FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Implementace datové struktury pro polyhedrální sítě v knihovně TNL
Student:	Bc. Ján Bobot
Vedoucí:	Ing. Jakub Klinkovský
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Template Numerical Library (TNL, www.tnl-project.org) je knihovna jejímž cílem je usnadnit vývoj programů pro náročné počítačové simulace. Cílem tohoto tématu je rozšířit stávající implementaci podpory nestrukturovaných numerických sítí na možnost práce s polyhedrálními sítěmi.

1. Seznamte se s návrhem knihovny TNL a základními návrhovými vzory použitými při implementaci – lambda funkce, ParallelFor, zařízení, základní datové struktury – array, vector, atd.
2. Seznamte se s aktuální implementací datové struktury pro nestrukturované sítě v knihovně TNL.
3. Navrhněte způsob zobecnění této datové struktury pro obecné polygonální a polyhedrální sítě.
4. Zobecněnou datovou strukturu implementujte v jazyku C++ v rámci knihovny TNL.
5. Správnost implementace ověřte pomocí unit testů a jednoduchých testovacích úloh.
6. Pro různé reálné polygonální a polyhedrální sítě zkoumejte efektivitu implementované datové struktury a proveďte srovnání s původní implementací.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 26. října 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práca

Implementace datové struktury pro polyhedrální sítě v knihovně TNL

Bc. Ján Bobot

Katedra teoretické informatiky
Vedúci práce: Ing. Jakub Klinkovský

3. januára 2022

Poďakovanie

Chcel by som poďakovať vedúcemu Ing. Jakubovi Klinkovskému za vedenie tejto práce a za poskytnutie veľmi cenných rád a spätnej väzby pri vytváraní práce. Ďalej by som chcel tiež poďakovať Ing. Tomášovi Oberhuberovi, Ph.D za dodatočné vedenie práce a za poskytnutie dodatočnej spätnej väzby pri vytváraní práce.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov. V súlade s ustanovením § 46 odst. 6 tohoto zákona týmto udeľujem bezvýhradné oprávnenie (licenciu) k užívaniu tejto mojej práce, a to vrátane všetkých počítačových programov ktoré sú jej súčasťou alebo prílohou a tiež všetkej ich dokumentácie (ďalej len „Dielo“), a to všetkým osobám, ktoré si prajú Dielo užívať.

Tieto osoby sú oprávnené Dielo používať akýmkoľvek spôsobom, ktorý neznižuje hodnotu Diela, a za akýmkoľvek účelom (vrátane komerčného využitia). Toto oprávnenie je časovo, územne a množstevne neobmedzené. Každá osoba, ktorá využije vyššie uvedenú licenciu, sa však zaväzuje priradiť každému dielu, ktoré vznikne (čo i len čiastočne) na základe Diela, úpravou Diela, spojením Diela s iným dielom, zaradením Diela do diela súborného či zpracovaním Diela (vrátane prekladu), licenciu aspoň vo vyššie uvedenom rozsahu a zároveň sa zaväzuje sprístupniť zdrojový kód takého diela aspoň zrovnateľným spôsobom a v zrovnateľnom rozsahu ako je zprístupnený zdrojový kód Diela.

V Prahe 3. januára 2022 . . . . . .

Odkaz na túto prácu

Bobot, Ján. *Implementace datové struktury pro polyhedrální sítě v knihovně TNL*. Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Jedno zo zameraní knižnice TNL (Template Numerical Library) je podpora dátovej štruktúry pre neštruktúrované numerické siete. Jej hlavnou výhodou je schopnosť byť použitá vo vedeckých výpočtoch, ktoré sa vykonávajú na rôznych paralelných hardvérových architektúrach, ako napríklad viacjadrové procesory a grafické akcelerátory. Dátová štruktúra obsahuje podporu pre 2D štvoruholníkové, 3D šesťstenné a ľubovoľné n-dimenzionálne simplex siete. Táto práca sa zaoberá návrhom a implementáciou rozšírenia pôvodnej dátovej štruktúry o podporu obecných mnohouholníkových a mnohostenných sietí. Správnosť implementovaného rozšírenia je overená pomocou unit testov na malých sietiach. Nasledovne je skúmaná efektivita rozšírenej dátovej štruktúry pre rôzne reálne mnohouholníkové a mnohostenné siete na základe niekoľkých testovacích úloh a je tiež vykonané porovnanie s pôvodnou implementáciou.

Klíčová slova neštrukturovaná sieť, dátová štruktúra, mnohouholník, mnohosten, C++, TNL

Abstract

One of the goals of the Template Numerical Library (TNL) is the support of the data structure for unstructured numerical meshes. The main advantage of the data structure is its ability to be used in the scientific calculations, that are executed on various parallel hardware architetures such as multi-core processors and graphics accelerators. Data structure supports 2D quadrilateral, 3D hexahedron and arbitrary n-dimensional simplex meshes. This thesis deals with the extension of the original data structure to support general polygonal and polyhedral meshes. The correctness of the implemented data structure is verified by unit tests for small meshes. Its effectiveness for various real polygonal and polyhedral meshes is investigated using several benchmark problems and the comparison with the original implementation is also performed.

Keywords unstructured mesh, data structure, polygon, polyhedron, C++, TNL

Obsah

Úvod **1**
Ciele a požiadavky práce 2
Organizácia práce 2

1 Teoretická časť **3**
1.1 Základné pojmy 3
1.2 Programovanie grafických akcelerátorov 4
1.3 Reprezentácia matíc v pamäti 6
1.3.1 Compressed Sparse Row (CSR) formát 6
1.3.2 Ellpack formát 7
1.3.3 Sliced Ellpack formát 8
1.4 Template Numerical Library (TNL) 9
1.4.1 Koncept zariadení v TNL 9
1.4.2 Dátové štruktúry v TNL 9
1.4.2.1 Polia a vektory 9
1.4.2.2 Matice 10
1.4.3 Využitie lambda funkcií v TNL 10
1.4.4 Algoritmy v TNL 10
1.4.4.1 Paralelný for cyklus 10
1.4.4.2 Statický for cyklus 11
1.4.4.3 Paralelná redukcia 11
1.4.4.4 Exkluzívna prefixová suma 11
1.4.5 TNL View štruktúry 11

2 Dátová štruktúra pre neštruktúrované siete **13**
2.1 Reprezentácia štruktúry siete 13
2.1.1 Reprezentácia priestorových súradníc 14
2.1.2 Reprezentácia pod-/nad-entít 15
2.1.3 Susedstvo bunky 17

2.1.4 Značky entít 19
2.2 Implementačné detaily 20
2.2.1 Rozhranie 20
2.2.2 Statická konfigurácia 22
2.2.3 Vrstvy dátovej štruktúry 23
2.2.4 Reprezentácia typov topológii 27
2.2.5 Traits triedy 28
2.2.6 Inicializácia siete 29
2.2.6.1 Inicializačný algoritmus 30
2.2.7 Vstup/Výstup z/do súboru 33

3 Rozšírenie dátovej štruktúry 35
3.1 Nové topológie 35
3.2 Zmeny v reprezentácii štruktúry siete 39
3.3 Implementačné detaily 40
3.3.1 Definovanie typov nových topológií 40
3.3.2 Rozlišovanie medzi statickými a dynamickými topológiami 42
3.3.3 Definovanie vrstiev pre dynamické topológie 42
3.3.4 Načítanie sieti nových topológii zo súboru 45
3.3.5 Zmeny a zovšeobecnenia v inicializácii siete 46

4 Testovacie úlohy 51
4.1 Dekompozícia 52
4.1.1 Dekompozícia mnohouholníka 53
4.1.1.1 Vejárová triangulácia (dekompozícia typu P) . 53
4.1.1.2 Ťažisková triangulácia (dekompozícia typu C) 54
4.1.2 Dekompozícia mnohostenu 55
4.1.3 Implementácia dekompozície entít 56
4.1.4 Implementácia dekompozície siete 58
4.2 Korekcia nerovinných mnohouholníkov 60
4.2.1 Detekovanie rovinných mnohouholníkových entít 61
4.2.2 Implementácia rovinnej korekcie siete 61
4.3 Miera 64
4.3.1 Miera mnohouholníka 64
4.3.1.1 2D mnohouholník 64
4.3.1.2 3D rovinný mnohouholník 65
4.3.2 Miera mnohostenu 66
4.3.3 Implementácia miery siete 67

5 Testovanie a experimentálna analýza 69
5.1 Testovanie 69
5.2 Metológia merania 70
5.2.1 Dáta 71
5.2.2 Platforma 72

5.3 Meranie 73
5.3.1 Pamäťová náročnosť 73
5.3.2 Načítanie zo súboru 75
5.3.3 Inicializácia 77
5.3.4 Kópia medzi CPU a GPU pamäťou 79
5.3.5 Výpočet miery 81
5.3.6 Dekompozícia 84
5.3.7 Korekcia nerovinných mnohouholníkov 87

Záver **89**
Ciele a výsledky 89
Práca do budúcnosti 91

Literatúra **93**

A Zoznam použitých skratiek **95**

B Definície konfigurácií sietí použitých pri meraní **97**
B.1 Maximálna konfigurácia 97
B.2 Minimálna konfigurácia 98

C Príklad inicializácie siete pomocou triedy MeshBuilder **99**
C.1 Štvoruholníková sieť 99
C.2 Mnohostenná sieť 100

D Príklad súboru vo formáte FPMA **103**

E Obsah priloženého CD **105**

Zoznam obrázkov

1.1 CSR formát [1] 7
1.2 Ellpack formát (riadková orientácia) [1] 7
1.3 Sliced Ellpack formát (stĺpcová orientácia) [1] 8

2.1 Reprezentácia priestorových súradníc v pamäti 15
2.2 Príklad štvoruholníkovej siete s bunkami c_0, c_1, stenami f_0, f_1, f_2, f_3, f_4, f_5, f_6 a vrcholmi v_0, v_1, v_2, v_3, v_4, v_5 17
2.3 Incidentné matice pre sieť z obrázku 2.2 17
2.4 Príklad štvoruholníkovej siete s bunkami c_0, ..., c_8, stenami f_0, ..., f_{23} a vrcholmi v_0, ..., v_{15} 18
2.5 Ajacenčná matica duálneho grafu pre sieť z obrázku 2.4 19
2.6 Reprezentácia značiek entít pre sieť z obrázku 2.4 20
2.7 Diagram dedičnosti šablónovej triedy `Mesh` 24
2.8 Diagram dedičnosti d-dimenzionálnej špecializácie šablónovej triedy `StorageLayer` 25
2.9 Diagram dedičnosti šablónovej triedy `EntityTags::LayerFamily` . 26
2.10 Triedy a volané metódy pri inicializácii D-dimenzionálnej siete. `EntityInitializer< Config, d, d >` slúži len pre ukončenie rekurzie 31

3.1 Rôzne mnohouholníky 36
3.2 Toroidný mnohosten 36
3.3 20-sten 36
3.4 Ihlan so štvoruholníkovou podstavou 37
3.5 Klin 37

4.1 Vejárová dekompozícia 7-uholníka 53
4.2 Ťažisková dekompozícia 7-uholníka 54

Zoznam tabuliek

5.1 Vlastnosti mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle_X}$) sietí použitých pri meraní: počet vrcholov #V, počet stien #F, počet buniek #C 71

5.2 Vlastnosti mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle_{XY}}$) sietí použitých pri meraní: počet vrcholov #V, počet stien #F, počet buniek #C 72

5.3 Parametre systému použitého pri meraní 73

5.4 Porovnanie pamäťovej náročnosti Pamäť [MB] mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle_X}$) sietí. $Fa_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle_C}$, $2D_i^{\triangle_P}$ 73

5.5 Porovnanie pamäťovej náročnosti Pamäť [MB] mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle_{XY}}$) sietí. $Fa_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle_{CC}}$, $3D_i^{\triangle_{CP}}$, $3D_i^{\triangle_{PC}}$, $3D_i^{\triangle_{PP}}$ 74

5.6 Porovnanie výpočtového času CT [ms] pre operáciu načítania zo súboru na mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle_X}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle_C}$, $2D_i^{\triangle_P}$ . . 75

5.7 Porovnanie výpočtového času CT [ms] pre operáciu načítania zo súboru na mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle_{XY}}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle_{CC}}$, $3D_i^{\triangle_{CP}}$, $3D_i^{\triangle_{PC}}$, $3D_i^{\triangle_{PP}}$ 76

5.8 Porovnanie výpočtového času CT [s], CPU paralelného zrýchlenia Sp pre operáciu inicializácie na mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle_X}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle_C}$, $2D_i^{\triangle_P}$ 77

5.9 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre operáciu inicializácie na mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle_{XY}}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle_{CC}}$, $3D_i^{\triangle_{CP}}$, $3D_i^{\triangle_{PC}}$, $3D_i^{\triangle_{PP}}$ 78

5.10 Porovnanie výpočtového času CT [ms] pre operáciu kópie medzi CPU a GPU pamäťou na mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle X}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle C}$, $2D_i^{\triangle P}$ 79
5.11 Porovnanie výpočtového času CT [ms] pre operáciu kópie medzi CPU a GPU pamäťou na mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle XY}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle CC}$, $3D_i^{\triangle CP}$, $3D_i^{\triangle PC}$, $3D_i^{\triangle PP}$. Pre siete $3D_5^{\triangle CC}$ a $3D_5^{\triangle PC}$ nebola pre maximálnu konfiguráciu operácia vykonaná kvôli nedostatku pamäte GPU 80
5.12 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp, GPU paralelného zrýchlenia GSp pre úlohu výpočtu mieri buniek na mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle X}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle C}$, $2D_i^{\triangle P}$ 82
5.13 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp, GPU paralelného zrýchlenia GSp pre úlohu výpočtu mieri buniek na mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle XY}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle CC}$, $3D_i^{\triangle CP}$, $3D_i^{\triangle PC}$, $3D_i^{\triangle PP}$ 83
5.14 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu C na mnohouholníkových ($2D_i^{\square}$) sietiach 85
5.15 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu P na mnohouholníkových ($2D_i^{\square}$) sietiach 85
5.16 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu CC na mnohostenných ($3D_i^{\square}$) sietiach 85
5.17 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu CP na mnohostenných ($3D_i^{\square}$) sietiach 86
5.18 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu PC na mnohostenných ($3D_i^{\square}$) sietiach 86
5.19 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu PP na mnohostenných ($3D_i^{\square}$) sietiach 86
5.20 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu korekcie nerovinných mnohouholníkov s dekompozíciou typu C na mnohouholníkových ($2D_i^{\square}$) sietiach 87
5.21 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu korekcie nerovinných mnohouholníkov s dekompozíciou typu P na mnohouholníkových ($2D_i^{\square}$) sietiach 88

5.22 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu korekcie nerovinných mnohouholníkov s dekompozícou typu C na mnohostenných ($3D_i^{\square}$) sietiach 88
5.23 Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu korekcie nerovinných mnohouholníkov s dekompozíciou typu P na mnohostenných ($3D_i^{\square}$) sietiach 88

Úvod

Numerické siete patria medzi základné štruktúry, ktoré sa používajú v rôznych numerických metódach aplikovaných na problémy z rôznych odvetví. Pre dostatočnú aproximáciu problémov so zložitými geometriami, je často potrebné zvažovať neštrukturované siete namiesto štruktúrovaných sietí. Pre zaistenie vysokého výkonu pri vedeckých výpočtoch je požadované, aby reprezentácia a manipulácia neštrukturovanej siete v pamäti počítača, bola čo najefektívnejšia.

Existuje niekoľko softvérových nástrojov a knižníc, ktoré poskytujú dátovú štruktúru pre reprezentáciu neštruktúrovaných sietí vo vedeckých výpočtoch. Medzi najznámejšie patria OpenMesh [2], ViennaGrid [3], deal.II [4], OpenFoam [5], libMesh [6], VTK [7], CVGlib [8] a DUNE [9, 10]. Z výkonnostného hľadiska, sa vývoj neštruktúrovaných sietí v predošlo spomenutých riešeniach zameriaval prevažne na paralelné výpočty v systémoch s distribuovanou pamäťou a škálovanie na tisícky uzlov, ktoré sú dostupné na dnešných superpočítačoch. Podstatná funkcia, ktorá predošlo spomenutým riešeniam chýba, je možnosť efektívnych výpočtov na grafických akcelerátoroch.

Dátová štruktúra implementovaná v knižnici TNL [11], sa práve zameriava na efektívnu reprezentáciu určitej triedy neštruktúrovaných sietí pre GPU, ale aj tiež pre klasické CPU. Siete, ktoré knižnica TNL implementuje, patria do triedy neštruktúrovaných konformných homogénnych sietí, ktoré sú dostatočne obecné pre mnoho aplikácií, pričom medzi podporované siete patria 2D štvoruholníkové, 3D šesťstenné a ľubovoľné n-dimenzionálne simplex siete. Implementácia používa štandard C++14 spolu s knižnicou CUDA [12] pre využitie GPU. Medzi najviac významné vlastnosti patria vysoká statická konfigurovateľnosť pomocou C++ šablón a efektívna interná pamäťová štruktúra, ktorá využíva moderné maticové formáty [13, 1]. Použitie statickej konfigurácie zabraňuje ukladaniu nepotrebných dynamických dát, čo minimalizuje veľkosť dátovej štruktúry v pamäti.

Ciele a požiadavky práce

Účelom práce je navrhnúť a implementovať zovšeobecnenie dátovej štruktúry pre neštruktúrované siete v knižnici TNL o rozšírenie podpory pre mnohouholníkové a mnohostenné topológie.

Požiadavkom je overiť správnosť implementácie pomocou unit testov pre vybrané malé siete. V poslednej rade je požadované skúmať efektivitu implementácie na rôznych reálnych mnohouholníkových a mnohostenných sieťach a vykonať porovnanie s pôvodnou implementáciou.

Organizácia práce

Kapitola 1 slúži pre vysvetlenie teórie a terminológie, na ktorú sa bude odkazovať v nasledujúcich kapitolách.

Kapitola 2 predstavuje, akým spôsobom je pôvodná dátová štruktúra pre reprezentáciu neštruktúrovaných sietí v TNL navrhnutá a implementovaná.

Kapitola 3 sa zaoberá samotným rozšírením dátovej štruktúry z kapitoly 2 o podporu mnohouholníkových, mnohostenných a dodatočne ihlanových a klinových sietí.

Kapitola 4 popisuje testovacie úlohy, ktoré sú v kapitole 5 použité pre experimentálnu analýzu výkonu implementovaného rozšírenia dátovej štruktúry z kapitoly 3. Sú vysvetlené algoritmy testovacích úloh a tiež detaily ich implementácie.

Na záver, kapitola 5 experimentálne skúma výkon/efektivitu implementovaného rozšírenia dátovej štruktúry na základe testovacích úloh z kapitoly 4 a vykonáva porovnanie s pôvodnou implementáciou z kapitoly 2.

Kapitola 1

Teoretická časť

Táto kapitola sa zaoberá vysvetlením základných pojmov a konceptov, ktoré súvisia s témou tejto práce. Najprv sú vysvetlené základné pojmy z oblasti sietí. Potom je predstavený krátky úvod do programovania grafických akcelerátorov, na ktorý sa bude hlavne naväzovať pri experimentálnom meraní efektivity implementovanej dátovej štruktúry. Sú tiež zhrnuté dôležité formáty pre reprezentáciu riedkych matíc z dôvodu, že riedke matice sú jedným z najpodstatnejších dátových štruktúr pre reprezentáciu neštruktúrovaných sietí. Na záver sú popísané dôležité koncepty knižnice TNL [11] a tiež sú zhrnuté podstatné dátové štruktúry a algoritmy, ktoré sú v TNL implementované a sú dôležité z pohľadu implementácie dátovej štruktúry pre neštruktúrované siete a tiež pre implementáciu testovacích úloh pre experimentálnu analýzu efektivity implementovanej dátovej štruktúry. Je tiež potrebné upresniť, že poskytnuté definície v tejto kapitole nemusia byť úplne matematicky korektné a slúžia hlavne len na vysvetlenie významu daných pojmov v rámci tejto práce.

1.1 Základné pojmy

Definícia 1.1 (sieť)**.** *Sieť* je kolekcia geometrických objektov s jednoduchým tvarom, usporiadaných tak, aby pokryli určitú oblasť v priestore. Objekty tvoriace sieť sa nazývajú *entity siete*. Sieť má určitú dimenziu $D > 0$, ktorá je daná najvyššou dimenziou jej entít.

Definícia 1.2 (topológia entity)**.** *Topológia* definuje aký geometrický tvar má daná D-dimenzionálna entita siete a tiež aké tvary majú entity nižšej dimenzie, z ktorých je daná entita zhotovená.

Napríklad trojuholník je entita dimenzie 2, ktorý sa skladá z 3 hrán dimenzie 1 a z 3 vrcholov dimenzie 0. V prípade D-dimenzionálnej siete sa tiež entitám určitých dimenzií prideľujú špecifické názvy. D-dimenzionálne entity sa nazývajú *bunky*, $(D-1)$-dimenzionálne entity sú *steny*, 1-dimenzionálne entity sú *hrany* a 0-dimenzionálne entity sú vrcholy.

Definícia 1.3 (incidencia entít)**.** Vzťahy medzi entitami rôznych dimenzií je možné reprezentovať reláciou incidencie. V prípade, že d_1-dimenzionálna entita e_1 je súčasťou d_2-dimenzionálnej entity e_2, kde $d_1 < d_2$, sa dá povedať, že e_1 je *pod-entitou* e_2 a naopak e_2 je *nad-entitou* e_1. V oboch situáciách sú e_1 a e_2 navzájom incidentné.

Definícia 1.4 (incidentná matica)**.** Majme dve množiny $E_{d_1} = \{e_1, \ldots, e_m\}$ a $F_{d_2} = \{f_1, \ldots, f_n\}$, ktoré obsahujú d_1-dimenzionálne a d_2-dimenzionálne entity siete, pričom $d_1 \neq d_2$. Reláciu incidencie pre všetky entity z množiny E_{d_1} s entitami z množiny E_{d_2} je možné vyjadriť pomocou *matice incidencie*. Matica incidencie I_{d_1,d_2} je binárnou maticou, pre ktorú platí, že $[I_{d_1,d_2}]_{i,j} = 1$ práve vtedy, keď je entita e_i incidentná s entitou f_j. Naopak sa podobne dá definovať matice incidencie I_{d_2,d_1}, pre ktorú platí, že $I_{d_2,d_1} = I^T_{d_1,d_2}$.

Definícia 1.5 (konformná sieť)**.** Majme sieť M. Ak pre všetky entity siete $e_1, e_2 \in M$ platí, že prienik ich uzáverov $\overline{e_1} \cap \overline{e_2}$ je buď prázdna množina, alebo entita siete, tak platí že M je konformná sieť.

Definícia 1.6 (neštruktúrovaná sieť)**.** Sieť sa nazýva neštruktúrovaná ak platí, že každý vrchol siete je pod-entita nekonštantného počtu buniek siete.

Definícia 1.6 implikuje, že počet nad-entít entity v neštruktúrovanej sieti nie je konštantný. To je v kontraste s počtom pod-entít entity siete, ktorý závisí len na tvaru danej entity a nie na susedných entitách. Počet pod-entít entity je obecne nekonštantný (ako napríklad pri mnohouholníkových alebo mnohostenných sieťach), ale môže byť aj konštantný (ako napríklad pri trojuholníkových alebo štvorstenných sieťach).

Definícia 1.7 (homogénna sieť)**.** V prípade, že pre každú bunku siete platí, že majú rovnaký tvar, tak sieť je *slabo homogénna*. Keď naviac platí, že všetky entity siete s rovnakou dimenziou majú rovnaký tvar, tak sieť je *silno homogénna*.

Je potrebné poznamenať, že slabo homogénna sieť nie je automaticky silno homogénna. To vyplýva napríklad z toho, že bunka tvaru ihlanu obsahuje steny dvoch rôznych tvarov: trojuholník a štvoruholník.

1.2 Programovanie grafických akcelerátorov

Grafický akcelerátor (GPU) je zariadenie, ktoré je v spolupráci s centrálnym procesorom (CPU) použité ku urýchleniu výpočtu, čo je umožnené pomocou využitia tzv. *masívneho paralelizmu*. Najväčším rozdielom medzi nimi je, že CPU je optimalizované pre efektívne vykonávaniu sekvenčného kódu, pričom jednotlivé jadrá sú veľmi zložité a je ich rádovo len desiatky. Na druhej strane, GPU je optimalizované pre vykonávanie paralelných aplikácií, pričom jadrá sú naopak veľmi jednoduché a rádovo dosahujú počty v tisíckach [12].

Aj napriek názvu, moderné grafické akcelerátory je možné použiť aj ku obecným výpočtom (GPGPU), nie len pre akceleráciu počítačovej grafiky. Najpoužívanejšou platformou pre GPGPU je CUDA [12] od firmy Nvidia. CUDA je určená špecificky len pre programovanie GPU, ktoré boli vyrobené firmou Nvidia. Pre programovanie GPU od firmy AMD môže byť použitá napríklad platforma AMD ROCm™ [14]. Tieto dve platformy sú si veľmi podobné, ale používajú rozdielnu terminológiu. Naďalej sa bude používať terminológia platformy CUDA.

GPU je vybavené svojou vlastnou pamäťou, ktorá sa nazýva *globálna pamäť*. Výhodou tejto pamäte je, že má podstatne väčšiu priepustnosť ako operačná pamäť, ktorú používa CPU. Na druhej strane, nevýhodou je, že CPU komunikuje s touto pamäťou pomocou zbernice PCI Express, ktorá je podstatne pomalšia. To znamená, že GPU je často nevhodné pre algoritmy, ktoré vyžadujú extenzívnu komunikáciu medzi CPU a GPU. Pre efektívne využitie priepustnosti globálnej pamäte je tiež potrebné, aby sa ku nej pristupovalo po veľkých súvislých blokoch. Prístupy do globálnej pamäte sú kešované podobným spôsobom, ako to funguje medzi hlavnom operačnou pamäťou a CPU. Rozdiel je v tom, že v prípade GPU je možné združiť prístupy do pamäte z viacerých susedných vlákien (po skupinách o veľkosti 32) do čo najmenšieho počtu riadkov vyrovnávacej pamäte. Tomuto sa hovorí *združený prístup do pamäte*. Z toho tiež vyplýva, že pre dosiahnute maximálnej priepustnosti, je potrebné, aby susedné vlákna pristupovali súčasne na susedné pozície dát v pamäti. Na druhej strane, keď sa s globálnou pamäťou komunikuje pomocou tzv. *náhodného prístupu*, je priepustnosť podstatne menšia a to kvôli tomu, že sa prístupy do pamäte nezdružujú, čo má negatívny vplyv na efektivitu výpočtu [11].

GPU pozostáva z niekoľkých nezávislých *multiprocesorov*, ktoré môžu komunikovať s globálnou pamäťou. Navyše, každý z nich má k dispozícii tzv. *zdieľanú pamäť*, ktorá je podstatne rýchlejšia ako globálna pamäť. Na druhej strane je veľmi mála, kvôli čomu je často použitá ako vyrovnávacia pamäť pre zníženie duplicitnej komunikácie s globálnou pamäťou. Každý multiprocesor vykonáva vlákna paralelne po blokoch o veľkosti 32. Takýto blok sa nazýva *warp*. Vlákna vo warpe sú vykonávané ako vlákna v SIMD architektúre. To znamená, že by sa pre každé vlákno mala vykonávať tá istá inštrukcia, a to aj napriek tomu, že v moderných GPU architektúrach je toto správanie o trochu voľnejšie. V prípade, že vlákna toto pravidlo porušujú, dochádza k tzv. *divergencii warpu*, čo spôsobuje serializáciu vlákien vo warpe. Dôsledkom tejto serializácie je zníženie efektivity výpočtu. Divergencia warpu môže nastať napríklad v prípade, že jednotlivé vlákna vo warpe vykonávajú rôzne množstvá práce. To spôsobí, že jadrá, ktoré vykonali vlákna s menším množstvom práce, budú nútené počkať na dokončenie vlákien s väčším počtom práce, čo znamená, že tieto jadrá budú po dobu tohoto čakania nečinné.

Základným stavebným kameňom CUDA programu je *kernel*. Kernel je funkcia, ktorej kód je vykonávaný paralelne na GPU špecifikovaným množstvom

vlákien. Vlákna vykonávajúce kernel sú navyše ešte rozdelené do potencionálne niekoľkých blokov, pričom každý blok je pridelený špecifickému multiprocesoru. Pre súčasné verzie CUDA je limit počtov vlákien na blok nastavený na 1024 vlákien. Keďže vlákna potrebujú ku svojmu behu určité obmedzené prostriedky multiprocesoru, ako napríklad registre a zdieľanú pamäť, môže byť v praxi skutočný limit počtu vlákien na blok menší ako 1024, čo závisí na použitom GPU a danom vykonávanom kerneli. Multiprocesor si ešte rozdelí vlákna bloku do skupín o veľkosti 32, ktoré sú vykonávané ako warpy, ako to už bolo predtým spomenuté [12].

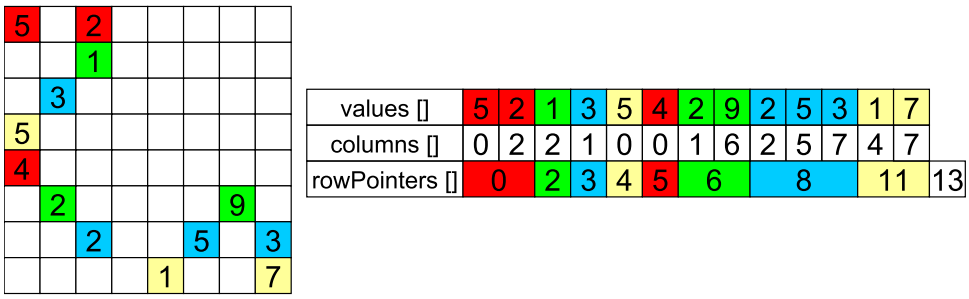
1.3 Reprezentácia matíc v pamäti

Najprirodzenejším a najjednoduchším spôsobom, ako je možné reprezentovať maticu s dimenziou $M \times N$ v pamäti, je pomocou uloženia každého prvku matice v 2D poli s dimenziou $M \times N$. Tento spôsob funguje celkom dobre v prípade, keď je matice považovaná za *hustú* (väčšina prvkov matice je nenulová). Na druhej strane, keď je matica *riedka* (väčšina prvkov matice je nulová), je tento spôsob veľmi neefektívny. Ne-efektivita spočíva v tom, že uloženie nulových prvkov v prípade riedkej matice je veľmi redundantné. Lepším spôsobom je sústrediť sa len na ukladanie nenulových prvkov, pričom implicitne považovať všetky ostatné prvky za nulové.

Pri ukladaní matice pomocou 2D poľa nie je potrebné ukladať žiadne režijne informácie naviac, keďže pozíciu ľubovoľného elementu matice v pamäti je možné triviálne zistiť len na základe indexu riadku a stĺpca vďaka pravidelnej štruktúre. Na druhej strane, pri ukladaní len nenulových prvkov, nemusí byť štruktúra prvkov v pamäti pravidelná ako v predošlom prípade, čo znamená, že je potrebné ukladať aj nejaké režijne informácie naviac. Existuje veľa spôsobov, akým je možné v pamäti reprezentovať riedke matice, pričom každý z nich má určité výhody a nevýhody. V rámci tejto práce sa pozrieme len na pár *formátov riedkych matíc*, ktoré majú určitú významnosť z pohľadu implementácie dátovej štruktúry pre neštruktúrované siete.

1.3.1 Compressed Sparse Row (CSR) formát

Compressed Sparse Row (CSR) formát [1] sa líši od spôsobu uloženia pomocou 2D poľa jednoducho tým, že vynechá všetky nulové prvky a tie nenulové uloží sekvenčne po riadkoch do 1D poľa (`values`). Pre každú nenulovú hodnotu je ešte potrebné si zapamätať jej stĺpcový index. Stĺpcové indexy sa typicky uložia do 1D poľa (`columns`), pričom pozícia indexu v tomto poli musí korešpondovať s pozíciou nenulového prvku v poli `values`. Ako posledné si formát tiež pamätá počiatočné indexy riadkov, ktoré sú uložené v 1D poli (`rowPointers`), pričom dĺžka poľa je o jeden index dlhšia ako počet riadkov matice.

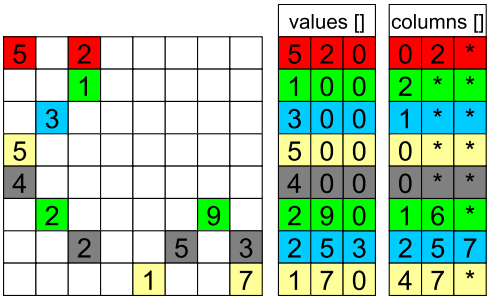


Obr. 1.1: CSR formát [1]

Výhodou tohoto formátu je, že je jednoduchý na implementáciu a dosahuje nízku spotrebu pamäte kvôli tomu, že do riadkov nepridáva žiadne nadbytočné nulové prvky súvisiace so zarovnávaním. Je tiež zaručené, že veľkosť kroku medzi elementami na tom istom riadku je vždy 1. Na druhej strane, nevýhodou tohoto formátu je to, že nie je vhodný pre výpočty na GPU, a to kvôli tomu, že sa nedodržiava združený prístup do globálnej pamäte [12]. To vyplýva z toho, že typicky sa 1 vlákno mapuje na 1 riadok matice, napríklad v prípade i-tého riadku s M nenulovými prvkami existuje medzi prvým prvkom i-tého a $(i+1)$-tého riadku $M-1$ ďalších prvkov. To znamená, že sa do pamäte pristupuje s krokom väčším ako 1.

1.3.2 Ellpack formát

Ellpack formát [1] sa od CSR formátu líši tým, že do každého skomprimovaného riadku sa pridajú dodatočné vyplňujúce nuly tak, aby každý riadok mal rovnakú veľkosť. To znamená, že pre maticu s N riadkami a s maximálnym počtom nenulových prvkov na riadok L, sa pre polia `values` a `columns` alokuje miesto pre $N \times L$ prvkov. Keďže teraz má každý riadok rovnakú veľkosť, nie je tiež potrebné ukladať pole `rowPointers`.



Obr. 1.2: Ellpack formát (riadková orientácia) [1]

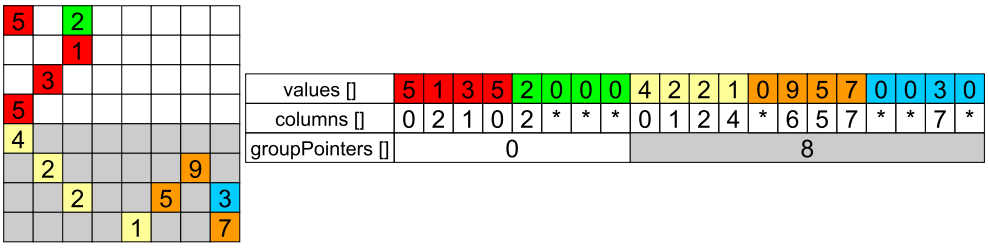
Tento formát je vhodný pre matice s riadkami majúce podobné počty ne-

nulových prvkov. Na druhej strane, je tento formát nevhodný napríklad pre matice, ktoré obsahujú riadok s podstatne viac nenulovými prvkami ako ostatné. V tomto prípade by bola réžia ukladanie vyplňujúcich nulových prvkov celkom podstatná.

Formát umožňuje ukladať skomprimované prvky nielen po riadkoch, ale aj po stĺpcoch. V prvom prípade by bola pri mapovaní 1 CUDA vlákna na jeden riadok, veľkosť kroku medzi prvkami na tom istom riadku rovná 1 a v druhom prípade by bola rovná počtu riadkov matice. V prípade orientácie prvkov po stĺpcoch, je tiež tento formát vhodný pre výpočty na GPU, keďže sa zachováva združený prístup do globálnej pamäte. To vyplýva z toho, že sú nenulové prvky s rovnakým poradím v riadku, uložené v pamäti sekvenčne vedľa seba.

