VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VOXELIZACE 3D MODELŮ A JEJICH ZPRACOVÁNÍ S VYUŽITÍM GPU

3D MODEL VOXELIZATION USING GPU FOR FURTHER PROCESSING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JÁN BRÍDA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MICHAL ŠPANĚL, Ph.D.**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Brída Ján, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Voxelizace 3D modelů a jejich zpracování s využitím GPU**
3D Model Voxelization Using GPU for Further Processing
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte dostupné materiály na téma voxelizace 3D modelů a možnosti využití voxelizace pro zobrazování a zpracování 3D modelů.
2. Zorientujte se v současných metodách voxelizace modelů s využitím GPU.
3. Vyberte vhodnou metodu a navrhněte způsob její implementace s využitím GPU.
4. Experimentujte s vaší implementací a případně navrhněte vlastní modifikace metod.
5. Porovnejte dosažené výsledky a diskutujte možnosti budoucího vývoje.
6. Vytvořte stručný plakát nebo video prezentující vaši diplomovou práci, její cíle a výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění prvních tří bodů zadaní.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

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

Vedoucí: **Španěl Michal, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Táto práca sa zameriava na analýzu súčasných techník pre povrchovú a úplnú binárnú voxelizáciu 3D modelov. Stručne popisuje aktuálné trendy v tejto problematike a identifikuje vhodnú metódu s cieľom paralelizácie daného riešenia na grafických kartách. Konkrétne vysvetľuje implementačný proces zvoleného algoritmu, ktorý bol popísaný v práci *Fast Parallel Surface and Solid Voxelization on GPUs*, produkujúci riedky voxelový oktálový strom. Výsledky projektu se blížia meraním pôvodných autorov. Taktiež je tu prezentované nové riešenie paralelnej extrakcie hladkej izoplochy z tejto štruktúry založené na *Marching Cubes*, ktoré redukuje počet prechádzaných kociek až o 98 % vo vysokých rozlíšeniach. Výsledkom implementácie je framework použiteľný pri ďalšom spracování voxelových scén.

Abstract

This thesis focuses on the analysis of the latest techniques for surface and solid binary voxelization of 3D models. It briefly describes current trends in this problematics and identifies a suitable method with an aim to parallelize the given solution on GPUs. It concretely explains the implementation process of the selected algorithm described in the paper *Fast Parallel Surface and Solid Voxelization on GPUs*, which produces a sparse voxel octree. The results are very close to those of the original authors. A new solution for extracting a smooth isosurface from this structure based on *Marching Cubes* is presented as well, providing up to 98 % reduction of the traversed cubes in higher resolutions. The resulting implementation is a framework usable for further voxel scene processing.

Kľúčové slová

voxelizácia, paralelné spracovanie, výpočtová geometria, riedky voxelový oktálový strom, extrakcia izoplochy

Keywords

voxelization, parallel processing, computational geometry, sparse voxel octree, isosurface extraction

Citácia

BRÍDA, Ján. *Voxelizace 3D modelů a jejich zpracování s využitím GPU*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Španěl, Ph.D.

Voxelizace 3D modelů a jejich zpracování s využitím GPU

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Michala Španěla, Ph.D.. Uviedol som všetky literárne pramene a publikácie, zo ktorých som čerpal.

.....
Ján Brída
24. mája 2017

Poďakovanie

Rád by som poďakoval svôjmu vedúcemu Ing. Michalovi Španělovi, Ph.D., za cenné rady, ktoré mi boli poskytnuté v rámci konzultácií.

Obsah

Kapitola 1

Úvod

Posledným časom sa v oblasti počítačovej grafiky vykresľovanej za behu začínajú čoraz viac objavovať voxelové reprezentácie 3D scén. Na vytváranie voxelových scén slúži nástroj zvaný 3D skenovacia konverzia, bežnejšie označovaná ako voxelizácia. Jedná sa o metódu prevodu (typicky) polygonovej siete do homogénnej diskrétnej mriežky buniek, čomu sa často hovorí *objem*. Dôležitými aspektmi voxelizácie sú presnosť, výkonnosť a hlavne pamäťová náročnosť. Táto konverzia je prirodzene paralelného charakteru, kedy testovanie náležitosti 3D objektu do objemu prebieha zvlášť pre všetky primitíva. S rozmachom moderných grafických čipov a odpovedajúcich programovacích rozhraní prišlo na aplikovanie týchto technologických vymožeností vo voxelizácii. Medzitým sa objavili nové rýchle postupy založené na explicítnych geometrických testoch a operáciach s využitím GPGPU. Postupy, ktoré nezakladajú voxelizáciu na vykresľovacom reťazci grafických kariet bežne nesprevádzajú problémy s ním spojené, aj poskytujú väčšiu flexibilitu pri návrhu a implementácii. Vlastnosťou efektívnej voxelizácie je nastavenie minimálneho počtu buniek objemu podľa spracovávaného vstupu, aby boli v diskrétnom priestore správne definované podmnožiny vyplnených a prázdnych oblastí. Dodatočne môžu byť odkazy na prázdny priestor vypustené z výslednej štruktúry (riedky voxelový oktálový strom). Splnenie podmienok minimality a riedkosti garantuje väčšiu efektívitu pri ďalšom spracovaní. Tým môže byť mimo iné modelovanie rozmanitejších objemov z viacerých vstupných útvarov. Často potom výstupom po takýchto operáciách je aproximovaná izoplocha, preto zostáva otázkou šetrne vygenerovať spojitý povrch z komplexnej štruktúry a vyhnúť sa tvorbe ostrých hrán z binárnej voxelizácie.

Obsahom tohto dokumentu je prehľad problematiky voxelizácie a väčšiny (prevažne) aktuálnych metód vykonávania tejto netriviálnej operácie. Začína sa jemným teoretickým úvodom v tejto oblasti kapitolou 2. Ďalšia kapitola 3 popisuje predovšetkým moderné techniky voxelizácie. Porovnáva jednotlivé riešenia a diskutuje dôvod vybrania sa cestou zvoleného implementačného návrhu [46], ktorý nakoniec preberie do hĺbky. Pretože po spracovaní voxelových dát môže byť vyžadovaný spätný prevod na trojuholníkovú sieť, bolo súčasťou projektu taktiež poskytnutie rýchlej extrakcie hladkej izoplochy z výstupu voxelizačného algoritmu. Teória za touto technikou je prebraná v kapitole 4. Kapitola 5 popisuje návrh postaveného frameworku pre voxelizáciu a generovanie izoplochy, určuje taktiež aplikačné rozhranie pre programátorov. Implementačná časť (kapitola 6) konkrétne definuje proces tvorby frameworku. Výsledky merania a testovania poskytnutého riešenia sú uvedené v predposlednej kapitole 7. Nakoniec záverečná kapitola 8 zhodnocuje prínosy tejto práce a naznačuje ďalšie smerovanie vývoja frameworku. Táto práca naväzuje na predchádzajúci semestrálny projekt autora a do určitej miery preberá text v kapitolách 2, 3.

Kapitola 2

Voxelizácia 3D modelov

Alternatívou tradičnej polygonovej reprezentácie 3D scén je voxelová reprezentácia. Tá poskytuje robustný a jednotný popis objektov ako základ objemovej grafiky. Objemovým elementom sa bežne hovorí *voxely*. Tie sú definované iba svojím stavom a informácia o ich pozícii je odvodená podľa ich adresy v diskrétnej mriežke objemu. Následkom toho je voxelový priestor schopný popísať priestorové závislosti, čo dobre nedokáže bežná polygonová sieť. Proces generovania voxelovej mriežky je diskretizácia spojitého priestoru definovaného hranicami vstupného modelu. Táto problematika má viacero riešení, logicky však vychádza z istých teoretických základov, ktoré sú ďalej rozvedené.

2.1 Pojmy a definície

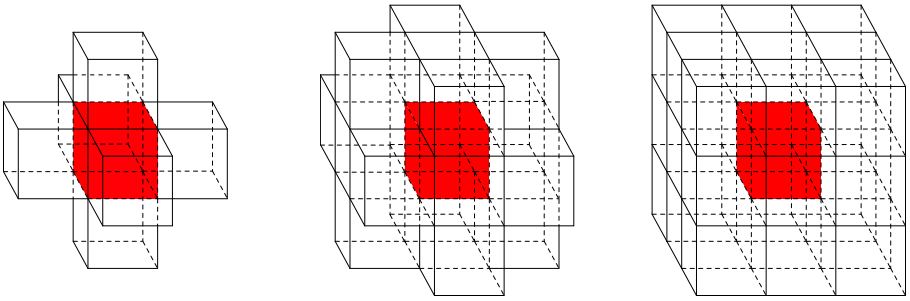
Voxelový priestor možno definovať množinou $\Sigma \subset \mathcal{R}^3_+$, v ktorej sa nachádza vstupný model Ω. Voxelová diskrétna mriežka Γ sa ďalej chápe ako podmnožina $\mathcal{N}^3_0$ o veľkosti limitovanej množinou Σ. Každý prvok mriežky ν je 0-dimenzionálny objekt a identifikuje ho stred voxela na pozícii $\vec{p} = (i, j, k)^\intercal \in \Gamma$. *Voronoi susedstvo* bodu $\vec{p}$ je zasa množina všetkých bodov Σ, ktoré sú k bodu $\vec{p}$ bližšie než ku ktorémukoľvek inému bodu v Γ a tvoria kocku identifikujúcu priestor ν [9]. Agregátom všetkých voxelov je teselácia 3D Euklidovského priestoru [29]. Vzorkovacia funkcia (2.1) v doméne Γ definuje binárny stav ν, kedy náležitosť determinuje prienik primitíva z Ω s kockou voxela ν.

$$f(\vec{p}) = \begin{cases} 1, & \text{ak voxel } \nu \text{ so stredom } \vec{p} \text{ náleží } \Omega \\ 0, & \text{inak} \end{cases} \tag{2.1}$$

Pre geometrické operácie môže byť voxel definovaný podľa pozície spodného rohu kocky voxela a $\delta \in \mathcal{N}$, kde δ uvádza rozmer kocky vo všetkých dimenziách. Užitočný pojem predstavuje ešte voxelový *stĺp*: to je množina voxelov, pre ktorú platí, že dve zo súradníc $(i, j, k)^\intercal \in \Gamma$ sú pre všetky voxely tejto množiny nemenné.

Susednosť voxelov sa definuje pomocou N-susednosti [9, 29], kde $N = \{6, 18, 26\}$. 26-susedstvo sa uvažuje, keď dva voxely zdieľajú vrchol, hranu alebo stenu. Pre 18-susednosť stačí, ak je to hrana alebo stena. Nakoniec pre 6-susednosť platí, že voxely susedia, ak zdieľajú stenu [9, 29, 46], ako je názorne ukázané na obrázku 2.1. N-cesta je sekvencia voxelov, kde pre každý pár po sebe idúcich voxelov platí kritérium N-susednosti. Pre lubovoľný pár voxelov platí, že sú N-spojené, ak existuje N-cesta, ktorá obsahuje daný pár [9].

Presnosť voxelizácie závisí od toho, ako sú v diskrétnom priestore definované spojitosti. Neúspech spojitosti pre korektnú charakteristiku povrchu je motiváciou vyjadrenia sepa-



Obr. 2.1: N-susedstvo voxelov, zľava: 6-susedstvo, 18-susedstvo, 26-susedstvo.

rability. Separabilita je topologická vlastnosť, udávajúca to, ako sú množiny voxelov Φ, Π oddelené v $\Xi = \Lambda \cup \Phi \cup \Pi$ podľa Λ. Viac formálne, Λ N-separuje Φ a Π, ak pre $\phi \in \Phi$ a $\pi \in \Pi$ platí, že sú N-spojené v Ξ a daná N-cesta obsahuje aspoň jeden voxel z Λ. Voxel ν patriaci N-separabilnemu povrchu Λ sa nazýva N-prostý, ak sa neporuší N-separabilita po $\Lambda - \{\nu\}$ [29]. N-separabilny povrch je N-minimálny, ak neobsahuje žiadne N-prosté voxely.

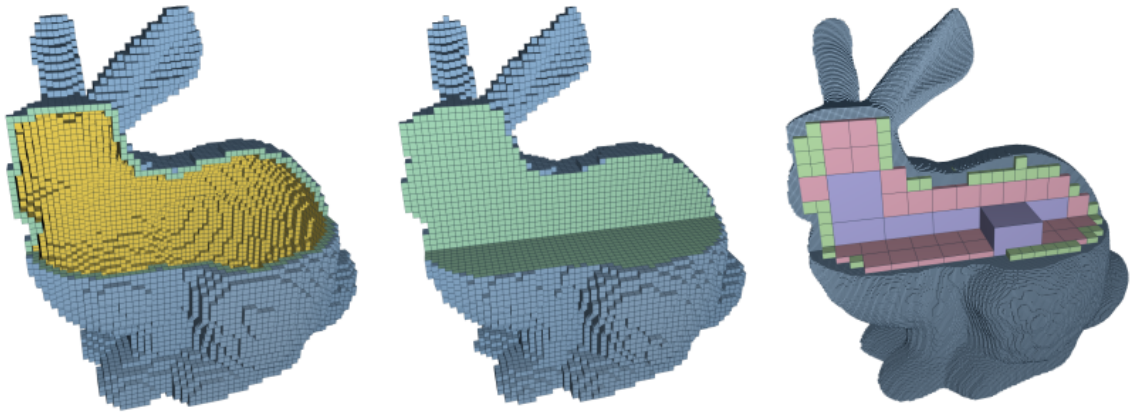
Ak je uvažovaný spojitý priestor Σ, na ktorom sú počas voxelizácie nastavené všetky voxely, ktoré sa i len dotýkajú hranice objektu v jedinom bode, potom je táto voxelizácia nazývaná *superkryt* Σ [29]. Takáto voxelizácia sa často označuje ako konzervatívna a býva 26-separabilna.

2.2 Typy voxelizácií

Podľa spôsobu vyplnenia priestoru sa hovorí o binárnej voxelizácii diskrétnej mriežky Γ buď povrchovej alebo úplnej. Povrchová voxelizácia spracováva vstupnú polygonovú sieť a zaznamenáva jej povrch nastavením voxelov, ktoré boli prekryté aspoň jedným primitívom [45]. Úplná voxelizácia vypĺňa priestor medzi náprotivými hranicami objektu prevrátením hodnoty všetkých voxelov v danom voxelovom stĺpci od miesta, kde došlo k nastaveniu voxela geometriou objektu [46, 45]. Podmienkou úplnej voxelizácie je vodotesná geometria [22, 46], inak dochádza k nesprávenmu vyplneniu Γ.

Separabilita a minimalita (viz podkapitola 2.1) voxelizácie vystihujú jemnosť, resp. presnosť generovania voxelov pri mapovaní na povrch 3D objektu. Voxelizácia by určite mala minimalizovať počet nastavených voxelov, a zároveň musí platiť, že nie je porušená podmienka zvolenej separability. Tento princíp umožňuje vynechať spracovanie voxelov v objemových algoritmoch a tým pádom zrýchliť dané riešenie. Príčinou porušenia týchto požiadavkov je typicky nedokonalá procedúra testovania náležitosti, preto je potrebné brať na tento fakt ohľad počas štúdie navrhovaných riešení.

Voxelové mriežky vo vysokom rozlíšení dokážu aj po kompresii zaberať veľkú časť pamäte. To je problém prevažne pri práci s objemom na grafických kartách, ktorých video pamäť býva na bežnom užívateľskom hardvéri obmedzená. Účinným typom potlačenia pamäťovej náročnosti je delenie priestoru na bloky skupín voxelov rovnakého stavu [46], typicky počas úplnej voxelizácie (obrázok 2.2). Naviac vyplýva, že záujem je často len o nastavené voxely a prázdny priestor nevyžaduje uchovanie informácie v nižších úrovniach delenia priestoru (pojem *riedkosti*). Problémom však býva skutočnosť zvýšenej komplexnosti voxelizácie do takejto štruktúry.

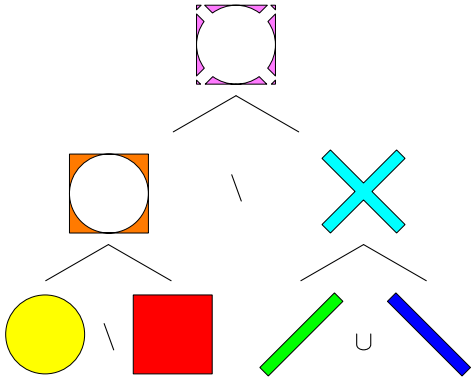


Obr. 2.2: Druhy voxelizácií [46].

2.3 Využitie voxelových reprezentácií 3D modelov

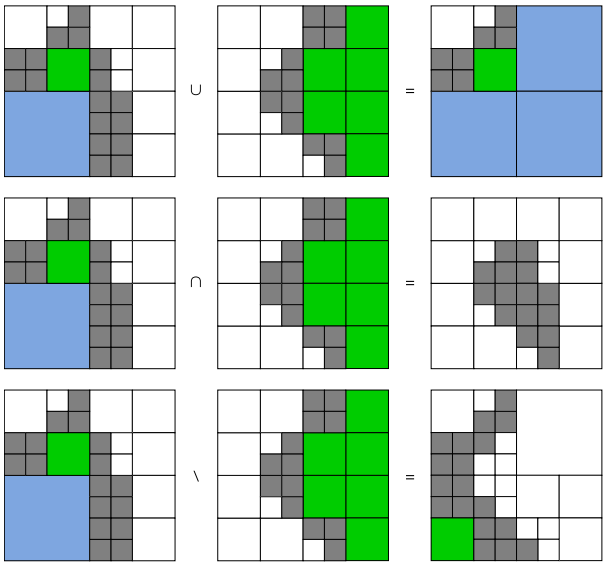
Objemové reprezentácie geometrických modelov sú užitočné v mnohých inžinierskych, vedeckých a medicínskych disciplínach. Veľkú výhodu oproti iným reprezentáciam poskytuje pravidelnosť členenia a usporiadania priestoru, čímž s voxelmi ľahko narábať. Dobre bojujú proti aliasingu adaptovaním rozlíšenia mriežky na rozlíšenie zobrazenia, to však v dobe HD monitorov nie je častým úkonom. Oveľa bežnejším postupom je pri delení priestoru prepínať úrovne detailu [14]. Vďaka popisnosti lokality dokážu byť využité v oblastiach zhodovania 3D objektov so vzorom [25] a určovaní viditeľnosti v takto definovanom prostredí. Interakcia voxelov medzi sebou sa hodí k simulácii tekutín [11] a prirodzene detekcie kolízií [2]. Dokážu dobre aproximovať vizuálne efekty, napr. dym, sneženie [38] alebo i oheň [49], lebo propagovanie takýchto javov nie je príliš zložité v diskrétnej mriežke. Voxely sa využívajú aj v aplikáciach výpočtov globálneho osvetlenia scény [15, 47, 30]. Medicínske dáta sa často vizualizujú zo sady voxelov na základe výstupu počítačovej tomografie či magnetickej rezonancie [34]. Objemovú geometriu možno ďalej násť pri škálovaní zobrazenia obrovských scén (kedy je v pamäti uložená len vyžadovaná časť voxelovej štruktúry), simulácii hĺbky ostrosti, vykreslenia komplikovanej transparentnosti povrchov, tvorbe interaktívneho prostredia a iné.

Booleovskými operáciami nad úplne voxelizovanými modelmi možno vykonávať procedúru konštruovania tuhej geometrie (CSG) a vytvárať tak zložitejšie modely [28]. CSG pracuje rozdelením tejto postupnosti aplikovaných operátorov do binárneho stromu, ktorého listy reprezentujú spracovávané modely a vnútorné uzly popisujú jednotlivé transformácie. Koreňom je výsledný CSG model (obrázkok 2.3). Bežne sa ako listy CSG stromu uvádzajú primitíva popísané analyticky, napr. sféry, válce, ale aj diskrétne definované mriežky binárnych hodnôt [28, 24]. Objemová reprezentácia objektov je teda potenciálne dobre prispôsobená pre použitie v CSG (VCSG), keďže operandmi sú voxelové množiny, ktoré obsahujú 0-rozmerné prvky. Taktiež obor hodnôt $f(\vec{p} \in \Gamma)$ pre ich vnútorný stav je prakticky totožný s pravdivostnou množinou Booleovej algebry. V porovnaní s klasickými trojuholníkovými sieťami sa voxely dajú považovať za robustnú optimalizáciu implementovania CSG procedúr. I keď bolo k téme (V)CSG napísaných pomerne dosť technických dokumentov (dokonca kompozícia počas voxelizácie [20, 3, 5, 24]), prieskum týchto techník ukazuje, že táto oblasť doposiaľ neposkytuje jednoznačné riešenia vytvárania VCSG do riedkeho priestoru.



Obr. 2.3: Popisná ukázka komponovania CSG v rámci stromovej štruktúry.

Motiváciou projektu je poskytnúť podklady pre schopnú konverziu 3D geometrie do takej štruktúry, ktorá umožní hierarchické aplikovanie CSG operácií medzi operandmi. Takáto konštrukcia bude pomerne komplikovaná vzhľadom na charakter určovania zmeny hierarchie výslednej kompozície, čo dobre popisuje obrázok 2.4, avšak táto problematika je momentálne mimo rozsahu tejto práce a môže byť rozobratá v jej budúcom vývoji.



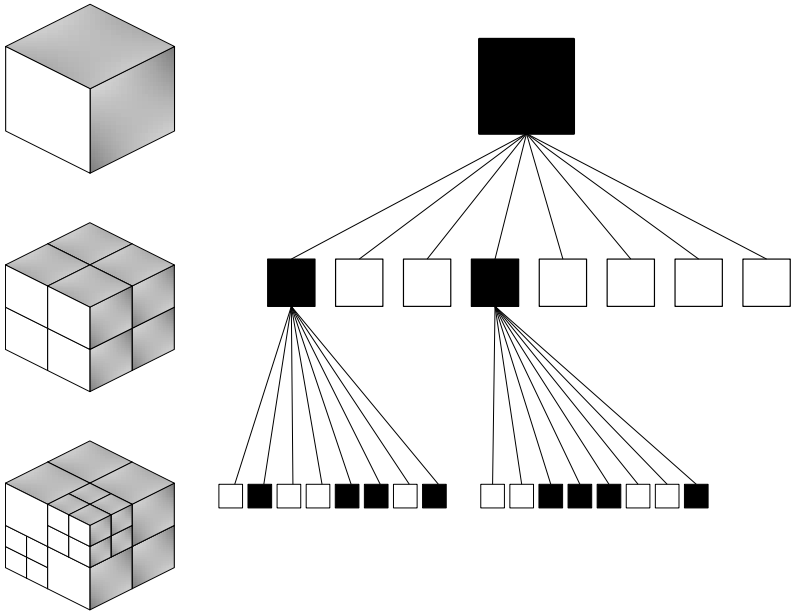
Obr. 2.4: Meniaci sa riedky priestor po aplikovaní CSG operácií.

2.4 Hierarchické delenie priestoru

Spôsob rozdeľovania daného priestoru na menšie neprekrývajúce sa podpriestory sa hojne využíva nielen vo vykreslovacích algoritmoch. Rozdelený priestor dokáže uľahčiť a urýchliť prácu hierarchickým vyraďovaním ostatných objektov. Pre určité prípady typicky dobre poslúži binárny strom/Kd-strom.

Špeciálnym druhom štruktúry je oktálový strom. Ten hierarchicky rozdeľuje daný priestor do ôsmich rovnako veľkých podpriestorov nazývaných oktety. Pre každý podstrom až po najnižšiu úroveň sa operácia opakuje (rekurzívny prístup). Veľmi často nie všetky ok-

tety obsahujú niečo iné než prázdny priestor. Z tohto dôvodu existuje rozšírenie oktálového stromu: *riedky* oktálový strom (názorne na obrázku 2.5). Typicky pri organizovaní obsiahnutých objektov do oktálového stromu sú na ne v jeho listoch uložené odpovedajúce referencie. V objemovej terminológii však listy stromu sú práve voxely, preto sa v tejto oblasti používa označenie riedky voxelový oktálový strom (SVO) [12]. Oktálové stromy podporujú veľmi jednoduchý spôsob prepnutia rozlíšenia detailu zmenou hĺbky zanorenia do štruktúry. Ak sa pracuje so skalárnymi hodnotami, možno hodnoty dedičného uzlu v hrubšej úrovni vypočítať podľa priemeru (binárnych) hodnôt jeho potomkov, poskytujúc faktor pokrytia. To dokáže byť užitočné v rôznych situáciách a zrýchliť výpočet vykreslenia/spracovania scény. Naviac možno obmedziť veľkosť prenášaných dát na grafickú kartu, kedy riedšie podstromy nie je potreba analyzovať. Nedávna doba priniesela technológiu virtuálnych textúr – tie takéto záležitosti uľahčujú [13].



Obr. 2.5: Riedky oktálový strom: iba záujmové oblasti sú ďalej rozdelené.

