Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Konstrukce pseudotriangulace inkrementálním vkládáním

Plzeň, 2010 Ladislav Hobza

Poděkování

Tímto bych chtěl poděkovat paní Doc. Dr. Ing. Ivaně Kolingerové za hodiny konzultací a veškeré poskytnuté materiály k pseudo-triangulacím.

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 3.5.2010 Ladislav Hobza

Abstract:

Pseudo-triangulation construction by incremental insertion

This work was an upgrade of already existing thesis – “Pseudo-triangulations and their use in applied computational geometry”. It was focused on generalized swap (flip) in pseudo-triangulations. Two ways of that swap with the same function were proposed and implemented. The swap was executed on the pseudo-triangulation either as post-processing, pseudo-triangulation had been created in the first step and then was swap performed, or the swap was a part of the incremental algorithm for the pseudo-triangulation construction. The resulting pseudo-triangulations were also tested whether their shape has been improved. Furthermore, several criterions for swap had been tested which of them gave the most optimal pseudo-triangulations. All proposed algorithms have been implemented and tested.

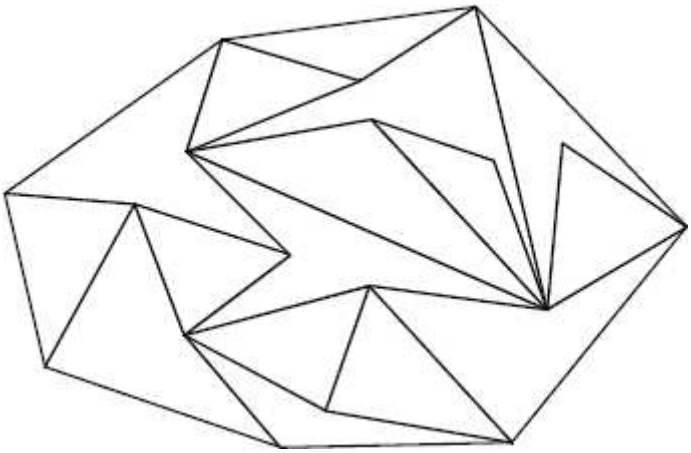
Obsah

1 Úvod

1.1 Úvod do problematiky

Úlohy počítačové grafiky nebo geometrie často pracují s množinou bodů. Velkou většinu z nich pak spojuje fakt, že při jejich řešení je daný prostor rozdělen pomocí bodů a přidaných hran na menší části. Pokud jsou body umístěné v rovině, hovoří se o tzv. rovinných děleních, jako jsou triangulace, mapy viditelnosti nebo třeba Voronoiovy diagramy. Jinými slovy, vstupní data (body) jsou převedena do struktury podobné mozaice. [Trčka07]

V posledních letech pak byla věnována pozornost dělení s názvem pseudo-triangulace[1] (viz obrázek 1.1). Jde o rozdělení na oblasti zvané pseudo-trojúhelníky – mnohoúhelníky s třemi vnitřními úhly menšími než 180°. Přestože první zmínka o pseudo-triangulacích je již více jak 10 let stará, podrobnějšího zkoumání došla struktura až v několika posledních letech, a tak jsou znalosti o kombinatorických a geometrických vlastnostech stále v počátcích a tedy je i omezený počet aplikací, které pseudo-triangulaci využívají. [Trčka07]



Obrázek 1.1. Pseudo-triangulace množiny bodů [Trčka07].

1.2 Hlavní cíle

Naším úkolem v rámci této práce je v první části seznámení se s problematikou pseudo-triangulací a existujících algoritmů pro ně. Dále pak doplnění diplomové práce [Trčka07] o zobecněné prohazování hran[2] v pseudo-triangulaci.

Vzhledem k faktu, že jde o rozšíření již existující práce [Trčka07], byl obsah následující kapitoly (2 Pseudo-triangulace) převzat z této práce. Jedná se především o teoretické informace k pseudo-triangulacím a o stručný popis algoritmů, které jsme přímo využívali. Naše řešení začíná prakticky až kapitolou 3 Swap v pseudo-triangulaci.

[1] Pseudo-triangulace – anglicky pseudo-triangulation, možno psát i bez pomlčky.
[2] Prohazování hran – anglicky “swap“ nebo také “flip“.

Tato práce obsahuje v prvé řadě návrh algoritmu prohazování hran tzv. *brute-force* metodou, což je algoritmicky nejjednodušší, avšak časová složitost je relativně velká - $O(k^3)$, kde k je počet vrcholů v sousedních pseudo-trojúhelnících. Následně se spolu s vedoucí práce budeme snažit nalézt urychlení tohoto řešení a zabudovat prohazování hran do již existujícího algoritmu vytváření inkrementální pseudo-triangulace. Prohazování hran by tedy mělo být možné uskutečnit buď jako tzv. *post-processing*, kdy se nejprve vytvoří pseudo-triangulace programem [Trčka07] a swap bude proveden na celé struktuře, nebo jako součást inkrementálního algoritmu navrženého ve výchozí práci [Trčka07].

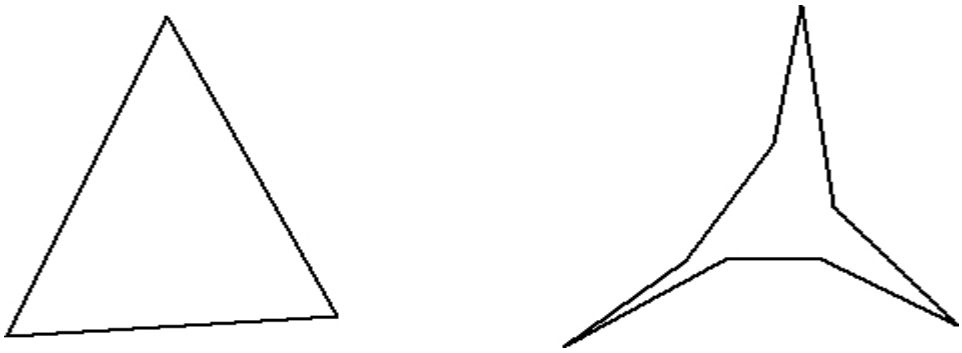
Nakonec budeme testovat nově vzniklé pseudo-trojúhelníky, zda došlo ke zlepšení jejich tvaru, a zároveň porovnávat navržené algoritmy z hlediska časové náročnosti.

2 Pseudo-triangulace

2.1 Základní pojmy a definice

Triangulace T konečné množiny *S n* bodů v rovině je maximální rovinný graf, jehož vrcholy jsou právě body z množiny *S*. Není těžké nahlédnout, že jednotlivé stěny triangulace jsou trojúhelníky s vrcholy z množiny *S*. Pokud nebude uvedeno jinak, budeme v textu předpokládat, že body z množiny *S* jsou v obecné poloze, tedy že žádné tři body neleží v přímce.