1.3.3 Sliced Ellpack formát

Sliced Ellpack formát [13, 1] je bloková modifikácia Ellpack formátu. Riadky matice sú rozdelené do blokov o konštantnej veľkosti a skomprimované nenulové prvky sú uložené buď v riadkovej, alebo stĺpcovej orientácii. Riadky sú zarovnané na základe riadku s najväčším počtom nenulových prvkov, ktorý patrí do rovnakého bloku. To znamená, že tento formát rieši nedostatok Ellpack formátu tak, že keď má matica riadok s podstatne viac nenulovými prvkami ako ostatné, budú zarovnaním negatívne ovplyvnené len riadky z toho istého bloku a nie všetky riadky matice. Formát tiež potrebuje naviac uložiť pole s počiatočnými indexmi skupín riadkov (`groupPointers`).



Obr. 1.3: Sliced Ellpack formát (stĺpcová orientácia) [1]

Podobne ako Ellpack formát, v prípade stĺpcovej orientácie je tiež zachovaný združený prístup do pamäte, ale musí platiť, že veľkosť bloku je násobok veľkosti CUDA warpu, ktorý je pre súčasne architektúry rovný 32 [12]. V prípade, že by sa tento požiadavok nezachoval, by sa mohlo stať, že niektoré vlákna z warpu by mohli pristupovať na riadky, ktoré nepatria do toho istého bloku, čo by spôsobilo porušenie združeného prístupu do pamäte.

1.4 Template Numerical Library (TNL)

TNL [11] je knižnica, ktorej cieľom je uľahčenie vývoja efektívnych programov pre náročné počítačové simulácie. Pre splnenie tohoto cieľa, knižnica obsahuje podporu pre moderné paralelné architektúry, ako napríklad viacjadrové procesory, grafické akcelerátory a distribuované systémy. Hlavným programovacím jazykom použitým v TNL je C++, čo jej umožňuje využitie C++ šablón, šablónových špecializácií a šablónového meta-programovania pre vygenerovanie efektívneho kódu závislého na špecifickej architektúre bez žiadnej nadbytočnej réžie [11]. Jednými z hlavných dôvodov využitia C++ šablón je to, že je často možné sa vyhnúť virtuálnym metódam, ktoré sú menej efektívne [15] a tiež veľmi obmedzené v CUDA [12], ktorá sa v knižnici extenzívne používa. Tento prístup teda zvyšuje efektivitu výpočtu počas behu programu na úkor zvýšeného množstva práce vykonanej počas kompilácie programu a tiež potencionálne zvýšenej veľkosti skompilovaného binárneho súboru programu. Výhodou je tiež to, že je viac podporovaná detekcia programovacích chýb už počas kompilácie.

1.4.1 Koncept zariadení v TNL

Jedným zo základných konceptov TNL je zjednotenie rozhrania dátových štruktúr a (typicky paralelných) algoritmov pre rôzne pamäťové priestory a exekučné modely. Napríklad, dátová štruktúra môže byť alokovaná v hlavnej operačnej pamäti, v globálnej pamäti GPU alebo v *CUDA Unified Memory*, ku ktorej je možné pristúpiť zo strany CPU a GPU. Na druhej strane, algoritmy môžu byť vykonané buď pomocou CPU, alebo GPU. Špecifikovanie pamäťového priestoru a exekučného modelu je možné vykonať prostredníctvom šablónového parametru `Device`, pričom napríklad `TNL::Devices::Host` špecifikuje CPU a `TNL::Devices::Cuda` špecifikuje Nvidia GPU [11].

1.4.2 Dátové štruktúry v TNL

1.4.2.1 Polia a vektory

Jedným z najzákladnejších dátových štruktúr, ktoré TNL podporuje je pole, ktoré je reprezentované šablónovou triedou `TNL::Containers::Array`. Trieda poskytuje bežné rozhranie, ako napríklad metódu pre alokáciu (`setSize`), operátory pre porovnanie a priradenie alebo manipuláciu s elementami poľa pomocou metód `getElement`, `setElement` a `operator[]`. Výhodou prvých dvoch metód pre manipuláciu s elementami je to, že môžu byť volané zo strany CPU aj v prípade, že je pole alokované v pamäti GPU. Na druhej strane, pomocou poslednej metódy je možné pristupovať ku elementom len zo zariadenia, v ktorej pamäti je pole alokované [11].

TNL podporuje aj vektory, ktoré sú reprezentované šablónovou triedou `TNL::Containers::Vector`. Vektory rozširujú funkcionalitu polí o operácie z

lineárnej algebry, ako napríklad vektorový súčet, skalárny súčin alebo výpočty rôznych noriem. Kvôli tomu sú vektory tiež obmedzené len pre číselné dátové typy elementov [11].

Sú podporované tiež tzv. statické polia, pri ktorých je veľkosť daného poľa špecifikovaná ako konštanta pomocou šablónového argumentu, čo umožňuje vyhnutiu sa dynamickej alokácii. Takéto polia sú implementované pomocou šablónovej triedy `TNL::Containers::StaticArray`.

1.4.2.2 Matice

TNL tiež podporuje dátové štruktúry pre husté a riedke matice, ktorých implementáciu je možné nájsť v mennom priestore `TNL::Matrices`. Je podporovaných niekoľko formátov pre riedke matice, ako napríklad *tridiagonálne* alebo *multi-diagonálne* matice, *Ellpack* formát, *Sliced Ellpack* formát alebo *CSR* formát. Pri práci s maticou je potrebné najprv vykonať alokáciu pamäte pre riadky matice pomocou metódy `setCompressedRowLengths`, ktorá prijíma ako argument vektor celých čísiel, kde i-ty element vektoru špecifikuje počet nenulových elementov i-teho riadku matice. Potom, čo sú riadky alokované, je možné do matice pridávať jednotlivé elementy aj dokonca paralelne. Je potrebné ale zaručiť, aby počet pridaných elementov na danom riadku nepresahoval počet alokovaných elementov [11].

1.4.3 Využitie lambda funkcií v TNL

Lambda funkcie sú v TNL celkom extenzívne používané. Ich hlavnou výhodou je, že umožňujú špecifikovať rozhranie rôznych implementovaných algoritmov veľmi flexibilne z pohľadu užívateľa. TNL využíva toho faktu, že je možné lambda funkcie používať vo vnútri CUDA kernelov, čo umožňuje rozhranie definovať flexibilne tak, že užívateľ špecifikuje danú operáciu pomocou rovnakej lambdy funkcie pre rôzne zariadenia, a to ktoré zariadenie sa použije, je možné zmeniť len pomocou šablónového parametru `Device`. Pre použitie lambdy v kernele je ale ešte potrebné lambdu označiť pomocou špecifikátora `__device__`. Kvôli tomu TNL definovalo makro `__cuda_callable__` [11], ktoré zjednocuje špecifikátory `__host__` a `__device__`, čo umožňuje vykonanie lambdy funkcie nielen v klasickej CPU funkcii, ale aj v CUDA kernele.

1.4.4 Algoritmy v TNL

1.4.4.1 Paralelný for cyklus

Paralelným for cyklom sa myslí for cyklus, pri ktorom iterácie reprezentujú úlohy, ktoré sú paralelne spracované viacerými vláknami. Algoritmus je v TNL implementovaný pomocou šablónovej triedy `TNL::Algorithms::ParallelFor`. Pri použití algoritmu je potrebné špecifikovať rozsah indexov a lambda funkciu, ktorá definuje, čo sa má vykonať v i-tej iterácii. Je tiež potrebné zachovať

rozhranie lambda funkcie, ktoré vyžaduje, aby prvý parameter reprezentoval celé číslo, určujúce o akú iteráciu zo špecifikovaného rozsahu indexov sa jedná.

1.4.4.2 Statický for cyklus

Statickým for cyklom sa myslí for cyklus, pri ktorom je možné jednotlivé indexy iterácií použiť v konštantných výrazoch, ako napríklad v šablónových argumentoch. Tento algoritmus je v TNL implementovaný ako šablónová funkcia `TNL::Algorithms::staticFor`. Rozhranie algoritmu je podobné ako pri ParallelFor, len s tým rozdielom, že rozsah indexov je potrebné predať ako šablónové argumenty.

1.4.4.3 Paralelná redukcia

Paralelnou redukciou [16] sa myslí operácia, ktorá zredukuje určitý rozsah hodnôt na skalárnu hodnotu pomocou opakovaného použitia binárnej asociatívnej funkcie, ktorá je volaná na hodnoty z rozsahu. Tento algoritmus je implementovaný v TNL pomocou šablónovej funkcie `TNL::Algorithms::reduce`. Pri volaní tejto funkcie je potrebné špecifikovať rozsah indexov, ale tentokrát je potrebné tiež aj špecifikovať dve lambda funkcie. Úlohou prvej lambda funkcie `fetch` je na základe vstupného argumentu indexu zo špecifikovaného rozsahu vrátiť hodnotu na i-tej pozícii, ktorá sa bude redukovať. Účelom druhej lambda funkcie `reduction` je špecifikovanie samotnej redukčnej funkcie. Posledným požadovaným argumentom algoritmu je tiež hodnota neutrálneho prvku pre špecifikovanú binárnu operáciu.

1.4.4.4 Exkluzívna prefixová suma

Exkluzívnou prefixovou sumou [17] sa myslí operácia, pri ktorej je sekvencia $a_1, \ldots, a_n$ modifikovaná na sekvenciu $s_1, \ldots, s_n$, pričom $s_i = \sum_{j=1}^{i-1} a_j$ pre všetky $i > 1$ a $s_1 = 0$. Tento algoritmus je implementovaný v TNL pomocou šablónovej funkcie `TNL::Algorithms::inplaceExclusiveScan`. Funkcia má podobné rozhranie ako `TNL::Algorithms::reduce`, len s tým rozdielom, že nie je možné špecifikovať `fetch` lambda funkciu. Namiesto toho sa ako argument predá samotné pole hodnôt, pre ktoré sa má operácia vykonať. Toto pole je tiež následne funkciou modifikované, čo vyplýva z jej názvu (inplace).

1.4.5 TNL View štruktúry

Pre každú dátovú štruktúru v TNL existuje tzv. `view` štruktúra. Napríklad pre `TNL::Containers::Array` existuje `TNL::Containers::ArrayView`. Účelom týchto štruktúr je umožnenie vykonania plytkej kópie danej dátovej štruktúry. To znamená, že poskytujú určitú alternatívu referencií na dátovú štruktúru. Dôvod, prečo sú tieto štruktúry potrebné, je, že pre lambda funkcie, ktoré sú použité v CUDA kerneloch, je povolené zachytiť objekty len hodnotou. To

vyplýva z toho, že pre kód bežiaci na GPU sú referencie na objekty uložené v hlavnej operačnej pamäti neplatné, kvôli tomu, že kód na GPU pracuje vo svojom vlastnom pamäťovom priestore. Je síce možné hodnotou tiež zachytiť samotnú dátovú štruktúru (ako napríklad `Array`), ale vykonala by sa v tomto prípade menej efektívna hlboká kópia, ktorá by tiež bola vo väčšine prípadoch nežiadúca, keďže by nedovoľovala modifikáciu pôvodnej dátovej štruktúry.

Kapitola 2

Dátová štruktúra pre neštruktúrované siete

Táto kapitola sa zaoberá popisom dátovej štruktúry pre reprezentáciu konformnej neštruktúrovanej homogénnej siete, implementovanej v TNL [11]. Kapitola prezentuje požiadavky a realizáciu dátovej štruktúry spolu aj s vysvetlením ako jednotlivé časti dátovej štruktúry fungujú a akú majú funkciu. Je tiež popísaná samotná implementácia dátovej štruktúry z pohľadu triednej štruktúry a programového rozhrania, ktoré dátová štruktúra poskytuje pre užívateľa. Popis je kvôli prehľadnosti v zjednodušenej podobe, ale implementáciu je možné detailnejšie preskúmať pomocou dokumentácie TNL [18].

Hlavným účelom dátovej štruktúry pre reprezentáciu konformnej neštruktúrovanej homogénnej siete je jej použitie v pokročilých numerických metódach v HPC aplikáciach, čo znamená, že sa kladie veľký doraz na efektivitu. Bohužiaľ, nie je prakticky možné navrhnúť jedinú efektívnu reprezentáciu neštruktúrovaných sietí, ktorá by bola vhodná pre všetky možné situácie. To znamená, že implementovaná dátová štruktúra bola navrhnutá s určitými kompromismi pre dosiahnutie pôvodných návrhových cieľov.

2.1 Reprezentácia štruktúry siete

Reprezentácia konformnej neštruktúrovanej homogénnej siete je založená na základe buniek a vrcholov. Vrcholy sú jediné entity siete, ktoré si držia svoje priestorové súradnice, pričom priestorové súradnice buniek a ostatných $(D > 0)$-dimenzionálnych entít sú realizované pomocou ich incidencie s danými vrcholmi (vrcholy sú pod-entity všetkých ostatných entít siete). Výhodou tohoto spôsobu reprezentácie je to, že to dovoľuje prácu s tzv. *plávajúcimi sieťami*, pri ktorých sa priestorové súradnice môžu meniť počas výpočtu. Bunky sú veľmi dôležité pre reprezentáciu konformných homogénnych sietí, pretože všetky ostatné entity s nižšou dimenziou a ich vzájomné incidencie

môžu byť odvodené z informáciách o bunkách a ich vrcholoch (konkrétne z incidencií vrcholov a buniek). Kvôli tomu je možné inicializovať celú sieť zo vstupu, ktorý pozostáva zo zoznamu priestorových súradnici a mapovania buniek a ich incidentných vrcholov (pre každú bunku existuje list indexov vrcholov). Mapovaniu medzi bunkou a jej incidentnými vrcholmi sa v kontexte inicializácie siete bude naďalej hovoriť *zárodok bunky* (cell seed). Inicializácia siete bude popísaná v sekcii 2.2.6.

Samotná dátová štruktúra pozostáva interne z polí a riedkych matíc, ktoré sú vyjadrené pomocou príslušného efektívneho formátu. Výhodou použitia polí a vhodných formátov riedkych matíc je to, že to umožňuje primerané využitie priepustnosti pamäte pomocou efektívneho využitia vyrovnávacích pamätí nielen na CPU, ale aj na GPU (pomocou združeného prístupu do pamäte). Na druhej strane, nevýhodou je to, že vykonávanie nových zmien do už vytvorenej reprezentácie siete by bolo veľmi zložité a neefektívne kvôli tomu, že by to vyžadovalo realokáciu polí a matíc. Kvôli tomu nie je táto funkcia podporovaná, čo znamená, že už pri inicializácii musia byť poskytnuté všetky informácie o vnútorných prepojeniach siete.

2.1.1 Reprezentácia priestorových súradníc

Priestorové súradnice sú jednými s najdôležitejších údajov, ktoré musí dátová štruktúra reprezentovať. Ako už bolo spomenuté, priestorové súradnice sú viazané len na 0-dimenzionálne entity siete (vrcholy) a tak sa globálne vyskytujú len na jednom mieste v pamäti. Každá súradnica pozostáva z niekoľkých položiek, ktorých počet závisí na priestorovej dimenzii celej siete, ktorá sa naprieč celej implementácii označuje ako `SpaceDimension`. Je dôležité ju odlíšiť od `MeshDimension` (dimenzia siete), ktorá označuje dimenziu topológie bunky siete. Takéto rozlíšenie dimenzií dovoľuje napríklad reprezentovať trojuholníkovú sieť nielen v 2D, ale aj v 3D priestore a podobne. Je samozrejmé, že `SpaceDimension` je limitovaná len na dimenzie, ktoré sú aspoň také ako `MeshDimension`.

Jednotlivé priestorové súradnice sú reprezentované v pamäti pomocou poľa o dĺžke `SpaceDimension`. Keďže `SpaceDimension` je konštanta, je možné použiť statické pole `TNL::Containers::StaticArray` a vyhnúť sa tak dynamickej alokácii. Polia jednotlivých priestorových súradníc sú potom organizované do jedného veľkého poľa, ktoré ukladá všetky súradnice naprieč celou sieťou. Toto pole samozrejme nedáva zmysel deklarovať ako statické, keďže počet vrcholov v sieti môže byť často veľmi rozdielny. Kvôli tomu je pre toto pole použitá dátová štruktúra `TNL::Containers::Array`.

Na Obrázku 2.1 je možné vidieť príklad pamäťovej reprezentácie 3D vrcholov o počte n pomocou vyššie popísanej realizácie poliami, pričom $(p_i)_j$ značí j-tu priestorovú súradnicu i-teho vrcholu.

Obr. 2.1: Reprezentácia priestorových súradníc v pamäti

2.1.2 Reprezentácia pod-/nad-entít

Ďalším dôležitým údajom, ktoré dátová štruktúra musí reprezentovať je to, akú vzájomnú incidenciu majú entity siete rôznych dimenzií. To znamená, že pre každú entitu, je potrebné mať znalosť, aké pod-entity (entity s nižšou dimenziou) a nad-entity (entity s vyššou dimenziou) sú incidentné s danou entitou siete. Táto znalosť incidencií nám tiež napríklad umožní uložiť informáciu o tom, z akých priestorových súradníc sa skladajú entity vyšších dimenzií ako napríklad bunky. Ako bolo naznačené v sekcii 1.1, jednotlivé incidencie entít siete s dimenziami d_1, d_2 je možné vyjadriť pomocou incidentnej (binárnej) matice I_{d_1,d_2}. V prípade, že $d_1 > d_2$, sa matici hovorí matici pod-entít. Na druhej strane, keď platí $d_1 < d_2$, sa matici hovorí matici nad-entít.

Každá entita má svoj *globálny index*, ktorý je jednoznačný v danej sieti pre určitú dimenziu entít. Keďže je dátová štruktúra implementovaná v C++, indexy začínajú od hodnoty 0. Index príslušnej entity potom prirodzene odpovedá pozícii prvku v nejakom poli, pozícii riadku v matici, a podobne. Ako napríklad, keby sme chceli zistiť d_2-dimenzionálne incidentné entity danej d_1-dimenzionálnej entity s indexom i, tak by sme museli prečítať i-ty riadok incidentnej matice I_{d_1,d_2}, pričom globálne indexy daných incidetných entít by odpovedali indexom stĺpcov tých elementov riadku, ktoré majú hodnotu 1.

Pre adresovanie jednotlivých d_2-dimenzionálnych pod-/nad-entít danej entity sa definujú tzv. *lokálne indexy*. Teda lokálny index s hodnotou j odpovedá j-tej d_2-dimenzionálnej pod-/nad-entite danej entity. Ako napríklad, keby sme chceli zistiť j-tu d_2-dimenzionálnu incidentnú entitu danej d_1-dimenzionálnej entity s indexom i, tak by sme museli prečítať index stĺpca j-teho jednotkového elementu i-teho riadku incidentnej matice I_{d_1,d_2}.

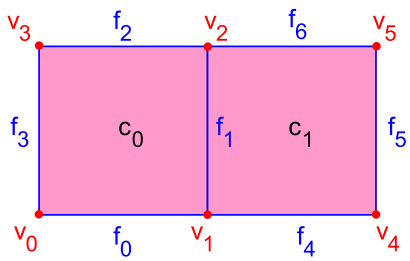
V predošlom texte bolo spomenuté, že incidencia buniek a vrcholov poskytuje prostriedky na odvodenie incidencií medzi ostatnými entitami siete. To hlavne vyplýva z toho, že sa táto dátová štruktúra zameriava na silno homogénne siete, pri ktorých je tvar entít rovnakej dimenzie vždy rovnaký. Dôsledkom tejto vlastnosti je to, že topológia bunky siete implicitne deklaruje topológiu všetkých entít nižších dimenzií. Napríklad keď máme sieť štvorstenov, tak vieme, že 2-dimenzionálne entity sú trojuholníky, 1-dimenzionálne entity sú hrany a 0-dimenzionálne entity sú vrcholy. Z tejto vlastnosti vyplýva aj to, že počet pod-entít danej entity je vždy konštantný a táto hodnota je tiež teda známa v čase kompilácie. To znamená, že matice pod-entít majú konštantný

počet nenulových prvkov naprieč všetkým riadkom. Na druhej strane, to isté sa o matici nad-entít povedať nedá, pretože je sieť tiež neštruktúrovaná.

Majme D-dimenzionálnu sieť, pre každý zoradený pár dimenzií (d_1, d_2), kde $d_1, d_2 \in \{0, 1, \ldots, D\}$ a $d_1 \neq d_2$, môže existovať len jedna incidetná matica, ktorá vyjadruje vzťahy incidencie medzi entitami dimenzií d_1 a d_2. Reprezentácia siete môže teda obsahovať celkovo až $2(D^2 - D + 1)$ matíc pod-/nad-entít. Nie vždy sú pre výpočet potrebné všetky tieto matice, kvôli čomu, je z pohľadu efektivity potrebné definovať, ktoré incidentné matice budú uložené. Táto záležitosť spolu aj s nastavením ostatných parametrov je riešená pomocou *konfigurácie siete*, o ktorej sa bude hovoriť v sekcii 2.2.2.

Keďže incidentné matice sú vo väčšine prípadov riedke, sú pre ich uloženie v pamäti použité určité formáty riedkych matíc, ktoré boli vysvetlené v sekcii 1.3. Keďže sú matice tiež aj binárne, dajú sa použité formáty riedkych matíc modifikovať tak, že hodnoty matice (0 a 1) nie sú v skutočnosti ani uložené, pretože všetky prítomné hodnoty elementov matice sú len jedničky. Stĺpcové indexy v matici tiež nie sú zoradené podľa veľkosti. Vďaka tomu je možné k incidentným entitám pristupovať pomocou lokálnych indexov pod-/nad-entít. Najväčším požiadavkom na formáty riedkych matíc z pohľadu tejto dátovej štruktúry, sú združený prístup do pamäte na GPU a konštantná veľkosť kroku medzi elementami toho istého riadku. Kvôli tomu je pre maticu pod-entít použitý formát Ellpack a pre matice nad-entít použitý formát Sliced Ellpack. V prípade matice pod-entít, je dĺžka všetkých riadkov konštantná, kvôli čomu je zarovnanie na základe najdlhšieho riadku matice irelevantné a tak je v tomto prípade Ellpack formát vhodnejší ako Sliced Ellpack, ktorý ukladá v tomto prípade nadbytočné režijné informácie. Na druhej strane, v prípade matíc nad-entít, nie je zaručené, že všetky riadky budú mať rovnakú dĺžku. Kvôli tomu je použitý formát Sliced Ellpack, aby sa vyhlo katastrofickému prípadu zarovnávania, kedy by niektoré entity mali podstatne viac nad-entít ako ostatné. Pri výpočtoch je tiež potrebné mať informáciu o počte pod-/nad-entít danej entity. Spomenuté formáty explicitne počty nenulových elementov neukladajú, kvôli čomu je potrebné, aby tieto počty boli naviac uložené v poli popri danej incidentnej matici. Keďže v prípade matíc pod-entít, je počet vždy konštantný vďaka (silnej) homogénnosti, je potrebné tieto počty ukladať len popri matíc nad-entít. V prípade matíc pod-entít je počet nenulových elementov poskytnutý pomocou konštanty známej už počas prekladu.

Obrázok 2.3 znázorňuje príklad reprezentácie štvoruholníkovej siete o dvomi bunkami, pri ktorom je možné vidieť všetky možné incidentné matice $I_{i,j}$, pričom pri maticiach nad-entít je tiež možné vidieť dodatočné polia s počtami nenulových prvkov riadkov, kde pole $C_{k,l}$ prislúcha matici nad-entít $I_{k,l}$.



Obr. 2.2: Príklad štvoruholníkovej siete s bunkami c_0, c_1, stenami f_0, f_1, f_2, f_3, f_4, f_5, f_6 a vrcholmi v_0, v_1, v_2, v_3, v_4, v_5

$I_{0,1}$ =

	f_0	f_1	f_2	f_3	f_4	f_5	f_6
v_0	1			1			
v_1	1	1			1		
v_2		1	1				1
v_3			1	1			
v_4					1	1	
v_5						1	1

$I_{1,0}$ =

	v_0	v_1	v_2	v_3	v_4	v_5
f_0	1	1				
f_1		1	1			
f_2			1	1		
f_3	1			1		
f_4		1			1	
f_5					1	1
f_6			1			1

$C_{0,1}$ =

v_0	v_1	v_2	v_3	v_4	v_5
2	3	3	2	2	2

$I_{0,2}$ =

	c_0	c_1
v_0	1	
v_1	1	1
v_2	1	1
v_3	1	
v_4		1
v_5		1

$I_{1,2}$ =

	c_0	c_1
f_0	1	
f_1	1	1
f_2	1	
f_3	1	
f_4		1
f_5		1
f_6		1

$I_{2,0}$ =

	v_0	v_1	v_2	v_3	v_4	v_5
c_0	1	1	1	1		
c_1		1	1		1	1

$I_{2,1}$ =

	f_0	f_1	f_2	f_3	f_4	f_5	f_6
c_0	1	1	1	1			
c_1		1			1	1	1

$C_{0,2}$ =

v_0	v_1	v_2	v_3	v_4	v_5
1	2	2	1	1	1

$C_{1,2}$ =

f_0	f_1	f_2	f_3	f_4	f_5	f_6
1	2	1	1	1	1	1

Obr. 2.3: Incidentné matice pre sieť z obrázku 2.2

2.1.3 Susedstvo bunky

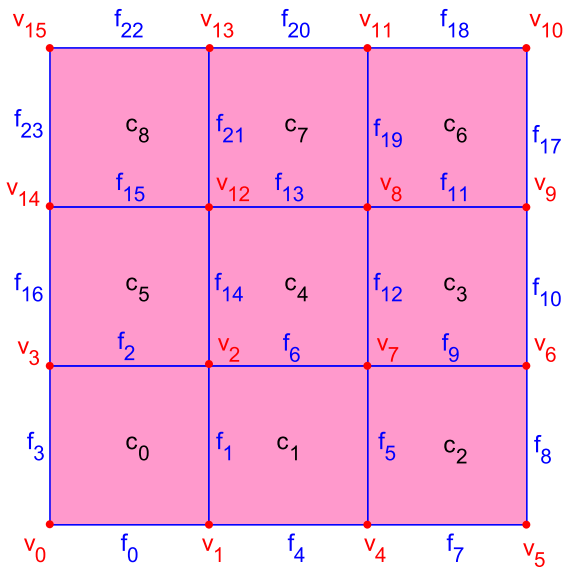
Ďalším údajom, ktorý dátová štruktúra reprezentuje je informácia o tom, aké bunky sú susedné s danou bunkou siete. Všetky bunky, ktoré sú s danou bunkou siete susedné tvoria tzv. *susedstvo bunky*. Bunka je susedná s inou bunkou vtedy, keď spolu zdieľajú určitý počet spoločných vrcholov. Tento

počet je typicky rovný dimenzii siete, čo je tiež aj jeho maximálna možná hodnota, ale je možné si počet zvoliť aj pomocou konštanty v konfigurácii siete označenej ako `dualGraphMinCommonVertices`. Napríklad v prípade 3-dimenzionálnej štvorstennej siete, pre hodnotu 1 musia mať susedné bunky spoločný aspoň jeden vrchol, pre hodnotu 2 musia mať spoločnú aspoň jednu hranu a pre hodnotu 3 musia mať spoločnú celú stenu.

Susedstvo buniek siete môže byť reprezentované pomocou duálneho grafu, ktorý je jednoznačne určený svojou *adjacenčnou maticou*. Adjacenčná matica duálneho grafu siete A je štvorcová binárna matica, kde počet riadkov a stĺpcov odpovedá počtu buniek siete a platí, že $[A]_{i,j} = 1$ práve vtedy, keď bunky siete s indexmi i a j spolu susedia. Keďže adjacenčná matica duálneho grafu siete je typicky riedka a počty nenulových prvkov riadkov sú si obecne navzájom rôzne, je možné ju efektívne reprezentovať formátom Sliced Ellpack spolu s poľom počtov nenulových prvkov riadkov matice, podobne ako tomu bolo pri reprezentácií matíc nad-entít v sekcii 2.1.2. Dá sa teda tiež povedať, že duálny graf siete reprezentuje incidencie medzi entitami s rovnakou dimenziou D (bunkami) a teda je adjacenčnú maticu duálneho grafu možno tiež označiť ako $I_{D,D}$, podobne ako incidentné matice I_{d_1,d_2}.

Informácia o susedstve bunky nie je dôležitá pre všetky výpočty a tak je možné tento údaj zo štruktúry siete vynechať pomocou jeho zákazu v konfigurácii siete, podobne ako tomu bolo v sekcii 2.1.2.

Obrázok 2.5 znázorňuje príklad adjacenčnej matice duálneho grafu $I_{2,2}$ s poľom počtov nenulových prvkov riadkov matice $C_{2,2}$ pre štvoruholníkovú sieť z obrázku 2.4.



Obr. 2.4: Príklad štvoruholníkovej siete s bunkami $c_0, \ldots, c_8$, stenami $f_0, \ldots, f_{23}$ a vrcholmi $v_0, \ldots, v_{15}$

$I_{2,2}$ =

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
c_0		1				1			
c_1	1		1		1				
c_2		1		1					
c_3			1		1		1		
c_4		1		1		1		1	
c_5	1				1				1
c_6				1				1	
c_7					1		1		1
c_8						1		1	

$C_{2,2}$ =

c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
2	3	2	3	4	3	2	3	2

Obr. 2.5: Ajacenčná matica duálneho grafu pre sieť z obrázku 2.4

2.1.4 Značky entít

Posledným údajom, ktorý dátová štruktúra obsahuje sú tzv. značky entít. Značka entity určitej dimenzie reprezentuje binárnu informáciu o tom, či daná entita splňuje špecifikovanú vlastnosť alebo nie. Z pohľadu neštruktúrovaných sietí sú relevantné tieto podporované dve vlastnosti: hraničnosť a vnútornosť entity. Keďže sú tieto dva vlastnosti navzájom výlučné, sú reprezentované jednou binárnou značkou *boundary tag*. Hodnota 1 indikuje, že je entita hraničná a naopak hodnota 0 indikuje, že je entita vnútorná.

To či sú entity hraničné alebo vnútorné je možné zistiť na základe entít dimenzie $D - 1$ (stien): keď je stena incidentná s 2 bunkami, jedná sa o vnútornú stenu a keď je incidentná len s 1 bunkou, tak sa jedná o hraničnú stenu. Bunky sú hraničné, keď sú incidentné s aspoň jednou hraničnou stenou. Ostatné entity sú hraničné, keď sú pod-entitou nejakej hraničnej steny.

Pre každú entitu môžu byť všetky značky reprezentované bitovým poľom. Keďže je momentálny počet podporovaných značiek menší ako 8, je bitové pole reprezentované pomocou jedného bytu, pričom ostatné bity sú nevyužité, ale na druhej strane to dovoľuje priestor pre pridanie nových značiek v budúcnosti. Značky všetkých entít sú teda jednoducho reprezentované ako pole všetkých bitových polí, čiže pole bytov.

Pri výpočte môže byť tiež potrebné vykonávať iteráciu nad všetkými entitami, ktoré splňujú určitú vlastnosť. Kvôli tomu je pre každú vlastnosť držané pole indexov entít (v prípade neštruktúrovaných sietí, existuje jedno pole pre vnútorné a druhé pole pre hraničné entity), ktoré danú vlastnosť splňujú. Samozrejme je možné tieto entity zistiť aj prostredníctvom poľa bitových polí z predošlého paragrafu, ale vyžadovalo by to iteráciu naprieč všetkým entitám siete danej dimenzie (aj tými, pre ktoré je vlastnosť nesplnená), čo by naviac muselo byť nasledované kontrolou príslušného bitu v bitovou poli. Kvôli tomu

je výsledok tejto operácie vopred vypočítaný a uložený v podobe poľa indexov entít, ktoré splňujú príslušnú vlastnosť. To sa hlavne hodí v prípade, keď je iteráciu nad entitami s danou vlastnosťou potrebné vykonať viackrát.

Obrázok 2.6 znázorňuje značky entít pre bunky (T_c), steny (T_f) a vrcholy (T_v), pričom sú znázornené aj polia entít splňujúce určitú vlastnosť, kde $(T_c)_b$, $(T_f)_b$ a $(T_v)_b$ reprezentujú hraničné entity a $(T_c)_i$, $(T_f)_i$ a $(T_v)_i$ reprezentujú vnútorné entity.

T_c =

c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
1	1	1	1	0	1	1	1	1

$(T_c)_b$ =

c_0	c_1	c_2	c_3	c_5	c_6	c_7	c_8

$(T_c)_i$ =

c_4

T_f =

f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}	f_{19}	f_{20}	f_{21}	f_{22}	f_{23}
1	0	0	1	1	0	0	1	1	0	1	0	0	0	0	0	1	1	1	0	1	0	1	1

$(T_f)_b$ =

f_0	f_3	f_4	f_7	f_8	f_{10}	f_{16}	f_{17}	f_{18}	f_{20}	f_{22}	f_{23}

$(T_f)_i$ =

f_1	f_2	f_5	f_6	f_9	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{19}	f_{21}

T_v =

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
1	1	0	1	1	1	1	0	0	1	1	1	0	1	1	1

$(T_v)_b$ =

v_0	v_1	v_3	v_4	v_5	v_6	v_9	v_{10}	v_{11}	v_{13}	v_{14}	v_{15}

$(T_v)_i$ =

v_2	v_7	v_8	v_{12}

Obr. 2.6: Reprezentácia značiek entít pre sieť z obrázku 2.4

2.2 Implementačné detaily

2.2.1 Rozhranie

Rozhranie dátovej štruktúry je vystavené hlavnou šablónovou triedou `Mesh`:

```
1 template< typename Config ,
2           typename Device = TNL::Devices::Host >
3 class Mesh;
```

`Mesh` závisí na dvoch šablónových parametroch: `Config` a `Device`. Parameter `Config` reprezentuje statickú konfiguráciu siete (viď sekcia 2.2.2) a `Device` reprezentuje výpočtovú platformu (viď sekcia 1.4.1).

Hlavným účelom triedy `Mesh` je poskytnutie prístupu ku jednotlivým entitám siete. Prvou dôležitou metódou je `getEntitiesCount`, ktorá poskytuje počet entít dimenzie `Dimension`:

```
template< int Dimension >
__cuda_callable__
GlobalIndexType getEntitiesCount() const;
```

Pre prístup ku samotnej entite dimenzie `Dimension` a indexu `entityIndex` existuje metóda `getEntity`:

```
template< int Dimension >
__cuda_callable__
EntityType< Dimension >&
getEntity( const GlobalIndexType& entityIndex );
```

Typ `GlobalIndexType` je alias korešpondujúci ku typu globálnych indexov, ktorý je konfigurovaný ako súčasť šablónového argumentu `Config` (viď sekciu 2.2.2). Podobne, typ `EntityType` je alias korešpondujúci ku typu entity dimenzie `Dimension`.

Entita siete je reprezentovaná šablónovou triedou `MeshEntity`, pričom prvé dva šablónové parametre odpovedajú parametrom triedy `Mesh` a posledný parameter `EntityTopology` reprezentuje typ topológie entity (viď sekcia 2.2.4):

```
template< typename Config,
          typename Device,
          typename EntityTopology >
class MeshEntity;
```

Objekt typu `MeshEntity` je tzv. *proxy objektom*, ktorý si interne drží ukazovateľ na objekt siete a tiež globálny index entity, čo umožňuje extrahovať dáta relevantné ku danej entite siete elegantným spôsobom.