Výstavba SVO behom voxelizácie na GPU býva netriviálnou záležitosťou. Hľadané riešenie sa týka problému alokácie iba vyžadovaných oktetov (kompakcia voxelového priestoru), do ktorých padne výsledná voxelizácia najjemnejšej úrovne. Základom riešenia tejto problematiky je otázka definície SVO s ohľadom na minimalizáciu vyžadovanej pamäte potrebnej pre jeho uloženie. Oktety sú referované ukazovateľmi svojich priamych predkov. Naívna implementácia ukladá ukazovatele pre všetkých potomkov, možno však využiť usporiadanie stromu v pamäti tak, aby bol celý podstrom uložený lineárne (za sebou), vďaka čomu je potrebné uchovať len informáciu o prvom uzle. Podobne podľa topologického členenia je výhodné alokovanie stromu rozdeliť na bloky obsahujúce iba uzly tej istej úrovne. Dodatočná informácia o ofsetoch do týchto úrovní môže byť pridaná, ak je strom uložený súvisle v 1D poli. Binárna voxelizácia vyžaduje zakódovanie stavu jediným bitom, ktorý môže byť súčasťou hodnoty, v ktorej sa vyskytuje ukazovateľ. Ak je uvažovaný povrchový druh voxelizácie do SVO, konverzia by sa v najlepšom prípade mala týkať iba najjemnejšej úrovne – teda uchovávať informáciu o stave v listových uzloch. Prípadné prepínanie úrovne detailu dopočíta faktor pokrytia za behu. Zložitejšia je úplná voxelizácia. Vzhľadom na to, že SVO vynecháva prázdne priestory, nie je možné využiť bežný postup propagovania

stavovej informácie vo voxelovom stĺpe ako v pôvodnej verzii bez delenia priestoru. Z toho dôvodu sa pre efektívnu úplnú voxelizáciu pridávajú do štruktúry uzlu pomocné ukazovatele v smere propagačnej osi [46]. Propagovanie do hrubších úrovní sa naviac domáha uloženia informácie o umiestení priamych predkov, uľahčujúc tak prechádzanie stromu a schopnosť vracať sa do vyšších úrovní. Úplná voxelizácia by mala taktiež zaručiť to, aby iba hrubšie úrovne definovali stav vnútra objektu. Kompaktnejšie riešenia existujú pre špecifické účely, napr. vykresľovanie, kedy sa do dedičného uzlu ukladá stav potomkov [35], použitelné aj v rámci binárnej voxelizácie. Ďalším ideálnym spôsobom redukcie vyžadovaného pamäťového miesta môže byť uloženie do tzv. *tehál* (objemovo napríklad 4^3), popisujúcich stav spodných úrovní, ktoré sú definované ako bitové masky s požadovanou dĺžkou [46]. Takto pamäťovo minimalizované nižšie úrovne ani nevyžadujú uloženie ukazovateľov.

Prechádzanie oktálového stromu

Ak je známa pozícia $\vec{p} = (i, j, k)^\intercal \in \Gamma$, adresovanie odpovedajúceho uzlu v SVO býva často požadovaný úkon. Riešením môže byť usporiadanie voxelovej reprezentácie v pamäti podľa tzv. z-krivky. z-krivka predstavuje funkciu mapujúcu N-dimenzionálny priestor do 1D podoby zachovávajúc priestorovú lokalitu dát. V roku 1966 túto techniku prvý-krát predviedol Guy Morton [40], preto zakódovaným 1D hodnotám sa často hovorí Mortonov kód. Mortonov kód bodu $\vec{p}$ je vypočítaný prekladaním bitov jednotlivých komponent $\vec{p}$, formálne definíciou v rovnici (2.2), kde n predstavuje bitovú dĺžku komponenty $\vec{p}$. Podľa poradia súradných osí počas prekladaní bitov možno docieliť iné usporiadanie priestoru, na čo musí byť braný ohľad pri práci so z-krivkou.

$$z(\vec{p}) = (k_{n-1}j_{n-1}i_{n-1}\dots k_0j_0i_0)_2 \tag{2.2}$$

Mortonove kódy sú používané pri práci s rôznymi štruktúrami ako binárne stromy. Na základe počtu dimenzií môže byť z-krivka použitá pre popis konzistentne deleného priestoru, kde počet podpriestorov dedičného uzla je mocninou dvoch. To je evidentne oktálový strom (obrázok 2.6). Nech m je Mortonov kód 3D pozície $\vec{p}$. Potom sa k uzlu v oktálovom strome O asociovaného s $\vec{p}$ možno dostať na základe pseudokódu algoritmu 1.

Algoritmus 1 Prechádzanie riedkeho oktálového stromu.

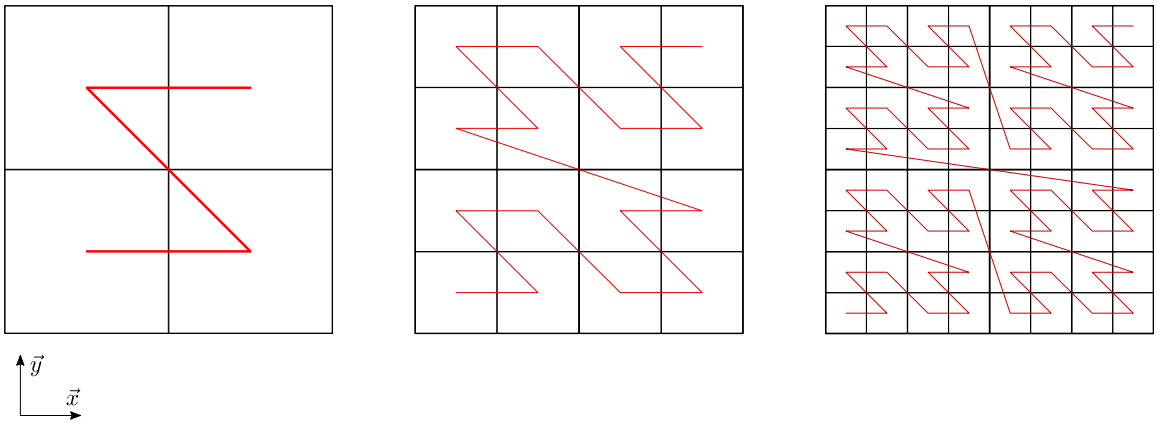
```
1: function Z-TRAVERSAL(m, O)
2:     n ← O_0
3:     for i ∈ {0, ..., DEPTH(O) − 2} do
4:         j ← (m ≫ 3 · (DEPTH(O) − i)) & (111)_2
5:         n ← O_{FIRST_CHILD(n)+j}
       return n
```

Na algoritme 1 je vidieť ekvivalentnosť k *depth-first* prechodu. Funkcia prechádza zhora postupne všetky úrovne O, okrem najspodnejšej. Adresovanie jednotlivých úrovní prebieha bitovým posunutím m vpravo o toľko miest, čo trojnásobok úrovní, na ktoré sa ešte nenarazilo. Výsledok je nakoniec upravený bitovým AND a použitý ako ofset od miesta prvého potomka uzla n. Týmto postupom sa dá rýchlo dopracovať k požadovanému uzlu na pozícii $\vec{p}$. U SVO bude pre úplnosť potreba ošetriť prípady, kedy sa na vstupu objaví Mortonov kód asociovaný s bodom v prázdnom (nealokovanom) priestore.

Ďalšou výhodnou operáciou pri vyjadrení priestoru podľa z-krivky je prechádzanie oktálového stromu zdola nahor. Ak je definovaný Mortonov kód m, potom jednoduchým bitovým

posuvom vpravo o 3 miesta dostať adresu na z-krivke odpovedajúceho priameho predka. Pri potrebe adresovať tento uzol v danej úrovni stačí odkázať sa naň podľa tohto indexu, avšak to neplatí u SVO, keďže daný podpriestor nemusí byť plne definovaný (preto sa typicky radšej používajú ukazovatele na dedičný uzol). Potomkovia so spoločným predkom vo vyššej úrovni majú po bitovom posuve Mortonové kódy totožné. Tým pádom nie je potrebné vždy prechádzať celý strom od jeho koreňa pri ďalšej adresácii spodnejšieho uzla, stačí len vzostúpiť do spoločného predka a odtiaľ pokračovať ku žiadanému potomkovi.

Uvedomenie si, že kardinalita voxelového priestoru určuje počet adresovacích bitov sa vyplatí, ak je potrebné nadefinovať jeho podpriestor. z-krivka je schopná popísať celý vymedzený rozsah, kde veľkosť dimenzie je jednotná vo všetkých osách. Pre 8-prvkovú množinu platí $\log_2 8 = 3$ bitov, v nižšej úrovni $\log_2 64 = 6$ apod. Vidieť to zasa na obrázku 2.6. Výhodou je relatívne adresovanie takéhoto podpriestoru počas voxelizácie, napr. keď sú jemné úrovne uložené vo voxelových tehlách.



Obr. 2.6: Mapovanie priestoru podľa z-krivky vo viacerých iteráciach. Za povšimnutie stojí hierarchia podobná kvartálovému (oktálovému) stromu.

Kapitola 3

Techniky voxelizácie

Algoritmy voxelizácie sú charakterizované vlastnosťami vysvetlenými v podkapitole 2.2. Implementácia nastavenia náležitosti voxela je podstatným kritériom v optimalizačných úlohách, aby dochádzalo na minimálne opakovanie jadra výpočtu. Taktiež by daná technika mala podporovať intuitívny prechod k úplnej voxelizácii. Zároveň je vhodné mať existujúci popis rozdelenia priestoru do SVO, čo býva viac komplikovaná časť. Nie je príliš podstatné, či diskutovaná metóda nastavuje skalárnu alebo len binárnu hodnotu stavu voxela, lebo typicky tie prvé sa dajú triviálne transformovať na binárnu voxelizáciu. Bežne sa voxelizácie s koreňmi viac vo výpočtovej geometrii odvodzujú od analogických 2D riešení, najčastejšie v oblasti 2D rasterizácie a i keď zakladanie na vykresľovacom reťazci grafických kariet je implementačne menej náročné, poskytujú GPGPU prístupy viac flexibility, robustnosti a chcený nárast rýchlosti. Taká voxelizácia netrpí problémami spojenými so vstavaným vykresľovacím hardvérovým reťazcom, konkrétne diery v mriežke na hraniciach objektov.

3.1 Historický vývoj

Objemovej grafike bola v minulosti venovaná veľká pozornosť a už počiatkom 80. rokov sa začali objavovať prvé spôsoby voxelizácie 3D geometrie [36, 8], založené napríklad na 3D rozšírení Bresenhamovho DDA algoritmu [33], konverzné algoritmy pre parametricky definované útvary (hlavne krivky a splajny) [32], zjemňovanie objektu až po voxelovú úroveň pomocou delenia polygonovej siete [7] a ďalšie, využívajúce rôzne typy separability k diskretizácii implicítnych plôch [39], podobne ako presná voxelizácia 3D úsečky [10] a trojuholníka s vyhodnotením skalárnej voxelizácie pre rôzne rozlíšenia a okamžitý anti-aliasing [16], založený na meraní vzdialenosti testovaného voxela od povrchu.

S príchodom bežne dostupných grafických kariet (90. roky) sa objavili voxelizácie postavené na využití vykresľovacieho reťazca, Z-bufferu a 3D textúr [6]. Na podobnom princípe fungovala technika [31]. Tá umožňovala úplnú voxelizáciu nastavením buniek 3D mriežky podľa náležitosti ich stredu do hĺbkových intervalov. Nevýhodou bolo vynechávanie konkávných častí modelu, nezaznamenané z dôvodu zakrytia inými časťami modelu. V roku 2000 autor práce [23] postupoval pri povrchovej voxelizácii krájaním objemového priestoru do z-rovín paralelných k priemetni. Každý krajec sa ukladal do odpovedajúcej časti 3D textúry. Úplná voxelizácia upotrebila XOR techniky miešania bariev pri prepnutí stavov voxelov. Výsledok však často neobsahoval jemné regióny a vyskytovali sa v ňom nekompletné diskretizácie hranice trojuholníka. Riešením bolo kombinovanie s povrchovou voxelizáciou

so zapnutým anti-aliasingom. Prínosom metódy [23] bol popis objemovej CSG operáciami miešania bariev v snímkovej pamäti.

Uvedením grafických kariet s programovateľnými vrcholovými/fragmentovými jednotkami, v práci [17] bola popísaná povrchová aj úplná voxelizácia aplikovaním rasterizácie scény. Braním v úvahu neexistujúcu podporu 3D textúr na vtedajšom komoditnom hardvéri, je voxelová mriežka reprezentovaná 2D RGBA textúrou s dostatočnou bitovou dĺžkou. Voxelizovanie prebiehalo do 3 pomocných textúr delených na podoblasti. Objem bol vygenerovaný postupne, oblasť po oblasti. Vrcholový shader transformoval vstupnú geometriu do voxelového priestoru, zatiaľ čo sa vykresľovala do roviny maximálnej projekcie. Rasterizáciou sa vyplnili oblasti pomocných textúr a tie boli nakoniec zhutnené do konečného objemu prvej 2D textúry. Úplná voxelizácia prebiehala vyplňovaním v jednotlivých krajcoch. Na podobných princípoch bol postavený algoritmus [21], zostrojujúci objem perspektívne. Rasterizácia prebiehala iba v smere sledovania scény, vyzerala jemnejšie, trpela však veľkými nepresnosťami, čo nevadilo pri vytváraní niektorých vizuálnych efektov. Technika [48] riešila problém presného mapovania fragmentov trojuholníka na celý obsiahnutý voxelový stĺp. Predchádzajúce postupy boli schopné nastaviť len voxely, ktoré odpovedali rasterizovaným fragmentom. Riešením bolo vypočítať minimálnu a maximálnu hĺbku pre každý fragment. Hĺbkový rozsah voxelov stĺpca bol počítaný na základe prekrytia fragmentu s premietaným trojuholníkom. Toto prekrytie definuje 2D polygon (maximálne 7-hranový), ktorého každý vrchol je preskúmaný pri zisťovaní limít.

Tieto implementácie boli výrazne obmedzené možnosťami vtedajších programovacích rozhraní. Veľkým problémom (i dnešných) rasterizačných jednotiek grafických kariet vo voxelizácii je, že spracovávané fragmenty sú len na tých pixeloch, ktorých stred bol prekrytý rovinou trojuholníka. To do výsledku voxelizácie uvádza diery. Tento problém v minulosti riešilo aplikovanie anti-aliasingu [23], čo typicky zabraňuje minimalite riešenia (objavujú sa 26-prosté voxely). Zároveň nedokonalé mapovanie na pixely pod veľkým úhlom medzi smerom pohľadu pozorovateľa a normálou trojuholníka spôsobuje trhliny vo voxelizácii. Identifikácia takýchto prípadov funguje vypočítaním gradientov hĺbky fragmentu a skontrolovaním, že táto hodnota neprekročí 1. Ak áno, riešenie spočíva vo vykreslení podľa dominantnej osi trojuholníka.

3.2 Moderné paralelné algoritmy voxelizácie

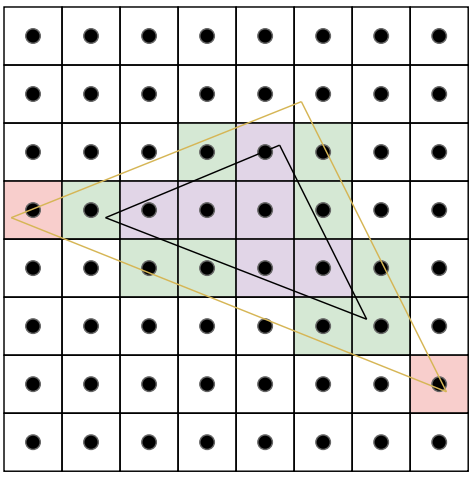
Veľa sľubných algoritmov voxelizácie s ohľadom na rýchlosť, minimalizáciu a pamäťové požiadavky bolo v nedávnej dobe navrhnutých. Mnoho scenárov požaduje aby bola voxelizácia presná a uskutočniteľná za behu.

Problém optimalizácie využívania zdrojov grafikej karty bol analyzovaný v technickom článku [13]. Cieľom je využiť výhod voxelovej reprezentácie pre poskytnutie veľmi detailne vykreslenej grafiky s nízkou záťažou na hardvér a predovšetkým šetriť video pamäťou. Ideálny kandidát pre túto úlohu je SVO. Pokrytý priestor definujú tehly určitej veľkosti a všetky sú uložené v 3D textúre. Prínosom tejto práce je predovšetkým aktualizácia pamäte pre momentálny pohľad do scény (idea *out-of-core*) a spôsob uloženia dát (vrátane skalárnych veličín).

Inovatívny prístup k povrchovej/úplnej riedkej binárnej voxelizácii prišiel s nástupom GPGPU programovacích rozhraní (CUDA/OpenCL) v článku [46]. Nevyužíva sa vstavaný hardvérový vykreslovací reťazec GPU, ale metóda prispôsobená požiadavkom paralelnej voxelizácie. Test náležitosti je optimalizovaným rozšírením teorému oddeľujúcej osi kvádra-trojuholníka [1]. Pre kompaktnú voxelizáciu taktiež používa SVO, vybudovaného za účasti

Mortonových kódov a iných paralelných algoritmov (exkluzívny sken, kompakcia). Táto technika je podrobnejšie preskúmaná v podkapitole 3.2.

Návrh [12] poskytuje voxelizáciu spĺňajúcu kritéria 6-separability i delenia priestoru. Nastavovanie voxelov vychádza z postupu [1] a zároveň využíva vykreslovací reťazec grafických kariet. Podľa dominantnej osi autori zvolia projekciu trojuholníka v geometrickom shaderi. Rasterizácia modelu prebieha v rozlíšení voxelovej mriežky. Veľa per-fragmentových operácii je vypnutých pre zlepšenie výkonu a spolieha sa na priamy prístup do textúry bez zapisovania do mimo-obrazovkovej snímkovej pamäti. Riešenie [12] sa tiež potýka s problémom dier, vznikajúcich pri rasterizácii. Riešenie spočíva v rozšírení trojuholníka geometrickým shaderom 3.1 a orezaní fragmentov (ktoré boli prekryté nadbytočne) osovo-zarovnaným hraničným kvádrom pôvodných súradníc vrcholov trojuholníka. Pre delenie priestoru je použitý SVO definovaný podobne ako v práci [13], kde skalárne veličiny sú vzorkované z 3D textúry pomocou trilineárnej interpolácie. Voxelizácia do SVO najprv prebehne iba pre rozlíšenie najnižšej úrovne, vytvárajúc voxel-fragment zoznam, ďalej použitý na rozdelenie priestoru a alokáciu vyšších úrovní. Zoznam obsahuje pozíciu a (lubovoľne) odpovedajúce skalárne hodnoty. Proces zapisovania do zoznamu vyžaduje atomicky aktualizovať voľnú pozíciu v pomocnej globálnej premennej. Pre vyššie úrovne prebieha označenie aktívnych uzlov a ich následná alokácia ôsmich potomkov označených voxelov. Výhodou tejto implementácie je réžia štádií pomocou nepriamej vykreslovaciej pamäte, do ktorej sa zapisuje špeciálnym shaderom za behu bez dirigovania procesorom. To poskytuje obrovskú výhodu – proces nie je zaťažovaný hostiteľskou aplikáciou. Na druhú stranu, nepríjemnosťou ostáva predalokácia pamäti pre oktálový strom, kedy ešte nie je známa jeho veľkosť [12].



Obr. 3.1: Zväčšenie primitíva geometrickým shaderom: potenciálne sa objavujú N-prosté voxely (ružové pixely).

Práca [42] predkladá reťazec vizualizácie voxelizovanej scény za behu. Obsahuje programovateľné časti vrcholového a fragmentového shaderu. Ich voxelizácia stavia na riešení [46] a lepšie vyvažuje záťaž na multiprocesoroch dlaždicovým prístupom, kedy sú trojuholníky vstupnej siete asociované odpovedajúcim dlaždiciam, zvlášť spracované jedným vlánkom za druhého behu (prevencia nevyrovnanej záťaže v situáciach trojuholníkov rôznych veľkostí). Potom dochádza k rozdeleniu na tzv. virtuálne dlaždice, pre prípady, kedy pôvodné obsahujú veľký počet trojuholníkov na spracovanie.

Za zmienku stojí implementácia [44] pre povrchovú voxelizáciu. Aplikuje postup [46] vo vykresľovacom reťazci. Faktorizovaním výpočtov je práca prerozdelená medzi shader

jednotky – vrcholový shader prevádza vstupné primitívum do voxelového priestoru, geometrický shader predpočíta termy testovania náležitosti nezávislé od pozície stredu voxela a fragmentový shader nakoniec vykonáva testovaciu časť výpočtu. Daný systém taktiež umožňuje vyvarovať sa vyššej záťaže vlákien väčších trojuholníkov medzi malými. Sledovaním veľkosti 2D projekcie trojuholníka, takéto prípady prepnú na paralelizmus vlákien podľa fragmentov (dlaždíc). Veľmi dobre zároveň redukuje počet 1D, 2D a 3D prípadov [44, 46] zamenením súradních podľa dominantnej osi trojuholníka.

Prispôsobená GPU voxelizácia

Následujúci výklad cituje [46, 45]. Ako bolo spomenuté, táto technika prevodu na SVO aplikuje voxelizáciu definovaním vlastnej funkcie testovania náležitosti trojuholníka do voxelovej oblasti – nahrádza tým zabudovanú rasterizáciu z väčšiny predchádzajúcich metód. Vyhýba sa tak problémom vzniku dier (i N-prostých voxelov) a je schopná robustne definovať taktiež 6-separabílnu voxelizáciu. Naviac netrpí neduhom pred-alokácie SVO ako rýchly [12]. V zjednodušenom poňatí sa daný algoritmus rozdeľuje do troch (resp. štyroch) štádií:

1. zistenie a alokovanie aktívnych oktetov najnižšej úrovne,
2. rekurzívne vybudovanie SVO podľa skrátených Mortonových kódov predchádzajúcej úrovne,
3. povrchová/úplná voxelizácia do vybudovaného SVO v najjemnejšom rozlíšení,
4. dodatočne: propagovanie bitových prepnutí medzi úrovňami SVO.

Test náležitosti

Na začiatku musí byť vstupná geometria transformovaná do voxelového priestoru Σ. Zistenie prekrytia sa organizuje do dvoch štádií: prípravy — kedy nastáva predpočítanie termov závislých len na atribútoch testovaného trojuholníka — a testovacej fázi, ktorá kontroluje danú objemovú oblasť na pozitívny prípad prekrytia. Ďalej formálne, nech existuje trojuholník τ identifikovaný vrcholmi $\vec{v}_i \in \Sigma$ pre $i \in \{0, 1, 2\}$ a osovo-zarovananá kocka voxela ν so spodným rohom $\vec{p} \in \Gamma$. Najprv popísaná je 26-separabílna voxelizácia, pri ktorej platí, že ν je nastavený, ak ho τ i len čiastočne prekrýva:

a) rovina τ musí presiahnuť ν (3D test),

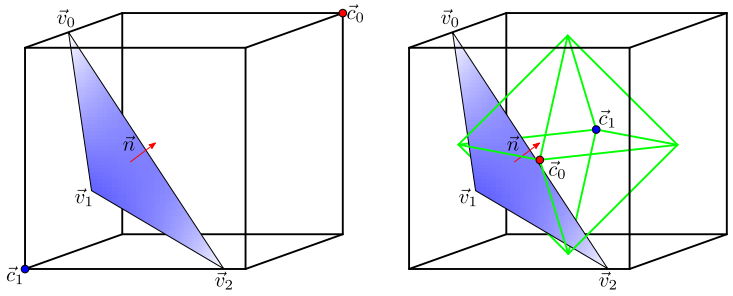
b) 2D projekcie τ a ν vo všetkých súradných rovinách (xy, yz, zx) sa prekrývajú.

Zisťovanie či rovina prekrýva kocku sa robí za pomoci tzv. kritického bodu

$$\vec{c}_0 = \left(\begin{Bmatrix} \delta, & \vec{n}_x > 0 \\ 0, & \vec{n}_x \leq 0 \end{Bmatrix}, \begin{Bmatrix} \delta, & \vec{n}_y > 0 \\ 0, & \vec{n}_y \leq 0 \end{Bmatrix}, \begin{Bmatrix} \delta, & \vec{n}_z > 0 \\ 0, & \vec{n}_z \leq 0 \end{Bmatrix} \right)$$

určeného normálou $\vec{n}$ daného trojuholníka. Tento bod je vyjadrený relatívne k $\vec{p}$. Bod $\vec{c}_1$ sa potom nachádza na rohu opačnom k $\vec{c}_0$. Dosadením týchto bodov do rovnice pre rovinu a prenásobením výsledkov možno zostrojiť test (obrázok 3.2), definujúci diskutované prekrytie

$$\begin{aligned} \langle \vec{n}, (\vec{p} + \vec{c}_0 - \vec{v}_0) \rangle \cdot \langle \vec{n}, (\vec{p} + \vec{c}_1 - \vec{v}_0) \rangle \leq 0 \\ (\langle \vec{n}, \vec{p} \rangle + d_0) \cdot (\langle \vec{n}, \vec{p} \rangle + d_1) \leq 0, \end{aligned} \tag{3.1}$$



Obr. 3.2: 3D test kritických bodov $\vec{c}_0$, $\vec{c}_1$: vľavo konzervatívna, vpravo 6-separabílna voxelizácia.

kde $d_i = \langle \vec{n}, \vec{c}_i - \vec{v}_0 \rangle$ je term po faktorizovaní, aby tento výpočet nemusel prebiehať viackrát pre každý voxel.