Pseudo-trojúhelník PT je rovinný polygon, který má právě tři konvexní vrcholy (dále nazývané *rohy*[3]) s vnitřním úhlem menším než 180°. Obrázek 2.1 ukazuje, že jednotlivé *rohy* spojují až tři konkávní řetězce hran. Vrcholy, v nichž má pseudo-trojúhelník vnitřní úhel větší než úhel přímý, se nazývají *tupoúhlé vrcholy*. Trojúhelník je jednoduchým příkladem pseudo-trojúhelníku.



Obrázek 2.1. Ukázka pseudo-trojúhelníků.

Pseudo-triangulace konečné množiny *S n* bodů v rovině je pak rovinné dělení konvexního obalu množiny *S* na pseudo-trojúhelníky, jejichž množina vrcholů je právě množina *S*. Pseudo-triangulace množiny bodů zobrazuje obrázek 1.1. Triangulace je tak speciálním případem pseudo-triangulace.

Pojmy *roh* a *tupoúhlý vrchol* se přenášejí i do pseudo-triangulace. Tedy se může stát, že jeden vrchol je současně pro jeden pseudo-trojúhelník rohem, zatímco pro jiný pseudo-trojúhelník je tupoúhlým vrcholem.

Strana pseudo-trojúhelníku je konkávní řetězec hran spojující rohy pseudo-trojúhelníků. Každý pseudo-trojúhelník má právě tři strany. Strana je tvořena minimálně jednou hranou.

Pojem *hrana* označuje spojnici dvou bodů. Pokud by se všechny tři strany pseudo-trojúhelníku skládaly každá pouze z jedné hrany, jednalo by se o trojúhelník.

Mnohoúhelníky v triangulaci či pseudo-triangulaci jsou také nazývány *stěny*.

Počet vrcholů mnohoúhelníku budeme značit *k*.

[3] V přiloženém programu jsou používány anglické názvy corner, edge, vertex, face (stěna) apod.

2.2 Minimální pseudo-triangulace

Teorie

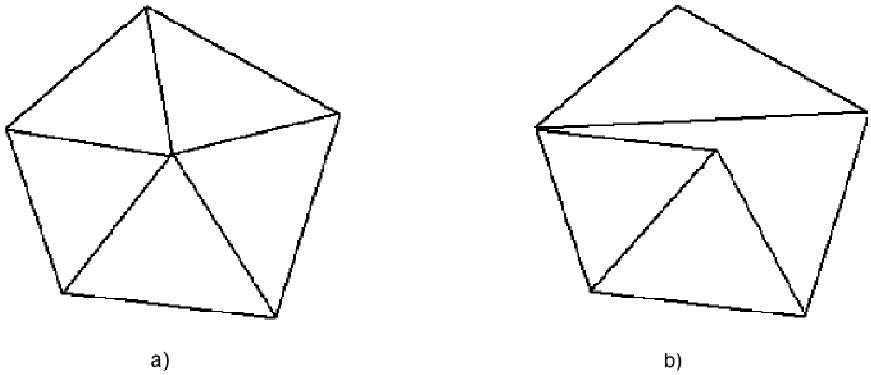
Pseudo-triangulace je zde brána jako graf, tedy jako množina hran a množina vrcholů.

Nechť T je označení pro triangulaci množiny S a nechť PT^T je označení pro pseudo-triangulaci, která je podgrafem T, což znamená, že množina hran PT^T je podmnožinou množiny hran T. Někdy se říká, že PT^T je omezena T.

Pseudo-triangulace PT^T se nazývá *minimální*, pokud v ní neexistuje hrana, po jejímž odebrání struktura zůstane pseudo-triangulací.

Pseudo-triangulace PT^T se nazývá *absolutně minimální*, pokud má nejmenší počet hran ze všech možných pseudo-triangulací omezených na T. Jinými slovy, každá jiná pseudo-triangulace má větší nebo rovný počet hran než *absolutně minimální* pseudo-triangulace. *Absolutně minimální* pseudo-triangulace má několik různých definic a charakterizací. Například nemusí být spojována s triangulací T a pak se definuje jednoduše jako pseudo-triangulace na množině S, která má nejmenší počet hran. Jako příklad jiné charakterizace je možno uvést, že *absolutně minimální pseudo-triangulace n* bodů má právě *n-2* stěn a *2n-3* hran.

Obrázek 2.2 ilustruje rozdíl mezi *minimální* a *absolutně minimální* pseudo-triangulací. Obrázek 2.2 a) ukazuje (pseudo-)triangulaci, ve které odebrání libovolné vnitřní hrany způsobí vznik čtyřúhelníkové stěny, tedy se jedná o *minimální pseudo-triangulaci*. Přesto nejde o nejmenší počet hran, jak ukazuje Obrázek 2.2 b), který zobrazuje pseudo-triangulaci na stejné množině bodů. Jedná se totiž o *absolutně minimální pseudo-triangulaci*, protože má 9 hran a 4 stěny.



Obrázek 2.2 - minimální a absolutně minimální pseudo-triangulace.

2.3 Inkrementální pseudo-triangulace

2.3.1 Teorie

Existuje více způsobů, jak vytvořit na množině bodů pseudo-triangulaci. Jedním z nich je také metoda **Inkrementální vkládání vrcholů**, navržená a implementovaná v programu [Trčka07]. Inkrementální algoritmy založené na postupném vkládání bodů bývají používané v tzv. *on-line aplikacích*, kdy na počátku nejsou známa všechna vstupní data, jež přichází za běhu programu (on-line).

Takový on-line algoritmus umožňuje do již vytvořené struktury vložit vrchol, aniž by se musela přebudovat celá struktura. Stačí pouze pozměnit okolí vkládaného bodu, typicky stěnu, do které je bod vkládán. Algoritmus inkrementální pseudo-triangulace se skládá ze dvou kroků:

1) Vyhledání stěny, ve které se vkládaný bod nachází[4].
2) Vložení bodu a úprava struktury na řádnou pseudo-triangulaci[5].