Trieda `MeshEntity` je špecializovaná pre entity topológie `Vertex` (vrcholy), keďže vrcholy sa líšia od ostatných entít tým, že si tiež držia priestorové súradnice. Pre prístup ku súradniciam existujú metódy `getPoint` a `setPoint` (tieto dve metódy nie sú súčasťou obecnej implementácie `MeshEntity`):

```
__cuda_callable__
PointType getPoint() const;

__cuda_callable__
void setPoint( const PointType& point );
```

Typ `PointType` reprezentuje alias pre typ jednej priestorovej súradnice pozostávajúcej s niekoľko položiek (súradníc) (viď sekcia 2.1.1).

Ku informáciám o pod-entitách dimenzie `Subdimension` je možné pristúpiť pomocou metód `getSubentitiesCount` a `getSubentityIndex`:

```
template< int Subdimension >
static constexpr LocalIndexType getSubentitiesCount();

template< int Subdimension >
__cuda_callable__
GlobalIndexType
getSubentityIndex( const LocalIndexType localIndex ) const;
```

Podobne ako `GlobalIndexType`, `LocalIndexType` je alias pre typ lokálnych indexov, ktorý je súčasťou šablónového argumentu `Config` (viď sekciu 2.2.2).

Metódy `getSuperentitiesCount` a `getSuperentityIndex` slúžia pre poskytnutie informácii o nad-entitách dimenzie `Superdimension`:

```
1 template< int Superdimension >
2 __cuda_callable__
3 LocalIndexType getSuperentitiesCount() const;
4
5 template< int Superdimension >
6 __cuda_callable__
7 GlobalIndexType
8 getSuperentityIndex( const LocalIndexType localIndex ) const;
```

Argument `localIndex`, ktorý sa vyskytuje pri metódach `getSubentityIndex` a `getSuperentityIndex` označuje i-tu nad-/pod-entitu danej entity.

Okrem toho, ku dátam duálneho grafu sa môže pristúpiť pomocou metód `getCellNeighborsCount` a `getCellNeighborIndex`. Tiež existujú metódy, ktoré umožňujú paralelne vykonať určitý funktor pre buď všetky entity siete danej dimenzie alebo len pre entity siete, ktoré splňujú určitú vlastnosť na základe ich značiek. Takéto metódy sú napríklad `forAll`, `forBoundary` a `forInterior`.

2.2.2 Statická konfigurácia

Ako bolo spomenuté v sekcii 2.1, nie všetky položky dátovej štruktúry musia byť pre určitý výpočet potrebné. Kvôli tomu, implementácia umožňuje dátovú štruktúru nakonfigurovať pomocou tzv. *statickej konfigurácie*, to znamená konfigurácie, ktorá je dodaná a vyriešená už počas času kompilácie. Ako bolo spomenuté, konfigurácia je predaná ako šablónový argument do hlavnej šablónovej triedy objektu siete `Mesh`. Konfigurácia samotná je reprezentovaná ako šablónová trieda, ktorá obsahuje príslušné statické typy, premenné a metódy, ktoré definujú dané položky konfigurácie. Je potrebné zdôrazniť, že všetky členy šablónovej triedy konfigurácie musia byť riešiteľné už počas kompilácie, čo znamená, že pri deklarácii členov je potrebné využiť kľúčového slova `constexpr`. Nevýhodou oproti konfigurácii, ktorá by sa riešila počas behu programu je to, že je potrebné už pri kompilácii programu rozhodnúť o konečnej množine všetkých konfigurácii siete, s ktorými sa bude pracovať. To môže v závislosti na počte použitých konfigurácií podstatne zvýšiť veľkosť skompilovaného binárneho súboru.

Jedným z hlavných účelov konfigurácie je umožniť voľbu incidentných matíc, ktoré budú z reprezentácie eliminované. Okrem toho tiež napríklad umožňuje aj zvoliť interné dátové typy, ktoré budú použité pre ukladanie rôznych typov indexov a tiež priestorových súradnici, čo môže byť použité pre dodatočné redukovanie pamäťových požiadavkov dátovej štruktúry. V konfigurácii sa tiež odlišujú dva typy indexov: globálne a lokálne. Globálne indexy označujú indexy, ktorými sa indexujú entity v rámci celej siete. Na druhej

strane, lokálne indexy sú použité na indexovanie incidentných entít z pohľadu entity, ktorá je s nimi incidentná (to znamená na indexovanie nenulových prvkov v rámci určitého riadku incidentnej matice). Celková konfigurácia pozostáva z nasledujúcich položiek:

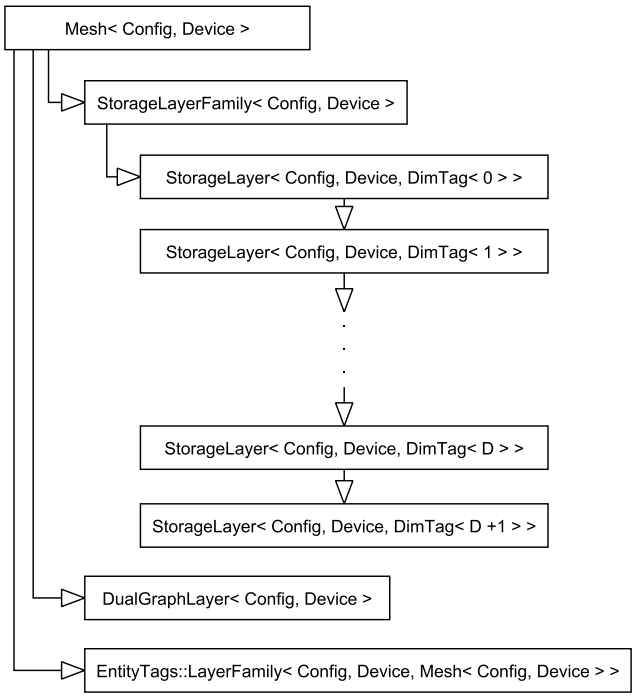
- Topológia bunky (`CellTopology`), ktorá tiež implikuje samotnú dimenziu siete D a topológiu všetkých ostatných entít siete.
- Dimenziu $N \geq D$ (`SpaceDimension`), ktorá odpovedá priestorovej dimenzii, v ktorom sa sieť nachádza a tým pádom implikuje počet položiek priestorových súradníc.
- Dátový typ položiek priestorových súradníc (`RealType`) (ako napríklad `float`, `double`), globálnych a lokálnych indexov (`GlobalIndexType`, `LocalIndexType`) (ako napríklad `int`, `short int`).
- Páry dimenzií d_1, d_2, kde $d_1 > d_2$, čo špecifikuje, ktoré matice pod-entít sú uložené. Je potrebné, aby všetky spomenuté d_1-dimenzionálne entity mali povolené incidentnú maticu s vrcholmi ($d_2 = 0$), kvôli ich prístupu ku geometrickej reprezentácii.
- Páry dimenzií d_1, d_2, kde $d_1 < d_2$, čo špecifikuje, ktoré matice nad-entít sú uložené.
- Binárna hodnota, ktorá špecifikuje, či duálny graf bude uložený.
- Minimálny počet spoločných vrcholov, pričom sú dva bunky považované sa susedné (`dualGraphMinCommonVertices`) a tak majú spoločnú hranu v duálnom grafe. Je potrebné, aby táto hodnota bola maximálne rovná dimenzii siete D.
- Dimenzie entít, pre ktoré budú uložené vnútorné/hraničné značky.

Členy šablónovej triedy konfigurácie vyžadujú splnenie určitého rozhrania, ktoré je očakávané zbytkom implementácie. Pre ukážku rozhrania spolu s kompletným príkladom definície šablónovej triedy konfigurácie, viď prílohu B.

2.2.3 Vrstvy dátovej štruktúry

Ako bolo spomenuté v sekcii 2.1, dátová štruktúra pre reprezentáciu siete obsahuje niekoľko polí a incidentných matíc, ktoré každé sprostredkúvajú určitú funkcionalitu a ich dostupnosť záleží na konfigurácii siete. Dátová štruktúra je organizovaná tak, že jednotlivé konfigurovateľné položky sú umiestnené do individuálnych vrstiev, ktoré sú reprezentované šablónovými triedami parametrizované dimenziou a/alebo typom topológie entity. Každá vrstva má dve špecializácie: špecializáciu, pri ktorej je daná interná dátová štruktúra členom vrstvi a tak je daná položka povolená a špecializáciu, pri ktorej je vrstva

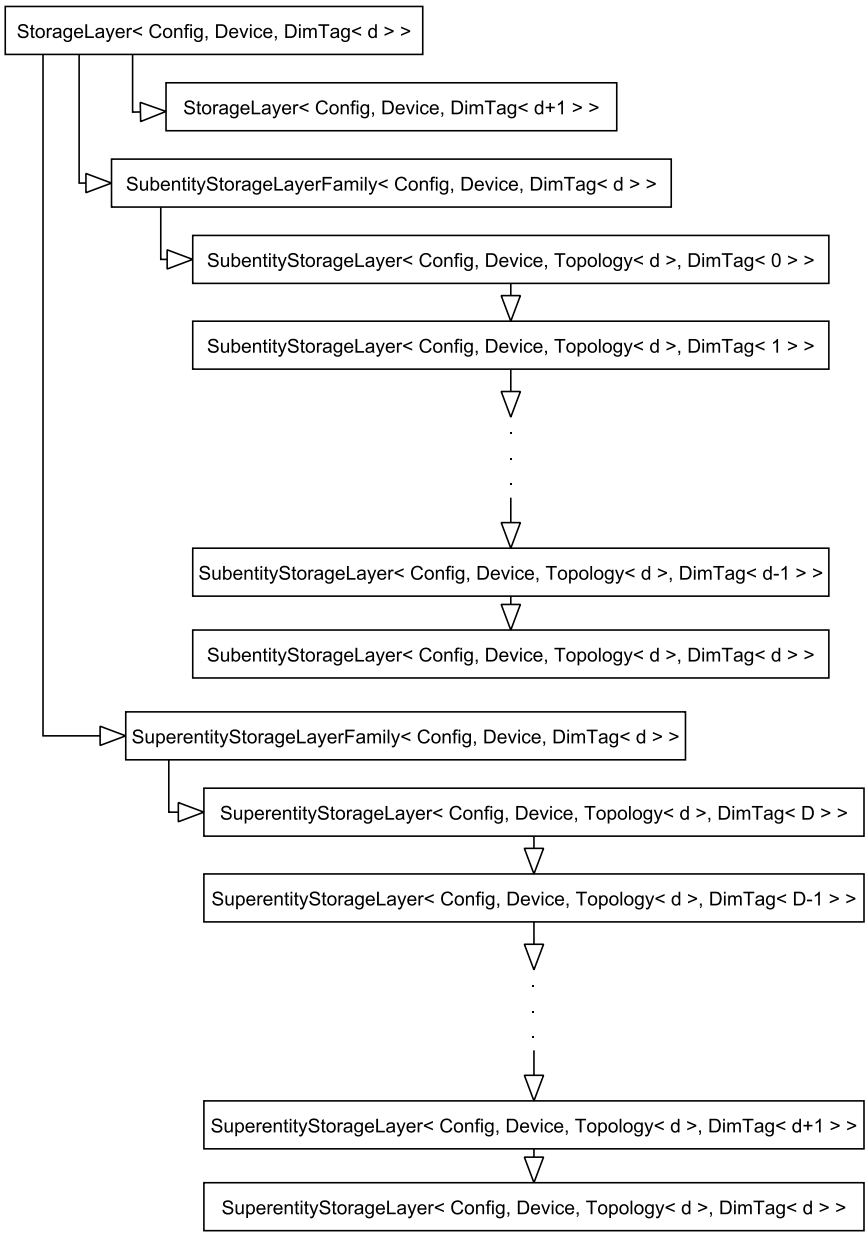
prázdna a tak je daná položka zakázaná. Jednotlivé vrstvy sú skombinované pomocou *rekurzívnej dedičnosti*, ktorá je ukončená pomocou použitia konceptu *čiastočných šablónových špecializácií*. Rekurzívna dedičnosť je návrhovým vzorom v jazyku C++, pri ktorom šablónová trieda dedí od tej istej šablónovej triedy s inými šablónovými argumentmi. To znamená, že dedičnosť nie je použitá na vyjadrenie vzťahu medzi dvomi rôznymi objektmi, ale je použitá pre obsiahnutie konečného počtu atribút z rôznych tried do jedného objektu.



Obr. 2.7: Diagram dedičnosti šablónovej triedy `Mesh`

Diagram dedičnosti hlavnej šablónovej triedy `Mesh` je možné vidieť na obrázku 2.7. Okrem toho, že táto trieda slúži hlavne ako počiatočný bod celej dedičnosti a pre vystavenie hlavného rozhrania dátovej štruktúry, je súčasťou tejto triedy aj pole jednotlivých priestorových súradníc, ktoré je v dátovej štruktúre vždy prítomné. Prvou dôležitou zdedenou triedou je šablónová trieda `StorageLayerFamily`, ktorá hlavne slúži ako štartovací bod pre rekurzívnu dedičnosť ostatných vrstiev a tiež pre presmerovanie volaní šablónových metód do ich príslušných vrstiev pomocou *tag dispatch*. Tag dispatch umožňuje zavolanie správnej zdedenej metódy zo špecifickej vrstvy pomocou *preťaženia metód*, pričom je hodnota dimenzie použitá ako parameter funkcie. Keďže preťaženie metód funguje len na základe typov parametrov, je potrebné aby špecifická hodnota dimenzie v parametroch bola reprezentovaná určitom typom a nie len hodnotou celočíselného dátového typu. Pre tieto účely je v implementácii používaná šablónová trieda `DimensionTag<d>`, ktorá umožňuje

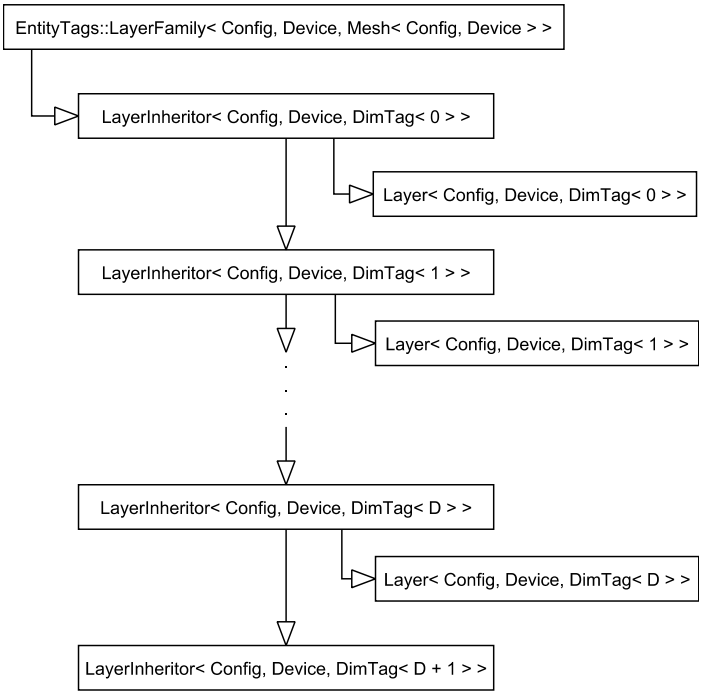
špecifikovať dimenziu šablónovým argumentom `d`.



Obr. 2.8: Diagram dedičnosti d-dimenzionálnej špecializácie šablónovej triedy `StorageLayer`

Rekurzívna dedičnosť počína šablónovou triedou `StorageLayer`, ktorá je parametrizovaná dimenziou `DimensionTag<d>` a slúži na obsiahnutie všetkých dát, ktoré súvisia s *d*-dimenzionálnymi entitami siete. Šablónová trieda `StorageLayerFamily` iniciuje rekurzívnu dedičnosť s dimenziou 0 a každá

d-dimenzionálna vrstva dedí od $(d+1)$-dimenzionálnej vrstvy. Rekurzia je ukončená pomocou prázdnej čiastočne špecializovanej $(D+1)$-dimenzionálnej vrstvy. Na obrázku 2.8 je možné vidieť diagram dedičnosti pre d-dimenzionálnu `StorageLayer`. Schéma rekurzívnej dedičnosti je podobná ako v obrázku 2.7, len s tým rozdielom, že sa dedia dve rôzne rodiny vrstiev pre uloženie dát súvisiacich s pod-entitami a nad-entitami d-dimenzionálnych entít. Triedy `SubentityStorageLayer` a `SuperentityStorageLayer` sú parametrizované unikátnymi pármi dimenzií (d_1, d_2), pričom každá z nich slúži pre uloženie incidentnej matice I_{d_1,d_2}. `SubentityStorageLayer` ukladá maticu pod-entít, t.j. I_{d_1,d_2}, kde $d_1 > d_2$. `SuperentityStorageLayer` ukladá maticu nad-entít, t.j. I_{d_1,d_2}, kde $d_1 < d_2$, ktorá je doplnená dodatočným poľom s počtami nenulových elementov riadkov matice. Je ešte potrebné upozorniť na poradie dimenzií v rekurzívnej dedičnosti týchto dvoch rodín vrstiev. `SubentityStorageLayer` začína s $d_2 = 0$ a pokračuje inkrementáciou d_2. `SuperentityStorageLayer` naopak začína s $d_2 = D$ a pokračuje dekrementáciou d_2. V oboch prípadoch sa rekurzívna dedičnosť ukončí pomocou prázdnej čiastočnej špecializácie keď dimenzie $d_1 = d_2$.



Obr. 2.9: Diagram dedičnosti šablónovej triedy `EntityTags::LayerFamily`

Vnútorné/hraničné značky entít sú tiež realizované pomocou rekurzívnej dedičnosti, keďže značky sú rozlišované pre entity všetkých dimenzií. Obrázok 2.9 znázorňuje hierarchiu dedičnosti pre uloženie značiek entít. Počiatočná trieda rodiny vrstiev značiek je `EntityTags::LayerFamily`, ktorá

hraje podobnú rolu ako `StorageLayerFamily`. Rekurzívna dedičnosť naprieč všetkým dimenziám je vykonávaná pomocou triedy `LayerInheritor`, ktorá je podobne ako `StorageLayer`, parametrizovaná šablónovým argumentov dimenzie d, ktorý sa postupne inkrementuje. Rekurzia je opäť ukončená pomocou prázdnej čiastočnej špecializácie. Samotné dáta pre značky d-dimenzionálnych entít sú uložené v triede `Layer`, ktorú dedí d-dimenzionálny `LayerInheritor`. Trieda `EntityTags::LayerFamily` je v hierarchii dedičnosti umiestnená ako rodič hlavnej triedy `Mesh`.

Duálny graf pre vyjadrenie susedstva buniek siete je tiež súčasťou svojej príslušnej vrstvy. Keďže je duálny graf reprezentovaný len jednou maticou spolu s poľom počtov nenulových elementov riadkov matice, nie je v tomto prípade nutné použiť rekurzívnu dedičnosť, a tak je táto položka realizovaná len pomocou jednej šablónovej triedy `DualGraphLayer`. Táto vrstva je v hierarchii umiestnená ako rodič šablónovej triedy `StorageLayerFamily`.

2.2.4 Reprezentácia typov topológii

Jedným z najdôležitejších položiek konfigurácie siete, je informácia o tom, akú topológiu majú bunky siete. Topológie sú reprezentované ako typy, pričom ich definícia je v podobe štruktúr, ktoré sú organizované do menného priestoru `TNL::Meshes::Topologies`. Napríklad topológia trojuholníka je v TNL definovaná nasledovne:

```
1  struct Triangle
2  { static constexpr int dimension = 2; };
3
4  template<>
5  struct Subtopology< Triangle, 0 >
6  {
7     typedef Vertex Topology;
8     static constexpr int count = 3;
9  };
10
11 template<>
12 struct Subtopology< Triangle, 1 >
13 {
14    typedef Edge Topology;
15    static constexpr int count = 3;
16 };
17
18 template<> struct SubentityVertexMap< Triangle, Edge, 0, 0>
19 { enum { index = 1 }; };
20 template<> struct SubentityVertexMap< Triangle, Edge, 0, 1>
21 { enum { index = 2 }; };
22
23 template<> struct SubentityVertexMap< Triangle, Edge, 1, 0>
24 { enum { index = 2 }; };
25 template<> struct SubentityVertexMap< Triangle, Edge, 1, 1>
26 { enum { index = 0 }; };
27
```

```
28 template<> struct SubentityVertexMap< Triangle, Edge, 2, 0>
29 { enum { index = 0 }; };
30 template<> struct SubentityVertexMap< Triangle, Edge, 2, 1>
31 { enum { index = 1 }; };
```

Štruktúra `Triangle` reprezentuje typ topológie, ktorá tiež obsahuje statický atribút `dimension` určujúci dimenziu danej topológie. V prípade, že by táto topológia bola bunkou nejakej siete, informácia o dimenzii siete by sa určila práve z tohoto atribútu. Keďže každá topológia tiež definuje topológiu všetkých svojich pod-entít, je informácia o štruktúre pod-entít reprezentovaná pomocou špecializácií štruktúr `Subtopology` a `SubentityVertexMap`. `Subtopology` definuje typ a množstvo pod-entít danej pod-dimenzie pomocou statických atribút `Topology` a `count`. Atribút `SubentityVertexMap` slúži pre určenie, ktoré indexy vrcholov entity danej topológie priliehajú na jednotlivé pod-entity daných pod-dimenzií. Prvé dva šablónové parametre určujú typ topológie a pod-topológie, pričom tretí a štvrtý určuje index a lokálny index vrcholu pod-entity. Samotná hodnota indexu vrcholu je reprezentovaná atribútom `index`.

2.2.5 Traits triedy

Pre zjednodušenie a sprehľadnenie definícií tried naprieč celou implementáciou, je definovaných niekoľko pomocných šablónových tried so sufixom `Traits`. Tieto triedy hlavne poskytujú deklarácie rôznych užitočných typov v podobe statických atribút na základe konfigurácie siete, ktorá je poskytnutá pomocou šablónového argumentu. Najdôležitejšie sú tieto pomocné šablónové triedy:

- `MeshTraits<Config, Device>`
- `MeshSubentityTraits<Config, Device, Topology, Dim>`
- `MeshSuperentityTraits<Config, Device, Topology, Dim>`
- `MeshEntityTraits<Config, Device, Dim>`

Šablónová trieda `MeshTraits` deklaruje pomocné typy, ktoré súvisia so sieťou ako celkom. To sú hlavne typy pochádzajúce z konfigurácie siete, ako napríklad `CellTopology`, `GlobalIndexType`, `LocalIndexType` a `RealType`. Okrem toho sú tiež deklarované typy, ktoré súvisia priamo s návrhom internej štruktúry siete, ako napríklad typ dátovej štruktúry použitej pre uloženie priestorových súradníc, pod-entít, nad-entít a duálneho grafu. Umiestnenie deklarácií typov použitých dátových štruktúr do pomocnej triedy má napríklad takú výhodu, že je veľmi jednoduché zmeniť typ použitého formátu riedkej matice pre incidentné matice v prípade, že by to upráva návrhu dátovej štruktúry vyžadovala.

Šablónové triedy `MeshSubentityTraits` a `MeshSuperentityTraits` deklarujú typy, ktoré umožňujú jednoduchšiu prácu so štrukturálnymi informáciami pod-entít a nad-entít. Šablónové parametre `Topology` a `Dim` špecifikujú o ktoré

pod-entity alebo nad-entity sa jedná. Tieto triedy vlastne uľahčujú prácu so štruktúrami zo sekcie 2.2.4.

Šablónová trieda `MeshEntityTraits` deklaruje typy, ktoré súvisia s entitami dimenzie `Dim` ako celku. Dôležitou funkciou triedy je to, že umožňuje skonvertovať dimenziu entity na typ jej topológie. Okrem toho tiež deklaruje rôzne typy súvisiace so zárodkami entít danej dimenzie.

2.2.6 Inicializácia siete

Každý objekt siete, tzn. inštancia šablónovej triedy `Mesh<Config, Device>`, musí byť pred použitím inicializovaný. Inicializácia je vykonaná pomocou členskej funkcie `init`:

```
1 void init( PointArrayType& points, CellSeedArrayType& cellSeeds );
```

Funkcia `init` je dostupná len v prípade, že parameter `Device` je špecifikovaný ako `TNL::Devices::Host`. Dôvodom je to, že pre inicializáciu sú dostupné len siete, ktorú sú na strane CPU, keďže proces inicializácie je sekvenčný. Pre použitie siete na strane GPU, je teda potrebné sieť najprv inicializovať na strane CPU a potom ju skopírovať do pamäte GPU. To je možné dosiahnuť buď pomocou kopírovacieho konštruktoru:

```
1 Mesh<Config, Devices::Host> hostMesh;
2 ...
3 Mesh<Config, Devices::Cuda> deviceMesh( hostMesh );
```

alebo operátoru priradenia:

```
1 Mesh<Config, Devices::Host> hostMesh;
2 ...
3 Mesh<Config, Devices::Cuda> deviceMesh;
4 deviceMesh = hostMesh;
```

Prvý parameter funkcie `init` je pole objektov `PointType`, pričom i-ty objekt reprezentuje Karteziánske súradnice i-teho vrcholu v Euklidovskom priestore. Keďže je priestorová dimenzia siete známa pri kompilácii, je objekt typu `PointType` reprezentovaný v pamäti ako statické pole o fixnej veľkosti. Druhý parameter funkcie je pole objektov typu `CellSeedType` (šablónový alias pre triedu `Seed`), ktoré reprezentujú zárodky buniek. Objekty pozostávajú zo zoznamov indexov vrcholov, ktoré umožňujú unikátne identifikovať ľubovoľnú ne-vrcholovú entitu siete. Konkrétne, j-ty objekt typu `CellSeedType` reprezentuje zárodok j-tej bunky siete. Keďže všetky podporované topológie majú staticky známu štruktúru, sú tieto zárodky realizované v pamäti ako statické pole o fixnej veľkosti. Ostatné pod-entity každej bunky sú odvodené na základe šablónových štruktúr `SubentityVertexMap`, ktoré unikátne mapujú jednotlivé vrcholy pod-entít na vrcholy bunky (viď sekcia 2.2.4).

Pre účely rudimentárnej validácie a zhotovenia objektov, ktoré sa dodajú funkcii `init`, existuje pomocná šablónová trieda `MeshBuilder`. Objekt tohoto typu interne obsahuje objekty typov `PointArrayType` a `CellSeedArrayType` a poskytuje rozhranie pre nastavenie jednotlivých objektov daných polí:

```
1 void setPointsCount( const GlobalIndexType& cellsCount );
2 void setCellsCount( const GlobalIndexType& cellsCount );
3 void setPoint( GlobalIndexType i, const PointType& point );
4 CellSeedType& getCellSeed( GlobalIndexType j );
```

Funkcie `setPointsCount` a `setCellsCount` slúžia pre alokáciu interných polí. Funkcie `setPoint` slúži pre nastavenie i-teho vrcholu a funkcia `getCellSeed` umožňuje prístup ku j-temu zárodku bunky, ktorého jednotlivé indexy je možné nastaviť indexovým operátorom. Po nastavení všetkých vrcholov a indexov, je možné samotnú sieť inicializovať pomocou členskej funkcie `build`:

```
1 bool build( MeshType& mesh );
```

Táto funkcia najprv vykoná validáciu samotných nastavených údajov a potom zavolá funkciu `init`. V prípade, že sa validácia nepodarí, sieť nie je inicializovaná a funkcia vráti hodnotu `false`. Príklad inicializácie jednoduchej štvorstennej siete je možné vidieť v prílohe C.1.

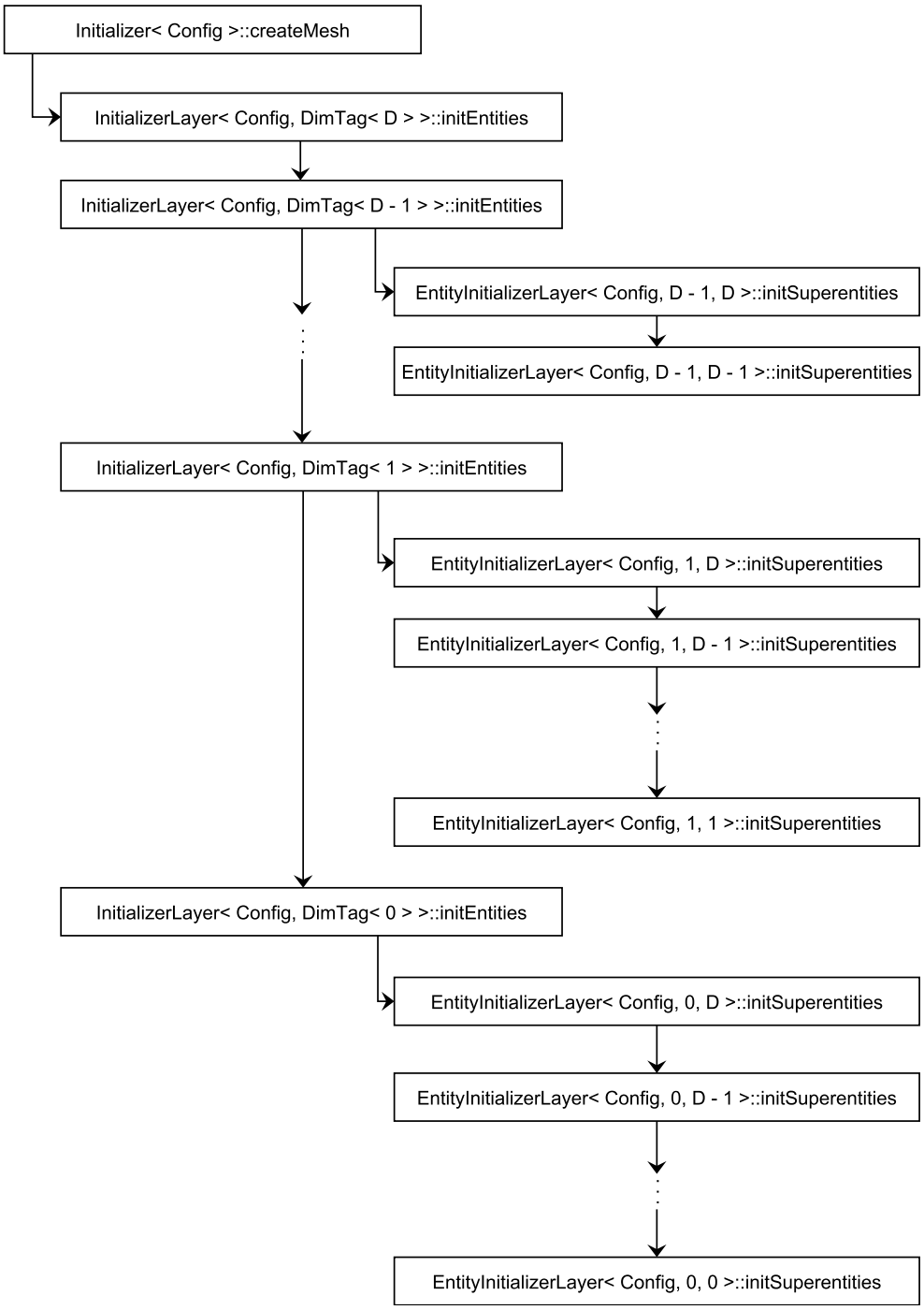
2.2.6.1 Inicializačný algoritmus

Implementácia inicializácie siete je organizovaná do rodiny šablónových tried, ktoré podobne ako to bolo pri definícii vrstiev objektu siete (viď sekciu 2.2.3), využívajú rekurzívnu dedičnosť pre iteráciu naprieč dimenziám v šablónových argumentoch a pre vynechanie inicializácie tých incidentných matíc, ktoré sú v konfigurácii siete zakázané. Znázornenie týchto tried spolu s konkrétnymi metódami, ktoré sú počas celkovej inicializácie volané je možné vidieť na obrázku 2.10.

Inicializácia D-dimenzionálnej siete začína triedou `Initializer`, menovite v členskej funkcii `createMesh`:

```
1 void createMesh( PointArrayType& points,
2                  CellSeedArrayType& cellSeeds,
3                  MeshType& mesh );
```

Účelom funkcie `createMesh` je najprv inicializovať pole priestorových súradníc siete pomocou presunu obsahu z poľa `points` a potom započať inicializáciu entít siete zavolaním členskej funkcie `initEntities`, ktorá patrí šablónovej triede `InitializerLayer<Config, d>`. Účelom tejto triedy je inicializácia matíc pod-entít a nad-entít súvisiace s d-dimenzionálnymi entitami siete. Entity sa inicializujú v poradí od entít dimenzie D až po dimenziu 0, t.j. od `InitializerLayer<Config, D>` až po `InitializerLayer<Config, 0>`.



Obr. 2.10: Triedy a volané metódy pri inicializácii D-dimenzionálnej siete. `EntityInitializer< Config, d, d >` slúži len pre ukončenie rekurzie

Špecializácia `InitializerLayer<Config, D>` inicializuje incidentnú maticu $I_{D,0}$, ktorá je realizovaná skopírovaním listov vrcholov z poľa `cellSeeds` do patričných riadkov matice. Pole `cellSeeds` je nasledovne de-alokované. Hlavným dôvodom prečo sa incidentná matica $I_{D,0}$ inicializuje ako prvá je to, že vrcholy buniek sú dôležité pre enumeráciu všetkých ostatných entít nižšej dimenzie okrem vrcholov, ako to bolo predtým spomenuté.

Špecializácie `InitializerLayer<Config, d>`, kde $0 < d < D$, inicializujú incidentnú maticu $I_{d,0}$ a incidentné matice $I_{T,d}$ a $I_{d,T}$, kde $d < T < D$. Matica $I_{d,0}$ je inicializovaná tak, že sa na začiatku pre každú bunku vytvoria zárodky (zoznam vrcholov) jej patričných d-dimenzionálnych pod-entít a uložia sa do indexovej množiny, ktorá je realizovaná pomocou kontajnera `std::unordered_map<Seed, Index>`. O vytvorenie zárodkov pod-entít sa stará šablónová trieda `SubentitySeedsCreator`, obsahujúca statickú členskú metódu `create` s účelom vytvoriť pole zárodkov pod-entít na základne statických štrukturálnych údajov topológii, ktoré sú poskytnuté špecializáciami šablónovej štruktúry `SubentityVertexMap` (viď sekciu 2.2.4). Dôvodom prečo sú zárodky pod-entít ukladané do množiny je to, aby sa vyhlo duplicitným entitám, keďže niektoré bunky siete môžu zdieľať rovnaké pod-entity (ako napríklad dva trojuholníky môžu zdieľať hranu). Indexová množina tiež entitám poskytne unikátny index, čo reprezentuje ich globálny index. Dôvod prečo sa používa kontajner realizovaný hash tabuľkou je ten, že porovnanie zárodkov entít vyžaduje radenie daných zoznamov vrcholov, čo je podstatne náročná operácia. Hashová tabuľka umožní počet celkových radení zredukovať tým, že sa vo väčšine prípadoch pre porovnanie zárodkov vypočíta len hash, čo je podstatne menej náročná operácia. Je tiež potrebné, aby hash jednotlivých vrcholov v liste bol skombinovaný komutatívnou operáciou (ako napríklad sčítanie), keďže vrcholy indexov v liste reprezentujúce tú istú entitu, nemusia byť poskytnuté v tom istom poradí. Po vytvorení indexovej množiny, sú jednotlivé unikátne zoznamy vrcholov d-dimenzionálnych entít skopírované do patričných riadkov incidentnej matice $I_{d,0}$. Inicializácia ostatných matíc pod-entít a nad-entít prebieha v rodine šablónových tried, ktoré pozostávajú z počiatočnej triedy `EntityInitializer` a tried `EntityInitializerLayer`, ktoré pomocou rekurzívnej dedičnosti iterujú nad pármi dimenzií (d, P), kde $d < P < D$. Jednotlivé `EntityInitializerLayer<Config, d, P>` majú za úlohu inicializovať incidentné matice $I_{d,P}$ a $I_{P,d}$. Matica pod-entít $I_{P,d}$ je inicializovaná podobne ako matica $I_{d,0}$, len s tým rozdielom, že zoznamy vrcholov pod-entít nemusia byť vkladané do indexovej množiny, pretože sa množina vytvorila už predtým pri inicializácii matice $I_{d,0}$. Kvôli tomu je krok vytvárania indexovej množiny preskočený a už vytvorená množina je použitá pre získanie indexov entít, ktoré odpovedajú daným listom vrcholov. Matica nad-entít $I_{d,P}$ sa inicializuje podobne ako $I_{P,d}$, len s tým rozdielom, že je mapovanie obrátené. V prípade, že konfigurácia siete povoľuje súčasne $I_{d,P}$ aj $I_{P,d}$, je inicializácia $I_{d,P}$ implementovaná trochu efektívnejšie, keďže je možné využiť už vytvorenú $I_{P,d}$ a vyhnúť sa tak tvoreniu zoznamov vrcholov pod-entít pomocou

`SubentitySeedsCreator` a hľadaniu v indexovej množine.

