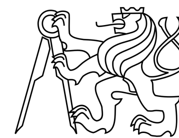
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název: Julia rozhraní pro knihovnu Triangle
Student: Martin Kuzma
Vedoucí: Ing. Tomáš Kalvoda, Ph.D.
Studijní program: Informatika
Studijní obor: Softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce letního semestru 2016/17

Pokyny pro vypracování

1. Seznamte se programovacím jazykem Julia a C knihovnou Triangle. Zejména nastudujte rozdíly mezi datovými typy jazyka Julia a C. Dále se seznamte s metodami umož ujícími volat C funkce z prost edí Julia.
2. Popište algoritmus použitý v knihovn Triangle pro konstrukci Delaunayho triangulace.
3. Navrhn te rozhraní (balí ek) umož ující uživateli z prost edí Julia snadno p istupovat k funkcím z knihovny Triangle. P i návrhu využijte znalostí z p edchozího bodu zadání. Tento balí ek implementujte.
4. Balí ek d kladn otestujte a dokumentujte. Vytvo te automatizované testy kontrolující funk nost implementace.

Seznam odborné literatury

1. de Berg, Mark; Otfried Cheong; Marc van Kreveld; Mark Overmars, Computational Geometry: Algorithms and Applications. Springer-Verlag, 2008.
2. Dokumentace jazyka Julia, http://julialang.org/
3. Dokumentace Triangle, https://www.cs.cmu.edu/ quake/triangle.html

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 27. listopadu 2015

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra softwarového inženýrství

Bakalářská práce

Julia rozhraní pro knihovnu Triangle

Martin Kuzma

Vedoucí práce: Ing. Tomáš Kalvoda, Ph.D.

10. ledna 2017

Poděkování

Děkuji svému vedoucímu Tomášovi Kalvodovi za neocenitelné rady při psaní této bakalářské práce. Dále děkuji Jonathanovi Shewchukovi za poskytnutí knihovny Triangle a její kvalitní dokumentace. Poděkování patří i Miroslavovi Hrončokovi za skvělý návod pro psaní bakalářské práce a za poskytnutí šablony. Své díky si zaslouží i uživatelé jazyka Julia, kteří mi vždy ochotně pomohli, při problémech s implementací.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. ledna 2017 …………………

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2017 Martin Kuzma. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

KUZMA, Martin. *Julia rozhraní pro knihovnu Triangle*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Cílem této práce je navrhnout a implementovat rozhraní pro knihovnu Triangle v jazyce Julia.

Triangle je knihovna napsaná v jazyce C a umožňuje generovat mimo jiné Delaunayho triangulace. Jejím autorem je Jonathan Shewchuk.

Výsledkem je balíček, který umí pracovat s Triangle v jazyce Julia. Balíček také umožňuje vygenerovat graf triangulace. Testování se provádí jednotkovými testy.

Klíčová slova Triangle, Jonathan Shewchuk, CTriangle, CTriangle.jl, rozhraní pro Triangle v Julia

Abstract

The goal of this thesis is to implement an interface in Julia for the Triangle library.

Triangle is written in C and can genarate Delaunay triangulations. The author of Triangle is Jonathan Shewchuk.

The result of this thesis is a package that can work with Triangle in Julia. Graph of the triangulation can be generated also. The package is tested using unit tests.

Keywords Triangle, Jonathan Shewchuk, CTriangle, CTriangle.jl, interface for Triangle in Julia

Obsah

Úvod **21**

1 Úvod do problematiky **23**

1.1 Výpočetní geometrie 23

2 Analýza **35**

2.1 Knihovna Triangle 35

2.2 Komunikace mezi Triangle a Julia 43

2.3 Vytvoření balíčku CTriangle 47

2.4 Nefunkční požadavky 49

2.5 Funkční požadavky 50

3 Návrh **53**

3.1 Generování triangulací 53

3.2 Vstup a výstup 53

3.3 Generování grafů 68

4 Realizace **71**

4.1 Parsování přepínačů 71

4.2 Komunikace s Triangle 72

4.3 Čtení vstupních souborů 74

4.4 Generování grafu 76

5 Testování **79**

5.1 Testování s TravisCI 81

Závěr **83**

Zdroje **85**

A Seznam použitých zkratek **89**

B Obsah přiloženého média **91**

Seznam ukázek kódu

2.1 Funkce triangulate pro komunikaci s Triangle 43
2.2 Struktura triangulateio pro vstup a výstup 44
2.3 Funkce pro získání ukazatele na pole typu Vector [4] 45
2.4 Funkce pro převod ukazatele na pole typu Vector [5] 45
2.5 Funkce pro zavolání funkce z C knihovny [6] 46
2.6 Složený typ triangulateio pro vstup a výstup 46
2.7 Konfigurace git repositáře 48
2.8 Soubor build.jl - instalace Triangle do CTriangle 50
2.9 Soubor deps.jl - cesta k souboru triangle.so 51
2.10 Vložení souboru deps.jl do CTriangle.jl 51

3.1 Funkce pro triangulace 61
3.2 Funkce pro triangulace 69

4.1 Definice typů a funkcí v souboru Includes.jl 72
4.2 Parsování přepínačů 73
4.3 Komunikace s Triangle 75
4.4 Parser 76
4.5 parsování řádky 77
4.6 Generování .dat souboru pro vrcholy 78

5.1 Příklad použití techniky *mocking* 80

Seznam adresářových struktur

2.1 Struktura balíčku CTriangle 47
2.2 Umístění binárních závislostí 49
4.1 Organizace kódu 72
B.1 Obsah přiloženého média 91

Seznam algoritmů

1 DELAUNAY-TRIANGULATION(P) 29
2 LEGALIZE-EDGE($p_r, \overline{p_i p_j}, T$) 30

Úvod

Julia je dynamicky typovaný programovací jazyk. O její vznik se zasloužili Jeff Bezanson, Stefan Karpinski, Viral Shah a Alan Edelman. Julia je nový stále se vyvíjející jazyk vytvořený speciálně pro vědecké a matematické výpočty. V době psaní této bakalářské práce je aktuální stabilní verze jazyka 0.5.

Julia byla vytvořená s myšlenkou odstranit potřebu tzv. prototypování. Programátoři řešící numericky náročné výpočty často volí pro první funkční verzi (prototyp) jazyky jako jsou python, R, MATLAB, které umožňují napsat program velmi rychle ale za cenu pomalejšího výsledku. Pro zrychlení programu pak přepisují své řešení v jazycích C nebo FORTRAN.

Cílem této práce je zanalyzovat, navrhnout a implementovat rozhraní pro knihovnu Triangle v jazyce Julia. Výsledkem je funkční balíček, který umí pracovat s knihovnou Triangle. Knihovna je program v jazyce C a umí mimo jiné generovat Delaunayho triangulace, jak se v práci dozvíme. Delaunayho Triangulace má široké využití v mnoha oborech.

Tím že zakomponujeme Triangle do Julia, otevřeme tak dveře potencionálním uživatelům, kteří chtějí pracovat s Triangle, ale nevyhovuje jim jazyk C.

V teoretické části se věnujeme oboru výpočetní geometrie. To nám poslouží jako teoretický základ pro pochopení knihovny Triangle a Delaunayho triangulace. V analýze prozkoumáme rozhraní knihovny a způsob, jakým ji integrujeme do Julia.

Praktická část se skládá z návrhu rozhraní balíčku, jeho implementace. Nechybí testování ani dokumentace.

Kapitola 1

Úvod do problematiky

Před tím než se pustíme do analýzy samotné knihovny Triangle, musíme nejprve porozumět problémům z oblasti, které knihovna Triangle řeší. Tato kapitola je rozdělena na dvě části. V první části se seznámíme s oborem Výpočetní geometrie a definujeme pojmy potřebné pro pochopení Triangle. A protože je Julia poměrně nový programovací jazyk podíváme se v druhé části na Julii.

1.1 Výpočetní geometrie

Geometrie je obor, který se zabývá měřením a popisem různých vlastností těles jako jsou například objem, povrch, tvar, vzájemná poloha těles vůči sobě. Geometrická tělesa jako jsou body, přímky a polygony slouží jako základ pro široké množství aplikací a umožňují existenci zajímavé skupiny problémů.

Rozvoj informačních technologií v druhé polovině minulého století umožnil vznik řadě algoritmů, které mnohé z těchto problémů řeší. Disciplína, která se zabývá těmito problémy a algoritmy, se nazývá Výpočetní geometrie.

Přelomovým rokem pro Výpočetní geometrii se stal rok 1978. Tento rok napsal Michael Shamos diplomovou práci s názvem Výpočetní geometrie

(*Computational geometry*)[1], která přitáhla značnou pozornost. V této práci se zabýval analýzou a návrhem efektivních algoritmů pro různé geometrické problémy.

V roce 1985, byla vydána první knížka Úvod do výpočetní geometrie (*Computational geometry an introduction*)[2] a časopis Diskrétní a Výpočetní geometrie (*Discrete And Computational Geometry*)[3] pod vydavatelstvím Springer-Verlag. Uskutečnila se také první konference ACM Symposia na téma Výpočetní geometrie.

Výpočetní geometrie je důležitá z praktického hlediska, protože se s její pomocí dá řešit mnoho problémů z reálného světa. Metody výpočetní geometrie se využívají při řešení problémů z oblasti rozpoznávání vzorů, počítačová grafika, zpracování obrazů, operační výzkum. Velké množství výrobních problémů zahrnuje rozmístění vodičů, umístění různých zařízení, dělení materiálů (*cutting-stock problem*). Více o algoritmech a Výpočetní geometrii si můžete přečíst v [12].

V následujících odstavcích se seznámíme s pojmy konvexní obal, Voroného diagram, Delaunayho triangulace a Delaunayho zjemnění. Budeme se zabývat pouze jejich verzemi v $\mathbb{R}^2$.

1.1.1 Konvexní obal

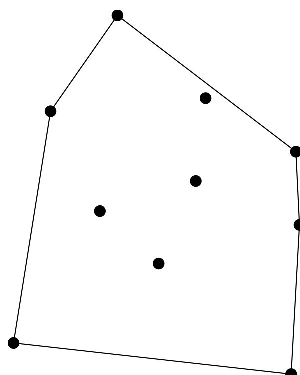
Definice 1 ([14], Konvexní obal). *O množině bodů řekneme, že je konvexní, pokud pro každé dva body obsahuje i celou úsečku mezi nimi. Konvexní obal dané množiny bodů M je nejmenší konvexní množina, která obsahuje všechny body množiny M.*

Příklad konvexního obalu můžeme vidět na obrázku 1.1. Problém nalezení konvexního obalu je jedním ze základních problémů ve výpočetní geometrii a má řadu praktických využití. Konvexní obal své využití najde například při plánování pohybu robotů a detekce jejich kolizí, analýze rozptylů a odhadů

[1]Diplomová práce je dostupná z `http://euro.ecom.cmu.edu/people/faculty/mshamos/1978ShamosThesis.pdf`.

[2]Knížka je dostupná z `http://link.springer.com/book/10.1007%2F978-1-4612-1098-6`.

[3]Časopis je dostupný z `http://link.springer.com/journal/454`.



Obrázek 1.1: Konvexní obal

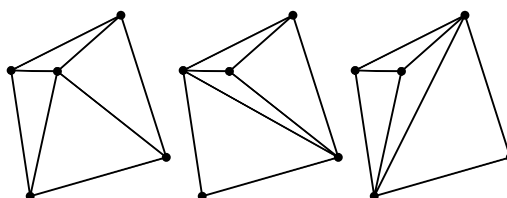
ve statistice [1]. V uvedeném zdroji najdeme i rozbor algoritmů pro nalezení konvexního obalu.

1.1.2 Triangulace

Triangulace obecně znamená spojení množinu bodů v rovině tak, aby vznikly trojúhelníky. Žádné dva trojúhelníky se ale nesmí překrývat. Pro přesnost uvedeme i formální definici.

Definice 2 ([16, 10]). *Triangulace* T *konečné množiny bodů* $M \in \mathbb{R}^2$ *je rozklad množiny M na trojúhelníky* $\mathrm{T} = t_i$, *kde* $i \in \mathbb{N}$. *Dva sousední trojúhelníky mají společný nejvýše jeden vrchol nebo hranu. Dále musí platit, že sjednocení těchto tvoří formuje konvexní obal množiny M.*

Obrázek 1.2 ukazuje možnosti, jak triangulovat takovou množinu bodů. Na ob-



Obrázek 1.2: Tři možné triangulace rovinné oblasti.

rázku 1.2 vidíme, že existuje více způsobů, jak nějakou množinu bodů triangulovat. Někdy se nám ale hodí, aby triangulace splňovala určité vlastnosti. Například nechceme mít v triangulaci příliš hubené a dlouhé trojúhelníky.

Definice 3 ([3], úhlový vektor). *Nechť* T *je triangulace mnoziny bodů* M *s* m *trojúhelníky.* $A(\mathrm{T}) = (\alpha_1, ..., \alpha_{3m})$ *kde* $\alpha_1, ..., \alpha_{3m}$ *jsou úhly v* T *seřazené vzestupně, nazveme* ***úhlovým vektorem*** *triangulace* T.

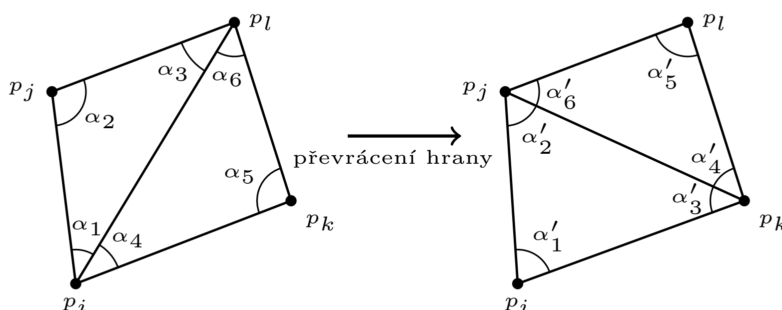
Definice 4 ([3]). *Nechť* T' *je další triangulace množiny bodů M, pro kterou platí:* $\mathrm{T}' \neq \mathrm{T}$. *Definujeme* $A(\mathrm{T}) > A(\mathrm{T}')$ *pokud je* $A(\mathrm{T})$ *lexikograficky větší než* $A(\mathrm{T}')$.

Definice 5 ([3], optimální triangulace). *Triangulaci* T *nazveme optimální vzhledem k úhlovému vektoru* $A(\mathrm{T})$*, pokud* $A(\mathrm{T}) \geq A(\mathrm{T}')$.

Na obrázku 1.3 můžeme vidět, že přehozením hrany $\overline{p_i p_l}$ na hranu $\overline{p_j p_k}$ došlo ke zvětšení úhlového vektoru, protože

$$\min_{1 \leq i \leq 6} \alpha_i \leq \min_{1 \leq i \leq 6} \alpha_i'$$

. Jinými slovy došlo k lokálnímu zvětšení minimálního úhlu. Hrany, jejíchž převrácením dojde ke zvětšení úhlového vektoru nazveme *ilegálními* [3]. Trian-



Obrázek 1.3: Zvětšení úhlového vektoru převrácením hrany

gulace, která eliminuje tyto ilegální hrany se nazývá Delaunayho triangulace. Formální definici uvádíme níže.

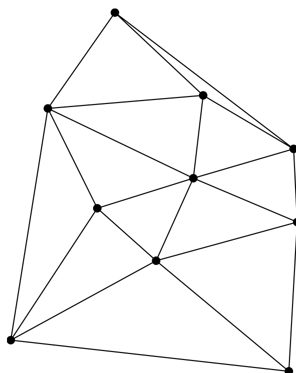
1.1.3 Delaunayho triangulace

Delaunayho triangulaci objevil v roce 1934[4]ruský uznávaný matematik Boris Nikolaevich Delaunay. Kromě matematiky se Boris věnoval i horolezectví a v roce 1937 vydal příručku pro horolezce *Summits of the Western Caucasus* [9].

[4]Práce je dostupná ve francouštině z `http://www.mathnet.ru/links/598dd842c40d826ddc2e9e682ec197f5/im4937.pdf`.