2.3.2 Přidání bodu dovnitř pseudo-triangulace

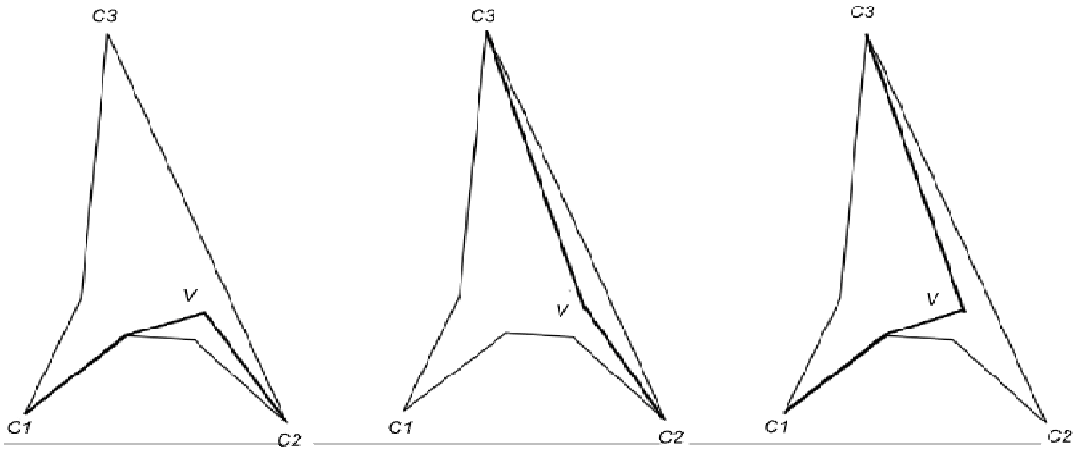
Pokud je vkládaný bod nalezen uvnitř nějakého pseudo-trojúhelníku, je třeba jisté okolí vkládaného bodu upravit tak, aby struktura zůstala po vložení bodu pseudo-triangulací. Ukáže se, že vložením bodu je možné danou stěnu rozdělit na dva menší pseudo-trojúhelníky (upravovaným okolím bodu je tak pouze původní stěna, do které se bod vkládá). Není prozatím uvažován případ, kdy je bod lokalizován ve vrcholu nebo hraně stávající pseudo-triangulace.

Geodetická cesta mezi dvěma body v pseudo-trojúhelníku PT je nejkratší možná cesta mezi zmíněnými body, která leží celá v PT (uvnitř nebo na hranici). Z definice např. plyne, že strana pseudo-trojúhelníku PT je geodetickou cestou mezi příslušnými rohy PT. Protože se jedná o cestu nejkratší, mezi každými dvěma body z PT existuje právě jedna geodetická cesta.

Pro každý bod uvnitř pseudo-trojúhelníku PT, který neleží na hranici PT, tedy existují právě tři geodetické cesty do jednotlivých rohů PT. Situaci ilustruje obrázek 2.3, kde je vkládán bod *V* a geodetické cesty do rohů *PT* jsou zobrazeny tučnou čárou.

Lemma: Nechť máme pseudo-trojúhelník PT a jeden jeho vnitřní bod V, který neleží na hranici PT. Potom dvě geodetické cesty z bodu V do libovolných dvou rohů PT rozdělují PT na dva menší pseudo-trojúhelníky.

Důkaz: Důkaz tohoto tvrzení je možno nalézt na straně 33 diplomové práce [Trčka07]



Obrázek 2.3 - rozdělení pseudo-trojúhelníku geodetickými cestami.

[4] Tento krok je řešen pomocí náhodné procházky implementované v programu [Trčka07]
[5] Úprava nově vzniklé pseudo-triangulace na „řádnou“ pseudotriangulaci byla též mým cílem, pouze v jiném slova smyslu. Mým úkolem je v rámci tohoto 2. kroku prohazovat hrany nových pseudotriangulací, za účelem získání optimálních pseudo-trojúhelníků, viz. kapitola 4.2.

2.3.3 Přidání bodu na hraně pseudo-triangulace

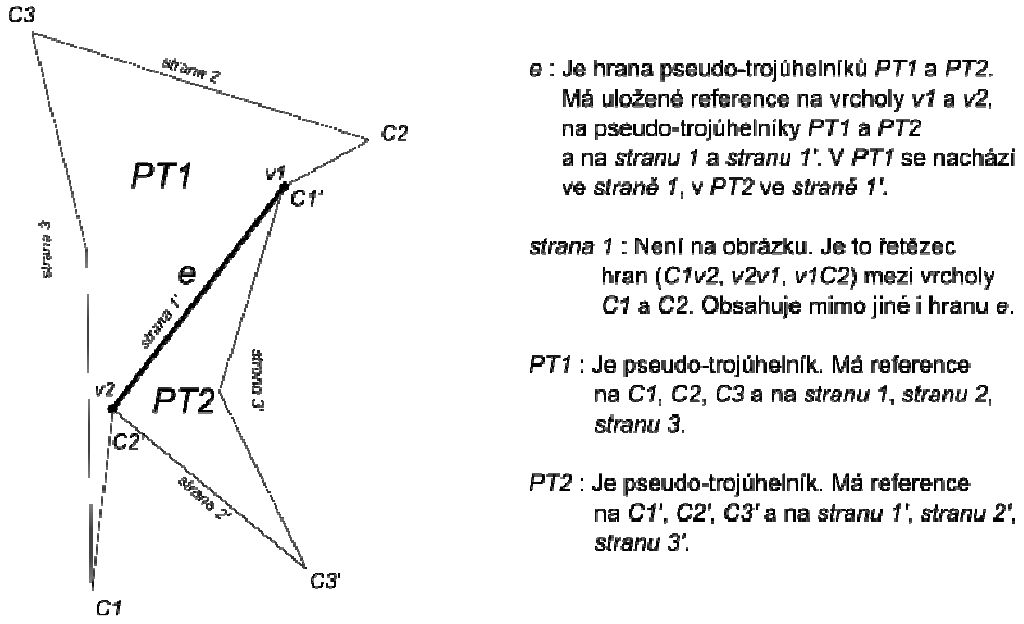
Vložení bodu *V*, který leží na již existující hraně V_1V_2, je implementováno nahrazením hrany V_1V_2 dvěma novými hranami V_1V a VV_2. Následně se rozdělí až dva pseudo-trojúhelníky PT_1 a PT_2 (které původní hranu V_1V_2 obsahovaly) pomocí geodetických cest z *V* do protilehlých rohů PT_1 a PT_2. Pokud je hrana V_1V_2 obsažena jen v jednom pseudo—trojúhelníku, pak se jedná o hranu na konvexním obalu vrcholů pseudo-triangulace. Proto je třeba také příslušně upravit řetězec hran konvexního obalu.

2.4 Implementace

2.4.1 Datová struktura

Veškeré algoritmy využívají stejnou datovou strukturu. Důraz byl při návrhu kladen spíše na praktické ověření teoretických úvah a algoritmů než na nejrychlejší a nejúspornější implementaci.