Špecializácia `InitializerLayer<Config, 0>` inicializuje incidentné matice $I_{0,U}$, kde $0 < U < D$, pričom iterácia nad dimenziami je opäť vykonaná rekurzívnou dedičnosťou triedy `EntityInitializerLayer`. Inicializácia prebieha podobne ako v predošlom odseku, len s tým rozdielom, že nie je potrebné použiť `SubentitySeedsCreator` a indexovú množinu, pretože vrcholy nemajú žiadne pod-entity a tým pádom sú sami o sebe unikátne (zárodky vrcholov sú zoznamy vrcholov o veľkosti 1). Je tiež potrebné pripomenúť, že v tomto prípade nie je potrebné inicializovať matice pod-entít $I_{d,0}$, kde $0 < d < D$, pretože tieto matice už boli inicializované po vytváraní indexových množín v predošlých `InitializerLayer`.

Po dokončení inicializácie všetkých incidentných matíc ešte prebieha inicializácia duálneho grafu a vnútorných/hraničných značiek, ale keďže tieto dve položky nie sú veľmi dôležité pre účely tejto práce, budú detaily ich inicializácie vynechané.

2.2.7 Vstup/Výstup z/do súboru

Funkcionalita čítania siete zo súboru je veľmi dôležitá v praxi, kde je sieť väčšinou pripravená dopredu nejakým nástrojom, uložená na disk a potom sa táto sieť použije na výpočet pomocou nejakého iného programu. Kvôli tomu knižnica TNL podporuje čítanie a zapísanie siete z/do súboru vo formátoch ako napríklad: legacy VTK, VTK v XML a Netgen.

Proces čítania siete zo súboru a vytvorenia objektu `Mesh` sa dá zhrnúť do niekoľko fáz. Prvou fázou je parsovanie vstupného súboru a zistenie jednotlivých vlastností siete, ktoré súvisia s parametrami konfigurácie siete, ako je napríklad topológia bunky, priestorová dimenzia a podobne. Ďalším krokom je vyriešiť typ konfigurácie siete počas behu programu, ktorý je použitý ako šablónový argument objektu `Mesh`. Posledným krokom je inicializácia siete na základe dát zo súboru.

Vyriešenie typu šablónovej triedy konfigurácie siete počas behu programu je realizované pomocou variadickej šablónovej triedy `MeshTypeResolver`. Táto trieda zhotovuje typ konfigurácie siete počas behu programu tak, že na základe hodnôt položiek konfigurácie v podobe reťazca, ktoré boli obdržané v prvej fáze, sú postupne zvolené jednotlivé typy šablónových argumentov pomocou kaskády if/else príkazov, ktoré sú následne poskytnuté predvolenej šablónovej triede konfigurácie siete. Keďže počet všetkých možných kombinácií šablónových argumentov, ktoré môžu špecifikovať konfiguráciu siete, je potencionálne celkom vysoký, je potrebné hodnoty parametrov limitovať len na tie, ktoré budú skutočne použité. To sa staticky špecifikuje pomocou špecializácií určitých šablónových tried pre všetky žiaduce hodnoty položiek konfigurácie, čo limituje `MeshTypeResolver` len na overovanie týchto hodnôt, a tak limituje počet možných konfigurácií siete, ktoré sa zo súboru môžu

prečítať. Implementácia `MeshTypeResolver` je celkom komplikovaná, ale jej použitie je možné zhrnúť nasledujúcim pseudo-kódom:

```
using Reader = [based on file type detection]
Reader reader(input_file);
using MeshType = [resolved by MeshTypeResolver from the reader]
MeshType mesh = reader.readMesh();
return functor(reader, mesh);
```

Objekt `functor` je poskytnutý užívateľom a je to typicky C++ lambda funkcia s generickými parametrami (pre typy parametrov je použité kľúčové slovo `auto`), čo umožňuje do funkcie predať objekt `Mesh` ľubovoľného typu. Sieť je pred poslaním do lambda funkcie tiež inicializovaná.

Sieť je v súbore reprezentovaná pomocou zoznamu priestorových súradníc vrcholov a listami indexov vrcholov pre každú bunku (zárodky buniek). Tieto údaje sú počas parsovania uložené do dočasných polí. Inicializácia siete potom prebieha tak, že dáta z dočasných polí sú skopírované do interných polí objektu `MeshBuilder`, ktorý danú sieť inicializuje (viď sekcia 2.2.6).

Kapitola 3

Rozšírenie dátovej štruktúry

Táto kapitola sa zaoberá rozšírením dátovej štruktúry, ktorá bola popísaná v kapitole 2, o podporu niekoľko nových topológii. Aj napriek tomu, že hlavným cieľom práce bolo rozšírenie podpory o topológiu mnohouholníka a mnohostenu, bola tiež ako vedľajší účinok pridania podpory pre topológiu mnohouholníka, pridaná podpora pre topológiu ihlana a klinu. Najprv sa kapitola zameriava na popis rozdielov, ktorými sa nové topológie líšia od topológii podporovaných pôvodnou dátovou štruktúrou. Potom sú prezentované návrhové zmeny štruktúry siete, ktoré umožnia rozšíriť podporu dátovej štruktúry o nové topológie. Na záver sú popísané detaily spojené s implementáciou rozšírenia dátovej štruktúry.

3.1 Nové topológie

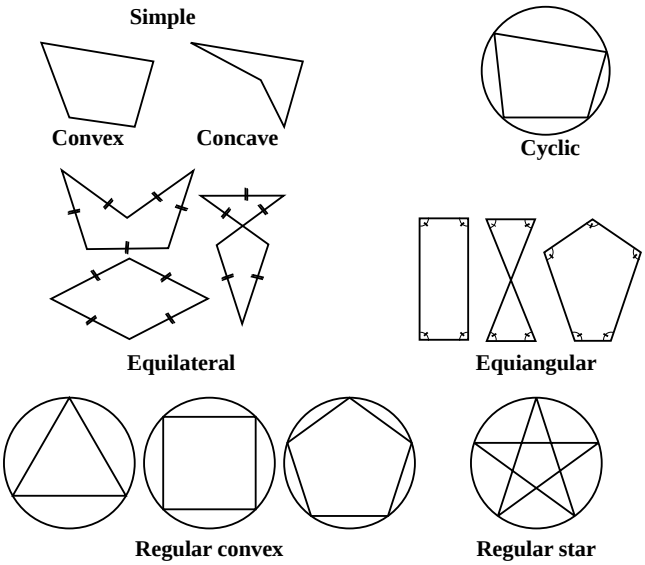
Pôvodná dátová štruktúra, popísaná v kapitole 2, obsahovala podporu pre silno homogénne siete, kde bunky a jednotlivé pod-entity majú konštantný tvar, čo umožňuje definovať typ topológie úplne staticky. Z toho tiež vyplýva, že počty pod-entít jednotlivých pod-dimenzii každej bunky, sú konštantné. Týmto topológiam sa bude naďalej hovoriť ako *statické topológie*.

Ešte predtým než sa porovnajú statické topológie a novo implementované topológie, budú nové topológie definované pomocou definícii nižšie.

Definícia 3.1 (mnohouholník)**.** *Mnohouholník* je 2-dimenzionálny geometrický útvar, ktorý je definovaný uzatvorenou konečnou sériou úsečiek (konečný bod poslednej úsečky je počiatočným bodom prvej úsečky). Séria úsečiek ohraničuje 2-dimenzionálnu rovinnú oblasť, ktorej sa hovorí *mnohouholník*.

Ohraničujúce úsečky reprezentujú hrany. Body, kde sa úsečky stretávajú, reprezentujú vrcholy mnohouholníka. Mnohouholníku s n hranami sa hovorí *n-uholník*, ako napríklad trojuholník (3-uholník). Každý mnohouholník musí mať aspoň 3 hrany, inak nie je možné vytvoriť uzatvorenú sériu úsečiek. Pre

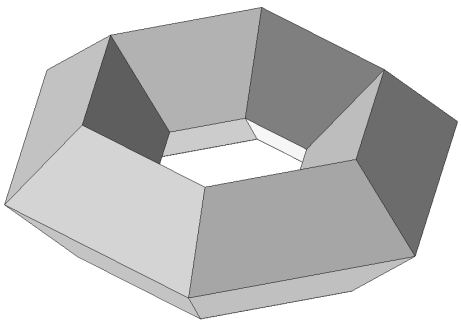
každý n-uholník tiež platí, že počet vrcholov je rovný n. Znázornenie príkladov niekoľkých mnohouholníkov je možné vidieť na obrázku 3.1.



Obr. 3.1: Rôzne mnohouholníky

Definícia 3.2 (mnohosten)**.** Existuje mnoho rôznych spôsobov definície, ale pre účely tejto práce, je *mnohosten* 3-dimenzionálny geometrický útvar, ktorý je definovaný konečnou množinou mnohouholníkových stien, reprezentujúcich ohraničenie určitej konečnej 3-dimenzionálnej oblasti.

Hranica mnohostenu je spojená, tzn. hranica neobsahuje žiadne diery. Mnohostenu s n stenami sa hovorí *n-sten*, ako napríklad štvorsten (4-sten). Keďže ohraničenie nemôže mať žiadne diery, minimálny počet stien mnohostenu je 4. Príklady mnohostenov je možné vidieť na obrázkoch 3.2 a 3.3.

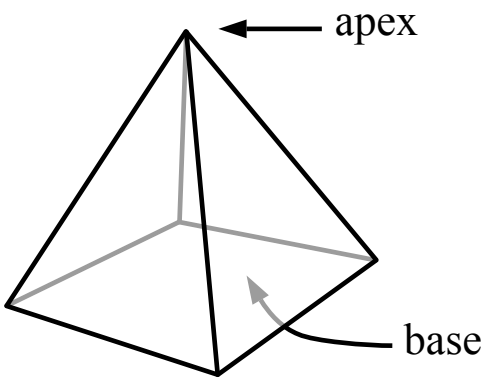


Obr. 3.2: Toroidný mnohosten

Obr. 3.3: 20-sten

Definícia 3.3 (ihlan)**.** *Ihlan* alebo *pyramída* je špeciálny prípad mnohostenu, ktorý pozostáva z jedného n-uholníka (podstava) a n trojuholníkových stien, pričom trojuholníkové steny vzniknú spojením jednotlivých hrán podstavy s jedným bodom (vrcholom ihlana), ktorý sa nachádza mimo podstavy.

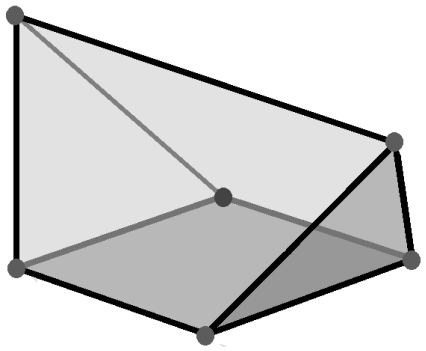
V tejto práci sa bude predpokladať len ihlan so štvoruholníkovou podstavou kvôli konzistencii s knižnicou VTK [11]. To znamená, že ihlan bude stále pozostávať z 5 stien, 8 hrán a 5 vrcholov. Znázornenie takéhoto ihlanu je možne vidieť na obrázku 3.4.



Obr. 3.4: Ihlan so štvoruholníkovou podstavou

Definícia 3.4 (klin)**.** Klin je špeciálny prípad mnohostenu, ktorý pozostáva s 2 trojuholníkových a 3 štvoruholníkových stien, pričom trojuholníkové steny reprezentujú podstavy a jednotlivé štvoruholníkové steny vzniknú tak, že sa spoja konce jednotlivých unikátnych párov hrán, pričom prvá hrana páru je z prvej podstavy a druhá hrana páru je z druhej podstavy.

Klin vždy pozostáva z 5 stien, 9 hrán a 6 vrcholov. Znázornenie klinu je možné vidieť na obrázku 3.5.



Obr. 3.5: Klin

Definície 3.1 a 3.2 implikujú, že topológie mnohouholníka a mnohostenu nemôžu byť považované za statické topológie. To je kvôli tomu, že mnohouholníky a mnohosteny sú zostavené z ľubovoľného počtu hrán a stien, čo znamená, že počty pod-entít jednotlivých pod-dimenzii každej bunky, nie sú zaručene konštantné, a tak typ topológie nie je možné definovať úplne statický. Keďže jednotlivé bunky teda môžu mať ľubovoľné počty pod-entít, je potrebné, aby táto informácia bola uložená dynamicky pre každú bunku.

Topológie ihlana a klinu, ako sú popísané v definícii 3.3 a definícii 3.4, majú síce konštantné počty pod-entít jednotlivých pod-dimenzií, ale aj napriek tomu, nemôžu byť úplne považované za statické topológie. To je kvôli tomu, že ihlany a kliny obsahujú trojuholníky a štvoruholníky súčasne ako pod-entity druhej dimenzie, čo porušuje vlastnosť silnej homogénnosti. Z tohoto dôvodu, nie je možné tieto topológie reprezentovať v pôvodnej implementácii. Na druhej strane, topológia mnohouholníka otvára možnosť tieto topológie reprezentovať, keďže každý mnohouholník môže mať rôzne počty hrán a tak je možné reprezentovať trojuholníky a štvoruholníky súčasne ako 2-dimenzionálne pod-entity. Typ týchto topológii je možné definovať úplne staticky, ale je potrebné pridať informáciu o tom, koľko hrán obsahujú jednotlivé pod-entity druhej dimenzie, t.j. mnohouholníky.

Bolo spomenuté, že mnohouholník je definovaný ako uzatvorená konečná séria úsečiek, ale je možné zápis mnohouholníka skrátiť len na sériu vrcholov, ktoré reprezentujú počiatočné vrcholy úsečiek v sérii. Konečný vrchol i-tej úsečky je teda $(i+1)$-ty vrchol v sérii, pričom konečný vrchol poslednej úsečky je rovný prvému vrcholu v sérii. Kvôli tomu je možné mnohouholníkové bunky siete unikátne definovať pomocou zoznamu vrcholov, podobne ako tomu bolo pri statických topológiach. Keďže štruktúra ihlanov a klinov je definovaná staticky, je tiež možné bunky týchto topológii unikátne definovať pomocou zoznamov vrcholov.

Na druhej strane, pri mnohostenoch nie je zoznam vrcholov postačujúci. To je kvôli tomu, že zoznam vrcholov nedefinuje unikátne jednotlivé steny mnohostenu. Kvôli tomu bunky s topológiou mnohostenu vyžadujú unikátnu definíciu pomocou zoznamov stien, ktoré musia byť doplnené definíciami jednotlivých stien, reprezentované ako zoznamy vrcholov. Táto myšlienka je inšpirovaná formátom FPMA, ktorý je používaný pre ukladanie mnohostenných sietí v súbore. Formát FPMA je detailnejšie popísaný v sekcii 3.3.4.

V zhrnutí, topológie mnohouholníka a mnohostenu sa teda líšia od statických topológii hlavne tým, že počet pod-entít jednotlivých dimenzií nie je konštantný, ale variabilný. Kvôli tomu budú tieto dve topológie považované za tzv. *dynamické* topológie. Topológie klinu a ihlanu síce používajú topológiu mnohouholníka pre 2-dimenzionálne pod-entity, ale budú zaradené medzi statické topológie, keďže počet entít jednotlivých dimenzií je konštantný.

3.2 Zmeny v reprezentácii štruktúry siete

Čo sa týka reprezentácie štruktúry siete, dynamické topológie sa oproti statickým topológiam líšia na úrovni interných dátových štruktúr len v jednom aspekte. Tým aspektom je to, že matice pod-entít teraz obsahujú riadky, ktorých dĺžka nie je zaručene konštantná, ale variabilná pre každý riadok. Ako bolo popísané v sekcii 2.1.2, matice pod-entít, pri statických topológiach, sú vyjadrené pomocou maticového formátu Ellpack. Formát implicitne neukladá dĺžky jednotlivých riadkom, čo znamená, že v prípade dynamických topológii, je potrebné maticu doplniť poľom s dĺžkami jednotlivých riadkov matice. Formát Ellpack teoreticky podporuje matice s rôznymi dĺžkami riadkov, ale v prípade, že riadky majú ľubovoľné dĺžky, môže nastať v určitých prípadoch ku vysokej nadbytočnej pamäťovej réžii kvôli zarovnaniu riadkov. Z tohoto dôvodu je vhodnejšie pre matice pod-entít, pri dynamických topológiach, použiť formát Sliced Ellpack. Je si možné všimnúť, že presne týmto spôsobom sú reprezentované matice nad-entít pri statických topológiach. To dáva zmysel, keďže počty nad-entít rôznych entít môžu byť rozdielne, čo znamená, že situácia ukladania matíc nad-entít pri statických topológiach, je identická so situáciou ukladania matíc pod-entít pri dynamických topológiach. V zhrnutí, sú teda matice pod-entít, pri dynamických topológiach, uložené pomocou formátu Sliced Ellpack s poľom obsahujúce dĺžky riadkov matice, namiesto len formátu Ellpack bez poľa dĺžok riadkov matice, ako to bolo pri pôvodnom návrhu dátovej štruktúry. Pri pod-entitách mnohouholníkov je ešte možné vykonať menšiu optimalizáciu, ktorá spočíva v tom, že incidentné matice $I_{2,0}$ a $I_{2,1}$ môžu zdieľať to isté pole dĺžok riadkov matice. To vyplýva z toho, že počet vrcholov a hrán mnohouholníka je identický.

Čo sa týka ostatných položiek reprezentácie štruktúry siete (t.j. priestorové súradnice, matice nad-entít, duálny graf buniek a vnútorne/hraničné značky), je návrh interných dátových štruktúr, pri dynamických topológiach, úplne identický ako pri statických topológiách. To znamená, že variabilný počet pod-entít jednotlivých dimenzií entít s dynamickou topológiou nekladie na tieto položky pôvodnej reprezentácie siete žiadne dodatočné návrhové požiadavky.

Je tiež potrebné upresniť, že sieť teraz môže súčasne pozostávať s entít, ktoré majú statickú topológiu a s entít, ktoré majú dynamickú topológiu. Príkladom je sieť s mnohouholníkových bunkami. V tomto prípade, entity druhej dimenzie (mnohouholníky) majú dynamickú topológiu, a entity prvej a nultej dimenzie (hrany a vrcholy) majú statickú topológiu. Z toho vyplýva, že incidentné matice $I_{2,1}$, $I_{2,0}$ sú uložené pomocou formátu Sliced Ellpack s poľom dĺžok riadkov a incidentná matica $I_{1,0}$ je uložená pomocou formátu Ellpack. To znamená, že návrhová zmena dátovej štruktúry ovplyvní len entity siete s topológiami mnohouholníka a mnohostenu, ale pôvodný návrh uloženia ostatných entít so statickými topológiami bude zachovaný, čo tiež umožní zachovať efektivitu interných dátových štruktúr reprezentácie siete pri sietiach podporovaných pôvodným návrhom dátovej štruktúry.

3.3 Implementačné detaily

Táto sekcia popisuje detaily súvisiace s rozšírením implementácie dátovej štruktúry pre neštruktúrované siete (viď sekciu 2.2) o podporu nových topológií zo sekcie 3.1.

3.3.1 Definovanie typov nových topológií

Prvým krokom v rozšírení dátovej štruktúry o podporu nových topológii bolo pridanie samotných definícii typov nových topológií. Ako bolo ukázané v sekcii 2.2.4, typy statických topológií sú definované pomocou nasledovných štruktúr:

- štruktúra reprezentujúca typ topológie, ktorá tiež definuje jej dimenziu,
- špecializácie štruktúry `Subtopology`, ktoré pre každú pod-dimenziu definujú typ topológie pod-entity a počet pod-entít,
- špecializácie štruktúry `SubentityVertexMap`, ktoré špecifikujú indexy vrcholov jednotlivých pod-entít všetkých pod-dimenzií.

Keďže topológie mnohouholníka a mnohostenu môžu mať rôzne počty pod-entít, nie je možné ako súčasť ich definícií zahrnúť špecializácie štruktúr `SubentityVertexMap` a špecializácie štruktúr `Subtopology` nemôžu obsahovať informáciu o počte pod-entít (statická členská premenná `count`). Definíciu topológie mnohostenu je možné teda definovať nasledovne:

```
1  struct Polyhedron
2  { static constexpr int dimension = 3; };
3
4  template<>
5  struct Subtopology< Polyhedron, 0 >
6  { typedef Vertex Topology; };
7
8  template<>
9  struct Subtopology< Polyhedron, 1 >
10 { typedef Edge Topology; };
11
12 template<>
13 struct Subtopology< Polyhedron, 2 >
14 { typedef Polygon Topology; };
```

Definícia topológie mnohouholníka je definovaná podobne až na rozdielne údaje.

Boli tiež definované typy topológii ihlanu (pyramid) a klinu (wedge). Keďže sú tieto topológie statické, môžu sa definovať podobne ako to bolo ukázané v sekcii 2.2.4, až na to, že je potrebné ešte pridať informáciu o tom, z koľkých vrcholov pozostávajú jednotlivé pod-entity druhej dimenzie (mnohouholníky). To je kvôli tomu, že medzi 2-dimenzionálne pod-entity ihlanu a klinu patria

trojuholníky a štvoruholníky súčasne. Pri pôvodnej implementácii táto situácia nikdy nestala, pretože daná topológia mala vždy konštantný počet vrcholov. Kvôli tomu boli všetky typy statických topológii rozšírené o štruktúry `SubentityVertexCount`, ktorej primárna šablóna je definovaná nasledujúcim spôsobom:

```
1  template< typename EntityTopology ,
2            typename SubentityTopology ,
3            int SubentityIndex >
4  struct SubentityVertexCount
5  {
6     static constexpr int count =
7                        Subtopology< SubentityTopology , 0 >::count;
8  };
```

Šablónové parametre `EntityTopology` a `SubentityTopology` špecifikujú, o ktorú topológiu a pod-topológiu sa jedná, pričom `SubentityIndex` špecifikuje index pod-entity, pre ktorý sa bude špecifikovať počet vrcholov. Je možné vidieť, že primárna šablóna špecifikuje tento počet ako počet vrcholov danej pod-topológie. To samozrejme funguje pre všetky statické pod-topológie, ale v prípade ihlana a klinu to fungovať nebude pre pod-topológiu mnohouholníka, keďže táto topológia túto informáciu staticky neposkytuje. Kvôli tomu je potrebné túto informáciu definovať pre všetky jednotlivé mnohouholníky ihlanu a klinu v podobe špecializácii tejto štruktúry. Pre topológiu ihlanu sú tieto špecializácie definované nasledujúcim spôsobom:

```
1  template <>
2  struct SubentityVertexCount< Pyramid, Polygon, 0 >
3  { static constexpr int count = 4; };
4
5  template <>
6  struct SubentityVertexCount< Pyramid, Polygon, 1 >
7  { static constexpr int count = 3; };
8
9  template <>
10 struct SubentityVertexCount< Pyramid, Polygon, 2 >
11 { static constexpr int count = 3; };
12
13 template <>
14 struct SubentityVertexCount< Pyramid, Polygon, 3 >
15 { static constexpr int count = 3; };
16
17 template <>
18 struct SubentityVertexCount< Pyramid, Polygon, 4 >
19 { static constexpr int count = 3; };
```

Špecializácie tejto štruktúry v prípade topológie klinu sú definované podobným spôsobom, až na iné údaje.

3.3.2 Rozlišovanie medzi statickými a dynamickými topológiami

Kvôli odlišnostiam medzi statickými a dynamickými topológiami, je potrebné niektoré triedy špecializovať pre prípad, keď je topológia dynamická (t.j. topológia je mnohouholník alebo mnohosten). Príkladom takýchto tried je `MeshSubentityTraits` alebo `SubentityStorageLayer`, ktorých pôvodné definície sú určené výhradne len pre statické topológie, pričom dynamické topológie vyžadujú rozdielnu definíciu. Dá sa to vyriešiť tak, že do definície daných tried sa pridá šablónový parameter reprezentujúci boolovskú hodnotu o tom, či daná topológia je dynamická alebo nie. To znamená, že pôvodná definícia triedy pre statické topológie je definovaná v špecializácii pre hodnotu parametra `false`, pričom definícia triedy pre dynamické topológie je definovaná v špecializácii pre hodnotu parametra `true`. Pre skrátenie definícii šablónových tried, bola táto statická kontrola typu topológie implementovaná do pomocnej štruktúry `IsDynamicTopology<Topology>`. Statický člen štruktúry `value` je nastavený na hodnotu `true` v prípade, že existuje aspoň jedna špecializácia štruktúry `Subtopology<Topology, d>`, kde $0 \leq d < D$, ktorá neobsahuje statickú členskú premennú `count`. To znamená, že počet pod-entít aspoň jednej pod-dimenzie je variabilný, čo v súčasnosti platí len pre topológie mnohouholníka a mnohostenu. Táto statická kontrola typu je docielená pomocou C++ meta-programovacích techník.

Pre ilustráciu, aplikácia tejto techniky na triedu `MeshSubentityTraits` vyzerá nasledovne:

```
1  template< ...,
2            bool IsDynTop = IsDynamicTopology< Topology >::value >
3  class MeshSubentityTraits;
4
5  template< ... >
6  class MeshSubentityTraits< ..., false >
7  { ... };
8
9  template< ... >
10 class MeshSubentityTraits< ..., true >
11 { ... };
```

3.3.3 Definovanie vrstiev pre dynamické topológie

Ako bolo vysvetlené v sekcii 3.2, statické a dynamické topológie sa líšia v reprezentácii štruktúry siete tým, aké dátové štruktúry sú použité pre uloženie pod-entít. Zo sekcie 2.2.3 je známe, že definícia dátových štruktúr pre uloženie pod-entít je definovaná v triede `SubentityStorageLayer`. Keďže je potrebné špecifikovať dve verzie tejto triedy (prvá pre statické topológie a druhá pre dynamické topológie), boli šablónové parametre tejto triedy rozšírené o jeden šablónový parameter, ktorý rozhoduje o tom, či je daná topológia dynamická alebo nie (viď sekciu 3.3.2). Uloženie pod-entít pre statické topológie

vyžaduje maticu o formáte Ellpack, pričom dynamické topológie vyžadujú maticu o formáte SlicedEllpack s poľom dĺžok riadkov matice. Spomenuté dve špecializácie triedy `SubentityStorageLayer` teda vyzerajú v zjednodušenej podobe nasledujúcim spôsobom:

```
1  template< ... >
2  class SubentityStorageLayer< ..., false >
3     : public SubentityStorageLayer< ... >
4  {
5  ...
6     SubentityMatrixType matrix;
7  };
8
9  template< ... >
10 class SubentityStorageLayer< ..., true >
11    : public SubentityStorageLayer< ... >
12 {
13 ...
14    NeighborCountsArray subentitiesCounts;
15    SubentityMatrixType matrix;
16 };
```

Je potrebné upresniť, že samotná trieda `SubentityStorageLayer` nerozhoduje o tom, ktorý formát matice bude použitý pre uloženie pod-entít. Tento údaj je v pôvodnej implementácii špecifikovaný v triede `MeshTraits`. Keďže táto trieda obsahuje pomocné typy týkajúce sa siete ako celku, nie je možné túto triedu špecializovať na základe topológie. Kvôli tomu bola deklarácia typu matice pre uloženie pod-entít presunutá do triedy `MeshEntityTraits`, ktorú je možné špecializovať na základe dimenzie entity, ktorá sa dá skonvertovať do patričnej topológie siete. Boli teda definované nasledovné dve špecializácie triedy `MeshEntityTraits`, ktoré definujú typ matice pod-entít na základe toho, či topológia entity je statická alebo dynamická:

```
1  template< ...,
2            int Dimension >
3  class MeshEntityTraits< ..., Dimension, false >
4  {
5  ...
6     using SubentityMatrixType = Matrices::SparseMatrix
7                                 < ..., EllpackSegments >;
8  };
9
10 template< ...,
11           int Dimension >
12 class MeshEntityTraits< ..., Dimension, true >
13 {
14 ...
15    using SubentityMatrixType = Matrices::SparseMatrix
16                                < ..., SlicedEllpackSegments >;
17 };
```

V sekcii 3.2 bola tiež spomenutá menšia optimalizácia pre uloženie pod-entít mnohouholníkových sietí, ktorá sa týkala toho, že dĺžky riadkov inci-

dentných matíc $I_{2,1}$ a $I_{2,0}$ sú identické. Z toho vyplýva, že postačuje toto pole dĺžok riadkov matice uložiť len pre jednu z dvoch incidentných matíc. Na základe pravidiel statickej konfigurácie zo sekcie 2.2.2 vyplýva, že je vhodnejšie toto pole deklarovať popri matici $I_{2,0}$, pretože táto matica je vždy povinná v prípadoch, keď je povolená aspoň jedna z matíc pod-entít mnohouholníkov. Pre prístup ku jednotlivým dĺžkam riadkov matice pod-entít, obsahuje trieda `SubentityStorageLayer` metódu `getSubentitiesCount`, ktorá má nasledujúcu hlavičku v zjednodušenej podobe:

```
LocalIndex getSubentitiesCount( SubdimensionTag,
                                GlobalIndex entityIndex );
```

Parameter `SubdimensionTag` špecifikuje, pre ktorú pod-dimenziu je daná `getSubentitiesCount` určená, pomocou C++ konceptu preťaženia funkcií. Pre implementovanie spomenutej optimalizácie je potrebné definovať funkciu `getSubentitiesCount`, ktorá obsahuje ako parameter `SubdimensionTag` pre dimenziu 1. Táto inštancia funkcie potom presmeruje požiadavky dĺžok riadkov matice $I_{2,1}$ na dĺžky riadkov matice $I_{2,0}$. Ako bolo ukázané v sekcii 2.2.3, rekurzívna dedičnosť tried `SubentityStorageLayer` prebieha od dimenzie 0 po dimenziu D. Z toho vyplýva, že vrstva pre dimenziu 0 dedí od vrstvy pre dimenziu 1, čo znamená, že pole dĺžok riadkov pre $I_{2,0}$ nie je dostupné, na základe pravidiel C++, pre vrstvu dimenzie 1. Kvôli tomu je potrebné premiestniť definíciu funkcie `getSubentitiesCount`, pre `SubdimensionTag` rovný 1, do vrstvy dimenzie 0 a definíciu tejto funkcie vo vrstve dimenzie 1 je potrebné vynechať. Optimalizácia je teda v zjednodušenej podobe implementovaná nasledujúcim spôsobom:

```
template< ... >
class SubentityStorageLayer< ...,
                             Polygon,
                             DimensionTag<0>,
                             ... >
: public SubentityStorageLayer< ..., Polygon, DimensionTag<1> >
{
...
   LocalIndex getSubentitiesCount( DimensionTag<0>,
                                   const GlobalIndex entityIndex )
   { return subentitiesCounts[ entityIndex ]; }

   LocalIndex getSubentitiesCount( DimensionTag<1>,
                                   const GlobalIndex entityIndex )
   { return getSubentitiesCount( DimensionTag<0>(),
                                 entityIndex ); }
...
   NeighborCountsArray subentitiesCounts;
   SubentityMatrixType matrix;
}

template< ... >
class SubentityStorageLayer< ...,
                             Polygon,
```

```
25                                  DimensionTag<1>,
26                                  ... >
27 : public SubentityStorageLayer< ..., Polygon, DimensionTag<2> >
28 {
29 ...
30    SubentityMatrixType matrix;
31 }
```

3.3.4 Načítanie sieti nových topológii zo súboru

Siete s topológiou bunky mnohouholníka, klinu a ihlana sú podporované formátom VTK. Ako bolo spomenuté v sekcii 2.2.7, knižnica TNL formát VTK už podporuje. Keďže bola implementácia formátu VTK v knižnici TNL dostatočne obecná, rozšírenie podpory pre načítanie sieti týchto topológii vyžadovala len pridanie definícii daných topológii a ostatné drobné úpravy, ktorých detaily nie sú veľmi dôležité.

Čo sa týka mnohostenných sietí, formát VTK síce podporuje ukladanie takýchto sietí, ale nie je to vôbec dokumentované a neboli k dispozícii žiadne vzorové súbory. Kvôli tomu bola pridaná podpora pre formát FPMA, ktorý je súčasťou simulačného balíčka AVL FIRE™ [19]. Jedná sa o formát, ktorého jediným cieľom je reprezentácia mnohostennej siete v súbore, ktorá pozostáva z troch častí:

- zoznam 3D priestorových súradníc vrcholov,
- zoznam indexov vrcholov pre jednotlivé steny (`faceSeeds`),
- zoznam indexov stien pre jednotlivé bunky (`cellSeeds`).

Konkrétnu štruktúru formátu, ilustrovanú na príklade, je možné vidieť v prílohe D.

Hlavným rozdielom v reprezentácii definície mnohostenných sietí a sietí ostatných implementovaných topológii je to, že zárodky buniek (`cellSeeds`) teraz neobsahujú indexy vrcholov, ale indexy stien. Okrem toho je tiež potrebné ešte dodatočne ukladať samotné zárodky stien (`faceSeeds`), ktoré obsahujú indexy vrcholov (podobne ako to je pri `cellSeeds` ostatných topológii). Načítanie a uloženie mnohostenných sieti pomocou formátu FPMA bolo implementované do tried `FPMAReader` a `FPMAWriter`.

Formát FPMA tiež podporuje uloženie rôznych metadát, ktoré sa týkajú danej mnohostennej siete, ale keďže tieto metadáta nemajú v súčasnej implementácií neštruktúrovaných sietí v TNL žiadne použitie, sú pri načítaní siete ignorované. Existuje tiež binárna verzia formátu (formát FPMB) a komprimované verzie formátu (formát FPMAZ, FPMBZ). Keďže tieto formáty sú veľmi nedostatočne dokumentované a vzorové súbory boli dostupné len pre formát FPMA, neboli tieto dodatočne variácie formátu FPMA implementované.

3.3.5 Zmeny a zovšeobecnenia v inicializácii siete

Prvým krokom ku zovšeobecneniu inicializácie siete pre podporu dynamických topológii buniek, je umožnenie variabilnej dĺžky zárodkov pre jednotlivé bunky siete. Ako bolo vysvetlené v sekcii 2.2.6, v pôvodnej implementácii mali všetky zárodky konštantnú dĺžku, čo znamenalo, že nebolo potrebné pre jednotlivé zárodky alokovať veľkosť polí indexov, keďže boli použité statické polia. V prípade dynamických topológii buniek, je potrebné pre každý zárodok alokovať určitú dynamickú veľkosť, čo znamená, že nie je možne použiť polia so statickou veľkosťou. Jednoduchým riešením by mohlo byť to, že namiesto statických polí (`TNL::Containers::staticArray`), by sa použili dynamické polia, ktoré umožňujú pred-alokovať potrebnú veľkosť (`TNL::Containers::Array`). Prvom nevýhodou tohoto riešenia je to, že každý zárodok by vyžadoval vykonať jednu dynamickú alokáciu, čo v C++ predstavuje relatívne pomalú operáciu. Druhom nevýhodou je to, že každý zárodok by predstavoval potenciálne individuálny úsek pamäte, čo by spôsobovalo fragmentáciu pamäte a tiež neefektívne využitie vyrovnávacích pamätí pri iterácii naprieč všetkým zárodkom. Lepším riešením je pre uloženie zárodkov použiť maticu, tzn. tú istú dátovú štruktúru, ktorá je použitá pre uloženie incidentnej matice, do ktorej sú indexy zárodkov pri inicializácii siete skopírované. Výhodou tohoto riešenia nie je len konštantný počet dynamických alokácii a efektívne využitie vyrovnávacej pamäte, ale aj to, že pri inicializácii nie je ani potrebné vykonať hlbokú kópiu matice, ale stačí vykonať len plytkú kópiu pomocou *move* sémantiky, ktorá je podstatne efektívnejšia.