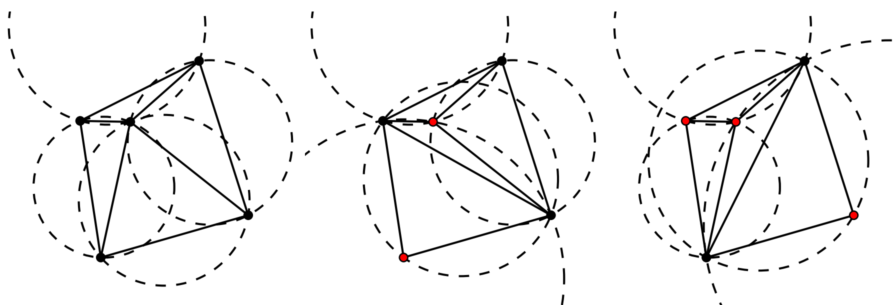
Definice 6 ([16]). *Triangulaci T konečné množiny bodů M nazveme* ***Delaunayho triangulací*** *pokud kružnice opsaná každého trojúhelníku v T je prázdná. To znamená, že žádný bod z M neleží uvnitř této kružnice.*

Na obrázku1.4 můžeme vidět příklad Delaunayho triangulace. Obrázek 1.5



Obrázek 1.4: Delaunayho triangulace

znázorňuje vlastnost Delaunayho triangulace. Můžeme vidět, že ze všech tří možných triangulací 1.2 je Delaunayho pouze ta vlevo, protože žádný z bodů neleží uvnitř žádné opsané kružnice trojúhelníku. V ostatních triangulacích jsou body, které porušují vlastnost Delaunayho triangulace označeny červeně.



Obrázek 1.5: Vlastnost Delaunayho triangulace

Delaunayho triangulace (zkráceně DT) má mnohá využití například pří modelování terénu nebo v tzv. metodě konečných prvků (*finite element method*). Existuje i několik algoritmů pro její konstrukci.

Ukážeme si pro představu jeden takový algoritmus, který najdeme v textu [3]. Pseudokódy 1, 2 naznačují, jak algoritmus funguje. Oba pseudokódy jsou převzaty plně z daného textu.

Ve stručnosti algoritmus funguje, tak že na začátku vytvoří trojúhelník obklopující množinu *P*. Ten vytvoří tak, že přidá dva body p_{-1} a p_{-2} dostatečně daleko od zbytku bodů množiny *P* a ty spojí s bodem $p_0 \in P$, který má nejvyšší souřadnici *y*.

Z tohoto trojúhelníku se vytvoří počáteční triangulace *T*. Zbytek bodů z $P \backslash p_0$ se prochází a vkládají se do triangulace. Pro každý tento bod se najde trojúhelník, ve kterém bod leží a rozdělí se tak na menší trojúhelníky.

Všechny hrany těchto trojúhelníků, které neincidují s vkládaným bodem se otestují pomocí funkce `LEGALIZE-EDGE` a pokud je potřeba tak se přehodí.

Na konci se odstraní body p_{-1} a p_{-2} se všemi jejich incidentními hranami. Výsledkem je *DT* množiny *P*. Přesnější popis algoritmu a důkaz jeho správnosti najdeme v textu [3].

1.1.4 *Constrained* Delaunayho triangulace

V praxi však potřebujeme triangulovat mnohdy složitější strukturu než jen množinu bodů. Můžeme například chtít triangulovat různé polygony a útvary, které nejsou nutně konvexní a mohou obsahovat díry a hrany, které musí triangulace zachovat. Těmto útvarům budeme říkat planární úsečkové grafy (*planar straight line graphs*) zkrácené PSLG. Povinné hrany nazveme segmenty, abychom je odlišili od hran triangulace.

DT, které berou na vstup PSLG se nazývají *Constrained* Delaunayho triangulace (zkráceně CDT). Nutno poznamenat, že některé trojúhelníky nemusí nutně splňovat vlastnost DT 1.5, protože musí brát v potaz segmenty. Pro přesnost uvedeme definici podle [10, 22].

Definice 7 (*Constrained Delaunayho triangulace*). *Nechť G je PSLG. Triangulace T , je Constrained Delaunayova triangulace G, jestliže každá hrana G je hrana z T a pro každou hranu e z T existuje kružnice C s následujícími parametry:*

- *Koncové body e jsou na kružnici C.*

Algoritmus 1 DELAUNAY-TRIANGULATION(*P*)

Vstup: množina *P* s $n+1$ vrcholy ve dvourozměrném prostoru
Výstup: *DT* množiny *P*

1: Nechť je bod p_0 lexikograficky nejvyšší bod. Má tedy největší souřadnici *y*.
2: Nechť jsou body p_{-1} a p_{-2} dostatečně daleko tak, že celá množina *P* leží v tojúhelníku $p_0 p_{-1} p_{-2}$.
3: Vytvoř triangulaci *T*, která obsahuje pouze trojúhelník $p_0 p_{-1} p_{-2}$
4: Vytvoř náhodnou permutaci $p_1, p_2, ..., p_n$ z množiny $P \setminus p_0$.
5: **for** $r = 1$ to n **do**
6: (* Vlož p_r do *T*. *)
7: Najdi trojúhelník $p_i p_j p_k \in T$, ve kterém leží p_r.
8: **if** p_r leží uvnitř trojúhelníku $p_i p_j p_k$. **then**
9: Veď hrany od p_r ke každému vrcholu trojúhelníku $p_i p_j p_k$. Trojúhelník $p_i p_j p_k$ se tak rozdělí na tři menší.
10: LEGALIZE-EDGE(p_r, $\overline{p_i p_j}$, *T*)
11: LEGALIZE-EDGE(p_r, $\overline{p_j p_k}$, *T*)
12: LEGALIZE-EDGE(p_r, $\overline{p_k p_i}$, *T*)
13: **else**(* p_r leží na hraně trojúhelníka $p_i p_j p_k$. Řekněme například, že leží na hraně $\overline{p_i p_j}$. *)
14: Veď hranu od p_r k p_k a třetímu bodu p_l protějšího trojúhelníka, který je s hranou $p_i p_j p_k$ také incidentní. Tyto dva trojúhelníky se tak rozdělí na menší čtyři.
15: LEGALIZE-EDGE(p_r, $\overline{p_i p_l}$, *T*)
16: LEGALIZE-EDGE(p_r, $\overline{p_l p_j}$, *T*)
17: LEGALIZE-EDGE(p_r, $\overline{p_j p_k}$, *T*)
18: LEGALIZE-EDGE(p_r, $\overline{p_k p_i}$, *T*)
19: **end if**
20: **end for**
21: Odstraň body p_{-1} a p_{-2} se všemi jejich incidentními hranami.
22: **return** T

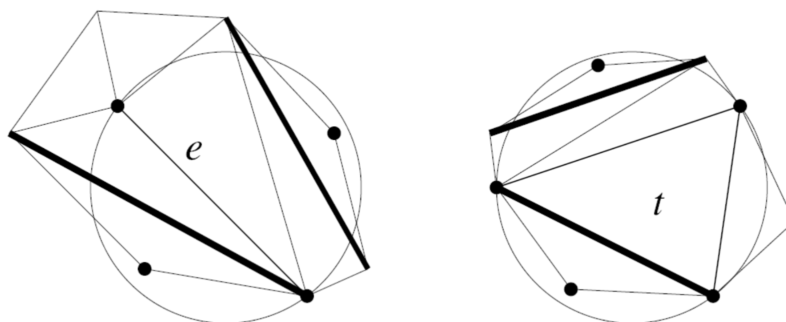
Algoritmus 2 LEGALIZE-EDGE($p_r, \overline{p_i p_j}, T$)

Vstup: p_r je vkládaný bod.
Vstup: $\overline{p_i p_j}$ je hrana, která se může převrátit.
Vstup: T je triangulace.
1: **if** $\overline{p_i p_j}$ je nelegální **then**
2: Nechť je $p_i p_j p_k$ trojúhelník sousední k trojúhelníku $p_r p_i p_j$ podél hrany $\overline{p_i p_j}$.
3: (* Přehoď $\overline{p_i p_j}$ *) Nahraď $\overline{p_r p_k}$ za $\overline{p_i p_j}$.
4: LEGALIZE-EDGE(p_r, $\overline{p_i p_k}$, T)
5: LEGALIZE-EDGE(p_r, $\overline{p_k p_j}$, T)
6: **end if**

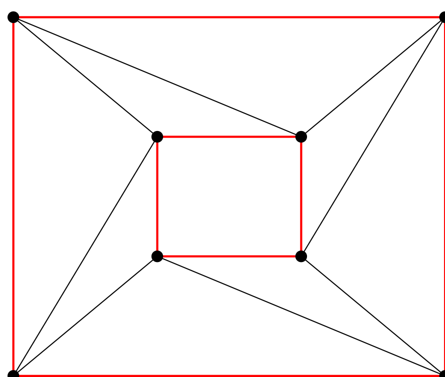
- *Jestliže některý vrchol z G je uvnitř C, potom nesmí být viděn nejméně z jednoho konců e. Tzn. jestliže nakreslíme úsečku z bodu v do každého z vrcholů e, potom nejméně jedna z přímek kříží hranu z G.*

Vrcholy z G se navzájem vidí, pokud mezi nimi neleží žádný segment. O trojúhelníku nebo hraně v CDT řekneme, že jsou *Constrained Delaunay* (CD), pokud jsou jejich vrcholy navzájem viditelné. Dále existuje taková kružnice pro kterou platí, že prochází hranou nebo trojúhelníkem a zároveň v ní neleží žádný vrchol, který by byl viděn z hrany nebo trojúhelníku. Názorný příklad ukazuje obrázek 1.6. V levé části vidíme hranu *e*, která je CD a vpravo trojúhelník *t*, který je CD. Segmenty jsou vyznačeny tučně.

Obrázek 1.7 znázorňuje příklad CDT. Segmenty jsou vyznačeny červené. Je také vidět, že čtyři vnitřní segmenty vyznačují díru. Podrobnější informace o CDT nejdeme v [22].



Obrázek 1.6: CD hrana *e* vlevo. CD trojúhelník *t* vpravo. zdroj [22]



Obrázek 1.7: *Constrained* Delaunayho triangulace

1.1.5 Voroného diagram

Podle [2] uvedeme definici Voroného diagramu.

Definice 8. *Voroného diagram V(P) představuje rozklad množiny bodů P na n otevřených či uzavřených oblastí V(P) =* $V(p_1), V(p_2), ..., V(p_n)$ *takových, že každý bod q* $\in V(p_i)$ *je blíže k bodu* p_i *než k jakémukoliv bodu* $p_j \in P$.

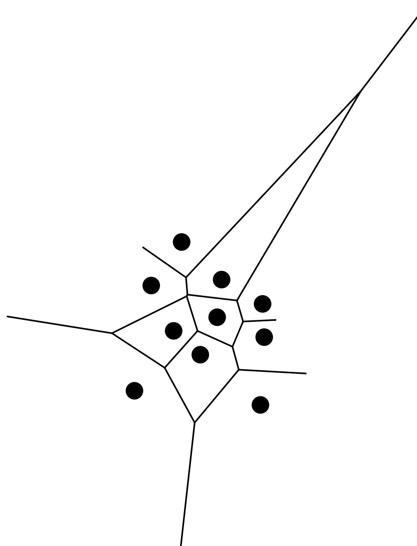
- *Uzavřenou oblast* $V(p_i)$ *nazveme Voroného buňkou.*
- *Pro libovolný bod q* $\in V(p_i)$ *a libovolnou buňku* $V(p_j)$ *platí:*

$$d(q, p_i) \leq d(q, p_j)$$

d značí vzdálenost mezi bodem q libovolnými body p_i *a* p_j.

Voroného diagram má také speciální vztah k DT. Voroného diagram a DT si jdou navzájem duálními grafy. Na obrázku 1.8 vidíme příklad Voroného diagramu. Obrázek 1.9 ukazuje vztah DT a Voroného diagramu. Vidíme, že středy kružnic opsaných trojúhelníků leží na spojnicích Voroného buněk.

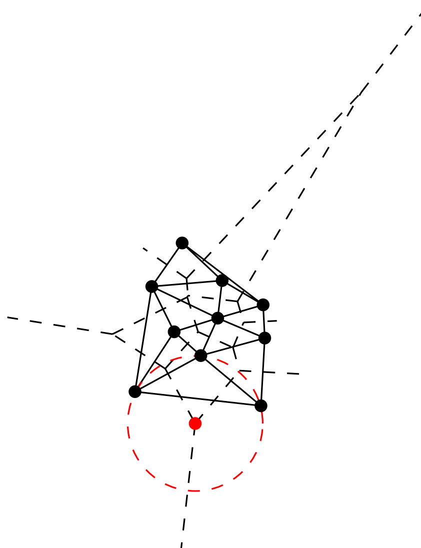
Voroného diagram je jedna z nejzákladnějších struktur výpočetní geometrie a používá se při řešení například poštovního problému [2]. Uvedený zdroj také rozebírá algoritmy pro konstrukci Voroného diagramu.



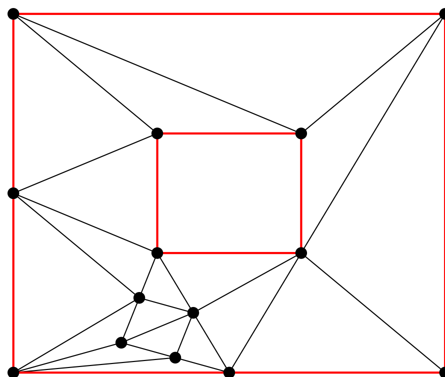
Obrázek 1.8: Voroného diagram

1.1.6 Delaunayho zjemnění

Při interpolaci metodou konečných těles nebo metodou konečných prvků chceme mít větší kontrolu nad vytvořenými trojúhelníky v DT nebo CDT. Technika Delaunayho zjemnění *Delaunay refinement* umožňuje trojúhelníkům omezit jejich velikost nebo jejich vnitřní úhly. Aby byly splněny podmínky kladené na trojúhelníky a zachované vlastnosti DT 6 a nebo CDT 7, pracují algoritmy Delaunayho zjemnění tak, že do CD nebo CDT přidávají vhodně vybrané body a oblast s těmito trojúhelníky se retrianguluje pomocí CD nebo CDT. Podrobnější informace o algoritmech Delaunayho zjemnění najdeme v [23]. Obrázek 1.10 znázorňuje zjemnění CDT z 1.7.



Obrázek 1.9: Voroného diagram a Delaunayho triangulace



Obrázek 1.10: zjemnění CDT

Kapitola 2

Analýza

Naším úkolem je implementovat balíček, který umožní pracovat s knihovnou Triangle v jazyce Julia. K tomu, abychom úspěšně vytvořili takový balíček, musíme nejdříve zjistit, jak Triangle funguje a co všechno umí počítat. Dále musíme přijít na to, jak zařídíme komunikaci mezi naším balíčkem a Triangle. V neposlední řadě se podíváme, jak se takový balíček do Julia implementuje. Nakonec si vymezíme funkční a nefunkční požadavky.

2.1 Knihovna Triangle

Triangle je program napsaný v jazyce C a jeho autorem je americký profesor Jonathan Shewchuck [19], který vyučuje na univerzitě Berkeley v Kalifornii. Triangle je volně dostupná. Licencovaná je samotným autorem. Pokud píšeme nějaký text související s Triangle, chce autor uvést v sekci poděkování. Pro komerční účely je potřeba se dohodnout se samotným autorem.

Triangle umí generovat Delaunayho triangulaci (*DT*), Constrained Delaunayho triangulaci (*CDT*), zjemnění Delaunayho triangulace (*ZDT*) a zjemnění Constrained Delaunayho triangulace (*ZCDT*). Mimo těchto triangulací umí Triangle vygenerovat k *DT* i Voroného diagram. Po dohodě s vedoucím této práce se jím dále nezabýváme a soustředíme se pouze na triangulace.

Chování Triangle ovlivňuje několik přepínačů. Celý jejich seznam najdeme v dokumentaci [18]. My se seznámíme jenom s těmi přepínači, které mají vliv na

vstupní data. Pro ně slouží několik souborů s pevně daným formátem. Než se pustíme do rozboru přepínačů a souborů, ujasníme si pár pojmů.

- **Segment** je hrana v *PSLG*, která musí zůstat po triangulaci zachována. Může se také jednat o hranu v konvexním obalu triangulace.

- **Díra** je oblast určená bodem, kterou triangulace musí vynechat. Díru ohraničují segmenty.

- **Atribut** je reálné číslo, které lze přiřadit trojúhelníku nebo bodu. Většinou slouží k reprezentaci nějaké jejich fyzikální vlastnosti. Pro samotný výpočet triangulace ale nemají atributy žádný význam.

- **Hraniční značka** je celé číslo. Které slouží k asociaci vrcholů a hran triangulace se segmenty. Označují také vrcholy a hrany triangulace, které leží na jejím okraji. O tom, proč můžou být hraniční značky užitečné si můžeme přečíst v dokumentaci [17].

- **Region** je oblast určená bodem, ve které každému trojúhelníku můžeme přiřadit atribut nebo omezení na maximální velikost trojúhelníku.

2.1.1 Vstupní soubory

Triangle pro vstup používá čtyři typy souborů. Z dokumentace v sekci Formáty souborů (*File formats*) [20] se můžeme dočíst, že jsou to následující soubory.

