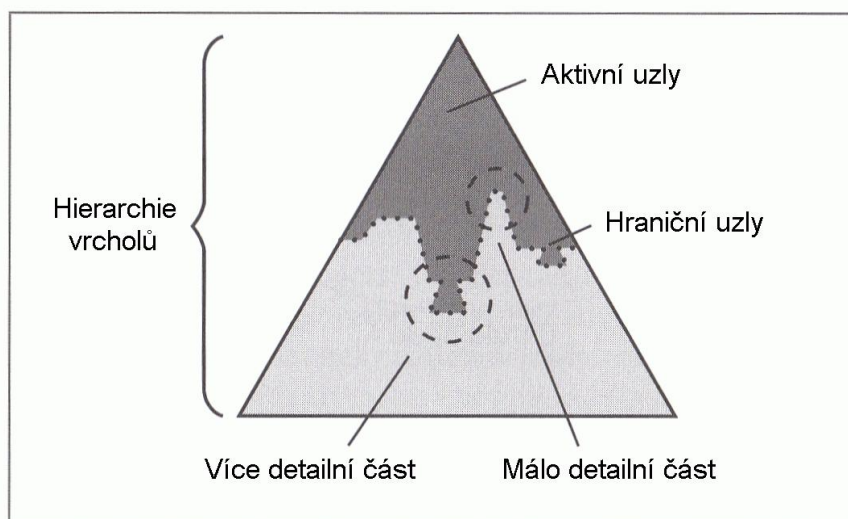
Spojitý LOD je stejně jako diskrétní pohledově nezávislý.

3.1.3 Pohledově závislý LOD

Tento typ LODu závisí na aktuálním směru pohledu na zobrazovaný objekt. Bližší části objektu se zobrazují detailněji než vzdálenější nebo obrysy objektu se zobrazují ve vyšším rozlišení než vnitřek. Body nebo polygony se tak využijí v těch částech objektu, kde jsou potřeba.

Pohledově závislý LOD využívá pro zobrazování hierarchii vrcholů („vertex hierarchy“), která je vygenerována offline při preprocessingu. Tato stromová hierarchie vzniká opakovaným použitím zjednodušovacích operátorů na vrcholy či body objektu. Listy obsahují přímo původní vrcholy či body, vnitřní uzly obsahují vhodnou reprezentaci sjednocení vrcholů pod nimi – například vážený průměr nebo nejvýznamnější vrchol.

Při zobrazování objektu se tato hierarchie prochází podél současné hranice, tj. prochází se aktuálně zobrazované vrcholy a ty jsou testovány podle zvolených kritérií (viz Obrázek 3.1). Podle výsledků se hranice posunuje nahoru – dojde tedy ke zjednodušení této části objektu, nebo dolů – tato část pak bude zobrazena detailněji.



Obrázek 3.1 Schematické znázornění hierarchie vrcholů. Hraniční uzly definují zjednodušení objektu. Zjemnění části objektu posouvá hranici dolů, naopak méně detailní zobrazení ji posouvá nahoru. Vytvořeno podle obrázku v [18].

Vyhodnocování vrcholů tvořících hranici a samotná struktura způsobuje, že tato metoda je náročnější na čas a paměť než obě předchozí metody.

3.2 Generování LOD

Generování LODu pro objekt reprezentovaný body probíhá pomocí downsamplingu na body či splaty objektu (viz paragraf 2.4).

Při použití iterativního algoritmu či hierarchického shlukování vzniká postupnou aplikací daného operátoru podle vybrané chybové metriky strom – binární, octree apod. podle použité datové struktury a operátoru. Tento strom se pak využívá jako hierarchie vrcholů pro pohledově závislý LOD nebo se z něj využije jen část pro vygenerování diskrétního či spojitého LODu objektu.

Při použití optimalizovaného splat-decimation algoritmu strom nevzniká, algoritmus generuje pouze jeden LOD podle požadovaného počtu bodů. Tato metoda se tedy dá využít pouze pro diskrétní či spojitý LOD, pro pohledově závislý LOD není vhodná.

3.3 Kritéria pro výběr diskrétního LOD

Při zobrazování objektu pomocí diskrétního nebo spojitého LODu se pro každý snímek provádí výběr vhodné verze objektu. Tento výběr může záviset na vzdálenosti objektu od pozorovatele, jeho velikosti na obrazovce nebo třeba na prioritě. Je také možné použít několik kritérií najednou.

3.3.1 Vzdálenost

Při použití kritéria vzdálenosti se každé úrovni objektu přiřadí rozsah vzdáleností, ve které se má pro objekt použít tato úroveň detailu. Pro objekt se spočítá vzdálenost od pozorovatele a podle ní se zobrazí příslušný LOD objektu.

Vzhledem k tomu, že objekt je tvořen mnoha body, je nutné zvolit jeden bod, jehož vzdálenost od pozorovatele se bude počítat a používat jako vzdálenost celého objektu. Tímto bodem může být například centroid objektu nebo může být zvolen při vytváření objektu designérem anebo se může počítat vzdálenost vždy k bodu, který je nejbližší k pozorovateli.

Nevýhodou je, že při změně měřítka objektu, při změně rozlišení displeje nebo při změně rozsahu zorného pole neplatí původně nastavené vzdálenosti (prahy, „thresholds“). Také aktuální vzdálenost je závislá na orientaci objektu.

3.3.2 Plocha průmětu

Dalším možným kritériem je plocha průmětu objektu na obrazovce. Verze LODu objektu se opět mění podle zadaných prahů.

Výhodou tohoto přístupu je, že, na rozdíl od vzdálenosti, nezávisí na škálování objektu, rozlišení displeje atd. Není také potřeba vybírat jeden bod, pro výpočet je použit celý objekt.

Toto kritérium je náročnější na výpočet, provádí se projekce souřadnic objektu. Pro zjednodušení výpočtu se nemusí počítat přímo s objektem, ale například jen s obalovým kvádrem („bounding box“) nebo koulí („bounding sphere“) – pak se počítá projekce pouze několika bodů pro odhad plochy pokrytí. Obalový kvádr není invariantní vůči rotaci, takže se musí přepočítávat častěji než obalová koule. Obalová koule poskytuje zase o něco horší odhad.

3.3.3 Priorita

Objekty je také možné zjednodušovat podle priority – některé objekty ve scéně jsou důležitější než jiné, ty se pak zjednodušují jako poslední nebo se případně nezjednodušují vůbec. Příkladem objektu, který by se neměl zjednodušovat, je reprezentace ruky ve virtuální realitě.

3.4 Kritéria pro pohledově závislý LOD

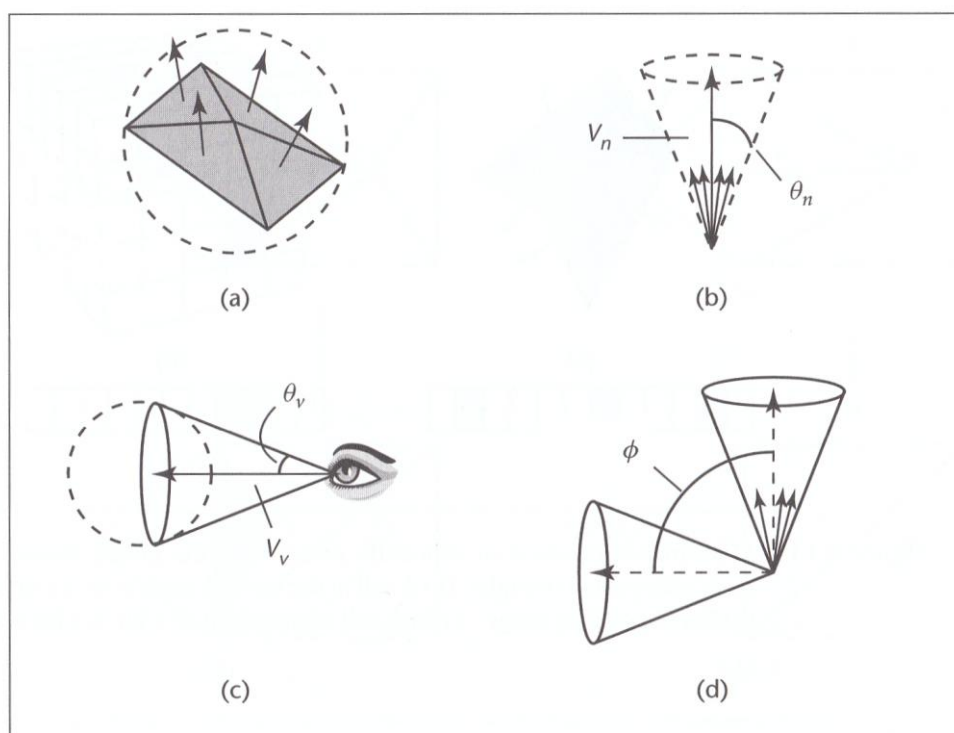
Mezi možná kritéria patří, podobně jako u diskrétního LODu, plocha průmětu uzlu, dále například osvětlení (při velkém gradientu osvětlení se tato část objektu bude zobrazovat detailněji [35]), viditelnost [5] nebo pohyb objektu [6] a silueta.

3.4.1 Plocha průřezu

Toto kritérium funguje podobně jako u diskrétního LODu s tím rozdílem, že se neprovádí projekce celého objektu, ale jen části pokryté právě testovaným uzlem [17]. Spočítá se projekce obalové koule a pokud je menší než daný práh, tak se hranice posune nahoru a naopak.

3.4.2 Silueta

Test na siluetu využívá předpočítaný kužel normál v každém uzlu („normal cone“) [17]. Porovnává se úhel, který svírá průměrná normála v kuželu pro daný uzel a vektor zorného kuželu („viewing cone“, viz Obrázek 3.2). Uzel může být klasifikován jako odvrácený, přivrácený nebo ležící v siluetě. Uzly ležící v siluetě objektu se budou zobrazovat detailněji než uzly ležící uprostřed.



Obrázek 3.2 Testování siluety pomocí kuželu normál. (a) Uzel náležející čtyřem trojúhelníkům. Šipky reprezentují normály, čárkovaný kruh reprezentuje obalovou kouli uzlu. (b) Kužel normál pro uzel reprezentovaný průměrnou normálou V_n a úhlem θ_n . (c) Zorný kužel těsně obsahuje obalovou kouli uzlu, s vektorem V_v vedoucím od pozorovatele do středu koule a úhlem θ_v . (d) Vektory V_n a V_v svírají úhel Φ . Porovnáváním úhlů θ_n , θ_v a Φ může být uzel klasifikován jako přivrácený, odvrácený nebo ležící v siluetě. Převzato z [18].

3.5 Vlivy lidského vnímání

Výběr verze LODu objektu také může záviset na dalších faktorech – pohybující se objekty se mohou zobrazovat v jednodušší verzi, také objekty na okraji periferního vidění se mohou zobrazit jednodušeji.

Tato kritéria (nazývaná „perceptual factors“) jsou založeny na směru pohledu uživatele. Pro výběr LODu se využívá znalost toho směru.

K plnému využití těchto kritérií je potřeba eye-tracking nebo head-tracking systém, v případě, že tento systém není k dispozici, se obvykle předpokládá, že uživatel se dívá do středu obrazovky. Ne vždy je tento předpoklad správný a dostačující: pohybující se objekt přitahuje pozornost a uživatel pak takový objekt sleduje a jeho zjednodušení není na místě.

Směr pohledu lze také v případě, kdy je k dispozici jen head-tracking systém, odhadnout podle směru otočení hlavy – směr pohledu se liší maximálně o 15 stupňů.

Pokud není k dispozici žádný systém pro detekci směru pohledu, lze pro scénu odhadnout, kam se nejspíš uživatel dívá a co sleduje. Pozornost přitahují jasné barvy, pohyb ve statické scéně, objekty či postavy vstupující do scény. Jinak uživatel sleduje popředí scény, obličej postavy nebo se dívá stejným směrem jako ostatní postavy ve scéně apod.

Aplikace také může LOD využívat k přitáhnutí pozornosti uživatele k vybraným objektům, nedůležité objekty se zobrazí rozmazané v nižším rozlišení a tak nepřitahují pozornost [15].

3.5.1 Excentricita

Úroveň detailu objektu lze měnit podle jeho úhlové vzdálenosti od směru pohledu pozorovatele. Pokud je větší, objekt se může zobrazit jednodušeji, naopak objekty blízko směru pohledu se zobrazí detailně. Toto kritérium je založeno na vlastnostech lidského zraku, periferním viděním člověk vnímá méně detailů.

Pohledově závislý LOD je při využívání toho kritéria výhodnější, části jednoho objektu ve směru pohledu vykreslí detailně a části mimo směr pohledu méně detailně. Diskrétní LOD musí celý objekt zobrazit detailně, i když ve směru uživatelského pohledu je jen jeho část.

3.5.2 Rychlost

Další kritérium využívá relativní rychlost objektu vůči pohledu pozorovatele. Pokud pozorovatel sleduje pohybující se objekt na nehybném pozadí, potom se objekt kreslí

detailně a pozadí se vykreslí v nižším rozlišení. Naopak pokud uživatel sleduje pozadí, tak se renderuje detailně a pohybující se objekt méně detailně.

V případě, že eye-tracking nebo head-tracking není k dispozici, je možné rychlost objektu aproximovat jeho rychlostí pohybu po displeji.

3.5.3 Hloubka

Kritérium hloubky mění úroveň detailu objektu podle toho, kam je zaostřen pohled pozorovatele. Objekty, na které není zaostřeno, se jeví rozmazané, a mohou se proto zobrazovat méně detailně.

Při počítání se porovnává vzdálenost objektu od pozorovatele a vzdálenost, kam je zaostřen jeho pohled. Pokud je objekt mimo zaostřenou vzdálenost, zobrazuje se méně detailně.

Toto kritérium je vhodné spíše pro stereoskopické displeje, kde se obraz generuje pro každé oko zvlášť.

3.6 Další techniky

Při častém přepínání mezi sousedními LODy dochází k „blikání“ objektu, kdy se objevují a mizí drobné detaily („popping“). Pro redukci tohoto jevu, ke kterému může dojít při pohybu okolo objektu v „kritické“ vzdálenosti, se dá použít například hystereze nebo alpha blending.

3.6.1 Hystereze

Jedním ze způsobu, jak redukovat „blikání“ objektu, je použití hystereze. To znamená, že na jednodušší verzi objektu se bude přepínat ještě o něco dále než je nastavený práh, na detailnější verzi se bude naopak přepínat až o něco blíže. Vznikne tedy rozsah vzdáleností, kdy se mohou zobrazovat obě verze – záleží na tom, zda se pozorovatel k objektu přibližuje nebo vzdaluje.