Datová struktura pseudo-triangulace obsahuje následující: **seznam (pole) vrcholů**, **seznam (lineární obousměrný) hran** a **seznam (lineární obousměrný) stěn**. Každý **vrchol obsahuje své souřadnice v rovině**. Každá hrana je implementována jako obdoba struktury zvané **okřídlená hrana**, jinými slovy **každá hrana obsahuje ukazatele na sousední stěny**, má uloženou informaci o tom, **ve kterých stranách pseudo-trojúhelníků se nachází**, a má **ukazatele na své koncové vrcholy**. Každá **stěna zná své rohy** (orientované proti směru hodinových ručiček) a každá **strana je implementována jako postupný seznam (lineární obousměrný) hran v pořadí proti směru hodinových ručiček**. Struktura byla navrhnuta s optimalizací právě pro algoritmus minimální pseudo-triangulace, a tak základní operace, které algoritmus na struktuře provádí, probíhají v čase O(1). Mezi tyto operace se řadí získání souřadnic vrcholu (přístup do pole), zjištění hran sousedících s danou hranou na okraji jedné stěny (následník a předchůdce dané hrany v lineárním obousměrném seznamu), odebrání hrany nebo stěny (odebrání prvku z obousměrného lineárního seznamu), přidání hrany nebo stěny (přidání prvku na konec obousměrného seznamu) a zjištění dvou sousedních pseudo-trojúhelníků pro každou hranu. Na obrázku 2.4 je ilustrace datové struktury.



Obrázek 2.4. Grafická ukázka datové struktury.

2.4.2 Test vzájemné polohy tří bodů - orientace bodu vůči přímce

Základní dva numerické testy, se kterými program [Trčka07] pracuje, jsou:

1) Test vzájemné polohy tří bodů, resp. test na úhel daný třemi vrcholy nebo dvěma hranami, orientovaný proti směru hodinových ručiček.
2) Test na orientaci bodu vůči přímce.

Počítání úhlů nebývá ani rychlé ani přesné. V našem případě stačí zjistit, zdali je svíraný úhel větší, menší nebo roven úhlu přímému. To se dá zjistit bez použití goniometrických funkcí, a tedy i mnohem přesněji pomocí tzv. orientovaného obsahu trojúhelníku. Jsou-li *A=[a.x, a.y]*, *B=[b.x, b.y]*, *C=[c.x, c.y]* vrcholy trojúhelníku v rovině, pak velikost orientovaného obsahu P trojúhelníku ABC je dána vztahem:

P = (b.x-a.x)(c.y-a.y)-(b.y-a.y)*(c.x-a.x)*

a zároveň platí: pokud P>0, potom vrchol C leží v levé polorovině dané orientovanou přímkou AB (zkráceně C leží nalevo od AB). Pokud P<0, potom C leží v pravé polorovině (napravo od AB). Pokud P=0, potom C leží na přímce AB.

To je ekvivalentní dalšímu požadovanému tvrzení: pokud P>0, potom je orientovaný úhel (proti směru hodinových ručiček) svíraný po řadě stranami AB a AC menší než 180°. Pokud P<0, potom je orientovaný úhel svíraný stranami AB a AC vetší než 180°. Pokud P=0, potom body A, B, C leží v přímce (úhel 0° nebo 180°).

A pro úplnost, předchozí dvě tvrzení jsou ekvivalentní tvrzení následujícímu. Pokud P>0, potom vrcholy A, B, C jsou v tomto pořadí orientovány proti směru hodinových ručiček. Pokud P<0, vrcholy A, B, C jsou v tomto pořadí orientovány po směru hodinových ručiček. Pokud P=0, potom body A, B, C leží v přímce.

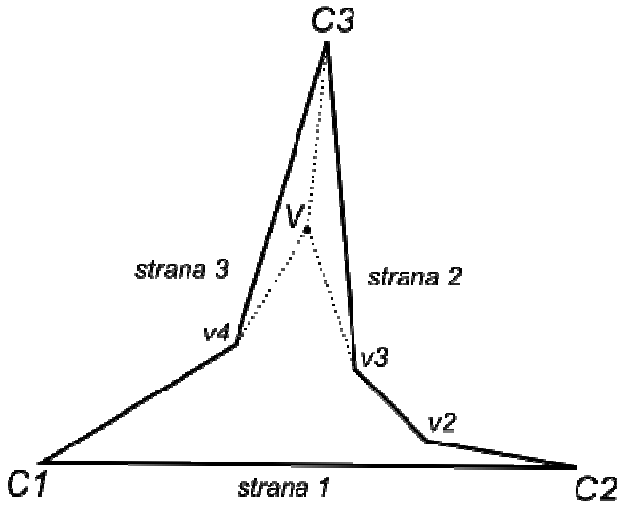
2.4.3 Hledání geodetické cesty mezi vrcholem a vkládaným bodem

Při hledání geodetické cesty budeme používat pojem **viditelnost vrcholu** nebo budeme říkat, že hrana nebo strana zakrývá výhled na vrchol. Tato tvrzení vychází z testu vzájemné polohy tří bodů viz minulá kapitola. Řekneme, že bod je viditelný v orientaci proti směru hodinových ručiček, pokud leží od dané hrany nalevo. V případě orientace po směru hodinových ručiček je bod viditelný, pokud leží od dané hrany napravo. Například na obrázku 2.5 bod *V* a *strana 3*. Pokud se díváme na bod *V* z vrcholu *C1*, *strana 3* zakrývá výhled na *V*. Je to dáno tím, že bod *V* leží od hrany *C1,v4* nalevo. V případě, že se díváme na bod *V* z rohu *C3*, je *V* viditelný, neboť leží od hrany *C3,v4* nalevo a od hrany *C3,v3* napravo. Pokud by se stalo, že bod leží na přímce – výsledek znaménkového testu by vyšel 0, bod je ještě považován za neviditelný (a nastaví se příznak pro přímý úhel).

Pro lepší pochopení je vyhledávání geodetické cesty znázorněno na obrázku 2.5. Geodetická cesta je vyhledávána z rohu C_i, i = 1,2,3, do bodu *V*. Nejdříve se zjistí, která strana zakrývá přímý pohled z rohu C_i do *V*. Může to být buď strana, jejíž koncový vrchol je C_i, nebo strana s počátečním vrcholem C_i. Pro zakrývající stranu pak algoritmus postupně zkontroluje hranu po hraně od vrcholu C_i, zdali už bod *V* není přímo viditelný z vrcholu právě kontrolované hrany. Tento vrchol pak bude druhým koncem nově vytvořené hrany. Metoda pro hledání geodetické cesty *findGeodesic()* naprogramovaná v práci [Trčka07] potom vrací 1. vrchol, náležící některé straně pseudo-trojúhelníku, který bude jedním z vrcholů nové hrany (tím druhým bude *V*), 2. stranu pseudo-trojúhelníku, ve které leží tento vrchol, tj. stranu, která zakrývá pohled z rohu C_i do *V*. Pokud by byl *V* přímo viditelný z rohu C_i, vrátí metoda stranu následující po vrcholu C_i.