- Soubor s koncovku `.node` slouží pro uložení vrcholů triangulace. Každému bodu lze přiřadit seznam atributů a hraniční značku.

  Na prvním řádku je hlavička. Hlavička obsahuje čtyři celá čísla označující po sobě, počet řádků (vrcholů), dimenzi v prostoru (vždy 2), počet atributů a příznak hraniční značky. Jednička znamená, že body mají hraniční značku. Nula značí opak.

  Následující řádky začínají číslem označující jejich pořadí (index vrcholu). Následuje x-ová a y-ová souřadnice vrcholu. Za vrcholem je seznam atributů, za kterým následuje hraniční značka. Atributy a hraniční značka nejsou povinné.

- Soubor s koncovkou `.poly`, který reprezentuje *PSLG*(vrcholy, segmenty, díry) a regiony. Je složen ze čtyř sekcí. První má stejný obsah jako soubor `.node`. Následuje sekce se segmenty.

  První řádek je hlavička s dvěma celými čísly, které označují počet řádek (segmentů) a příznak hraniční značky. Na každém dalším řádku je pak číslo řádku (index segmentu), index prvního a druhého vrcholu a hraniční značka. Pokud mají hraniční značku jenom některé segmenty, musí být u ostatních segmentů uvedena 0.

  Třetí sekce označuje díry. Začíná hlavičkou obsahující počet děr. Každý další řádek obsahuje index díry, x-ovou a y-ovou souřadnici bodu, který díru označuje.

  Poslední sekce představuje regiony. První řádek je zase hlavička s počtem regionů. Následující řádky začínají indexem regionu, x-ovou a y-ovou souřadnici bodu, který region označuje. Za souřadnicemi se nachází jedno nebo dvě čísla. První značí atribut a druhý omezení na maximální velikost trojúhelníku. Pokud se za souřadnicemi vyskytuje jen jedno číslo, tak se může jednat buď o atribut nebo omezení na velikost trojúhelníku.

  Sekce s dírami a regiony nejsou povinné. Sekce s dírami však musí mít vždy hlavičku. Pokud je sekce s vrcholy prázdná, znamená to, že je najdeme v odpovídajícím souboru `.node`.

- Soubor s koncovku `.ele` reprezentuje trojúhelníky triangulace. První řádek je hlavička, která se skládá ze tří čísel. První značí počet trojúhelníků. Druhé označuje počet vrcholů trojúhelníku. Třetí značí počet atributů.

  Každý další řádek reprezentuje trojúhelník. Řádek začíná indexem trojúhelníku. Následují indexy vrcholů a potom seznam atributů.

- Soubor s koncovku `.area` je určen k přiřazení maximální velikosti každému trojúhelníku. Formát má podobný jako ostatní soubory výše. První řádek je hlavička s počtem trojúhelníků. Každý další řádek začíná indexem trojúhelníku a pokračuje omezením na maximální velikost trojúhelníku.

> Soubor musí obsahovat maximální velikosti pro všechny trojúhelníky ze souboru `.ele`. Pokud nějaký trojúhelník nechceme omezovat, uvedeme velikost -1.

Všechna čísla v souborech jsou oddělena nějakým bílým znakem (mezerou, tabulátorem). Triangle podporuje číslování řádků jak od jedničky tak od nuly. Pro výpočet *DT* se data předávají v souboru `.node`. K výpočtu *CDT* potřebujeme soubor `.poly` případně `.node`. Pro výpočet *ZDT* složí soubory `.node`, `.ele` případně `.area`. K výpočtu *ZCDT* potřebujeme soubory `.poly`, `.ele`, případně `.node` nebo `.area`.

Právě přepínače určují, jaká triangulace se provádí a jaké soubory se tím pádem čtou.

2.1.2 Přepínače

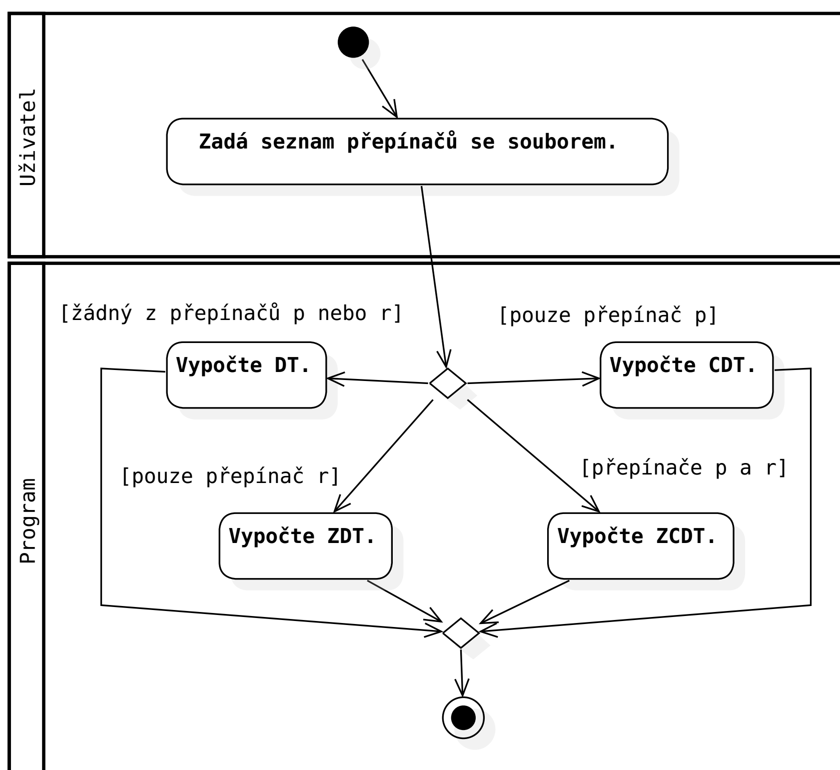
Důležité přepínače pro implementaci balíčku jsou přepínače *p*, *r*, *O*, *a*, *A*.

Kombinace přepínačů *p* a *r* nám určuje, která triangulace se má provádět. Všechny situace ukazuje diagram 2.1. Pro každý výpočet i víme, ve kterých souborech najdeme příslušná data. Čtení dat pro výpočet *ZCDT* ukazuje diagram 2.3. Diagram 2.2 pak ukazuje čtení dat pro výpočet *ZDT*. Pro výpočet *DT* nebo *CDT* stačí pouze přečíst soubor `.node` respektive `.poly`.

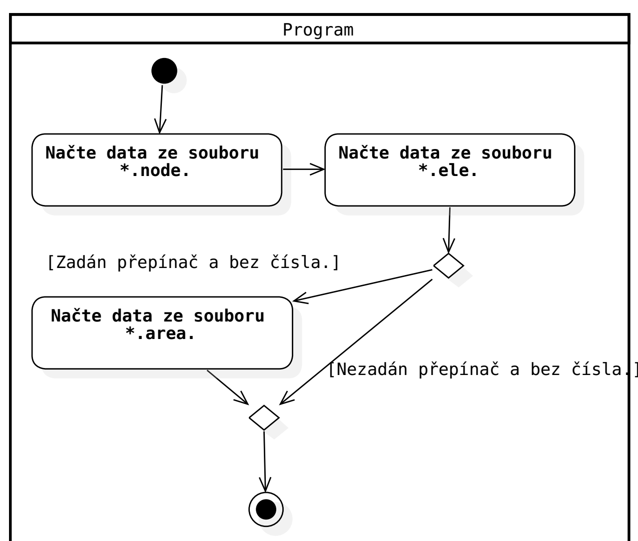
Soubor `.poly` se čte jak při výpočtu *CDT* tak *ZCDT*. V případě *ZCDT* se nečte sekce s regiony. Čtení souboru `.poly` ukazují diagramy 2.4 a 2.5. Přepínač *a* může následovat i číslo označující maximální velikost všech trojúhelníků. Nás zajímá pouze varianta bez čísla. Ta totiž říká, že velikosti najdeme v souboru `.poly` nebo v souboru `.area`. Záleží ns tom, jakou triangulaci zrovna provádíme.

Přepínač *O* znamená, že chceme ignorovat díry v souboru `.poly`. Přepínač *A* pak říká, že každému trojúhelníku v určitém regionu chceme přiřadit atribut ze souboru `.poly`.

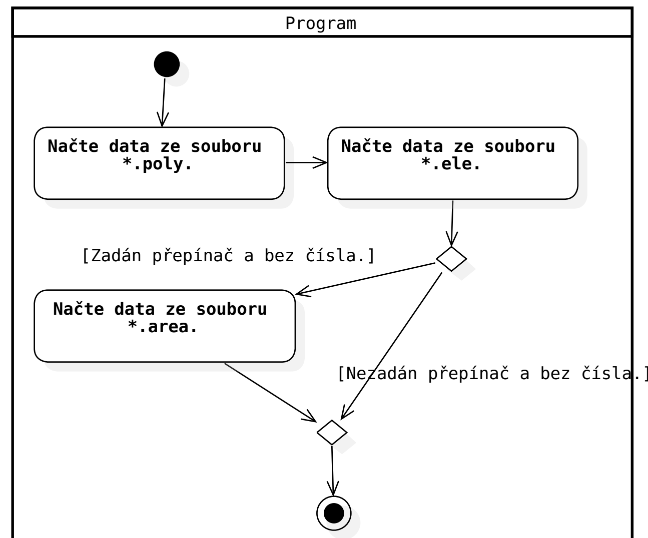
Už máme celkem dobrý přehled o tom, jak Triangle pracuje se vstupními daty a kdy se jaká triangulace provádí. Musíme se ještě podívat na to, jak se vůbec data do Triangle dostanou.



Obrázek 2.1: Určení typu triangulace podle přepínačů.



Obrázek 2.2: Čtení dat pro výpočet ZDT.



Obrázek 2.3: Čtení dat pro výpočet ZCDT.

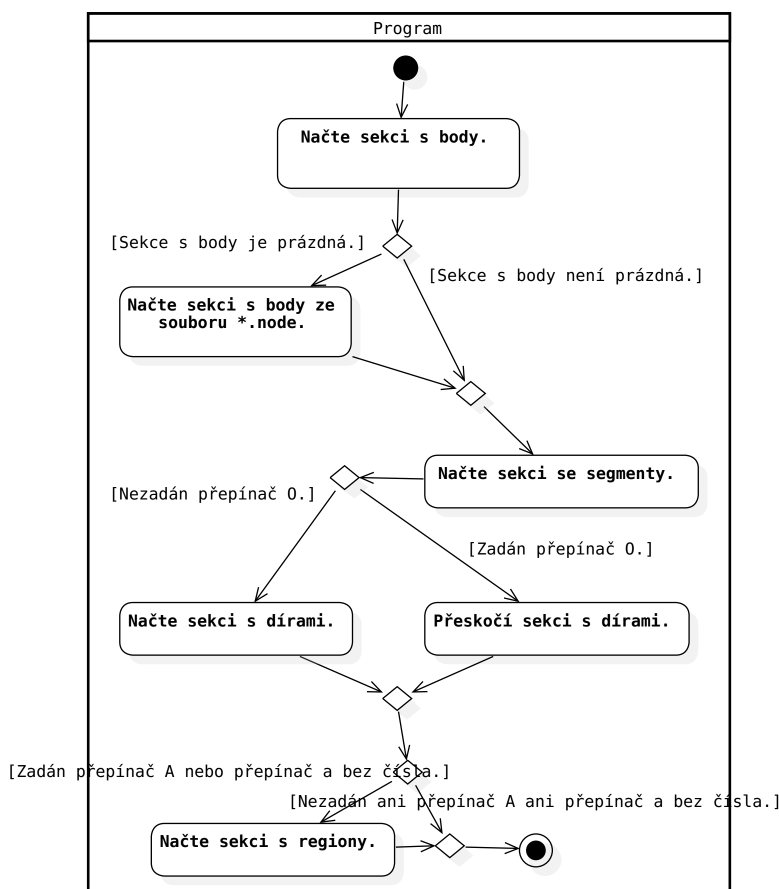
2.1.3 Rozhraní Triangle

Pro práci s Triangle máme k dispozici funkci `tringulate`2.1. Prvním parametrem se předává seznam přepínačů. Druhý až poslední parametr je ukazatel na strukturu 2.2. Druhým parametrem se předávají data pro triangulace. Třetí parametr slouží pro výstup samotného výpočtu. Čtvrtý parametr je určený pro vygenerování Voroného diagramu.

Poslední tři parametry funkce `tringulate` 2.1 jsou všechny stejného typu i přesto, že se každý používá pro něco jiného. Musíme tak rozlišit, které atributy jsou pro vstup a které pro výstup. Popis najdeme v hlavičkovém souboru `triangle.h`. Ten najdeme v dokumentaci [21]. Všechna data ve vstupu a výstupu jsou jednorozměrná pole.

Vstupní pole jsou následující.

- `pointlist` - seznam vrcholů triangulace. Pro každý vrchol uvádíme dvě souřadnice x a y v tomto pořadí. Počet vrcholů udává `numberofpoints`. Velikost pole je pak dvojnásobek počtu vrcholů.
- `pointattributelist` - seznam atributů vrcholů. Pro každý vrchol máme `numberofpointattributes` atributů. Velikost pole tedy zjistíme, tak že vynásobíme `numberofpoints` a `numberofpointattributes`.

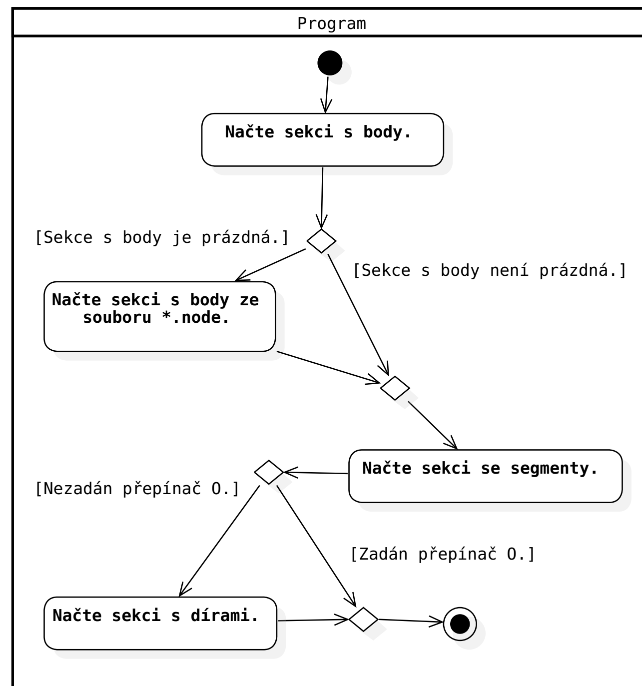


Obrázek 2.4: Čtení souboru .poly pro výpočet CDT.

- `pointmarkerlist` - seznam hraničních značek všech vrcholů. Pro každý vrchol máme jednu hraniční značku. Číslo `numberofpoints` udává velikost seznamu.

- `trianglelist` - seznam indexů vrcholů pro každý trojúhelník. Každý trojúhelník má `!numberofcorners!` vrcholů. Délku pole spočítáme vynásobením `numberofcorners` a `numberofcorners`.

- `triangleattributelist` - seznam atributů všech trojúhelníků. Pro každý trojúhelník máme `numberoftriangleattributes` atributů. Vynásobením `numberoftriangles` a `numberoftriangleattributes` získáme velikost seznamu.



Obrázek 2.5: Čtení souboru .poly pro výpočet ZCDT.

- `trianglearealist` - seznam velikostí všech trojúhelníků. Každému trojúhelníku patří jedna velikost. Číslo `numberoftriangles` tak udává délku seznamu.

- `segmentlist` - seznam segmentů. Pro každý segment uvádíme dva indexy vrcholů z `pointlist`. Velikost seznamu je pak dvojnásobek `numberofsegments`.

- `segmentmarkerlist` - seznam hraničních značek všech segmentů. Každému segmentu náleží jedna hraniční značka. Číslo `numberofsegments` označuje veliklost seznamu.

- `holelist` - seznam děr. Pro každou díru uvádíme dvě souřadnice x a y. Délka seznamu je tak dvojnásobek `numberofholes`.

- `regionlist` - seznam regionů. Každému regionu náleží čtyři čísla po řadě, souřadnice x, souřadnice y, atribut a velikost trojúhelníků v daném regionu. Délka seznamu je potom čtyřnásobek `numberofregions`.

Triangle funguje tak, že všechny vstupní pole kromě `triangearealist` kopíruje na výstup. Mezi ně ještě patří následující pole.