Nakoniec treba skontrolovať prekrytie τ, ν vo všetkých troch vzájomne kolmých 2D projekciách. Juan Pineda predstavil spôsob, akým toto realizovať na základe tzv. hranových funkcií [43]. Nech uvažovaný test vychádza z hranovej funkcie

$$g^{\mathrm{xy}}(\vec{p}_{xy} + \vec{b}_i) = \langle \vec{n}^{\mathrm{xy}}_{\vec{e}_i}, \vec{p}_{xy} + \vec{b}_i - \vec{v}_{i,xy} \rangle \geq 0, \tag{3.2}$$

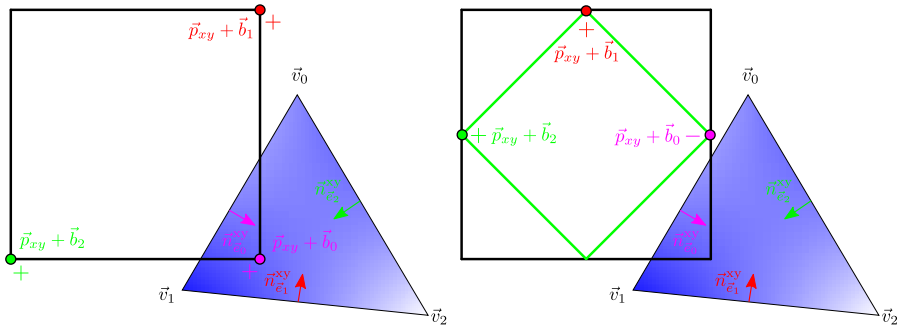
kde $\vec{n}^{\mathrm{xy}}_{\vec{e}_i}$ predstavuje normálu hrany $\vec{e}_i = \vec{v}_{i+1 \bmod 3} - \vec{v}_i$ v xy projekcii podľa (3.3). $\vec{n}^{\mathrm{xy}}_{\vec{e}_i}$ determinuje kritický bod $\vec{b}_i$, pre ktorý je (3.2) vyhodnotená (viz obrázok 3.3). Následne $\vec{p}_{xy} + \vec{b}_i$ je vo vnútri trojuholníka, ak pre všetky hrany $\vec{e}_i$ platí podmienka (3.2). Toto musí platiť v každej uvažovanej rovine (xy, yz, zx), aby bolo možné prehlásiť τ prekrýva ν. Faktorizáciou g^{xy} potom ďalej dostať

$$g^{\mathrm{xy}}(\vec{p}_{xy} + \vec{b}_i) = \langle \vec{n}^{\mathrm{xy}}_{\vec{e}_i}, \vec{p}_{xy} \rangle + d^{\mathrm{xy}}_{\vec{e}_i}$$

s $d^{\mathrm{xy}}_{\vec{e}_i}$ podľa (3.4).

$$\vec{n}^{\mathrm{xy}}_{\vec{e}_i} = (-\vec{e}_{i,y}, \vec{e}_{i,x})^{\intercal} \cdot \left\{ \begin{array}{ll} 1, & \vec{n}_z \geq 0 \\ -1, & \vec{n}_z < 0 \end{array} \right\}, \tag{3.3}$$

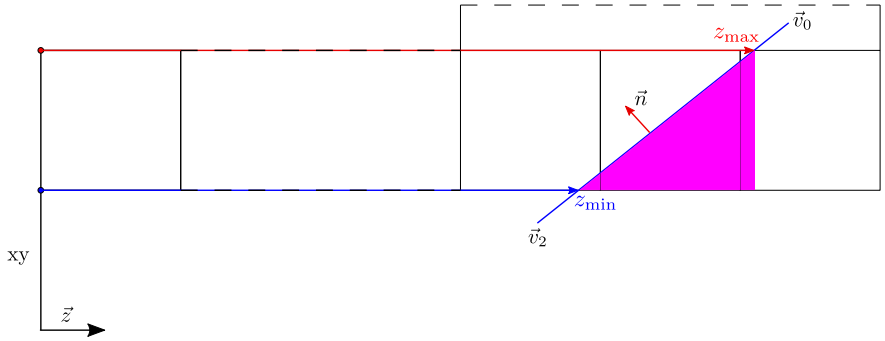
$$d^{\mathrm{xy}}_{\vec{e}_i} = -\langle \vec{n}^{\mathrm{xy}}_{\vec{e}_i}, \vec{v}_{i,xy} \rangle + \max\left\{0, \vec{b}_{i,x} \cdot \vec{n}^{\mathrm{xy}}_{\vec{e}_i,x}\right\} + \max\left\{0, \vec{b}_{i,y} \cdot \vec{n}^{\mathrm{xy}}_{\vec{e}_i,y}\right\} \tag{3.4}$$



Obr. 3.3: 2D test kritických bodov $\vec{p}_{xy} + \vec{b}_i$ v rovine xy: vľavo konzervatívna, vpravo 6-separabílna voxelizácia.

Je vidno, že $\vec{n}$, d_0, d_1, $\vec{n}^{xy}_{\vec{e}_i}$, $d^{xy}_{\vec{e}_i}$, $\vec{n}^{yz}_{\vec{e}_i}$, $d^{yz}_{\vec{e}_i}$, $\vec{n}^{zx}_{\vec{e}_i}$, $d^{zx}_{\vec{e}_i}$ môžu byť predpočítané vo fázi prípravy. Skutočný test nakoniec zostáva na (3.1) a (3.2) podľa pozície $\vec{p}$ v diskrétnej reprezentácii osovo-zarovnaného hraničného kvádra τ. Existuje niekoľko kandidátnych optimalizačných prípadov. Jedným z nich je špecializácia celého testu pre obmedzenie počtu testovaných voxelov. Kľúčovým poznatkom je (keďže trojuholník je rovinný útvar), že voxelizácia môže prebehnúť len pre maximálne 3 voxely v smere dominatnej osi trojuholníka. Potom teda jestvuje riešenie pre každú súradnú osu, kedy najprv prebehne test 2D projekcie útvarov v počiatočnej rovine. Ak test prejde, sú podľa $\vec{n}_x$ a $\vec{n}_y$ identifikované kritické body štvorca 2D projekcie. Následným sledovaním paprsku na rovinu τ (viz (3.5), kedy $\vec{z} = (0, 0, 1)^\intercal$, pre zistenie $\vec{p}_z$) je definovaný interval potenciálne prekrytých voxelov, pre ktoré sa uskutočnia zostávajúce 2D testy (obrázok 3.4). Za povšimnutie stojí fakt, že už ďalej nenastáva test (3.1), pretože z princípu sú kontrolované len voxely, ktoré sú v intervale definovanom rovinou τ.

$$\vec{p}_z = \frac{\langle \vec{v}_0 - (\vec{p}_{xy}, 0)^\intercal, \vec{n} \rangle}{\langle \vec{n}, \vec{z} \rangle} \tag{3.5}$$



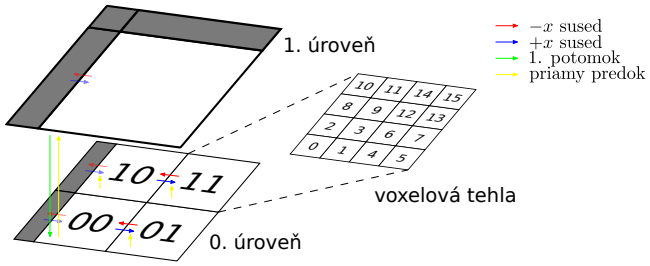
Obr. 3.4: Zisťovanie hĺbkového intervalu kandidátnych voxelov na prekrytie premietnutím na rovinu trojuholníka podľa osi $\vec{z}$.

Vyplýva, že celkový počet prípadov 1D testov (hraničné kvádre), 2D testov (projekcie na rovinu) a 3D testov (rovnica roviny, resp. sledovanie paprskov) modifikovanej verzie je 9. To pri implementácii znamená komplikované vetvenie programu na základe detekovanej dominantnej osi. Výsledkom by bolo písanie redundantného kódu (kvôli zmenám pri adresovaní vektorových komponent) a pravdepodobne vysoká divergencia vlákien za behu. Existuje však elegantnejšie riešenie: zamenením (*swizzlením*) vektorových komponent podľa dominantnej osi možno docieliť adresovanie len v jednom nastavení a teda zjednodušiť počet prípadov na 3. Treba však nezabudnúť na to, že výsledná pozícia v Γ musí byť náležito modifikovaná [44]. Medzi ďalšie optimalizácie ešte patrí:

- ak pre hĺbkový rozsah voxelového stĺpca platí, že obsahuje len 1 voxel, potom tento voxel je nastavený,
- ak osovo-zarovnaný hraničný kváder τ je len 1 voxel tlstý, je celý test zjednodušený do jedinej kontroly v (aktuálnej) 2D rovine,
- nakoniec, ak osovo-zarovnaný hraničný kváder τ zahŕňa v dvoch smeroch len 1 voxel, všetky voxely v ňom majú byť nastavené.

Vybudovanie SVO

Vyššie popísaná (optimalizovaná) povrchová binárna voxelizácia je prvým krokom vybudovania SVO. Jeho štruktúra vyzerá ako na obrázku 3.5. Najnižšia úroveň (0) uchováva voxelovú tehlu o veľkosti 4^3, kde sú voxely usporiadané podľa z-krivky, reprezentujúc tým dve najjemnejšie úrovne. Tieto tehly môžu byť zakódované pomocou dvoch 32-bitových celočíselných hodnôt a šetriť tak miestom vo video pamäti. Všetky uzly v jeden úrovni sú uložené v súvislom 1D poli a taktiež usporiadané na základe Mortonovho vyplňovania priestoru. To znamená, že index prvého potomka dedičného uzlu je násobkom ôsmich. Navyše ku existencii ukazovateľov na prvého priameho potomka a priameho predka je uchovaná informácia o susednosti uzlov v kladnom/zápornom x-ovom smere pre zefektívnenie pohybu štruktúrou pri úplnej voxelizácii.



Obr. 3.5: Návrh SVO (2D analógia). Nulté uzly reprezentujú 2 najjemnejšie úrovne.

Na začiatku je potrebné určiť všetky *aktívne* uzly nultej úrovne, do ktorých prebehne voxelizácia. Keďže uzol SVO má buď 0 alebo 8 potomkov, konzervatívnou voxelizáciou úrovne 1 ($\delta = 8$) možno efektívne určiť túto oblasť. Konverzia zapisuje do poľa 32-bitových hodnôt (použitím hardvérovej atomickej OR operácie). Aby neskoršia voxelizácia nultej úrovne prebehla správne pre všetky hraničné regiony (uzol $*$ na obrázku 3.6), je treba upraviť predchádzajúci algoritmus determinovania náležitosti trojuholníka vo voxelovom priestore. Najskôr je geometria posunutá v kladnom x-ovom smere o polovicu jednotkovej veľkosti, zodpovedajúc za vzorkovanie na voxelovom strede. Ďalej, hraničný kváder trojuholníka je rozšírený v zápornom smere osi $\vec{x}$ o jednotkovú veľkosť. Zároveň je treba upraviť (3.4) posunutím všetkých kritických bodov, ležiacich na +x stene, t.j. $\vec{b}_{i,x} = \delta$,

$$d^{\text{xy}}_{\vec{e}_i} = -\langle \vec{n}^{\text{xy}}_{\vec{e}_i}, \vec{v}_{i,xy} \rangle + \max\left\{0, (\vec{b}_{i,x} + \vec{\chi}_x) \cdot \vec{n}^{\text{xy}}_{\vec{e}_i,x}\right\} + \max\left\{0, (\vec{b}_{i,y} + \vec{\chi}_y) \cdot \vec{n}^{\text{xy}}_{\vec{e}_i,y}\right\}, \qquad (3.6)$$

kde χ je vektor s jedinou komponentov rovnou 1 podľa toho, aká osa je dominantná pre τ.

Po určení aktívnych voxelov v prvej úrovni nastáva vytvorenie zoznamu ich Mortonových kódov ActNodes_1. Najprv sa pre každý 32-bitový blok voxelizácie spustí vlákno počítajúce nastavené bity. Nad týmto novým bufferom je urobený exkluzívny sken [27], výsledkom čoho je zoznam ofsetov do ActNodes_1 a dĺžka tohto zoznamu je posledný člen skenu sčítaný s posledným prvkom bitových počtov. Znovu sa paralelne spustí vlákno pre každý 32-bitový blok, aby konečne zapísalo vypočítané Mortonove kódy nastavených voxelov na správne miesta v ActNodes_1.

Po zistení aktívnych uzlov prvej úrovne je možné vytvoriť SVO zdola nahor. Najprv sa alokuje $8 \cdot |\text{ActNodes}_1|$ uzlov nultej úrovne Nodes_0, ktorým sa nainicializujú x-ukazovatele na susedné uzly s rovnakým priamym predkom. Porovnaním $m_i \gg 3 = m_{i+1} \gg 3$, kde m_i je Mortonov kód, sa paralelne pre každý aktívny uzol určí, či zdieľa priameho predka s m_{i+1}, zápisom 0, ak áno, alebo 1, ak nie, do nového zoznamu Flag_1. Nad Flag_1 sa potom uskutoční

exkluzívny sken, dávajúc buffer $\textsc{ParentIndex}_1$. Jeho posledný člen plus jedna definuje počet aktívnych uzlov druhej úrovne N_2^a. Toto číslo je použité pri alokácii $8 \cdot N_2^a$ uzlov prvej úrovne $\textsc{Nodes}_1$. Podobne ako pri predchádzajúcej inicializácii, x-ukazovatele vrátane indexov na prvého potomka a priameho predka v nultej úrovni sú nastavené. Tu koresponduje i-tý prvok $\textsc{ActNodes}_1$ k $\textsc{Nodes}_1[j]$, kde $j = 8 \cdot \textsc{ParentIndex}_1[i] + (\textsc{ActNodes}_1[i] \bmod 8)$ a berie $\textsc{Nodes}_0[8 \cdot i]$ za svôjho prvého potomka.

Skrátením Mortonových kódov $\textsc{ActNodes}_1$ a vykonaním kompakcie [4] za pomoci $\textsc{Flag}_1$ a $\textsc{ParentIndex}_1$ sa ľahko dopracovať k $\textsc{ActNodes}_2$. Tento zoznam je ďalej spracovaný podobne ako predtým $\textsc{ActNodes}_1$. Proces budovania SVO sa môže skončiť v úrovni $N-3$, kde N je výška stromu, pretože logicky všetky uzly v troch najvyšších úrovniach musia byť alokované. Nakoniec je potreba ešte propagovať x-ukazovatele medzi uzlami v úrovni $i-1$ z vyššej úrovne i.

Voxelizácia do SVO

Hneď ako je SVO vybudovaný, začína sa voxelizácia najjemnejšej úrovne ($\delta = 1$), paralelne pre primitíva modelu. V rámci povrchovej voxelizácie je možné pre jemnejšiu diskretizáciu aplikovať 6-separabílnu modifikáciu. V praxi to znamená úpravu testu náležitosti zmenou planárneho testu

$$\begin{aligned} d_0 &= \langle \vec{n}, \tfrac{1}{2} - \vec{v}_0 \rangle + \tfrac{1}{2} \left| \vec{n}_\diamondsuit \right| \\ d_1 &= \langle \vec{n}, \tfrac{1}{2} - \vec{v}_0 \rangle - \tfrac{1}{2} \left| \vec{n}_\diamondsuit \right| \end{aligned} \quad \text{kde } \diamondsuit = \arg\max_{\square=x,y,z} |\vec{n}_\square|$$

(vzhľadom pripomínajúcim diamant, teda oktaedr – obrázok 3.2) a pre 2D test

$$d^{\text{xy}}_{\vec{e}_i} = \langle \vec{n}^{\text{xy}}_{\vec{e}_i}, \tfrac{1}{2} - \vec{v}_{i,xy} \rangle + \tfrac{1}{2} \left| \vec{n}^{\text{xy}}_{\vec{e}_i,\diamondsuit} \right|, \quad \text{kde } \diamondsuit = \arg\max_{\square=x,y} \left| \vec{n}^{\text{xy}}_{\vec{e}_i,\square} \right|.$$

Braním v úvahu dominantnú osu trojuholníka, je vyžadované adaptovať zisťovanie intervalu potenciálne prekrytých voxelov pre túto modifikáciu. Konkrétne to znamená premietnuť stred voxelového stĺpca (v xy rovine) pozdĺž z-ovej osi na rovinu τ. Necelá hodnota je potom zaokrúhlená nadol a nahor, identifikujúc potenciálne prekryté voxely. Iba v prípade, že 2 voxely sú dotknuté touto projekciou, viac než jeden voxel je braný v úvahu počas zostávajúcich 2D testov.

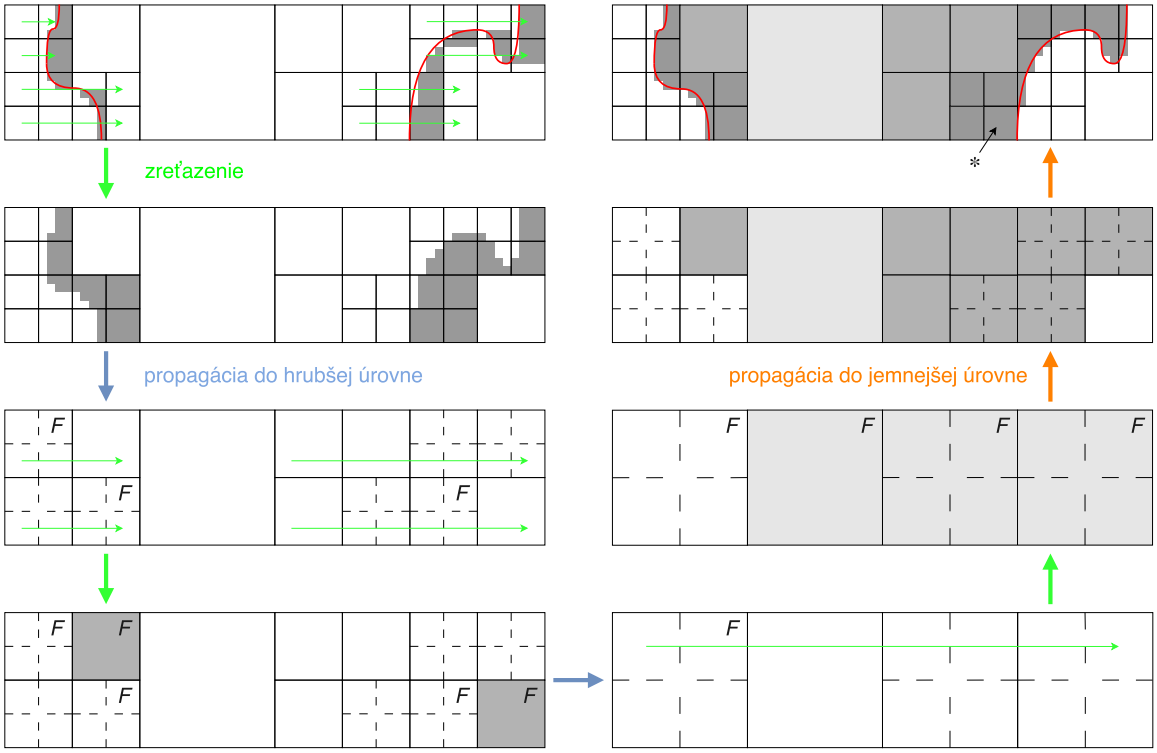
Ak je daný voxel nastavený, zostúpi sa podľa algoritmu prechádzania oktálového stromu podľa Mortonovho kódu m (algoritmus 1) k uzlu nultej úrovne, ktorý obsahuje tento voxel. $m \,\&\, (111111)_2$ definuje pozíciu bitu v odpovedajúcej voxelovej tehle. Podľa $m \gg 3 \,\&\, (111)_2$ sa identifikuje 32-bitový blok, zatiaľ čo $m \,\&\, (111)_2$ informuje o tom, ktorý bit v $2\times2\times2$ kocke sa nastavuje.

Úplná voxelizácia je oveľa komplikovanejšia. Najprv 2D rasterizácia do yz roviny zistí vyplnenie nultej úrovne. Jedná sa o 2-urovňový postup, kedy je najprv premietnutý trojuholník τ testovaný na prekrytie 4×4 oblasti. Ak τ prekrýva yz 2D projekciu tejto tehly (dlaždice), $d^{\text{yz}}_{\vec{e}_i} = -\langle \vec{n}^{\text{yz}}_{\vec{e}_i}, \vec{v}_{i,yz} \rangle$ je použitá v rámci kontroly voxelového stredu na prekrytie každého štvorca ν v lokálnom priestore dlaždice. Ak test prejde, stred ν je premietnutý na rovinu τ pozdĺž osi $\vec{x}$ (rovnica (3.5)). Výsledná x súradnica q udáva rozsah $\left\lfloor q + \frac{1}{2} \right\rfloor \coloneqq \bar{q} \bmod 4, \ldots, 3$ všetkých voxelov v lokálnom priestore aktuálne skúmanej tehly, ktorých stavový bit má byť prepnutý atomickou operáciou XOR .

Po voxelizácii do nultej úrovne začína hierarchická propagácia v kladnom smere osi $\vec{x}$. Ak posledný bit vo voxelovom stĺpci uzla nultej úrovne je nastavený, potom všetky voxely uzlov najjemnejšej úrovne v tomto stĺpci musia byť prepnuté. Paralelná iterácia prebieha

pre všetky uzly nultej úrovne s párnym indexom v danom 1D poli. Ak jeho záporný x-ukazovateľ nie je platný, je daný uzol počiatkom voxelového stĺpca, v ktorom prebieha táto procedúra. Na konci takto zreťazených voxelov sú v uzloch nultej úrovne s rovnakým priamym predkom koncové bity (v kladom smere x-ovej osi) zhodné.

Táto informácia slúži pri propagovaní do vyššej úrovne. V nej sú pre všetky uzly nadefinované bity FI, kde prvý bit identifikuje stav prepnutia a I výplň vnútra. Maska FI je na začiatku nastavená na $(00)_2$. Potom pre všetky uzly 1. úrovne, ak pre ktorýkoľvek z potomkov aktuálne skúmaného uzla platí, že majú nastavené tehlové voxely s lokálnym x indexom 3, nastaví sa bit F. Prepínacia informácia je znovu propagovaná v kladnom smere osi $\vec{x}$: keď $F = (1)_2$, potom $FI \hat{} (11)_2$ je nová hodnota masky susediaceho uzla (viď obrázok 3.6).



Obr. 3.6: Ilustrácia úplnej voxelizácie do SVO propagovaním medzi úrovňami.

Analogicky prebieha propagácia do ďalších úrovní, nakoľko dochádza k sledovaniu situácií, kedy je bit F nastavený v niektorom z uzlov dotýkajúcich sa v smere osi $\vec{x}$ nedelených miest vyššej úrovne. Ak k tomu dôjde, nastaví sa tiež bit F v tejto úrovni a pokračuje sa jej zreťazením. Ku koncu je I bit propagovaný zhora dolu. Pre každý uzol aktuálnej úrovne, ktorého I bit je nastavený a má potomkov, ich bit I alebo bitový stav vo voxelovej tehle je prepnutý a vlastný bit I spracovávaného uzla nastavený na 0.

Kapitola 4

Rýchla extrakcia hladkej izoplochy

Objemové dáta sú často užitočnou internou reprezentáciou pre rôzne geometrické úkony, avšak veľakrát nie je potrebné ich po spracovaní ďalej uchovávať a vystačí sa i s povrchovou reprezentáciou. Tá sa v tejto problematike uvádza ako izoplocha, pretože aproximuje hranicu objemu izohodnotou podľa hodnôt skalárneho poľa. Táto kapitola navrhuje riešenie rýchlej aproximácie hladkej izoplochy podvzorkovaním navrhnutého SVO (viz podkapitola 3.2) a zároveň vyhýbaniu sa oblastí voxelového priestoru, v ktorých nemôže dochádzať k prieniku s izoplochou a teda šetrí pamäťovým miestom i znižuje výpočtové nároky.

Problém extrakcie izoplochy bol už v roku 1987 skvele vyriešený pomocou slavného algoritmu *Marching Cubes* (MC) [37]. Zjednodušene povedané: táto metóda prechádza diskrétnu voxelovú mriežku abštraktnými kockami, ktorej vrcholy sú zarovnané na pozíciu odpovedajúcich objemových elementov, a vyhodnocuje skalárne hodnoty v týchto rohoch pre vstupnú izohodnotu, označujúc tak vrcholy kocky podľa toho, či sú vnútri či vonku od izoplochy. Podľa tohto označenia sú identifikované hrany obsahujúce vrcholy získaných trojuholníkov – presná pozícia v priestore je definovaná výsledkom interpolácie medzi susednými vrcholmi so svojimi skalárnymi hodnotami (tie udávajú váhu). Produktom MC je typicky hladká izoplocha, avšak v extrémnych prípadoch (vysoké rozlíšenie diskrétneho priestoru) má tendenciu vygenerovať zbytočne veľké množstvo trojuholníkov.

Binárný objem priamo neposkytuje skalárne hodnoty, na základe ktorých by bolo možné generovať hladkú izoplochu, dôsledkom čoho sú uvedené ostré hrany v nerovinných oblastiach. Gibsonová formulovala iteratívny algoritmus, ktorého princípom je minimalizovať energiu v tzv. *povrchovej sieti* získanej zo segmentovaných binárnych dát a uhladiť tým pôvodnú izoplochu vrátane zachovania detailov, lebo manipulácia s uzlami povrchovej siete je obmedzená pôvodnou hranicou binárneho objemu [26]. Povrchová sieť musí byť naviac ešte triangulovaná – teda tento prístup je pomerne pomalý a komplikovanejší na paralelizáciu. Nasledujúci výklad podrobne popisuje nové riešenie.

4.1 Polygonizácia skalárneho poľa na GPU

Je predložená jednoduchá metóda založená na MC, ktorá spriemerovaním nastavených bitov voxelovej tehly v najnižšej úrovni SVO vypočíta faktor pokrytia použiteľný ako skalárnu hodnotu v intervale $(0, 1)$ prechádzanej mriežky. Keďže voxelová tehla obsahuje 4^3 buniek, dochádza k 64-násobnému podvzorkovaniu SVO, výhodou čoho je nižšia pamäťová náročnosť extrakcie izoplochy (menšia veľkosť dimenzie), avšak za cenu straty detailov výsledného povrchu, čomu sa však dá predísť voxelizáciou do jemnejšieho rozlíšenia. Základnou moti-

váciou podvzorkovania SVO je možnosť interpolácie medzi vrcholmi kociek, vytvárajúc tak elastickú izoplochu prispôsobenú meniacej sa topológii objemu.

Nepríjemnou vlastnosťou naívnej implementácie MC môže byť prechádzanie prázdneho priestoru, nad ktorým zbytočne prebieha vyhodnotenie prekrytia vrcholov kocky podľa izohodnoty. To by predstavovalo redundantné množstvo práce a mrhanie priestorom vo video pamäti GPU pri extrakcii izoplochy z SVO. Vzhľadom na to, že faktor pokrytia vnútorného priestoru úplnej binárnej voxelizácie má vždy skalárnu hodnotu 1 a prázdny priestor 0 (respektíve, nie je definovaný), je možné uskutočniť jej získavanie len na základe hraničnej oblasti, v ktorej sú definované uzly nultej úrovne NODES_0 (čitateľ by si mal spomenúť, že voxelizovaný povrch je pokrytý práve touto úrovňou).