3.6.2 Alpha blending

Další způsob, jak zlepšit přechody mezi jednotlivými LODy, je použití alpha blendingu. Každému LODu se přiřadí průhlednost či neprůhlednost (alpha) – LOD, ze kterého se přepíná, bude na začátku neprůhledný, naopak LOD, na který se přepíná, bude průhledný. Dále se zvolí rozsah, ve kterém se budou oba LODy prolínat, se středem ve vzdálenosti, kde se LODy přepínají. V tomto rozsahu se budou vykreslovat oba LODy, jeden s alpha od nuly do jedné, druhý naopak. Alpha se v intervalu lineárně interpoluje.

Vzhledem k tomu, že v daném intervalu se kreslí objekt dvakrát, je potřeba volit interval co nejkratší.

Prolínání se dá namísto v prostoru provádět i v čase, blending se rozloží na několik snímků, i když se objekt vůči pozorovateli nehýbe.

3.7 Budget-based simplification

Tato metoda je založená na hierarchii vrcholů pro pohledově závislý LOD objektu. Cílem je minimalizovat chybu určenou zvolenými kritérii pro testování hraničních uzlů a zůstat v daném rozmezí zobrazovaných vrcholů [17].

Pro uzly tvořící aktuální hranici se počítá chyba a uzel s největší chybou se nahradí svými následovníky, pokud přitom není překročen maximální požadovaný počet vrcholů.

3.8 Globální zjednodušování

Pokud se celá scéna bude považovat za jeden objekt, může se scéna zjednodušovat jako jeden celek, místo aby se zjednodušoval každý objekt zvlášť. Při globálním zjednodušování pak dochází k agregaci malých objektů a ty se poté reprezentují jedním LODem namísto několika různými.

Nevýhodou tohoto přístupu je výpočetní náročnost, kdy se neustále prochází hierarchická struktura vrcholů reprezentujících celou scénu.

3.9 Konstantní snímková frekvence

Jednou z aplikací, kde se využívá LOD, jsou „fixed frame rate schedulers“ – metody pro udržení konstantní snímkové frekvence.

Jednoduchý systém, který nemá žádná omezení na snímkovou frekvenci, kreslí každý snímek tak dlouho, jak je potřeba. Pokud scéna obsahuje jen několik jednoduchých předmětů, frekvence je vysoká, naopak scéna s velkým počtem složitějších objektů se bude vykreslovat dlouho a výsledkem může být velmi proměnlivá frekvence. To pak může způsobovat problémy například s pohybujícími se objekty, kdy se špatně odhaduje jejich rychlost apod. Ve většině aplikací je proto vhodné udržovat stabilní snímkovou frekvenci.

Zachovávat snímkovou frekvenci je možné dvěma základními způsoby - podle doby vykreslování předchozího snímku se upraví složitost zobrazovaných předmětů v následujícím snímku, tzv. reaktivní plánování, nebo se odhadne složitost aktuálně vykreslovaného snímku a podle toho se mění LOD objektů, tzv. prediktivní plánování.

3.9.1 Reaktivní plánování

Reaktivní plánovač sleduje dobu vykreslení posledního snímku a podle toho upravuje LODy zobrazovaných objektů. Pokud se předchozí snímek renderoval příliš dlouho, tak

se objekty zjednodušují, pokud byl dokončen před vyhrazeným časem, objekty se zobrazí detailněji.

Při tomto způsobu práce není zaručena omezená frekvence, LODy objektů se jednoduše upravují podle toho, jestli byl snímek dokončen v požadovaném čase. Jednoduchým řešením, jak frekvenci omezit, je ukončit renderování snímku, pokud byl dosažen deadline. Ovšem tento přístup může vést k tomu, že se některé objekty nevykreslí celé a mohou proto obsahovat různé díry, nebo chybí úplně.

Také co nejrychlejší přizpůsobování vede k příliš rychlým změnám v obraze, proto je lepší postupné přizpůsobování detailů s využitím hystereze [11].

3.9.2 Prediktivní plánování

Prediktivní plánovač odhaduje složitost aktuálně kresleného snímku a podle toho vybírá LODy jednotlivých objektů, aby nepřekročil čas vymezený na vykreslení snímku. Tato metoda je složitější než předchozí a je potřeba dobře modelovat, jak dlouho se bude scéna vykreslovat na daném hardwaru. Při tom se pochopitelně musí započítat i čas výpočtu odhadu. Příklady této techniky pro diskrétní LOD lze najít v [7, 19, 20], pro spojitý LOD v [8].

Tato metoda zaručuje, že snímková frekvence bude v daném rozmezí, a to i pro scény, které se v následujících snímcích hodně liší.

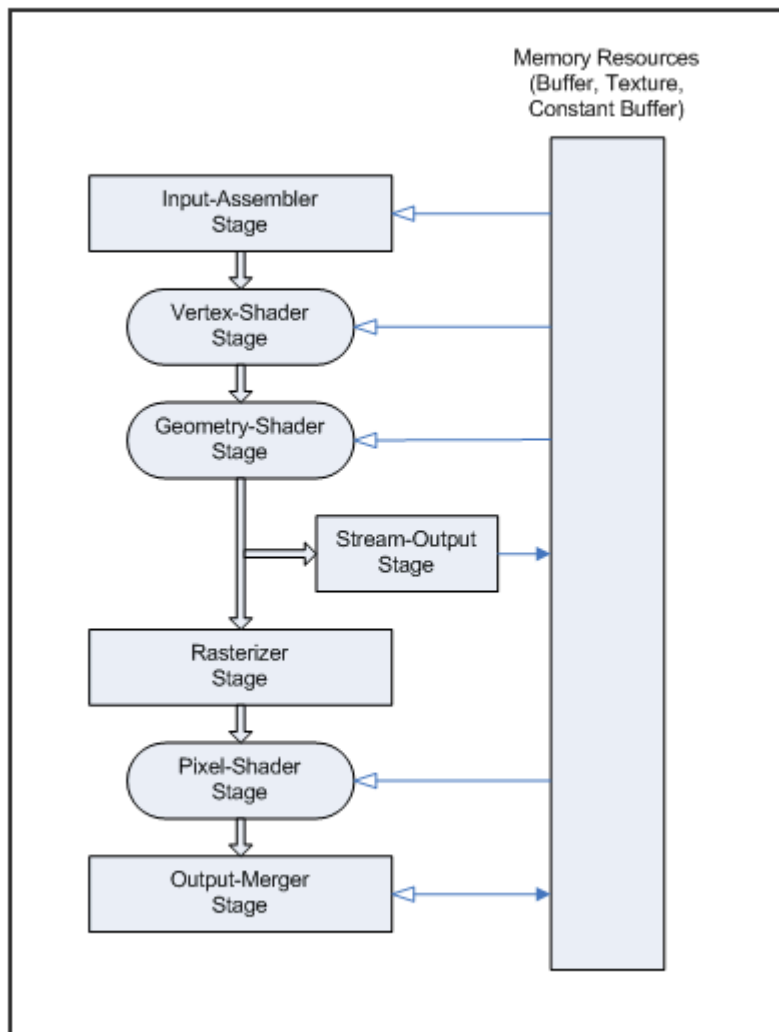
4 DirectX 10

V této kapitole je představen Microsoft DirectX 10 a Shader Model 4.0, který je využit při implementaci. V první části je popsána grafická pipeline, v druhé části je shrnuto několik důležitých změn proti DirectX 9.

Podrobnosti jsou dostupné na webových stránkách [4], přehledné informace jsou také v různých prezentacích z konferencí, například [32].

4.1 Grafická pipeline

Grafická pipeline se skládá z několika programovatelných částí („programmable stages“) a několika fixních částí („fixed-function stages“) – viz Obrázek 4.1. Dále jsou jednotlivé části stručně popsány.



Obrázek 4.1 Grafická pipeline DirectX 10. Převzato z [4].

4.1.1 Input Assembler

Input assembler je fixní část pipeline. Jeho úkolem je načíst a zkonvertovat data ze vstupních streamů do kanonického tvaru. Vstupní streamy jsou napojeny na vertex buffery, těch může být až 8. Vrcholy se z vertex bufferů načítají sekvenčně, pokud je ale specifikován index buffer, pak se vrcholy z vertex bufferů čtou podle pořadí daného index bufferem. Index buffer je jen jeden společný pro všechny vertex buffery.

Input assembler také podporuje „*instancing*“ – umí replikovat jeden objekt n -krát. S vrcholy, které tvoří objekt, je asociován počet opakování objektu n a každý vrchol dostane identifikátor (ID) spojený s instancí objektu. K těmto ID lze přistupovat v programovatelných částech pipeline a počítat podle nich transformace a další hodnoty vztažené k instanci objektu.

4.1.2 Vertex Shader

Vertex shader je programovatelná část pipeline, většinou se využívá k transformacím souřadnic vrcholů ze souřadnic modelu („*object space*“) do světových souřadnic („*world space*“) a ořezávacích souřadnic („*clip space*“). Lze v něm také počítat například osvětlení ve vrcholech. Vertex shader má přístup k bufferům v paměti (texturám), těch může být až 128, a až k 16 konstantním bufferům.

Vertex shader zpracovává vždy jeden vrchol a jeho výstupem je opět právě jeden vrchol. Nelze v něm změnit počet zpracovávaných vrcholů.

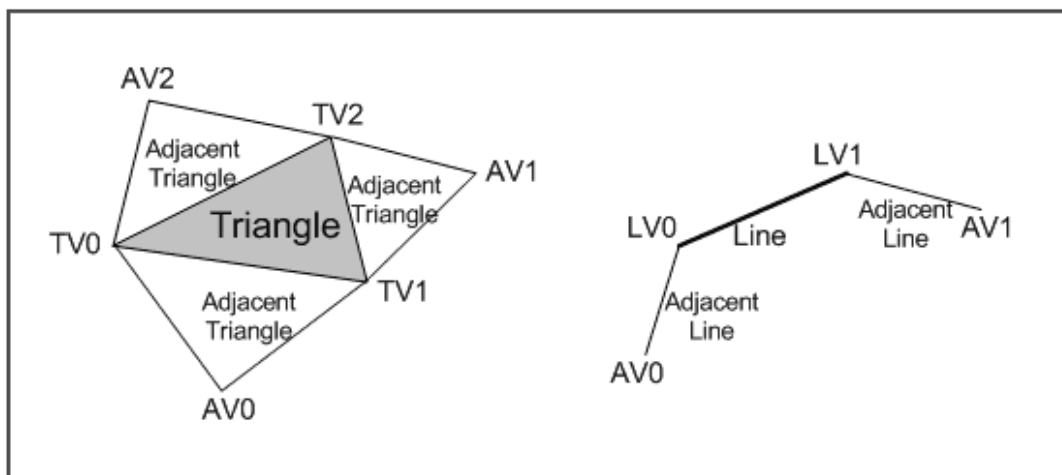
4.1.3 Geometry Shader

Geometry shader je nová část pipeline, patří mezi programovatelné části. Jeho vstupem je jedno primitivum – bod, úsečka nebo trojúhelník. Výstup tvoří vrcholy primitiv, těchto primitiv může být i více než jedno, ale také nemusí být žádné. Typ vstupních a výstupních primitiv může být různý, ale je dán parametry shaderu, nelze ho dynamicky měnit. Shader také nemusí nijak měnit počet primitiv, může pouze dopočítávat atributy, které závisí na více vrcholech daného primitiva.

Vstupní primitiva mohou být zpracovávána včetně sousedních vrcholů, trojúhelník má tři sousední vrcholy, úsečka dva (viz Obrázek 4.2). Tyto sousední vrcholy jsou specifikovány ve vertex bufferu.

Geometry shader má stejně jako vertex shader přístup ke konstantním bufferům a k teksturám v paměti.

Geometry shader je volitelnou součástí pipeline, nemusí být tedy definován.



Obrázek 4.2 Trojúhelník a úsečka se sousedními vrcholy. TV – vrchol trojúhelníka, LV – vrchol úsečky, AV - sousední vrchol. Převzato z [4].

4.1.4 Stream Output

Stream output je také nová část pipeline, tato část je fixní. Slouží ke kopírování vybraných dat vrcholů do paměti, kopírovaná data jsou výstupem geometry shaderu.

Buffer, do kterého je zapsán výstup geometry shaderu přes stream output, lze připojit v dalším průchodu jako vstupní vertex buffer. Vzhledem k neznámému počtu primitiv, které může geometry shader generovat, se pak pro vykreslení používá jiná funkce než při standardním renderování vertex bufferu.

4.1.5 Rasterization Stage

Rasterization stage je fixní část, která provádí ořezávání („clipping“), perspektivní dělení a další operace a hlavně generuje z primitiv fragmenty. Vstupem jsou vrcholy a atributy jednoho grafického primitiva, výstupem jsou fragmenty. Hodnoty atributů fragmentů jsou interpolované z hodnot ve vrcholech.

4.1.6 Pixel Shader

Pixel shader je programovatelná část pipeline. V pixel shaderu se provádí výpočet osvětlení. Vstupem je fragment a jeho vlastnosti, výstupem je barva fragmentu (RGB nebo RGBA) případně hloubka. Fragment může být také zahozen. Pixel shader nemůže měnit pozici fragmentu na obrazovce (souřadnice x, y), pouze hloubku (souřadnici z).

Také pixel shader má, podobně jako vertex shader a geometry shader, přístup ke konstantním bufferům a k texturám v paměti.

4.1.7 Output Merger

Poslední část pipeline je Output Merger, který provádí depth a stencil testy a color blending. Output Merger pracuje s fragmentem. Může na něj být napojeno až 8 výstupních bufferů („*render targets*“).

4.2 Shrnutí změn

Jednou z důležitých změn je rozšíření grafické pipeline. Do pipeline, jak bylo popsáno výše, byly přidány nové části – geometry shader pracující s celými primitivy a stream output umožňující výstup dat z geometry shaderu do bufferu a následného použití bufferu jako vstupního. To umožňuje například generovat částice v geometry shaderu bez použití CPU.

Shadery mají nově společné jádro – vertex shader i pixel shader mají stejnou instrukční sadu, stejný přístup ke zdrojům (buffery v paměti, ...).

Dále byly zavedeny „*constant buffers*“ – parametry shaderů se rozdělí do bufferů podle četnosti jejich změn. To vede k nižším přenosům dat mezi CPU a GPU. Při změně se na GPU přenáší a aktualizují jen ty parametry, které jsou opravdu potřeba.