- `edgelist` - seznam hran. Každé hraně náleží dva indexy vrcholů v seznamu `pointlist`. Dvojnásobek čísla `numberofedges` udává délku seznamu. Ve výstupu se seznam ukáže pokud je použit přepínač *e*.
- `neighborlist` - seznam sousedních trojúhelníků pro každý trojúhelník. Všem trojúhelníkům jsou přiřazení tří indexy sousedních trojúhelníků. Pokud má trojúhelník méně jak tří sousedi, je na těch místech uvedena -1. Ve výstupu se objeví při použití přepínače *n*.

Teď když už víme, jak Triangle funguje se můžeme podívat, jak se dá z Julia pracovat s C knihovnami.

```
void triangulate(
  char *, // seznam přepínačů
  struct triangulateio *, // vstupní data
  struct triangulateio *, // výstupní data
  struct triangulateio * // Voronéhо diagram
);
```

Ukázka kódu 2.1: Funkce triangulate pro komunikaci s Triangle

2.2 Komunikace mezi Triangle a Julia

Na jedné straně máme knihovnu Triangle psanou v jazyce C a na druhé straně balíček v Julia, který má s Triangle pracovat. Sama Julia přímo poskytuje rozhraní pro práci s C programy. Dokumentace má tomu věnovanou celou jednu sekci [7].

Dočteme se tam [24], že pro strukturu 2.2 musíme vytvořit ekvivalentní složený typ, který se skládá ze stejných atributů. Julia má přímo pro C datové typy `int`, `double`, `int *` a `double *` ekvivalenty `Cint`, `Cdouble`, `Ptr{Cint}` a `Ptr{Cdouble}`. Složený typ 2.6 přesně odpovída struktuře2.2.

```
struct triangulateio {
double *pointlist; // seznam vrcholů triangulace
double *pointattributelist; // seznam atributů pro vrcholy
int *pointmarkerlist; // seznam hraničních značek pro vrcholy
int numberofpoints; // počet vrcholů
int numberofpointattributes; // počet atributů pro každý vrchol

int *trianglelist; // seznam trojúhelníků
double *triangleattributelist; // seznam atributů pro trojúhelníky
double *trianglearealist; // seznam maximálních velikostí pro každý
↪ trojúhelník
int *neighborlist; // seznam indexů sousedních trojúhelníků pro každý
↪ trojúhelník
int numberoftriangles; // počet trojúhelníků
int numberofcorners; // počet vrcholů trojúhelníku
int numberoftriangleattributes; // počet atributů pro každý
↪ trojúhelník

int *segmentlist; // seznam segmentů
int *segmentmarkerlist; // seznam hraničních značek pro každý segment
int numberofsegments; // počet segmentů

double *holelist; // seznam děr
int numberofholes; // počet děr

double *regionlist; // seznam regionů
int numberofregions; // počet regionů

int *edgelist; // seznam hran
int *edgemarkerlist; // seznam hraničních značek pro každou hranu
int numberofedges; // počet hran

double *normlist; // seznam normálních vektorů (pro Voroného diagram)
};
```

Ukázka kódu 2.2: Struktura triangulateio pro vstup a výstup

Potom, co náš program načte všechna data, je potřeba je nějakým způsobem převést na objekt typu 2.6. Víme, že všechna data se do struktury 2.2 předávají jako jednorozměrná pole. Je tedy logické zvolit stejnou reprezentaci i v Julia. Jako jednorozměrné pole slouží v Julia datový typ `Vector`.

Při převodu načtených dat na objekt typu2.6 je potřeba získat z pole typu `Vector` ukazatel typu `Ptr`. Při výstupu z funkce 2.1 je zase potřeba opačný postup - získat objekt typu `Vector` z ukazatele typu `Ptr`. Pro převod z pole na ukazatel slouží funkce 2.3. Pro opačný případ máme k dispozici funkci 2.4. Funkce 2.5 nám umožňuje zavolat funkci2.1 z Julia.

Máme vše k tomu, abychom dokázali pracovat s Triangle v Julia. V následující sekci se podíváme na to, jak se implementuje balíček do Julia a jak se do něho integruje knihovna Triangle.

```
pointer(
  array # pole
  [, index] # index prvku, na který chceme ukazatel
)
```

Ukázka kódu 2.3: Funkce pro získání ukazatele na pole typu Vector [4]

```
unsafe_wrap(
  Array, # Značí, že chceme z ukazatele získat pole.
  pointer::Ptr{T}, # ukazatel na typ T
  dims, # délka pole
  own=false # zda má Julia převzít kontrolu nad pamětí pole
)
```

Ukázka kódu 2.4: Funkce pro převod ukazatele na pole typu Vector [5]

```
ccall(
  (symbol, library) or function_pointer, # funkce, kterou voláme
  ReturnType, # návratový typ
  (ArgumentType1, ...), # tuple typů, které mají jednotlivé parametry
   ↪  volané funkce
  ArgumentValue1, ... # parametry volané funkce
)
```

Ukázka kódu 2.5: Funkce pro zavolání funkce z C knihovny [6]

```
type triangulateio
pointlist::Ptr{Cdouble}
pointattributelist::Ptr{Cdouble}
pointmarkerlist::Ptr{Cint}
numberofpoints::Cint
numberofpointattributes::Cint
trianglelist::Ptr{Cint}
triangleattributelist::Ptr{Cdouble}
trianglearealist::Ptr{Cdouble}
neighborlist::Ptr{Cint}
numberoftriangles::Cint
numberofcorners::Cint
numberoftriangleattributes::Cint
segmentlist::Ptr{Cint}
segmentmarkerlist::Ptr{Cint}
numberofsegments::Cint
holelist::Ptr{Cdouble}
numberofholes::Cint
regionlist::Ptr{Cdouble}
numberofregions::Cint
edgelist::Ptr{Cint}
edgemarkerlist::Ptr{Cint}
normlist::Ptr{Cdouble}
numberofedges::Cint
end
```

Ukázka kódu 2.6: Složený typ triangulateio pro vstup a výstup

2.3 Vytvoření balíčku CTriangle

Vývoj balíčku do Julia předpokládá, že máme v systému nainstalovaný verzovací systém git. Pro samotný vývoj balíčků Julia poskytuje balíček `PkgDev.jl` [13], který na `gitu` závisí. Nejdříve se podíváme, jak se zavádí nový balíček do Julia a následně do něho inegrujeme Triangle. Předpokládáme, že vyvíjíme balíček v Julia verze 5.0 na operačním systému Linux.

2.3.1 Zavedení nového balíčku do Julia

Fukce `PkgDev.generate("CTriangle", "MIT")` [26] vygeneruje balíček s licencí `MIT` a názvem `CTriangle.jl` v adresáři `~/.julia/v0.5/CTriangle.jl`. Znak ~ označuje domovský adresář aktuálně přihlášeného uživatele. Vygenerovaný balíček má následující strukturu 2.1. Máme vytvořený balíček a zároveň se nám

- `CTriangle.jl` .... kořenový adresář balíčku
  - `src` .... adresář se zdrojovými soubory
    - `CTriangle.jl` .... hlavní modul balíčku
  - `test` .... testy
    - `runtests.jl` .... soubor spouštěný při testech
  - `.git` .... skrytý adresář gitu
  - `README.md` .... popis balíčku
  - `LICENSE.md` .... informace o licenci
  - `travis.yml` .... konfigurační soubor pro testovací server Travis CI
  - `REQUIRE` . soubor s verzí Julia a seznamem dalších balíčků, na kterých je náš závislý

Adresářová struktura 2.1: Struktura balíčku CTriangle

i inicializoval git repositář ve složce `.git`. V dokumentaci [11] se píše, že pokud máme vytvořený účet na GitHubu a git o něm ví, tak se nám automaticky vytvoří vzdálený repositář na GitHubu. Že se nám repositář správně nakonfiguroval poznáme tak, že soubor `.git/config` má obsah podobný ukázce 2.7. Ne každý používá pro hostování repositářů GitHub, proto se nemusí nakonfigurovat repositář správně. Stačí udělat následující dva kroky.

1. Vytvořit prázdný vzdálený repositář na nějakém hostovacím serveru (GitHub, GitLab, BitBucket…).

2. Vložit jeho url adresu do sekce `[remote "origin"]` souboru `.git/config`.

Potom stačí už jen nahrát nově vytvořený balíček na vzdálený repositář příkazem `git push`.

```
[core]
  bare = false
  repositoryformatversion = 0
  filemode = true
  logallrefupdates = true
[remote "origin"]
  url = https://github.com/kuzmamar/CTriangle.jl.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
  remote = origin
  merge = refs/heads/master
```

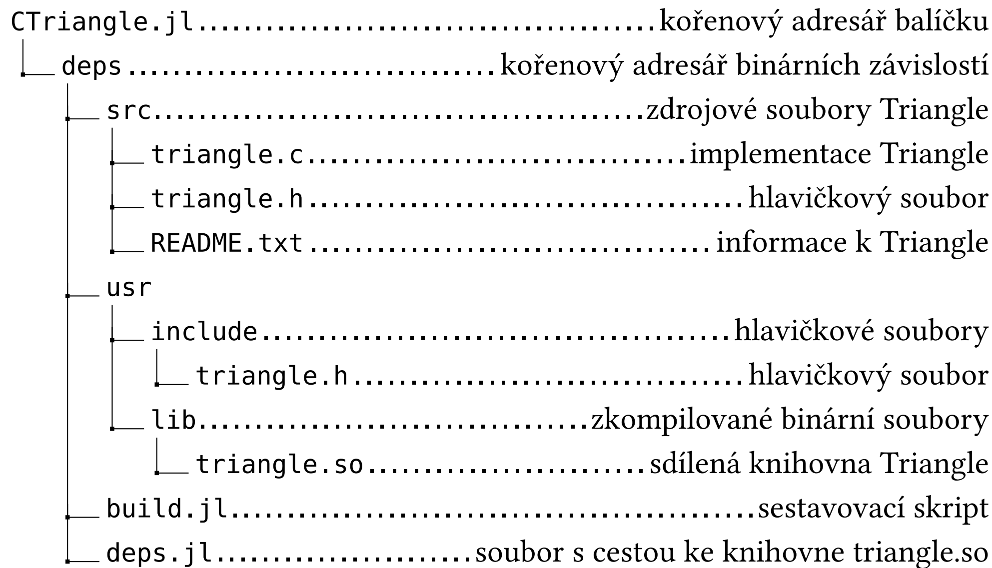
Ukázka kódu 2.7: Konfigurace git repositáře

2.3.2 Integrace Triangle do CTriangle

Pro instalaci binárních závislostí poskytuje Julia balíček `BinDeps.jl` [27]. Předpokládá se, že náš balíček `CTriangle` obsahuje v kořenovém adresáři adresář `deps` s obsahem v ukázce 2.2. Klíčový soubor pro instalaci Triangle do našeho balíčku je soubor `build.jl`. Ten zařídí kompilaci Triangle a vytvoření binárního souboru `triangle.so`. Nutno poznamenat, že je potřeba Triangle zkompilovat jako sdílenou knihovnu. Na ukázce můžeme vidět soubor `build.jl`, který kompilaci provede.

Řádek 7 deklaruje seznam binárních závislostí, které chceme vytvořit. Protože máme je knihovnu Triangle, máme jen jednu závislost `lib`. Ta říká, že se výsledný soubor jmenuje `triangle.so`. Parameter `runtime` má hodnotu `true`. To znamená, že je binární soubor `triangle.so` vyžadovaný při běhu programu. Parametr `os` má hodnotu `:Unix`. To říká, že tato závislost vznikne jen na unixových operačních systémech.

Dále jsou důležité řádky 14 až 24. Zde probíhá samotná kompilace závislosti `lib`. Řádek 15 říka, že pokud neexistuje soubor `triangle.so`, tak se vytvoří. Poslední



Adresářová struktura 2.2: Umístění binárních závislostí

řádek 26 vytvoří soubor `deps.jl`, který do proměnné `_lib _jl_libtriangle` vloží cestu k vytvořenému souboru `triangle.so`. Jeden takový soubor `deps.jl` můžeme vidět na ukázce 2.9. Tento soubor pak vložíme v hlavním modulu našeho balíčku `CTriangle.jl`2.10.

Máme už vše, co pro implementaci balíčku `CTriangle` potřebujeme. Nakonec si shrneme funkční a nefunkční požadavky.

2.4 Nefunkční požadavky

Skript 2.8 kompiluje knihovnu Triangle jen pro unixové systémy. Balíček tak nebude fungovat například na operačním systému Windows. Po konzultaci s vedoucím práce jsme se rozhodli, že za účelem této práce to zatím dostačuje.

A protože každý software by měl mít dokumentaci, tak ani náš balíček nebude výjimkou. Julia má k tomuto účelu k dispozici balíček `Documenter.jl` [8]. Nefunkční požadavky máme tak tři.

1. Balíček bude fungovat prozatím jen na unixových systémech.

2. Implementace bude provedena v jazyce Julia.

```
using BinDeps

#Remove deps.jl before build.
isfile("deps.jl") && rm("deps.jl")

@BinDeps.setup
deps = [lib = library_dependency("lib", aliases = ["triangle.so"],
 ↪  runtime = true, os = :Unix)]
rootdir = BinDeps.depsdir(lib)
srcdir = joinpath(rootdir, "src")
prefix = joinpath(rootdir, "usr")
libdir = joinpath(prefix, "lib")
headerdir = joinpath(prefix, "include")
libfile = joinpath(libdir, "triangle.so")
provides(BinDeps.BuildProcess, (@build_steps begin
  FileRule(libfile, @build_steps begin
    BinDeps.ChangeDirectory(srcdir)
    `gcc -O -DLINUX -DANSI_DECLARATORS -fpic -DTRILIBRARY -o
     ↪  triangle.o -c triangle.c`
    `gcc -shared -o triangle.so triangle.o`
    `cp triangle.so $libdir`
    `cp triangle.h $headerdir`
    `rm triangle.so`
    `rm triangle.o`
  end)
end), lib)

@BinDeps.install Dict([(:lib, :_jl_libtriangle)])
```

Ukázka kódu 2.8: Soubor build.jl - instalace Triangle do CTriangle

3. Balíček zdokumentujeme za pomoci balíčku `Documenter.jl`

2.5 Funkční požadavky

Víme, že Triangle umí generovat *DT*, *CDT*, *ZDT*, *ZCDT*. Vedoucí práce chce také, aby bylo možné generovat *DT* přímo ze seznamu vrcholů vytvořeného v Julia.

Protože se jedná o triangulace, bylo by hezké, kdyby jsme umožnili generovat jejich graf. Domluvili jsme se s vedoucím, že stačí, když balíček umožní

```
1  # This is an auto-generated file; do not edit
2
3  # Pre-hooks
4
5  # Macro to load a library
6  macro checked_lib(libname, path)
7  ((VERSION >= v"0.4.0-dev+3844" ? Base.Libdl.dlopen_e :
   ↪   Base.dlopen_e)(path) == C_NULL) && error("Unable to load
   ↪   \n\n$libname ($path)\n\nPlease re-run Pkg.build(package), and
   ↪   restart Julia.")
8  quote const $(esc(libname)) = $path end
9  end
10
11 # Load dependencies
12 @checked_lib _jl_libtriangle
   ↪   "/home/martin/.julia/v0.5/CTriangle.jl/deps/usr/lib/triangle.so"
13
14 # Load-hooks
```

Ukázka kódu 2.9: Soubor deps.jl - cesta k souboru triangle.so

```
1  module CTriangle
2
3  depsjl = joinpath(dirname(@__FILE__), "..", "deps", "deps.jl")
4
5  if isfile(depsjl)
6  include(depsjl)
7  else
8  error("CTriangle is not properly installed. Please try to
   ↪   run\nPkg.build(\"CTriangle\")")
9  end
```

Ukázka kódu 2.10: Vložení souboru deps.jl do CTriangle.jl

generovat graf jako `.tex` soubor za použití balíčku `PGFPlots` [15]. Máme tak celkem devět funkčních požadavků.

1. generování DT ze seznamu vrcholů
2. generování DT ze souboru
3. generování CDT ze souboru
4. generování ZDT ze souboru
5. generování ZCDT ze souboru
6. generování grafu DT
7. generování grafu CDT
8. generování grafu ZDT
9. generování grafu ZCDT

2.5.1 Generování grafu

Na výsledném grafu zobrazíme vrcholy, hrany a segmenty. Je dobré odlišit segmenty barvou. Potom dobře poznáme, co je segment a co hrana. Segmenty zobrazíme jako poslední. Jinak se může stát, že je hrany překryjí a segmenty tak neuvidíme.

Hrany jsou přítomny, ale pouze v případě, že uživatel zvolí přepínač *e*. V opačném případě zobrazíme trojúhelníky. Je pravda, že jsou trojúhelníky přítomny vždy. Můžeme zobrazovat tedy jen je. Zobrazení hran je pouze optimalizace, protože u trojúhelníků některé jeho strany zobrazíme vícekrát.