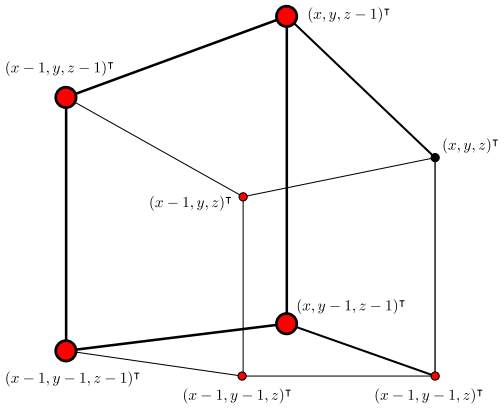
Nanešťastie však vystáva najavo, že nie je možné iba triviálne paralelizovať MC nad NODES_0 bez toho, aby do konečného povrchu neboli uvedené diery. MC by sa totiž nikdy nedostali k miestam, kde východzí vrchol kocky nenáleží do žiadneho uzlu nultej úrovne. Zistenie takýchto kociek prebieha následovne:

1. pre každý uzol v nultej úrovni SVO je paralelne spočítaný počet susediacich buniek v prázdnom alebo vnútornom priestore objemu (obrázok 4.1), produktom čoho je zoznam NullCounts,
2. nad NullCounts je vykonaný exkluzívny sken (nový zoznam Offsets),
3. paralelne znovu pre každý uzol v úrovni 0 sú vypočítané Mortonove kódy[1] susediacich buniek prázdneho priestoru a podľa Offsets sa zapíšu do ďalšieho zoznamu NullCubes, ktorého veľkosť je definovaná ako súčet posledných prvkov Offsets a NullCounts,
4. uskutoční sa paralelné radixové triedenie nad NullCubes,
5. každá položka i v NullCubes sa porovná s Mortonovým kódom položky $i+1$: ak sú rozdielne, zapíše sa do zoznamu Flag na odpovedajúce miesto 1, inak 0,
6. nad Flag sa uskutoční exkluzívny sken, ktorého výsledok je použitý pri kompakcii NullCubes určenej hodnotami Flag pre zápis na správnu pozíciu do transformovaného NullCubes, zbavujúc tým pôvodný zoznam duplicítnych kociek.

Nad množinou kociek pozicovaných Mortonovými kódmi NODES_0 sjednotenou s NullCubes konečne prebieha MC. Uvažovaný algoritmus stavia na vyhľadávacích tabuľkách (LUT), ako boli navrhnuté v práci Paula Bourka[2]. `edges` LUT obsahujúca 256 12-bitových položiek mapuje vrcholy pod izoplochou na hrany, ktorými izoplocha prechádza. Maximálne množstvo trojuholníkov, ktoré môže každá kocka vygenerovať je 5, preto ďalšia LUT `templates` poskytuje vyhľadanie šablóny formovaných polygonov z 256 15-tíc (každá trojica identifikuje odpovedajúce hrany, na ktorých sa nachádzajú interpolované vrcholy trojuholníka). V prvej časti paralelného MC dochádza k zisteniu veľkosti priestoru nutného k uloženiu hotovej izoplochy. Typicky sa pre tento účel používa vstup-centrický prístup exkluzívneho skenu. Ako bolo ukázané v prácach [19, 18], histogramová pyramída môže byť alternatívou pre efektívne určovanie pozície vo výstupnom prúde dát s možnosťou vyvolávať pracovné skupiny vlákien nad celkovým počtom výstupných dát. Zisťovanie pozície vo výstupe prebieha pri prechádzaní histogramovej pyramídy zhora nadol. Celkový postup možno zhrnúť do 3 následujúcich bodov:

[1] Mortonove kódy sú získané kombináciou ACTNODES_1 s indexom pracovnej položky.

[2] Bourke, Paul: Polygonising a Scalar Field, 1994: http://paulbourke.net/geometry/polygonise/.



Obr. 4.1: Susedstvo východzích vrcholov všetkých kociek vyhodnocujúcich uzol na pozícii $(x, y, z)^\intercal$.

1. najprv je paralelne klasifikovaných $|\text{NODES}_0| + |\text{NULLCUBES}|$ kociek na prípad prekrytia izoplochou, tzn. jednotlivé vrcholy kocky sú označené nastavením odpovedajúcich bitov (8-bitový index), následkom čoho je do TRICOUNTS zapísaný počet alokovaných trojuholníkov na odpovedajúce miesto,

2. vybudovanie histogramovej pyramídy z TRICOUNTS, jej vrchol obsahuje počet výstupných primitív M,

3. nakoniec paralelne pre M trojuholníkov je prejdená histogramová pyramída k odpovedajúcemu indexu TRICOUNTS, znovu sa určí prípad prekrytia izoplochou i, načo sa ďalej dopočítajú interpolované vrcholy na hranách definovaných indexovanou položkou $\texttt{edges}_i$ a z týchto vektorov sú zostrojené trojuholníky na základe šablóny $\texttt{templates}_i$.

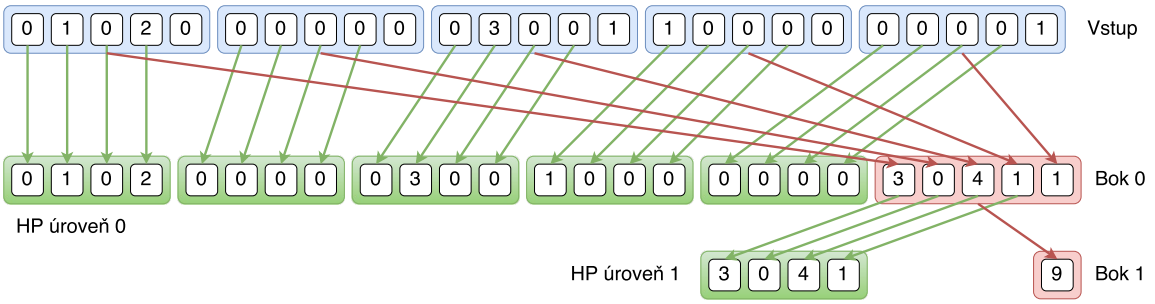
Proces vyššie sa odohráva najskôr nad množinami NODES_0 a následne NULLCUBES, avšak pri rozumnom navrhnutí nie je potrebné vytvárať špecializované kernely pre jednotlivé prípady.

Histogramové pyramídy

V predchádzajúcich odstavcoch sa vyskytol pojem histogramovej pyramídy (skrátene histopyramída). MC ako ukázkový príklad kompakcie a expanzie dátových prúdov vyžaduje z voxelového priestoru zavrhnúť kocky mimo izoplochu a zo zostávajúcich vytvoriť výstupný prúd trojuholníkov.

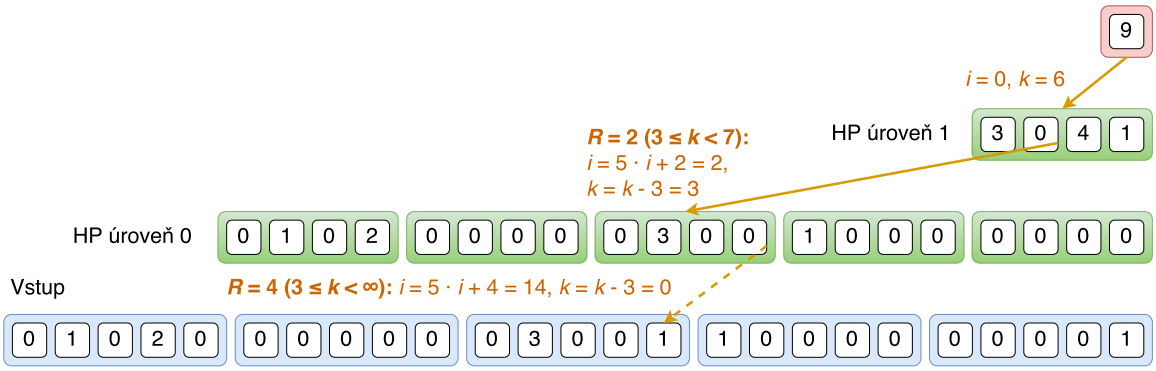
Pre tieto účely elegantne poslúži histopyramída, akceleračná dátová štruktúra parciálnych súm slúžiaca na efektívnu kompakciu i expanziu prúdov dát [18]. Predstavuje inteligentnú náhradu za skenovacie paralelné algoritmy. Emitovací proces je totiž spustený len nad celkovým počtom výstupných elementov a na základe svojho výstupného indexu si dohľadáva pôvodný vstupný index (element). Histopyramída sa svojou štruktúrou silno podobá k-nárnemu stromu, pretože jej budovanie prebieha postupnou redukciou k elementov nižších úrovní do nadradených uzlov. Evidentným nedostatkom oproti skenovacím algoritmom je nutnosť prechádzať pyramídu, čo je v prípade k-nárnej štruktúry histopyramídy logaritmická zložitosť pri základe k. Na druhú stranu umožňuje paralelizovať prácu iba nad výstupným prúdom.

Existuje niekoľko variant histogramových pyramíd odvíjajúcich sa od veľkosti redukcie do vyšších úrovní. Východzou z nich je 2:1 histopyramída, kedy každý uzol vyššej úrovne uchováva parciálnu sumu 2 asociovaných potomkov. V prípade plnej histopyramídy to znamená alokáciu $2 \cdot N - 1$ položiek, kde N je počet prvkov na vstupe, zarovnaný na mocninu dvoch. Operačná zložitosť získania vstupného indexu/elementu je potom $O(\log_2 N)$. Zníženie pamäťových nárokov a počtu krokov prechádzania histopyramídy sa dá dosiahnuť vyššou mierou redukcie. Modifikácia 4:1 (použitá pri vizualizácii izoplôch v práci [19]) naviac zmenšuje celkovú latenciu prechádzania stromovej štruktúry cachovaním hodnôt z mipmape podobnej štruktúry a hardvarom podporované získavanie hodnôt textúr je schopné nahrať všetky 4 hodnoty naraz. Ďalšiu možnosť redukcie predstavuje 5:1 histopyramída, aplikovaná v nadstavbe pôvodného algoritmu [18]. Táto verzia dopočítava posledný element pätice, vďaka čomu je schopná redukovať množstvo uchovávaných prvkov. Vybudovanie 5:1 histopyramídy znázorňuje obrázok 4.2.



Obr. 4.2: Budovanie 5:1 histopyramídy: v 1. iterácii sú zo vstupného zoznamu presunuté prvé štvorice pätíc do spodnej úrovne histopyramídy. Parciálne sumy týchto pätíc sa dočasne uložia do bočného poľa, nad ktorým sa celý proces opakuje, dokedy v novom bočnom poli nezostáva iba jediný prvok.

Prechádzanie podľa kľúča k (výstupný index) spočíva v určení intervalov aktuálnej štvorice rozhodujúcich o pohybe v štruktúre. Hodnoty (x, y, z, w) štvoríc definujú intervaly následovne: $\langle 0, x)$, $\langle x, x + y)$, $\langle x + y, x + y + z)$, $\langle x + y + z, x + y + z + w)$, $\langle x + y + z + w, \infty)$. Obrázok 4.3 ukazuje prechod podľa tejto logiky.



Obr. 4.3: Prechádzanie 5:1 histopyramídy: index vstupu i upravuje počas prechodu výraz $i = 5 \cdot i + R$, kde R určuje zvolený interval. Od k sa postupne odčíta najnižšia hodnota intervalu. Výsledné i nakoniec identifikuje index vo vstupe asociovaného s počiatočným k. Naviac, k na konci môže obsahovať zvyšok redukcie, potom skutočný index $i = i + k$.

Kapitola 5

Framework voCCeL

Sprievodným projektom tejto diplomovej práce je framework voCCeL (**vo**x**e**l + a**cc**e**l**erated) pre voxelizáciu do a extrakciu izoplochy z SVO na GPU. Realizácia vychádza z požiadavkov na rýchlosť, presnosť a optimálne využívanie zdrojov grafickej karty. voCCeL by mal slúžiť ako prípravok pre ďalšiu efektívnu a šetrnú prácu s binárnou objemovou reprezentáciou. Motiváciou tvorby tohto frameworku bolo pripravenie podkladov pre rýchlu VCSG konštrukciu zo vstupných 3D modelov a následnú rekonštrukciu povrchu ako trojuholníkovej siete. Je predstaviteľné, že dané riešenie by mohlo byť taktiež aplikovatelné aj v iných prípadoch použitia (podkapitola 2.3).

Táto kapitola popisuje návrh riešenia a aplikačné programovacie rozhranie (API). Konkrétne implementačné detaily sú rozvedené až v kapitole 6. Taktiež je prezentovaná statická dátová štruktúra SVO, ktorej návrh mal na mysli prácu s ňou na grafickej karte. Ďalej sa tu možno dočítať o zvolenom GPGPU API, riešeniach niektorých paralelných algoritmov ale aj optimalizáciu kódovania/dekódovania Mortonových kódov.

5.1 Návrh riešenia

Princíp povrchovej a úplnej voxelizácie je založený na technike popísanej v podkapitole 3.2, teda poskytuje paralelnú binárnu konverziu 3D modelov do SVO na GPU. Vstupom voxelizačnej techniky je vždy zoznam vrcholov trojuholníkovej siete (transformovanej do voxelového priestoru) a odpovedajúci zoznam indexov popisujúci jednotlivé steny povrchu modelu. Pre pohodlnosť poskytuje voCCeL vlastný Wavefront OBJ nahrávač geometrických útvarov, avšak pretože bolo cieľom frameworku minimalizovať počet závislostí na knižniciach tretích strán, predpokladá sa, že nahrávanie a transformáciu iných formátov bude riešiť samotný programátor používajúci toto aplikačné programovacie rozhranie. Z dôvodu návrhu spôsobu voxelizácie je minimálnou dimenziou diskrétnej mriežky hodnota 64. Výsledný SVO môže byť taktiež exportovaný do formátu XML z úložných dôvodov.

Extrakcia izoplochy prebiehajúca nad použitým SVO vychádza z postupu popísanom v podkapitole 4.1, teda vykonáva riedky MC algoritmus, tzn. sú prechádzané iba relevantné kocky vyhodnocujúce hraničnú oblasť okolo voxelov nultej úrovne SVO. Tento prístup znižuje pamäťovú náročnosť pre SVO voxelizovanom vo vyššom rozlíšení a vyhýba sa nepodstatným miestam objemu, ktorých spracovanie by nevyprodukovalo žiadnu časť izoplochy.

OpenCL

Bolo zvolené otvorené API OpenCL[1] pre paralelné spracovanie na komoditnom grafickom hardvéri vrátane CPU a vstavaných systémoch. Od verzie 1.1 poskytuje atomické operácie nad 32-bitovými celočíselnými hodnotami, nutná podmienka vypracovania testu náležitosti, ak sa objem komprimuje do poľa hodnôt tejto bitovej dĺžky.

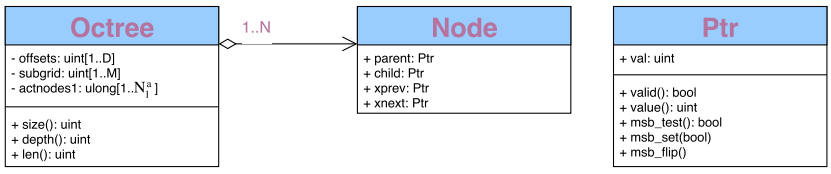
U zložitejších projektov môže byť štandardné API nepraktické z hľadiska prehľadnosti kódu a réžie pri kontrolovaní chybových stavov, preto bolo zapuzdrené do sady tried schopných synchronizovať vykonávané OpenCL operácie bez zásahu programátora, zjednodušiť prácu s vyrovnávacími pamäťmi GPU, jednoduchšie volať kernely alebo profilovať výkon aplikácie.

Riedky voxelový oktálový strom

Stavebným kameňom riedkej voxelizácie je veľakrát spomínaný SVO, udržiavajúci iba relevantné bloky priestoru pre definíciu nastaveného binárneho objemu. Riedka stromová štruktúra môže byť problematická pri transporte z hostiteľského programu do pamäťového priestoru grafickej karty, lebo typicky ukazovatele medzi uzlami dvoch úrovní prestanú obsahovať valídnu adresu. Evidentnou možnosťou je nahradiť pamäťové ukazovatele za indexy do danej úrovne a tým zachovať platnosť referencií. Je ideálne nastavovať indexy relatívne, teda indexovanie sa reštartuje pre danú úroveň – týmto spôsobom sa šetrí bitovým rozsahom interpretácie odkazov – a logicky je možnosť ukladať len inkrementované indexy, pretože sú vždy násobkom ôsmich.

Vzhľadom na to, že SVO musí byť parametrom voxelizačných kernelov, nie je jednoduché ani praktické poskytovať túto štruktúru vo forme globálnych vyrovnávacích pamätí pre jednotlivé úrovne. Kompilačný proces OpenCL programu sa tým totiž komplikuje. Ideálnym stavom je vytvorenie statického SVO ako jednotné 1D pole. Jednotlivé úrovne sú rozpoznateľné konštantnou pamäťou obsahujúcu ofsety do odpovedajúcej časti tohto poľa. Pri adresácii sa teda (podľa aktuálneho zanorenia) k indexu pripočíta daný ofset, aby sa zachovalo správne odkazovanie na uzly.

Datová reprezentácia uzlov založená na definícii na obrázku 3.5 je popísaná diagramom tried nižšie (obrázok 5.1), kde N je celkové množstvo uzlov v 1D poli a M počet 64-bitových voxelových tehál. D charakterizuje hĺbku stromu. V rámci úplnej voxelizácie dochádza v hrubších úrovniach k nastavovaniu bitov F a I kvôli propagácii prepínacej a stavovej informácie. Táto maska FI je rozdelená medzi ukazovatele `xprev` a `child` a komprimovaná do ich najvyššieho bitu b. Keďže po voxelizácii jestvuje eventualita, že x-ukazovatele budú zo štruktúry odstránené, bol práve `child` zvolený pre I, aby sa pri manipulácii zachovala táto informácia. Samotnú validitu ukazovateľov určuje bit $b - 1$.



Obr. 5.1: Diagram tried SVO a uzlu stromu.

[1] Špecifikácia OpenCL 1.1: https://www.khronos.org/registry/OpenCL/specs/opencl-1.1.pdf.

Mortonove kódy

Častá práca s Mortonovými kódmi predstavuje kritickú oblasť optimalizačného problému celkového návrhu. Zakódovanie prebieha podľa rovnice (2.2) a teda pre voxelizáciu v priestoroch s dimenziou väčšou ako 255 je nutné zvoliť 16-bitovú celočíselnú aritmetiku. Z toho vyplýva, že výsledok prekladania má $3 \cdot 16$ bitov. Nanešťastie to znamená, že dátovým typom Mortonových kódov musí byť 64-bitové číslo.

Existuje niekoľko spôsobom výpočtu indexov na z-krivke a veľmi efektívnym je použitie posúvacích LUT tabuliek[2], ktoré obsahujú predpočítané hodnoty pre jednotlivé bajty vstupných súradníc. Tie sú teda počas kódovania rozdelené na 8-bitové hodnoty, ktoré slúžia ako index do LUT a adresovaná hodnota sa už potom iba posunie na správne miesto. Na podobnom princípe funguje dekódovanie: znovu 3 LUT, ale tentoraz s 512 bajtovými hodnotami – každá súradnica sa dekóduje po 3 trojiciach (9 bitov, tzn. $2^9 = 512$). Hodnota z LUT sa nakoniec odsunie na odpovedajúce miesto medzi iteráciami, aby bolo konečné číslo súradnice bitovo správne usporiadané. Výsledky (namerané autorom knižnice `libmorton`) ukazujú, že použitie LUT poskytuje približne 15-násobné zrýchlenie. Vybratie sa cestou posúvacích LUT logicky znamená dedikovať vyhradené pamäťové miesto (typicky v konštantnej cache) pre predpočítané hodnoty. Pre kódovacie tabuľky to znamená asi 3 kB ($3 \cdot 256 \cdot 4$) a dekódovacie 1,5 kB ($3 \cdot 512$).

Povrchová voxelizácia

Framework poskytuje možnosť konzervatívnej aj 6-separabílnej povrchovej voxelizácie pre jemnejšie spracovanie vstupnej geometrie. Aby sa predišlo rozdeľovaniu jednotlivých rozhodovacích prípadov podľa dominantnej osi trojuholníka do viacerých kernelov (ako to urobili pôvodní autori navrhnutej voxelizácie [46]), je lepšie swizzliť komponenty vrcholov podľa rozhodujúcej osi a ujistiť sa, že počas atomickej operácie OR sa zapisuje na správne miesto v pamäti (*rozswizzlenie*).

Test náležitosti, najlepšie jeho optimalizovaná verzia (podkapitola 3.2), zodpovedá za správne a minimálne určovanie prekrytia voxelu geometriou. Všetky na voxelovej pozícii nezávislé hodnoty sú najprv predpočítané a testovacia fáza dopočítava ostatné termy. Zisťovanie hĺbkového intervalu taktiež predpočítava nezávislé hodnoty (obrázok 3.4).

V podkapitole 2.4 bol popísaný princíp prechádzania SVO podľa Mortonovho kódu pre nájdenie odpovedajúceho uzla. Pre navigáciu platí algoritmus 1, ale oplatí sa zapamätať si odkaz na uzol n z posledného prechodu a jeho Mortonov kód w. Pri ďalšom zápise do uzla SVO na pozícii m sa najprv zistí, ako vysoko leží jeho spoločný predok s s n (algoritmus 2). Ak je l dostatočne vysoké (teda s leží nízko – typicky polovica výšky SVO) je výhodnejšie vrátiť sa do s pomocou otcovských ukazovateľov a pokračovať z tohto miesta k hľadanému uzlu než začínať od koreňa stromu.

Voxelizácia 1. úrovne

Prvým krokom voxelizačného procesu je určenie všetkých aktívnych uzlov 1. úrovne. Ak je zvolené bežné 1D mapovanie

$$i(\vec{p}) = \vec{p}_z \cdot d^2 + \vec{p}_y \cdot d + \vec{p}_x,$$

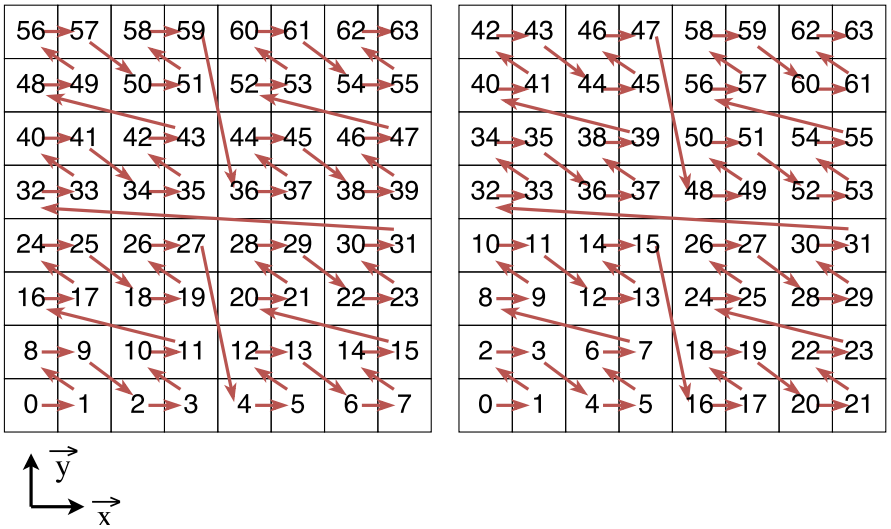
kde d je dimenzia voxelového priestoru, potreba potom spočítané Mortonove kódy zoznamu ACTNODES$_1$ zotriediť pred porovnaním na spoločného predka (viz spôsob mapovania $i(\vec{p})$

[2]Definícia v knižnici `libmorton`: https://github.com/Forceflow/libmorton.

Algoritmus 2 Zisťovanie úrovne spoločného predka: parameter l určuje počiatočné zanorenie.

```
1: function Z-ANCESTOR(m, w, l)
2:     while m ≠ w do
3:         m ← m » 3
4:         w ← w » 3
5:         l ← l − 1
       return l
```

na $z(\vec{p})$ na obrázku 5.2). Tento úkon sa dá vypustiť zapisovaním na miesta pamäte podľa z-krivky priestoru definovanom prvou úrovňou. Nech m je Mortonov kód, potom $w = m \,\&\, (111111)_2$ definuje index 4^3 tehly $(m \gg 6)$ a bitovú pozíciu $1 \ll w$ v nej. Pri zapisovaní do ACTNODES_1 je potom výsledný Mortonov kód definovaný ako $(\text{id} \ll 6) \mid i$, kde id je číslo pracovného objektu a i pozícia voxelu v tehle.



Obr. 5.2: Porovnanie prechádzania voxelového priestoru usporiadaného podľa $i(\vec{p})$ a z-krivky (2D analógia).

Úplná voxelizácia

Schopnosť vyplniť vnútro objektu blokmi rôznej veľkosti vyžaduje najprv XOR voxelizáciu povrchu do uzlov nultej úrovne SVO. V tejto fázi sú na počiatku predpočítané dve sady termov $d^{\text{yz},d}_{\vec{e}_i}$: $d^{\text{yz},4}_{\vec{e}_i}$ patrí voxelom dimenzie 4 testu jeho náprotivých vrcholov (výpočet známy z povrchovej voxelizácie), $d^{\text{yz},1}_{\vec{e}_i}$ evidentne pre plné rozlíšenie, avšak tentoraz už pre testovanie na stred voxela. K tomu sa predpočíta sada pravdivostných hodnôt t_i, ktoré identifikujú stav rasterizácie trojuholníka v rovine podľa horne-ľavého vyplňovacieho pravidla bežného v grafických čipoch. Termy

$$t_i = (\vec{n}^{\text{yz}}_{\vec{e}_i,x} > 0) \vee (\vec{n}^{\text{yz}}_{\vec{e}_i,x} = 0 \wedge \vec{n}^{\text{yz}}_{\vec{e}_i,y} < 0)$$

podľa hrán trojuholníka umožňujú preventívne vynechať mylné opakovanie procesu prepínania stavov voxelových tehál pre susedné trojuholníky. Toto určenie prebieha nasledovane:

$$(\langle \vec{n}^{\text{yz}}_{\vec{e}_i}, \vec{p}_{yz} \rangle + d^{\text{yz},1}_{\vec{e}_i} = 0) \wedge \neg t_i$$

Podobne ako u povrchovej voxelizácie je rozumné udržiavať v registre informáciu o poslednom dosiahnutom uzle a jeho Mortonovom kóde pre kratšie prechádzanie SVO. Po premietnutí na rovinu trojuholníka je určený interval $\bar{q} \bmod 4, \ldots, 3$, na ktorom prebieha prepínanie voxelov vo voxelovej 4^3 tehle. Navigáciu v nej urýchľuje predpočítaná LUT. Dočasný register obsahujúci prepínaciu 4^3 tehlu, ktorá sa na konci zisťovania, ktoré voxely prepnúť (bitová OR operácia), použije pri atomickom XOR globálnej pamäte.