Také byla zobecněna práce se zdroji („*resources*“ – textury, apod.). Zdroje jsou většinou beztypové, pro práci se zdrojem je potřeba definovat „pohled“ („*ShadeResourceView*“), který interpretuje data v bufferu a provádí navázání k vybrané části pipeline. Několik „pohledů“ může využívat stejný buffer, každý si interpretuje data podle vlastních potřeb.

5 Progressive Splatting

Algoritmus „*Progressive Splatting*“ navržený v [34] používá standardní hladový přístup pro výběr aplikovaného operátoru, pro odhad chyby a vytvoření nového splatu využívá celou geometrii obou splatů. Splatem je elipsa v libovolné poloze v prostoru. Algoritmus je odvozený z algoritmu „progressive meshes“ pro trojúhelníky [12]. Použité chybové metriky jsou zobecněny z metrik v [3].

5.1 Algoritmus

Vstupem algoritmu je množina bodů reprezentujících objekt, dále se pracuje s celými splaty. V prvním kroku je tedy pro každý daný bod vytvořen iniciální splat pomocí jeho k nejbližších sousedů. Potom jsou všechny „splat merge“ operátory seříděny do fronty podle chyby, první je operátor s nejmenší chybou. Chyba se měří podle vybrané chybové metriky. Operátory se postupně aplikují na množinu splatů. Operátor sloučí dva vybrané splaty do jednoho, poté se přepočítá chyba u operátorů, které pracují s jedním z odstraněných splatů, a zařadí se znovu do fronty nebo se odstraní, pokud dojde k jejich degeneraci.

Výstupem algoritmu je buď požadovaný počet splatů reprezentujících zpracovávaný objekt nebo celý strom splatů, pokud se zaznamenávají provedené operátory a slučování dojde až k jednomu výslednému splatu.

5.1.1 Inicializace

Vstupem je, jak bylo popsáno výše, množina bodů $P = \{ \mathbf{p}_i \}$ reprezentující povrch objektu a případně další vlastnosti bodů (normála, barva atd.). Pro každý bod \mathbf{p}_i se vytvoří splat T_i , což je obecná 3D elipsa, se středem \mathbf{c}_i , jednotkovým normálovým vektorem \mathbf{n}_i a vektory \mathbf{u}_i a \mathbf{v}_i , které definují hlavní a vedlejší osu elipsy. Vektory \mathbf{u}_i a \mathbf{v}_i nejsou jednotkové.

Pro analýzu lokálních změn povrchu a pro vytvoření iniciálních splatů T_i , se pro každý bod \mathbf{p}_i spočítá jeho k nejbližších sousedů - množina $N_k(\mathbf{p}_i)$. Střed splatu \mathbf{c}_i bude v bodě \mathbf{p}_i (tj. $\mathbf{c}_i = \mathbf{p}_i$). Dále se spočte normálový vektor \mathbf{n}_i – určí se rovina nejmenších čtverců L („least square plane“) pro \mathbf{p}_i a $N_k(\mathbf{p}_i)$ a její normála se použije jako normála splatu T_i . Iniciální splaty se vytváří jako kruhy, vektory \mathbf{u}_i a \mathbf{v}_i tedy mohou být libovolné ortogonální vektory rovnoběžné s rovinou L se stejnou délkou r :

$$r = \max \left\| (\mathbf{p}_j - \mathbf{c}_i) - n_i^T (\mathbf{p}_j - \mathbf{c}_i) \mathbf{n}_i \right\| \quad (1)$$

pro všechny $\mathbf{p}_j \in N_k(\mathbf{p}_i)$.

Zároveň se vytváří graf $G = (P, E)$, kde P je množina vstupních bodů a hrana (i, j) patří do E , právě když \mathbf{p}_j je v $N_k(\mathbf{p}_i)$. Graf bude podporovat dynamickou topologii během slučování splatů.

5.1.2 Operátor

Operátorem Φ je „splat merge“ operátor, analogický k operátoru kontrakce hrany u trojúhelníkových sítí.

Operátor Φ sloučí dva splaty T_l a T_r , pokud v grafu G existuje hrana e taková, že $e = (l, r)$, tj. v grafu jsou body \mathbf{p}_l a \mathbf{p}_r spojeny hranou. Po aplikování operátoru Φ vznikne nový splat T_m se středem \mathbf{c}_m . Z množiny P se odstraní body \mathbf{p}_l a \mathbf{p}_r a jsou nahrazeny novým bodem $\mathbf{p}_m = \mathbf{c}_m$. Bod \mathbf{p}_m získá jako sousedy všechny původní sousedy bodů \mathbf{p}_l a \mathbf{p}_r . Hrana e je z grafu odstraněna a s ní také případné další degenerované hrany.

5.1.3 Chybové metriky

V článku [34] jsou navrženy dvě chybové metriky pro odhad odchylky vzdálenosti a odchylky normály. Na zvolené metrice je operátor a tedy i algoritmus výrazně závislý.

5.1.3.1 L^2 metrika

Tato metrika je založena na Euklidovské vzdálenosti. Chyba způsobená sloučením dvou splatů se počítá vzhledem k původním bodům, proto se pro každý splat T_i uchovává pole indexů původních bodů $\{f_i\}$. Na začátku je toto pole inicializováno jedním indexem $\{i\}$, vztahujícím se k bodu \mathbf{p}_i . Pole indexů nového splatu, vzniklého sloučením dvou splatů, se vytvoří spojením jejich polí indexů.

Pro operátor Φ , který sloučí splaty T_l a T_r do nového splatu T_m , se chyba definuje následovně:

$$\varepsilon_\Phi = \|e\| \cdot \sum_{f \in \{f_m\}} \left| \text{dist}(\mathbf{p}_f, T_m) \right|^2, \quad \{f_m\} = \{f_l\} + \{f_r\}. \quad (2)$$

Metrika je vážená délkou hrany e , tj. vzdáleností mezi středy dvou splatů, tak je penalizováno spojování dvou vzdálených splatů, čímž by jinak vznikaly příliš velké splaty.

Určení nového splatu

Určení parametrů nového splatu – jeho středu, normály a os, probíhá pomocí „Principal Component Analysis“ ([14]) na množině bodů $P_m = \{\mathbf{p}_f\}$, $f \in \{f_m\}$ přímo v 3D prostoru. Tím se získá průměrný bod $\bar{\mathbf{p}}$, tři reálná vlastní čísla $\lambda_1 \geq \lambda_2 \geq \lambda_3$ a vlastní vektory \mathbf{v}_1 , \mathbf{v}_2 , \mathbf{v}_3 . Střed nového splatu bude průměrný bod $\mathbf{c}_m = \bar{\mathbf{p}}$, normála $\mathbf{n}_m = \mathbf{v}_3$ a jako osy elipsy \mathbf{u}_m a \mathbf{v}_m se použijí \mathbf{v}_1 a \mathbf{v}_2 , s poměrem jejich délek $\sqrt{\lambda_1 / \lambda_2}$. Výsledná

délka os je škálována tak, aby splat pokrýval všechny body P_m ve 2D prostoru při jejich projekci na rovinu splatu.

5.1.3.2 $L^{2,1}$ metrika

$L^{2,1}$ metrika měří odchylku normály. Výpočet chyby podle této metriky je jednodušší, není potřeba uchovávat pole indexů.

Pro operátor Φ slučující splaty T_l a T_r je chyba dána vzorcem:

$$\varepsilon_\Phi = \|e\| \cdot |T_l| \cdot |T_r| \cdot \|n_l - n_r\|^2, \quad (3)$$

kde $|T_l|$ a $|T_r|$ jsou plochy těchto splatů a $\|e\|$ je délka hrany (l, r).

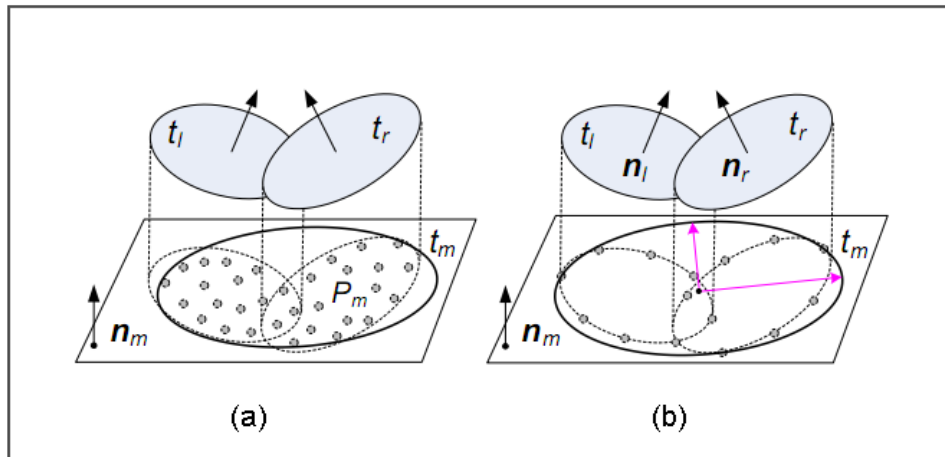
Střed nového splatu je definován následovně:

$$c_m = \frac{|T_l| \cdot c_l + |T_r| \cdot c_r}{|T_l| + |T_r|} \quad (4)$$

a normála

$$n_m = \frac{|T_l| \cdot n_l + |T_r| \cdot n_r}{|T_l| + |T_r|}. \quad (5)$$

Osy splatu T_m se počítají podobně jako u L^2 metriky, ovšem místo projekce bodů P_m se uniformně vzorkují hranice obou elips. Poté se provede projekce takto získaných bodů na rovinu splatu T_m pro zjištění os a jejich délek.



Obrázek 5.1 „Splat merge“ operátor pro L^2 metriku (a) a $L^{2,1}$ metriku (b). Převzato z [34].

5.1.3.3 Srovnání metrik

V článku [34] je také obsaženo srovnání výsledné distribuce splatů získaných podle uvedených metrik. Při výpočtu splatů pomocí L^2 metriky jsou splaty rovnoměrněji distribuované, $L^{2,1}$ metrika zase lépe zachycuje lokální změny povrchu.

Výpočet s $L^{2,1}$ metrikou je také rychlejší než při použití L^2 metriky.

5.2 Implementace

Cílem implementace zvoleného algoritmu je převést část výpočtů na GPU, tím uvolnit CPU pro zpracování jiných částí algoritmu, které na GPU provádět nelze. Výpočet se pak provádí paralelně a ve výsledku by měl proběhnout rychleji. Ideální stavem je plné vytížení CPU i GPU.

Na GPU lze převést výpočet iniciálních splatů a vytvoření nového splatu. Změny v grafu G a ve frontě operátorů na GPU je nutné provádět na CPU.

Důležitá je „synchronizace“ obou procesorů, kdy CPU pošle data GPU a pak provádí další vlastní výpočty. V určitém místě výpočtu potřebuje výsledky výpočtů GPU, pokud by na ně musel čekat příliš často nebo příliš dlouho, je možné, že se tato část výpočtů na GPU nehodí nebo je třeba zmenšit objem dat posílaných na GPU a výpočet na zbylých datech provádět na CPU tak, aby oba výpočty skončily přibližně stejně.

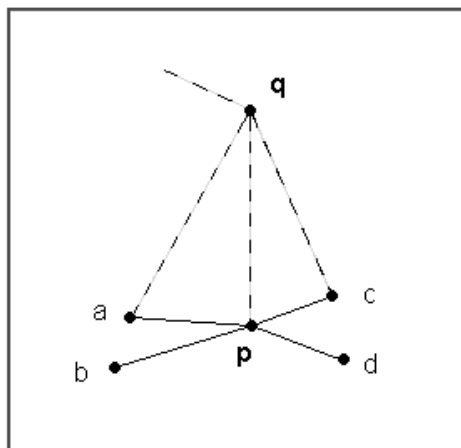
Dále v textu jsou popsány postupy použité při implementaci algoritmu. K výpočtu na GPU se využívá vertex shader a geometry shader; pixel shader u žádného výpočtu není definován. Jedna část této kapitoly je věnována ukázce přenosu výsledků z GPU zpět na CPU, tento přenos totiž musí probíhat přes dva buffery a pomocný pointer na pole.

5.2.1 Inicializace

Vstupní body jsou uloženy v poli, každý bod má souřadnice a seznam svých sousedů, po ukončení inicializace mu přibudou ještě parametry jeho splatu – normálový vektor a osy elipsy.

Ke spočítání k nejbližších sousedů se z bodů postaví kd-strom, body jsou jak v listech tak i ve vnitřních uzlech. Poté se strom prohledává a ke každému bodu se přiřadí seznam jeho nejbližších sousedů – indexů do pole bodů, a zároveň se pro každého souseda vytvoří hrana.

Z těchto hran vzniká graf G , graf je neorientovaný, v seznamu sousedů pro daný bod jsou tedy jak jeho nejbližší sousedi tak i body, jejichž je nejbližším sousedem.



Obrázek 5.2 Bod **p** má za sousedy body *a*, *b*, *c*, *d*. Bod **q** mezi jeho nejbližší sousedy nepatří. Bod **q** mezi nejbližšími sousedy bod **p** má. V seznamu sousedů bodu **p** proto bude i bod **q**.

Po zjištění sousedů se spočítají parametry iniciálních splatů na GPU ve vertex shaderu. Vstupními daty jsou souřadnice bodu a jeho sousedů. Spočítají se vlastní čísla kovarianční matice těchto bodů (velikost matice 3×3), k nejmenšímu číslu se určí vlastní vektor - normálový vektor splatu. K normálovému vektoru se dopočítají dva libovolné kolmé vektory – poloosy elipsy, a určí se velikost těchto vektorů podle vzorce (1).

Dále je potřeba vypočítat chyby operátorů při slučování splatů, tedy ohodnotit všechny hrany v grafu. Výpočet probíhá podle $L^{2,1}$ metriky (vzorec (3)). Poté se z hran postaví halda, která umožňuje jejich rychlé odebírání a změnu ohodnocení.

5.2.2 Aplikace operátoru

Aplikace operátoru se opakuje, dokud je v haldě nějaká hrana. Hrana s nejnižším ohodnocením je z haldy odebrána a provede se sloučení splatů, které jsou spojeny touto hranou.

Při aplikaci „splat-merge“ operátoru se na GPU vytváří nový splat. Geometry shader pracuje se splaty jako s úsečkami („*D3D10_PRIMITIVE_TOPOLOGY_LINELIST*“) – má tedy k dispozici oba původní splaty.