Pro vložení dat do grafu je nejprve uložíme do souboru `.dat`. Na ten potom odkážeme v souboru `.tex`. Oddělíme tak prezentaci od samotných dat a umožníme tím jejich znovu použití.

Kapitola 3

Návrh

V této kapitole navrhneme rozhraní v Julia podle funkčních požadavků 2.5. V podstatě máme dva úkoly - vygenerovat výsledek výpočtu *DT*, *CDT*, *ZDT* a *ZCDT* a zobrazit pro něj graf. Nejprve navrhneme proces generování výpočtu a poté generování grafu.

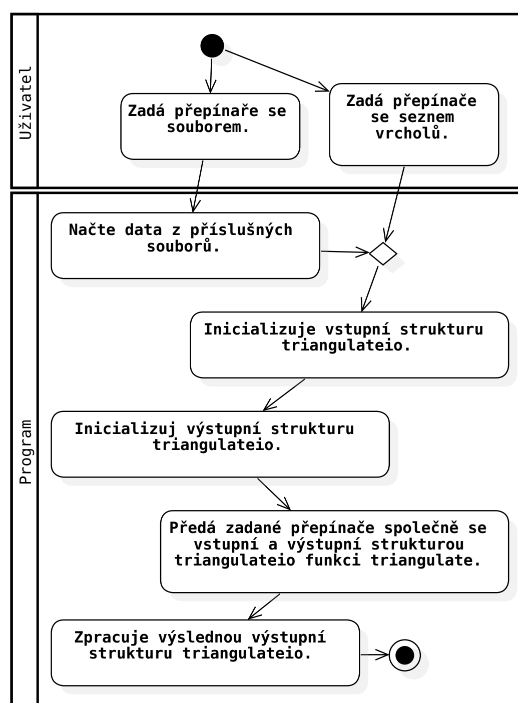
3.1 Generování triangulací

Diagram 3.1 znázorňuje postup provádění výpočtu, pro který navrhneme řešení. Začneme tím, že vytvoříme datové struktury, které reprezentují vstup a výstup. Dále navrhneme rozhraní, přes které je uživatel schopen pracovat s Triangle. Nakonec se podíváme na to, jak rozpoznáme ze zadaných přepínačů, který výpočet se má provádět.

3.2 Vstup a výstup

Víme, že každý soubor začíná hlavičkou, která mimo jiné udává, kolik řádků se má číst. Soubor `.poly` má takovýchto hlaviček hned několik. To nás vede k návrhu reprezentace souboru jako objektu, který je rozdělen do několika částí. Diagram 3.2 znázorňuje takovou reprezentaci. Diagramy 3.3 a 3.4 ukazují jednotlivé části souborů. Následuje popis souborů a jejich sekcí.

- `NodeFile` - soubor `.node`



Obrázek 3.1: Generování triangulace

- `PolyFile` - soubor `.poly`
- `AreaFile` - soubor `.area`
- `EleFile` - soubor `.ele`
- `NodeFileSectionInterface` - sekce s vrcholy
- `SegmentFileSectionInterface` - sekce se segmenty
- `HoleFileSectionInterface` - sekce s dírami
- `RegionFileSectionInterface` - sekce s regiony
- `AreaFileSectionInterface` - sekce s velikostmi trojúhelníků
- `ElementFileSectionInterface` - sekce s trojúhelníky

Všechny soubory implementují funkci `init` a `read`. Funkce `init` má za úkol inicializovat atributy struktury 2.6. Fukce `read` zase načte data do jednotlivých částí. Soubory `PolyFile` a `NodeFile` ještě navíc implementují funkci `getStartIndex`,

která určuje jestli se indexuje od jedničky či od nuly. `NoNodeFileSection` a `NodeFileSection` implementují funkci `isEmpty` díky ní poznáme, jestli je soubor `.node` prázdný nebo ne.

Pro každou část vytvoříme dva typy. Jeden vyznačuje prázdnou (neexistující) část a druhý, který obsahuje data odpovídající části. Například pro část týkající se segmentů máme prázdný typ `NoSegmentFileSection` a typ s vlastními daty pro segmenty `SegmentFileSection`. Každá sekce (část) má atribut `cnt`, který označuje, kolik řádek je v sekci obsaženo. `NodeFileSection`, `EdgeFileSection`, `ElementFileSection` a `SegmentFileSection` obsahují atribut `startIndex`, který určuje jestli se indexuje od nuly nebo od jedničky. Ostatní atributy odpovídají těm v struktuře 2.6.

Atributy `points`, `attrs`, `attrCnt` a `markers` sekce `NodeFileSection` se mapují na atributy `pointlist`, `pointattributelist`, `numberofpointattributes` a `pointmarkerlist` v tomto pořadí.

Atributy `segments` a `markers` sekce `SegmentFileSection` se mapují na atributy `segmentlist` a `segmentmarkerlist` v tomto pořadí.

Atributy `elems`, `cornerCnt`, `attrs`, `attrCnt` sekce `ElementFileSection` se mapují na atributy `trianglelist`, `numberofcorners`, `triangleattributelist` a `numberoftriangleattributes` v tomto pořadí.

Atribut `holes` sekce `HoleFileSection` se mapuje na atribut `holelist`.

Atribut `regions` v sekci `RegionFileSection` se mapuje na atribut `regionlist`.

Atribut `areas` sekce `AreaFileSection` se mapuje na atribut `arealist`.

Jako vstup každého výpočtu je řetězec přepínačů a k němu odpovídající soubory. Nabízí se tedy vytvořit typ, který drží soubory a přepínače pohromadě. Pro každý výpočet vytvoříme jeden takový typ. Na diagramu 3.5 můžeme vidět řešení.

- `DelaunayFileInput` - vstup výpočtu *DT*
- `ConstrainedDelaunayFileInput` - vstup výpočtu *CDT*
- `DelaunayRefinementFileInput` - vstup výpočtu *ZDT*

- `ConstrainedDelaunayRefinementFileInput` - vstup výpočtu *ZCDT*

Kromě těchto vstupů máme ještě vstup `DelaunayUserInput`, který představuje případ, kdy uživatel zadá seznam vrcholů pro výpočet *DT* přímo s Julia.

Funkce `init` inicializuje celou strukturu 2.6. Právě ve funkci `triangulate` se provede samotná komunikace mezi Triangle a naším balíčkem. Inicializuje se tam vstupní a výstupní struktura 2.6. Poté se předají data Triangle a z výsledku se vytvoří reprezentace triangulace. A protože výsledkem výpočtu *ZDT* i *CDT* je *DT* respektive *CDT*, vrací funkce `triangulate` triangulacei podle diagramu 3.6.

Na diagramu vidíme, že každá triangulace je rozdělena do sekcí podobně jako soubory 3.2. Je logické zvolit toto řešení, protože se v podstatě jedná o strukturu 2.6. V případě souborů se na ní díváme jako na vstup. Kdežto u triangulací je pro nás výstupem. Reprezentaci sekcí v triangulacích ukazují diagramy 3.7 a 3.8. Následuje popis triangulací a jejich sekcí.

- `DelaunayTriangulation` - reprezentace *DT*
- `ConstrainedDelaunayTriangulation` - reprezentace *CDT*
- `NodeTriangulationSection` - sekce s vrcholy
- `SegmentTriangulationSectionInterface` - sekce se segmenty
- `HoleTriangulationSectionInterface` - sekce s dírami
- `RegionTriangulationSectionInterface` - sekce s regiony
- `AreaTriangulationSectionInterface` - sekce s velikostmi trojúhelníků
- `ElementTriangulationSectionInterface` - sekce s trojúhelníky
- `EdgeTriangulationSectionInterface` - sekce s hranami triangulace

Pro každý objekt triangulace(vrchol, segment, díra, region, trojúhelník, hrana) vytvoříme typ, který drží všechna data daného objektu.

- `Point` - Předsatvuje bod ve 2D. Použijeme i pro reprezentaci díry.
- `Node` - vrchol s bodem, atributy a hraniční značkou

- `Segment` - segment s oběma krajními body a hraniční značkou
- `Region` - region s bodem, atributem a velikostí trojúhelníků
- `Element` - trojúhelník se seznamem svých bodů a indexů sousedních trojúhelníků
- `Edge` - hrana triangulace s oběma krajními body a hraniční značkou

Diagram 3.9 znázorňuje dané typy.

Obě triangulace implementují tyto metody.

- `getNode` - Vrátí vrchol na indexu `index` se všemi jeho atributy a hraniční značkou.
- `getNodes` - Vrátí iterátor na vrcholy.
- `getSegment` - Vrátí segment na indexu `index` s oběma krajními vrcholy a hraniční značkou.
- `getSegments` - Vrátí iterátor na segmenty
- `getElement` - Vrátí trojúhelník na indexu `index` s jeho atributy a seznamem indexů jeho sousedních trojúhelníků.
- `getElements` - Vrátí iterátor na trojúhelníky.
- `getEdge`- Vrátí hranu triangulace na indexu `index` s její hraniční značkou.
- `getEdges` - Vrátí iterátor na hrany.
- `getNeighbors` - Vrátí seznam sousedních trojúhelníků k trojúhelníku `element`.

`ConstrainedDelaunayTriangulation` navíc ještě implementuje následující metody.

- `getHoles` - Vrátí iterátor na seznam děr.
- `getRegions` - Vrátí iterátor na seznam regionů.

Důvodem toho, že vracíme iterátory místo celého seznamu je čistě paměťový. Dejme tomu, že například funkce `getNodes` vrací celý seznam objektů typu

`Node`. To znamená, že vytvoří tolik objektů `Node`, kolik máme vrcholů. V případě, že je seznam velice dlouhý, alokujeme zbytečně další paměť při každém zavolání funkce `getNodes`. Iterátor funguje tak, že prochází původní seznam vrcholů a objekt typu `Node` vytváří, až když si vrchol vyžádáme. Tím tak šetříme paměť.

Každý z iterátorů iteruje přes odpovídající sekce. Například pro vytvoření objektu typu `Edge`, potřebujeme znát dva krajní vrcholy a hraniční značku.

Všechny iterátory musí implementovat funkce, které Julia vyžaduje.

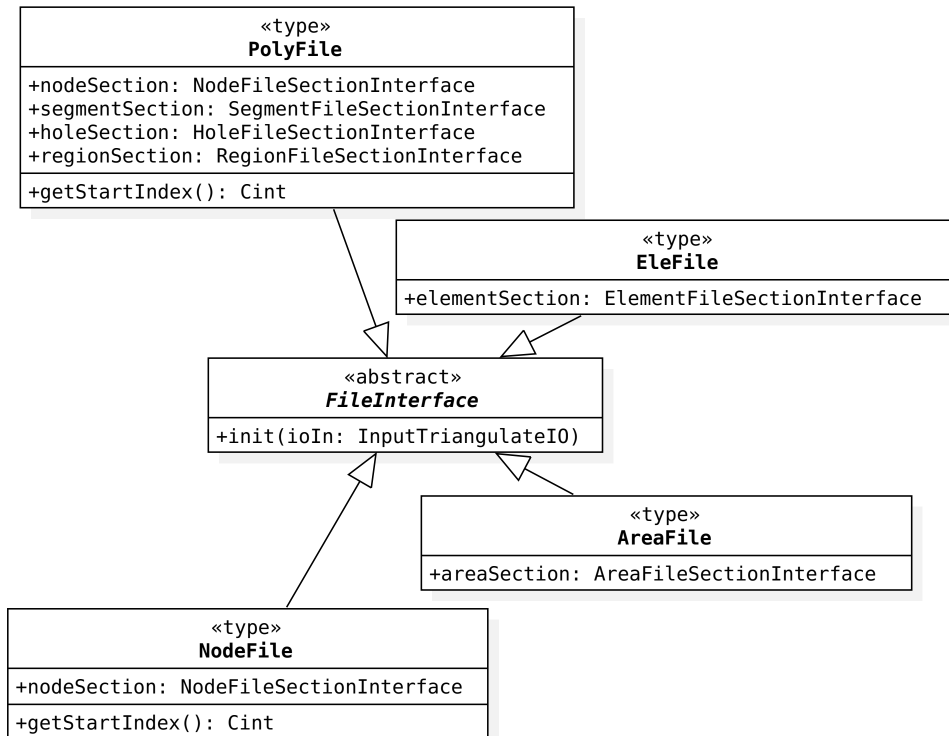
- `start` - Vrátí počáteční index iterace.
- `next` - Vrátí dvojici, kde první prvek je objekt na indexu `state` a druhý prvek je následující index.
- `eltype` - Vrátí typ objektu.
- `length` - Vrátí délku seznamu, přes který se iteruje.

Jednotlivé iterátory ukazují obrázky 3.10 a 3.11.

- `NodeIterator` - iterátor přes vrcholy.
- `SegmentIterator` - iterátor přes segmenty
- `HoleIterator` - iterátor přes díry
- `RegionIterator` - iterator přes regiony
- `ElementInterator` - iterator přes trojúhelníky
- `EdgeInterator` - iterator přes hrany

Také si všimněme, že všechny typy reprezentující triangulace 3.6, 3.9, 3.7, 3.8 jsou typu `immutable`. Jedná se o speciální typ, který umožňuje pouze číst data. Žádná modifikace dat není možná.

Stejně tak všechny seznamy nejsou reprezentovány polem typu `Vector`, ale speciálním seznamem typu `Tuple`. Z tohoto seznamu lze také jen pouze číst data.



Obrázek 3.2: Soubory

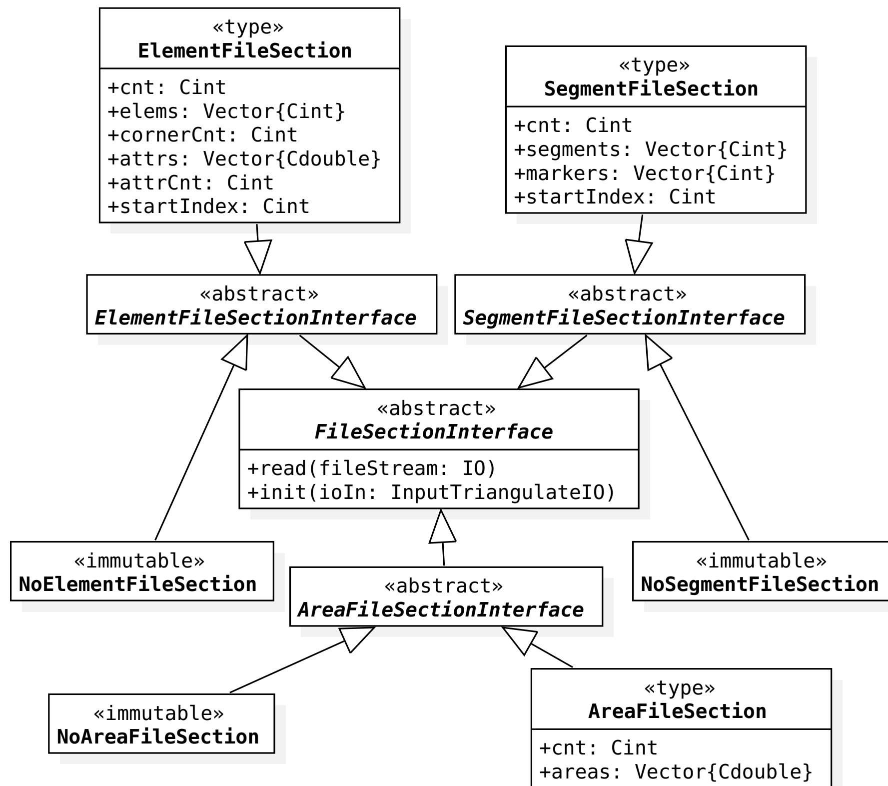
3.2.1 Rozhraní triangulace

Pro uživatelskou část algoritmu 3.1 vytvoříme funkci `triangulate` se třemi implmentacemi. Návrh ukazuje kód 3.1. Všechny implementace vrátí nějakou z triangulací *DT*, *CDT* podle druhu výpočtu.

První implementace `triangulate` odpovídá kroku, kde je uživatel schopen zadat soubor s daty a přepínači. Prametry jsou následující.

- `filename::String` - cesta k souboru s daty. Lze zadat soubor s příponou i bez přípony. Přepínače *r*, *p*, *a* určují, které soubory se čtou. Všechny musí mít však stejné jméno. Cesta k souboru je povinná.
- `options::String` - seznam přepínačů. Uživatel nemusí zadat žádný přepínač. V tom případě se provádí *DT*.