Rozeberme příklad na obrázku 2.5 hledání geodetické cesty mezi *C1* a *V*. Nejprve se vezme poslední hrana ze *strany 3* a provede se test, zda orientovaná hrana *C1*,*v4* zakrývá výhled na vrchol *V*. V tomto případě leží bod *V* nalevo od hrany *C1*,*v4*, je tedy zakryt *stranou 3*. Dále se budou procházet hrany *strany 3* směrem od vrcholu *C1* a bude se zjišťovat, zdali je výhled na *V* zakryt aktuální hranou. Pokud ne, vrátí se první bod hrany (orientované ve směru procházení). V tomto případě vrchol *v4*.

Pokud je bod *V* přímo viditelný z rohu *C*, pak metoda vrátí stranu, která následuje po vrcholu *C*. Tento případ nastává pro hledání geodetické cesty z *C3* do *V*.

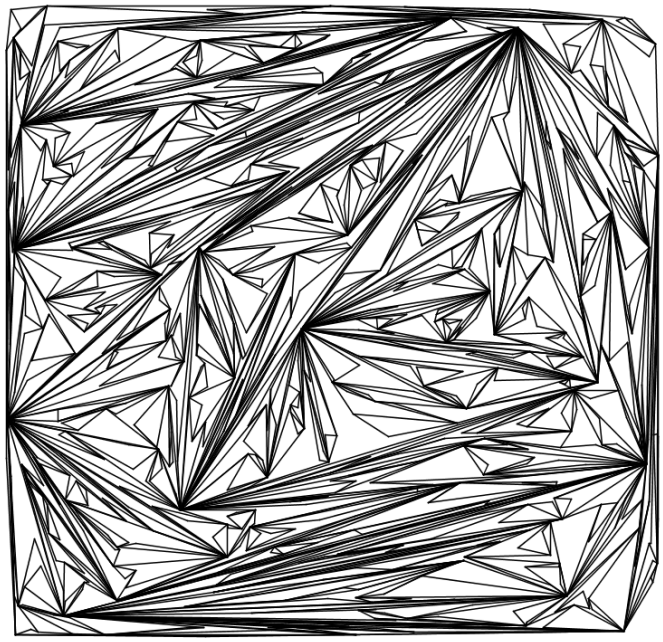


Obrázek 2.5. Obrázek ilustruje hledání geodetické cesty mezi vkládaným vrcholem V a rohy pseudo-trojúhelníku.

Metoda hledání geodetické cesty zde byla jen stručně vyložena. Pro lepší seznámení s touto metodou bych odkázal na práci [Trčka07], kde je mimo jiné i zápis pseudo-kódem.

3 Swap v pseudo-triangulaci

Pseudo-trojúhelníky vzniklé inkrementálním vkládáním nemají ideální tvar. I přes používání různých pomocných kritérií při vkládání bodu do pseudo-triangulace dochází ke kupení hran v některých vrcholech a vznikají „hubené trojúhelníky“ s malými vnitřními úhly. Jeden příklad se nachází na obrázku 3.1. Další příklady takových struktur je možno shlédnout v přílohách práce [Trčka07].



Obrázek 3.1. Ukázka pseudo-triangulace vytvořené inkrementálním vkládáním.

3.1 Zobecněný swap

V závěru diplomové práce [Trčka07] je uvedeno, že struktury vzniklé inkrementálním vkládáním by bylo možné vylepšit začleněním prohazování hran do algoritmu. Spolu s vedoucí práce jsem navrhli a implementovali algoritmus prohazování hran, dále jen *swap*, do již existujícího programového vybavení.

Swap je možné v programu provádět dvěma cestami. Buď se nejprve libovolným způsobem (načtením ze souboru, inkrementálním vkládáním nebo odebíráním hran z triangulace, viz [Trčka07]) vytvoří pseudo-triangulace na množině bodů a pak se provede swap na celé struktuře. Druhou možností je swap zabudovaný přímo do inkrementálního algoritmu, kde po každém přidání bodu dochází k prohazování hran v okolí právě přidaného bodu. Dodatečné prohazování hran ve struktuře jsme nazvali *post-processing swap* a prohazování hran v inkrementálním algoritmu *inkrementální swap*.

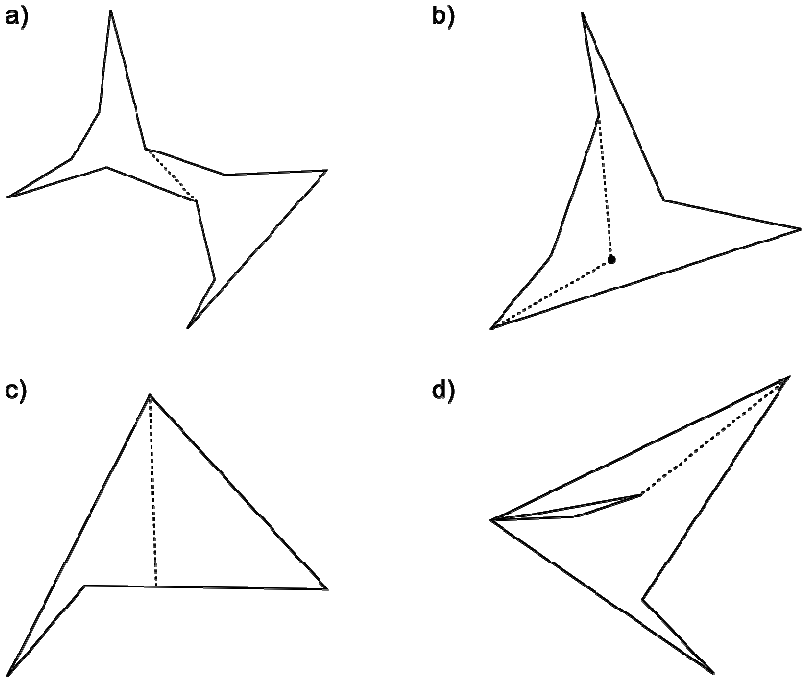
Algoritmus swapu je „rozdělen“ do následujících kroků :

1) vyhledání dvou sousedních pseudo-trojúhelníků
2) vytvoření polygonu spojením dvou sousedních pseudo-trojúhelníků
3) nalezení všech vnitřních diagonál nově vzniklého polygonu
4) zjištění, které diagonály rozdělují polygon na dva pseudo-trojúhelníky
5) zvolení nejvhodnější/nejvhodnějších diagonály/diagonál a rozdělení polygonu na dva nové pseudo-trojúhelníky

Sousední PT budeme nazývat dva pseudo-trojúhelníky, které spolu sousedí, tj. mají jednu nebo dvě společné hrany.