Ve vertex shaderu se provede vzorkování původních splatů, v geometry shaderu se spočte střed a normála nového splatu podle vzorců (4) a (5). Dále se provede projekce vzorků do roviny nového splatu a spočtou se vlastní čísla kovarianční matice těchto vzorků (velikost 2×2). Určí se vlastní vektor k většímu z těchto čísel (jedna poloosa elipsy) a dopočítá se vektor kolmý k normále a k určenému vlastnímu vektoru (druhá poloosa elipsy). Délka os elipsy se vezme nejmenší taková, aby v novém splatu byly obsaženy všechny navzorkované body a zároveň byla délka os v poměru uvedeném v paragrafu 5.1.3.

Zároveň se upravuje graf G – odstraní se hrana, která spojuje slučované splaty. Bod hrany, který má více sousedů, se ponechá, pouze se změní jeho souřadnice a další vlastnosti tak, aby odpovídaly novému splatu. Druhý bod, který má sousedů méně, se označí za neplatný a jeho hrany se přepojí k prvnímu bodu, který teď představuje nový splat. Pokud už taková hrana existuje, je z grafu odstraněna. Neplatné body jsou jednou za čas smazány. Všechny změněné hrany – od obou bodů odstraněné hrany, se znovu ohodnotí a přesunou se v haldě na správné místo.

Z dvou původních splatů se vytvoří nové uzly pro výsledný strom splatů.

5.2.3 Algoritmy pro zpracování hran

V programu je implementováno několik algoritmů pro aplikaci operátorů, tj. zpracování hran. Algoritmus určuje pořadí hran, způsob jejich zpracování a množství dat posílaných na GPU při počítání nových splatů – kolik nových splatů najednou se počítá.

Hladový algoritmus

Hladový algoritmus (v grafech je označen jako „Greedy“) po odebrání minima z haldy vše ihned přepočítá – parametry splatu i ohodnocení změněných hran. Na GPU tedy posílá vždy pouze dva splaty, ze kterých se vytváří jeden nový. Vzhledem k tomu je jeho využití GPU velmi nízké.

Jednoduchý líný algoritmus

Jednoduchý líný algoritmus („Lazy Simple“) mění oproti hladovému algoritmu pouze přístup k přepočítání ohodnocení hran – přepočet se provádí, až když se hrana dostane na vrchol haldy, mezitím je označena jako nepřepočítaná („dirty“). Na GPU se stále posílají jen dva splaty a to ihned při zpracovávání hrany.

Líný algoritmus varianta č. 2

Druhá varianta líného algoritmu („Lazy 2“) odkládá kromě přepočítání hrany i výpočet nového splatu. Při zpracování hrany se vytvoří dva nové uzly stromu splatů, ale v seznamu bodů zůstávají původní data, bod je pouze označen za nepřepočítaný („bad“).

Data se na GPU posílají, když se v minimu objeví nepřepočítaná hrana, kterou nelze ohodnotit bez nových parametrů splatu, nebo když je nepřepočítaných splatů více než daná konstanta. Po poslání dat na GPU se zpracuje další hrana a poté se přečtou výsledky z GPU (pokud nejsou, tak se na ně čeká).

Na GPU se v tomto algoritmu počítá více nových splatů najednou, během práce GPU se zpracuje pouze jedna další hrana.

Líný algoritmus varianta č. 3

Tento algoritmus nečeká, až se nepřepočítaná hrana objeví v minimu, ale kontroluje prvních N hran v haldě (N je daná konstanta). Dokud mezi nimi není nepřepočítaná hrana, tak se jen opravuje graf a označují se hrany a body. Pokud se objeví hrana označená jako

„dirty“, spustí se výpočet na GPU všech dosud nepřepočítaných splatů. Výpočet na GPU se také spouští, pokud je označených splatů více než daná konstanta.

Po poslání dat na GPU se dále odebírají hrany z haldy, dokud je možné je zpracovat bez výsledků GPU anebo dokud není výpočet hotov. Poté se zkontroluje prvních M hran v haldě a pokud jsou mezi nimi označené, tak se přepočítá jejich ohodnocení ($M > N$).

Stejně jako v předchozím algoritmu se na GPU počítá více splatů najednou, navíc se během tohoto výpočtu, pokud je to možné, zpracovávají další hrany.

5.2.4 Výstup

Výsledkem výpočtu je strom splatů, který vzniká postupnou aplikací operátorů. Stromy se budou lišit v závislosti na zvoleném algoritmu.

Objekt reprezentovaný splaty je možné uložit do souboru – uloží se buď jedno patro stromu s počtem splatů, který je nejbližší požadovanému počtu, nebo je možné uložit celý strom.

5.2.5 Přenos dat z GPU na CPU

Přenos dat z GPU na CPU probíhá, jak bylo zmíněno výše, přes dva buffery a pomocný pointer na pole. Výstup z geometry shaderu se posílá do bufferu napojeného na StreamOutput Stage, tento buffer se poté zkopíruje do staging bufferu přístupného pro CPU. Teprve tento staging buffer se mapuje na pointer na pole, ze kterého se čtou data.

Zde jsou uvedeny ukázky z postupu přenosu dat z GPU na CPU, všechny proměnné typu *v_*Buffer* jsou definovány jako pointer *ID3D10Buffer**.

Vytvoření stream output bufferu pro výstup geometry shaderu:

```
D3D10_BUFFER_DESC bdStreamOut;
bdStreamOut.Usage = D3D10_USAGE_DEFAULT;
bdStreamOut.ByteWidth = sizeof (ISOutput_InitSplatValues) * size;
bdStreamOut.BindFlags = D3D10_BIND_STREAM_OUTPUT;
bdStreamOut.CPUAccessFlags = 0;
bdStreamOut.MiscFlags = 0;
hr = g_pd3dDevice->CreateBuffer (&bdStreamOut, NULL,
                                &v_pStreamToBuffer);
```

Vytvoření bufferu přístupného pro CPU:

```
D3D10_BUFFER_DESC bdCPU;
bdCPU.Usage = D3D10_USAGE_STAGING;
bdCPU.ByteWidth = sizeof (ISOutput_InitSplatValues) * size;
bdCPU.BindFlags = 0;
bdCPU.CPUAccessFlags = D3D10_CPU_ACCESS_READ;
bdCPU.MiscFlags = 0;
hr = g_pd3dDevice->CreateBuffer (&bdCPU, NULL,
                                &v_pStageOutputBuffer);
```

Zápis výsledných dat do output bufferu, geometry shader musí být zkompilován s výstupem do StreamOutput Stage:

```
// Set output buffer
ID3D10Buffer * pBuffer[1];
UINT offset [1] = { 0 };
pBuffer [0] = v_pStreamToBuffer;
g_pd3dDevice->SOSetTargets (1, pBuffer, offset);
```

Vypnutí zápisu probíhá zavoláním *SOSetTargets()* s parametrem *NULL*.

Čtení dat z GPU na CPU:

```
// Copy data to stage buffer for CPU access
g_pd3dDevice->CopyResource (v_pStageOutputBuffer,
    v_pStreamToBuffer);

ISOutput_InitSplatValues * pInitValues = NULL;

// Map data to array
HRESULT hr = v_pStageOutputBuffer->Map (D3D10_MAP_READ, 0,
    (void **) &pInitValues);
```

Poté se použije for-cyklus ke zkopírování dat do požadovaného pole. Po ukončení se buffer uvolní použitím funkce *Unmap ()*:

```
v_pStageOutputBuffer->Unmap ();
```

5.2.6 Měření

Měření probíhalo na počítači s Microsoft Windows 7 Ultimate, procesorem Intel Core i7, 3.07 GHz, 6 GB paměti, s grafickou kartou NVIDIA GeForce 8800 GTS.

Měření probíhalo na osmi modelech, byla měřena jak celková doba trvání jednotlivých algoritmů, tak i doba trvání jednotlivých částí algoritmů. Modely *drill shaft*, *bunny*, *dragon* a *happy buddha* jsou z [38], modely *swan* a *angel 1* jsou z [39], model *eg07 dragon* je z [36] a poslední model nazvaný *angel 2* je z [37].

První měřenou částí výpočtu je vytváření kd-stromu, který se v další fázi použije pro vyhledání nejbližších sousedů jednotlivých bodů. Kd-strom je binární, body obsahuje jak v listech, tak i ve vnitřních vrcholech. Ve třetí fázi probíhá vytváření iniciálních splatů – spočtou se normálové vektory a osy splatů. Všechny tyto tři fáze probíhají stejně bez ohledu na zvolený algoritmus pro zpracování hran.

| Model | Počet bodů |
|--------------|------------|
| drill shaft | 1961 |
| bunny | 35947 |
| swan | 113238 |
| angel 1 | 170355 |
| angel 2 | 237018 |
| eg07 dragon | 336017 |
| dragon | 437645 |
| happy buddha | 543652 |

Tabulka 5.1 Seznam modelů použitých pro měření a počet původních bodů modelů. Modely jsou seřazeny podle počtu bodů.

Čtvrtou měřenou fází je slučování splatů – vybírají se hrany a dochází ke slučování splatů podle zvoleného algoritmu pro zpracování hran.

Doba trvání celého výpočtu je také závislá na počtu sousedů, kteří se zjišťují ve druhé fázi – výpočet je možné spustit pro 4 nebo 8 sousedů.

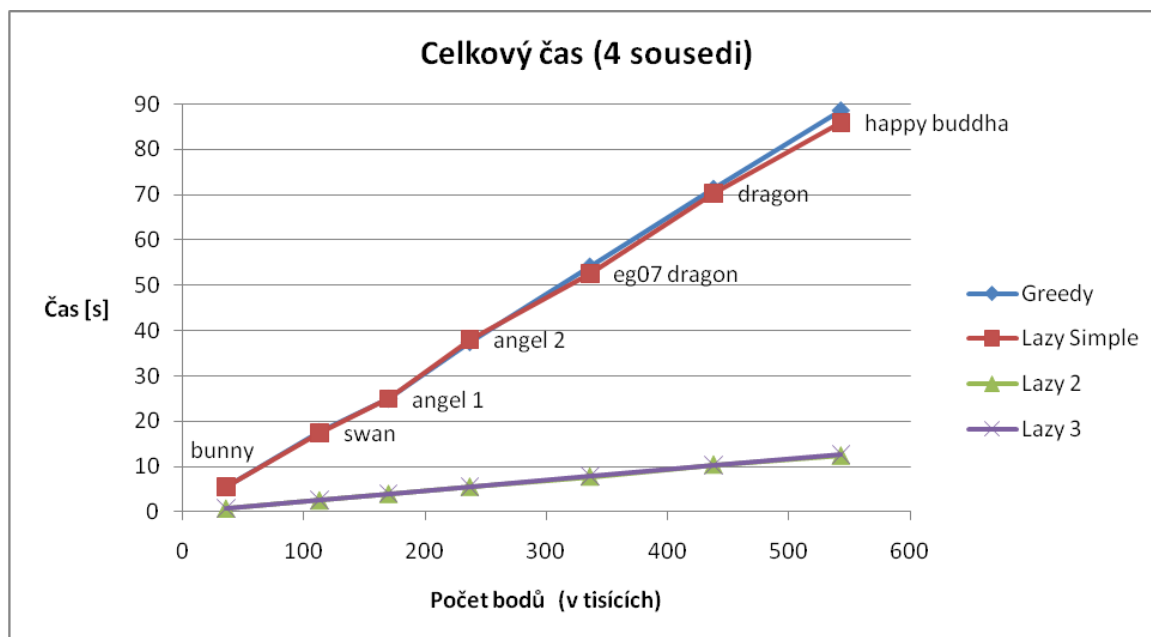
| Měřené fáze výpočtu | |
|------------------------------|---------|
| Vytváření kd-stromu | CPU |
| Hledání nejbližších sousedů | CPU |
| Vytváření iniciálních splatů | GPU |
| Slučování | CPU/GPU |

Tabulka 5.2 Jednotlivé části výpočtu, které byly měřeny.

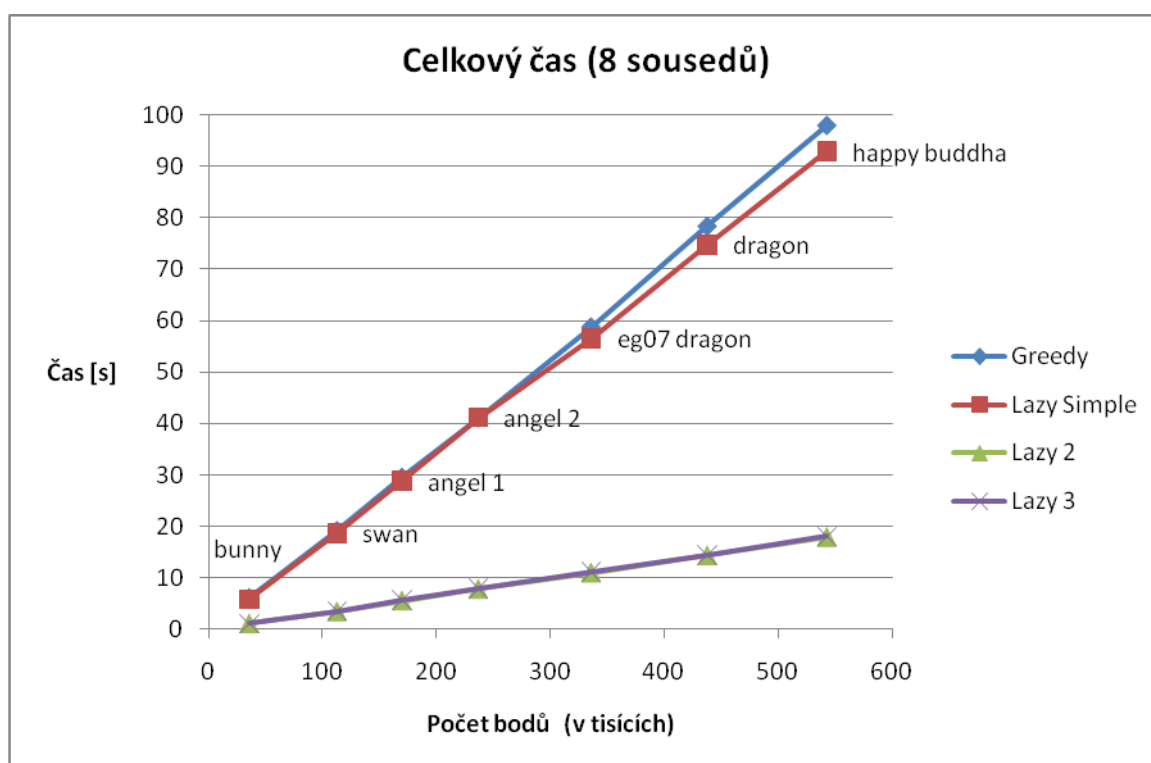
V grafech není zobrazen model *drill shaft*, který obsahuje velmi málo bodů. Všechny naměřené hodnoty k zobrazeným grafům, včetně hodnot pro model *drill shaft*, jsou uvedeny v příloze.