Druhá a třetí implementace umožňuje uživateli zadat přepínače se seznamem bodů. Druhý parametr je stejný jako v první implementaci. První parametr



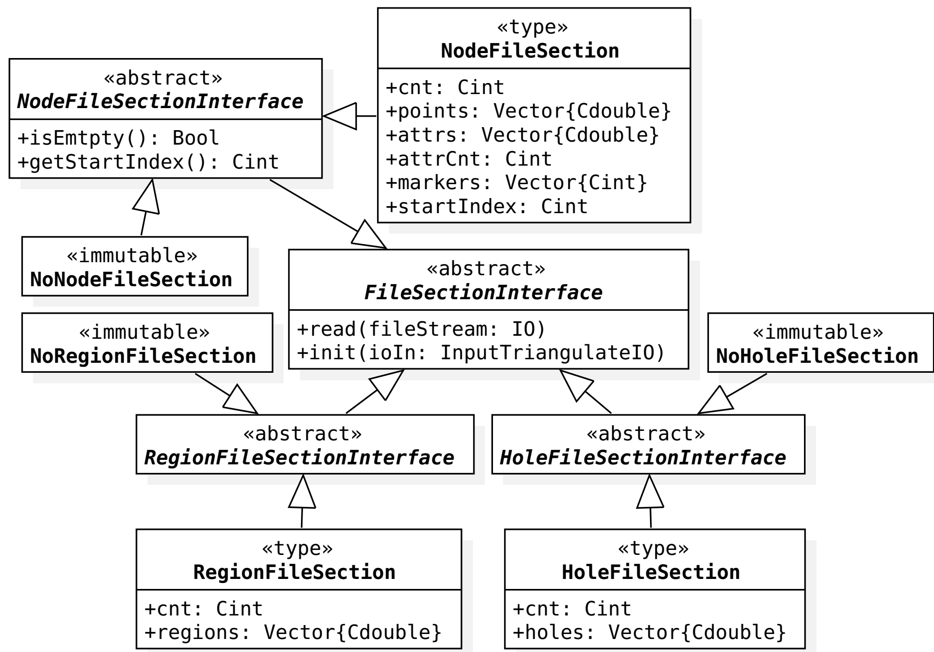
Obrázek 3.3: Rozdělení souborů do částí

je matice s vrcholy, kde první řádek jsou x-ové souřadnice a druhý řádek y-souřadnice vrcholů.

3.2.2 Parsování přepínačů

Druhý krok algoritmu triangulace 3.1 se týká čtení dat ze souborů. Nejprve musíme zjistit, jakou triangulaci máme provádět 2.1, abychom věděli, které soubory číst. Diagram 3.12 ukazuje, jak implementujeme tento proces.

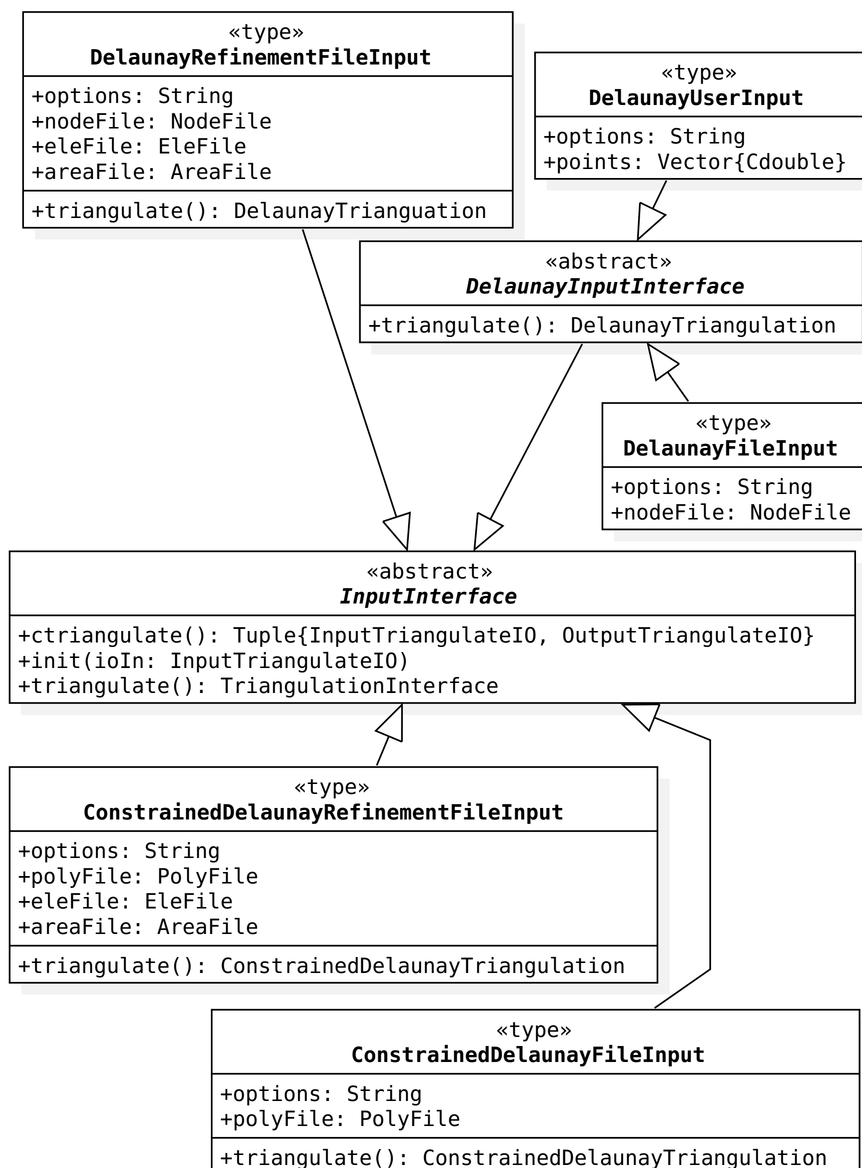
Myšlenka je taková, že simulujeme příkazovou řádku `CommandLine`, která nejdříve rozparsuje přepínače funkcí `parseOptions`. Tím zjistíme, kterou triangulaci provádíme. Podle toho vytvoříme odpovídající příkaz funkcí `createCommand`. Na něj zavoláme funkci `execute`, která provede celý proces triangulace 3.1. Dále následuje popis diagramu 3.12.



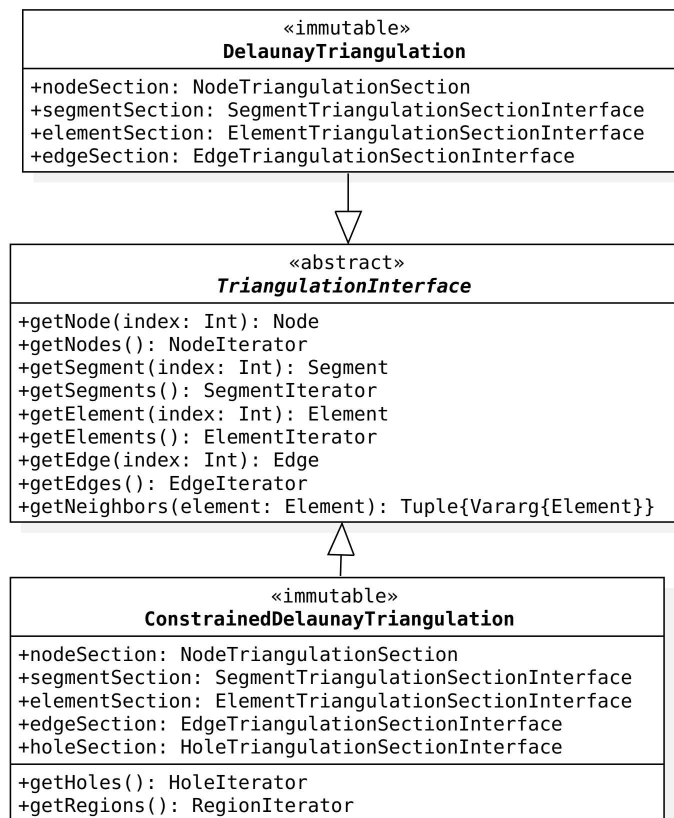
Obrázek 3.4: Rozdělení souborů do částí

```
function triangulate(
  fileName::String, # cesta k souboru
  options::String = "" # seznam přepínačů
)
  # implementace
end
function triangulate(
  points::Matrix{Cdouble}, # matice vrcholů
  options::String = "" # seznam přepínačů
)
  # implementace
end
function triangulate(
  points::Matrix{Int}, # matice vrcholů
  options::String = "" # seznam přepínačů
)
  # implementace
end
```

Ukázka kódu 3.1: Funkce pro triangulace



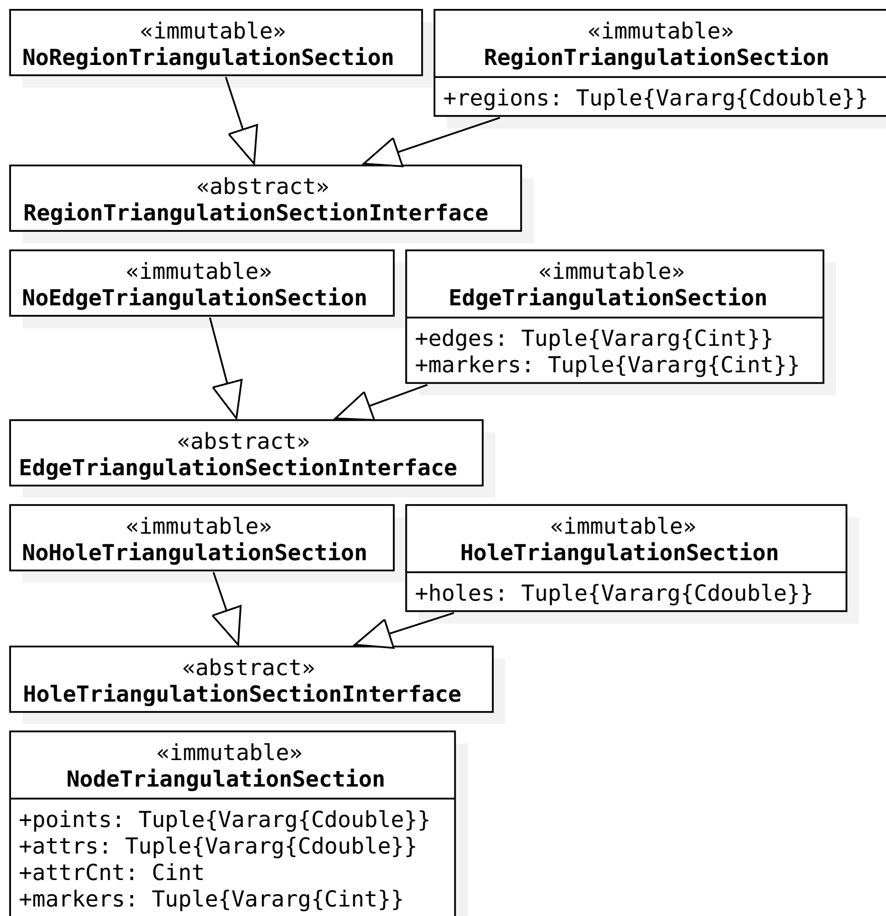
Obrázek 3.5: Vstup



Obrázek 3.6: Triangulace

`CommandLine` je typ, který reprezentuje příkazovou řádku. Jeho atributy jsou následující.

- `delaunay` - Říká, že se provádí buď *DT* nebo *ZDT*. Záleží, jestli je použitý přepínač *r*.
- `constrainedDelaunay` - Reprezentuje přepínač *p*.
- `refinement` - Reprezentuje přepínač *r*.
- `useAreas` - Reprezentuje přepínač *a* bez čísla, kdy se pří výpočtu *ZDT* a *ZCDT* čte soubor `.area`
- `useRegions` - Reprezentuje přepínač *a* bez čísla a přepínač *A*. Při výpočtu *CDT* se čte sekce s regiony v souboru `.poly`.
- `useHoles` - Označuje, zda se mají číst díry ze souboru `.poly`.



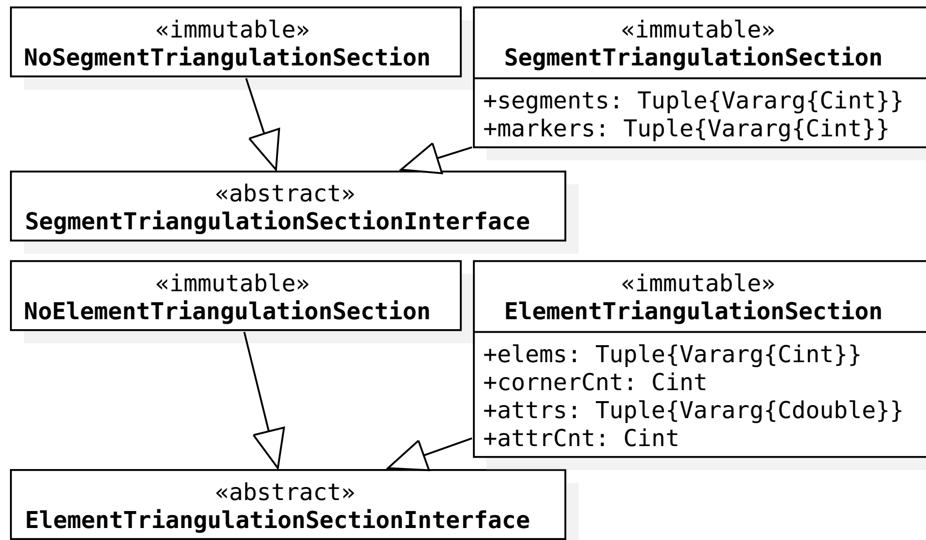
Obrázek 3.7: Sekce triangulací

Funkce pro příkazovou řádku `CommandLine` jsou:

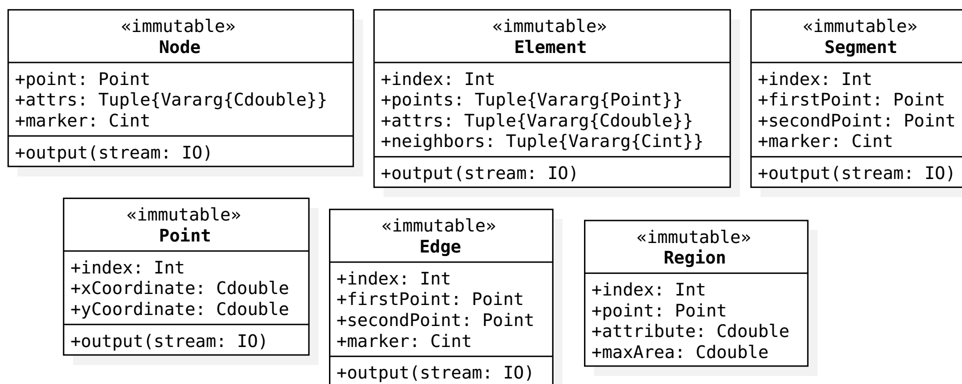
- `parseOptions` - Nastaví atributy příkazové řádky `CommandLine` podle přepínačů `options`. Zároveň odstraní přepínače, které nezná. Vrátí rozparsované přepínače zpět.
- `createCommand` - Vytvoří příkaz, který odpovídá nastavené příkazové řádce po rozparsování přepínačů. Parametr `options` jsou ony přepínače. Parametr `fileName` je cesta k souboru se vstupními daty.

Každý možný výpočet triangulace reprezentuje jeden příkaz. Ty jsou následující.