Bola definovaná trieda `EntitySeedMatrix`, ktorá reprezentuje maticu zárodkov a tiež definuje rozhranie pre nastavovanie jednotlivých zárodkov. Pre triedu `EntitySeedMatrix` existujú 2 špecializácie: jedna pre statické a druhá pre dynamické topológie. Typy interných matíc sú identické s typmi matíc pod-entít na základe typu topológie (viď sekciu 3.2). Výhodou triedy `EntitySeedMatrix` je aj to, že v prípade statickej topológie je možné pamäť pre jednotlivé riadky alokovať priamo pri nastavovaní dimenzie matice, čo umožňuje zachovať podobné užívateľské rozhranie ako pri pôvodnej implementácii, kde nebolo potrebné explicitne nastavovať dĺžky zárodkov.

Ďalším krokom bolo zovšeobecnenie triedy `MeshBuilder`, ktorá poskytuje prívetivé užívateľské rozhranie pre vykonanie inicializácie siete (viď sekciu 2.2.6). Okrem toho, že pole zárodkov buniek bolo nahradené maticou zárodkov buniek, bola tiež pridaná matica zárodkov stien, ktorá je vyžadovaná pri inicializácii mnohostenných sietí. Zovšeobecnené užívateľské rozhranie triedy `MeshBuilder` vyzerá teraz nasledovne:

```
1 void setEntitiesCount( const GlobalIndexType& points,
2                        const GlobalIndexType& cells = 0,
3                        const GlobalIndexType& faces = 0 );
4 void setFaceCornersCounts( const NeighborCountsArray& counts );
5 void setCellCornersCounts( const NeighborCountsArray& counts );
6 FaceSeedType getFaceSeed( GlobalIndexType index );
```

```
7 CellSeedType getCellSeed( GlobalIndexType index );
```

Funkcie `setPointsCount` a `setCellsCount` boli odstránené a nahradené funkciou `setEntitiesCount`, ktorá umožňuje nastavenie počtu všetkých typov zárodkov súčasne. Dôvodom je to, že nastavenie dimenzií matíc zárodkov vyžaduje znalosť počtu všetkých typov zárodkov súčasne. V prípade, že by boli jednotlive počty nastavované individuálne, vyžadovalo by to pridanie dodatočnej funkcie, ktorá by vykonala samotné nastavenie dimenzií matíc a musela by sa tiež vykonať validácia, či jednotlivé počty boli predošlo nastavené.

Narozdiel od pôvodnej implementácie, pribudli funkcie pre určenie počtu indexov zárodkov: `setFaceCornersCounts` a `setCellCornersCounts`. Funkcie teda vykonávajú alokáciu pamäte pre riadky matíc zárodkov. Vstupný parameter `counts` reprezentuje pole dĺžok jednotlivých riadkov, kde i-ty element špecifikuje počet indexov i-tého zárodku. V prípade statických topológii je možné tieto dve funkcie ignorovať, keďže dĺžka jednotlivých riadkov matice zárodkov je odvodená od statického údaju o počte vrcholov pre danú topológiu.

Podobne ako v pôvodnej implementácii, funkcia `getCellSeed` a dodatočná funkcia `getFaceSeed`, sprostredkúvajú prístup ku jednotlivým zárodkom, čo umožňuje nastavenie jednotlivých indexov pre daný zárodok. Narozdiel od pôvodnej implementácie, zárodok teraz nepredstavuje referenciu na statické pole, ale objekt novo definovanej triedy `EntitySeedMatrixSeed`. Inštancie tejto triedy reprezentujú proxy objekty, ktoré si interne držia ukazovateľ na riadok matice zárodkov, ktorý patrí danému zárodku. Účelom tejto triedy je zachovať rozhranie, ktoré bolo použité pôvodnou implementáciou, čo minimalizuje potenciálnu refaktorizáciu kódu súvisiacu s triedou `MeshBuilder`, ktorá by musela byť inak vykonaná na mnohých miestach v knižnici TNL. Modifikované užívateľské rozhranie pre inicializáciu siete je demonštrované na inicializácii mnohostennej siete v prílohe C.2.

Ďalším krokom bolo zovšeobecnenie algoritmu, ktorý vykonáva samotnú inicializáciu siete. V prvom rade bolo potrebné rozšíriť parametre počiatočnej funkcie inicializácie o parameter zárodkov stien, ktorý je potrebný pri inicializácii mnohostenných sietí. To znamená, že funkcia `createMesh` triedy `Initializer` má teraz nasledovné rozhranie:

```
1 void createMesh( PointArrayType& points ,
2                  FaceSeedMatrixType& faceSeeds ,
3                  CellSeedMatrixType& cellSeeds ,
4                  MeshType& mesh );
```

V prípade, že je matica `faceSeeds` neprázdna, jedná sa o inicializáciu mnohostenných sietí. Keď je matica naopak prázdna, jedná sa o inicializáciu sieti ostatných topológii a parameter `faceSeeds` je teda ignorovaný.

Ďalším krokom bolo pridanie špecializácii triedy `SubentitySeedsCreator` pre topológie mnohouholníka a mnohostenu. Pôvodná definícia tejto triedy zhotovovala pole zárodkov pod-entít bunky na základe statických údajov o

topológii danej bunky. Keďže tieto údaje sú dostupné len pre bunky so statickou topológiou, je potrebné definovať konkrétne špecializácie tejto triedy pre bunky s dynamickou topológiou:

1. Pre mnohouholníky je potrebné špecifikovať prípad pre 1-dimenzionálne a 0-dimenzionálne pod-entity (hrany a vrcholy). Dvojice indexov pre hrany mnohouholníka je možné jednoducho vytvoriť tak, že sa iterujú indexy vrcholov daného mnohouholníka a každé dva susedné vrcholové indexy reprezentujú hranu, pričom posledný vrchol je spárovaný s prvým vrcholom. Špecializácia pre indexy vrcholov mnohouholníka je triviálna, keďže tieto indexy sú už dostupné zo zárodkov buniek/stien.

2. Čo sa týka mnohostenov, je potrebné opäť špecifikovať prípad len pre 1-dimenzionálne a 0-dimenzionálne pod-entity (hrany a vrcholy). Prípad pre 2-dimenzionálne entity je vynechaný, pretože, ako bude neskôr spomenuté, incidentné matice $I_{3,2}$ a $I_{2,3}$ sú inicializované na základe zárodkov stien (`faceSeeds`) a tým pádom nie je `SubentitySeedsCreator` v ich prípade potrebný. Zárodky hrán a vrcholov pre mnohosteny môžu byť zhotovené tak, že sa zárodky pre hrany a vrcholy všetkých stien mnohostena vložia do množiny a táto množina potom obsahuje výsledný zoznam zárodkov. Dôvodom prečo je potrebné zárodky hrán a vrcholov stien vložiť do množiny je to, že steny mnohostenu môžu zdieľať niektorý hrany a vrcholy, čo by mohlo spôsobiť duplicitné výsledné zarodky.

Bol tiež modifikovaný spôsob, akým sa predávajú výsledné zárodky pod-entít z triedy `SubentitySeedsCreator` volajúcemu. V pôvodnej implementácii boli výsledné zárodky uložené do statického poľa, ktoré sa vrátilo volajúcemu. V prípade dynamických topológii nie je tento spôsob efektívny, keďže by bolo potrebné výsledky zapisovať do dynamického poľa, ktoré by vyžadovalo vykonať dynamickú alokáciu pri každom volaní `SubentitySeedsCreator::create`. Kvôli tomu bol spôsob predávanie výsledku modifikovaný tak, že funkcia teraz vyžaduje dodatočný parameter, reprezentujúci funktor, ktorému sú jednotlivé zárodky predané argumentom. To umožňuje volajúcemu iterovať nad variabilným počtom výsledných zárodkov pod-entít bez nutnosti alokácie dynamického poľa pre uloženie výsledných zárodkov. Táto optimalizácia sa hlavne prejaví v prípade špecializácie triedy `SubentitySeedsCreator` pre mnohosteny, kde je N-krát volaná špecializácia tejto triedy pre mnohouholníky, pričom N je počet stien daného mnohostenu. Funkcia `create` bola kvôli odlišnému predaniu výsledku tiež premenovaná na `iterate`. Do triedy `SubentitySeedsCreator` bola tiež pridaná funkcia `getSubentitiesCount`, ktorá informuje volajúceho o počte výsledných zárodkov pod-entít. To je hlavne dôležité pri alokácii riadkov incidentných matíc, kde v prípade dynamických topológii, majú riadky rôzne dĺžky.

Ďalším krokom bolo zovšeobecnenie šablónových tried `InitializerLayer` a `EntityInitializerLayer`. Hlavná zmena spočívala v tom, že polia dĺžok

riadkov, ktoré boli použité pre alokáciu pamäte riadkov incidentných matíc, sú teraz zhotovované obecnejším spôsobom. Pôvodne, elementy poľa boli nastavené na konštantnú hodnotu, ktorá bola definovaná staticky na základe topológie pod-entity. Teraz sú elementy poľa dĺžok nastavené pomocou zavolania funkcie `SubentitySeedsCreator::getSubentitiesCount`, čo umožňuje nastaviť dĺžku riadka pre individuálne pod-entity.

Čo sa týka inicializácie mnohostenných sieti, bolo potrebné zaručiť správne poradie inicializácie indidečných matíc. To je kvôli tomu, že prvou incidentnou maticou, ktorá je obvykle inicializovaná, je matica $I_{D,0}$, ktorá je priamo mapovaná na indexy zárodkov buniek. Toto poradie inicializácie je správne pre všetky statické topológie a dokonca aj pre mnohouholníkové siete. V prípade mnohostenných sieti, sú zárodky buniek mapované na incidentnú maticu $I_{D,D-1}$ a zárodky stien mapované na incidentnú maticu $I_{D-1,0}$. Matica $I_{D,0}$ môže byť v ich prípade inicializovaná až po inicializácii $I_{D,D-1}$ a $I_{D-1,0}$. V prípade mnohostenných sietí je teda inicializácia $I_{D,0}$ v špecializácii triedy `InitializerLayer` pre dimenziu D vynechaná a odložená na neskôr. Matica $I_{D,0}$ je konkrétne inicializovaná triedou `EntityInitializerLayer` pre pár dimenzií $(D, 0)$, ktorá je volaná špecializáciou `InitializerLayer` pre dimenziu 0. Matica $I_{D-1,0}$ je inicializovaná v špecializácii triedy `InitializerLayer` pre dimenzie d, kde $0 < d < D$. Obvykle je táto matica zhotovená pomocou použitia triedy `SubentitySeedsCreator`, ale v prípade mnohostenných sieti je inicializovaná priamo zo zárodkov stien. Matica $I_{D,D-1}$ je obvykle inicializovaná triedou `EntityInitializerLayer` pre pár dimenzií $(D, D-1)$. Kvôli tomu je potrebné v prípade mnohostenných sietí definovať špecializáciu tejto triedy pre tento pár dimenzii, pričom je daná matica pod-entít inicializovaná priamo zo zárodkov buniek.

Kapitola 4

Testovacie úlohy

Táto kapitola sa zaoberá popisom testovacích úloh, ktoré sú v kapitole 5 použité pre experimentálne meranie výkonnosti implementovaného rozšírenia dátovej štruktúry pre reprezentáciu neštruktúrovaných sietí z kapitoly 3. Budú vysvetlené samotné algoritmy a implementačné detaily daných testovacích úloh. Všetky úlohy sa týkajú sietí s novo implementovanými topológiami buniek: mnohouholníky a mnohosteny. Konkrétne sa jedná o nasledujúce úlohy:

- dekompozícia mnohouholníkovej a mnohostennej siete na trojuholníkovú a štvorstennú sieť, respektívne,
- korekcia nerovinných mnohouholníkov na rovinné mnohouholníky pre mnohouholníkové a mnohostenné siete,
- výpočet miery mnohouholníkovej a mnohostennej siete.

Všetky uvedené algoritmy budú predpokladať, že vstupná sieť vždy pozostáva z konvexných mnohouholníkov/mnohostenov. To je hlavne kvôli tomu, že varianty algoritmov, ktoré sú vhodné aj pre nekonvexný prípad, sú typicky oveľa zložitejšie ako čisto konvexné varianty algoritmov. Keďže hlavným cieľom implementácie algoritmov bolo len otestovanie výkonnosti novo implementovaných topológii, boli kvôli časovému obmedzeniu implementované len tieto jednoduchšie čisto konvexné varianty algoritmov.

Pre úplnosť sú nižšie uvedené definície, ktoré súvisia so špecifickými typmi mnohouholníkov a mnohostenov, pre ktoré sú implementované testovacie úlohy určené.

Definícia 4.1 (jednoduchý mnohouholník)**.** *Jednoduchý mnohouholník* je mnohouholník (viď definíciu 3.1), ktorého hrany sa navzájom nepretínajú a jeho vnútrajšok neobsahuje žiadne diery.

Definícia 4.2 (konvexná množina)**.** *Konvexná množina M* je podmnožina Euklidovského priestoru alebo reálneho afinného priestoru, pre ktorú platí,

že úsečka spájajúca ľubovolné dva body tejto množiny je obsiahnutá v danej množine. Ide teda o takú množinu M, že pre všetky body $A, B \in M$ platí, že $AB \subseteq M$.

Definícia 4.3 (konvexný mnohouholník)**.** *Konvexný mnohouholník* je jednoduchý mnohouholník, ktorého hrany tvoria hranicu konvexnej množiny, čo znamená, že úsečka medzi ľubovoľnými dvomi vrcholmi mnohouholníka je obsiahnutá mnohouholníkom.

Taktiež platí, že všetky vnútorne uhly konvexného mnohouholníka sú konvexné, tj. sú menšie alebo rovné ako 180 stupňov.

Definícia 4.4 (nerovinný mnohouholník)**.** Striktne matematicky sú všetky mnohouholníky rovinné, ale pre účely tejto práce, sa *nerovinným n-uholníkom*, v porovnaní s n-uholníkom (viď definíciu 3.1), myslí len hranica (lomená čiara spájajúca n vrcholov), pričom nie všetky vrcholy sú navzájom koplanárne.

Takéto nerovinné mnohouholníky môžu vznikať v praxi pri vstupných sieťach, ktoré sú určitým spôsobom poškodené, čoho dôsledkom sú napríklad nepresnosti pri aritmetike s plávajúcou rádovou čiarkou. Na túto problematiku naväzuje sekcia 4.2, ktorá popisuje spôsob korekcie takýchto nerovinných mnohouholníkov vo vstupných sieťach na rovinné.

Definícia 4.5 (konvexný mnohosten)**.** *Konvexný mnohosten* je taký mnohosten (viď definíciu 3.2), ktorého vnútorné body tvoria konvexnú množinu.

Tiež platí, že konvexný mnohosten je najmenšou konvexnou množinou, ktorá obsahuje všetky vrcholy mnohostenu a steny konvexného mnohostenu sú konvexné mnohouholníky.

4.1 Dekompozícia

Cieľom tejto úlohy je dekomponovať konvexnú mnohouholníkovú a mnohostennú sieť na trojuholníkovú a štvorstennú sieť, respektívne. Zo vstupnej mnohouholníkovej, resp. mnohostennej siete X, je potrebné vytvoriť výstupnú trojuholníkovú, resp. štvorstennú sieť Y, pričom Y je priestorovo identická s X.

Dekompozícia je vykonávaná pre každú bunku siete nezávislo. To znamená, že každá mnohouholníková/mnohostenná bunka siete X je dekomponovaná na určitú množinu trojuholníkových/štvorstenných entít. Sieť Y je teda zostavená zjednotením jednotlivých dekomponovaných množín trojuholníkových/štvorstenných entít, ktoré sú v Y považované za bunky. Jednotlivé dekomponované entity sú vyjadrené pomocou zoznamu indexov vrcholov (zárodkov buniek), ktoré môžu byť nasledovne použité pre inicializáciu siete Y, podobne ako to bolo popísané v sekcii 2.2.6.

4.1.1 Dekompozícia mnohouholníka

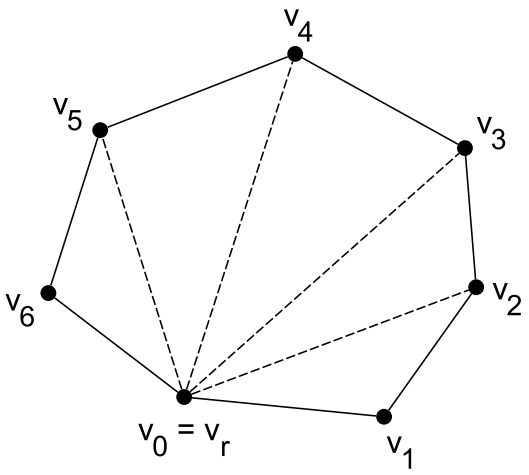
Dekompozícia konvexného n-uholníka Ω, definovaný pomocou zoznamu vrcholov $(v_0, v_1, \ldots, v_{n-1})$, na množinu trojuholníkov $\Omega_\triangle$, môže byť vykonaná niekoľkými spôsobmi, ale pre účely tejto práce boli implementované len nasledovné dva algoritmy:

- vejárová (fan) triangulácia [20],
- ťažisková (centroid) triangulácia [20].

V rámci tejto práce bola v prípade mnohouholníkov vejárová triangulácia označená ako dekompozícia typu P a ťažisková triangulácia označená ako dekompozícia typu C.

4.1.1.1 Vejárová triangulácia (dekompozícia typu P)

Dekompozícia pomocou *vejárovej triangulácie* [20] funguje tak, že sa jeden z vrcholov mnohouholníka Ω označí ako referenčný vrchol v_r (ako napríklad $v_r = v_0$), a vytvoria sa spojnice do všetkých ostatných vrcholov Ω, ktoré nie sú incidetné s dvomi hranami, ktorých je v_r súčasťou. Tento typ dekompozície je ilustrovaný na obrázku 4.1.



Obr. 4.1: Vejárová dekompozícia 7-uholníka

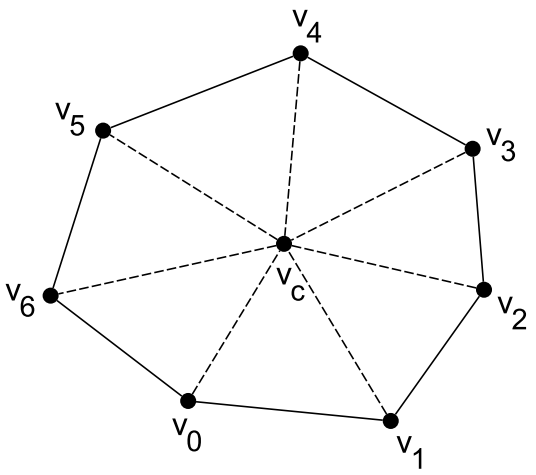
Na základe obrázka 4.1 je možno odvodiť, že keď $v_r = v_0$, tak Ω je možné dekomponovať na množinu trojuholníkov:

$$\Omega_\triangle = \{(v_0, v_i, v_{i+1}), i \in (1, \ldots, n-2)\}$$

Tento algoritmus teda vyprodukuje $n-2$ trojuholníkov a pridá $n-3$ nových hrán.

4.1.1.2 Ťažisková triangulácia (dekompozícia typu C)

Nevýhodou vejárovej triangulácie je to, že môže vyprodukovať celkom nerovnomerné trojuholníky s veľmi ostrými uhlami, čo môže podstatne znížiť numerickú presnosť pri algoritmoch, ako napríklad pri výpočte miery trojuholníka. Túto nevýhodu dokáže vo väčšine prípadoch vyriešiť tzv. *ťažisková triangulácia* [20]. Tento algoritmus sa od vejárovej triangulácie líši tým, že ako v_r sa zvolí ťažisko (centroid) v_c mnohouholníka Ω, namiesto jedného z vrcholov. To umožní vytvoriť rovnomernejšie trojuholníky s menej ostrými uhlami. Tento typ dekompozície je ilustrovaný na obrázku 4.2.



Obr. 4.2: Ťažisková dekompozícia 7-uholníka

Súradnice ťažiska v_c konvexného mnohouholníka Ω, sa triviálne vypočítajú pomocou aritmetického priemeru súradníc jednotlivých vrcholov Ω:

$$(v_c)_i = \frac{1}{n}\sum_{j=0}^{n-1}(v_j)_i$$

Po výpočte v_c sa vytvoria spojnice z v_c do všetkých ostatných vrcholov Ω. Na základe obrázka 4.2 je možno vidieť, že množina dekomponovaných trojuholníkov je teda definovaná ako:

$$\Omega_\triangle = \{(v_c, v_i, v_{(i+1 \mod n)}), i \in (0, 1, \ldots, n-1)\}$$

Tento algoritmus vyprodukuje n trojuholníkov a pridá n nových hrán a 1 nový vrchol. Ťažisková triangulácia, v porovnaní s vejárovou trianguláciou, teda generuje trojuholníkové siete s väčším počtom entít, na úkor zvýšenej numerickej presnosti.

V prípade, že $n = 3$, je ešte možné pridať optimalizáciu, ktorá v tomto prípade preskočí dekompozíciu, keďže entita už je trojuholník. To znamená, že entita samotná je v tomto prípade výsledok dekompozície.

4.1.2 Dekompozícia mnohostenu

Podľa [21], dekompozícia konvexného n-stenu Φ, ktorý je definovaný pomocou zoznamu mnohouholníkových stien $(s_0, s_1, ..., s_{n-1})$, na množinu štvorstenov $\Phi_\triangle$, môže byť vykonaná tak, že sa jednotlivé steny s_i dekomponujú na množinu trojuholníkov $\Omega_{\triangle,s_i}$ (viď sekcia 4.1.1), a z každého vrcholu jednotlivých trojuholníkov sa vytvoria spojnice do určitého vnútorného/hraničného referenčného vrcholu v_R. Dekomponovaná množina štvorstenov $\Phi_\triangle$ je teda definovaná ako:

$$\Phi_\triangle = \{(v_0, v_1, v_2, v_r), (v_0, v_1, v_2) \in \Omega_{\triangle,s_i} \wedge i \in (0, 1, \ldots, n-1)\}$$

Ako je známe zo sekcie 4.1.1, dekompozícia mnohouholníkových stien môže byť vykonaná pomocou dekompozície typu P alebo C, ktoré sa líšia voľbou referenčného vrcholu. Keďže pri dekompozícii mnohostenu Φ je opäť možné ako referenčný vrchol v_R označiť ľubovoľný bod ležiaci vo vnútri alebo na hranici Φ, je teda možné ako v_R označiť buď ľubovoľný vrchol mnohostenu Φ alebo ťažisko v_c. To nám teda dáva 4 možné typy dekompozície mnohostenu Φ, ktoré boli v rámci tejto práce označené ako: CC, CP, PC, PP. Prvé písmeno označuje, či sa jednotlivé vrcholy trojuholníkov dekomponovaných stien spoja s vrcholom mnohostenu (P) alebo s ťažiskom v_c (C). Druhé písmeno označuje typ dekompozície pre mnohouholníkové steny mnohostenu Φ.

V prípade dekompozícií typov PC a PP, pre zjednodušenie implementácie, sa vždy ako v_R volí nultý vrchol mnohostenu v_0, keďže určite platí, že mnohosten má vždy aspoň jeden vrchol. Pri týchto dvoch dekompozíciách tiež môžu vznikať tzv. *degenerované štvorsteny*, ktoré majú nulový objem. To sa stáva vtedy, keď vrchol označený ako v_R je jedným z vrcholov samotnej dekomponovanej steny mnohostenu. V tomto prípade ležia všetky vrcholy štvorstenov na tej istej rovine (predpokladá sa, že steny sú rovinné mnohouholníky), čo spôsobuje generovanie spomenutých degenerovaných štvorstenov. Aby sa tejto situácii vyhlo, je potrebné pred dekomponovaním steny skontrolovať, či v_r nie je jedným z vrcholov danej steny. Ak áno, je dekompozícia danej steny preskočená.

Pri dekompozíciách typu CC a CP, sa ťažisko v_c vypočíta identicky ako v sekcii 4.1.1, tzn. pomocou aritmetického priemeru jednotlivých priestorových súradníc vrcholov mnohostenu.

Čo sa týka množstva entít dekomponovanej siete, je najefektívnejšia dekompozícia typu PP, keďže nepridáva žiadne nové vrcholy a generuje menší počet štvorstenov, kvôli použitej dekompozícii typu P pre steny. Najmenej efektívna je dekompozícia typu CC, ktorá vždy pridáva $n+1$ nových vrcholov (n ťažísk stien a 1 ťažisko mnohostenu) a generuje väčší počet štvorstenov, kvôli použitej dekompozícii typu C pre steny.

Čo sa týka rovnomernosti generovaných štvorstenov, je na tom najlepšie dekompozícia typu CC, kvôli voľbe ťažísk pre referenčné vrcholy stien a mnohostenu. Najhoršie je na tom dekompozícia typu PP, kvôli voľbe vrcholov

na hraniciach pre referenčné vrcholy stien a mnohostenu, čo má typicky za následok generovanie štvorstenov s ostrými uhlami.

4.1.3 Implementácia dekompozície entít

Algoritmy dekompozície mnohouholníkových, resp. mnohostenných entít, boli implementované ako špecializácie šablónovej triedy `EntityDecomposer`, ktorá má nasledovnú hlavičku:

```
1 template< typename MeshConfig,
2           typename Topology,
3           EntityDecomposerVersion EntityVer,
4           EntityDecomposerVersion SubentityVer >
5 struct EntityDecomposer;
```

Parameter `Topology` určuje typ topológie dekomponovanej entity. Tento parameter je použitý pri čiastočnej špecializácii na základe toho, či sa jedná o dekompozíciu mnohouholníka (`Topologies::Polygon`) alebo mnohostenu (`Topologies::Polyhedron`).

Parametre `EntityVer` a `SubentityVer` určujú typ dekompozície na úrovni entity a stien entity, respektívne. Pomocou týchto parametrov sa trieda dodatočne špecializuje na základe toho, o ktorú dekompozíciu sa jedná. Typom týchto parametrov je `EntityDecomposerVersion`, ktorá je definovaná ako enumerácia:

```
1 enum class EntityDecomposerVersion
2 { ConnectEdgesToCentroid, ConnectEdgesToPoint };
```

Hodnota `ConnectEdgesToCentroid` reprezentuje typ dekompozície C a hodnota `ConnectEdgesToPoint` reprezentuje typ dekompozície P. V prípade mnohouholníkov je parameter `SubentityVer` nepoužitý a existuje len kvôli kompatibilite s prípadom mnohostenu, ktorý tento parameter využíva. Je teda možné pomocou týchto dvoch parametrov zvoliť špecifický typ dekompozície daného typu entity (viď sekcie 4.1.1 a 4.1.2).

Rozhranie triedy `EntityDecomposer` pozostáva z týchto dvoch metód:

```
1 static std::pair< GlobalIndexType, GlobalIndexType >
2 getExtraPointsAndEntitiesCount( const MeshEntityType& entity );
3
4 template< typename AddPointFunctor, typename AddCellFunctor >
5 static void
6 decompose( const MeshEntityType& entity,
7              AddPointFunctor&& addPoint,
8              AddCellFunctor&& addEntity );
```

Metóda `getExtraPointsAndEntitiesCount` slúži pre zistenie počtu vygenerovaných entít danej dekomponovanej entity. Prvou položkou je počet vrcholov, ktoré sú vytvorené počas dekompozície. Dôvodom prečo to nie je celkový počet vrcholov vygenerovaných entít je kvôli tomu, že žiadne z implementovaných dekompozícií existujúce vrcholy neodstraňujú. Druhá položka je počet vygenerovaných entít vytvorených počas dekompozície. Údaje o týchto počtoch sú

dôležité hlavne pre účely zistenia celkových počtov entít výslednej dekomponovanej siete, čo súvisí s alokáciou pamäte pre zárodky pri inicializácii.

Metóda `decompose` slúži pre vykonanie samotnej dekompozície danej vstupnej entity. Predanie novo vytvorených entít volajúcemu je docielené pomocou použitia užívateľsky definovaných funktorov, ktoré ako parameter prijímajú výsledne entity. Výhodou funktorov v tomto prípade je to, že predanie výsledku je veľmi flexibilné a je možné túto triedu tak použiť v rôznych situáciách.

Predanie samotných vygenerovaných entít je docielené pomocou funktoru `addEntity`, ktorý je možné napríklad v prípade mnohouholníkovej entity, definovať ako lambda funkciu s hlavičkou:

```
1 auto addEntity = [&] ( GlobalIndexType v0,
2                        GlobalIndexType v1,
3                        GlobalIndexType v2 ) { ... }
```

Pomocou parametrov `v0`, `v1`, `v2` je teda možné predať indexy vrcholov jednotlivých vygenerovaných trojuholníkov.

Keďže počas dekompozície môže nastať situácia, že je potrebné vytvoriť nové vrcholy, ktorých indexy budú použité pre definovanie nových entít, je potrebné, aby užívateľ špecifikoval, aké hodnoty indexov môžu byť použité pre tieto nové vrcholy. To je riešené pomocou funktoru `addPoint`, ktorý môže byť napríklad definovaný pomocou lambda funkcie:

```
1 auto addPoint = [&] ( const PointType& point ) {
2    ...
3    return index; };
```

Pomocou parametra `point` je možné volajúcemu predať priestorové súradnice novo vytvoreného vrcholu. Od volajúceho je požadované, aby následne bol vrátený index pre nový vrchol, ktorý je neskôr použitý pri predaní vygenerovaných entít pomocou funktora `addEntity`.

Algoritmy rôznych typov dekompozícií, zo sekcií 4.1.1 a 4.1.2, sú implementované v jednotlivých špecializáciách triedy `EntityDecomposer`. Stojí tiež za zmienku, že výhodou takého to návrhu bolo aj to, že pri definícii špecializácie pre mnohosteny, bolo možné využiť špecializáciu pre mnohouholníky pre tú časť algoritmu, ktorá dekomponovala steny na trojuholníky a teda bolo možné implementovať dekompozíciu mnohostenu celkom elegantne, tzn. len pridaním výberu referenčného vrcholu mnohostenu a iterácie nad jednotlivými stenami. Pre ilustráciu, telo funkcie *decompose* pre dekompozíciu typu CP a CC je v zjednodušenej podobe definované takto:

```
1 const auto& mesh = entity.getMesh();
2 const auto v3 = addPoint( getEntityCenter(mesh,entity) );
3 auto addFace = [&] ( v0,v1,v2 ) {
4    addEntity( v0,v1,v2,v3 );
5 };
6 const auto facesCount = entity.getSubentitiesCount<2>();
7 for( LocalIndexType i = 0; i < facesCount; i++ ) {
8    const auto faceIdx = entity.getSubentityIndex<2>(i);
```

```
9     const auto face = mesh.getEntity<2>( faceIdx );
10    SubentityDecomposer::decompose( face,addPoint,addFace );
11 }
```

Špecializácia triedy `EntityDecomposer` pre steny je v kóde reprezentovaná pomocou aliasu:

```
1 using SubentityDecomposer = EntityDecomposer< MeshConfig,
2                                               Topologies::Polygon,
3                                               SubentityVer >;
```

To umožňuje použiť tú istú funkciu pre obidva typy dekompozície steny, pomocou šablónového parametru `SubentityVer`.

Je ešte potrebné spomenúť, že výpočet ťažiska (centroidu) entity bolo v implementácii vykonané pomocou šablónovej funkcie `getEntityCenter`, ktorej implementácia už existovala v knižnici TNL.

4.1.4 Implementácia dekompozície siete

Samotná dekompozícia mnohouholníkových, resp. mnohostenných sietí na trojuholníkové, resp. štvorstenné siete, bola implementovaná pomocou funkcie `getDecomposedMesh`. Boli definované dve verzie funkcie: jedna pre mnohouholníkové a druhá pre mnohostenné siete. Hlavičky funkcií vyzerajú v zjednodušenej podobe nasledovne:

```
1 template< EntityDecomposerVersion EntityVer,
2           EntityDecomposerVersion SubentityVer,
3           typename Config,
4           std::enable_if_t< check whether polygon/polyhedron >
5 auto // returns outMesh
6 getDecomposedMesh( const Mesh< Config, Devices::Host >& inMesh );
```

Špecializácie funkcií nie sú v C++ podporované, ale bolo možné dosiahnuť podobného efektu pre špecializáciu na základe topológie, pomocou použitia meta-programovacej štruktúry/techniky `std::enable_if_t` v šablónových parametroch funkcie. To umožnilo definovať samotnú verziu funkcie pre mnohouholníkové a mnohostenné siete.

Funkcia `getDecomposedMesh` vracia objekt typu `Mesh`, ktorý reprezentuje výslednú inicializovanú trojuholníkovú, resp. štvorstennú sieť `outMesh`. Aby sa zachovala statická konfigurácia vstupnej siete `inMesh`, šablónové triedy konfigurácie siete `outMesh` boli definované nasledovne:

```
1 template< typename ParentConfig >
2 struct TriangleConfig: public ParentConfig
3 { using CellTopology = Topologies::Triangle; };
4
5 template< typename ParentConfig >
6 struct TetrahedronConfig: public ParentConfig
7 { using CellTopology = Topologies::Tetrahedron; };
```

Konfigurácia `Config` siete `inMesh` slúži ako rodičovská trieda pre konfiguráciu siete `outMesh`, pričom alias `CellTopology` je pre-definovaný na topológiu mnohouholníka, resp. mnohostenu. Konfigurácie siete `inMesh` a `outMesh` sú teda identické, až na typ topológie bunky.

Šablónové parametre `EntityVer` a `SubentityVer` špecifikujú, ktorý typ dekompozície bude pre bunky siete `inMesh` použitý (viď sekcia 4.1.3).

Sieť `outMesh` sa zostavuje pomocou triedy `MeshBuilder` (viď sekcia 2.2.6), čo znamená, že je pred nastavovaním zárodkov jednotlivých výsledných dekomponovaných entít siete `outMesh`, potrebné alokovať dátové štruktúry pre uloženie vrcholov a zárodkov buniek. Samotný algoritmus dekompozície siete `inMesh` je teda možno zhrnúť do nasledujúcich fáz:

1. Zistenie počtov nových vrcholov a entít pre jednotlivé dekompozície buniek siete `inMesh`.

2. Zistenie celkového počtu vrcholov a buniek siete `outMesh`.

3. Skopírovanie vrcholov siete `inMesh` do siete `outMesh`.

4. Pridanie nových vrcholov a nastavenie indexov zárodkov buniek pre sieť `outMesh`, na základe dekompozície jednotlivých buniek siete `inMesh`.

Pre zistenie počtu výsledných entít pre jednotlivé dekompozície vo fáze č. 1, je použitá metóda `EntityDecomposer::getExtraPointsAndEntitiesCount` (viď sekcia 4.1.3). Jednotlivé dvojice počtov nových vrcholov a entít (a_i, b_i), ktoré vznikli dekompozíciou i-tých buniek siete `outMesh`, sú uložené do pomocného poľa `indices`. Toto pole je dodatočne rozšírené o jednú nulovú dvojicu počtov a následne je na pole vykonaná operácia exkluzívneho prefixového súčtu, pomocou TNL funkcie `inplaceExclusiveScan` (viď sekcia 1.4.4.4). Pole `indices` teraz reprezentuje rozsah indexov (a_i, a_{i+1}) pre nové vrcholy a (b_i, b_{i+1}) pre nové bunky siete `outMesh`, ktoré vznikli dekompozíciou i-tej bunky siete `inMesh`. Inak povedané, dvojice na pozíciach i a $i+1$ poľa `indices`, teraz reprezentujú začiatok a koniec rozsahu indexov, ktoré je možné priradiť novým vrcholom/bunkám siete `outMesh`, ktoré vznikli dekompozíciou i-tej bunky siete `inMesh`.