5.2.6.1 Grafy

V prvních dvou grafech (5.1 a 5.2) je zobrazen celkový naměřený čas pro jednotlivé modely a jednotlivé algoritmy. Je zde vidět velký rozdíl mezi hladovým a jednoduchým líným algoritmem a zbylými dvěma variantami líného algoritmu, který je způsoben podstatně lepším využitím GPU druhými dvěma algoritmy.

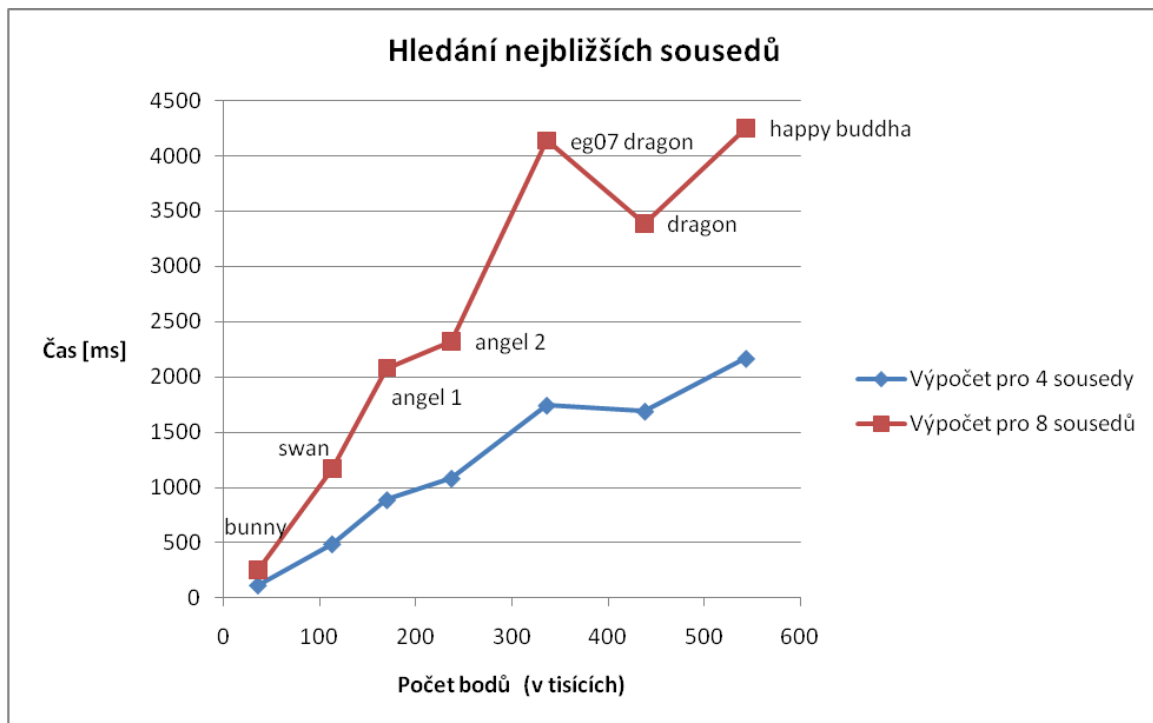


Graf 5.1 Graf celkového trvání výpočtu pro jednotlivé algoritmy v závislosti na počtu bodů modelu. Výpočet probíhal pro 4 sousedy.

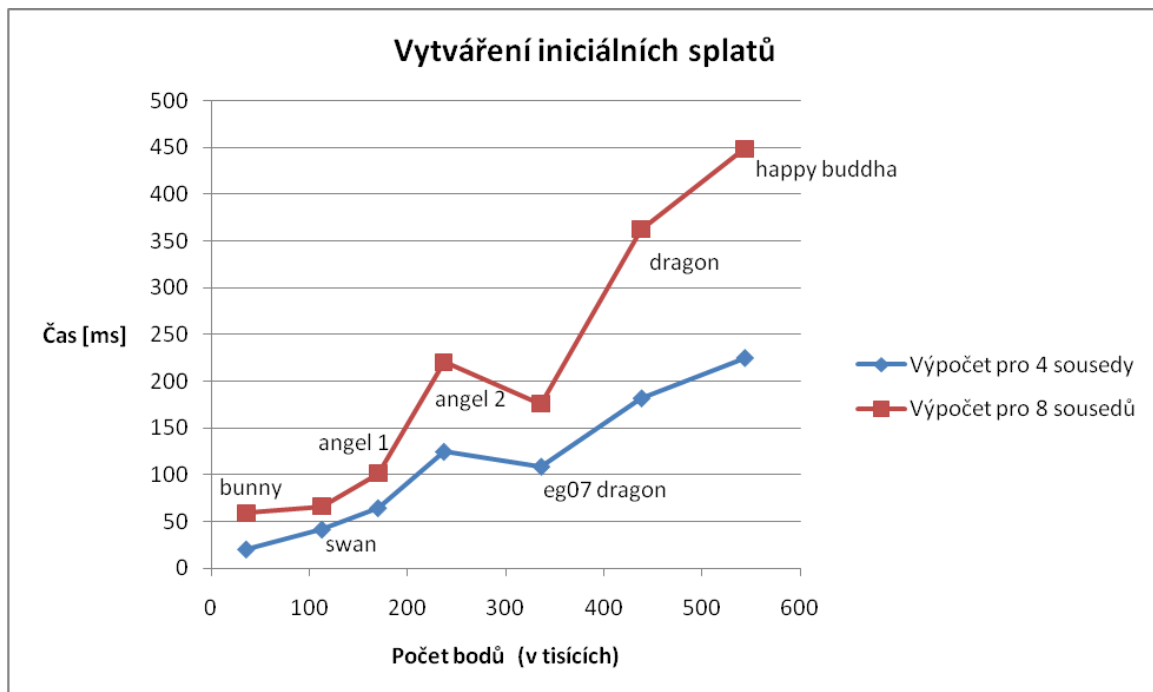


Graf 5.2 Graf celkového trvání výpočtu pro jednotlivé algoritmy v závislosti na počtu bodů modelu. Výpočet probíhal pro 8 sousedů.

V grafech 5.3 a 5.4 je porovnání doby trvání jednotlivých fází výpočtu pro 4 a 8 sousedů. Hledání nejblížešších sousedů trvá model *dragon* kratší dobu než pro model *eg07 dragon*, přestože má více bodů – to je pravděpodobně způsobeno tvarem objektu, u modelu *dragon* se zřejmě prohledává menší část kd-stromu než u druhého modelu.

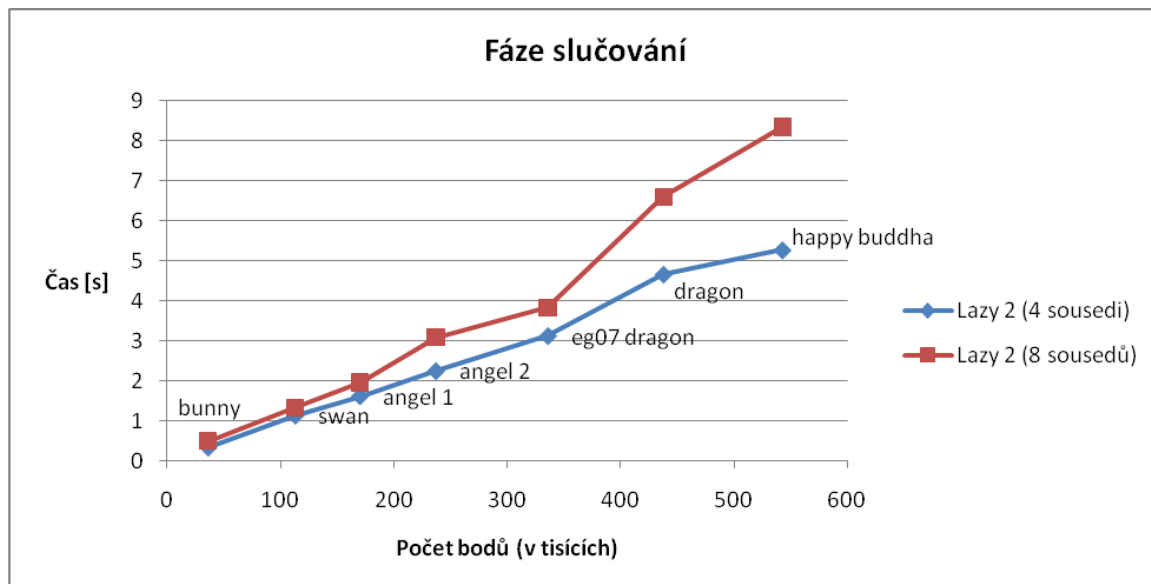


Graf 5.3 Srovnání doby trvání hledání nejblížešších sousedů pro 4 a 8 sousedů.



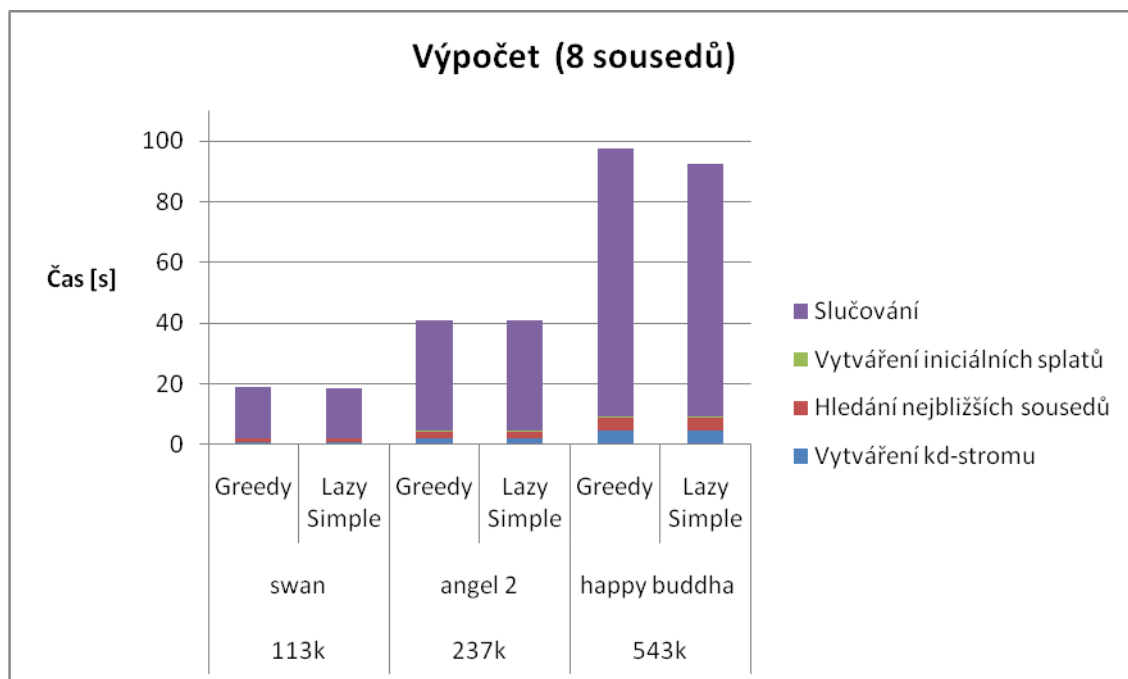
Graf 5.4 Srovnání doby trvání vytváření iniciálních splatů pro 4 a 8 sousedů.

V grafu 5.5 je srovnání doby trvání slučovací fáze pro stejný algoritmus a různý počet sousedů. I když se počet sousedů zdvojnásobil, u menších modelů je časová náročnost výpočtu jen o málo vyšší – při sloučení splatek dochází k degeneraci více hran, které se odstraní a dále nezpracovávají.



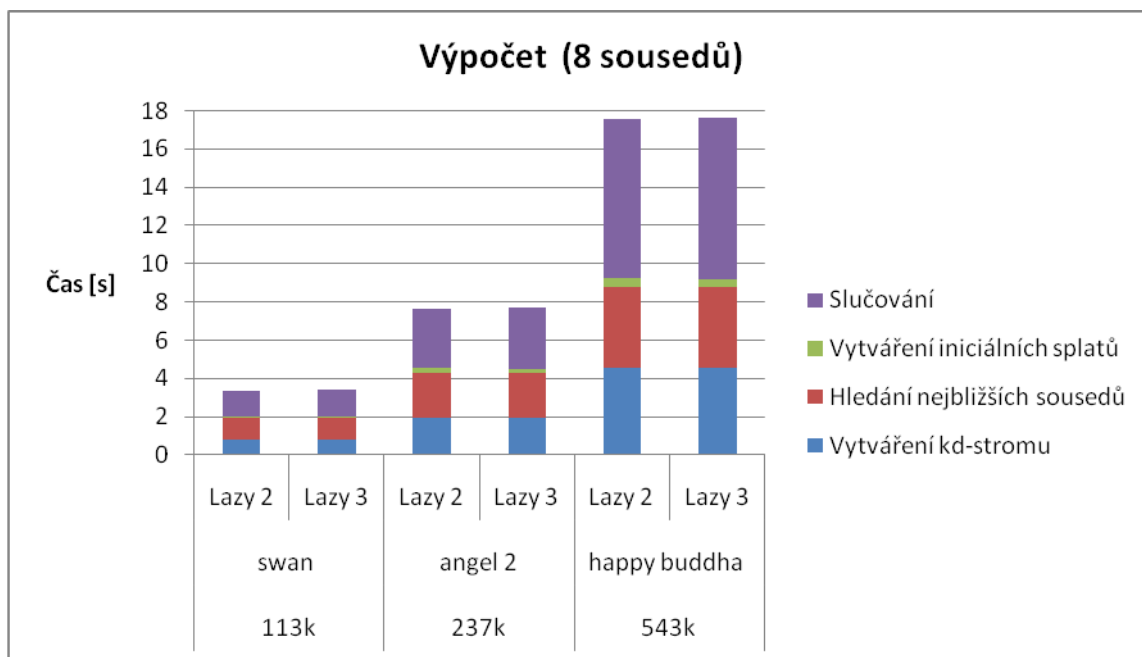
Graf 5.5 Srovnání doby trvání fáze slučování splatek pro algoritmus Lazy 2 pro 4 a 8 sousedů.