Propagácia do vyšších úrovní SVO potrebuje niekoľko špecializovaných kernelov prispôsobených pre hraničné úrovne 0 (obsahuje 64 bitové hodnoty dvoch najjemnejších úrovní SVO) a 1 (maska FI). Všeobecné kernely pre úrovne $i > 1$ sú potom využívané v iteračnom štýle. Konkrétne zdrojové kódy týchto kernelov sú k dispozícii vo výpisoch v prílohe A.

Extrakcia izoplochy

voCCeL ponúka modifikáciu MC nad binárnym objemom produkujúcu hladkú izoplochu a charakterizovanú riedkosťou prechádzania voxelového priestoru pod vedením štruktúry SVO. Stanovenie všetkých kociek vyhodnocujúcich aspoň 1 uzol úrovne 0 (podkapitola 4.1) sa podobá klasickému paralelnému algoritmu zunikátnenia zoznamu hodnôt. Len čo došlo na zistenie všetkých relevantných východzích vrcholov, je nad týmto zoznamom kociek spustený klasický MC algoritmus. LUT pre určovanie prípadov prekrytia voxelu izoplochou boli prevzaté z práce Paula Bourka. Konvencia usporiadania vrcholov uzlov je totožná s 2^3 z-usporiadanou voxelovou tehlou. Určenie Mortonových kódov nultých uzlov možno na základe skorej vypočítaného zoznamu aktívnych uzlov 1. úrovne $m = (\text{ActNodes}_1[\frac{\text{id}}{8}] \ll 3) \mid (\text{id} \bmod 8)$. Zistenie 3D pozície a ofsetovanie na ostatné vrcholy kocky sa odvíja od dekódovaného m. Nutnosťou MC metódy je taktiež nezabudnúť ošetriť prípady, kedy kocky prechádzajú priestor na hraniciach diskrétnej mriežky. V týchto prípadoch dochádza k vzorkovaniu mimo definovaný priestor, čo je možné ošetriť v rámci kódu priamo alebo pred voxelizáciou zmenou mierky vstupnej geometrie klasickým vzťahom

$$\frac{(\vec{v}_\Diamond - \min_\Diamond) \cdot (d-4)}{\max_\Diamond - \min_\Diamond} + 4, \tag{5.1}$$

kde $\vec{v}$ je vrchol trojuholníka, d dimenzia Γ, $\Diamond = \{x, y, z\}$ a min/max sú vektory minimálnych/maximálnych hodnôt komponent prevádzanej trojuholníkovej siete. Posun o 4 vychádza z rozlíšenia voxelovej tehly.

Všeobecné paralelné algoritmy

Minimalizovanie počtu závislostí frameworku na iných knižniciach tretích strán znamenalo implementovať viaceré všeobecné paralelné algoritmy, bez ktorých by sa voxelizácia a izo-extrakcia nedokázala obísť. Ich implementácia ctí podmienku efektívnosti riešenia, aby nedochádzalo k ovlivneniu rýchlosti voxelizačného alebo MC procesu.

Exkluzívny sken

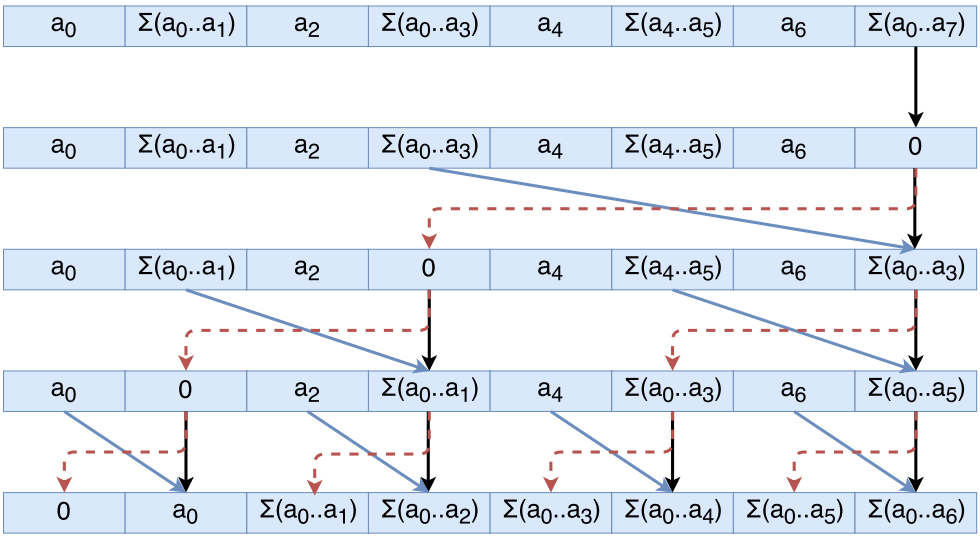
Prefixová suma (skrátene sken) slúži ako stavebný kameň mnohých paralelných algoritmov (kompakcia). Rozpoznávané sú dva typy skenovania: inkluzívny a exkluzívny sken. Implementácia tohto projektu vyžadovala použitie len exkluzívnej varianty. Tá je definovaná ako

$$[I, a_0, a_0 \odot a_1, \ldots, a_0 \odot a_1 \odot \cdots \odot a_{N-2}],$$

kde I predstavuje identitu operácie $\odot$ a a_i prvok vstupného poľa o veľkosti N. Asociativita a komutativita operátoru $\odot$ sú veľmi dôležitými predpokladmi pri vytváraní riešenia s rovnovážnym zaťažením zdrojov – (virtuálnych) procesorov. Naívne implementácie prefixovej sumy predpokladajú, že množstvo prvkov skenovaného poľa je blízky počtu výpočtových jednotiek. To typicky nenastáva ani pri použití grafických multiprocesorov, predovšetkým keď je vstupné pole veľké. Všeobecný prístup k tomuto problému je rozdeliť vstupné pole na niekoľko *porcií*, každá z nich je spracovávaná nezávisle skupinou vyhradených procesorov. Tie pracujú nad porciou, ktorej počet prvkov je aspoň dvojnásobkom veľkosti tejto skupiny, inak by mohli nastať zápisové konflikty v závislosti na skutočnosti, že sken je vykonávaný in-situ a počet vlákien bežiacich súčasne býva menší než rozsah warpu/wavefrontu. Z toho ďalej vyplýva, že v skupinách musí byť zamestnaná lokálna pamäť, ktorá je synchronizovaná medzi dvomi fázami algoritmu. Tie sú v preklade nazývané horné/dolné *zametanie* [27]. Myšlienkou zametania je vybudovanie vyváženého binárneho stromu (horné zametanie) pre výpočet prefixovej sumy. Z jeho koreňa (dolné zametanie) dochádza k propagácii vypočítaných parciálnych súm sčítaných (ak $\odot = +$) s obsahom ľavého potomka do pravého synovského uzla. Celý proces má časovú zložitosť $O(N)$, kde N znamená počet prvkov vstupného poľa. Redukčný krok popisuje algoritmus 3 a prechádzanie binárneho stromu smerom dolu znázorňuje obrázok 5.3.

Algoritmus 3 Algoritmus horného zametania.

1: **function** UP_SWEEP(a, N)
2: **for** $d \in \{0, \ldots, \log_2 N - 1\}$ **do**
3: **for** $k \in \{0, \ldots, N-1\}$ **by** 2^{d+1} **do in parallel**
4: $a_{k+2^{d+1}-1} \leftarrow a_{k+2^d-1} + a_{k+2^d+1-1}$
5: **end for**



Obr. 5.3: Princíp dolného zametania skenovacieho algoritmu.

Pre veľké polia je pred dosadením I uložený koreň do globálnej pamäti parciálnych súm jednotlivých porcií. Neskôr sa nad ňou vykoná sken a jednotlivé elementy sa pripočítajú prvkom týchto lokálne zoskenovaných porcií. Ukončenie rekurzívneho zanorovania nastáva, keď sa veľkosť poľa parciálnych súm rovná jednej.

Radixové triedenie

Extrakcia izoplochy sa nanešťastie nedokáže vyhnúť použitiu triedenia zoznamu Mortonových kódov (64-bitové čísla), preto bolo nutné paralelizovať túto časť výpočtu pre zunikátnenie NullCubes. Podstata tohto algoritmu, narozdiel od mnohých iných porovnávajúcich hodnoty triedeného zoznamu, spočíva v počítaní histogramu kľúčov, dávajúc pozíciu preusporiadania každej vstupnej hodnoty. Reprezentáciu kľúčov definuje radix $R = 2^r$, r je počet bitov nutných pre reprezentáciu R. Nech b určuje počet bitov jednotlivých hodnôt triedenej postupnosti, potom $p = \frac{b}{r}$ znamená množstvo iterácií algoritmu a teda

$$k = k_{p-1}k_{p-2}\dots k_0$$

sú číslice kľúča k, podľa ktorých prebieha triedenie. Implementácia pomocou OpenCL rozdelí vstupnú postupnosť na porcie, spracovávané jednou skupinou o veľkosti I. Týchto skupín je spolu G pre N prvkov, ktoré by malo byť deliteľné $G \cdot I$. Každá skupina $g \in \{0, \dots, G-1\}$ sa stará o porciu elementov s indexmi

$$j \in \left\{g \cdot \tfrac{N}{G}, \dots, (g+1) \cdot \tfrac{N}{G} - 1\right\}$$

a v každej skupine položka $i \in \{0, \dots, I-1\}$ spracováva prvky

$$j \in \left\{g \cdot \tfrac{N}{G} + i \cdot \tfrac{N}{G \cdot I}, \dots, g \cdot \tfrac{N}{G} + (i+1) \cdot \tfrac{N}{G \cdot I} - 1\right\}.$$

Vlákna počítajú odpovedajúce položky $H(c, g, i)$ – kvantita $H(c, g, i)$ určuje počet kľúčov rovných číslici c, tzn. $k_q = c$ a $q \in \{0, \dots, p-1\}$. V každej iterácii sa nad aktuálnym $H(c, g, i)_q$ spočíta exkluzívny sken, zisťujúc tým ofsety preusporiadania prvej z hodnôt $k_q = c$. Radixové triedenie možno považovať za stabílne, ak je dodržané pravidlo prechádzania k od najvyššieho významového bitu k najnižšiemu či obrátene.

Problematikou paralelného riešenia je zrýchlenie prístupu k histogramu pomocou lokálnej pamäte. Totiž, podľa uvažovaného radixu r vyžaduje každá skupina alokovanie až $S = 2^r \cdot I$ položiek s bitovým rozsahom b. Od použitého radixu sa odvíja aj počet iterácií algoritmu a pre Mortonove kódy (8 bajtové premenné) by pri $r = 1$ bolo $p = 64$. Pravdepodobne sa takto bude diať nad veľkým zoznamom hodnôt a je preto vhodné znížiť iteračné množstvo, avšak s ohľadom na dostupnú vyrovnávaciu pamäť multiprocesorov.

5.2 Aplikačné programovacie rozhranie

voCCeL predstavuje minimalistický framework pre účely voxelizácie do SVO a izo-extrakcie z tejto dátovej štruktúry. Väčšina detailov bola spomenutá v predchádzajúcich podkapitolách, pre úplnosť sú tu ešte raz vypísané. Čitateľ sa tu môže dočítať aj o XML definícii vymyslenej pre jednoduchý transport SVO medzi aplikáciami.

Aby bolo možné spúšťať výpočty tried frameworku pod OpenCL, je nutné explicítne vytvoriť `vc::cl::parallel`, nesúcu informáciu o prostredí, zariadeniach, kontexte a fronte príkazov tohto GPGPU API. Ďalej spolu štyri najdôležitejšie triedy predstavujú jadro celého rozhrania. Prvou z nich je `vc::tri_mesh`, ktorá musí byť vstupom voxelizačnej funkcie. Ide o štruktúru obsahujúcu zoznamy `vertices` a `indices` popisujúce geometriu prevádzanej trojuholníkovej siete. Jej vlastnosťou je, že pre úspešnú voxelizáciu musí byť vopred transformovaná do rozmerov voxelového priestoru podľa vzťahu (5.1).

Voxelizovanie má na starosti inštancia triedy `vc::voxelizer`. V tabuľke 5.1 sú popísané parametry jej verejnej metódy `voxelize`, kde `mode` type `vc::type` definuje typ voxelizácie a môže nabývať hodnôt

- `type::SURFACE26` – povrchová voxelizácia superkrytu (26-separabílna),
- `type::SURFACE6` – 6-separabílna povrchová voxelizácia,
- `type::SOLID` – úplná voxelizácia.

Extrakcia izoplochy na vstupu požaduje SVO s dátovým typom `vc::octree`, je teda výhodné snažiť sa ju používať i pri ďalšom spracovaní, ak má byť výsledkom práce polygonová sieť. Taktiež by mala byť dodržaná náležitosť izohodnoty `isovalue` do intervalu $(0, 1)$. Technika pracuje nad binárnou reprezentáciou, preto žiadne iné hodnoty nie sú povolené. Pre úspech izo-extrakcie zároveň platí, že SVO musí mať nastavené bity I identifikujúce vnútro modelu vo vyšších úrovniach, povrchová voxelizácia teda neposkytuje dostatočnú informáciu pre vyjadrenie izoplochy.

Menný priestor	`vc`		
Trieda	`voxelizer`		
Metóda	`voxelize`		
Parametry	`model` :	`vc::com::tri_mesh`	prevádzaná polygonová sieť,
	`dim` :	`ushort`	veľkosť dimenzie Γ,
	`mode` :	`vc::type`	druh voxelizácie.
Výstup	`scene` :	`vc::octree`	SVO.

Menný priestor	`vc`		
Trieda	`isoextractor`		
Metóda	`extract_isosurface`		
Parametry	`scene` :	`vc::octree`	SVO,
	`dim` :	`ushort`	veľkosť dimenzie Γ,
	`isovalue` :	`float`	izohodnota.
Výstup	`vertices` :	`std::vector<float>`	zoznam trojuholníkov.

Tabuľka 5.1: API voxelizačnej a izo-extrakčnej triedy.

SVO definovaný triedou `vc::octree` vychádza z diagramu na obrázku 5.1. Existuje možnosť exportovať dátové časti SVO pomocou verejnej metódy `export_xml`. DTD pre toto XML vo výpise 5.1. Ďalšie metódy popisuje tabuľka 5.2. Tie slúžia hlavne pre získanie dátových polí obsahujúcich uzly, ofsety a voxelové tehly SVO. Kontajnerom týchto polí je špecifická šablóna `vc::cl::vector<T>`, majúca referenciu na globálnu pamäť, v ktorej sa vyskytujú dané dáta. Ak si uživateľ želá previesť túto štruktúru na bežný STL nosič `std::vector<T>`, dá sa tak vykonať cez metódu `read`. Viac informácií o nej možno nájsť v dokumentácii, prípadne implementačných detailoch (ďalšia kapitola).

```
<!DOCTYPE octree (offsets, levels, subgrid)>
<!ELEMENT offset (#PCDATA)>
<!ELEMENT levels (level+)>
<!ELEMENT level (node+)>
<!ELEMENT node EMPTY>
<!ELEMENT subgrid (bitmask+)>
<!ELEMENT bitmask (#PCDATA)>
<!ATTLIST octree dim CDATA #REQUIRED>
<!ATTLIST octree size CDATA #IMPLIED>
<!ATTLIST octree depth CDATA #IMPLIED>
<!ATTLIST node parent CDATA #REQUIRED>
<!ATTLIST node child CDATA #REQUIRED>
<!ATTLIST subgrid size CDATA #REQUIRED>
```

Výpis 5.1: DTD pre XML exportovaného SVO.

Metóda	`vc::octree`			
`export_xml`	**Parametry**	`scene`	: `vc::cl::parallel`	OpenCL prostredie,
		`dim`	: `ushort`	veľkosť dimenzie Γ,
		`path`	: `std::string`	názov exportovaného súboru.
	Výstup	-		
`size`	**Parametry**	-		
	Výstup	`n`	: `uint`	veľkosť 1D poľa obsahu SVO.
`depth`	**Parametry**	-		
	Výstup	`d`	: `uchar`	výška stromu po nultú úroveň.
`len`	**Parametry**	-		
	Výstup	`l`	: `uint`	počet 32-bitových voxelových tehál.
`get_root`	**Parametry**	-		
	Výstup	`root`	: `cl::vector<node>`	uzly SVO.
`get_offsets`	**Parametry**	-		
	Výstup	`offsets`	: `cl::vector<uint>`	ofsety úrovní SVO.
`get_subgrid`	**Parametry**	-		
	Výstup	`subgrid`	: `cl::vector<bitmask32>`	voxelové tehly.

Tabuľka 5.2: API triedy definujúcej SVO.

Kapitola 6

Implementačné detaily

Táto kapitola slúži ako doplňková kapitola komentujúca konkrétne výpisy procedúr (viď príloha A) a detailnejší rozbor niektorých častí frameworku voCCeL. Sú zaujímavé hlavne pre vývojárov, ktorí by ďalej chceli rozšíriť/zmeniť správanie alebo funkčnosť tohto frameworku. Ten bol napísaný v C++ (štandard 17), využívajúc paradigma objektovo-orientovaných jazykov a návrhové vzory, uľahčujúc prácu pri programovaní komplexných projektov.

Štruktúra súborového systému projektu je rozdelená do 3 hlavných priečinkov

- `include` – obsahuje definície tried frameworku,
- `detail` – tu sa nachádzajú `inline` definície krátkych metód,
- `src` – zložitejšie metódy sú definované v tejto zložke,

kde sa možno dohladať zdrojových kódov knižnice voCCeL. Oddelenie `inline` definícií od kódu tried v hlavičkových súboroch zachováva čistotu celého API a skrýva implementáciu.

6.1 Zapuzdrenie OpenCL

Pre prácu s OpenCL bola naprogramovaná vlastná abstrakcia pod menným priestorom `vc::cl`. Vlastnosťou tohto súboru tried obaľujúcich funkčnosť GPGPU rozhrania je definícia OpenCL objektov pomocou CRTP[1], vymieňajúc tak dynamické používanie virtuálných tabuliek metód za statický polymorfizmus. Taktiež poskytuje možnosť nastavovania heterogénnych parametrov kernelov pomocou jedinej šablónovej funkcie s nešpecifikovaným počtom parametrov (*parameter pack*). Ziskom z tohto úsilia je oveľa prehľadnejší kód, zapuzdrenie kontroly chýb pomocou výnimiek a jednoduchšia správa GPU príkazov pomocou obaľujúcej triedy `parallel`. Tá sa taktiež stará o akumulovanie času stráveného vykonávaním kernelov, zjednodušujúc tým profilovanie. Rovnako sa programátor nemusí obťažovať so synchronizovaním vykonávanej práce kernelov, pretože si táto trieda pamätá udalosti, na ktoré treba čakať. Dá sa teda uvažovať nad lineárnym modelom pre synchronizáciu.

Dôležitým prvkom puzdra je spôsob práce s vyrovnávacími pamäťmi OpenCL. `vector` poskytuje priamočiary štýl manipulácie dát na výpočtovom zariadení. Jedná sa o šablónu, schopnú prispôsobiť výpočet čítania/zapisovania dát podľa zvoleného dátového typu. Čítanie z vyrovnávacej pamäte rieši metóda `read`, ktorej parametrami sú počiatok a veľkosť čítanej pamäte. Zapisovanie funguje podobným spôsobom (`write`).

[1] *Curiously Recurring Template Pattern*, pozri:
https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

6.2 Voxelizácia

Gro najvýznačnejšej časti práce vysvetľujú následujúce odstavce. Najprv je diskutovaná hostiteľská trieda `vc::voxelizer`, ktorú možno považovať za najvýznamnejší element frameworku voCCeL. Následne sú vysvetlené konkrétne riešenia kernelov určených pre voxelizačný proces.

Hostiteľský kód

`vc::voxelizer` implementuje voxelizáciu vstupnej 3D geometrie. Jediná verejná metóda `voxelize` spúšťa podčasti voxelizačného procesu. Jedná sa o tieto funkcie

- `lvl1_voxelize` – spočíta Mortonove kódy voxelizovanej prvej úrovne,
- `build_octree` – iteratívne vytvorí SVO na základe aktívnych uzlov jednotlivých úrovní,
- `subgrid_voxelize` – podľa druhu voxelizácie (premenná `mode`) spustí odpovedajúci voxelizačný kernel najjemnejšej úrovne (ide buď o povrchovú alebo úplnú voxelizáciu),
- `propagate_bitflips` – vykonáva propagovanie bitovej masky FI a prepínanie voxelových tehál

a ich definíciu možno nájsť v súbore `src/voxelizer.cc`. Vo výpise A.1 slúži volanie kernelu s názvom `nullify` na vynulovanie hodnôt alokovanej pamäte objemu `volume`. Zisťovanie dimenzie pre $\delta = 8$ prebieha na riadkoch 6-8, samozrejme je ešte celkový počet 32-bitových tehál vydelený bitovým rozsahom. `count_volume` vykonáva počítanie nastavených bitov. Metóda `prescan` sa postará o exkluzívny sken nad bitovým počtom. Keďže exkluzívny sken nezahŕňa poslednú položku vstupného zoznamu, je nutné pred transformáciou `tmp` vytiahnuť do dočasnej premennej túto hodnotu a následne ju pripočítať k poslednému prvku po skenovaní (celkový počet N_1^a). Výpočet Mortonových kódov aktívnych uzlov potom prebieha spustením `write_mortons` kernelu.

Po zistení ActNodes_1 môže byť vybudovaný riedky oktálový strom `vc::octree`. Tento zoznam sa najprv skopíruje do pomocného poľa (kvôli extrakcii izoplochy) a inicializuje sa celkový počet $8{\cdot}N_1^a$ uzlov 1. úrovne. Pretože budovanie prebieha po úrovniach a vopred nie je známa štruktúra stromu, je nutné dočasne ukladať inicializované úrovne do STL kontajnera `lvls`, ktoré sú na konci po úrovniach nakopírované do `root`. Cyklus spracovania jedného zoznamu `actnodesj` najprv určí zoznam Flag_j spustením kernelu `tag_parents`. Šetrenie miestom znamená, že bajtové hodnoty Flag_j sú zmutované na 32-bitové do ParentIndex_j kernelom `mutate`, inak by transformácia exkluzívnym skenom spôsobila pretečenie parciálnych súm. Hneď po zistení N_i^a sa do ďalšieho pomocného STL kontajnera `offsets` vloží nová nulová hodnota a požadované odsadenie prebieha za účasti `std::for_each`. Inicializácia ukazovateľov medzi uzlami spočíva vo vyvolaní kernelu `init_nodesi` a `link_lvli`. Posledný zmienený sa stará o spárovanie uzlov medzi úrovňami. Nakoniec `strip_mortons` zistí následujúcu sadu Mortonových kódov aktívnych uzlov v úrovni i. Až na posledný krok, podobná procedúra sa vykonáva i pre úroveň $N-3$, avšak nie je potrebné zisťovať N_{N-3}^a, lebo všetky uzly vyššie musia byť vždy alokované. Keďže ale musí dôjsť na spárovanie uzlov s nižšou úrovňou, je potrebné dopočítať ParentIndex_{N-3} a odpovedajúce Mortonove kódy. Potom počas kopírovania úrovní do vytvoreného `vc::octree`, spustí sa pre každú úroveň okrem nultej proces zošívania (`stitch_lvli`) x-ukazovateľov, tzn. ak uzol v úrovni i odkazuje svojím x-ukazovateľom na susedný uzol, musia sa ich potomkovia rovnako referencovať.

Samotné voxelizovanie do mriežky voxelových tehál pozostáva iba z volania dedikovaných kernelov, preto sa preskočí na záverečnú časť úplnej voxelizácie `propagate_bitflips` (výpis A.3). Využíva sa tzv. OpenCL *sub-bufferov* kontajnera `root` pre priamočiare spracovanie. Špecifikácia OpenCL vyžaduje, že počiatočná adresa sub-bufferu je vždy zarovnaná na násobok hodnoty `CL_DEVICE_MEM_BASE_ADDR_ALIGN` použitého zariadenia. Behom adresovania to teda znamená pripočítať dodatočný ofset v kerneloch (vo výpisoch nazvaný `align0`/`align1`/`aligni`). Puzdro `vc::cl` priamo zabezpečuje výpočet tejto hodnoty. Prvým krokom propagácie je konkatenácia voxelových stĺpov tehál 4×4×2. Tomu sa tak deje v kerneli `concatenate_subgrid`. Potom následne špecializovaná `flip_lvl1` nastaví prepínaciu informáciu v 1. úrovni SVO. Ďalej sa pokračuje cyklom analogicky k predchádzajúcim dvom krokom, s tým že sa použijú všeobecne navrhnuté kernely `concatenate_lvli` a `flip_lvli`. Propagácia smerom dolu prebieha podobne (`propagate_lvli`) a špecializácia existuje pre 1. úroveň do nultej, konkrétne `propagate_subgrid`.