Celkový počet vrcholov a buniek siete `outMesh` vo fáze č. 2, je možné zistiť tak, že sa prečíta posledná dvojica poľa `indices` (a_n, b_n), kde n je počet buniek siete `inMesh` (pole `indices` bolo rozšírené o jednu nulovú dvojicu práve kvôli tomuto dôvodu). Keďže a_n súvisí s počtom novo pridaných vrcholov, je potrebné ku a_n ešte pripočítať počet vrcholov siete `inMesh`, keďže tieto vrcholy sa dekompozíciou neodstránia.

Dekompozícia buniek siete `inMesh` vo fáze č. 4, je vykoná paralelne tak, že sa vlákna mapujú na dekompozície jednotlivých buniek siete `inMesh`, pričom výsledné nové vrcholy a bunky sa paralelne zapisujú do dátových štruktúr objektu `MeshBuilder`. Vlákno zistí informáciu o tom, na ktoré indexy môže

zapísať nové vrcholy/bunky dekomponovanej i-tej bunky siete `inMesh` tak, že sa pozrie na i-tú dvojicu indexov poľa `indices`. Pole `indices` teda umožňuje paralelné zapisovanie nových vrcholov/buniek do objektu `MeshBuilder`, bez použitia žiadnych atomických operácií, čo podstatne zvyšuje efektivitu paralelizácie dekompozície. Dekompozícia bunky siete `inMesh` je vykonaná pomocou funkcie `EntityDecomposer::decompose` (viď sekcia 4.1.3). Indexy z poľa `indices`, na ktoré sa zapisujú jednotlivé dekomponované vrcholy/bunky, sú priamo zabudované do lambda funkcii, ktoré reprezentujú funktory `addPoint` a `addEntity` pre funkciu `decompose`. To znamená, že objekty reprezentujúce lambdy sa musia vždy inicializovať pred každou dekompozíciou bunky, čo pridáva malú dodatočnú réžiu.

Ná záver sa sieť `outMesh` ešte inicializuje pomocou `MeshBuilder::build`. Keďže inicializácia `outMesh` trvá oveľa dlhšie ako samotná dekompozícia, boli fázy č. 1-4 izolované do funkcie `decomposeMesh`, pre účely experimentálneho merania v kapitole 5. Účelom funkcie `decomposeMesh` je len vytvoriť zárodky pre sieť `outMesh` pomocou triedy `MeshBuilder`. Funkcia `getDecomposedMesh` naviac ešte vykoná inicializáciu siete `outMesh`. Pri meraní v kapitole 5 je teda možné merať len čas behu funkcie `decomposeMesh`, tzn. čas behu algoritmu dekompozície bez časového skreslenia spôsobené inicializáciou siete.

Keďže inicializáciu siete je možné vykonať len na strane CPU, funkcia `getDecomposedMesh` je limitovaná len na vstupné a výstupné siete, ktoré sú na strane CPU. CUDA implementácia dekompozície by teda vyžadovala dodatočný prenos dát medzi CPU a GPU, čo by zaberalo podstatnú časť celkového času výpočtu a podstatne by to znížilo celkové paralelné zrýchlenie algoritmu. Kvôli tomu bola CUDA implementácia dekompozície vynechaná.

4.2 Korekcia nerovinných mnohouholníkov

Cieľom tejto úlohy je zo vstupnej 3D konvexnej mnohouholníkovej/, resp. mnohostennej siete X, vytvoriť výstupnú 3D mnohouholníkovú/, resp. mnohostennú sieť Y, ktorej všetky mnohouholníkové entity sú rovinné. Táto úloha sa netýka 2D mnohouholníkových sietí, keďže vrcholy mnohouholníka v 2D sú vždy koplanárne.

Dôvodom pre implementáciu tejto úlohy, okrem dôvodu pre jej využitie pri experimentálnom meraní v kapitole 5, bolo to, že formát FPMA nešpecifikuje limitáciu na to, že mnohouholníkové steny mnohostenov sú zaručene rovinné, kvôli čomu je možno naraziť v praxi aj na mnohosteny, ktorých steny sú nerovinné mnohouholníky (viď definíciu 4.4) Ako bolo spomenuté, nerovinné steny pri mnohostenoch môžu byť v praxi spôsobené rôznymi nepresnosťami pri výpočtoch, čo znamená, že korekcia nerovinných stien tiež umožní zvýšiť kvalitu vstupnej siete.

Korekcia nerovinných mnohouholníkových entít prebieha tak, že v prípade keď je daná mnohouholníková entita detekovaná ako nerovinná, je daná entita

dekomponovaná na niekoľko trojuholníkových entít (viď sekcia 4.1.1). Entita je následne danými trojuholníkovými entitami nahradená. Keďže pôvodné entity mali topológiu mnohouholníka, budú výsledné trojuholníkové entity tiež mať topológiu mnohouholníka (`Topologies::Polygon`), a nie topológiu trojuholníka (`Topologies::Triangle`), keďže nie všetky mnohouholníky siete X musia byť zaručene počas algoritmu dekomponované. Keďže trojuholníky sú vždy rovinné, bude sieť Y teda obsahovať len rovinné mnohouholníkové entity.

4.2.1 Detekovanie rovinných mnohouholníkových entít

Pre každú mnohouholníkovú entitu Ω vstupnej siete X, je potrebné overiť vlastnosť rovinnosti, aby sa zistilo, ktoré entity sa budú dekomponovať. Je teda pre každú n-uholníkovú entitu Ω potrebné overiť, či všetky jej vrcholy ležia na tej istej rovine. Toto overenie má zmysel len pre $n \geq 4$, keďže pre $n = 3$ sa jedná o trojuholník, ktorého vrcholy sú vždy koplanárne.

Overenie rovinnosti Ω je možné vykonať tak, že sa množina vrcholov $V = \{v_0, v_1, \ldots, v_{n-1}\}$ entity Ω rozdelí na určitý počet podmnožín $V_i \subset V$ obsahujúcich vždy 4 vrcholy, pričom každý vrchol se vyskytuje aspoň v jednej množine V_i. Množina V_i reprezentuje vrcholy i-teho štvorstenu, pre ktorý sa vypočíta jeho objem. V prípade, že objemy všetkých štvorstenov sú nulové, je možné povedať, že entita Ω je rovinná. Inak je entita Ω nerovinná.

Aby sa overila koplanárnosť vrcholov V, je možné jednotlivé štvorsteny V_i zvoliť napríklad nasledujúcim spôsobom:

$$V_i = \{v_0, v_1, v_{i+2}, v_{i+3}\}$$

Počet podmnožín V_i pre n-úholník Ω je teda $n - 3$, tzn. $i \in \{0, 1, \ldots, n - 4\}$. Je si možné všimnúť, že všetky podmnožiny obsahujú prvé dva vrcholy entity Ω: v_0 a v_1. Susedné podmnožiny V_i a V_{i+1} naviac ešte zdieľajú jeden dodatočný vrchol. Podmnožiny V_i a V_{i+1} teda vždy zdieľajú 3 vrcholy, pričom V_{i+1} vždy obsahuje jeden nový vrchol zo zostávajúcich vrcholov entity Ω, čo umožňuje overiť koplanárnosť všetky vrcholov Ω tak, že sa postupne kontroluje komplanárnosť susedných vrcholov v_j a v_{j+1} entity Ω, pre $j \in \{0, 1, \ldots, n-2\}$.

Algoritmus bol implementovaný do funkcie `isPlanar`. Funkcia dodatočne požaduje vstupný parameter `precision`, ktorý špecifikuje reálne číslo presnosti pre porovnávanie objemov štvorstenov s nulou. To je kvôli tomu, že pri výpočte objemu štvorstenov je použitá aritmetika s pohyblivou čiarkou, ktorá nezaručuje úplne presný výsledok objemu. Parameter `precision` umožňuje teda volajúcemu špecifikovať, ako blízko musia byť objemy štvorstenov ku nule, aby boli považované za nulové.

4.2.2 Implementácia rovinnej korekcie siete

Korekcia nerovinných mnohouholníkov pre mnohouholníkové, resp. mnohostenné siete, bola implementovaná do funkcie `getPlanarMesh`. Podobne ako

pri funkcii `getDecomposedMesh`, boli definované dve verzie funkcie: jedna pre mnohouholníkové a druha pre mnohostenné siete. Hlavička funkcie vyzerá v zjednodušenej podobe nasledovne:

```
1 template< EntityDecomposerVersion EntityVer,
2           typename Config,
3           std::enable_if_t< check whether 3D polygon/polyhedron >
4 Mesh< Config, Devices::Host >
5 getPlanarMesh( const Mesh< Config, Devices::Host >& inMesh )
```

Funkcia `getPlanarMesh` vracia objekt typu `Mesh`, reprezentujúci výslednú inicializovanú mnohouholníkovú, resp. mnohostennú sieť `outMesh`. Narozdiel od `getDecomposedMesh`, je tentokrát konfigurácia siete `inMesh` a `outMesh` identická, keďže korekcia nedekomponuje všetky mnohouholníky, ale len tie, ktoré nie sú rovinné.

Šablónový parameter `EntityVer` špecifikuje, ktorý typ dekompozície bude pre mnohouholníkové entity siete `inMesh` použitý (viď sekcia 4.1.3).

Algoritmus korekcie je veľmi podobný s algoritmom dekompozície siete (viď sekcia 4.1.4), až na niekoľko odlišností. Prvým rozdielom je to, že sa dekompozícia vykonáva len pre mnohouholníkové entity. To znamená, že v prípade mnohouholníkových sietí, sa dekomponujú bunky siete, ale v prípade mnohostenných sietí, sa dekomponujú len steny siete, pričom bunky sú zachované. Druhý rozdiel spočíva v tom, že dekompozícia sa vykonáva len v prípade, keď je daná mnohouholníková entita nerovinná. To znamená, že všetky ostatné rovinné mnohouholníkové entity sú zo siete `inMesh` do `outMesh` len skopírované. Posledným rozdielom je to, že sieť `outMesh` teraz obsahuje aj entity s dynamickou topológiou, čo znamená, že je pri zhotovovaní siete, pomocou triedy `MeshBuilder`, tiež potrebné dodatočne špecifikovať dĺžky jednotlivých zárodkov buniek/stien prostredníctvom funkcií `setCellCornersCounts` a `setFaceCornersCounts` (viď sekcia 3.3.5). Samotný algoritmus korekcie nerovinných mnohouholníkov pre sieť `inMesh`, je teda možno zhrnúť do nasledujúcich fáz:

1. Zistenie počtov nových vrcholov a entít pre jednotlivé dekompozície buniek/stien siete `inMesh`, ktoré sa vykonajú len v prípade, že daná entita nie je rovinná.

2. Zistenie celkového počtu vrcholov, buniek a stien (pri mnohostenných sieťach) siete `outMesh`.

3. Skopírovanie vrcholov siete `inMesh` do siete `outMesh`.

4. Nastavenie jednotlivých dĺžok pre zárodky buniek a stien (pri mnohostenných sieťach) siete `outMesh`.

5. Pridanie nových vrcholov a nastavenie indexov zárodkov buniek a stien (pri mnohostenných sieťach) siete `outMesh`, na základe dekompozície jednotlivých nerovinných buniek/stien siete `inMesh`.

Fáza č. 1 prebieha veľmi podobne ako vo funkcii `getDecomposedMesh`, len s tým rozdielom, že sa metóda `getExtraPointsAndEntitiesCount` volá len v prípade, že daná mnohouholníková entita nie je rovinná. Overenie rovinnosti mnohouholníkových entít je vykonané pomocou funkcie `isPlanar` (viď sekcia 4.2.1). V prípade, že je daná entita rovinná, sa na patričnú pozíciu v poli `indices`, uloží dvojica $(0, 1)$. To indikuje, že daná entita vyprodukuje 0 nových vrcholov a len 1 novú entitu (sama seba). Po zhotovení poľa `indices` sa na neho opäť vykoná operácia exkluzívneho prefixového súčtu, ktorá spôsobí to, že pole `indices` bude reprezentovať na i-tej pozícii počiatočné indexy pre rozsahy indexov, ktoré sa môže prideľovať novým vrcholom a entitám, vzniknutých na základe i-tej bunky/steny siete `inMesh`. Výhodou pola `indices` je aj to, že tiež uchováva informáciu o rovinnosti buniek/stien siete `inMesh`, keďže v prípade, že rozsah indexov pre nové entity i-tej bunky/steny je rovný 1, znamená to, že entita sa nedekomponuje a je teda rovinná. To sa využije vo fáze č. 5, kde je vykonávaná samotná dekompozícia nerovinných mnohouholníkových entít, pri ktorej je potrebné znovu pre každú entitu overiť rovinnosť. To teda znamená, že vo fáze č. 5 nie je potrebné vykonávať duplicitné volania funkcie `isPlanar`, ale len výsledok volania zistiť v konštantnom čase pomocou poľa `indices`.

Čo sa týka fázy č. 2, prebieha podobne ako pri funkcii `getDecomposedMesh`, len s tým rozdielom, že pri mnohostenných sieťach, druhá položka posledného elementu poľa `indices` reprezentuje celkový počet stien siete `outMesh`, pričom počet buniek siete `outMesh` je identický s počtom buniek siete `inMesh`.

Čo sa týka fázy č. 4, v prípade mnohouholníkových sietí je potrebné nastaviť dĺžky zárodkov jednotlivých buniek. Indexy buniek pochádzajú z rozsahov indexov poľa `indices`, pričom dĺžky daných buniek sa nastavujú na základe toho, či je rozsah dĺžky jedna alebo dlhší. V prípade, že je rozsah dĺžky jedna, bunka siete `inMesh` je rovinná, čo znamená, že dĺžka zárodku bunky siete `outMesh` je identická s počtom vrcholov danej bunky sieti `inMesh`. V prípade, že je rozsah dlhší ako jedna, bunka siete `inMesh` nie je rovinná, čo znamená, že dĺžky zárodkov buniek siete `outMesh` z daného rozsahu indexov sa nastavia na hodnotu 3 (počet vrcholov trojuholníka). V prípade mnohostenných sietí je potrebné nastaviť dĺžky nielen zárodkov buniek, ale aj stien. Dĺžka zárodku danej bunky siete `outMesh` sa rovná súčtu dĺžok rozsahov indexov každej steny bunky siete `outMesh`, keďže zárodok bunky v tomto prípade reprezentuje indexy stien. Dĺžky zárodkov stien sú nastavené identickým spôsobom, ako to bolo pri nastavovaní dĺžok zárodkov buniek mnohouholníkových sietí.

Fázy č. 5, v prípade mnohouholníkových sietí, je vykonávaná podobné ako fáza č. 4 pri funkcii `getDecomposedMesh`, len s tým rozdielom, že v prípade rovinnej bunky, je daná bunka siete `inMesh` len skopírovaná a dekompozícia sa nevykonáva. Ako už bolo spomenuté, overenie či je daná entita rovinná alebo nie, je teraz vykonávané na základe toho, či je dĺžka rozsahu indexov poľa `indices` rovná jedna alebo dlhšia. V prípade mnohostenných sietí, je potrebné indexy zárodkov buniek siete `outMesh` nastaviť tak, že sa jednotlivé pôvodne

indexy stien siete `inMesh` mapujú na patričné rozsahy indexov poľa `indices`, čo znamená, že sa v prípade nerovinných stien, pôvodný index steny nahradí niekoľkými novými indexmi rovinných trojuholníkov. Indexy zárodkov stien siete `outMesh` sa nastavujú podobným spôsobom, ako to bolo pri nastavovaní indexov zárodkov buniek pri mnohouholníkových sieťach.

Podobne ako to bolo pri funkcii `getDecomposedMesh`, sa na záver sieť `outMesh` inicializuje pomocou `MeshBuilder::build`. Inicializácia bola opäť izolovaná od samotného algoritmu, pre účely experimentálneho merania v kapitole 5. To znamená, že sa fázy č. 1-5 umiestnili do funkcie `planarCorrection`, ktorej účelom je vytvoriť zárodky pre sieť `outMesh` pomocou `MeshBuilder`, pričom funkcia `getPlanarMesh` ešte dodatočne vykonáva inicializáciu siete `outMesh`. To teda umožní merať čas samotného algoritmu (čas funkcie `planarCorrection`) bez časového skreslenia spôsobené inicializáciou siete.

Algoritmus bol implementovaný len pre siete na strane CPU (CUDA implementácia bola vynechaná), z rovnakého dôvodu, ako to bolo pri funkcii `getDecomposedMesh`.

4.3 Miera

Cieľom tejto úlohy je výpočet miery (obsahu/objemu) konvexnej mnohouholníkovej a mnohostennej siete. Keďže jednotlivé bunky siete sa neprekrývajú, je možné mieru siete vypočítať tak, že sa vypočíta miera pre jednotlivé bunky siete a jednotlivé miery sa sčítajú.

4.3.1 Miera mnohouholníka

Algoritmy pre výpočet miery 2D a 3D mnohouholníka boli implementované na základe [22]. Nasledovné popisy algoritmov boli oproti zdroju o trochu zjednodušené, takže úplný detailnejší popis je možné nájsť v [22].

4.3.1.1 2D mnohouholník

Podľa [22], existuje spôsob výpočtu obsahu 2D jednoduchého mnohouholníka, ktorý je založený na dekompozícii mnohouholníka na trojuholníky. Majme n-uholník Ω, ktorý je definovaný množinou vrcholov $V = \{v_0, v_1, \ldots, v_{n-1}\}$ a ďalej definujeme $v_n = v_0$, pričom priestorové súradnice vrcholov označíme ako $v_i = (x_i, y_i)$. Majme tiež ľubovoľný bod P a pre každú hranu $e_i = (v_i, v_{i+1})$ n-uholníka Ω, sa pri dekompozícii vytvorí trojuholník $\triangle_i = (P, v_i, v_{i+1})$. Obsah n-uholníka Ω je potom rovný súčtu obsahov trojuholníkov $\triangle_i$, pričom $i \in \{0, 1, \ldots, n-1\}$. Platí teda:

$$A(\Omega) = \sum_{i=0}^{n-1} A(\triangle_i)$$

Obsah $A(\triangle_i)$ nemusí byť len pozitívny, ale môže byť aj negatívny, čo závisí od toho, či sú vrcholy Ω orientované v smere hodinovej ručičky alebo proti smeru hodinovej ručičky. Pre prípad Ω, orientovaného proti smeru hodinovej ručičky, je $A(\triangle_i)$ pozitívny, ak sa bod P nachádza naľavo od hrany e_i; na druhej strane, $A(\triangle_i)$ je negatívny, ak sa bod P nachádza napravo od hrany e_i. V prípade, že je Ω orientovaný v smere hodinových ručičiek, sú znamienka obrátené.

Bod P tiež ani nemusí ležať vo vnútri Ω, čo znamená, že plocha trojuholníkov $\triangle_i$ môže čiastočne ležať mimo Ω. Ale vďaka tomu, že obsahy $A(\triangle_i)$ môžu byť negatívne/pozitívne, sa tie časti plochy trojuholníkov, ktoré ležia mimo Ω, vykrátia s obsahmi trojuholníkov opačného znamienka.

Keďže bod P môže byť ľubovoľný bod, [22] tiež spomína, že keď sa zvolí $P = (0, 0)$, tak je možné formulu pre obsah trojuholníka $\triangle_i$ zredukovať na $2A(\triangle_i) = (x_i y_{i+1} - x_{i+1} y_i)$. To nám pre obsah n-uholníka Ω dáva:

$$\begin{aligned} A(\Omega) &= \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \\ &= \sum_{i=0}^{n-1} (x_i + x_{i+1})(y_{i+1} - y_i) \\ &= \sum_{i=1}^{n} x_i (y_{i+1} - y_{i-1}) \end{aligned}$$

kde $v_i = (x_i, y_i)$ a pre indexy je použitá modulárna aritmetika, tj. pre $i \geq n$ platí $x_i = x_{i \mod n}$ a obdobne pre y_i.

Najstručnejšia posledná suma používa pre výpočet obsahu n-uholíka Ω len n násobení a $(2n - 1)$ sčítaní, kvôli čomu je oproti ostatným dvom z dôvodu efektivity preferovaná. Modulu vo výpočte indexu vrcholu sa je možné vyhnúť tak, že sa posledné iterácie umiestnenia mimo sumu.

Tento algoritmus bol implementovaný funkciou `getPolygon2DArea`, ktorá tiež umožňuje špecifikovať pomocou šablónových parametrov `Coord1` a `Coord2`, aké dva súradnice budú pre dané vrcholy mnohouholníka použité pri výpočte obsahu. To je využité pri výpočte obsahu 3D mnohouholníka v sekcii 4.3.1.2, ktorý len rozširuje výpočet 2D mnohouholníka o niektoré dodatočné operácie. Samotný výpočet obsahu 2D mnohouholníkovej entity je realizovaný pomocou preťaženia funkcie `getEntityMeasure` pre 2D mnohouholníky, ktorá zavolá funkciu `getPolygon2DArea` a špecifikuje jediné dva súradnice (x a y).

4.3.1.2 3D rovinný mnohouholník

V [22], je tiež prezentovaný spôsob výpočtu obsahu 3D rovinného n-uholníka Ω, ktorý využíva formulu pre obsah 2D mnohouholníka zo sekcie 4.3.1.1, pričom obsah n-uholníka Ω je možné obdržať pomocou škálovania 2D obsahu určitým faktorom. Táto metóda premietne 3D mnohouholník na osovo zarovnanú 2D nadrovinu, ignorovaním jedných z troch priestorových súradníc.

Konkrétne, keď máme 3D súradnice (x, y, z) pre štandardnú xyz bázu, tak projekcia, ktorá ignoruje súradnice x, y, z, vykoná premietnutie na nadroviny yz, zx, xy respektívne. Teda máme projekcie: $\mathrm{Proj}_x(x, y, z) = (y, z)$, $\mathrm{Proj}_y(x, y, z) = (x, z)$, $\mathrm{Proj}_z(x, y, z) = (x, y)$.

Zvolí sa jedna z troch projekcií, ktorá sa najlepšie vyhýba degenerácii a optimalizuje robustnosť tak, že sa pozrieme na normálový vektor $\mathbf{n} = (n_x, n_y, n_z)$ mnohouholníka Ω a pre ignorovanie sa zvolí súradnica c, pre ktorú má n_c najväčšiu absolútnu hodnotu. Projekcia, ktorá ignoruje zvolenú súradnicu c, je označená ako Proj_c. Pre pomer obsahov premietnutého n-uholníka $\mathrm{Proj}_c(\Omega)$ a n-uholníka Ω s normálovým vektorom $\mathbf{n}$ platí:

$$\frac{A(\mathrm{Proj}_c(\Omega))}{A(\Omega)} = \frac{n_c}{|\mathbf{n}|}$$

Pre obsah n-uholníka Ω teda platí:

$$A(\Omega) = A(\mathrm{Proj}_c(\Omega))\frac{|\mathbf{n}|}{n_c}$$

Pre výpočet obsahu 3D rovinného mnohouholníka, je teda v porovnaní s výpočtom 2D obsahu zo sekcie 4.3.1.1, potrebné vykonať jedno dodatočné násobenie, výpočet normálového vektoru a jeho dĺžky, a zvolenie ignorovanej súradnice.

Výpočet obsahu 3D mnohouholníkovej entity bol implementovaný pomocou preťaženia funkcie `getEntityMeasure` pre 3D mnohouholníky. Výpočet 2D obsahu mnohouholníka pre projekciu $\mathrm{Proj}_c(\Omega)$ je realizovaný pomocou volania funkcie `getPolygon2DArea` (viď sekcia 4.3.1.1), ktorá umožňuje pomocou šablónových parametrov špecifikovať, ktoré dva súradnice budú pri výpočte použité, čo realizuje ignorovanie danej súradnice c.

4.3.2 Miera mnohostenu

Výpočet miery, resp. objemu, konvexného mnohostenu Φ, môže byť triviálne vykonaná dekompozíciou Φ na množinu štvorstenov $\Phi_\triangle = \{t_0, t_1, \ldots, t_{|\Phi_\triangle|-1}\}$ a sčítaním jednotlivých objemov štvorstenov $V(t_i)$, kde $i \in \{0, 1, \ldots, |\Phi_\triangle|-1\}$. Pre objem Φ teda platí:

$$V(\Phi) = \sum_{i=0}^{|\Phi_\triangle|-1} V(t_i)$$

Pre výpočet objemu štvorstenu t_i s vrcholmi v_0, v_1, v_2 a v_3 platí:

$$V(t_i) = \frac{1}{6}\det(v_3 - v_0, v_2 - v_0, v_1 - v_0)$$

kde $\det(v_3 - v_0, v_2 - v_0, v_1 - v_0)$ reprezentuje determinant 3×3 matice, pričom vektor $v_3 - v_0$ je 1. stĺpec, $v_2 - v_0$ je 2. stĺpec a $v_1 - v_0$ je 3. stĺpec tejto matice.

Výpočet objemu štvorstenu je už v TNL implementovaný pomocou funkcie `getTetrahedronVolume`.

Ako bolo spomenuté v sekcii 4.1.2, dekompozícia mnohostenu na množinu štvorstenov môže byť vykonaná pomocou jedného zo štyroch implementovaných typov dekompozícií mnohostenu: CC, CP, PC, PP. Keďže typ dekompozície PP nevykonáva žiadne dodatočné výpočty súvisiace s ťažiskom mnohouholníkových stien a mnohostenu, bol tento typ dekompozície zvolený pre účely výpočtu objemu mnohostenu. Ostatné typy dekompozície, hlavne typ CC, môžu síce dosahovať lepšej numerickej presnosti výpočtu objemu, ale vyžadujú dodatočné výpočty ťažísk, čo by výrazne spomalilo celkový výpočet v porovnaní s typom PP. Typ PP je oproti ostatným tiež implementačné oveľa jednoduchší, čo má aj pozitívny vplyv pre výpočet objemu mnohostenu na GPU.

Výpočet objemu pre mnohostenné entity siete bol, podobne ako v prípade mnohouholníkových entít v predošlej sekcii 4.3.1, implementovaný preťažením funkcie `getEntityMeasure` pre mnohostenné entity.

4.3.3 Implementácia miery siete

Ako už bolo spomenuté, celková miera siete sa vypočíta ako súčet mier jednotlivých buniek siete. Výpočet mier všetkých buniek je možné paralelne vykonať pomocou paralelného for cyklu (viď sekcia 1.4.4.1), pričom i-ta iterácia vykonáva výpočet miery i-tej bunky siete. Jednotlivé výsledky mier buniek sa uložia do dočasného poľa `tmp`. Špecifikovanie výpočtu i-tej iterácie pre paralelný for cyklus je možné realizovať rozšírenou lambda funkciou, ktorá môže vyzerať nasledovne:

```
1 auto kernel_measures = [] __cuda_callable__
2          ( GlobalIndex i, const Mesh& mesh, Real* tmp )
3 {
4    constexpr int cellDim = Mesh::getMeshDimension();
5    const auto& cell = mesh.template getEntity< cellDim >( i );
6    tmp[ i ] = getEntityMeasure( mesh, cell );
7 };
```

Keďže hlavným účelom tejto testovacej úlohy je otestovať výkonnosť funkcie `getEntityMeasure` naprieč všetkým bunkám siete paralelne, je posledný krok sčítania jednotlivých medzi-výsledkov pre obsiahnutie celkovej miery siete, vynechaný.

Ale v prípade, že by nás celková miera siete zaujímala, je možné ju obdržať použitím operácie paralelnej redukcie (viď sekcia 1.4.4.3). Paralelná redukcia by tiež úplne nahradila paralelný for cyklus a dočasné pole `tmp`, keďže je možné špecifikovať i-tu redukovanú mieru pomocou lambdy `fetch`, ktorá by mohla vyzerať nasledovne:

```
auto fetch = [=] __cuda_callable__
         ( GlobalIndex i )
{
   constexpr int cellDim = Mesh::getMeshDimension();
   const auto& cell = meshPtr->template getEntity< cellDim >( i );
   return getEntityMeasure( *meshPtr, cell );
};
```

Pre skombinovanie jednotlivých mier by sa pre operáciu redukcie definovala tiež lambda `reduction`, ktorá by vyzerala takto:

```
auto reduction = [] __cuda_callable__
         ( const Real& a, const Real& b )
{ return a + b; };
```

Kapitola 5

Testovanie a experimentálna analýza

Táto kapitola sa zaoberá testovaním a analýzou výkonu implementovaného rozšírenia dátovej štruktúry pre neštruktúrované numerické siete (viď kapitolu 3). Najprv je uvedené, akým spôsobom bola testovaná korektnosť implementácie. To je nasledované popisom metodiky, ktorým bolo vykonané meranie. Meranie začína skúmaním toho, ako je dátová štruktúra pamäťovo náročná pre rôzne veľkosti sietí. Potom sa skúma čas vykonania rôznych podstatných režijných operácií, ktoré sú často nevyhnutné pred samotným použitím dátovej štruktúry pre výpočet. Na záver sa skúma efektivita dátovej štruktúry pomocou niekoľkých testovacích úloh (viď kapitolu 4). Implementované rozšírenie je tiež výkonnostne porovnané s pôvodnou implementáciou z hľadiska množstva pamäte, času výpočtu a paralelnej efektivity.

5.1 Testovanie

Unit testy pre overenie korektnosti implementácie boli implementované pomocou knižnice *GoogleTest* [23], ktorá je v TNL extenzívne používaná. Testy boli zakomponované do TNL, čo znamená, že je ich možné vykonať ako súčasť zostavenia knižnice.

Pre novo implementované topológie buniek siete z kapitoly 3, boli implementované nasledovné unit testy:

```
1 TEST( MeshTest, SevenPolygonsTest );
2 TEST( MeshTest, TwoWedgesTest );
3 TEST( MeshTest, TwoPyramidsTest );
4 TEST( MeshTest, TwoPolyhedronsTest )
```

Tieto testy zostavia jednoduchú sieť pomocou triedy `MeshBuilder` a následne skontrolujú presné obsahy jednotlivých incidentných matíc, duálneho grafu a značiek entít. Tieto testy teda testujú správnosť dátovej štruktúry a implementovaného algoritmu pre inicializáciu siete (viď sekcie 2.2.6 a 3.3.5).

Pre algoritmy z kapitoly 4, boli implementované nasledujúce unit testy:

```
1 TEST( MeshGeometryTest, Polygon2DAreaTest );
2 TEST( MeshGeometryTest, Polygon3DAreaTest );
3 TEST( MeshGeometryTest, PolyhedronAreaTest );
4 TEST( MeshGeometryTest, Polygon3DIsPlanarTest );
```

Prvé tri testy overujú správnosť implementácie algoritmov pre výpočet miery (viď sekciu 4.3) tak, že vypočítaná miera je porovnaná s dopredu známou mierov jednoduchých, ale netriviálnych entít. Štvrtý test overuje správnosť funkcie pre detekciu rovinnosti mnohouholníka (viď sekciu 4.2.1) tak, že sa pre menší rovinný mnohouholník postupne aplikuje určitá odchýlka na priestorové súradnice jednotlivých vrcholov a overí sa, že je mnohouholník nerovinný.

Pre overenie správnosti algoritmov dekompozície a korekcie nerovinných mnohouholníkov (viď sekcie 4.1 a 4.2) sa implementovali nástroje, ktoré na sieť zo vstupného súboru aplikujú daný algoritmus a zapíšu výstupnú sieť do súboru. Potom bola vykonaná len manuálna inšpekcia korektnosti výstupných sietí pomocou nástroja pre vizualizáciu sietí. Správnosť bola overená manuálne z toho dôvodu, že presný obsah incidentných matíc výstupných sietí nie je kvôli dekompozícii triviálne predvídateľný, kvôli čomu by bolo overenie presného obsah matíc veľmi komplikované.

5.2 Metológia merania

Množstvo pamäte, ktoré zaberá dátová štruktúra, je merané pomocou rozdielu celkovej virtuálnej pamäte procesu po a pred vytvorením objektu siete (inštancia triedy `Mesh<Config,Device>`). Tento spôsob nemusí byť sám o sebe veľmi presný kvôli tomu, že sa virtuálna pamäť alokuje po stránkach. Pre zvýšenie presnosti merania je vykonaných niekoľko meraní po sebe (maximálne 10 v prípade dostatku pamäte), ktoré sa nakoniec spriemerujú.

Čas výpočtu meraných režijných operácií a testovacích úloh je meraný pomocou TNL funkcie `Benchmark::time` [11], ktorá umožňuje meranie opakovať určitý počet krát na základe vstupného argumentu. Časy jednotlivých meraní sa nakoniec spriemerujú. Kvôli časovému obmedzeniu boli úlohy bežiace na CPU opakované 10-krát a úlohy bežiace na GPU opakované 20-krát.

Porovnanie efektivity s pôvodnou implementáciou bolo vykonané pomocou prirovnania množstva pamäte a časov výpočtu meraných operácií mnohouholníkových a mnohostenných sietí ku ich dekomponovaným variantám, tj. ku trojuholníkovým a štvorstenným sietiam.

Merania prebehli pre každú sieť v minimálnej a maximálne konfigurácii, označené ako **Min. konf.** a **Max. konf.**. Minimálna konfigurácia zakazuje položky dátovej štruktúry, ktoré sú pre merané testy nepotrebné. Na druhej strane maximálna konfigurácia povoľuje všetky možné položky, ktoré dátová štruktúra môže obsahovať. Obidve konfigurácie používajú `float` pre parameter `RealType`, `int` pre parameter `GlobalIndexType` a `short int` pre parame-

ter `LocalIndexType`. Definície týchto dvoch konfigurácií je možné detailnejšie preskúmať v prílohe B.

Pre testovacie úlohy bežiace na GPU sa tiež predpokladalo, že sieť sa už nachádzala v pamäti GPU, čo znamená, že sa ignoroval čas prenosu siete medzi pamäťou CPU a GPU.

5.2.1 Dáta

Pre analýzu výkonnosti mnohouholníkových sietí bola použitá sieť jednotkového štvorca rôznych granularít, značených ako $2D_i^{\square}$, kde $i \in \{0, 1, 2, 3, 4\}$. Za účelom porovnania výkonu s pôvodnou implementáciou boli použité trojuholníkové siete, značené ako $2D_i^{\triangle X}$, pričom sieť $2D_i^{\triangle X}$ vznikla dekomponovaním siete $2D_i^{\square}$ pomocou mnohouholníkovej dekompozície typu X (viď sekciu 4.1.1). Vlastnosti týchto sietí popisuje tabuľka 5.1.

Id.	#V	#F	#C
$2D_0^{\square}$	702	1051	350
$2D_0^{\triangle C}$	1050	3049	2000
$2D_0^{\triangle P}$	702	2005	1304
$2D_1^{\square}$	2440	3658	1219
$2D_1^{\triangle C}$	3651	10751	7101
$2D_1^{\triangle P}$	2440	7118	4679
$2D_2^{\square}$	9018	13527	4510
$2D_2^{\triangle C}$	13474	40001	26528
$2D_2^{\triangle P}$	9018	26633	17616
$2D_3^{\square}$	34350	51556	17207
$2D_3^{\triangle C}$	51364	153263	101900
$2D_3^{\triangle P}$	34350	102221	67872
$2D_4^{\square}$	134159	201258	67100
$2D_4^{\triangle C}$	200531	600022	399492
$2D_4^{\triangle P}$	134159	400906	266748

Tabuľka 5.1: Vlastnosti mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle X}$) sietí použitých pri meraní: počet vrcholov #V, počet stien #F, počet buniek #C

Pre analýzu výkonnosti mnohostenných sietí bola použitá sieť jednotkovej kocky rôznych granularít, značených ako $3D_i^{\square}$, kde $i \in \{0, 1, 2, 3, 4, 5\}$. Pre porovnanie výkonu s pôvodnou implementáciou boli použité štvorstenné siete, značené ako $3D_i^{\triangle XY}$, pričom sieť $3D_i^{\triangle XY}$ vznikla dekomponovaním siete $3D_i^{\square}$ pomocou mnohostennej dekompozície typu XY (viď sekciu 4.1.2). Vlastnosti týchto sietí popisuje tabuľka 5.2.