Nalezení *sousední PT* není díky implementaci datové struktury, resp. implementaci hrany jako tzv. „okřídlená hrana”, nijak závažný problém. Pro každou hranu lze nalézt sousední PT v čase O(1) viz odstavec 2.4.1. Společnou hranu sousedních PT budeme v dalším textu nazývat *hranicí sousedních PT*.

V dalším kroku je nutné spojit sousední PT do jednoho polygonu. Tento polygon bude obsahovat všechny hrany sousedních PT kromě jejich společných hranic. Na obrázku 3.2 můžeme vidět několik příkladů vytvoření polygonu. Přerušovanou čárou jsou vyznačeny hranice sousedních PT. Při vytváření polygonu se budeme držet orientace hran proti směru hodinových ručiček.



Obrázek 3.2. Ukázky polygonů vzniklých spojením sousedních PT.

Vnitřní diagonálou budeme nazývat všechny vnitřní hrany polygonu, vzniklého spojením sousedních PT. Vnitřní hrany leží celé uvnitř polygonu a nikde polygon neprotínají. Nalezení vnitřních diagonál je časově nejnáročnější operací (složitost $O(k^3)$, viz další podkapitola). K hledání diagonál polygonu slouží i *test na křížení hran* a *test na vnitřní diagonálu*. Po konzultaci s vedoucí práce jsme navrhli řešení těchto testů.

Test na křížení hran prochází všechny hrany obalu a testuje, zda nová diagonála neprotíná některou z hran polygonu. Test vychází ze znaménkového testu (kapitola 2.4.2). Nechť jsou dány orientované hrany *e1 (v1,v2)* a *e2 (v3,v4)*. Potom test na křížení hran provedeme tak, že spočítáme orientaci bodů *v3* a *v4* vůči hraně *e1* a orientaci bodů *v1* a *v2* vůči hraně *e2*. Pokud bude orientace bodů *v3* a *v4* vůči *e1* různá, tj. pokud bude jeden bod nalevo od *e1* a druhý napravo od *e1* a zároveň bude různá i orientace bodů *v1* a *v2* vůči *e2*, potom se budou hrany *e1* a *e2* křížit. V případě, že jeden vrchol hrany *e1*/*e2* leží na přímce dané hranou e2/*e1*, budou hodnoty orientace vrcholů *e1*/*e2* vyhodnoceny jako opačné.

V tabulce 3.1 jsou zaznamenány všechny možnosti výsledků testů pro test na křížení hran. Logická hodnota *0* znamená, že vrchol leží od hrany napravo a logická 1, že leží nalevo. V prvním sloupci je uvedeno, zda je bod *v3* nalevo od *e1*. Ve druhém, zda je *v4* nalevo od *e1* atd. Poslední sloupec říká, zda se hrany kříží.

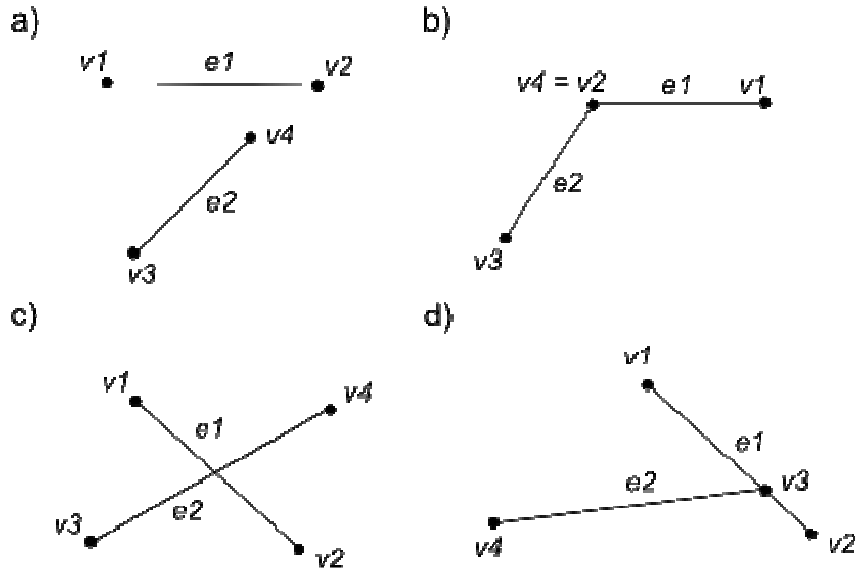
Z tabulky 3.1 lze vyvodit následující závěr:

Hrany *e1* a *e2* se kříží, jestliže: $(e1\_v3 \oplus e1\_v4) \cdot (e2\_v1 \oplus e2\_v2)$ - vrcholy *v3, v4* mají různou orientaci vůči *e1* a vrcholy *v1, v2* mají různou orientaci vůči *e2*.

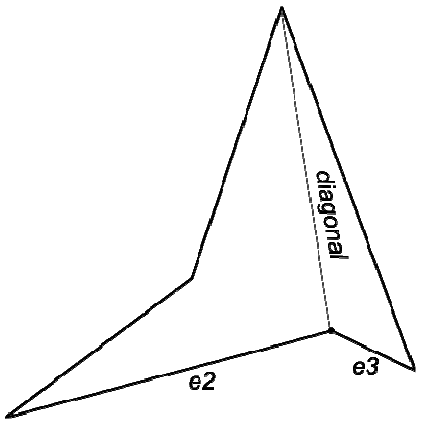
e1_v3	e1_v4	e2_v1	e2_v2	kříží se
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	**1**	**0**	**1**	**1**
0	**1**	**1**	**0**	**1**
0	1	1	1	0
1	0	0	0	0
1	**0**	**0**	**1**	**1**
1	**0**	**1**	**0**	**1**
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Tabulka 3.1. Testování křížení hran.

Na obrázku 3.3 a) můžeme vidět hrany *e1* a *e2*, které se nekříží. V tabulce 3.1 odpovídá tato pozice hran 3. řádku. Na obrázku 3.3 c) jsou hrany, které se kříží. V tabulce 3.1 je taková pozice hran na 7. řádku. Obrázek 3.3 b) ukazuje případ, kdy je průsečík hran v jednom z vrcholů. V takovém případě řekneme, že se hrany nekříží. Jedná se totiž o případ na obrázku 3.4, kde je patrné, že *diagonal* má společný bod s *e2* i *e3*, a je vnitřní diagonálou pseudo-trojúhelníku. A konečně případ na obrázku 3.3 d). Poloha vrcholů hrany *e1* vůči *e2* je zřejmá. Ovšem bod *v3* hrany *e2* leží na hraně *e1* a bude mu přiřazena opačná logická hodnota než bodu *v4*. Tento krok zajistí, že hrany budou vyhodnoceny jako křížící se. V tabulce odpovídá tato situace řádku 7.