Kernely

Navrhnutý algoritmus (podkapitola 3.2) beží volaním paralelizovaného kódu súboru kernelov na relatívnej systémovej ceste `resources/kernels/voxelization.cl`. V tomto súbore sú zahrnuté dodatočné definície pre prácu s geometriou, Mortonovmi kódmi a SVO. Všetky 4^3 tehly zmienené v predchádzajúcej kapitole sú rozdelené na dva 32-bitové bloky a každá operácia nad ňou prispôbená tomuto rozdeleniu. Povrchová voxelizácia voxelových tehál je v tejto podkapitole z dôvodu šetrenia miestom vynechaná – táto procedúra je okrem prechádzania SVO prakticky totožná s voxelizáciou 1. úrovne. V následovne odkazovaných výpisoch si možno povšimnúť snahu minimalizovať počet vyvolaní atomických bitových operácií, deje sa tak prostredníctvom pomocných registrov (32-bitovej šírky). Konečný stav tehly sa do globálnej pamäti dostane až po zmene adresovania.

Voxelizácia 1. úrovne

Podstatnou funkciou procesu 3D skenovacej konverzie je `lvl1_voxelize`, kde sa začína 1. krok voxelizácie, t.j. prevod vstupnej geometrie na voxely prvej úrovne ($\delta = 8$). Podľa rovnice (3.6) dochádza k posunutiu kritických bodov, čo možno vidieť vo výpise A.4 na použití premennej `c_shift`. Možno si povšimnúť veľa predpočítaných konštantných premenných fázi prípravy. Získavanie voxelového rozsahu sa deje jednoduchým uzavretím hodnôt osovo-orientovaného hraničného kvádra (štruktúra `aabb`) do medzí dimenzie Γ. Prvé vyhodnotenie hranových funkcií prebieha v rovine podľa dominantnej osi, avšak vďaka swizzleniu sa vždy pracuje s komponentami x, y. Najvnorenejší cyklus prechádza hĺbkový interval, zistený sledovaním paprsku z kritických bodov v rovine xy. Pri zapisovaní nastavených voxelov sa berie v úvahu usporiadanie objemového priesotru podľa z-krivky a zároveň sa pečlivo identifikuje odpovedajúca 4×4×2 tehla ku správnemu adresovaniu 32-bitových hodnôt.

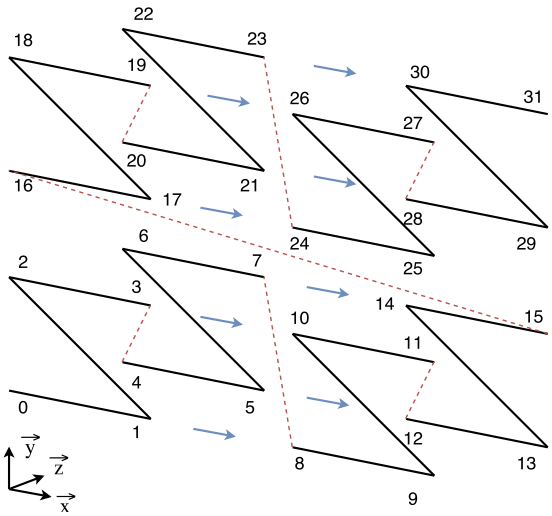
Podobne výpis A.5 ukazuje, akým spôsobom sú dopočítané Mortonove kódy v zozname $\textsc{ActNodes}_1$ po prvej voxelizácii. Ako bolo spomenuté v podkapitole 5.1, po zápise do diskrétnej mriežky objemu 1. úrovne usporiadanej podľa z-krivky a správnej rekonštrukcii aktívnych Mortonových kódov, netreba ďalej triediť pole výsledných hodnôt, keďže ich výpočet bol uskutočnený nad konzistentnými súradnicami. Podstatným poznatkom pre pochopenie tohto procesu je nezabudnúť, že bloky veľkosti 4^3 sú rozdelené do dvoch 32-bitových celočíselných premenných.

Úplná voxelizácia

Proces vyplnenia najjemnejších voxelov stavia na detekcii dlaždíc nultej úrovne (`DELTA4` definuje $\delta = 4$) v rovine yz a proces prepínania prebieha v smere osi $\vec{x}$. Znamená to teda predpočítanie hodnôt $d^{\mathrm{xy},4}_{\vec{e}_i}$, $d^{\mathrm{xy},1}_{\vec{e}_i}$ v prípravnej fázi algoritmu, viditeľné vo výpise A.6. Ďalej `tl` obsahuje termy pravdivostných hodnôt stavu rasterizačného pravidla pre zabránenie opakovaného spustenia propagácie bitových prepínaní pre ľavé a horné hrany τ. Podľa osovozarovnaného hraničného kvádra je určený rozsah $\delta = 4$ dlaždíc, ktorý testuje $d^{\mathrm{yz},4}_{\vec{e}_i}$. Detekované prekrytie v úrovni 0 následne spustí prechod po dlaždici. Ak dojde k prekrytiu i v najjemnejšej úrovni, dochádza k projekcii na rovinu trojuholníka, čím je vypočítaná x-ová súradnica $\bar{q}$, identifikujúca voxelovú tehlu a pozíciu v nej, odkiaľ sa začína prepínanie. Použitie LUT pre navigáciu vo voxelovej tehle sprostredkúva `ztrav_4x4x2`. Toto pole je indexované aktuálnou pozíciou počas postupovania pozdĺž x-ovej osi v lokálnom systéme z-usporiadanej 32-bitovej reprezentácie rozdelenej 4^3 tehly. K atomickému XOR dochádza až po vytvorení prepínacej masky `f`. V tej sú bitovou operáciou OR nastavené všetky bitové miesta, kde musí dojsť k prepnuti stavovej informácie.

Propagácia medzi úrovňami

Komplikovaná posledná časť úplnej voxelizácie spočíva v nastavení bitových prepnutí a výplne vnútra objektu pozdĺž osi $\vec{x}$. Obrázok 6.1 uľahčuje predstavenie si spôsobu, akým dochádza k propagácii v 32-bitových objemových hodnotách. Zo z-usporiadania potom vychádza aj princíp prepínania bitových stavov vo voxelovom stĺpe (výpis A.8) počas konkatenácie uzlov nultej úrovne v smere x-ovej osi.



Obr. 6.1: Usporiadanie voxelovej tehly 4×4×2 a propagácia v smere osi $\vec{x}$.

Na rozhraní nultej a prvej úrovne dochádza ku špecializovanému nastaveniu prepínacích stavov otcovských uzlov. Nastavenie všetkých relevantných bitov (s lokálnym x indexom 3) je možné otestovať podľa hexadecimálnej hodnoty `0xAA00AA00`. Bit F uzlu 1. úrovne je definovaný v MSB ukazovateľa `xprev`. Konkatenácia i-tej úrovne prebieha analogicky k verzii s voxelovými tehlami avšak tentokrát sa prepína nielen prepínací bit F, ale aj I, uložený v ukazovateľovi `child`. Krátky všeobecný kernel popisujúci tento proces zobrazuje výpis A.10. Činnosť propagácie stavov prepnutia zdola nahor pokračuje kódov z výpisu A.11.

Znovu sa jedná o všeobecný kernel spracovávajúci vyššie úrovne ($i > 1$). Zasa prebieha prepínanie do rodiča na základe prepínacej informácie F uzlov s lokálnou x-súradnicou 1 (je vidieť analógia s predchádzajúcim kernelom vo výpise A.8).

Posledné výpisy A.12 a A.7 propagačných kernelov implementujú konečnú fázu úplnej voxelizácie. Tá prebieha zhora dolu a podobne ako v prípade minulých operácií má špecializovaný kernel pre hranicu rozdielnych reprezentácií uzlov medzi 1. a nultou úrovňou. Aby bol výsledný objem správne voxelizovaný, netreba zabudnúť na vymazanie bitu I v uzle, z ktorého dochádza k propagácii do nižšej úrovne (jeho potomkov). Inak by zostali nastavené výplne oblastí, ktoré nenáležia vnútru objemu. Tento proces propagovania sa evidentne uskutočňuje (ako vyplýva z použití všeobecných verzií konkatenácie, prepínania a dolnej propagácie) v cykle pre všetky úrovne i okrem $i = N - 1$. Najhrubšia úroveň SVO totiž nikdy nemôže byť vyplnená, dokonca ani $i = N - 2$, avšak môže dochádzať k dočasnému nastaveni FI a teda propagácii z týchto vrchných uzlov do nižších úrovní SVO.

6.3 Extrakcia izoplochy

MC algoritmus implementovaný pre SVO definuje nový postup získania hladkej izoplochy na GPU, prispôsobený veľkosti tejto štruktúry. Krok číslo 1 predstavuje zistenie a zunikátnenie zoznamu NullCubes. Toto je proces známy z všeobecných paralelných algoritmov, preto bude vysvetlený len odpovedajúci hostiteľský kód hlavnej triedy. Následná extrakcia izoplochy je prakticky totožná s všeobecnou implementáciou MC a nie je ďalej uvedená.

Hostiteľský kód

Trieda `vc::isoextractor` predstavuje jednotku získania izoplochy z SVO. Metóda `extract_isosurface` volá najprv `calc_null`, vracajúcu zotriedený zoznam NullCubes. Triedenie prebieha pomocou paralelného radixého triedenia (metóda `radix_sort`). Pretože tento zoznam ešte obsahuje duplicitné Mortonove kódy východzích vrcholov susedných kociek, musí nastať zunikátnenie (`uniquify`). To nie je nič viac než kompakcia podľa masky identifikujúcej unikátne hodnoty porovnaním susedných prvkov. Výpis A.13 zobrazuje, ako ďalej `null_cubes` súvisí so spracovaním riedkeho objemu.

Histopyramída

Za zmienku stojí implementácia 5:1 histopyramídy. Kvôli šetreniu miestom nie je vstup (o počte N) zarovnaný na mocninu piatich a vyplýva z toho použitie zoznamu ofsetov na určovanie počiatku jednotlivých úrovní v 1D poli. Potom výška 5:1 histopyramídy sa vypočíta vzťahom $h = \left\lceil \frac{\log_2 N}{\log_2 5} \right\rceil$. Následne $n_h = N$, a v iteráciách $i \in \{h-1, \ldots, 0\}$ sa určí veľkosť úrovne i ako $n_i = \left\lceil \frac{n_{i+1}}{5} \right\rceil$. Zostáva dopočítať zoznam ofsetov na základe hodnôt n_i.

Budovanie histopyramídy prebieha rekurzívnym štýlom pre každú úroveň. Rozmer boku (bočná pamäť pre redukciu) je totožný s počtom `uint4` elementov v úrovni histopyramídy. V každej iterácii je alokovaný odpovedajúci počet skupín o šírke $5 \cdot x$ vlákien, kde x sa rovná veľkosti warpu/wavefrontu. Rovnako je rezervovaná lokálna pamäť ($5 \cdot x$ prvkov). Kód kernelu (výpis A.14) spočiatku zaplní lokálnu pamäť pomocou všetkých vlákien v skupine. Následne jediný warp/wavefront dosadí na správnu pozíciu v úrovni 4D vektor hodnôt a vypočíta parciálnu sumu boku.

Kapitola 7

Merania a výsledky

Testy prebehli pre voxelizáciu i extrakciu izoplochy. Niektoré experimenty pre porovnanie optimalizácií kódu sú tu taktiež uvedené. Pre meranie bol použitý CPU Intel Core i3-3120M @ 2,5 GHz, 4 GB RAM a grafický čip nVidia GeForce GT 635M (1350 MHz, 1 GB VRAM).

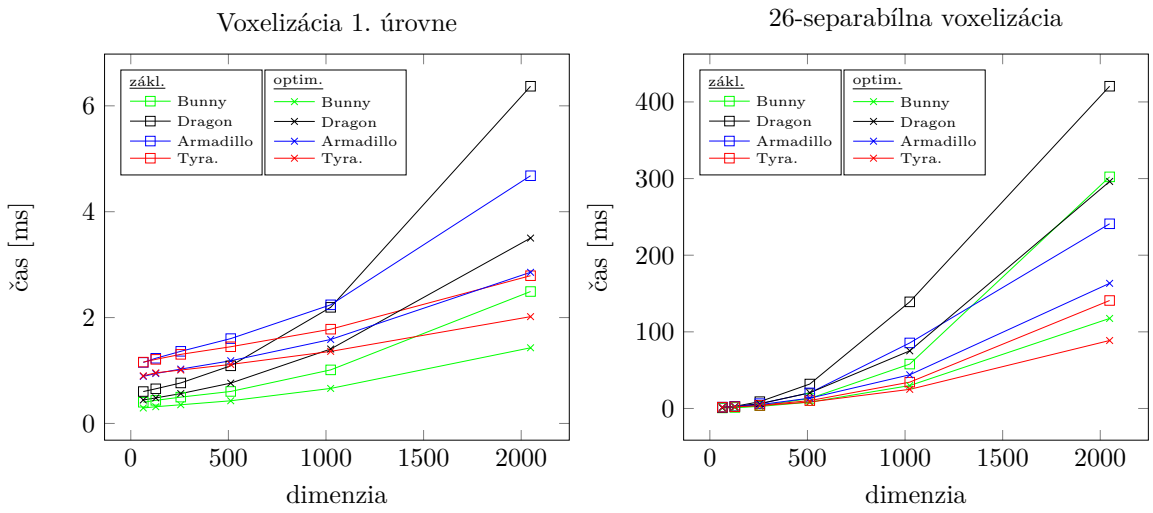
7.1 Voxelizácia

Tabuľka 7.1 ukazuje rozloženie časovej náročnosti medzi časťami celého procesu voxelizácie všetkých variant. Meranie prebehlo nad verejne dostupnými známymi 3D modelmi[1].

3D model	**rozl.**	**zistenie** ActNodes$_1$	**vybudovanie SVO**	**voxelizácia 0. úrovne**			**propagácia v smere x**	**veľkosť SVO**
				26-sep.	6-sep.	úplná		
Stanford Bunny (69630 troj.)	64^3	0,3045	0,0189	0,6246	0,451	0,492	0,0121	39,944
	128^3	0,3307	0,0336	1,1153	0,7838	0,7182	0,0305	171,976
	256^3	0,3713	0,0755	2,675	1,8195	1,2322	0,0901	679,752
	512^3	0,4629	0,2067	8,2425	5,5877	2,5693	0,3431	2700,3
	1024^3	0,8091	0,7214	29,8987	20,4483	7,33	1,3983	11103
	2048^3	2,3299	2,8614	117,492	81,0766	24,8021	6,0443	46826,8
Stanford Dragon (100000 troj.)	64^3	0,4533	0,0193	1,1169	0,8059	0,7466	0,0108	48,712
	128^3	0,4906	0,0362	2,2739	1,5874	1,1507	0,0357	228,168
	256^3	0,5694	0,088	6,1348	4,266	2,2623	0,1084	940,936
	512^3	0,7833	0,2674	20,5121	14,5356	5,8985	0,4011	3775,11
	1024^3	1,5222	0,9663	75,1887	54,2373	18,8986	1,6289	155526,3
	2048^3	4,3413	3,961	296,206	215,462	69,5096	7,1436	937,432
Stanford Armadillo (212574 troj.)	64^3	0,8943	0,0188	1,6053	1,1941	1,258	0,0111	38,472
	128^3	0,9517	0,0335	2,367	1,672	1,4968	0,0221	163,272
	256^3	1,0378	0,0735	5,0336	3,4628	2,4636	0,0735	642,76
	512^3	1,2007	0,1953	13,452	9,1336	4,6662	0,2844	2556,81
	1024^3	1,7006	0,6569	43,9551	30,205	11,1116	1,1537	10194,7
	2048^3	3,6737	2,5681	163,302	113,956	33,5768	4,8571	630,043
Stanford Tyrannosaurus (200000 troj.)	64^3	0,916	0,0184	1,5276	1,1762	1,1978	0,0097	32,2
	128^3	0,9593	0,0312	2,1223	1,5536	1,3974	0,0203	119,688
	256^3	1,0359	0,0602	3,6929	2,5493	1,9241	0,0586	432,968
	512^3	1,1353	0,1422	8,4545	5,5932	3,2319	0,1882	1634,5
	1024^3	1,4726	0,4418	24,9663	16,4607	6,5062	0,7361	6443,53
	2048^3	2,8216	1,6439	88,6884	59,0887	17,3554	3,1063	26227,2

Tabuľka 7.1: Čas jednotlivých procedúr voxelizácie (ms) a veľkosť SVO (kB) bez ukazateľov na susedné uzly v x-ovom smere. Pre toto testovanie boli vybraté najvýkonnejšie varianty.

[1] Vo Wavefront OBJ formáte sú dostupné z http://www.prinmath.com/csci5229/OBJ/index.html a ešte http://www.mrbluesummers.com/3572/downloads/stanford-dragon-model pre draka.



Obr. 7.1: Porovnanie doby voxelizácie s optimalizovanou variantou.

Testované varianty používali posúvacie LUT pre výpočet Mortonových kódov, optimalizáciu určovania hĺbkového intervalu počas testu náležitosti a v prípade úplnej voxelizácie sa 0. úroveň najprv skúmala v rámci 4×4 dlaždíc. Povrchová 26-separabílna voxelizácia sa stáva najpomalejším druhom, tesne následovaná 6-separabílnou modifikáciou. Najväčším dopadom na výkon je samozrejme počet vykonaných 2D testov, ktorých je v prípade determinovania superkrytu najväčší počet. U 6-separabílnej voxelizácie dochádza k nižšiemu počtu prekrytí a podobne hĺbkový interval je oproti predchádzajúcej verzii o 1 voxel kratší. Toto všetko naznačuje výsledky úplnej voxelizácie. Behom nej sa sleduje náležitosť stredu daného voxela v jedinej rovine a teda tomu odpovedajú viditeľne nízke časové nároky výpočtu. Propagácia k tomu nie je vôbec extrémne kritickou oblasťou a vo väčšine prípadov netrvá dlhšie než voxelizácia 1. úrovne. Evidentne je však postihnutá predovšetkým počtom volaní kernelov, čo sa ukazuje najviac vo vysokom rozlíšení. Kumulatívny čas je konečne zaznamenaný v tabuľke 7.2.

Dodatočne k povrchovej voxelizácii, zaujímavé je porovnanie základnej a optimalizovanej verzii testu náležitosti. Experimenty ukazujú (graf na obrázku 7.1), aký vplyv má zisťovanie hĺbkového intervalu na celkovú dobu výpočtov. Viac je to vidieť na voxelizácii nultej úrovne, avšak i procedúra počas zisťovania ActNodes_1 bola značne postihnutá. Taktiež bola vyskúšaná varianta s 4×4 dlaždicami v rovine xy, avšak ukázalo sa, že neposkytuje chcené zrýchlenie, ba až zvyšuje časovú zložitosť pri nízkych dimenziách Γ. Tento prístup sa teda vyplatil iba u úplnej voxelizácie.

Sklamaním sa stalo porovnanie techník výpočtu Mortonových kódov. Testovaná bola klasická verzia s cyklom a modifikácia využívajúca posúvacie LUT. Táto varianta na použitom hardvér nanešťastie nepriniesla takmer žiadne zlepšenie. Napriek tomu je pravdepodobné, že na menej výkonných grafických kartách môže byť LUT verzia úspešnejšia.

V kontraste s výsledkami v pôvodnej práci [46] bol zaznamenaný výkonnostný rozdiel voxelizácie nultej úrovne, kedy ich implementácia vykazuje väčšie zrýchlenie. V tomto prípade meranie prebiehalo na nVidia GeForce GTX 285 [46], ktoré poskytuje o niečo lepší procesorový takt (1476 MHz) a až 240 CUDA jadier oproti 144 na GT 635M. Prekvapivo sa však väčšina ostatných procedúr pohybuje buď veľmi blízko, alebo dokonca beží rýchlejšie (model Stanford Bunny) – typicky sa jedná o prvé dve procedúry a hierarchickú propagáciu.

3D model	Stanford Bunny			Stanford Dragon			Stanford Armadillo			Stanford Tyrannosaurus		
rozl.	64^3	128^3	256^3	64^3	128^3	256^3	64^3	128^3	256^3	64^3	128^3	256^3
26-sep.	0,948	1,4796	3,1218	1,5895	2,8007	6,7922	2,5184	3,3522	6,1449	2,462	3,1128	4,789
6-sep.	0,7744	1,1481	2,2663	1,2785	2,1142	4,9234	2,1072	2,6572	4,5741	2,1106	2,5441	3,6454
úplná	0,8275	1,113	1,7691	1,23	1,7132	3,0281	2,1822	2,5041	3,6484	2,1419	2,4082	3,0788
rozl.	512^3	1024^3	2048^3	512^3	1024^3	2048^3	512^3	1024^3	2048^3	512^3	1024^3	2048^3
26-sep.	8,9121	31,4292	122,6835	21,5628	77,6772	304,5083	14,848	46,3126	169,5438	9,732	26,8807	93,1539
6-sep.	6,2573	21,9788	86,2681	15,5863	56,7258	223,7643	10,5296	32,5625	120,1978	6,8707	18,3751	63,5542
úplná	3,582	10,2588	36,0379	7,3503	23,016	84,9555	6,3466	17,6228	44,6757	4,6976	9,1567	24,9272

Tabuľka 7.2: Celkový čas voxelizácie (ms).

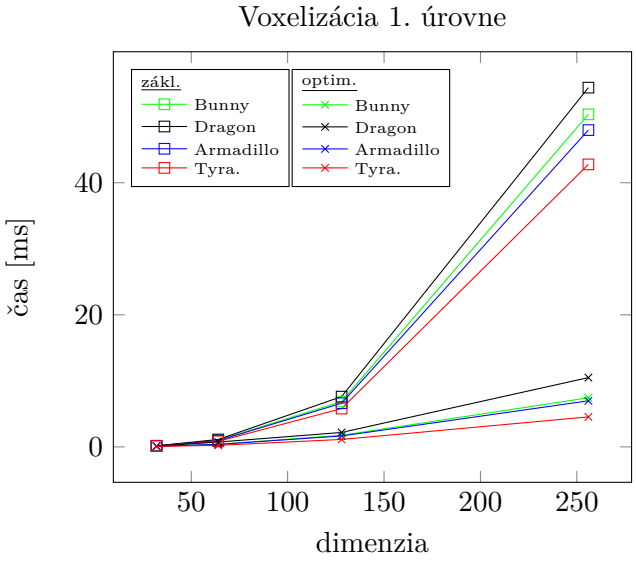
7.2 Extrakcia izoplochy

Podobne v tabuľke 7.3 vidieť namerané hodnoty pre navrhnutý postup extrakcie izoplochy z SVO. Povšimnúť si treba drastického zníženia prechádzaného objemu vo vyšších dimenziách, pričom v najnižšom testovanom rozlíšení (32^3) sa niekedy jedná už o viac ako polovičnú redukciu.

3D model	**rozlíšenie**	**určenie** NullCubes	**izo-extrakcia**	**celkovo**	**počet kociek**
Stanford Bunny 	$128^3 \Rightarrow 32^3$	0,08541	0,1073	0,19271	17302 (52,8 %)
	$256^3 \Rightarrow 64^3$	0,31085	0,3925	0,70335	44863 (17,11 %)
	$512^3 \Rightarrow 128^3$	1,2716	1,7585	3,0301	319888 (15,25 %)
	$1024^3 \Rightarrow 256^3$	5,477	7,437	12,914	1297786 (7,74 %)
	$2048^3 \Rightarrow 512^3$	24,1428	32,2691	56,4119	5518020 (4,11 %)
Stanford Dragon 	$128^3 \Rightarrow 32^3$	0,1039	0,1256	0,2295	16134 (49,24 %)
	$256^3 \Rightarrow 64^3$	0,4131	0,7336	1,1467	101997 (38,91 %)
	$512^3 \Rightarrow 128^3$	1,7285	2,1887	3,9172	227841 (10,86 %)
	$1024^3 \Rightarrow 256^3$	7,5462	10,4914	18,0376	1779193 (10,6 %)
	$2048^3 \Rightarrow 512^3$	33,8111	45,664	79,4751	7801427 (5,81 %)
Stanford Armadillo 	$128^3 \Rightarrow 32^3$	0,1255	0,0862	0,2117	16930 (51,67 %)
	$256^3 \Rightarrow 64^3$	0,293	0,3844	0,6774	71791 (27,39 %)
	$512^3 \Rightarrow 128^3$	1,1809	1,6555	2,8364	300776 (14,34 %)
	$1024^3 \Rightarrow 256^3$	4,9969	6,9752	11,9721	1196492 (7,13 %)
	$2048^3 \Rightarrow 512^3$	21,7831	29,7531	51,5362	5109443 (3,81 %)
Stanford Tyranno-saurus 	$128^3 \Rightarrow 32^3$	0,0632	0,0697	0,1329	11907 (36,34 %)
	$256^3 \Rightarrow 64^3$	0,2002	0,2383	0,4385	27629 (10,54 %)
	$512^3 \Rightarrow 128^3$	0,7728	1,1288	1,9016	181086 (8,63 %)
	$1024^3 \Rightarrow 256^3$	3,1703	4,5409	7,7112	726136 (4,33 %)
	$2048^3 \Rightarrow 512^3$	13,5915	19,0288	32,6203	2953250 (2,2 %)

Tabuľka 7.3: Celkový čas extrakcie izoplochy (ms) a počet prechádzaných kociek – percentá vyjadrujú, aký zlomok predstavujú z celého voxelového priestoru. Stĺpec rozlíšenia definuje podvzorkovanie daného objemu.

Graf na obrázku 7.2 ďalej dobre znázorňuje, aký efekt má redukcia prechádzaného priestoru na časové požiadavky extrakcie izoplochy. Samozrejme vidieť, že sa takýto postup vypláca a približne aspoň 85% zrýchlenie možno pozorovať v tejto časti algoritmu, čo úspešně zanedbáva čas spotrebovaný pre zisťovanie NullCubes. Z pamäťovej nenáročnosti taktiež vyplýva, že nový postup je schopný spracovať i väčšie rozlíšenia SVO, ktoré nebolo možné namerať u bežnej varianty. Evidentne sa dá tento algoritmus považovať za použiteľný a skvele sa hodí pre prácu s SVO tak, aby sa dali počas extrakcie izoplochy zachovať detaily.