V posledních dvou grafech 5.6 a 5.7 je zobrazen poměr doby trvání jednotlivých částí výpočtu pro tři vybrané modely *swan*, *angel 2* a *happy buddha*. Pro hladový a jednoduchý líný algoritmus zabírá slučování největší část celého výpočtu, což je způsobeno neefektivním využitím GPU. Pro zbylé dvě varianty líného algoritmu je to přibližně pouze polovina času výpočtu pro model *happy buddha*, u druhých dvou menších modelů jen asi třetina. V tabulce u každého grafu je uveden čas trvání jednotlivých fází zobrazených v grafu, pouze celkový čas v grafu zobrazen není.



| Výpočet (8 sousedů) [s] | swan (113k) | | angel 2 (237k) | | happy buddha (543k) | |
|----------------------------|-------------|-------------|----------------|-------------|---------------------|-------------|
| | Greedy | Lazy Simple | Greedy | Lazy Simple | Greedy | Lazy Simple |
| Vytváření kd-stromu | 0,792 | 0,795 | 1,971 | 1,959 | 4,524 | 4,537 |
| Hledání sousedů | 1,191 | 1,162 | 2,304 | 2,315 | 4,218 | 4,314 |
| Vytváření splatů | 0,066 | 0,066 | 0,217 | 0,221 | 0,449 | 0,451 |
| Slučování | 17,087 | 16,505 | 36,468 | 36,465 | 88,381 | 83,125 |
| Celkový čas | 19,216 | 18,610 | 41,161 | 41,167 | 97,984 | 92,837 |

Graf 5.6 Srovnání trvání jednotlivých fází výpočtu pro tři vybrané modely a tabulka s hodnotami. Celkový čas v tabulce je změřený čas trvání výpočtu, nikoli součet uvedených hodnot.



| Výpočet (8 sousedů) [s] | swan (113k) | | angel 2 (237k) | | happy buddha (543k) | |
|----------------------------|-------------|--------|----------------|--------|---------------------|--------|
| | Lazy 2 | Lazy 3 | Lazy 2 | Lazy 3 | Lazy 2 | Lazy 3 |
| Vytváření kd-stromu | 0,790 | 0,791 | 1,955 | 1,954 | 4,534 | 4,533 |
| Hledání sousedů | 1,160 | 1,172 | 2,351 | 2,311 | 4,238 | 4,236 |
| Vytváření splatů | 0,066 | 0,066 | 0,221 | 0,222 | 0,446 | 0,447 |
| Slučování | 1,322 | 1,410 | 3,082 | 3,215 | 8,333 | 8,437 |
| Celkový čas | 3,418 | 3,523 | 7,811 | 7,901 | 17,962 | 18,063 |

Graf 5.7 Srovnání trvání jednotlivých fází výpočtu pro tři vybrané modely a tabulka s hodnotami. Celkový čas v tabulce je změřený čas trvání výpočtu, nikoli součet uvedených hodnot.

5.2.7 Možná vylepšení

Vytváření kd-stromu se zatím jeví jako část předzpracování, která zabírá poměrně dost času a je možné ji urychlit. Po prvním kroku by se další vytváření stromu dalo paralelizovat, větve jsou na sobě nezávislé.

Při výpočtu normálových vektorů iniciálních splatů se zjišťuje pouze směrová přímka vektoru, orientace samotného vektoru není určena. Při výpočtu chyby se pak počítá s menším z úhlů mezi vektory, což není vždy správně. Pro lepší výsledky by bylo vhodné provést určení správného směru normálových vektorů.

Dále by bylo možné ještě lépe optimalizovat komunikaci mezi CPU a GPU – varianta č. 3 líného algoritmu obsahuje nastavení, které hrany se testují před posláním dat na GPU a které se přepočítávají po přečtení výsledků, což ovlivňuje četnost posílání dat.

Pro lepší využití GPU by mohlo být zajímavé zaznamenávat úspěšné a neúspěšné pokusy o čtení dat, vytvořit si tak odhad délky trvání výpočtů na GPU a případně i vytvořit vhodný „konfigurační soubor“ s informacemi pro CPU, kdy má přistupovat k datům z GPU. Při opakovaných výpočtech na téže stroji by se tak více zatížilo GPU.

Také by bylo vhodné pro srovnání implementovat výpočet kompletně na CPU. Tak by bylo možné porovnat, jak se výpočet zrychlil přesunutím vybraných částí na GPU.

Závěr

Bodovou reprezentaci objektu lze využít, stejně jako reprezentaci pomocí trojúhelníkové sítě, pro všechny úlohy související se zpracováním a modelováním objektů, jak bylo popsáno ve druhé kapitole. Body trojúhelníky nenahradí, ale velmi dobře je doplňují. Mají jiné možnosti, některé typy úloh se s nimi řeší lépe a jednodušeji.

V implementaci jsou předvedeny možné postupy komunikace mezi CPU a GPU pro DirectX 10 – hlavně s využitím výstupu přes stream output stage buffery. Tato část pipeline ulehčuje přenos spočtených dat zpět na CPU, není nutné renderovat výsledky do textury. Také bez geometry shaderu by se tolik částí algoritmu nemohlo provádět na GPU, vytváření nových splatů a ohodnocování hran grafu vyžaduje přístup k oběma splatům, které se slučují.

Díky této nové koncepci grafické pipeline bude možné přenést některé výpočetní části podobných algoritmů na GPU a tak je paralelizovat a urychlit výpočet LODu objektů.

Literatura

- [1] Belytschko T., Lo Y. a Gu L. (1994): Element-free galerkin methods. *Int. J. Numer. Meth. Eng.*, **37**, 229-256
- [2] Brodskij D. a Watson B. (2000): Model simplification through refinement. *Graphics Interface*, 221-228
- [3] Cohen-Steiner D., Alliez P. a Desbrun M. (2004): Variational shape approximation. *ACM Transactions on Graphics*. Special issue for SIGGRAPH conference, 905-914
- [4] *Direct3D 10 Graphics*, dostupné na [http://msdn.microsoft.com/en-us/library/bb205066\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb205066(VS.85).aspx)
- [5] El-Sana J., Sokolovsky N. a Silva C. (2001): Integrating Occlusion Culling with View-Dependent Rendering. *Proceedings IEEE Visualisation 2001*, 371-378
- [6] El-Sana J. a Hadar O. (2002): Motion-based View-Dependent Rendering. *Computer and Graphics*
- [7] Funkhouser T. A. a Séquin C. H. (1993): Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environment. *Proceedings of SIGGRAPH 93*, 247-254
- [8] Gobbetti E. a Bouvier E. (1999): Time-Critical Multiresolution Scene Rendering. *Proceedings of IEEE Visualisation 1999*, 123-130
- [9] Gross M. a Pfister H. (2007): Point-based Graphics. Morgan Kaufmann Publishers
- [10] Heikkilä J. a Silvén O. (1997): A four-step camera calibration procedure with implicit image correction. *Proceedings of the Conference on Computer Vision and Pattern Recognition*, 1106-1112
- [11] Holloway R. L. (1991): Viper: A Quasi-Real-Time Virtual-Worlds Application. *Technical Report No. TR-92-004*. Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC
- [12] Hopp H. (1996): Progressive meshes. *Computer Graphics, SIGGRAPH 1996 Proceedings*, ACM Press, New York, 99-108
- [13] Hopp H., DeRose T., Duchamp T., McDonald J. a Stuetzle W. (1992): Surface reconstruction from unorganized point cloud. Edwin E. Catmull, ed., *Computer Graphics, SIGGRAPH 1992 Proceedings*, **26**, 71-78
- [14] Jolliffe I. (1986): Principal Component Analysis. *Spring-Verlag*

- [15] Kosara R., Miksch S. a Hauser H. (2002): Focus+Context Taken Literally. *IEEE Computer Graphics and Applications*, **22** (1), 22-29
- [16] Liu G.-R..a Liu M. B. (2003): Smoothed Particle Hydrodynamics. *World Scientific*
- [17] Luebke D. a Erikson C. (1997): View-Dependent Simplification of Arbitrary Polygonal Enviroments. *Proceedings of SIGGRAPH 97*, 199-208
- [18] Luebke D., Reddy M., Cohen J. D., Varshney A., Watson B. a Huebner R. (2003): Level of Detail for 3D Graphics. Morgan Kaufmann Publishers
- [19] Maciel P. W. C. a Shirley P. (1995): Visual Navigation of Large Enviroments Using Textured Clusters. *Proceedings of Symposium on Interactive 3D Graphics*, 95-102
- [20] Mason A. a Blake E. H. (1997): Automatic Hierarchical Level of Detail Optimization in Computer Animation. *Computer Graphics Forum*, **16**(3): 191-199
- [21] Matusik W., Buehler C., Raskar R., Gortler S. J. a McMillan L. (2000): Image-based visual hulls. *Computer Graphics, SIGGRAPH 2000 Proceedings*, 396-374
- [22] Mitra N. J., Nguyen A. a Guibas L. (2004): Estimating surface normals in noisy point-cloud data. *International Journal of Computational Geometry and Applications*, **14** (4-5), 261-276
- [23] Monaghan J. J. (2005): Smoothed Particle Hydrodynamics. *Reports on Progress in Physics*, **68**, 1703-1759
- [24] Müller M., Keiser R., Nealen A., Pauly M., Gross M. a Alexa M. (2004): *Point-based animation of elastic, plastic and melting objects*. 141-151
- [25] O'Brien J. F., Bargteil A.W. a Hodgins J. K. (2002): Graphical modeling and animation of ductile fracture. *ACM Transactions on Graphics, SIGGRAPH 2002 Proceedings*, 291-294
- [26] Organ D., Fleming M., Terry T. a Belytschko T. (1996): Continuous meshless approximation for nonconvex bodies by diffraction and transparency. *Computational Mechanics*, **18**, 1-11
- [27] Pajarola R. (2003): Efficient level of details for point-based rendering. *Proceedings IASTED International Conference on Computer Graphics and Imaging (CGIM)*, 141-146
- [28] Pauly M., Gross M. a Kobbelt L. P. (2002): Efficient simplification of point-sampled surfaces. *Proceedings of the Conference on Visualization 2002*, 163-170

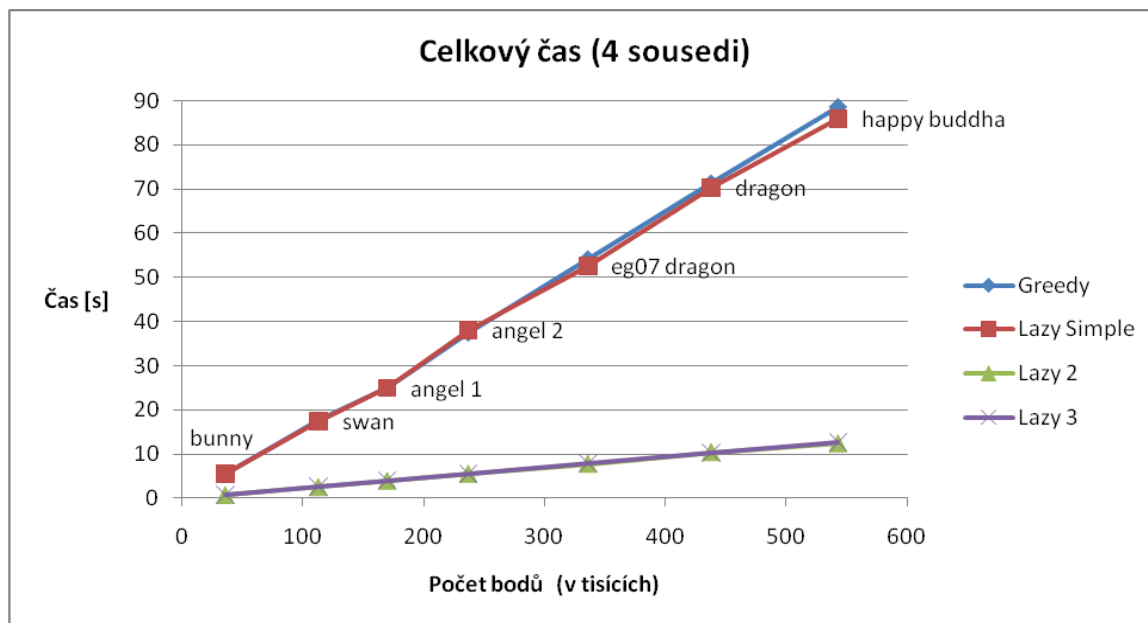
- [29] Pauly M., Keiser R., Kobbelt L. P. a Gross M. (2003): Shape modeling with point-sampled geometry. *ACM Transactions on Graphics*, SIGGRAPH 2003 Proceedings, **22**(3), 641-650
- [30] Shaffer E. a Garland M. (2001): Efficient adaptive simplification of massive meshes. *Proceedings of the Conference on Visualization 2001*, IEEE Computer Society 2001, 127-134
- [31] Turk G. (1991): Generating textures on arbitrary surfaces using reaction diffusion. *Computer Graphics*, SIGGRAPH 1991 Proceedings, 289-298
- [32] Walbourn C., Barrero D., Boyd D., Glassenberg S., Oneppo M., Porcino N., Service D., Wenzel C. (2007): *Introduction to Direc3D 10*, SIGGRAPH 2007, dostupné z [http://www.microsoft.com/downloads/ jako Intro_to_Direct3D10.zip](http://www.microsoft.com/downloads/jako_Intro_to_Direct3D10.zip)
- [33] Wu J. a Kobbelt L. P. (2004): Optimized subsampling of point sets for surface splatting. *Computer Graphics Forum*, EUROGRAPHICS 2004 Proceedings, **23**(3), 643-652
- [34] Wu J., Zang Z. a Kobbelt L. P. (2005): Progressive splatting. *Eurographics Symposium on Point-based Graphics*, 25-32
- [35] Xia J. C., El-Sana J. a Varshney A. (1997): Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics*, **3**(2), 171-183
- [36] EUROGRAPHICS 2007, <http://www.cgg.cvut.cz/eg07/>
- [37] Large Geometric Models Archive, http://www.cc.gatech.edu/projects/large_models/
- [38] The Stanford 3D Scanning Repository, <http://graphics.stanford.edu/data/3Dscanrep/>
- [39] University of Konstanz, <http://www.inf.uni-konstanz.de/cgip/projects/scanning/anteater/>