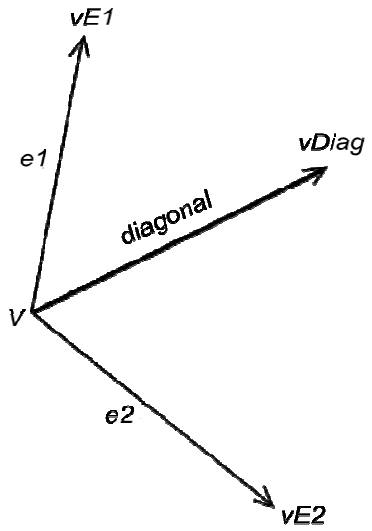
Obrázek 3.3. Ukázka možností křížení hran.



Obrázek 3.4. Ukázka vnitřní diagonály.

Pro test na křížení hran byl původně použit postup uvedený v příloze A. Jednalo se o počítání průsečíku diagonály a hrany polygonu. Takový postup se však ukázal být nevhodný kvůli větší časové náročnosti a také kvůli nepřesnostem spojených s numerickým výpočtem průsečíku.

Námi použitý test na vnitřní diagonálu vychází také z testu vzájemné polohy tří bodů. K popisu testu pomáhá obrázek 3.5. Nechť hrany *e1* a *e2* jsou sousední hrany polygonu. Nechť *diagonal* je testovaná diagonála. Vrcholy vE_1, vE_2, *vDiag* a *V* jsou vrcholy polygonu. Vrchol *V* je společný vrchol sousedních hran polygonu *e1* a *e2* a testované diagonály *diagonal*. Nechť jsou hrany *e1* a *e2* i diagonála *diagonal* orientovány směrem „od vrcholu *V*“, tj. *(V,vE1) (V,vE2) (V,vDiag).*Testuje se, zda se *diagonal* nachází nalevo od *e1*, resp. napravo od *e2*.



Obrázek 3.5. Ilustrace k testu na vnitřní diagonálu.

Výsledky testu jsou uvedeny v tabulce 3.2. Několik slov k této tabulce. Ve sloupci *nalevo od e1* je uloženo, zda testovaná diagonála, resp. bod *vDiag* leží nalevo od hrany *e1* ve směru její orientace. Sloupec *nalevo od e2* je analogií prvního sloupce pro hranu *e2*. V pravé části tabulky jsou uloženy výsledky, tj. zda testovaná diagonála leží v okolí bodu *V* uvnitř polygonu (1 - ano, 0 - ne). Sloupce *konvexní úhel*, resp. *nekonvexní úhel*, odlišují výsledky pro konvexní či nekonvexní úhel mezi hranami *e1* a *e2* orientovaný proti směru hodinových ručiček. Situace na obrázku 3.5: *diagonal* je nalevo od *e2* a napravo od *e1*, úhel mezi *e1* a *e2* je konvexní, výsledek testu je pozitivní.

nalevo od e1	nalevo od e2	konvexní úhel	nekonvexní úhel
0	0	0	1
0	1	1	1
1	0	0	0
1	1	0	1

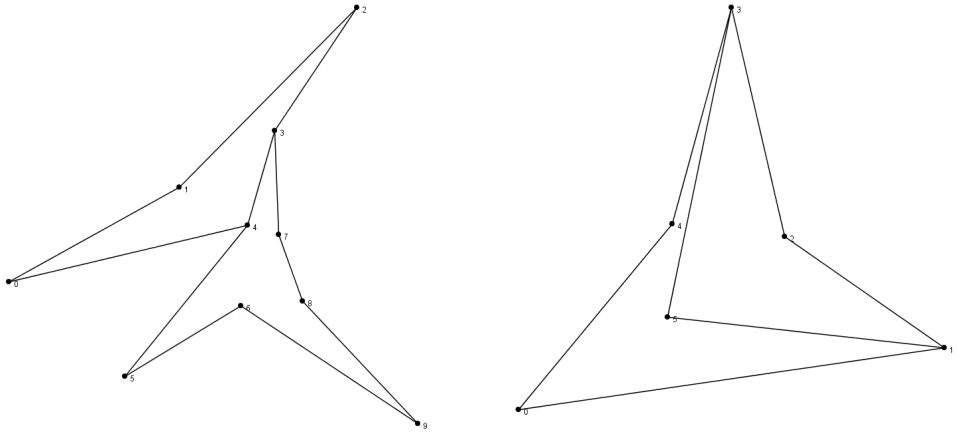
Tabulka 3.2. Test zda diagonála leží uvnitř polygonu v okolí bodu V.

Implementace swapu

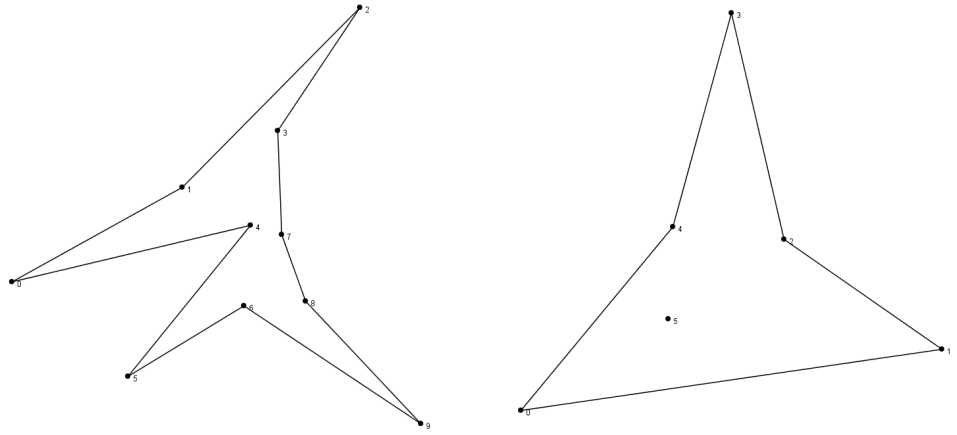
V algoritmu je klíčové vytvoření polygonu spojením sousedních PT a hledání diagonál v tomto polygonu.