Obr. 7.2: Porovnanie trvania základného a modifikovaného MC algoritmu.

7.3 Vyhodnotenie implementácie

Voxelizácia bola implementovaná ako plne GPU-akcelerovaná štruktúra algoritmov spracovajúcich vstupnú geometriu a na výstup produkuje presný objem v podobe SVO. Implementácia testov náležitosti minimalizuje počet atomických operácií sledovaním posledne adresovaného pamäťového miesta, optimalizuje samotné zisťovanie prekrytia viacerými spôsobmi a pokúša sa zjednodušiť prechádzanie oktálového stromu použitím stratégií ako vzostúpenie do spoločného predka alebo skrátenie doby trvania výpočtu 64-bitových Mortonových kódov (čo sa však na použitom hardvér nevyplatilo). Jednotlivé povrchové voxelizácie zrejme dokážu ideálne nastaviť obsiahnutý voxelový priestor, spĺňajúc tým podmienku presnosti a minimality. Úplná voxelizácia korektne vypĺňuje vnútorný priestor voxelizovaného modelu a snaží sa zrýchliť procedúru vyplnenia 0. úrovne testovaním 4×4 dlaždíc. Framework je schopný pre transportné účely transformovať SVO do XML, čo bolo otestované pri vizualizácii sledovaním paprsku. Rozumnejšie by ale bolo uskutočniť nejaký druh kompresie pre reálné scénáre.

Vylepšený MC algoritmus si kladie za cieľ rýchlu a pamäťovo nenáročnú extrakciu izoplochy. To sa aj naozaj podarilo a výsledky hovoria o potenciálne inteligentnej modifikácii MC, ktorá si zaiste zaslúži pozornosť v ďalšom výzkume. Kritickou oblasťou tejto techniky je prevažne stanovenie NullCubes a pretože je SVO usporiadaný podľa z-krivky, možno takejto skutočnosti skúsiť využiť pri priamom triedení. Nevýhodou implementovaného MC algoritmu je, že produkuje rýdzu trojuholníkovú sieť a teda obsahuje duplikované vrcholy, čo typicky nebýva problém pri okamžitom vykresľovaní (t.j. polygonizácia prebieha priamo v rámci fragmentového shaderu). Na druhú stranu by však bolo lepšie aplikovať určenie zoznamu indexov (a tým pádom znížiť počet vrcholov) pre uloženie danej geometrie do všeobecných súborových formátov. Ďalej zrejmým optimalizačným prípadom môže byť behom klasifikácie kociek zdieľať vyhodnotenie vrcholov kociek, avšak nepravidelná topológia SVO komplikuje tento proces. Zaujímavým vylepšením MC je využitie duality [41] – to nabáda k vytvoreniu lepšie usporiadanej siete s trojuholníkmi rôznej veľkosti, z časových dôvodov sa ale takýto prístup neskusil.

Kapitola 8

Záver

Bola naštudovaná a popísaná problematika povrchovej a úplnej binárnej voxelizácie 3D polygonových modelov so zameraním na výkon, presnosť, šetrenie pamäťovým miestom a využitím pri ďalšom spracovaní. S ohľadom na tieto požiadavky bola zvolená vhodná technika 3D skenovacej konverzie pre implementáciu sprievodného projektu. Produktom je framework voCCeL so schopnosťou 6-/26-separabílnej povrchovej a úplnej binárnej voxelizácie do SVO, z ktorého ďalej poskytuje možnosť extrakcie hladkej izoplochy šetrným algoritmom založeným na MC. Tieto komponenty spracovanie plne akcelerujú na grafickej karte a teda poskytujú vhodné podklady pre realizáciu kompozície VCSG z rôznej vstupnej geometrie v reálnom čase, čo môže byť ďalším vývojom tejto práce. Aplikovaných bolo v rámci implementácie voxelizácie mnoho optimalizácií zhromaždených z naštudovanej literatúry, niekedy doplnených o vlastné vylepšenia pre dané riešenie. Výsledky meraní ukázali, ako framework zvláda spracovať rozličné 3D modely jednotlivými druhmi voxelizácií. Celkovo sa výkon blíží práci pôvodných autorov so zohľadnením na použitý hardvér. Dobré výsledky preukázala modifikovaná varianta MC, ktorá dokáže redukovať prechádzaný počet kociek až o približne 98 % pri vysokých rozlíšeniach a znížiť dobu polygonizácie o asi 85 %, čím ju robí schopnú spracovať i veľmi hlboké SVO.

Literatúra

[1] Akenine-Möller, T.: Fast 3D Triangle-box Overlap Testing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA: ACM, 2005.

[2] Allard, J.; Faure, F.; Courtecuisse, H.; ai.: Volume Contact Constraints at Arbitrary Resolution. *ACM Trans. Graph.*, ročník 29, č. 4, Júl 2010: s. 82:1–82:10, ISSN 0730-0301.
URL https://www.cs.mcgill.ca/~kry/pubs/sig10/sig10ldi-preprint.pdf

[3] Bærentzen, A.; Christensen, N. J.: *A Technique for Volumetric CSG based on Morphology*. Viedeň, Rakúsko: Springer Vienna, 2001, ISBN 978-3-7091-6756-4, s. 117–130.

[4] Billeter, M.; Olsson, O.; Assarsson, U.: Efficient Stream Compaction on Wide SIMD Many-core Architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-603-8, s. 159–166.

[5] Breen, D. E.; Mauch, S.; Whitaker, R. T.: 3D Scan Conversion of CSG Models into Distance Volumes. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98, New York, NY, USA: ACM, 1998, ISBN 1-58113-105-4, s. 7–14.
URL https://www.cs.drexel.edu/~david/Papers/david_volviz98.pdf

[6] Chen, H.; Fang, S.: Fast Voxelization of Three-dimensional Synthetic Objects. *J. Graph. Tools*, ročník 3, č. 4, December 1998: s. 33–45, ISSN 1086-7651.

[7] Cohen, D.; Kaufman, A. E.; Wang, Y.: Generating a Smooth Voxel-Based Model from an Irregular Polygon Mesh. *The Visual Computer*, ročník 10, č. 6, 1994: s. 295–305, ISSN 0178-2789.
URL http://cvc.cs.sunysb.edu/Publications/1994/CKW94

[8] Cohen, J.; Hickey, T.: Two Algorithms for Determining Volumes of Convex Polyhedra. *J. ACM*, ročník 26, č. 3, Júl 1979: s. 401–414, ISSN 0004-5411.

[9] Cohen-Or, D.; Kaufman, A.: Fundamentals of Surface Voxelization. *Graph. Models Image Process.*, ročník 57, č. 6, November 1995: s. 453–461, ISSN 1077-3169.
URL http://cvc.cs.stonybrook.edu/Publications/1995/CK95/file.pdf

[10] Cohen-Or, D.; Kaufman, A.: 3D Line Voxelization and Connectivity Control. *IEEE Comput. Graph. Appl.*, ročník 17, č. 6, November 1997: s. 80–87, ISSN 0272-1716.

[11] Crane, K.; Llamas, I.; Tariq, S.: Real-Time Simulation and Rendering of 3D Fluids. In *GPU Pro 3: Advanced Rendering Techniques*, editácia W. Engel, Boca Raton, FL, USA: A K Peters/CRC Press, 2012, s. 615–634.

[12] Crassin, C.; Green, S.: CRC Press, Patrick Cozzi and Christophe Riccio, 2012. URL http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf

[13] Crassin, C.; Neyret, F.; Lefebvre, S.; ai.: GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, ACM, Boston, MA, Etats-Unis: ACM Press, feb 2009.
URL http://maverick.inria.fr/Publications/2009/CNLE09

[14] Crassin, C.; Neyret, F.; Sainz, M.; ai.: Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels. In *GPU Pro*, editácia W. Engel, A K Peters, Júl 2010, s. 643–676.
URL https://hal.inria.fr/inria-00516416

[15] Crassin, C.; Neyret, F.; Sainz, M.; ai.: Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011)*, ročník 30, č. 7, sep 2011.
URL http://maverick.inria.fr/Publications/2011/CNSGE11b

[16] Dachille, F.; Kaufman, A. E.: Incremental Triangle Voxelization. 2000: s. 205–212.

[17] Dong, Z.; Chen, W.; Bao, H.; ai.: Real-time Voxelization for Complex Polygonal Models. In *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, PG '04, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2234-3, s. 43–50.

[18] Dyken, C.; Zeigler, G.: GPU-Accelerated Data Expansion for the Marching Cubes Algorithm. GPU Technology Conference, San Jose, CA, 2010.
URL http://on-demand.gputechconf.com/gtc/2010/presentations/S12020-GPU-Accelerated-Data-Expansion-Marching-Cubes-Algorithm.pdf

[19] Dyken, C.; Ziegler, G.; Theobalt, C.; ai.: High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum*, 2008, ISSN 1467-8659.
URL http://heim.ifi.uio.no/~erikd/pdf/hpmarcher_draft.pdf

[20] Dyllong, E.; Grimm, C.: A Reliable Extended Octree Representation of CSG Objects with an Adaptive Subdivision Depth. In *Parallel Processing and Applied Mathematics*, ročník 7, PPAM, September 2007, ISBN 978-3-540-68105-2, s. 1341–1350.

[21] Eisemann, E.; Décoret, X.: Fast Scene Voxelization and Applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM SIGGRAPH, 2006, s. 71–78.
URL http://maverick.inria.fr/Publications/2006/ED06

[22] Eisemann, E.; Décoret, X.: Single-pass GPU solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, GI 2008, Toronto, Ontario,

Canada: Canadian Human-Computer Communications Society, 2008, ISBN 978-1-56881-423-0, ISSN 0713-5424, s. 73–80.

[23] Fang, S.; Chen, H.: Hardware Accelerated Voxelization. *Computers and Graphics*, ročník 24, 2000.

[24] Fang, S.; Liao, D.: Fast CSG Voxelization by Frame Buffer Pixel Mapping. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, VVS '00, New York, NY, USA: ACM, 2000, ISBN 1-58113-308-1, s. 43–48.

[25] Feng, C.; Jalba, A. C.; Telea, A. C.: A Descriptor for Voxel Shapes Based on the Skeleton Cut Space. In *Eurographics Workshop on 3D Object Retrieval*, editácia A. Ferreira; A. Giachetti; D. Giorgi, The Eurographics Association, 2016, ISBN 978-3-03868-004-8, ISSN 1997-0471, doi:10.2312/3dor.20161082.

[26] Gibson, S. F. F.: Constrained Elastic Surface Nets: Generating Smooth Surfaces from Binary Segmented Data. In *Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention*, MICCAI '98, London, UK, UK: Springer-Verlag, 1998, ISBN 3-540-65136-5, s. 888–898. URL http://dl.acm.org/citation.cfm?id=646921.709482

[27] Harris, M.; Sengupta, S.; Owens, J. D.: Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, editácia H. Nguyen, Addison Wesley, August 2007, ISBN 0-321-51526-9.

[28] Hoffmann, C. M.: *Geometric and Solid Modeling: An Introduction*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, ISBN 1-55860-067-1.

[29] Huang, J.; Yagel, R.; Filippov, V.; ai.: An Accurate Method for Voxelizing Polygon Meshes. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98, New York, NY, USA: ACM, 1998, ISBN 1-58113-105-4, s. 119–126. URL http://www.cs.utk.edu/~huangj/papers/polygon.pdf

[30] Kaplanyan, A.; Dachsbacher, C.: Cascaded Light Propagation Volumes for Real-time Indirect Illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, New York, NY, USA: ACM, 2010, ISBN 978-1-60558-939-8, s. 99–107.

[31] Karabassi, E.-A.; Papaioannou, G.; Theoharis, T.: A Fast Depth-buffer-based Voxelization Algorithm. *J. Graph. Tools*, ročník 4, č. 4, December 1999: s. 5–10, ISSN 1086-7651.

[32] Kaufman, A.: Efficient Algorithms for 3D Scan-conversion of Parametric Curves, Surfaces, and Volumes. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, New York, NY, USA: ACM, 1987, ISBN 0-89791-227-6, s. 171–179.

[33] Kaufman, A.; Shimony, E.: 3D Scan-conversion Algorithms for Voxel-based Graphics. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, I3D '86, New York, NY, USA: ACM, 1987, ISBN 0-89791-228-4, s. 45–75.

[34] Kaufman, A. E.: Volume Visualization. *ACM Comput. Surv.*, ročník 28, č. 1, Marec 1996: s. 165–167, ISSN 0360-0300.

[35] Laine, S.; Karras, T.: Efficient Sparse Voxel Octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, New York, NY, USA: ACM, 2010, ISBN 978-1-60558-939-8, s. 55–63.

[36] Lee, Y. T.; Requicha, A. A. G.: Algorithms for Computing the Volume and Other Integral Properties of Solids. I. Known Methods and Open Issues. *Commun. ACM*, ročník 25, č. 9, September 1982: s. 635–641, ISSN 0001-0782.

[37] Lorensen, W. E.; Cline, H. E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.*, ročník 21, č. 4, August 1987: s. 163–169, ISSN 0097-8930, doi:10.1145/37402.37422.
URL http://doi.acm.org/10.1145/37402.37422

[38] Marechal, N.; Guerin, E.; Galin, E.; ai.: Heat Transfer Simulation for Modeling Realistic Winter Sceneries. *Computer Graphics Forum*, 2010, ISSN 1467-8659, doi:10.1111/j.1467-8659.2009.01614.x.
URL http://liris.cnrs.fr/~egalin/Pdf/2010-snow.pdf

[39] Mokrzycki, W.: Algorithms of Discretization of Algebraic Spatial Curves on Homogeneous Cubical Grids. *Computers & Graphics*, ročník 12, č. 3–4, 1988: s. 477–487, ISSN 0097-84931.
URL http://www.sciencedirect.com/science/journal/00978493

[40] Morton, G.: A Computer Oriented Geodetic Database and a New Technique in File Sequencing. Technická správa, 1966.

[41] Nielson, G. M.: Dual Marching Cubes. In *Proceedings of the Conference on Visualization '04*, VIS '04, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7803-8788-0, s. 489–496.

[42] Pantaleoni, J.: VoxelPipe: A Programmable Pipeline for 3D Voxelization. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, editácia C. Dachsbacher; W. Mark; J. Pantaleoni, ACM, 2011, ISBN 978-1-4503-0896-0, ISSN 2079-8687.
URL https://research.nvidia.com/publication/voxelpipe-programmable-pipeline-3d-voxelization

[43] Pineda, J.: A Parallel Algorithm for Polygon Rasterization. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, New York, NY, USA: ACM, 1988, ISBN 0-89791-275-6, s. 17–20.

[44] Rauwendaal, R.; Bailey, M.: Hybrid Computational Voxelization Using the Graphics Pipeline. *Journal of Computer Graphics Techniques (JCGT)*, ročník 2, č. 1, 2013: s. 15–37, ISSN 2331-7418.
URL http://jcgt.org/published/0002/01/02/

[45] Schwarz, M.: Practical Binary Surface and Solid Voxelization with Direct3D 11. In *GPU Pro 3: Advanced Rendering Techniques*, editácia W. Engel, Boca Raton, FL, USA: A K Peters/CRC Press, 2012, s. 337–352.

[46] Schwarz, M.; Seidel, H.-P.: Fast Parallel Surface and Solid Voxelization on GPUs. *ACM Transactions on Graphics*, ročník 29, č. 6 (Proceedings of SIGGRAPH Asia

2010), December 2010: s. 179:1–179:9.
URL http://research.michael-schwarz.com/publ/2010/vox/

[47] Thiedemann, S.; Henrich, N.; Grosch, T.; ai.: Voxel-based Global Illumination. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, New York, NY, USA: ACM, 2011, ISBN 978-1-4503-0565-5, s. 103–110.

[48] Zhang, L.; Chen, W.; Ebert, D. S.; ai.: Conservative Voxelization. *Vis. Comput.*, ročník 23, č. 9, August 2007: s. 783–792, ISSN 0178-2789.

[49] Zhao, Y.; Wei, X.; Fan, Z.; ai.: Voxels on Fire. In *Proceedings of the 14th IEEE Visualization 2003 (VIS03)*, IEEE Computer Society Washington, DC, USA, 2003, ISBN 0-7803-8120-3.
URL http://cvc.cs.sunysb.edu/Publications/2003/ZWFKQ03

Prílohy

Príloha A

Výpisy procedúr

Táto príloha obsahuje výpisy vybratých hostiteľských kódov voxelizácie a izo-extrakcie spolu s niektorými podstatnými kernelmi referovanými v kapitole 6. Mala by byť použitá súčasne s touto kapitolou pretože ďalej neobsahuje žiadne komentáre, text je však rozčlenený do podkapitol pre ľahšiu orientáciu. Upozornenie: výpisy sú rozmiestnené voľne tak, aby bol text pekne usporiadaný, preto musí čitateľ dávať pozor na referenčné čísla výpisov behom štúdia.

A.1 Voxelizácia

Hostiteľský kód

```
1  cl::vector<morton> voxelizer::lvl1_voxelize(const tri_mesh& model,
2                                              const unsigned short dim) const
3  {
4    const unsigned tri_cnt = model.indices.size() / 3;
5
6    constexpr const unsigned char delta1{8};
7    const unsigned short dim1(dim / delta1);
8    const unsigned len = pow(dim, 3) /
9                         static_cast<cl_uint>(pow(delta1, 3)) /
10                        BITMASK32_LEN;
11
12   const cl::vector<bitmask32> volume{simd, CL_MEM_READ_WRITE, len};
13   nullify(volume);
14
15   exe["lvl1_voxelize"].set_args(model.vertices, model.indices,
16                                 dim1, volume, tri_cnt);
17   simd.launch(exe["lvl1_voxelize"], {}, {tri_cnt});
18
19   const cl::vector<cl_uint> tmp{simd, CL_MEM_READ_WRITE, len};
20   exe["count_volume"].set_args(volume, tmp, len);
21   simd.launch(exe["count_volume"], {}, {len});
22
23   unsigned n_a_1 = tmp.read(simd, len - 1, 1)[0];
24   prescan(tmp);
25   n_a_1 += tmp.read(simd, len - 1, 1)[0];
26
27   const cl::vector<morton> actnodes1{simd, CL_MEM_WRITE_ONLY, n_a_1};
28   exe["write_mortons"].set_args(volume, tmp, actnodes1, len);
29   simd.launch(exe["write_mortons"], {}, {len});
30
31   return actnodes1;
32 }
```

Výpis A.1: Hostiteľský kód určenia ActNodes_1.

```
octree voxelizer::build_octree(cl::vector<morton>& actnodesj) const
{
  const unsigned n_a_1 = actnodesj.size();
  const cl::vector<morton> actnodes1{simd, CL_MEM_READ_ONLY, n_a_1};
  actnodesj.copy(simd, actnodes1, 0, 0, n_a_1);

  std::vector<unsigned> offsets{0};
  std::vector<cl::vector<octree::node>> lvls{{simd, CL_MEM_READ_WRITE, 8 * n_a_1}};

  unsigned size = lvls[0].size();
  exe["init_nodes0"].set_args(lvls[0], size);
  simd.launch(exe["init_nodes0"], {}, {size});

  unsigned n_a_j;
  for (n_a_j = n_a_1; n_a_j > 64;) {
    const cl::vector<cl_uchar> flag{simd, CL_MEM_READ_WRITE,
                                    n_a_j};
    exe["tag_parents"].set_args(actnodesj, flag, n_a_j);
    simd.launch(exe["tag_parents"], {}, {n_a_j});

    const cl::vector<cl_uint> parent_index{simd, CL_MEM_READ_WRITE, n_a_j};
    mutate(flag, parent_index);

    prescan(parent_index);

    const unsigned n_a_i = parent_index.read(simd, n_a_j - 1, 1)[0] + 1;
    lvls.push_back({simd, CL_MEM_READ_WRITE, 8 * n_a_i});
    size += lvls.back().size();

    std::for_each(offsets.begin(), offsets.end(),
                  [&lvls] (unsigned& o) { o += lvls.back().size(); });
    offsets.insert(offsets.begin(), 1, 0);

    exe["init_nodesi"].set_args(lvls.back(), static_cast<cl_uint>(lvls.back().size()));
    simd.launch(exe["init_nodesi"], {}, {lvls.back().size()});

    exe["link_lvli"].set_args(lvls.back(), lvls[lvls.size() - 2],
                              actnodesj, parent_index, n_a_j);
    simd.launch(exe["link_lvli"], {}, {n_a_j});

    const cl::vector<morton> actnodesi{simd, CL_MEM_READ_WRITE, n_a_j};
    exe["strip_mortons"].set_args(actnodesj, actnodesi, flag, parent_index, n_a_j);
    simd.launch(exe["strip_mortons"], {}, {n_a_j});

    actnodesj = actnodesi;
    n_a_j = n_a_i;
  }

  // level n - 3
  const cl::vector<cl_uchar> flag{simd, CL_MEM_READ_WRITE, 64};
  const cl::vector<cl_uint> parent_index{simd, CL_MEM_READ_WRITE, 64};
  exe["tag_parents"].set_args(actnodesj, flag, 64);
  simd.launch(exe["tag_parents"], {}, {64});

  mutate(flag, parent_index);
  prescan(parent_index);

  lvls.push_back({simd, CL_MEM_READ_WRITE, 64});

  exe["init_nodesn3"].set_args(lvls.back(), 64);
  simd.launch(exe["init_nodesn3"], {}, {64});

  exe["link_lvli"].set_args(lvls.back(), lvls[lvls.size() - 2], actnodesj, parent_index, n_a_j);
  simd.launch(exe["link_lvli"], {}, {n_a_j});

  std::for_each(offsets.begin(), offsets.end(), [] (unsigned& o) { o += 73; });
  offsets.insert(offsets.begin(), 3, 0);
  offsets[1] = 1;
  offsets[2] = 9;
  offsets[3] = 73;

  size += 73;
  octree svo{simd, size, offsets, actnodes1};
  nullify(svo.get_subgrid());

  // stitching (x pointers) & copying levels to the SVO
  for (unsigned char i = 2; lvls.size() > 1; ++i) {
    exe["stitch_lvli"].set_args(lvls.back(), lvls[lvls.size() - 2],
                                static_cast<cl_uint>(lvls.back().size()));
    simd.launch(exe["stitch_lvli"], {}, {lvls.back().size()});

    lvls.back().copy(simd, svo.get_root(), 0, offsets[i], lvls.back().size());
    lvls.pop_back();
  }

  lvls.back().copy(simd, svo.get_root(), 0, offsets.back(), lvls.back().size());
  lvls.clear();

  return svo;
}
```

Výpis A.2: Metóda pre vybudovanie SVO.

```
void voxelizer::propagate_bitflips(const octree& svo) const
{
  const unsigned size = svo.size();
  const unsigned char depth = svo.depth();

  const std::vector<unsigned> offsets = svo.get_offsets().read(simd, 0,
                                                               svo.depth());

  unsigned sizei{size - offsets.back()},
           aligni;
  {
  const cl::vector<octree::node> root0{simd, svo.get_root(), 0,
                                       offsets.back(), sizei, &aligni};

  exe["concatenate_subgrid"].set_args(root0, aligni, svo.get_subgrid(),
                                      sizei >> 1);
  simd.launch(exe["concatenate_subgrid"], {}, {sizei >> 1});
  }

  {
  sizei = offsets.back() - offsets[depth - 2];
  const cl::vector<octree::node> root1{simd, svo.get_root(), 0,
                                       offsets[depth - 2],
                                       size - offsets[depth - 2],
                                       &aligni};
  exe["flip_lvl1"].set_args(root1, aligni, svo.get_subgrid(),
                            sizei);
  simd.launch(exe["flip_lvl1"], {}, {sizei});

  exe["concatenate_lvli"].set_args(root1, aligni, sizei >> 1);
  simd.launch(exe["concatenate_lvli"], {}, {sizei >> 1});
  }

  for (unsigned char i = depth - 2; i > 1; --i) {
    sizei = offsets[i] - offsets[i - 1];
    const cl::vector<octree::node> rooti{simd, svo.get_root(),
                                         0,
                                         offsets[i - 1],
                                         offsets[i + 1] -
                                         offsets[i - 1],
                                         &aligni};

    exe["flip_lvli"].set_args(rooti, aligni, sizei);
    simd.launch(exe["flip_lvli"], {}, {sizei});

    exe["concatenate_lvli"].set_args(rooti, aligni, sizei / 2);
    simd.launch(exe["concatenate_lvli"], {}, {sizei / 2});
  }

  for (unsigned char i{0}; i < depth - 2; ++i) {
    sizei = offsets[i + 1] - offsets[i];
    const cl::vector<octree::node> rooti{simd, svo.get_root(),
                                         0,
                                         offsets[i],
                                         offsets[i + 2] -
                                         offsets[i],
                                         &aligni};

    exe["propagate_lvli"].set_args(rooti, aligni, sizei);
    simd.launch(exe["propagate_lvli"], {}, {sizei});
  }

  {
  sizei = offsets.back() - offsets[depth - 2];
  const cl::vector<octree::node> root1{simd, svo.get_root(), 0,
                                       offsets[depth - 2],
                                       size - offsets[depth - 2],
                                       &aligni};

  exe["propagate_subgrid"].set_args(root1, aligni, svo.get_subgrid(),
                                    sizei);
  simd.launch(exe["propagate_subgrid"], {}, {sizei});
  }
}
```

Výpis A.3: Funkcia propagovania stavových prepnutí úplnej voxelizácie.