Přílohy

Grafy a tabulky

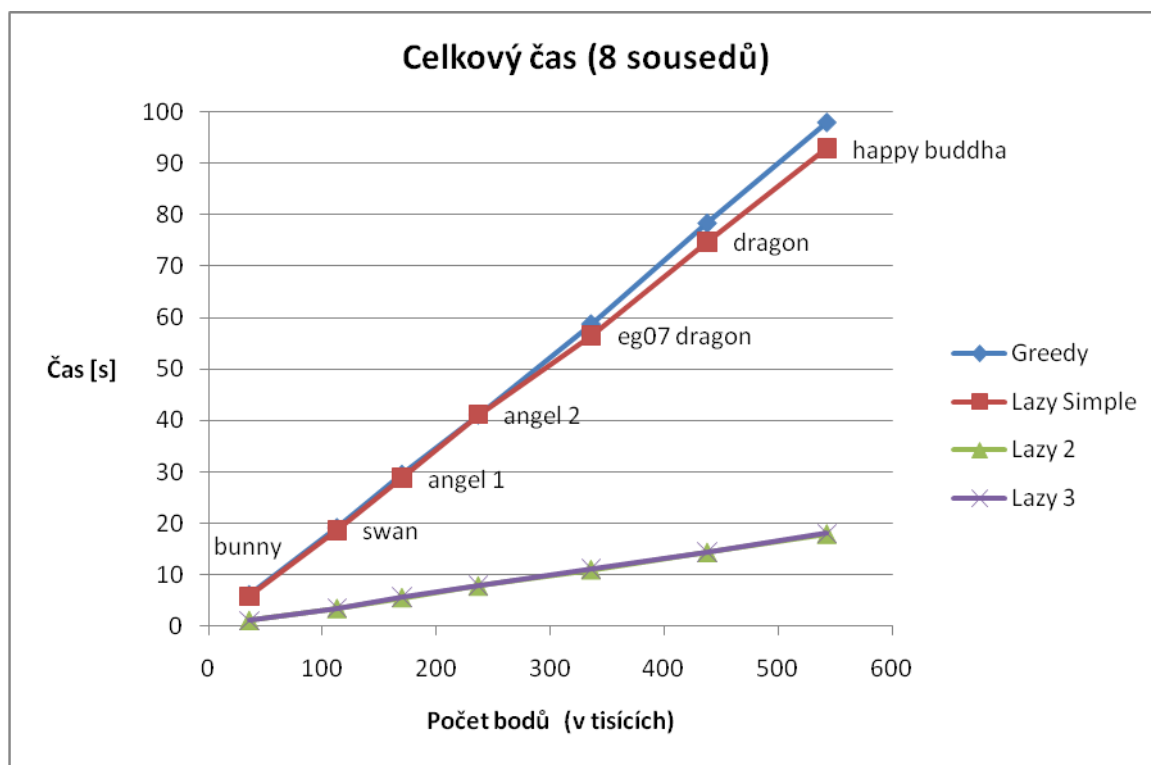
V této příloze jsou uvedeny tabulky a grafy ke kapitole 5.2, části Měření.

Graf 5.1 Graf a tabulka celkového trvání výpočtu pro jednotlivé algoritmy v závislosti na počtu bodů modelu. Výpočet probíhal pro 4 sousedy.



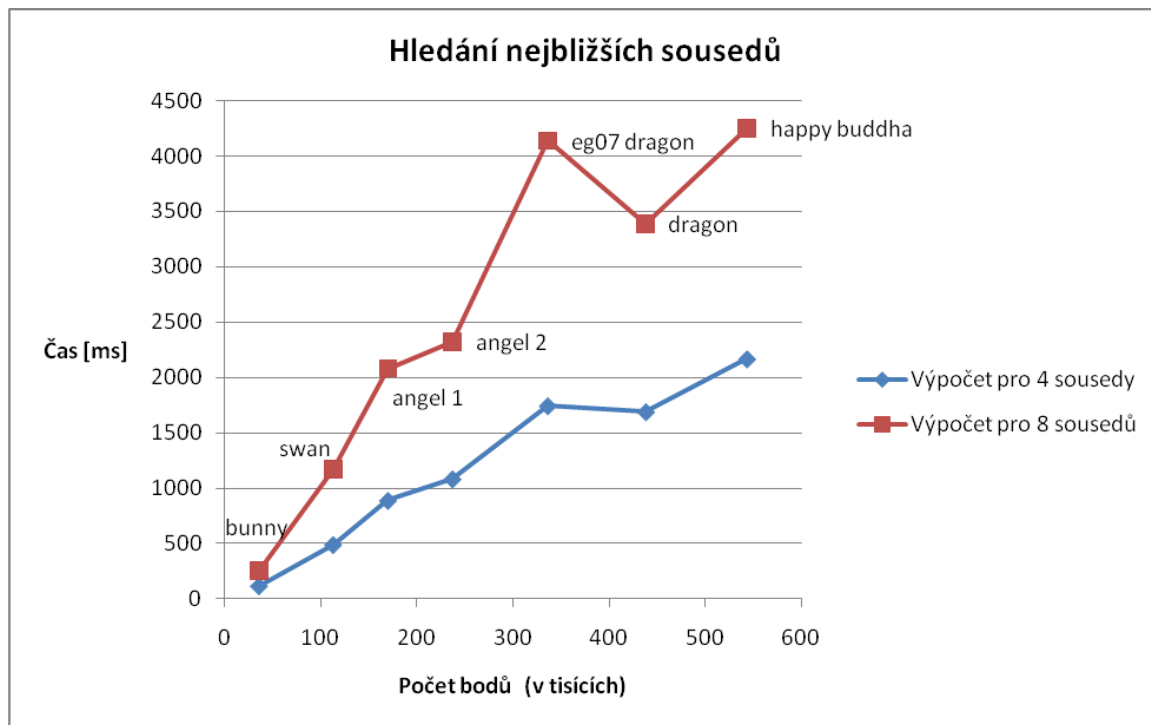
| Celkový čas (4 sousedi) [s] | drill shaft | bunny | swan | angel 1 | angel 2 | eg07 dragon | dragon | happy buddha |
|-----------------------------------|----------------|-------|--------|---------|---------|----------------|--------|-----------------|
| Počet bodů | 1961 | 35947 | 113238 | 170355 | 237018 | 336017 | 437645 | 543652 |
| Greedy | 0,322 | 5,564 | 17,688 | 25,057 | 37,560 | 54,212 | 71,374 | 88,661 |
| Lazy Simple | 0,313 | 5,559 | 17,397 | 24,980 | 38,071 | 52,540 | 70,254 | 85,956 |
| Lazy 2 | 0,084 | 0,701 | 2,511 | 3,928 | 5,529 | 7,754 | 10,386 | 12,362 |
| Lazy 3 | 0,090 | 0,732 | 2,646 | 4,047 | 5,585 | 7,903 | 10,306 | 12,710 |

Graf 5.2 Graf a tabulka celkového trvání výpočtu pro jednotlivé algoritmy v závislosti na počtu bodů modelu. Výpočet probíhal pro 8 sousedů.



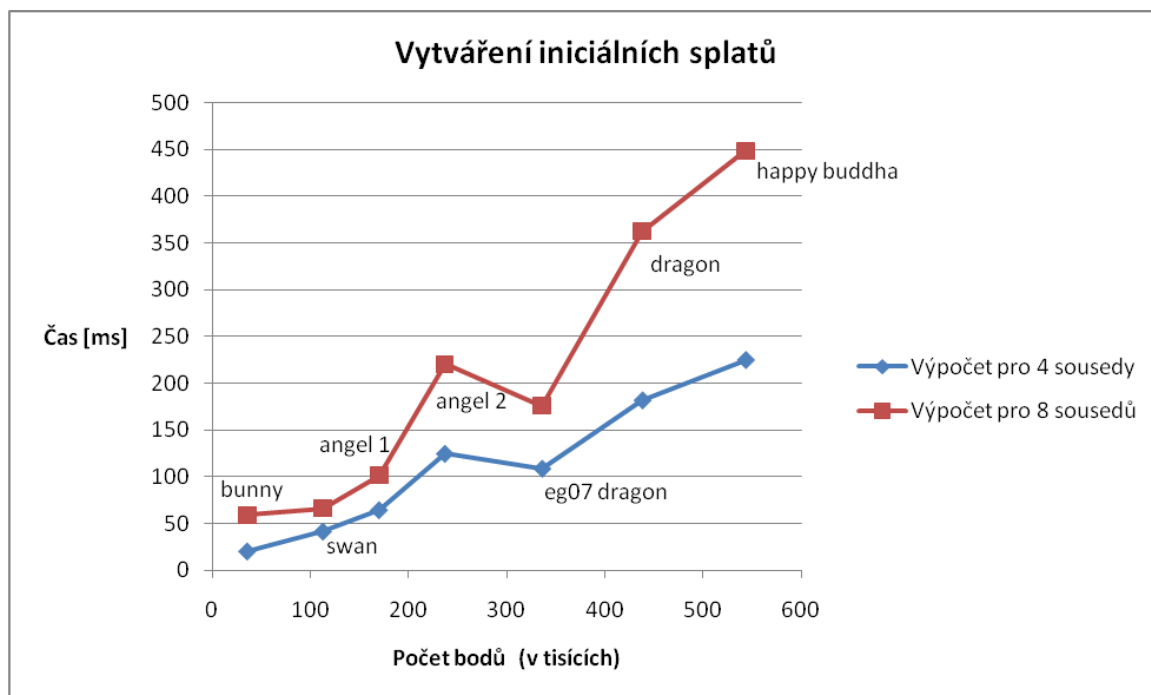
| Celkový čas (8 sousedů) [s] | drill shaft | bunny | swan | angel 1 | angel 2 | eg07 dragon | dragon | happy buddha |
|-----------------------------------|----------------|-------|--------|---------|---------|----------------|--------|-----------------|
| Počet bodů | 1961 | 35947 | 113238 | 170355 | 237018 | 336017 | 437645 | 543652 |
| Greedy | 0,343 | 6,178 | 19,216 | 29,566 | 41,161 | 58,797 | 78,391 | 97,984 |
| Lazy Simple | 0,259 | 5,832 | 18,610 | 28,784 | 41,167 | 56,560 | 74,761 | 92,837 |
| Lazy 2 | 0,089 | 1,080 | 3,418 | 5,521 | 7,811 | 10,995 | 14,358 | 17,962 |
| Lazy 3 | 0,098 | 1,070 | 3,523 | 5,689 | 7,901 | 11,113 | 14,342 | 18,063 |

Graf 5.3 Srovnání doby trvání hledání nejbližších sousedů pro 4 a 8 sousedů.



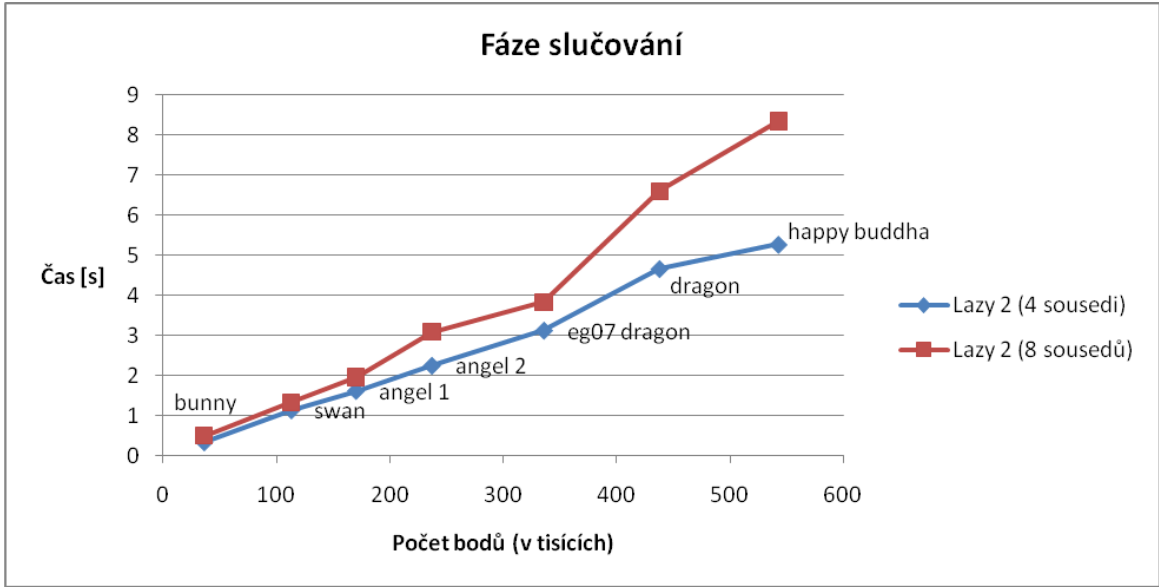
| Hledání nejbližších sousedů [ms] | drill shaft | bunny | swan | angel 1 | angel 2 | eg07 dragon | dragon | happy buddha |
|----------------------------------|-------------|---------|----------|----------|----------|-------------|----------|--------------|
| Počet bodů | 1961 | 35947 | 113238 | 170355 | 237018 | 336017 | 437645 | 543652 |
| Výpočet pro 4 sousedy | 7,545 | 117,583 | 490,337 | 889,295 | 1083,292 | 1744,376 | 1690,929 | 2168,183 |
| Výpočet pro 8 sousedů | 14,512 | 256,053 | 1171,556 | 2080,174 | 2321,405 | 4139,349 | 3388,022 | 4248,902 |

Graf 5.4 Srovnání doby trvání vytváření iniciálních splotů pro 4 a 8 sousedů.



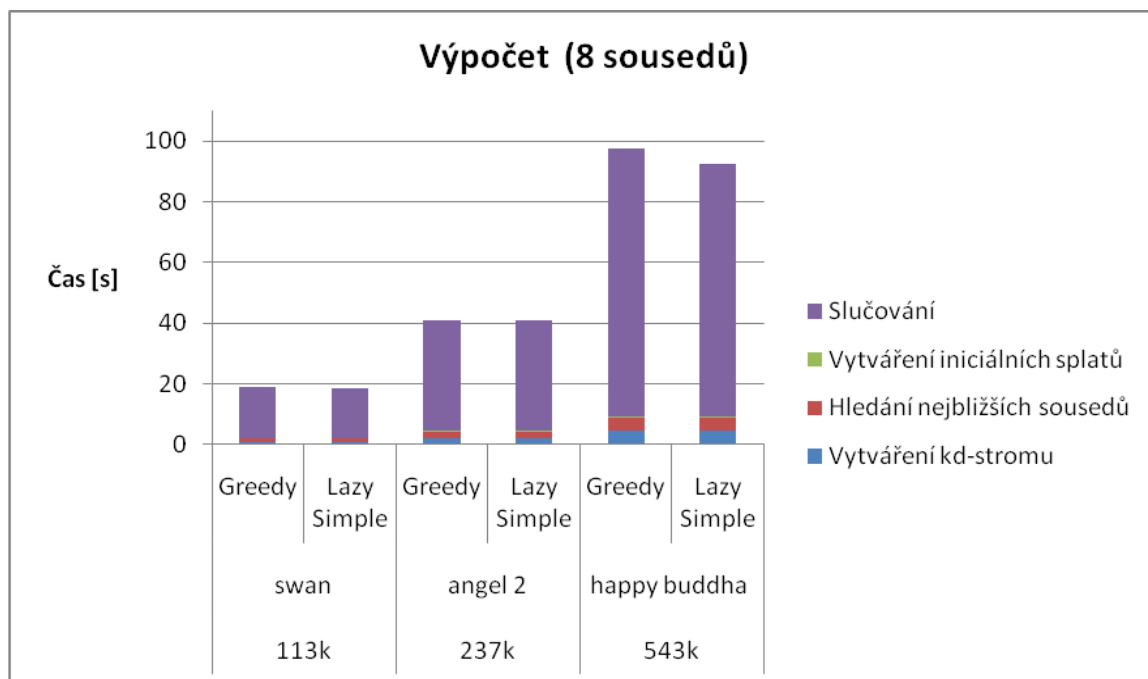
| Vytváření iniciálních splotů [ms] | drill shaft | bunny | swan | angel 1 | angel 2 | eg07 dragon | dragon | happy buddha |
|-----------------------------------|-------------|--------|--------|---------|---------|-------------|---------|--------------|
| Počet bodů | 1961 | 35947 | 113238 | 170355 | 237018 | 336017 | 437645 | 543652 |
| Výpočet pro 4 sousedy | 6,247 | 20,275 | 41,480 | 64,437 | 124,718 | 108,781 | 181,918 | 225,042 |
| Výpočet pro 8 sousedů | 6,512 | 59,016 | 66,278 | 101,743 | 220,507 | 176,053 | 362,465 | 448,402 |

Graf 5.5 Srovnání doby trvání fáze slučování splatek pro algoritmus Lazy 2 pro 4 a 8 sousedů.