- `DelaunayFileCommand` - výpočet *DT*



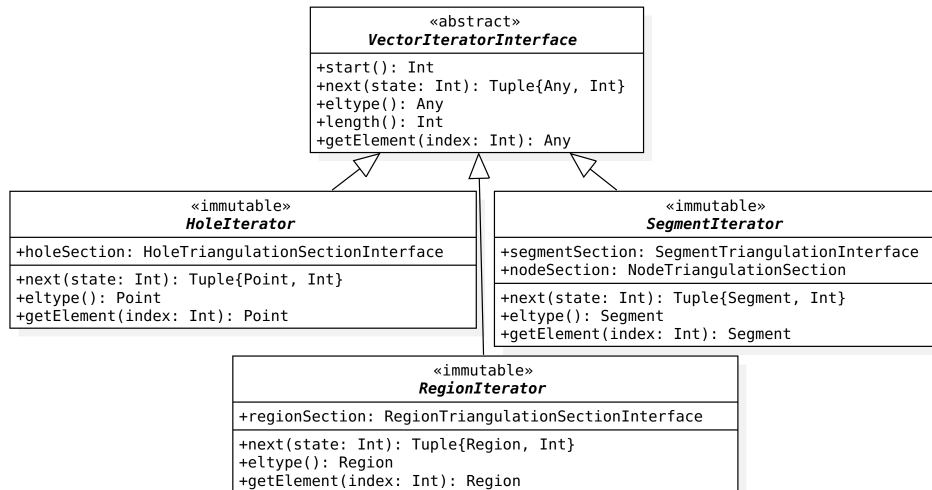
Obrázek 3.8: Sekce triangulací



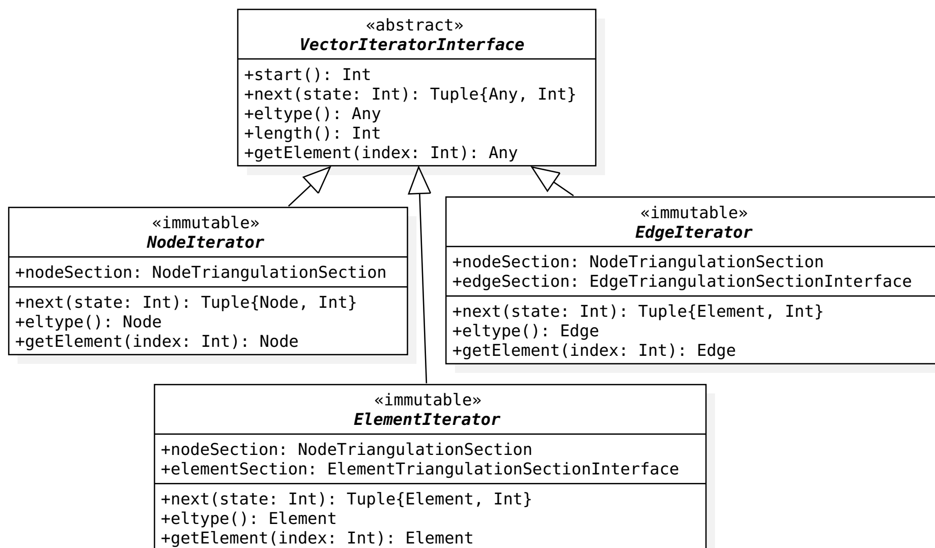
Obrázek 3.9: Typy pro všechny objekty triangulace

- `ConstrainedDelaunayFileCommand` - výpočet *CDT*
- `DelaunayRefinementFileCommand` - výpočet *ZDT*
- `ConstrainedDelaunayRefinementFileCommand` - výpočet *ZCDT*

Každý příkaz má jako atributy přepínače `options` a objekty, které reprezentují cesty k jednotlivým souborům. Objekty ukazuje diagram 3.13. Důvod, proč volíme cesty k souborům jako objekty a ne jednoduše jako řetězce objasníme v kapitole Testování.



Obrázek 3.10: Iterátory

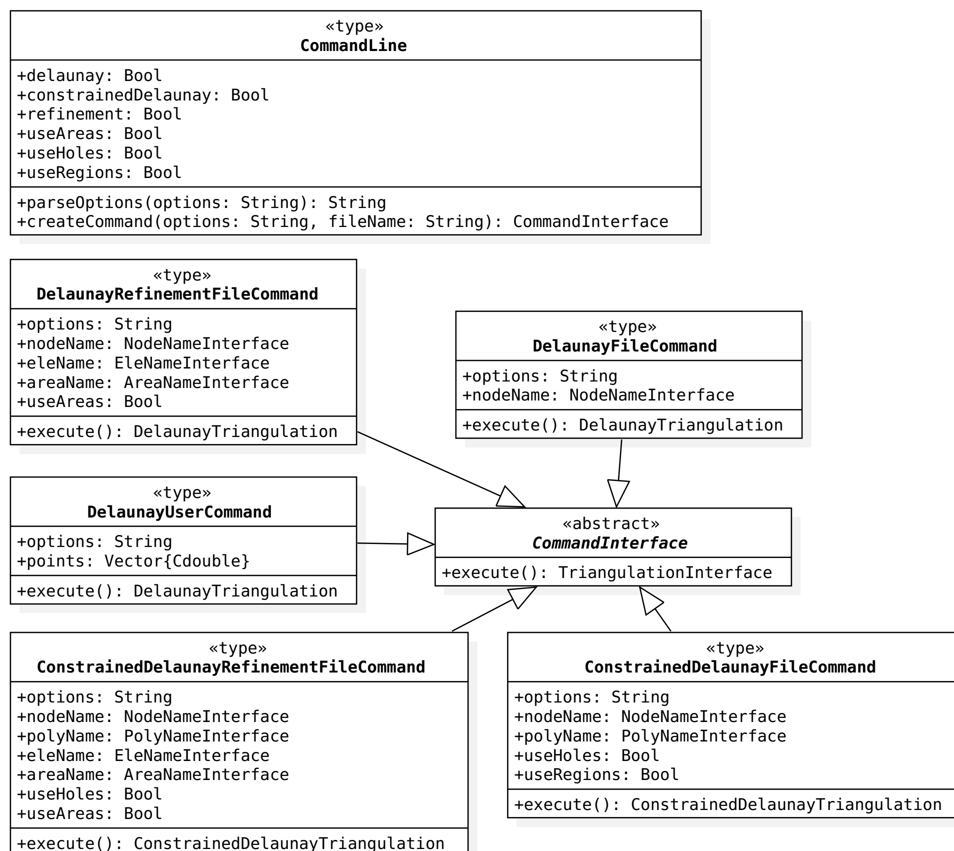


Obrázek 3.11: Iterátory

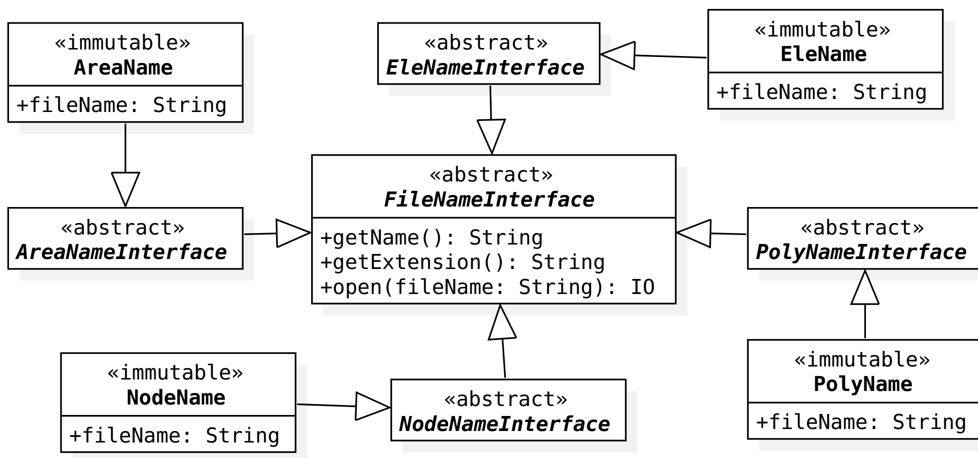
Každý objekt implementuje tyto funkce.

- `getName` - Vrátí cestu k souboru bez koncovky.
- `getExtension` - Vrátí koncovku souboru.
- `open` - Otevře soubor `filename` pro čtení.

Všechny příkazy implementují funkci `execute`, která spustí proces triangulace a na konci vrátí její odpovídající reprezentaci 3.6.



Obrázek 3.12: Simulace příkazové řádky

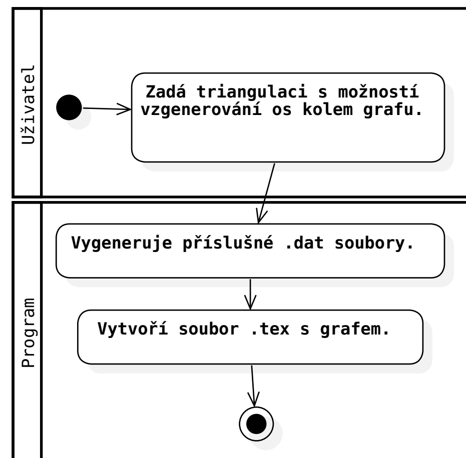


Obrázek 3.13: Objekty reprezentující cesty k souborům

3.3 Generování grafů

Na obrázku 3.14 vidíme, jak pracuje algoritmus pro vygenerování grafu *DT* a *CDT*. Uživateli poskytneme funkci `outputGraph` 3.2, která požadovaný graf vygeneruje.

Představa je taková, že uživatel zadá triangulaci a cestu k adresáři, kam se má vygenerovat. Dále si může zvolit jednotlivá jména `.dat` souborů `.tex` souboru pro graf. `PGFPlots` umožňuje volitelně vygenerovat osy kolem grafu. Necháme uživatele, ať si zvolí, jestli chce zobrazit osy nebo ne. Koneckonců si může vygenerovaný `.tex` soubor upravovat podle libosti.



Obrázek 3.14: Generování grafu triangulace

```
function outputGraph(
  triangulation::TriangulationInterface, # triangulace
  directory::String; # cesta k adresáři
  nodesDataFileName::String = "nodes", # jméno .dat souboru pro
  ↪  vrcholy
  edgesDataFileName::String = "edges", # jméno .dat souboru pro hrany
  elemesDataFileName::String = "elements", # jméno .dat souboru pro
  ↪  trojúhelníky
  segmentsDataFileName::String = "segments", # jméno .dat souboru pro
  ↪  segmenty
  triangulationFileName::String = "triangulation" # jméno .tex
  ↪  souboru pro graf
  displayAxis::Bool = false, # zobrazení os v grafu
)
  # implementace
end
```

Ukázka kódu 3.2: Funkce pro triangulace

Kapitola 4

Realizace

Důležitou součástí psaní balíčku v Julia je jeho organizace kódu. Můžeme velice snadno narazit na problém, pokud se někde v kódu odkazujeme na proměnnou, jejíž typ ještě není v té době definován. V takovém případě Julia vrátí chybu `UndefVarError`. Problému se vyhneme snadno tím, že nejdříve definujeme všechny typy ve správném pořadí a poté až funkce.

Organizaci složky `src` proto zvolíme, tak že pro každou skupinu typů vytvoříme jeden soubor s jejich definicemi a do druhého vložíme funkce, které nad nimi operují. Pro lepší orientaci v kódu vložíme každou skupinu do své vlastní složky. Například typy reprezentující vstupy do Triangle 3.5 organizujeme podle ukázky 4.1.

Podle rozsahu definovaných typů můžeme navíc vytvořit pomocný soubor, do kterého vložíme všechny definice typů a funkcí. Tento soubor pak dáme do hlavního modulu našeho balíčku.

Takto organizovaný kód se rozdělí logicky na menší celky. Každý soubor je tak i rozumně dlouhý a nemusíme se pročítat mnoha řádky kódu. Dále si ukážeme pro představu, jak implementujeme klíčové součásti balíčku.

4.1 Parsování přepínačů

Celý proces začíná zjišťováním, který výpočet se má provádět. Řešení vidíme v kódu 4.2. Na druhém řádku se nejdříve rozparsují přepínače funkcí `parseOptions`.

```
# soubor src/Includes.jl
include("Inputs/InputTypes.jl")
include("Inputs/InputMethods.jl")
# soubor src/CTriangle.jl
module CTriangle
include("Includes.jl")
end
```

Ukázka kódu 4.1: Definice typů a funkcí v souboru Includes.jl

Potom se vytvoří odpovídající příkaz funkcí `createCommand` a na něm se spustí funkce `execute`, která vrátí výslednou triangulaci.

Samotné parsování přepínačů probíhá, tak že se prochází postupně každý přepínač a řádek 20 zjišťuje, zda se jedná o jeden s přepínačů *p*, *r*, *a* bez čísla, *A* nebo *O*. Podle toho nastaví příkazovou řádku `CommandLine`. Nepodporované přepínače se ignorují na řádku 25. Na konci se z rozparsovaných přepínačů vytvoří řetězec a ten se vrátí.

```
src ........................................ adresář se zdrojovými soubory
└── Inputs ................................. složka pro celou skupinu typů
│   ├── InputTypes.jl ...................... definice typů
│   └── InputMethods.jl .................... definice funkcí
└── CTriangle.jl ........................... hlavní modul balíčku
    └── Includes.jl pomocný soubor pro vložení definic všech typů a fukcí
```

Adresářová struktura 4.1: Organizace kódu

4.2 Komunikace s Triangle

Asi nejdůležitějšími kroky je pro nás umět inicializovat strukturu 2.6 vstupními daty a zpracování výstupních dat. Řešení ukazuje kód 4.3. Na příkladu vidíme, jak se inicializuje struktura načtenými vrcholy pomocí funkce `setPoints`. V těle funke je použita právě zmiňovaná funkce `pointer` 2.3.

```julia
# soubor src/CTriangle.jl
function triangulate(fileName::String, options::String = "")
  commandLine::CommandLine = CommandLine()
  execute(
    createCommand(commandLine, parseOptions(commandLine, options),
     ↪  fileName)
  )
end
# soubor src/CommandLine/CommandLineMethods.jl
function parseOptions(commandLine::CommandLine, options::String)
  parsedOptions = Vector{String}(length(options))
  commandLineOptions = createCommandLineOptions()
  extraOptions = createExtraOptions()
  index = 1
  for option in options
    strOption = string(option)
    if isdigit(strOption) === true || strOption == "."
      parsedOptions[index] = strOption
    elseif in(strOption, extraOptions) === true
      parsedOptions[index] = strOption
    elseif haskey(commandLineOptions, strOption) === true
      parsedOptions[index] = commandLineOptions[strOption](
        options, index, commandLine
      )
    else
      parsedOptions[index] = ""
    end
    index = index + 1
  end
  join(parsedOptions, "")
end
```

Ukázka kódu 4.2: Parsování přepínačů

Pro získání seznamu vrcholů slouží funkce `getPoints`. Řádek 10 získá z ukazatele `pointlist` seznam vrcholů za pomocí funkce `usafe_wrap` 2.4. Posledním parametrem `own = true`, předáváme zodpovědnost za alokovanou pamět knihovnou Triangle do Julia. Nemusíme se tak sami starat o její uvolnění.

Zde si musíme dát pozor, protože pokud se zavolá funkce `getPoints` vícekrát, vznikne několik seznamů, které budou chtít uvolnit na konci programu stejnou pamět. To povede k systémové chybě. Z tohoto důvodu po získání seznamu vynulujeme ukazatel na řádku 13. Pokud je ukazatel nulový, vrací se prázdný seznam. V celém balíčku sice funkci `getPoints` voláme pouze jednou, ale jako pojistka to ničemu neuškodí.

Funkce `ctriangulate` provede triangulaci. Nejprve se inicializuje vstup na řádku 23. Potom se na řádku 24 zavolá fukce `triangulate` 2.1 v knihovně Triangle s načtenými daty a přepínači. Výstup se potom vrací.

4.3 Čtení vstupních souborů

Druhou nejdůležitější schopností našeho balíčku je správně přečíst data ze souborů. K tomu potřebujeme umět rozparsovat data na řádce v dané sekci. Jakmile umíme rozparsovat jednu řádku umíme načíst celou sekci. Přečtení souboru znamená přečíst postupně všechny jeho sekce.

Ukážeme si implementaci parsování řádků v sekci se segmenty souboru `.poly`. Stejný postup použijeme i pro ostatní sekce ve zbývajících souborech. U sekce se segmenty víme, že řádka obsahuje index řádku(číslo segmentu), indexy obou krajních bodů a možná hraniční značku.

Parser si představíme jako objekt 4.4, který si jako své atributy ponese funkci, seznam parametrů pro tuto funkci a název pole, do kterého se ukládají data z dané sekce. Funkce provede samotné načítání dat z řádku do daného pole. Řešení vidíme v kódu 4.5.

Funkce `read` iteruje přes celou sekci řádek po řádku a postupně je parsuje funkcí `parseLine`, které předá seznam parserů. Ta následně zavolá funkci `applyParsers`, která spustí funkci každého parseru ze seznamu. Jednou takovou funkcí je třeba `parseSegments`.

```
# soubor src/IO/IOMethods.jl
function setPoints(io::InputTriangulateIO, points::Vector{Cdouble})
  io.pointlist = pointer(points)
  io.numberofpoints = Cint(length(points) / 2)
end
function getPoints(io::OutputTriangulateIO)
  if io.pointlist == C_NULL
    ()
  else
    points::Vector{Cdouble} = unsafe_wrap(
      Array, io.pointlist, io.numberofpoints * 2, true
    )
    io.pointlist = C_NULL
    tuple(points...)
  end
end
# soubor src/Inputs/InputMethods.jl
function ctriangulate(input::InputInterface)
  ioIn = InputTriangulateIO()
  ioOut = OutputTriangulateIO()
  voronoi = VoronoiTriangulateIO()
  options = getOptions(input)
  init(input, ioIn)
  ccall(
    (:triangulate, _jl_libtriangle), Void, (Ptr{UInt8},
    ↪  Ref{InputTriangulateIO},
    Ref{OutputTriangulateIO}, Ref{VoronoiTriangulateIO}), options,
    ↪  Ref(ioIn),
    Ref(ioOut), Ref(voronoi)
  )
  (ioOut, voronoi)
end
```

Ukázka kódu 4.3: Komunikace s Triangle

Každá parsovací funkce jako parametr dostane pole, do kterého načte data, řádku ze které data přečte a index řádku. Ostatní parametry jsou ty z parseru.