Kernely

```
__kernel void lvl1_voxelize(__global const float3* const vertices,
                            __global const int3* const indices,
                            const ushort dim,
                            __global bitmask32* const volume,
                            const uint tri_cnt)
{
  const size_t gid = get_global_id(0);
  if (gid >= tri_cnt)
    return;

  triangle t = pull_polygon(gid, vertices, indices);
  // account for sampling at the voxel center
#pragma unroll
  for (uchar i = 0; i < 3; ++i)
    t.v[i].x += .5f;

  float3 n = normalize(cross(t.v[1] - t.v[0], t.v[2] - t.v[1]));
  const axis z_axis = dominant_axis(n);
  swizzle(&t, &n, z_axis);
  // shift edge function crit. points that are on the voxel's max (original) +x face
  const float3 c_shift = z_axis == X ? (float3) (0.f, 0.f, 1.f) :
                         z_axis == Y ? (float3) (0.f, 1.f, 0.f) :
                                       (float3) (1.f, 0.f, 0.f);

  const float3 e[3] = {t.v[1] - t.v[0],
                       t.v[2] - t.v[1],
                       t.v[0] - t.v[2]};

  // SETUP STAGE:
  const char3 sgn = {n.x < 0.f ? -1 : 1,
                     n.y < 0.f ? -1 : 1,
                     n.z < 0.f ? -1 : 1};
  const float2 ne_xy[3] = {(float2) (-e[0].y, e[0].x) * sgn.z,
                           (float2) (-e[1].y, e[1].x) * sgn.z,
                           (float2) (-e[2].y, e[2].x) * sgn.z},
               ne_yz[3] = {(float2) (-e[0].z, e[0].y) * sgn.x,
                           (float2) (-e[1].z, e[1].y) * sgn.x,
                           (float2) (-e[2].z, e[2].y) * sgn.x},
               ne_zx[3] = {(float2) (-e[0].x, e[0].z) * sgn.y,
                           (float2) (-e[1].x, e[1].z) * sgn.y,
                           (float2) (-e[2].x, e[2].z) * sgn.y};
  const float d_xy[3] = {-dot(ne_xy[0], t.v[0].xy) +
                         max(0.f, (DELTA1 + c_shift.x) * ne_xy[0].x) +
                         max(0.f, (DELTA1 + c_shift.y) * ne_xy[0].y),
                         -dot(ne_xy[1], t.v[1].xy) +
                         max(0.f, (DELTA1 + c_shift.x) * ne_xy[1].x) +
                         max(0.f, (DELTA1 + c_shift.y) * ne_xy[1].y),
                         -dot(ne_xy[2], t.v[2].xy) +
                         max(0.f, (DELTA1 + c_shift.x) * ne_xy[2].x) +
                         max(0.f, (DELTA1 + c_shift.y) * ne_xy[2].y)},
              d_yz[3] = {-dot(ne_yz[0], t.v[0].yz) +
                         max(0.f, (DELTA1 + c_shift.y) * ne_yz[0].x) +
                         max(0.f, (DELTA1 + c_shift.z) * ne_yz[0].y),
                         -dot(ne_yz[1], t.v[1].yz) +
                         max(0.f, (DELTA1 + c_shift.y) * ne_yz[1].x) +
                         max(0.f, (DELTA1 + c_shift.z) * ne_yz[1].y),
                         -dot(ne_yz[2], t.v[2].yz) +
                         max(0.f, (DELTA1 + c_shift.y) * ne_yz[2].x) +
                         max(0.f, (DELTA1 + c_shift.z) * ne_yz[2].y)},
              d_zx[3] = {-dot(ne_zx[0], t.v[0].zx) +
                         max(0.f, (DELTA1 + c_shift.z) * ne_zx[0].x) +
                         max(0.f, (DELTA1 + c_shift.x) * ne_zx[0].y),
                         -dot(ne_zx[1], t.v[1].zx) +
                         max(0.f, (DELTA1 + c_shift.z) * ne_zx[1].x) +
                         max(0.f, (DELTA1 + c_shift.x) * ne_zx[1].y),
                         -dot(ne_zx[2], t.v[2].zx) +
                         max(0.f, (DELTA1 + c_shift.z) * ne_zx[2].x) +
                         max(0.f, (DELTA1 + c_shift.x) * ne_zx[2].y)};

  n *= sgn.z; // ensure z_min < z_max
  const float dmin = dot(n, t.v[0]) - max(0.f, DELTA1 * n.x) -
                     max(0.f, DELTA1 * n.y),
              dmax = dot(n, t.v[0]) - min(0.f, DELTA1 * n.x) -
                     min(0.f, DELTA1 * n.y);

  // TEST STAGE:
  prev_val tmp = {true, 0, 0};

  aabb bbox = compute_bbox(&t);
  bbox.min -= c_shift; // enlarge the triangle's bounding box
                       // in the (original) -x direction
  bbox.min = clamp(floor(bbox.min / DELTA1), (float3) (0.f), (float3) (dim - 1)) * DELTA1;
  bbox.max = clamp(ceil(bbox.max / DELTA1), (float3) (0.f), (float3) (dim - 1)) * DELTA1;
  for (short x = bbox.min.x; x <= bbox.max.x; x += DELTA1)
    for (short y = bbox.min.y; y <= bbox.max.y; y += DELTA1) {
      const float2 b = {x, y};

      if (dot(ne_xy[0], b) + d_xy[0] < 0.f ||
          dot(ne_xy[1], b) + d_xy[1] < 0.f ||
          dot(ne_xy[2], b) + d_xy[2] < 0.f)
```

```
91          continue;
92
93        const float nb = -dot(n.xy, b);
94        const short zmin = max(floor2mul((nb + dmin) / n.z - DELTA1, DELTA1), bbox.min.z),
95                    zmax = min(ceil2mul((nb + dmax) / n.z, DELTA1), bbox.max.z);
96
97        for (short z = zmin; z <= zmax; z += DELTA1) {
98          const float3 p = {b.x, b.y, z};
99
100         if (dot(ne_yz[0], p.yz) + d_yz[0] < 0.f ||
101             dot(ne_yz[1], p.yz) + d_yz[1] < 0.f ||
102             dot(ne_yz[2], p.yz) + d_yz[2] < 0.f ||
103             dot(ne_zx[0], p.zx) + d_zx[0] < 0.f ||
104             dot(ne_zx[1], p.zx) + d_zx[1] < 0.f ||
105             dot(ne_zx[2], p.zx) + d_zx[2] < 0.f)
106           continue;
107
108         const morton m = morton3d_encode(unswizzle(p, z_axis) / DELTA1);
109         const uchar w = m & 0x3F;
110         update_or(((m >> 6) << 1) + (w < BITMASK32_LEN ? 0 : 1),
111                   1 << (w - (w < BITMASK32_LEN ? 0 : BITMASK32_LEN)),
112                   &tmp, volume);
113       }
114     }
115   if (!tmp.first)
116     atomic_or(&volume[tmp.i], tmp.v);
117 }
```

Výpis A.4: Voxelizácia prvej úrovne.

```
1  __kernel void write_mortons(__global const bitmask32* const volume,
2                              __global const uint* const offsets,
3                              __global morton* const actnodes1, const uint n)
4  {
5    const size_t gid = get_global_id(0);
6    if (gid >= n)
7      return;
8
9    const bitmask32 v = volume[gid];
10   const uint offset = offsets[gid];
11   for (uint i = 0, j = 0; i < BITMASK32_LEN; ++i)
12     if (v & (1 << i))
13       actnodes1[offset + j++] = (gid << 5) | (i + (gid % 2 ? BITMASK32_LEN : 0));
14 }
```

Výpis A.5: Získanie ActNodes$_1$.

```
1  __kernel void solid_voxelize(__global const float3* const vertices,
2                               __global const int3* const indices,
3                               const ushort dim,
4                               __global const octree_node* const root,
5                               __constant const uint* const offsets,
6                               const uchar depth,
7                               __global bitmask32* const subgrid,
8                               const uint tri_cnt)
9  {
10   const size_t gid = get_global_id(0);
11   if (gid >= tri_cnt)
12     return;
13
14   const octree svo = {root, offsets, depth};
15   const triangle t = pull_polygon(gid, vertices, indices);
16
17   const float3 e[3] = {t.v[1] - t.v[0], t.v[2] - t.v[1], t.v[0] - t.v[2]},
18                n = normalize(cross(e[0], e[1]));
19
20   // SETUP STAGE:
21   const char sgn_x = n.x < 0.f ? -1 : 1;
22   const float2 ne_yz[3] = {(float2) (-e[0].z, e[0].y) * sgn_x,
23                            (float2) (-e[1].z, e[1].y) * sgn_x,
24                            (float2) (-e[2].z, e[2].y) * sgn_x};
25   const float d_yz1[3] = {-dot(ne_yz[0], t.v[0].yz),
26                           -dot(ne_yz[1], t.v[1].yz),
27                           -dot(ne_yz[2], t.v[2].yz)},
28               d_yz4[3] = {d_yz1[0] +
29                           max(0.f, DELTA0 * ne_yz[0].x) +
30                           max(0.f, DELTA0 * ne_yz[0].y),
31                           d_yz1[1] +
32                           max(0.f, DELTA0 * ne_yz[1].x) +
33                           max(0.f, DELTA0 * ne_yz[1].y),
34                           d_yz1[2] +
35                           max(0.f, DELTA0 * ne_yz[2].x) +
36                           max(0.f, DELTA0 * ne_yz[2].y)};
37   const float d = dot(n, t.v[0]);
38   // top-left filling rule
39   const bool tl[3] = {ne_yz[0].x > 0.f ||
40                       (ne_yz[0].x == 0.f && ne_yz[0].y < 0.f),
41                       ne_yz[1].x > 0.f ||
```

```
                      (ne_yz[1].x == 0.f && ne_yz[1].y < 0.f),
                      ne_yz[2].x > 0.f ||
                      (ne_yz[2].x == 0.f && ne_yz[2].y < 0.f)};

  // TEST STAGE:
  prev_node last = {0, 0};
  // first loop over the abstract level-0 voxel tile
  // (its actual size is of 4x4 voxels in yz plane)
  aabb bbox = compute_bbox(&t);
  bbox.min = clamp(floor(bbox.min / DELTA0), (float3) (0.f),
                   (float3) (dim / DELTA0 - 1)) * DELTA0;
  bbox.max = clamp(ceil(bbox.max / DELTA0), (float3) (0.f),
                   (float3) (dim / DELTA0 - 1)) * DELTA0;
  for (short j = bbox.min.y; j <= bbox.max.y; j += DELTA0)
    for (short k = bbox.min.z; k <= bbox.max.z; k += DELTA0) {
      const float2 b = {j, k};

      if (dot(ne_yz[0], b) + d_yz4[0] < 0.f ||
          dot(ne_yz[1], b) + d_yz4[1] < 0.f ||
          dot(ne_yz[2], b) + d_yz4[2] < 0.f)
        continue;

      // active tile identified:
      // loop over its 4x4 voxel extents
      for (short y = b.x; y <= b.x + DELTA0 - 1; ++y)
        for (short z = b.y; z <= b.y + DELTA0 - 1; ++z) {
          const float2 p =  {y + .5f, z + .5f};

          const float c[3] = {dot(ne_yz[0], p) + d_yz1[0],
                              dot(ne_yz[1], p) + d_yz1[1],
                              dot(ne_yz[2], p) + d_yz1[2]};
          if (c[0] < 0.f || c[1] < 0.f || c[2] < 0.f ||
              (c[0] == 0.f && !tl[0]) || (c[1] == 0.f && !tl[1]) ||
              (c[2] == 0.f && !tl[2]))
            continue;

          // projection to get the x coordinate
          const short q = clamp(convert_int(((-dot(n.yz, p) + d) /
                                             n.x) + .5f),
                                0, dim - 1);
          // identify the affected level-0 node
          const morton m = morton3d_encode((uint3) (q, y, z));
          __global const octree_node* const node = octree_address(m,
                                                                   svo,
                                                                   &last);
          if (node) {
            const uchar w = m & 0x3F;
            bitmask32 f = 0;
            for (uchar x = q % DELTA0,
                 i = w - (w < BITMASK32_LEN ?
                          0 : BITMASK32_LEN);
                 x < DELTA0; ++x, i = ztrav_4x4x2[i])
              f |= 1 << i;
            atomic_xor(&subgrid[OCT_PTR_VALUE(node->child) +
                       (w < BITMASK32_LEN ? 0 : 1)], f);
          }
        }
    }
}
```

Výpis A.6: Úplná voxelizácia voxelových tehál.

```
__kernel void propagate_subgrid(__global octree_node* const root1,
                                const uint align1,
                                __global bitmask32* const subgrid,
                                const uint size1)
{
  const size_t gid = get_global_id(0);
  if (gid >= size1)
    return;

  __global octree_node* const node1 = &root1[align1 + gid];
  // test the inside flag
  if (OCT_PTR_VALID(node1->child) && OCT_MSB_TEST(node1->child)) {
#pragma unroll
    for (uchar j = 0; j < 8; ++j) {
      __global octree_node* const node0 = &root1[align1 +
                                                 size1 +
                                                 OCT_PTR_VALUE(node1->child) +
                                                 j];
      // flip all bits
      subgrid[OCT_PTR_VALUE(node0->child)] ^= 0xFFFFFFFF;
      subgrid[OCT_PTR_VALUE(node0->child) + 1] ^= 0xFFFFFFFF;
    }
    OCT_MSB_FLIP(node1->child);
  }
}
```

Výpis A.7: Propagácia do voxelových tehál.

```
__kernel void concatenate_subgrid(__global const octree_node* const root0,
                                  const uint align0,
                                  __global bitmask32* const subgrid,
                                  const uint size0)
{
  const size_t gid = get_global_id(0);
  if (gid >= size0)
    return;

  __global const octree_node* node0 = &root0[align0 + (gid << 1)];
  if (!OCT_PTR_VALID(node0->xprev))
    while (OCT_PTR_VALID(node0->xnext)) {
      const bitmask32 v[2] = {subgrid[OCT_PTR_VALUE(node0->child)],
                              subgrid[OCT_PTR_VALUE(node0->child) +
                                      1]};

      node0 = &root0[align0 + OCT_PTR_VALUE(node0->xnext)];
#pragma unroll
      for (uchar i = 0; i < 2; ++i) {
        const bitmask32 f = ((v[i] & 0x200) ? 0x303 : 0) |
                            ((v[i] & 0x800) ? 0xC0C : 0) |
                            ((v[i] & 0x2000) ? 0x3030 : 0) |
                            ((v[i] & 0x8000) ? 0xC0C0 : 0) |
                            ((v[i] & 0x2000000) ? 0x3030000 : 0) |
                            ((v[i] & 0x8000000) ? 0xC0C0000 : 0) |
                            ((v[i] & 0x20000000) ? 0x30300000 : 0) |
                            ((v[i] & 0x80000000) ? 0xC0C00000 : 0);

        if (f)
          subgrid[OCT_PTR_VALUE(node0->child) + i] ^= f;
      }
    }
}
```

Výpis A.8: Propagácia medzi voxelovými tehlami.

```
__kernel void flip_lvl1(__global octree_node* const root1, const uint align1,
                        __global const bitmask32* const subgrid,
                        const uint size1)
{
  const size_t gid = get_global_id(0);
  if (gid >= size1)
    return;

  __global octree_node* node1 = &root1[align1 + gid];
  if (OCT_PTR_VALID(node1->child)) {
    bool set = false;
    // we care about children that can be at the end of an x-linked
    // node string (odd indices)
    for (uchar i = 1; i < 8 && !set; i += 2) {
      // (level-0 nodes are also included!)
      __global const octree_node* const node0 = &root1[align1 +
                                                        size1 +
                                                        OCT_PTR_VALUE(node1->child) +
                                                        i];
      // must be at the end of an x-linked node string
      if (!OCT_PTR_VALID(node0->xnext)) {
        const bitmask32 v[2] = {subgrid[OCT_PTR_VALUE(node0->child)],
                                subgrid[OCT_PTR_VALUE(node0->child) + 1]};

        // test all relevant bits
        // (they have local x index = 3)
        set = ((v[0] & 0xAA00AA00) == 0xAA00AA00) &&
              ((v[1] & 0xAA00AA00) == 0xAA00AA00);
      }
    }

    // flip flag F is stored in the MSB of -x-neighbour pointer
    if (set)
      OCT_MSB_SET(node1->xprev);
  }
}
```

Výpis A.9: Prepnutie 1. úrovne.

```
__kernel void concatenate_lvli(__global octree_node* const rooti,
                               const uint aligni, const uint sizei)
{
  const size_t gid = get_global_id(0);
  if (gid >= sizei)
    return;

  __global octree_node* nodei = &rooti[aligni + (gid << 1)];
  if (!OCT_PTR_VALID(nodei->xprev))
    while (OCT_PTR_VALID(nodei->xnext)) {
      // test the flip flag
      const bool flip = OCT_MSB_TEST(nodei->xprev);
      nodei = &rooti[aligni + OCT_PTR_VALUE(nodei->xnext)];
      if (flip) {
        // flip the inside flag & the flip flag
        // of the +x neighbour
        OCT_MSB_FLIP(nodei->child);
        OCT_MSB_FLIP(nodei->xprev);
      }
    }
}
```

Výpis A.10: Konkatenácia x-zreťazených uzlov i-tej úrovne.

```
__kernel void flip_lvli(__global octree_node* const rooti, const uint aligni,
                        const uint sizei)
{
  const size_t gid = get_global_id(0);
  if (gid >= sizei)
    return;

  __global octree_node* const nodei = &rooti[aligni + gid];
  if (OCT_PTR_VALID(nodei->child)) {
    bool set = false;
    for (uchar j = 1; j < 8 && !set; j += 2) {
      __global const octree_node* const nodej = &rooti[aligni +
                                                       sizei +
                                                       OCT_PTR_VALUE(nodei->child) +
                                                       j];
      set = !OCT_PTR_VALID(nodej->xnext) &&
            OCT_MSB_TEST(nodej->xprev);
    }

    // set F
    if (set)
      OCT_MSB_SET(nodei->xprev);
  }
}
```

Výpis A.11: Prepnutie medzi úrovňami i a $i + 1$.

```
__kernel void propagate_lvli(__global octree_node* const rooti,
                             const uint aligni, const uint sizei)
{
  const size_t gid = get_global_id(0);
  if (gid >= sizei)
    return;

  __global octree_node* const nodei = &rooti[aligni + gid];
  // has children and its inside flag set:
  if (OCT_PTR_VALID(nodei->child) && OCT_MSB_TEST(nodei->child)) {
    // flip the children's inside flag
#pragma unroll
    for (uchar j = 0; j < 8; ++j) {
      __global octree_node* const nodej = &rooti[aligni +
                                                 sizei +
                                                 OCT_PTR_VALUE(nodei->child) +
                                                 j];
      OCT_MSB_FLIP(nodej->child);
    }
    // unset the node's inside flag
    OCT_MSB_FLIP(nodei->child);
  }
}
```

Výpis A.12: Všeobecná propagácia do spodnej úrovne.

A.2 Extrakcia izoplochy

Hostiteľský kód

```
std::vector<float> isoextractor::extract_isosurface(const octree& svo,
                                                    unsigned short dim,
                                                    float isovalue) const
{
  dim /= 4;
  if (isovalue <= 0.f)
    isovalue = .1f;
  else if (isovalue >= 1.f)
    isovalue = .9f;

  const cl::vector<morton> null_cubes{uniquify(calc_null(svo))};
  const unsigned size0 = svo.len() >> 1;
  const cl::vector<cl_uint> tri_cnts{simd, CL_MEM_READ_WRITE,
                                     size0 + null_cubes.size()};

  exe["classify_cubes"].set_args(svo.get_root(), svo.get_offsets(),
                                 svo.depth(), svo.get_subgrid(),
                                 svo.get_actnodes1(), null_cubes,
                                 tri_cnts, isovalue, dim, size0,
                                 static_cast<cl_uint>(null_cubes.size()));
  simd.launch(exe["classify_cubes"], {}, {tri_cnts.size()});

  histopyramid5 pyr{hp5_builder.build(tri_cnts)};

  const cl::vector<cl_float> vertices{simd, CL_MEM_WRITE_ONLY, 9 * pyr.get_sum()};
  exe["polygonise"].set_args(pyr.get_top(), pyr.get_steps(), pyr.height(),
                             pyr.get_sum(), svo.get_root(), svo.get_offsets(),
                             svo.depth(), svo.get_subgrid(),
                             svo.get_actnodes1(), null_cubes, vertices,
                             isovalue, dim, size0);
  simd.launch(exe["polygonise"], {}, {pyr.get_sum()});
  return vertices.read(simd, 0, vertices.size());
}
```

Výpis A.13: Hostiteľský kód izo-extrakcie.

Histopyramída

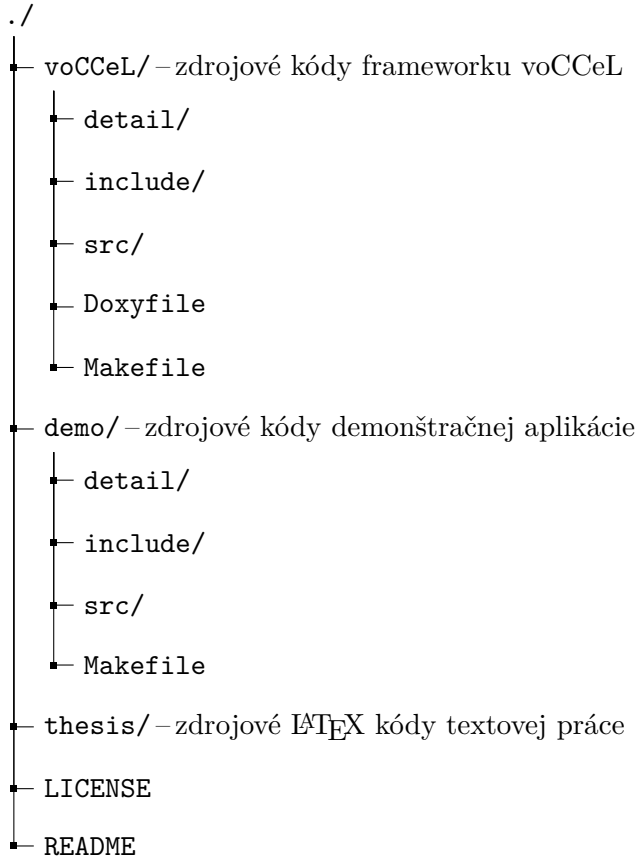
```
__attribute__((vec_type_hint(uint4)))
__attribute__((reqd_work_group_size(5 * WAVEFRONT_WIDTH, 1, 1)))
__kernel void hp5_build_lvl(__global const uint* const in, const uint n,
                            __local uint* const tmp,
                            __global uint4* const lvl, const uint align,
                            __global uint* const sideband, const uint m)
{
  const size_t gid = get_global_id(0),
               lid = get_local_id(0),
               grp = get_group_id(0);

  tmp[lid] = gid < n ? in[gid] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  // a single wavefront computes the elements of a level in a group
  if (lid < WAVEFRONT_WIDTH) {
    const uint4 v = {tmp[5 * lid], tmp[5 * lid + 1],
                     tmp[5 * lid + 2], tmp[5 * lid + 3]};

    const size_t i = WAVEFRONT_WIDTH * grp + lid;
    if (i < m) {
      lvl[align + i] = v;
      sideband[i] = v.x + v.y + v.z + v.w +
                    tmp[5 * lid + 4];
    }
  }
}
```

Výpis A.14: Kód redukcie úrovne 5:1 histopyramídy. Možno si všimnúť snahu zamedziť bankovým konfliktom využívaním rozostupov pri pristupovaní do vyrovnávacej pamäte.

Príloha B

Obsah priloženého pameťového média

Priložené CD poskytuje zdrojové súboy demonštračnej aplikácie a textu diplomovej práce. Pravidlá modifikácie/distribúcie špecifikuje priložená licenčná zmluva.



Príloha C

Manuál

Prvým krokom zprovoznenia demonštračnej aplikácie je kompilácia frameworku voCCeL. Užívateľ naviguje do `voCCeL/`, kde najprv spustí `make init` pre rozšírenie adresárovej štruktúry a následne `make`. voCCeL potrebuje pre zkompilovanie GCC schopné spracovať C++ 17, v systéme prítomnú implementáciu OpenCL (spolu s umiestnením hlavičkových súborov v `PATH`) a taktiež nainštalovanú knižnicu `pugixml` pre exportovanie SVO do XML. Pre manuálne určenie umiestnenia knižníc slúži premenná `EXTERN` v relatívnom `Makefile`.

Demonštračná aplikácia (adresár `demo/`) je závislá na viacerých knižniciach, ktorých výčet je k dispozícii v priloženom README. Odpovedajúci `Makefile` spúšťa GCC, ktorý by mal podporovať C++ štandard 17 (testované na verzii `6.3.1`). Užívateľ sa najprv ujistí, že `make` bude vedieť o umiestení všetkých vyžadovaných knižníc (premenná `EXTERN` v `Makefile`). Následne spustí `make init`, ktorý rozšíri adresárovú štruktúru pre umiestnenie spustiteľného súboru a potom `make` pre kompiláciu.

Upozornenie: na Windows môže byť postup komplikovanejší, ale `Makefile` je pomerne jednoznačne napísaný, nemal by byť teda problém pridať dodatočné informácie o knižniciach (viz premenná `LIBNAMES`) a preložiť pomocou niektorej z GNU distribúcií.

Po zkompilovaní môže užívateľ spustiť demonštračnú aplikáciu príkazom `bin/demo`. Nápovedu k tomuto programu možno získať spustením `bin/demo -h`. Jedná sa o OpenGL program, ktorý nechá voCCeL spracovať vstupnú geometriu a na základe argumentov z príkazového riadka buď zobrazí SVO priamo pomocou sledovania paprsku (neoptimalizované), prípadne spustí modifikovaný MC algoritmus pre zobrazenie trojuholníkovej siete (užívateľ by mal dať pozor na to, aby bola predtým spustená úplná voxelizácia). Ak bola zvolená varianta zobrazenia SVO sledovaním paprsku, môže užívateľ použiť klávesy `-` a `+` pre zobrazenie jednotlivých úrovní. Aplikácia nechá užívateľa ovládať kameru z pohľadu prvej osoby klasickou kombináciou klávies `WSAD` a pohybom myši, aby bol schopný lubovolného prieletu scénou. Vykreslovacie okno sa zatvára klávesou `Esc`.

Príloha D

Plagát