Id.	#V	#F	#C
$3D_0^{\square}$	404	452	63
$3D_0^{\triangle CC}$	1221	9480	3792
$3D_0^{\triangle CP}$	467	4756	2284
$3D_0^{\triangle PC}$	969	7314	2817
$3D_0^{\triangle PP}$	404	3728	1687
$3D_1^{\square}$	2358	2690	395
$3D_1^{\triangle CC}$	7454	59041	23618
$3D_1^{\triangle CP}$	2753	29316	14216
$3D_1^{\triangle PC}$	5874	45201	17390
$3D_1^{\triangle PP}$	2358	22652	10358
$3D_2^{\square}$	4905	5666	824
$3D_2^{\triangle CC}$	16290	134033	53614
$3D_2^{\triangle CP}$	5729	66066	32492
$3D_2^{\triangle PC}$	12994	105183	40604
$3D_2^{\triangle PP}$	4905	52566	24426
$3D_3^{\square}$	33572	39177	5740
$3D_3^{\triangle CC}$	115362	969514	387812
$3D_3^{\triangle CP}$	39312	475059	235712
$3D_3^{\triangle PC}$	92406	768991	297335
$3D_3^{\triangle PP}$	33572	384262	179667
$3D_4^{\square}$	254443	297457	43293
$3D_4^{\triangle CC}$	884760	7510116	3004092
$3D_4^{\triangle CP}$	297736	3673621	1830044
$3D_4^{\triangle PC}$	711623	5995668	2320178
$3D_4^{\triangle PP}$	254443	2999395	1405818
$3D_5^{\square}$	1992685	2328726	336608
$3D_5^{\triangle CC}$	6957178	59338473	23735718
$3D_5^{\triangle CP}$	2329293	29012036	14479948
$3D_5^{\triangle PC}$	5610948	47549017	18409693
$3D_5^{\triangle PP}$	1992685	23824648	11173167

Tabuľka 5.2: Vlastnosti mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle XY}$) sietí použitých pri meraní: počet vrcholov #V, počet stien #F, počet buniek #C

5.2.2 Platforma

Meranie bolo vykonané na systéme **gp14**, ktorý patrí Fakulte Jadernej a Fyzikálne Inženýrskej, ČVUT v Prahe. Parametre systému sú popísané v tabuľke 5.3. Zdrojový kód bol skompilovaný pomocou TNL **install** skriptu s **Release** konfiguráciou, ktorá používa g++ príznaky: **-O3**, **-march=native**, **-mtune=native** a **-std=c++14**.

Operačný systém:	Arch Linux 5.13.13
g++:	11.1.0
nvcc:	11.4
CPU:	Intel Core i7-9800X
GPU:	GeForce RTX 2070 SUPER
Veľkosť pamäte:	64 GB
Priepustnosť pamäte:	4× 21,3 GB/s

Tabuľka 5.3: Parametre systému použitého pri meraní

5.3 Meranie

5.3.1 Pamäťová náročnosť

Táto sekcia sa zaoberá meraním množstva pamäte, ktorú zaberá objekt siete.

	Min. konf.		Max. konf.	
Id.	Pamäť	$Fa_{\square}$	Pamäť	$Fa_{\square}$
$2D_0^{\square}$	0,06	1,00	0,09	1,00
$2D_0^{\triangle C}$	0,12	2,00	0,22	2,44
$2D_0^{\triangle P}$	0,10	1,66	0,16	1,77
$2D_1^{\square}$	0,21	1,00	0,31	1,00
$2D_1^{\triangle C}$	0,48	2,29	0,78	2,52
$2D_1^{\triangle P}$	0,35	1,67	0,58	1,87
$2D_2^{\square}$	0,78	1,00	1,19	1,00
$2D_2^{\triangle C}$	1,79	2,29	2,92	2,45
$2D_2^{\triangle P}$	1,32	1,69	2,21	1,86
$2D_3^{\square}$	2,97	1,00	4,56	1,00
$2D_3^{\triangle C}$	6,88	2,31	11,21	2,46
$2D_3^{\triangle P}$	5,09	1,71	8,49	1,86
$2D_4^{\square}$	11,66	1,00	17,86	1,00
$2D_4^{\triangle C}$	26,91	2,31	43,85	2,46
$2D_4^{\triangle P}$	20,00	1,72	33,35	1,87

Tabuľka 5.4: Porovnanie pamäťovej náročnosti Pamäť [MB] mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle X}$) sietí. $Fa_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle C}$, $2D_i^{\triangle P}$

Z Tabuľky 5.4 je vidieť, že mnohouholníkové siete pri maximálnej konfigurácii zaberajú približne o 1,5-krát viac pamäte ako pri minimálnej konfigurácii. Dekomponované trojuholníkové siete zaberajú v najlepšom prípade o približne 1,7-krát viac pamäte ako originálne mnohouholníkové siete.

Čo sa týka mnohostenných sietí, je z tabuľky 5.5 vidieť, že pri maximálnej konfigurácii zaberajú o približne 3-krát viac pamäte ako pri minimálnej kon-

figurácii. Dekomponované štvorstenné siete zaberajú v najlepšom prípade o približne 6-krát viac pamäte ako originálne mnohostenné siete. V porovnaní s mnohouholníkovými sieťami, má typ použitej dekompozície oveľa väčší vplyv na množstvo pamäte. Je vidieť, že najefektívnejšia dekompozícia typu PP vyprodukuje približne o 2,7-krát menšie siete ako najmenej efektívna dekompozícia typu CC.

	Min. konf.		Max. konf.	
Id.	Pamäť	$Fa_{\square}$	Pamäť	$Fa_{\square}$
$3D_0^{\square}$	0,04	1,00	0,11	1,00
$3D_0^{\triangle CC}$	0,61	15,25	1,88	17,09
$3D_0^{\triangle CP}$	0,25	6,25	0,78	7,09
$3D_0^{\triangle PC}$	0,41	10,25	1,26	11,45
$3D_0^{\triangle PP}$	0,22	5,50	0,69	6,27
$3D_1^{\square}$	0,27	1,00	0,71	1,00
$3D_1^{\triangle CC}$	3,91	14,48	11,88	16,73
$3D_1^{\triangle CP}$	1,56	5,77	4,76	6,70
$3D_1^{\triangle PC}$	2,59	9,59	8,03	11,31
$3D_1^{\triangle PP}$	1,40	5,19	4,29	6,04
$3D_2^{\square}$	0,57	1,00	1,49	1,00
$3D_2^{\triangle CC}$	9,01	15,81	27,18	18,24
$3D_2^{\triangle CP}$	3,47	6,09	10,43	7,00
$3D_2^{\triangle PC}$	6,04	10,60	18,59	12,48
$3D_2^{\triangle PP}$	3,25	5,70	9,85	6,61
$3D_3^{\square}$	4,03	1,00	10,42	1,00
$3D_3^{\triangle CC}$	65,69	16,30	197,41	18,95
$3D_3^{\triangle CP}$	24,85	6,17	73,64	7,07
$3D_3^{\triangle PC}$	44,43	11,02	136,33	13,08
$3D_3^{\triangle PP}$	23,97	5,95	72,17	6,93
$3D_4^{\square}$	30,80	1,00	79,46	1,00
$3D_4^{\triangle CC}$	511,87	16,62	1 536,34	19,33
$3D_4^{\triangle CP}$	192,05	6,24	567,00	7,14
$3D_4^{\triangle PC}$	347,64	11,29	1 065,74	13,41
$3D_4^{\triangle PP}$	187,89	6,10	564,97	7,11
$3D_5^{\square}$	241,93	1,00	623,69	1,00
$3D_5^{\triangle CC}$	4 058,46	16,78	12 177,53	19,52
$3D_5^{\triangle CP}$	1 516,59	6,27	4 472,09	7,17
$3D_5^{\triangle PC}$	2 757,41	11,40	8 450,42	13,55
$3D_5^{\triangle PP}$	1 492,10	6,17	4 485,24	7,19

Tabuľka 5.5: Porovnanie pamäťovej náročnosti Pamäť [MB] mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle XY}$) sietí. $Fa_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle CC}$, $3D_i^{\triangle CP}$, $3D_i^{\triangle PC}$, $3D_i^{\triangle PP}$

5.3.2 Načítanie zo súboru

Táto sekcia sa zaoberá meraním času načítania siete zo súboru. Je dôležité upresniť, že táto operácia vykonáva len parsovanie vstupného súboru, čo znamená, že výsledok operácie nie je objekt siete. Pre vytvorenie objektu siete je ešte potrebné vykonať inicializáciu siete, ktorá je analyzovaná v sekcii 5.3.3.

Z tabuľky 5.6 je vidieť, že načítanie mnohouholníkových sietí je v najhoršom prípade o približne 1,3-krát rýchlejšie ako načítanie dekomponovaných trojuholníkových sietí. To sa dá vysvetliť tým, že dekomponované siete obsahujú viacero buniek a vrcholov (pri $2D_i^{\triangle C}$), čo zvyšuje veľkosť súboru.

Čo sa týka mnohostenných sietí, je z tabuľky 5.7 vidieť, že načítanie mnohostenných sietí je v najhoršom prípade približne 2-krát pomalšie ako načítanie dekompovaných štvorstenných sietí. Dôvodom je to, že načítanie mnohostenných sietí vyžaduje naviac ešte parsovanie zárodkov pre steny, čo sa pri štvorstenných sieťach nevyskytuje. Ďalším dôvodom je aj to, že formát vstupných súborov FPMA neposkytuje celkové počty indexov zárodkov pre bunky a steny, čo znižuje efektivitu parsovania kvôli tomu, že sa polia, do ktorých sa indexy ukladajú, nemôžu predbežne alokovať s presne potrebnou veľkosťou.

Id.	*CT*	$Sp_{\square}$
$2D_0^{\square}$	0,16	1,00
$2D_0^{\triangle C}$	0,30	0,54
$2D_0^{\triangle P}$	0,21	0,75
$2D_1^{\square}$	0,44	1,00
$2D_1^{\triangle C}$	0,90	0,50
$2D_1^{\triangle P}$	0,60	0,74
$2D_2^{\square}$	1,57	1,00
$2D_2^{\triangle C}$	3,09	0,51
$2D_2^{\triangle P}$	2,12	0,74
$2D_3^{\square}$	5,63	1,00
$2D_3^{\triangle C}$	11,45	0,49
$2D_3^{\triangle P}$	7,65	0,74
$2D_4^{\square}$	21,54	1,00
$2D_4^{\triangle C}$	44,97	0,48
$2D_4^{\triangle P}$	29,77	0,72

Tabuľka 5.6: Porovnanie výpočtového času CT [ms] pre operáciu načítania zo súboru na mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle X}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle C}$, $2D_i^{\triangle P}$

Id.	*CT*	$Sp_{\square}$
$3D_0^{\square}$	0,43	1,00
$3D_0^{\triangle CC}$	0,39	1,11
$3D_0^{\triangle CP}$	0,24	1,78
$3D_0^{\triangle PC}$	0,30	1,43
$3D_0^{\triangle PP}$	0,20	2,12
$3D_1^{\square}$	2,87	1,00
$3D_1^{\triangle CC}$	2,82	1,02
$3D_1^{\triangle CP}$	1,56	1,85
$3D_1^{\triangle PC}$	2,16	1,33
$3D_1^{\triangle PP}$	1,18	2,44
$3D_2^{\square}$	6,17	1,00
$3D_2^{\triangle CC}$	6,33	0,97
$3D_2^{\triangle CP}$	3,40	1,81
$3D_2^{\triangle PC}$	5,41	1,14
$3D_2^{\triangle PP}$	2,72	2,27
$3D_3^{\square}$	43,12	1,00
$3D_3^{\triangle CC}$	47,08	0,92
$3D_3^{\triangle CP}$	24,55	1,76
$3D_3^{\triangle PC}$	37,69	1,14
$3D_3^{\triangle PP}$	19,59	2,20
$3D_4^{\square}$	333,69	1,00
$3D_4^{\triangle CC}$	368,04	0,91
$3D_4^{\triangle CP}$	193,22	1,73
$3D_4^{\triangle PC}$	299,64	1,11
$3D_4^{\triangle PP}$	155,80	2,14
$3D_5^{\square}$	2 695,76	1,00
$3D_5^{\triangle CC}$	2 923,74	0,92
$3D_5^{\triangle CP}$	1 541,07	1,75
$3D_5^{\triangle PC}$	2 533,30	1,06
$3D_5^{\triangle PP}$	1 243,36	2,17

Tabuľka 5.7: Porovnanie výpočtového času CT [ms] pre operáciu načítania zo súboru na mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle XY}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle CC}$, $3D_i^{\triangle CP}$, $3D_i^{\triangle PC}$, $3D_i^{\triangle PP}$

5.3.3 Inicializácia

Táto sekcia sa zaoberá meraním času inicializácie siete (viď sekcie 2.2.6 a 3.3.5). Pre upresnenie, inicializácie siete zahrňuje nastavenie potrebných zárodkov siete v inštancii triedy `MeshBuilder`, ktoré sú následne použité pri samotnej inicializácii položiek dátovej štruktúry na základe špecifikovanej konfigurácie. Výsledkom operácie je objekt siete (inštancia triedy `Mesh<Config, Devices::Host>`), čo znamená, že jej vykonanie je vždy nevyhnutné pred samotným použitím siete k výpočtu.

Z tabuliek 5.8 a 5.9 je vidieť, že inicializácia siete pri maximálnej konfigurácii je v priemere približne 1,5-krát pomalšia ako pri minimálnej konfigurácii. Pre mnohouholníkové siete je operácia v najlepšom prípade približne 2-krát rýchlejšia ako pre dekomponované trojuholníkové siete. Na druhej strane pre mnohostenné siete, je operácia v najlepšom prípade približne 8-krát rýchlejšia ako pre dekomponované štvorstenné siete. Hlavným dôvodom, prečo operácia pre dekomponované siete trvá dlhšie, je to, že obsahujú väčší počet buniek, ktorý tiež podstatne závisí na type použitej dekompozície.

Je si tiež možné všimnúť, že použitie viacerých vlákien neprinieslo podstatné paralelné zrýchlenie. Hlavným dôvodom je to, že väčšina času inicializácie je strávená vkladaním prvkov do hash tabuľky, ktorá beží len sekvenčne, keďže súčasná implementácia používa hash tabuľku zo štandardnej knižnice (STL). Pre pripomenutie, je hash tabuľka používaná pre pridelenie unikátneho indexu entitám na základe ich zárodkov (zoznamov vrcholov).

	Min. konf.					Max. konf.				
	1 vl.		8 vl.			1 vl.		8 vl.		
Id.	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*Sp*	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*Sp*
$2D_0^{\square}$	0,30	1,00	0,71	1,00	0,43	0,40	1,00	1,01	1,00	0,40
$2D_0^{\triangle C}$	0,77	0,39	0,68	1,03	1,13	1,20	0,33	1,17	0,86	1,03
$2D_0^{\triangle P}$	0,49	0,62	0,67	1,06	0,73	0,86	0,47	1,23	0,82	0,70
$2D_1^{\square}$	1,00	1,00	0,84	1,00	1,19	1,34	1,00	1,26	1,00	1,07
$2D_1^{\triangle C}$	2,58	0,39	2,10	0,40	1,23	3,88	0,35	3,67	0,34	1,06
$2D_1^{\triangle P}$	1,66	0,60	1,42	0,60	1,17	2,97	0,45	2,74	0,46	1,08
$2D_2^{\square}$	3,50	1,00	2,86	1,00	1,23	4,69	1,00	4,17	1,00	1,13
$2D_2^{\triangle C}$	10,51	0,33	8,28	0,34	1,27	15,63	0,30	13,31	0,31	1,17
$2D_2^{\triangle P}$	6,55	0,54	5,85	0,49	1,12	11,16	0,42	9,71	0,43	1,15
$2D_3^{\square}$	13,79	1,00	10,88	1,00	1,27	18,60	1,00	15,73	1,00	1,18
$2D_3^{\triangle C}$	41,33	0,33	31,76	0,34	1,30	63,44	0,29	50,98	0,31	1,24
$2D_3^{\triangle P}$	25,35	0,54	19,41	0,56	1,31	43,53	0,43	37,66	0,42	1,16
$2D_4^{\square}$	57,68	1,00	41,99	1,00	1,37	78,32	1,00	59,01	1,00	1,33
$2D_4^{\triangle C}$	222,45	0,26	161,38	0,26	1,38	306,77	0,26	232,91	0,25	1,32
$2D_4^{\triangle P}$	116,67	0,49	81,61	0,51	1,43	191,45	0,41	149,89	0,39	1,28

Tabuľka 5.8: Porovnanie výpočtového času CT [s], CPU paralelného zrýchlenia Sp pre operáciu inicializácie na mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle X}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle C}$, $2D_i^{\triangle P}$

	Min. konf.					Max. konf.				
	1 vl.		8 vl.			1 vl.		8 vl.		
Id.	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*Sp*	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*Sp*
$3D_0^{\square}$	0,001	1,00	0,001	1,00	0,61	0,001	1,00	0,002	1,00	0,60
$3D_0^{\triangle CC}$	0,005	0,13	0,004	0,23	1,10	0,011	0,12	0,009	0,26	1,28
$3D_0^{\triangle CP}$	0,003	0,24	0,003	0,38	0,95	0,006	0,21	0,006	0,38	1,06
$3D_0^{\triangle PC}$	0,004	0,15	0,004	0,29	1,17	0,009	0,15	0,008	0,29	1,16
$3D_0^{\triangle PP}$	0,002	0,32	0,002	0,53	1,03	0,005	0,28	0,005	0,49	1,04
$3D_1^{\square}$	0,004	1,00	0,004	1,00	0,81	0,008	1,00	0,009	1,00	0,90
$3D_1^{\triangle CC}$	0,035	0,10	0,028	0,16	1,26	0,077	0,10	0,056	0,16	1,38
$3D_1^{\triangle CP}$	0,018	0,20	0,016	0,27	1,09	0,042	0,19	0,032	0,27	1,30
$3D_1^{\triangle PC}$	0,028	0,13	0,022	0,20	1,25	0,061	0,13	0,045	0,19	1,36
$3D_1^{\triangle PP}$	0,013	0,28	0,012	0,37	1,06	0,031	0,25	0,024	0,36	1,27
$3D_2^{\square}$	0,008	1,00	0,006	1,00	1,29	0,018	1,00	0,009	1,00	1,90
$3D_2^{\triangle CC}$	0,083	0,10	0,064	0,10	1,28	0,182	0,10	0,127	0,07	1,43
$3D_2^{\triangle CP}$	0,044	0,19	0,035	0,18	1,26	0,102	0,17	0,076	0,12	1,35
$3D_2^{\triangle PC}$	0,069	0,12	0,054	0,12	1,28	0,149	0,12	0,106	0,09	1,40
$3D_2^{\triangle PP}$	0,031	0,26	0,025	0,25	1,24	0,076	0,24	0,057	0,16	1,32
$3D_3^{\square}$	0,060	1,00	0,042	1,00	1,42	0,128	1,00	0,078	1,00	1,66
$3D_3^{\triangle CC}$	0,993	0,06	0,802	0,05	1,24	1,773	0,07	1,246	0,06	1,42
$3D_3^{\triangle CP}$	0,437	0,14	0,337	0,13	1,30	0,908	0,14	0,630	0,12	1,44
$3D_3^{\triangle PC}$	0,784	0,08	0,617	0,07	1,27	1,460	0,09	1,006	0,08	1,45
$3D_3^{\triangle PP}$	0,312	0,19	0,238	0,18	1,31	0,665	0,19	0,470	0,16	1,41
$3D_4^{\square}$	0,526	1,00	0,359	1,00	1,47	1,127	1,00	0,665	1,00	1,70
$3D_4^{\triangle CC}$	9,125	0,06	7,454	0,05	1,22	15,455	0,07	10,972	0,06	1,41
$3D_4^{\triangle CP}$	4,960	0,11	3,966	0,09	1,25	8,841	0,13	6,292	0,11	1,41
$3D_4^{\triangle PC}$	7,713	0,07	6,223	0,06	1,24	13,188	0,09	9,205	0,07	1,43
$3D_4^{\triangle PP}$	3,418	0,15	2,765	0,13	1,24	6,273	0,18	4,552	0,15	1,38
$3D_5^{\square}$	4,638	1,00	3,331	1,00	1,39	9,521	1,00	4,477	1,00	2,13
$3D_5^{\triangle CC}$	78,477	0,06	63,562	0,05	1,23	130,845	0,07	91,157	0,05	1,44
$3D_5^{\triangle CP}$	44,757	0,10	35,166	0,09	1,27	77,647	0,12	54,227	0,08	1,43
$3D_5^{\triangle PC}$	69,483	0,07	55,345	0,06	1,26	115,711	0,08	79,937	0,06	1,45
$3D_5^{\triangle PP}$	30,657	0,15	24,218	0,14	1,27	54,693	0,17	38,557	0,12	1,42

Tabuľka 5.9: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre operáciu inicializácie na mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle XY}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle CC}$, $3D_i^{\triangle CP}$, $3D_i^{\triangle PC}$, $3D_i^{\triangle PP}$

5.3.4 Kópia medzi CPU a GPU pamäťou

Táto sekcia sa zaoberá meraním času kópie objektu siete (inštancie triedy `Mesh<Config, Device>`) medzi pamäťou CPU a GPU. Táto operácia je nevyhnutná v prípade, keď sa sieť používa pre výpočty na GPU.

Z tabuľky 5.10 si je možné všimnúť, že mnohouholníkové siete sa kopírovali v najhoršom prípade približne 2-krát rýchlejšie ako dekomponované trojuholníkové siete. Na druhej strane, z tabuľky 5.11 je vidno, že mnohostenné siete sa kopírovali v najhoršom prípade približne 7-krát rýchlejšie ako dekomponované štvorstenné siete. Hlavným dôvodom je to, že dekomponované siete majú väčší počet entít ako pôvodné siete.

Je si tiež možné všimnúť, že kopírovanie z pamäte GPU do pamäte CPU je podivuhodne v priemere približne 1,3-krát rýchlejšie ako kopírovanie opačným smerom. Dôvodom je to, že pri operácii dochádza ku kopírovaniu matíc, ktorých prvky sú uložené v pamäti v rôznych orientáciách. Orientácia závisí na tom, či sa matica nachádza v pamäti CPU alebo GPU. V prípade GPU sú prvky uložené po stĺpcoch, zatiaľ čo v prípade CPU sú uložené po riadkoch. Kvôli tomu sa ešte pri kopírovaniu matíc vykonáva transpozícia, čo spôsobuje rozdiely vo výkonnosti závisiace na smere kopírovania.

	$CPU \rightarrow GPU$				$CPU \leftarrow GPU$			
	Min. konf.		Max. konf.		Min. konf.		Max. konf.	
Id.	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$
$2D_0^{\square}$	0,96	1,00	1,40	1,00	0,65	1,00	0,95	1,00
$2D_0^{\triangle C}$	1,00	0,95	1,55	0,90	0,72	0,90	1,09	0,87
$2D_0^{\triangle P}$	0,92	1,04	1,47	0,95	0,66	0,98	1,03	0,92
$2D_1^{\square}$	1,35	1,00	1,93	1,00	0,86	1,00	1,25	1,00
$2D_1^{\triangle C}$	2,08	0,65	3,05	0,63	1,43	0,60	2,07	0,60
$2D_1^{\triangle P}$	1,75	0,77	2,64	0,73	1,18	0,73	1,77	0,71
$2D_2^{\square}$	3,21	1,00	4,61	1,00	2,19	1,00	3,13	1,00
$2D_2^{\triangle C}$	6,48	0,50	9,62	0,48	5,26	0,42	6,56	0,48
$2D_2^{\triangle P}$	5,14	0,62	8,01	0,58	3,49	0,63	5,19	0,6
$2D_3^{\square}$	10,23	1,00	14,61	1,00	6,82	1,00	9,59	1,00
$2D_3^{\triangle C}$	24,35	0,42	34,02	0,43	16,35	0,42	23,46	0,41
$2D_3^{\triangle P}$	18,75	0,55	27,40	0,53	12,55	0,54	17,72	0,54
$2D_4^{\square}$	37,89	1,00	53,37	1,00	25,57	1,00	36,08	1,00
$2D_4^{\triangle C}$	92,35	0,41	132,87	0,40	64,87	0,39	100,42	0,36
$2D_4^{\triangle P}$	71,52	0,53	105,94	0,50	47,63	0,54	71,38	0,51

Tabuľka 5.10: Porovnanie výpočtového času CT [ms] pre operáciu kópie medzi CPU a GPU pamäťou na mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle X}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle C}$, $2D_i^{\triangle P}$

	$CPU \rightarrow GPU$				$CPU \leftarrow GPU$			
	Min. konf.		Max. konf.		Min. konf.		Max. konf.	
Id.	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$
$3D_0^{\square}$	1,26	1,00	2,98	1,00	0,93	1,00	2,08	1,00
$3D_0^{\triangle CC}$	2,78	0,45	7,88	0,38	2,16	0,43	6,83	0,30
$3D_0^{\triangle CP}$	1,68	0,75	4,57	0,65	1,32	0,71	3,56	0,58
$3D_0^{\triangle PC}$	2,53	0,50	6,87	0,43	2,05	0,45	5,82	0,36
$3D_0^{\triangle PP}$	1,61	0,78	4,68	0,64	1,28	0,73	3,84	0,54
$3D_1^{\square}$	1,91	1,00	4,60	1,00	1,37	1,00	3,31	1,00
$3D_1^{\triangle CC}$	13,91	0,14	37,31	0,12	10,01	0,14	26,75	0,12
$3D_1^{\triangle CP}$	5,95	0,32	17,40	0,26	4,37	0,31	12,03	0,28
$3D_1^{\triangle PC}$	10,13	0,19	28,85	0,16	7,53	0,18	20,90	0,16
$3D_1^{\triangle PP}$	5,39	0,35	15,56	0,30	3,81	0,36	10,53	0,31
$3D_2^{\square}$	2,77	1,00	7,06	1,00	1,92	1,00	5,02	1,00
$3D_2^{\triangle CC}$	29,31	0,09	81,99	0,09	20,48	0,09	57,09	0,09
$3D_2^{\triangle CP}$	12,91	0,21	35,24	0,20	9,04	0,21	23,28	0,22
$3D_2^{\triangle PC}$	22,88	0,12	63,18	0,11	16,32	0,12	46,14	0,11
$3D_2^{\triangle PP}$	11,57	0,24	33,13	0,21	8,04	0,24	23,20	0,22
$3D_3^{\square}$	13,47	1,00	33,17	1,00	9,19	1,00	22,17	1,00
$3D_3^{\triangle CC}$	215,56	0,06	568,13	0,06	153,93	0,06	434,42	0,05
$3D_3^{\triangle CP}$	85,01	0,16	230,50	0,14	59,58	0,15	176,09	0,13
$3D_3^{\triangle PC}$	154,50	0,09	432,34	0,08	124,79	0,07	349,77	0,06
$3D_3^{\triangle PP}$	78,52	0,17	223,47	0,15	54,23	0,17	163,83	0,14
$3D_4^{\square}$	94,96	1,00	234,16	1,00	64,20	1,00	178,80	1,00
$3D_4^{\triangle CC}$	1 592,40	0,06	4 406,50	0,05	1 191,34	0,05	3 457,25	0,05
$3D_4^{\triangle CP}$	640,84	0,15	1 746,91	0,13	462,48	0,14	1 302,32	0,14
$3D_4^{\triangle PC}$	1 220,69	0,08	3 354,17	0,07	961,30	0,07	2 715,08	0,07
$3D_4^{\triangle PP}$	590,37	0,16	1 666,56	0,14	425,62	0,15	1 255,01	0,14
$3D_5^{\square}$	725,93	1,00	1 798,28	1,00	521,22	1,00	1 397,41	1,00
$3D_5^{\triangle CC}$	12 886,19	0,06			10 198,42	0,05		
$3D_5^{\triangle CP}$	5 184,00	0,14	13 934,66	0,13	3 995,68	0,13	11 132,88	0,13
$3D_5^{\triangle PC}$	9 811,47	0,07			8 532,87	0,06		
$3D_5^{\triangle PP}$	4 895,61	0,15	13 520,33	0,13	3 660,71	0,14	10 454,15	0,13

Tabuľka 5.11: Porovnanie výpočtového času CT [ms] pre operáciu kópie medzi CPU a GPU pamäťou na mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle XY}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle CC}$, $3D_i^{\triangle CP}$, $3D_i^{\triangle PC}$, $3D_i^{\triangle PP}$. Pre siete $3D_5^{\triangle CC}$ a $3D_5^{\triangle PC}$ nebola pre maximálnu konfiguráciu operácia vykonaná kvôli nedostatku pamäte GPU

5.3.5 Výpočet miery

Táto sekcia sa zaoberá meraním času výpočtu miery každej bunky siete (viď sekciu 4.3). Ako už bolo spomenuté, výpočtom miery sa myslí v prípade 2D sietí výpočet obsahu, zatiaľ čo v prípade 3D sietí sa myslí výpočet objemu. Jednotlivé výsledky mier sa ukladajú do globálneho poľa uloženého v pamäti, v ktorej sa nachádza daná sieť. Redukcia poľa výsledkov s cieľom získania celkovej miery siete je pre účely tejto testovacej úlohy vynechaná.

Miery jednotlivých buniek reprezentujú nezávislé úlohy, ktoré sú vykonané paralelne. Výpočet miery bunky vyžaduje prečítanie daných bodov bunky, ktoré sú následne použité vo formule pre výpočet miery. Mnohostenné siete naviac vyžadujú prečítanie indexov stien danej bunky, keďže výpočet miery vyžaduje dekompozíciu mnohostenu na štvorsteny, ktorých objemy sa sčítajú.

Z tabuľky 5.12 je vidieť, že pri mnohouholníkových a trojuholníkových sieťach sa na CPU dosahuje pri najväčšej sieti takmer lineárne paralelné zrýchlenie. Pri väčších sietiach je sekvenčný výpočet pre dekomponované siete v najlepšom prípade približne 2-krát rýchlejší, ale na druhej strane je 8-vláknový výpočet pre menšie siete časovo prakticky identický kvôli lepšiemu škálovaniu pre mnohouholníkové siete pri menších sietiach. Pri výpočtoch na GPU, mnohouholníkové siete dosiahli o čosi vyššieho paralelného zrýchlenia oproti dekomponovaným sietiam, ale na druhej strane je celkový výpočet pre dekomponované siete približne 1,25-krát rýchlejší.

Z tabuľky 5.13 je vidieť, že pri mnohostenných a štvorstenných sietiach sa dosahuje približne 7-násobného paralelného zrýchlenia pri 8 vláknach s výnimkou menších sietí. Z pohľadu celkového času výpočtu, sú dekompované siete na CPU v najlepšom prípade 1,25-krát rýchlejšie s výnimkou najmenších sietí. Pri najmenších sietiach tiež dosahovali dekomponované siete na CPU pri 8 vláknach podstatne lepšieho paralelného zrýchlenia ako mnohostenné siete. Čo sa týka výpočtov na GPU, dosahovali dekomponované siete podstatne lepšieho paralelného zrýchlenia oproti mnohostenným sietiam. Hlavnou príčinou ne-efektivity výpočtu miery mnohostenných sietí na GPU je *divergencia warpov*, ktorá je spôsobená nevyváženosťou množstva práce jednotlivých úloh. To vyplýva z toho, že mnohosteny sú tvorené rôznymi počtami mnohouholníkových stien, kde každá stena môže byť tvorená rôznymi počtami vrcholov. Je si možné všimnúť, že podobná situácia nastala aj pri výpočte miery mnohouholníkových sietí, keďže jednotlivé mnohouholníky mali potencionálne odlišné počty vrcholov, ale *divergencia warpov* nebola v ich prípade veľmi výrazná, keďže sa dosiahlo podstatného paralelného zrýchlenia.