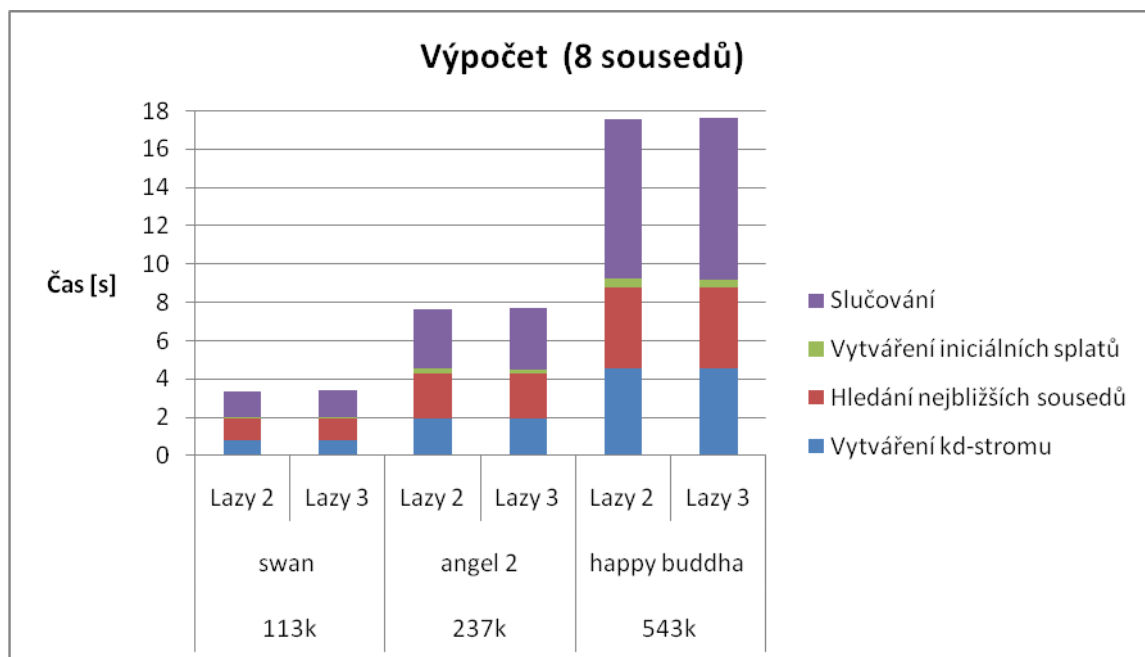
| Fáze slučování [s] | drill shaft | bunny | swan | angel 1 | angel 2 | eg07 dragon | dragon | happy buddha |
|--------------------|-------------|-------|--------|---------|---------|-------------|--------|--------------|
| Počet bodů | 1961 | 35947 | 113238 | 170355 | 237018 | 336017 | 437645 | 543652 |
| Lazy 2 (4 sousedi) | 0,040 | 0,329 | 1,131 | 1,614 | 2,242 | 3,120 | 4,659 | 5,266 |
| Lazy 2 (8 sousedů) | 0,040 | 0,495 | 1,322 | 1,961 | 3,082 | 3,827 | 6,587 | 8,333 |

Graf 5.6 Srovnání trvání jednotlivých fází výpočtu pro tři vybrané modely a tabulka s hodnotami. Celkový čas v tabulce je změřený čas trvání výpočtu, nikoli součet uvedených hodnot.



| Výpočet (8 sousedů) [s] | swan (113k) | | angel 2 (237k) | | happy buddha (543k) | |
|----------------------------|-------------|-------------|----------------|-------------|---------------------|-------------|
| | Greedy | Lazy Simple | Greedy | Lazy Simple | Greedy | Lazy Simple |
| Vytváření kd-stromu | 0,792 | 0,795 | 1,971 | 1,959 | 4,524 | 4,537 |
| Hledání sousedů | 1,191 | 1,162 | 2,304 | 2,315 | 4,218 | 4,314 |
| Vytváření splatů | 0,066 | 0,066 | 0,217 | 0,221 | 0,449 | 0,451 |
| Slučování | 17,087 | 16,505 | 36,468 | 36,465 | 88,381 | 83,125 |
| Celkový čas | 19,216 | 18,610 | 41,161 | 41,167 | 97,984 | 92,837 |

Graf 5.7 Srovnání trvání jednotlivých fází výpočtu pro tři vybrané modely a tabulka s hodnotami. Celkový čas v tabulce je změřený čas trvání výpočtu, nikoli součet uvedených hodnot.



| Výpočet (8 sousedů) [s] | swan (113k) | | angel 2 (237k) | | happy buddha (543k) | |
|----------------------------|-------------|--------|----------------|--------|---------------------|--------|
| | Lazy 2 | Lazy 3 | Lazy 2 | Lazy 3 | Lazy 2 | Lazy 3 |
| Vytváření kd-stromu | 0,790 | 0,791 | 1,955 | 1,954 | 4,534 | 4,533 |
| Hledání sousedů | 1,160 | 1,172 | 2,351 | 2,311 | 4,238 | 4,236 |
| Vytváření splatů | 0,066 | 0,066 | 0,221 | 0,222 | 0,446 | 0,447 |
| Slučování | 1,322 | 1,410 | 3,082 | 3,215 | 8,333 | 8,437 |
| Celkový čas | 3,418 | 3,523 | 7,811 | 7,901 | 17,962 | 18,063 |

Uživatelský manuál – program Generate Splats

Program **Generate Splats** slouží k vytváření stromu splatů ze vstupních bodů zadaného modelu. Vybraná patra stromu je možné zobrazit pomocí programu *Render Splats*. Program se neinstaluje, spouští se souborem „GenerateSplats.exe“ v adresáři „Bin“.

Ovládání programu

Po spuštění programu se zobrazí okno a vypíše se aktuální nastavení parametrů pro výpočet. Nastavení lze změnit v menu File → Change settings, zobrazení aktuálních parametrů je možné pomocí menu File → Actual settings.

Výpočet s aktuálním nastavením se spustí přes menu Run → Start computing. Po dokončení výpočtu je možné výsledky uložit do souboru.

Změna nastavení

Pomocí menu File → Change settings se zobrazí dialog, který umožňuje změnit parametry výpočtu. V dialogu je možné zadat vstupní a výstupní soubor a která patra stromu se mají uložit. Také je zde možné zvolit algoritmus, který se použije při výpočtu.

Vstupní soubor

Program načítá vstupní soubory v textovém formátu *ply* (ply format ascii) a ve formátu *obj*. Pokud soubor není v očekávaném formátu nebo obsahuje chybu, nebude načten.

Výstupní soubor

Výstupní soubor je uložen v textovém formátu. Do souboru je možné uložit jedno nebo více pater stromu (File → Save levels) nebo původní body modelu spolu s jejich spočtenými splaty (File → Save original). Také je možné do souboru uložit celý strom splatů (File → Save tree).

Výstupní soubory s několika patry stromu je možné zobrazit pomocí programu *Render Splats* (soubory s celým stromem tento program zobrazovat neumí).

Počet splatů

V dialogu se zadává požadovaný počet splatů v patře stromu. Program vybere takové patro stromu, jehož počet splatů je nejbližší požadovanému počtu. Toto patro bude uloženo do souboru. Pro uložení více pater se zadají požadované počty splatů oddělené novým řádkem nebo mezerou. Pro uložení splatů z původních bodů se zadává číslo 0.

Algoritmus

Greedy (hladový) algoritmus přepočítává všechny změny ihned, *Lazy Simple* (líný) algoritmus odkládá přepočítávání hran. Oba tyto algoritmy počítají na GPU jen jeden nový splat. *Lazy 2* algoritmus odkládá přepočítávání hran i splatů, během výpočtu nových

bodů na GPU zpracuje jen jednu další hranu. *Lazy 3* také všechny přepočty odkládá, a pokud je to možné, zpracuje více hran.

Poznámka: Výpočet pro větší modely a pro algoritmy *Greedy* a *Lazy Simple* trvá delší dobu. Může se stát, že operační systém hlásí, že program neodpovídá, výpočet ale nadále probíhá.

Parametry příkazové řádky

V příkazové řádce je možné použít tyto parametry:

- i [jméno souboru] Jméno souboru, ve kterém jsou uloženy souřadnice bodů modelu.
- f [jméno souboru] Jméno souboru, do kterého se uloží vygenerované splaty.
- a [číslo] Algoritmus pro zpracování hran:
 - 0 – Greedy
 - 1 – Lazy Simple
 - 2 – Lazy 2
 - 3 – Lazy 3
- s [číslo] Požadovaný počet splatů, podle tohoto počtu program vybere patro stromu, které bude uloženo do souboru. Pro uložení splatů z původních bodů se zadává číslo 0. Pro uložení více pater do jednoho souboru lze zadat několik čísel oddělených čárkou včetně hranatých závorek, nesmí být mezi nimi žádné mezery.
Příklad: -s [100,200]

Požadavky

Program pro svou práci vyžaduje Microsoft Windows Vista a vyšší a DirectX 10. Dále je potřeba grafická karta kompatibilní s DirectX 10.

Program dále používá soubory fx v adresáři *Shaders*, pokud některý z požadovaných souborů chybí nebo obsahuje chyby, není možné program spustit.

Uživatelský manuál – program Render Splats

Program Render Splats zobrazuje splaty uložené v souborech vygenerované programem *Generate Splats*. Program se neinstaluje, spustí se souborem „RenderSplats.exe“ umístěným v adresáři „Bin“.

Ovládání programu

K ovládání programu se používá myš: při stisknutí levého tlačítka se otáčí kamera, kolečkem se provádí zoom. V levém horním rohu se zobrazují parametry grafického zařízení, snímková frekvence (fps) a aktuální pozice kamery, vpravo jsou tlačítka pro další nastavení.

Po spuštění programu se načte soubor zadaný na příkazové řádce (pomocí parametru -f) a zobrazí se první z pater stromu, které jsou v něm uloženy.

Aktuálně zobrazované patro stromu je možné změnit výběrem z rozbalovacího seznamu pod názvem *Current level* – v seznamu se zobrazuje počet splatů v daném patře.

Pod názvem *Splat size* je možné změnit velikost splatů – na výběr je původní velikost splatů („Full“), kdy se splaty zobrazují tak, jak byly spočteny, a poloviční a třetinová velikost („Half“, „Third“).

Funkce jednotlivých tlačítek

| | |
|------------------------------|--|
| <i>Toggle full screen</i> | Přepíná mezi full screen a windowed módem. |
| <i>Change device (F2)</i> | Umožňuje změnit nastavení grafického zařízení. |
| <i>Back to position (F4)</i> | Vrací kameru do původní pozice. |
| <i>Rotation</i> | Zapíná a vypíná otáčení objektu. |

Klávesa *ESC* zavírá program.

V příkazové řádce je možné použít tyto parametry:

| | |
|--------------------|--|
| -f [jméno souboru] | Jméno souboru, ve které jsou uloženy splaty. |
|--------------------|--|

Vstupní soubor

Program načítá soubory, které obsahují jednotlivá patra stromu splatů – byly uloženy programem *Generate Splats* jako patra („levels“ nebo „original“). Soubory, které obsahují celý strom, nenačítá.

Dále je možné do souboru přidat počáteční pozici kamery pro tento model – na začátek souboru, před první LEVEL, se přidá řádek ve tvaru:

EYE x y z

kde x, y, z jsou souřadnice kamery (reálná čísla).

Poznámka: Tento řádek je nutné přidat ručně, program *Generate Splats* ho nezapíše.

Pokud není nalezen zadaný soubor nebo dojde k chybě při jeho načítání nebo není zadán žádný soubor, je o tom uživatel informován a program je ukončen.

Požadavky

Program pro svou práci vyžaduje Microsoft Windows Vista a vyšší a DirectX 10. Dále je potřeba grafická karta kompatibilní s DirectX 10.

Program dále používá soubory fx v adresáři *Shaders*, pokud některý z požadovaných souborů chybí nebo obsahuje chyby, není možné program spustit.

Obsah CD

| | |
|-----------------|--|
| Bin\ | Přeložené projekty Generate Splats a Render Splats |
| Shaders\ | HLSL shadery používané aplikacemi |
| Data\ | |
| Levels\ | Soubory modelů pro zobrazení v programu Render Splats |
| Models\ | Data modelů pro generování stromu splatů |
| DirectX\ | Instalační soubory DirectX 10 Redistributable |
| Documentation\ | Uživatelská a programátorská dokumentace |
| Sources\ | Složka se zdrojovými soubory. Projektové soubory MS Visual Studio 2008 |
| GenerateSplats\ | Program na generování stromu splatů |
| Ogl\ | Funkce použité pro měření času (knihovna třetí strany) |
| RenderSplats\ | Program pro zobrazování splatů v souborech vygenerovaných pomocí předchozího programu. |
| DXUT\ | Rozšíření DirectX použité pro GUI (knihovna třetí strany) |
| Shaders\ | HLSL shadery |
| Thesis\ | Text diplomové práce ve formátu pdf |