Všimněme si, že seznam parametrů z parseru je typu `Tuple{Vararg{Cint}}`. Tím říkáme, že může obsahovat lobovolný počet hodnot. To přesně potřebujeme, protože ne všechny parsery mají stejný počet těchto parmetrů. Funkce `parseSegments` jsou těmito parametry `start` a `startIndex`. První říká, kde na řádku se začíná parsovat a druhý říká jetli krajní body indexujeme od nuly nebo od jedničky. Pro parsování hraničních značek poutřebujeme například pouze paramter `start`.

```
1  # soubor src/Sections/FileSectionTypes.jl
2  immutable Parser
3    func::Function # parsovací funkce
4    args::Tuple{Vararg{Cint}} # seznam parametrů
5    vector::Symbol # jméno pole
6  end
```

Ukázka kódu 4.4: Parser

4.4 Generování grafu

Proces generování grafu implementujeme přesně podle algoritmu 3.14. Představme si tento proces jako posloupnost úkonů, které se provádí po sobě.

Pro vygenerování každého `.dat` souboru vytvoříme jeden úkon, který se o jeho vygenerování postará. Každý úkon zároveň ví, který `.dat` soubor se generuje jako další. Výsledkem každého úkonu je tedy další úkon. Poslední úkon vygeneruje graf do souboru `.tex`. Pro názornost si ukážeme implementaci úkonu pro generování souboru `.dat` pro vrcholy 4.6.

Na ukázce lze vidět, že se na řádku 3 otevře soubor `.dat`. Následně se do něho na řádku 8 zapíší všechny vrcholy triangulace. Na konci se vrátí úkon, který vygeneruje `.dat` soubor pro hrany. Jedním z parametrů úkonu pro generování hran je funkce `outputNodes`, která pří generování souboru `.tex` přidá odkaz na soubor `.dat` s vrcholy.

```
# soubor src/Sections/FileSectionMethods.jl
function Base.read(
  section::FileSectionInterface, fileStream::IO,
  parsers::Vector{Parser}
)
  for index in section
    parseLine(section, readFileLine(fileStream), parsers, index)
  end
end
function parseLine(
  section::FileSectionInterface, # sekce
  line::Vector{SubString{String}}, # načtená řádka
  parsers::Vector{Parser}, # seznam parserů
  index::Cint # číslo řádku
)
  applyParsers(section, line, parsers, index)
end
function applyParsers(
  section::FileSectionInterface, # sekce
  line::Vector{SubString{String}}, # načtená řádka
  parsers::Vector{Parser}, # seznam parserů
  index::Cint # číslo řádku
)
  for parser in parsers
    vector = parser.vector
    args = parser.args
    parser.func(getfield(section, vector), line, index, args...)
    end
end
# soubor src/UtilMethods.jl
function parseSegments(
  segments::Vector{Cint}, # seznam segmentů
  line::Vector{SubString{String}}, # načtená řádka
  index::Cint, # číslo řádky
  start::Cint, # index na řádku
  startIndex::Cint # index určující indexování bodů od 1 nebo 0
)
  second = 2 * index
  segments[second - 1] = getIndex(startIndex, parse(Cint,
   ↪  line[start]))
  segments[second] = getIndex(startIndex, parse(Cint, line[start +
   ↪  1]))
end
```

Ukázka kódu 4.5: parsování řádky

```
# soubor src/OutputTasks/OutputTaskMethods.jl
function output(task::OutputNodesTask)
  fileStream::IO = open(
    task.directory,
    getNodesDataFilePath(task.directory)
  )
  for node::Node in getNodes(task.triangulation)
    output(node, fileStream)
  end
  close(fileStream)
  createOutputEdgesTask(
    task.triangulation, task.directory, outputNodes,
     ↪  task.displayAxis, task.displaySegments
  )
end
# soubor src/DataTypes/DataTypeMethods.jl
function output(point::Point, stream::IO)
  write(stream, "$(point.xCoordinate) $(point.yCoordinate)\n")
end
function output(node::Node, stream::IO)
  output(node.point, stream)
end
# soubor src/OutputUtilMethods.jl
function outputNodes(fileStream::IO, fileName::String)
  write("\\addplot[")
  write(fileStream, "only marks, black, mark size=1pt")
  write("]")
  write(fileStream, " table {\$(getFileName(fileName,
   ↪  DATA_EXT))};\n")
end
```

Ukázka kódu 4.6: Generování .dat souboru pro vrcholy

Kapitola 5

Testování

Abychom aspoň do nějaké míry zaručili, že poskytujeme kvalitní software je potřeba ho mít otestovaný. Musíme zejména zaručit, že software dělá to, co jsme si vymezili ve funkčních požadavcích.

Nabízí se spousta druhů testů, kterými můžeme prověřit náš software. Vzhledem k tomu, že náš balíček není až tak funkčně rozsáhlý, nepracuje s žádnou databází, nemá žádné grafické rozhraní, zvolíme testování jednotlivých součástí tzv. jednotkovými testy (*unit tests*).

Jednotkový test se vyznačuje tím, že stav programu po spuštění testu je stejný jako před spuštěním. Testuje pouze jednotku (funkci, objekt) a nezajímají ho její závislosti. Většina programů ale pracuje s nějakou databází nebo přistupuje na disk pro přečtení/zápis souboru. Řešením, jak se zbavit v jednotkových testech těchto závislostí, je použít metodu tzv. *mocking*. To v podstatě znamená, že každou závislost vyměníme za napodobeninu(*mock*), která simuluje chování dané závislosti.

V našem případě se jedná například o čtení souborů z disku. V návrhu jsme dříve uvedli 3.13, že pro každý soubor vytvoříme typ, který ho reprezentuje. Například typ `NodeName` je podtypem `NodeNameInterface`. To je záměrem. Takto jsem schopni `NodeName` napodobit typem `FakeNodeName`, který se chová jako `NodeName`. Rozdílem je, že nečte data ze souboru ale z hlavní paměti. Test, tak proběhne rychleji.

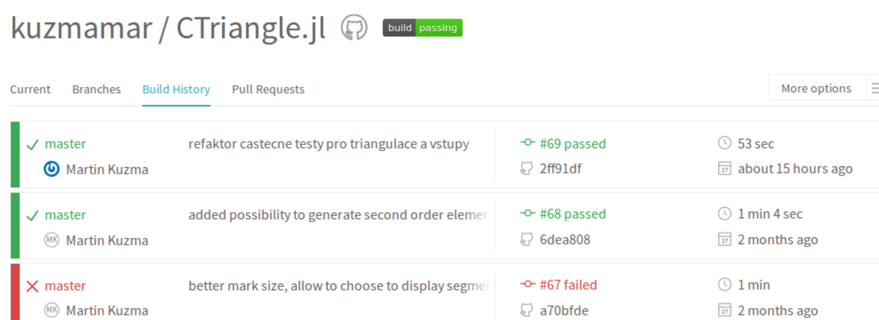
Soubor `.node` napodobíme tak, že vytvoříme objekt `FakeNodeName`, který si drží seznam předem definovaných řádků souboru. Při zavolání funkce `open` vytvoří další napodobeninu, předstírá otevřený soubor. Čtení řádku probíhá ve funkci `readline`. Ta funguje tak, že vrací řádek po řádku ze seznamu předdefinovaných řádků. Na konci čtení se volá funkce `close`. V případě `FakeIO` nedělá nic.

```
1  # soubor src/FileNames/FileNameTypes.jl
2  type FakeNodeName <: NodeNameInterface
3    lines::Vector{String} # řádky v souboru
4  end
5  # soubor src/FileNames/FileNameMethods.jl
6  function Base.open(name::FakeNodeName, fileName::String)
7    createFakeIO(name.lines)
8  end
9  # soubor src/IO/IOTypes.jl
10 type FakeIO <: IO
11   lines::Vector{String} # řádky v souboru
12   lineCnt::Int # počet řádků
13   lineNumber::Int # aktuální číslo řádku
14   FakeIO(lines::Vector{String}) = new(lines, length(lines), 1)
15 end
16 # soubor src/IO/IOMethods.jl
17 function Base.readline(io::FakeIO)
18   if eof(io) == true
19     return ""
20   end
21   line = io.lines[io.lineNumber]
22   io.lineNumber = io.lineNumber + 1
23   return line
24 end
25 Base.close(::FakeIO) = return
26 function Base.eof(io::FakeIO)
27   # Vrátí true pokud jsme na konci souboru.
28   return io.lineNumber > io.lineCnt
29 end
```

Ukázka kódu 5.1: Příklad použití techniky *mocking*

5.1 Testování s TravisCI

Tím že hostujeme implementovaný balíček na GitHubu[5], máme možnost automatizovat spouštění testů s pomocí serveru *TravisCI* [25]. Testy se spustí po každé operaci `push` do větve `master`. Lze nastavit i spouštění testů v nějakém intervalu. Obrázek 5.1 ukazuje přehled spouštěných testů. Po rozkliknutí na detail lze vidět podrobnější výstup z tesů.



Obrázek 5.1: Přehled spouštění testů

[5]Balíček je dostupný z `https://github.com/kuzmamar/CTriangle.jl`

Závěr

Výsledkem této práce je funkční balíček, který umí pracovat s knihovnou Triangle. Pro úspěšnou implementaci je důležité nejdříve provést důkladnou analýzu rozhraní Triangle, protože to určuje, jak rozhraní našeho balíčku navrhneme a implementujeme. Kvalitu doručovaného software do jisté míry zaručíme tím, že otestujeme funkční požadavky. Tím zajistíme, že funguje vše tak, jak se očekává.

Implementovaný balíček umí generovat grafy pro výpočty *DT*, *CDT*, *ZDT* a *ZCDT*. Neumí zatím vygenerovat Voroného diagram, který Triangle umožňuje vygenerovat. Je zde tak prostor rozšíření rozhraní o možnost vygenerovat Voroného diagram. Implementace je navržená tak, že rozšíření by nemělo znamenat značný zásah do kódu ve smyslu předělání stávajícího. Balíček je testovaný pomocí jednotkových testů. V době psaní práce je balíček volně dostupný na serveru GitHub na adrese `https://github.com/kuzmamar/CTriangle.jl`.

Zdroje

1. BAYER, Tomáš. *(Přednáška) Konvexní obálka množiny bodů* [online] [cit. 2016-05-09]. Dostupné z: `https://web.natur.cuni.cz/~bayertom/Adk/adk4.pdf`.
2. BAYER, Tomáš. *(Přednáška) Vlastnosti, použití, konstrukce. Zobecněné Voronoi diagramy* [online] [cit. 2016-05-10]. Dostupné z: `https://web.natur.cuni.cz/~bayertom/Adk/adk6.pdf`.
3. BERG, Mark de; CHEONG, Otfried; KREVELD, Marc van; OVERMARS, Mark. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN 3540779736, 9783540779735.
4. *C Interface* [online] [cit. 2016-12-31]. Dostupné z: `http://docs.julialang.org/en/release-0.5/stdlib/c/#Base.pointer`.
5. *C Interface* [online] [cit. 2016-12-31]. Dostupné z: `http://docs.julialang.org/en/release-0.5/stdlib/c/#Base.unsafe_wrap`.
6. *C Interface* [online] [cit. 2017-01-01]. Dostupné z: `http://docs.julialang.org/en/release-0.5/stdlib/c/#Base.ccall`.
7. *Calling C and Fortran Code* [online] [cit. 2016-12-31]. Dostupné z: `http://docs.julialang.org/en/release-0.5/manual/calling-c-and-fortran-code`.
8. *Documenter.jl* [online] [cit. 2017-01-01]. Dostupné z: `https://juliadocs.github.io/Documenter.jl/stable`.

9. DOLBILIN, N. P. Boris Nikolaevich Delone (Delaunay): Life and work. *Proceedings of the Steklov Institute of Mathematics*. 2011, roč. 275, č. 1, s. 1–14. ISSN 1531-8605. Dostupné z DOI: 10.1134/S0081543811080013.
10. DRTINA, Tomáš. *Generování trojúhelníkové sítě v AutoCADu* [online]. 2008 [cit. 2016-05-01]. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/drtint1_2008dipl.pdf. Diplomová práce. České vysoké učení technické v Praze Fakulta elektrotechnická.
11. *Generating the package* [online] [cit. 2017-01-01]. Dostupné z: http://docs.julialang.org/en/release-0.5/manual/packages/#generating-the-package.
12. CHEN, Jianer. *Computational Geometry: Methods and Applications* [online]. 1996 [cit. 2016-05-01]. Dostupné z: http://faculty.cs.tamu.edu/chen/notes/geo.pdf.
13. *Julia Package Development Kit* [online] [cit. 2017-01-01]. Dostupné z: https://github.com/JuliaLang/PkgDev.jl.
14. MAREŠ, Martin. *Algoritmy a datové struktury II: Geometrické algoritmy* [online]. 2012 [cit. 2016-05-09]. Dostupné z: http://mj.ucw.cz/vyuka/1112/ads2/6-geom.pdf.
15. *PGFPlots - A LaTeX Package to create normal/logarithmic plots in two and three dimensions.* [online] [cit. 2017-01-01]. Dostupné z: http://pgfplots.sourceforge.net.
16. *(Přednáška) Delaunay Triangulations* [online] [cit. 2016-05-09]. Dostupné z: http://www.ti.inf.ethz.ch/ew/Lehre/CG13/lecture/Chapter%5C%206.pdf.
17. SHEWCHUK, Jonathan. *Boundary markers* [online] [cit. 2016-12-28]. Dostupné z: https://www.cs.cmu.edu/~quake/triangle.markers.html.
18. SHEWCHUK, Jonathan. *Command line switches* [online] [cit. 2016-12-27]. Dostupné z: https://www.cs.cmu.edu/~quake/triangle.switch.html.
19. SHEWCHUK, Jonathan. *Jonathan Shewchuk* [online] [cit. 2016-12-27]. Dostupné z: https://people.eecs.berkeley.edu/~jrs/.
20. SHEWCHUK, Jonathan. *Triangle* [online] [cit. 2016-12-29]. Dostupné z: https://www.cs.cmu.edu/~quake/triangle.html.

21. SHEWCHUK, Jonathan. *Triangle* [online] [cit. 2016-12-29]. Dostupné z: https://www.cs.cmu.edu/~quake/triangle.html.
22. SHEWCHUK, Jonathan Richard. *Lecture Notes on Delaunay Mesh Generation* [online]. 1999 [cit. 2016-05-10]. Dostupné z: https://www.semanticscholar.org/paper/Lecture-Notes-on-Delaunay-Mesh-Generation-Shewchuk-Sloan/154a80e8181a2f1acc013202647a4ec1a16c63e9/pdf.
23. SHEWCHUK, Jonathan Richard. Delaunay refinement algorithms for triangular mesh generation. *Comput. Geom.* 2002, roč. 22, s. 21–74.
24. *Struct Type correspondences* [online] [cit. 2016-12-31]. Dostupné z: http://docs.julialang.org/en/release-0.5/manual/calling-c-and-fortran-code/#struct-type-correspondences.
25. *Test and Deploy with Confidence* [online] [cit. 2017-01-09]. Dostupné z: https://travis-ci.org.
26. *Tool for building binary dependencies for Julia modules* [online] [cit. 2017-01-01]. Dostupné z: https://github.com/JuliaLang/PkgDev.jl#generatepkg-license.
27. *Tool for building binary dependencies for Julia modules* [online] [cit. 2017-01-01]. Dostupné z: https://github.com/JuliaLang/BinDeps.jl.

Dodatek A

Seznam použitých zkratek

CDT Constained Delaunayho triangulace
DT Delaunayho triangulace
I/O vstup a výstup
PSLG planární úsečkový graf (*planar straight line graph*)
ZCDT Zjemnění Constrained Delaunayho triangulace
ZDT Zjemnění Delaunayho triangulace

Dodatek B

Obsah přiloženého média

```
├── README ............................................ popis obsahu média
├── BP_Kuzma_Martin_2017.pdf ...................... text práce ve formátu PDF
└── docs ........................................... adresář s dokumentací
    ├── assets ............................... adresář ze soubory a obrázky
    └── index.html ........................... úvodní stránka dokumentace
```

Adresářová struktura B.1: Obsah přiloženého média