	1 vl.		8 vl.			GPU			
Id.	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*Sp*	*CT*	$Sp_{\square}$	GSp_1	GSp_8
$2D_0^{\square}$	0,01	1,00	0,01	1,00	1,00	0,01	1,00	0,88	0,88
$2D_0^{\triangle_C}$	0,01	1,17	0,00	1,75	1,50	0,01	1,33	1,00	0,67
$2D_0^{\triangle_P}$	0,00	1,40	0,00	1,40	1,00	0,01	1,33	0,83	0,83
$2D_1^{\square}$	0,03	1,00	0,00	1,00	5,40	0,01	1,00	3,38	0,63
$2D_1^{\triangle_C}$	0,02	1,42	0,00	1,00	3,80	0,01	1,33	3,17	0,83
$2D_1^{\triangle_P}$	0,02	1,69	0,00	1,00	3,20	0,01	1,33	2,67	0,83
$2D_2^{\square}$	0,09	1,00	0,01	1,00	6,57	0,01	1,00	10,22	1,56
$2D_2^{\triangle_C}$	0,07	1,31	0,01	1,17	5,83	0,01	1,29	10,00	1,71
$2D_2^{\triangle_P}$	0,05	2,00	0,01	1,00	3,29	0,01	1,29	6,57	2,00
$2D_3^{\square}$	0,36	1,00	0,05	1,00	7,41	0,01	1,00	33,00	4,45
$2D_3^{\triangle_C}$	0,27	1,32	0,04	1,29	7,21	0,01	1,10	27,40	3,80
$2D_3^{\triangle_P}$	0,19	1,95	0,03	1,81	6,89	0,01	1,22	20,67	3,00
$2D_4^{\square}$	1,44	1,00	0,20	1,00	7,37	0,02	1,00	57,76	7,84
$2D_4^{\triangle_C}$	1,12	1,30	0,27	0,72	4,11	0,03	0,93	41,30	10,04
$2D_4^{\triangle_P}$	0,75	1,93	0,10	2,00	7,62	0,02	1,25	37,35	4,90

Tabuľka 5.12: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp, GPU paralelného zrýchlenia GSp pre úlohu výpočtu mieri buniek na mnohouholníkových ($2D_i^{\square}$) a trojuholníkových ($2D_i^{\triangle_X}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $2D_i^{\square}$ a $2D_i^{\triangle_C}$, $2D_i^{\triangle_P}$

	1 vl.		8 vl.			GPU			
Id.	*CT*	$Sp_{\square}$	*CT*	$Sp_{\square}$	*Sp*	*CT*	$Sp_{\square}$	GSp_1	GSp_8
$3D_0^{\square}$	0,01	1,00	0,01	1,00	0,92	0,02	1,00	0,48	0,52
$3D_0^{\triangle CC}$	0,02	0,48	0,01	1,71	3,29	0,01	3,83	3,83	1,17
$3D_0^{\triangle CP}$	0,02	0,73	0,00	3,00	3,75	0,01	3,83	2,50	0,67
$3D_0^{\triangle PC}$	0,02	0,65	0,01	1,71	2,43	0,01	3,83	2,83	1,17
$3D_0^{\triangle PP}$	0,01	1,00	0,01	1,71	1,57	0,01	3,83	1,83	1,17
$3D_1^{\square}$	0,08	1,00	0,08	1,00	0,93	0,04	1,00	1,85	2,00
$3D_1^{\triangle CC}$	0,14	0,55	0,02	3,90	6,52	0,01	5,86	19,57	3,00
$3D_1^{\triangle CP}$	0,08	0,92	0,02	3,73	3,77	0,01	5,86	11,86	3,14
$3D_1^{\triangle PC}$	0,10	0,75	0,02	4,82	5,94	0,01	5,86	14,43	2,43
$3D_1^{\triangle PP}$	0,06	1,25	0,02	5,12	3,81	0,01	5,86	8,71	2,29
$3D_2^{\square}$	0,18	1,00	0,03	1,00	6,81	0,05	1,00	3,47	0,51
$3D_2^{\triangle CC}$	0,31	0,57	0,05	0,58	6,91	0,01	5,67	34,56	5,00
$3D_2^{\triangle CP}$	0,19	0,94	0,03	0,90	6,52	0,01	6,38	23,63	3,62
$3D_2^{\triangle PC}$	0,24	0,74	0,03	0,74	6,86	0,01	6,38	30,00	4,38
$3D_2^{\triangle PP}$	0,14	1,22	0,02	1,18	6,59	0,01	6,38	18,12	2,75
$3D_3^{\square}$	1,27	1,00	0,18	1,00	6,92	0,06	1,00	21,23	3,07
$3D_3^{\triangle CC}$	2,26	0,56	0,32	0,58	7,11	0,03	1,76	66,53	9,35
$3D_3^{\triangle CP}$	1,37	0,93	0,19	0,96	7,16	0,02	2,61	59,74	8,35
$3D_3^{\triangle PC}$	1,74	0,73	0,24	0,75	7,14	0,03	2,14	62,25	8,71
$3D_3^{\triangle PP}$	1,05	1,21	0,15	1,25	7,16	0,02	3,53	61,88	8,65
$3D_4^{\square}$	10,25	1,00	1,47	1,00	6,98	0,61	1,00	16,83	2,41
$3D_4^{\triangle CC}$	17,88	0,57	2,57	0,57	6,95	0,22	2,82	82,79	11,92
$3D_4^{\triangle CP}$	11,05	0,93	1,60	0,92	6,93	0,14	4,32	78,34	11,31
$3D_4^{\triangle PC}$	13,97	0,73	2,00	0,73	6,99	0,17	3,60	82,68	11,82
$3D_4^{\triangle PP}$	8,53	1,20	1,24	1,19	6,90	0,11	5,49	76,89	11,14
$3D_5^{\square}$	89,53	1,00	14,86	1,00	6,03	4,74	1,00	18,88	3,13
$3D_5^{\triangle CC}$	147,17	0,61	21,64	0,69	6,80	Sieť sa nevošla do pamäte.			
$3D_5^{\triangle CP}$	92,02	0,97	13,62	1,09	6,76	1,08	4,39	85,12	12,60
$3D_5^{\triangle PC}$	115,78	0,77	17,17	0,87	6,74	Sieť sa nevošla do pamäte.			
$3D_5^{\triangle PP}$	71,63	1,25	10,78	1,38	6,64	0,84	5,68	85,79	12,91

Tabuľka 5.13: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp, GPU paralelného zrýchlenia GSp pre úlohu výpočtu mieri buniek na mnohostenných ($3D_i^{\square}$) a štvorstenných ($3D_i^{\triangle XY}$) sietiach. $Sp_{\square}$ vyjadruje porovnanie medzi $3D_i^{\square}$ a $3D_i^{\triangle CC}$, $3D_i^{\triangle CP}$, $3D_i^{\triangle PC}$, $3D_i^{\triangle PP}$

5.3.6 Dekompozícia

Táto sekcia sa zaoberá meraním času úlohy dekompozície mnohouholníkových a mnohostenných sietí na trojuholníkové a štvorstenné siete (viď sekciu 4.1). Ako už bolo spomenuté, výsledkom tejto úlohy nie je objekt siete, ale len inštancia triedy `MeshBuilder`, ktorá obsahuje matice reprezentujúce zárodky buniek, prípadne stien. Aby dekomponované siete boli použiteľné k výpočtu, je ich potrebné ešte inicializovať, ale ako bolo vidieť v sekcii 5.3.3, inicializácia nie je veľmi lacná operácia, z čoho dôvodu bola pre účely tejto testovacej úlohy vynechaná, keďže hlavným cieľom je zmerať len čas samotnej dekompozície siete.

Ako už bolo spomenuté v sekcii 4.1, táto úloha nebola implementovaná pre GPU, keďže výpočet by vyžadoval dodatočný prenos dát medzi CPU a GPU, ktorý by zaberal výraznú porciu celkového času výpočtu. To by spôsobilo veľmi malé alebo žiadne paralelné zrýchlenie.

Tabuľky 5.14 a 5.15 obsahujú výsledky pre mnohouholníkové siete. Obidve typy dekompozícií dosahovali pri najväčšej sieti viac ako 6-násobné paralelné zrýchlenie pri 8 vláknach. Ostatné siete neboli dostatočne veľké, kvôli čomu réžia ohľadne tvorby vlákien bola viac výraznejšia, ako sa veľkosť siete znižovala.

Na druhej strane, tabuľky 5.16, 5.17, 5.18 a 5.19 obsahujú výsledky pre mnohostenné siete. Naprieč všetkými 4 typmi dekompozícií sa vo väčšine prípadov dosiahlo 4- až 6-násobného paralelného zrýchlenia pri 8 vláknach s výnimkou menších sietí. Vo väčšine prípadoch dosahovali obidve konfigurácie relatívne podobných výsledkov, miestami ale siete v maximálnej konfigurácii, dosiahli výrazne menšieho zrýchlenia oproti minimálnej konfigurácii. To značí, že sa vyplatí vynechať nepotrebné položky dátovej štruktúry nielen výhradne kvôli úspore pamäti.

Pri všetkých meraných typoch dekompozície sa nepodarilo priblížiť skoro lineárnemu paralelnému zrýchleniu. Hlavný dôvod spočíva v samotnej povahe algoritmu, v ktorom prevládajú operácie s pamäťou, čo znamená, že je rýchlosť výpočtu limitovaná priepustnosťou pamäte.

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$2D_0^{\square}$	0,02	0,03	0,50	0,02	0,03	0,48
$2D_1^{\square}$	0,04	0,15	0,28	0,04	0,15	0,29
$2D_2^{\square}$	0,14	0,09	1,54	0,14	0,09	1,57
$2D_3^{\square}$	0,58	0,12	5,03	0,57	0,11	5,18
$2D_4^{\square}$	2,35	0,35	6,64	2,32	0,37	6,24

Tabuľka 5.14: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu C na mnohouholníkových ($2D_i^{\square}$) sietiach

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$2D_0^{\square}$	0,01	0,03	0,42	0,01	0,03	0,38
$2D_1^{\square}$	0,02	0,14	0,16	0,02	0,14	0,18
$2D_2^{\square}$	0,08	0,08	0,93	0,08	0,08	0,91
$2D_3^{\square}$	0,29	0,08	3,57	0,31	0,07	4,18
$2D_4^{\square}$	1,17	0,19	6,05	1,19	0,19	6,17

Tabuľka 5.15: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu P na mnohouholníkových ($2D_i^{\square}$) sietiach

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$3D_0^{\square}$	0,02	0,04	0,60	0,02	0,04	0,57
$3D_1^{\square}$	0,16	0,18	0,87	0,15	0,18	0,83
$3D_2^{\square}$	0,36	0,09	4,14	0,36	0,08	4,38
$3D_3^{\square}$	2,66	0,46	5,73	2,50	0,43	5,83
$3D_4^{\square}$	36,38	8,44	4,31	36,07	10,76	3,35
$3D_5^{\square}$	287,48	68,76	4,18	283,32	67,68	4,19

Tabuľka 5.16: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu CC na mnohostenných ($3D_i^{\square}$) sietiach

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$3D_0^{\square}$	0,01	0,03	0,54	0,01	0,03	0,54
$3D_1^{\square}$	0,08	0,11	0,79	0,08	0,11	0,79
$3D_2^{\square}$	0,20	0,06	3,14	0,19	0,06	3,24
$3D_3^{\square}$	1,37	0,28	4,98	1,38	0,26	5,24
$3D_4^{\square}$	13,59	3,15	4,32	13,80	3,44	4,01
$3D_5^{\square}$	171,54	43,77	3,92	116,83	42,89	2,72

Tabuľka 5.17: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu CP na mnohostenných ($3D_i^{\square}$) sietiach

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$3D_0^{\square}$	0,03	0,05	0,62	0,03	0,05	0,60
$3D_1^{\square}$	0,19	0,22	0,86	0,19	0,22	0,84
$3D_2^{\square}$	0,44	0,10	4,53	0,44	0,10	4,62
$3D_3^{\square}$	3,13	0,53	5,92	3,13	0,52	5,97
$3D_4^{\square}$	38,10	7,69	4,95	28,26	7,74	3,65
$3D_5^{\square}$	309,92	65,98	4,70	303,56	65,32	4,65

Tabuľka 5.18: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu PC na mnohostenných ($3D_i^{\square}$) sietiach

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$3D_0^{\square}$	0,02	0,03	0,65	0,02	0,03	0,58
$3D_1^{\square}$	0,14	0,17	0,83	0,14	0,16	0,84
$3D_2^{\square}$	0,32	0,08	4,21	0,32	0,08	4,21
$3D_3^{\square}$	2,28	0,41	5,53	2,29	0,43	5,38
$3D_4^{\square}$	20,07	3,53	5,68	19,78	3,62	5,47
$3D_5^{\square}$	166,56	34,65	4,81	166,97	42,71	3,91

Tabuľka 5.19: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu dekompozície typu PP na mnohostenných ($3D_i^{\square}$) sietiach

5.3.7 Korekcia nerovinných mnohouholníkov

Táto sekcia sa zaoberá meraním času úlohy korekcie nerovinných mnohouholníkov pre mnohouholníkové a mnohostenné siete (viď sekciu 4.2). Podobne ako to bolo pri úlohe dekompozície v sekcii 5.3.6, je výsledkom výpočtu opäť len inštancia triedy `MeshBuilder`, ktorá obsahuje matice zárodkov buniek, prípadne stien. Inicializácia siete bola tiež vynechaná a úloha nebola tiež implementovaná pre GPU z rovnakého dôvodu ako úloha dekompozície.

Ako už bolo spomenuté v sekcii 4.2, algoritmicky je táto úloha podobná dekompozícii, len s tým rozdielom, že sa výhradne dekomponujú len nerovinné mnohouholníky. To znamená že je tiež potrebné overiť vlastnosť rovinnosti pre všetky mnohouholníky vyskytujúce sa v sieti.

Z tabuliek 5.20, 5.21, 5.22 a 5.23 je vidieť, že sa pri mnohouholníkových a mnohostenných sietiach dosahuje v priemere približne 5-násobného paralelného zrýchlenia pri 8 vláknach s výnimkou menších sietí. Je tiež vidno, že typ konfigurácie ovplyvňuje efektivitu výpočtu len minimálne.

Podobne ako pri dekompozícii v sekcii 5.3.6, sa nepodarilo dosiahnuť skoro lineárneho paralelného zrýchlenia z rovnakého dôvodu.

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$2D_0^{\square}$	0,02	0,26	0,07	0,02	0,24	0,07
$2D_1^{\square}$	0,06	0,15	0,38	0,06	0,15	0,37
$2D_2^{\square}$	0,19	0,12	1,59	0,19	0,12	1,59
$2D_3^{\square}$	0,74	0,14	5,15	0,73	0,14	5,08
$2D_4^{\square}$	2,93	0,64	4,58	2,87	0,62	4,67

Tabuľka 5.20: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu korekcie nerovinných mnohouholníkov s dekompozíciou typu C na mnohouholníkových ($2D_i^{\square}$) sietiach

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$2D_0^{\square}$	0,02	0,26	0,07	0,02	0,24	0,08
$2D_1^{\square}$	0,05	0,16	0,33	0,06	0,14	0,40
$2D_2^{\square}$	0,18	0,12	1,48	0,18	0,12	1,52
$2D_3^{\square}$	0,69	0,14	4,98	0,71	0,14	5,14
$2D_4^{\square}$	2,70	0,60	4,48	2,73	0,59	4,62

Tabuľka 5.21: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu korekcie nerovinných mnohouholníkov s dekompozíciou typu P na mnohouholníkových ($2D_i^{\square}$) sietiach

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$3D_0^{\square}$	0,04	0,05	0,79	0,03	0,05	0,63
$3D_1^{\square}$	0,23	0,41	0,55	0,24	0,39	0,61
$3D_2^{\square}$	0,53	0,24	2,21	0,54	0,22	2,43
$3D_3^{\square}$	3,18	0,69	4,62	3,12	0,67	4,66
$3D_4^{\square}$	14,12	2,64	5,35	13,80	2,59	5,32
$3D_5^{\square}$	120,35	22,46	5,36	119,23	22,43	5,31

Tabuľka 5.22: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu korekcie nerovinných mnohouholníkov s dekompozícou typu C na mnohostenných ($3D_i^{\square}$) sietiach

	Min. konf.			Max. konf.		
	1 vl.	8 vl.		1 vl.	8 vl.	
Id.	*CT*	*CT*	*Sp*	*CT*	*CT*	*Sp*
$3D_0^{\square}$	0,03	0,06	0,42	0,02	0,07	0,38
$3D_1^{\square}$	0,18	0,38	0,46	0,18	0,36	0,49
$3D_2^{\square}$	0,40	0,27	1,46	0,40	0,27	1,50
$3D_3^{\square}$	2,58	0,66	3,93	2,61	0,63	4,14
$3D_4^{\square}$	12,97	2,51	5,17	12,98	2,49	5,21
$3D_5^{\square}$	112,96	22,07	5,12	113,32	22,05	5,14

Tabuľka 5.23: Porovnanie výpočtového času CT [ms], CPU paralelného zrýchlenia Sp pre úlohu korekcie nerovinných mnohouholníkov s dekompozíciou typu P na mnohostenných ($3D_i^{\square}$) sietiach

Záver

Ciele a výsledky

Účelom tejto práce bolo zovšeobecniť dátovú štruktúru pre reprezentáciu neštruktúrovaných numerických sietí z knižnice TNL [11], o podporu mnohouholníkových a mnohostenných topológii.

Pôvodná implementácia obsahovala podporu len pre statické topológie, ako napríklad trojuholník alebo štvorsten, pri ktorých je štruktúra bunky vždy konštantná a známa už v čase prekladu (viď kapitolu 2). Na druhej strane, mnohouholníky a mnohosteny nepatria do kategórie statických topológii, ale naopak do kategórie dynamických topológii, keďže jednotlivé bunky môžu mať štruktúru rôznych veľkostí, ktorá v čase prekladu známa nie je. Táto podstatná odlišnosť vyžadovala použitie iného formátu incidentných matíc pre ukladanie pod-entít a tiež zovšeobecnenie implementácie inicializácie siete, ktorá sa dokáže vysporiadať s entitami s dynamickou štruktúrou (viď kapitolu 3). Vďaka C++ šablónovým špecializáciám, bolo možné rozšíriť podporu o tieto dve nové topológie bez zmeny pôvodného návrhu v rámci už podporovaných statických topológii. To umožnilo zachovať efektivitu pôvodného návrhu pre statické topológie.

Ako vedľajší efekt pridania podpory pre mnohouholníkovú topológiu, bola tiež pridaná podpora pre statické topológie ihlana a klina. Pôvodná implementácia nepodporovala tieto dve topológie kvôli tomu, že vyžadujú použitie mnohouholníkovej topológie pre steny, keďže súčasne obsahujú trojuholníkové a štvoruholníkové steny.

Ako súčasť práce bola tiež implementovaná funkcionalita pre načítanie a zapisovanie mnohostenných sietí z/do súboru vo formáte FPMA, ktorý je súčasťou simulačného balíčka AVL FIRE™ [19]. Formát VTK [7], ktorý knižnica TNL čiastočne implementuje, síce mnohostenné siete podporuje, ale nie je to oficiálne dokumentované. Kvôli tomu bolo rozšírenie implementácie formátu VTK pre mnohostenné siete vynechané.

Pre účely experimentálnej analýzy výkonnosti v kapitole 5, boli tiež imple-

mentované algoritmy pre výpočet miery mnohouholníka a mnohostenu (viď sekciu 4.3). Ďalej sa tiež implementovalo niekoľko algoritmov pre dekompozíciu mnohouholníkových a mnohostenných sietí na trojuholníkové a štvorstenné siete (viď sekciu 4.1). Posledným implementovaným algoritmom, pre účely analýzy, bola korekcia nerovinných mnohouholníkov pre mnohouholníkovú a mnohostennú sieť (viď sekciu 4.2). Je tiež potrebné upresniť, že vyššie spomenuté implementované algoritmy, sú určené len pre konvexné mnohouholníky a mnohosteny.

Pre overenie správnosti implementovaného rozšírenia dátovej štruktúry (viď kapitolu 3) a implementácie algoritmov použitých pre experimentálne meranie (viď kapitolu 4), bolo implementovaných niekoľko unit testov (viď sekciu 5.1) pomocou knižnice *GoogleTest* [23], ktoré boli zakomponované medzi ostatné unit testy knižnice TNL. Okrem toho boli tiež implementované nástroje pre dekompozíciu a korekciu nerovinných mnohouholníkov 3D mnohouholníkových a mnohostenných sietí zo vstupného súboru, ktoré výslednú sieť zapíšu do výstupného súboru. Tieto nástroje boli tiež použité pre vytvorenie dekomponovaných sietí z mnohouholníkových a mnohostenných sietí, ktoré sa použili pri porovnaní rozšírenej dátovej štruktúry s pôvodnou implementáciou (viď sekciu 5.2.1).

Na záver sa v kapitole 5 vykonala experimentálna analýza výkonnosti implementovanej dátovej štruktúry. Porovnala sa efektivita novo implementovaných dynamických topológii s efektivitou statických topológii tak, že sa mnohouholníkové a mnohostenné siete porovnali s ich dekomponovanými variantami, t. j. trojuholníkovými a štvorstennými sieťami. Zistilo sa, že hlavnou výhodou nových topológii je menšia veľkosť siete (menší počet entít), čo malo za následok úsporu pamäte (trojuholníkové siete zaberali približne 1,7- až 2,5-krát viac pamäte ako mnohouholníkové siete a štvorstenné siete zaberali približne 6- až 20-krát viac pamäte ako mnohostenné siete). Menšia veľkosť siete mala tiež pozitívny vplyv na čas režijných operácií. Mnohouholníkové siete sa inicializovali približne 2- až 4-krát rýchlejšie ako trojuholníkové siete a mnohostenné siete sa inicializovali približne 7- až 20-krát rýchlejšie ako štvorstenné siete. Čo sa týka kópie medzi pamäťou CPU a GPU, mnohouholníkové siete sa kopírovali približne 2- až 3-krát rýchlejšie ako trojuholníkové siete a mnohostenné siete sa kopírovali približne 7- až 17-krát rýchlejšie ako štvorstenné siete. Načítanie siete zo súboru bolo pri mnohouholníkových sieťach približne až 2-krát rýchlejšie ako pri trojuholníkových sieťach, ale pri mnohostenných sieťach to bolo naopak približne až 2-krát pomalšie ako pri štvorstenných sieťach. Čo sa týka výkonu sietí pri úlohe výpočtu miery, mnohouholníkové siete dosahovali oproti dekomponovaným variantám na CPU podobnej paralelnej efektivity (v priemere približne 7-násobne zrýchlenie pri 8 vláknach) a pri výpočtoch na GPU dosahovali vyššej paralelnej efektivity (približne 60-násobné versus 40-násobné zrýchlenie oproti sekvenčnému výpočtu na CPU). Na druhej strane, mnohostenné siete dosahovali oproti dekomponovaným variantám podobnej paralelnej efektivity

na CPU (v priemere približne 7-násobne zrýchlenie pri 8 vláknach), ale na GPU bola efektivita výrazne nižšia (približne 20-násobné versus 80-násobne zrýchlenie oproti sekvenčnému výpočtu na CPU). Zistilo sa, že dôvodom nižšej efektivity na GPU bola výrazná divergencia vlákien, spôsobená nevyváženým množstvom práce susedných vlákien. Dôvodom nevyváženosti bolo, že jednotlivé susedné mnohosteny mali odlišnú veľkosť štruktúry (rôzne počty mnohouholníkových stien s rôznych počtami vrcholov), na ktorých závisela dĺžka výpočtu pre jednotlivé vlákna. Čo sa týka úloh dekompozície a korekcie nerovinných mnohouholníkov (viď sekcie 5.3.6 a 5.3.7), ktoré boli merané len na CPU, dosahovalo sa v priemere približne 5-násobného paralelného zrýchlenia pri 8 vláknach.

Práca do budúcnosti

Implementáciu zovšeobecnenia dátovej štruktúry je možno považovať za úplnú, ale ponúkajú sa ešte možnosti ďalších rozšírení. Prvé rozšírenie spočíva v implementácii dodatočných formátov pre načítanie mnohostenných sietí, ako napríklad variácií formátu FPMA (FPMB, FPMAZ, FPMBZ) a rozšírenia implementácie formátu VTK pre mnohostennú topológiu, ktorá nie je v dobe vzniku tejto práce oficiálne dokumentovaná.

Druhé rozšírenie spočíva v optimalizácií paralelnej efektivity na GPU pre mnohostenné, prípadne mnohouholníkové siete. Ako už bolo spomenuté, mnohostenné siete nedosahovali uspokojujúcej paralelnej efektivity kvôli divergencii vlákien spôsobenou výraznou odlišnosťou v počtu stien susedných mnohostenov. Bolo by zaujímavé napríklad preskúmať možnosti rôznych zoradení entít v sieti a zistiť, či by to nemalo pozitívny efekt na paralelnú efektivitu výpočtov na GPU pre mnohostenné, prípadne mnohouholníkové siete.

Tiež sa ponúka použitie samotnej dátovej štruktúry v komplikovaných vedeckých výpočtoch, ktoré zahrňujú mnohouholníkové a mnohostenné siete.

Literatúra

[1] T. Oberhuber, A. Suzuki, J. Vacata: New Row-grouped CSR format for storing sparse matrices on GPU with implementation in CUDA. *Acta Technica 4*, ročník 56, 2011: s. 447–466.

[2] M. Botsch, S. Steinberg, S. Bischoff, L. Kobbelt: Openmesh – a generic and efficient polygon mesh data structure. *1st OpenSG Symposium*, 2002.

[3] F. Rudolf, K. Rupp, J. Weinbub: Viennagrid 2.1.0 - user manual (2014) [online]. Dostupné z: `http://viennagrid.sourceforge.net/viennagrid-manual-current.pdf`, [cit. 2021-12-04].

[4] W. Bangerth, R. Hartmann, G. Kanschat: Deal.II—A General-Purpose Object-Oriented Finite Element Library. *ACM Trans. Math. Softw.*, ročník 33, č. 4, 2007: s. 24/1–24/27.

[5] H. Jasak, A. Jemcov, Ž. Tukovi: OpenFOAM: A C++ Library for Complex Physics Simulations. IUC Dubrovnik, Croatia: International workshop on coupled methods in numerical dynamics, 2007, s. 1–20.

[6] B. S. Kirk, J. W. Peterson, R. H. Stogner, G. F. Carey: `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers 22 (3–4)*, 2006: s. 237–254.

[7] W. Schroeder, K. Martin, B. Lorensen: The Visualization Toolkit: An Object-oriented Approach to 3D Graphics. *Kitware*, 2006.

[8] P. Cignoni, F. Ganovelli: Visualization and Computer Graphics Library (VCGlib) [online]. `http://vcg.isti.cnr.it/vcglib/`, [cit. 2021-12-04].

[9] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander: A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework. *Computing 82 (2-3)*, 2008: s. 103–119.

[10] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, O. Sander: A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE. *Computing 82 (2-3)*, 2008: s. 121–138.

[11] T. Oberhuber, J. Klinkovský, R. Fučík: TNL: Numerical Library for Modern Parallel Architectures. *Acta Polytechnica*, ročník 61, 02 2021: s. 122–134, doi:10.14311/AP.2021.61.0122.

[12] NVIDIA: CUDA Toolkit Documentation, version 11.5.0 [online]. Dostupné z: `https://docs.nvidia.com/cuda/archive/11.5.0/index.html`, 2021.

[13] A. Monakov, A. Lokhmotov, A. Avetisyan: Automatically tuning sparse matrix-vector multiplication for GPU architectures. *High Performance Embedded Architectures and Compilers*, 2010: s. 111—-125.

[14] AMD: ROCm™ Platform. Dostupné z: `https://rocmdocs.amd.com/en/latest/`.

[15] Nidito, F.: Replacing virtual methods with templates [online]. Dostupné z: `http://groups.di.unipi.it/~nids/docs/templates_vs_inheritance.html`, [cit. 2021-12-04].

[16] M. Harris: Optimizing parallel reduction in CUDA. *Nvidia developer technology 2.4 (2007)*: str. 70.

[17] M. Harris, S. Shub-habrata, J. D. Owens: Parallel prefix sum (scan) with CUDA. *GPU gems 3.39 (2007)*: s. 851–876.

[18] T. Oberhuber, J. Klinkovský: TNL project documentation [online]. Dostupné z: `https://tnl-project.org/documentation`.

[19] AVL: AVL FIRE™. Dostupné z: `https://www.avl.com/fire/`.

[20] L. Reid: Simple Mesh Tessellation & Triangulation Tutorial [online]. [cit. 2021-12-04]. Dostupné z: `https://lindenreidblog.com/2017/12/03/simple-mesh-tessellation-triangulation-tutorial/`

[21] M. Nelson: Consistent Subdivision of Convex Polyhedra into Tetrahedra. *Journal of Graphics Tools*, ročník 6, 01 2001: s. 29–36, doi: 10.1080/10867651.2001.10487543.

[22] D. Sunday: *Practical Geometry Algorithms: With C++ Code*. Amazon Digital Services LLC - KDP Print US, 2021, ISBN 9798749449730. Dostupné z: `https://books.google.sk/books?id=YBN0zgEACAAJ`

[23] Google LLC: GoogleTest [comp. software], Verzia 1.10.0. 2021, [cit. 2021-12-04]. Dostupné z: `https://github.com/google/googletest/`

Dodatok A

Zoznam použitých skratiek

2D Dvoj dimenzionálne

3D Troj dimenzionálne

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

GPGPU General-purpose Computing on Graphics Processing Units

GPU Graphics Processing Unit

HPC High Performance Computing

SIMD Single Instruction, Multiple Data

TNL Template Numerical Library

Dodatok B

Definície konfigurácií sietí použitých pri meraní

B.1 Maximálna konfigurácia

```
template< typename Cell,
          int SpaceDimension = Cell::dimension,
          typename Real = double,
          typename GlobalIndex = int,
          typename LocalIndex = GlobalIndex >
struct FullConfig
{
  using CellTopology = Cell;
  using RealType = Real;
  using GlobalIndexType = GlobalIndex;
  using LocalIndexType = LocalIndex;
  static constexpr int spaceDimension = SpaceDimension;
  static constexpr int meshDimension = Cell::dimension;

  static constexpr bool subentityStorage( int entityDim,
                                          int subentityDim )
  { return true; }

  static constexpr bool superentityStorage( int entityDim,
                                            int superentityDim )
  { return true; }

  static constexpr bool entityTagsStorage( int entityDim )
  { return true; }

  static constexpr bool dualGraphStorage()
  { return true; }

  static constexpr int dualGraphMinCommonVertices = meshDimension;
};
```

B.2 Minimálna konfigurácia

```
template< typename Cell,
          int SpaceDimension = Cell::dimension,
          typename Real = double,
          typename GlobalIndex = int,
          typename LocalIndex = GlobalIndex >
struct MinimalConfig
{
  using CellTopology = Cell;
  using RealType = Real;
  using GlobalIndexType = GlobalIndex;
  using LocalIndexType = LocalIndex;
  static constexpr int spaceDimension = SpaceDimension;
  static constexpr int meshDimension = Cell::dimension;

  static constexpr bool subentityStorage( int entityDim,
                                          int subentityDim )
  { return subentityDim == 0 ||
         ( subentityDim == meshDimension - 1 &&
           entityDim == meshDimension ); }

  static constexpr bool superentityStorage( int entityDim,
                                            int superentityDim )
  { return ( entityDim == 0 ||
             entityDim == meshDimension - 1 ) &&
             superentityDim == meshDimension; }

  static constexpr bool entityTagsStorage( int entityDim )
  { return false; }

  static constexpr bool dualGraphStorage()
  { return false; }

  static constexpr int dualGraphMinCommonVertices = meshDimension;
};
```

Dodatok C

Príklad inicializácie siete pomocou triedy MeshBuilder

C.1 Štvoruholníková sieť

Štvoruholníkovú sieť z obrázka 2.2, pričom je predpokladaná jednotkovosť obidvoch štvoruholníkov a používa sa arbitrárna konfigurácia definovaná triedou `QuadrangleMeshConfig`, je možné pomocou triedy `MeshBuilder` vytvoriť nasledovne:

```
using QuadrangleMesh = Mesh< QuadrangleMeshConfig >;
QuadrangleMesh mesh;
MeshBuilder< QuadrangleMesh > meshBuilder;

meshBuilder.setEntitiesCount( 6, 2 );

meshBuilder.setPoint( 0, { 0.0, 0.0 } );
meshBuilder.setPoint( 1, { 1.0, 0.0 } );
meshBuilder.setPoint( 2, { 1.0, 1.0 } );
meshBuilder.setPoint( 3, { 0.0, 1.0 } );
meshBuilder.setPoint( 4, { 2.0, 0.0 } );
meshBuilder.setPoint( 5, { 2.0, 1.0 } );

//   0     1     2     3
meshBuilder.getCellSeed( 0 ).setCornerId( 0, 0 );
meshBuilder.getCellSeed( 0 ).setCornerId( 1, 1 );
meshBuilder.getCellSeed( 0 ).setCornerId( 2, 2 );
meshBuilder.getCellSeed( 0 ).setCornerId( 3, 3 );

//   1     4     5     2
meshBuilder.getCellSeed( 1 ).setCornerId( 0, 1 );
meshBuilder.getCellSeed( 1 ).setCornerId( 1, 4 );
meshBuilder.getCellSeed( 1 ).setCornerId( 2, 5 );
meshBuilder.getCellSeed( 1 ).setCornerId( 3, 2 );

meshBuilder.build( mesh );
```

C.2 Mnohostenná sieť

Mnohostennú sieť, ktorá obsahuje 1 bunku reprezentujúcu jednotkovú kocku a používa arbitrárnu konfiguráciu definovanú triedou `PolyhedronMeshConfig`, je možné pomocou triedy `MeshBuilder` vytvoriť napríklad nasledovne:

```
1  using PolyhedronMesh = Mesh< PolyhedronMeshConfig >;
2  PolyhedronMesh mesh;
3  MeshBuilder< PolyhedronMesh > meshBuilder;
4
5  meshBuilder.setEntitiesCount( 8, 1, 6 );
6
7  meshBuilder.setPoint( 0, { 0.0, 0.0, 0.0 } );
8  meshBuilder.setPoint( 1, { 1.0, 0.0, 0.0 } );
9  meshBuilder.setPoint( 2, { 1.0, 1.0, 0.0 } );
10 meshBuilder.setPoint( 3, { 0.0, 1.0, 0.0 } );
11 meshBuilder.setPoint( 4, { 0.0, 0.0, 1.0 } );
12 meshBuilder.setPoint( 5, { 1.0, 0.0, 1.0 } );
13 meshBuilder.setPoint( 6, { 1.0, 1.0, 1.0 } );
14 meshBuilder.setPoint( 7, { 0.0, 1.0, 1.0 } );
15
16 meshBuilder.setFaceCornersCounts( { 4, 4, 4, 4, 4, 4 } );
17
18 //   0     1     2     3
19 meshBuilder.getFaceSeed( 0 ).setCornerId( 0, 0 );
20 meshBuilder.getFaceSeed( 0 ).setCornerId( 1, 1 );
21 meshBuilder.getFaceSeed( 0 ).setCornerId( 2, 2 );
22 meshBuilder.getFaceSeed( 0 ).setCornerId( 3, 3 );
23
24 //   1     5     6     2
25 meshBuilder.getFaceSeed( 1 ).setCornerId( 0, 1 );
26 meshBuilder.getFaceSeed( 1 ).setCornerId( 1, 5 );
27 meshBuilder.getFaceSeed( 1 ).setCornerId( 2, 6 );
28 meshBuilder.getFaceSeed( 1 ).setCornerId( 3, 2 );
29
30 //   5     4     7     6
31 meshBuilder.getFaceSeed( 2 ).setCornerId( 0, 5 );
32 meshBuilder.getFaceSeed( 2 ).setCornerId( 1, 4 );
33 meshBuilder.getFaceSeed( 2 ).setCornerId( 2, 7 );
34 meshBuilder.getFaceSeed( 2 ).setCornerId( 3, 6 );
35
36 //   4     0     3     7
37 meshBuilder.getFaceSeed( 3 ).setCornerId( 0, 4 );
38 meshBuilder.getFaceSeed( 3 ).setCornerId( 1, 0 );
39 meshBuilder.getFaceSeed( 3 ).setCornerId( 2, 3 );
40 meshBuilder.getFaceSeed( 3 ).setCornerId( 3, 7 );
41
42 //   0     1     5     4
43 meshBuilder.getFaceSeed( 4 ).setCornerId( 0, 0 );
44 meshBuilder.getFaceSeed( 4 ).setCornerId( 1, 1 );
45 meshBuilder.getFaceSeed( 4 ).setCornerId( 2, 5 );
46 meshBuilder.getFaceSeed( 4 ).setCornerId( 3, 4 );
47
48
```

```
49 //    3       2       6      7
50 meshBuilder.getFaceSeed( 5 ).setCornerId( 0, 3 );
51 meshBuilder.getFaceSeed( 5 ).setCornerId( 1, 2 );
52 meshBuilder.getFaceSeed( 5 ).setCornerId( 2, 6 );
53 meshBuilder.getFaceSeed( 5 ).setCornerId( 3, 7 );
54
55 meshBuilder.setCellCornersCounts( { 6 } );
56
57 //    0       1       2       3       4       5
58 meshBuilder.getCellSeed( 0 ).setCornerId( 0, 0 );
59 meshBuilder.getCellSeed( 0 ).setCornerId( 1, 1 );
60 meshBuilder.getCellSeed( 0 ).setCornerId( 2, 2 );
61 meshBuilder.getCellSeed( 0 ).setCornerId( 3, 3 );
62 meshBuilder.getCellSeed( 0 ).setCornerId( 4, 4 );
63 meshBuilder.getCellSeed( 0 ).setCornerId( 5, 5 );
64
65 meshBuilder.build( mesh );
```

Dodatok D

Príklad súboru vo formáte FPMA

Mnohostennú sieť s 1 bunkou (jednotková kocka), ktorej inicializáciu pomocou triedy `MeshBuilder` je možné vidieť v prílohe C.2, je v súbore vo formáte FPMA reprezentovaná nasledovne:

```
# single-line comments start with "#"

# 7: number of vertices
7
# list of vertex coordinates (x,y,z)-order
0.0 0.0 0.0
1.0 0.0 0.0
1.0 1.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
1.0 0.0 1.0
1.0 1.0 1.0
0.0 1.0 1.0
# 6: number of faces
6
# list of face seeds, where each line starts
# with the number of vertices of a face
4 0 1 2 3
4 1 5 6 2
4 5 4 7 6
4 4 0 3 7
4 0 1 5 4
4 3 2 6 7
# 1: number of cells
1
# list of cell seeds, where each line starts
# with the number of faces of a cell
6 0 1 2 3 4 5
```

Dodatok E

Obsah priloženého CD

```
readme.txt ....stručný popis obsahu CD + pokyny k vykonaniu merania
tnl ......................knižnica TNL s implementovaným rozšírením
text .......................................................text práce
  DP_bobot_jan_2021.pdf ..................text práce vo formáte PDF
  src ....................zdrojová forma textu práce vo formáte LaTeX
benchmarks ......................................súbory pre meranie
  data ......................................súbory meraných sietí
  scripts ............................skripty pre vykonanie merania
```