Získání polygonu spojením sousedních PT: Mějme dva sousední PT (obrázek 3.6). Potřebujeme tedy získat polygon, vzniklý jejich spojením (obrázek 3.7). Vybereme jeden pseudo-trojúhelník. Postupným procházením hran, proti směru hodinových ručiček, nalezneme společnou hranu s druhým sousedním PT. Pokračujeme dál proti směru hodinových ručiček, přitom každou další hranu vložíme do polygonu. Totéž provedeme i druhým sousedem[6]. Hrany v polygonu, vzniklém takovýmto spojením, budou orientovány proti směru hodinových ručiček.

[6] Tato metoda na první pohled není nijak zvlášť složitá. Uvedl jsem ji zde, protože se ukázala být užitečná i v dalších částech programu. Osobně se mi zdála z pohledu svojí univerzálnosti zajímavá.



Obrázek 3.6. Dva sousední pseudo-trojúhelníky.



Obrázek 3.7. Polygon vzniklý spojením sousedních pseudo-trojúhelníků.

Pro lepší pochopení algoritmu použijme obrázek 3.8 a)[7] a algoritmus 3.1 zapsaný pseudo-kódem. Mějme dva sousední PT – *PT1*, *PT2*. Společná hrana *PT1* a *PT2* bude nazývána *border* (bude vstupní parametr metody `getPolygon()`). Šipky na obrázku znázorňují směr procházení pseudo-trojúhelníkem – proti směru hodinových ručiček. Algoritmus vezme v první iteraci *PT1* (proměnná `pst`) a označí jeho strany *C4C5*, *C5C6* a *C6C4* lokálními proměnnými `chain1`, `chain2` a `chain3` tak, že: `chain1` bude odpovídat *C6C4*. V `chain1` se bude nacházet také hrana `border`. `Chain2` bude obsahovat stranu *C4C5*. V `chain3` bude uložena strana *C5C6*. Dál se budou postupně procházet strany `chain1`, `chain2`, `chain3` a všechny jejich hrany se přidají do polygonu (v `chain1` pouze hrany ležící za `border`). Polygon je reprezentován proměnnou `polygon`. Hrana je přidána vždy nakonec `polygonu`.

> *Poznámka 1:* Na tomto místě je vhodné poukázat na obrázek 3.8 b). Všimněme si, že *b2* bude v `polygonu` na prvním místě. Pokud bychom označili *b2* jako `border` – vstupní parametr metody, bude hrana *border*, z obrázku 3.8 b), po první iteraci na konci `polygonu`. Snadno zjistíme během přidávání hran druhého sousedního PT do `polygonu`, zda sousední PT mají jednu nebo dvě společné

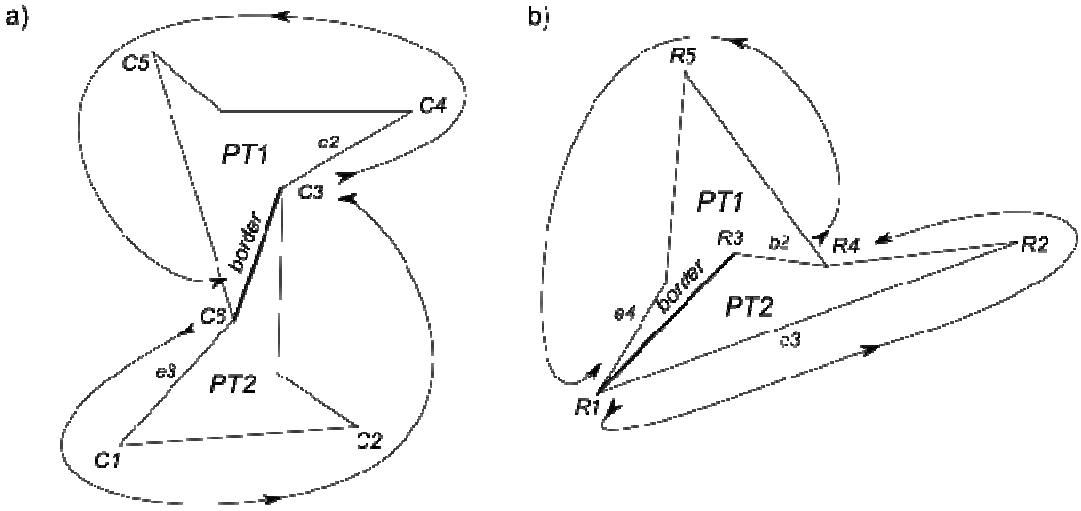
[7] K obrázku 3.8 b) se dostaneme pouze v úseku, kde se projeví přítomnost dvou společných hran sousedních PT.

hrany. Zjistíme to tak, že porovnáme právě vkládanou hranu druhého sousedního PT s první, resp. poslední vloženou hranou do `polygonu`.

Ve druhé iteraci cyklu se do `pst` uloží *PT2*. Do `chain1`, `chain2` a `chain3` se uloží strany `pst` (stejným způsobem jako v první iteraci, viz výše). Algoritmus bude postupně procházet `chain1`, `chain2` a `chain3` (stejně jako v první iteraci) a vkládat hrany do `polygonu` (stejně jako v první iteraci). Na rozdíl od prvního kroku se nyní bude každá vkládaná hrana testovat, zda není totožná s první, resp. poslední vloženou hranou do `polygonu`, viz poznámka 1. Pokud nastane případ, že právě vkládaná hrana bude totožná, hrana se do `polygonu` nevloží a bude také z `polygonu` odstraněna. Proměnná `secondBorder` se nastaví na aktuální hranu. Algoritmus pokračuje dál.

Na obrázku 3.8 b) pro vstupní hranu *border* ve druhé iteraci dojde algoritmus do fáze: `pst` = *PT2*, procházený řetězec `chain3` = *R2R3*, aktuální hrana `e` = *b2*. Z poznámky 1 víme, že *b2* byla již do `polygonu` přidána a nachází se na prvním místě. Algoritmus tedy vyhodnotí `e` a první hranu z `polygonu` jako totožné. Hranu *b2* z `polygonu` vyjme a uloží ji do `secondBorder`.

Poznámka 2: Pokud by byla `border` = *b2*, dospěl by algoritmus do fáze: druhá iterace, `pst` = *PT2*, aktuální hrana `e` = *border* (ta na obrázku 3.8 b). Hrana v `polygonu` na posledním místě je totožná s hranou `e` …



Obrázek 3.8. Ilustrace pro algoritmus vytváření polygonu spojením sousedních PT.

Algoritmus je zapsán pseudo-kódem jako algoritmus 3.1[8]. Informace, jestli sousední PT mají jednu nebo dvě společné hrany, je nutná pro hledání vnitřních diagonál polygonu, viz dále.

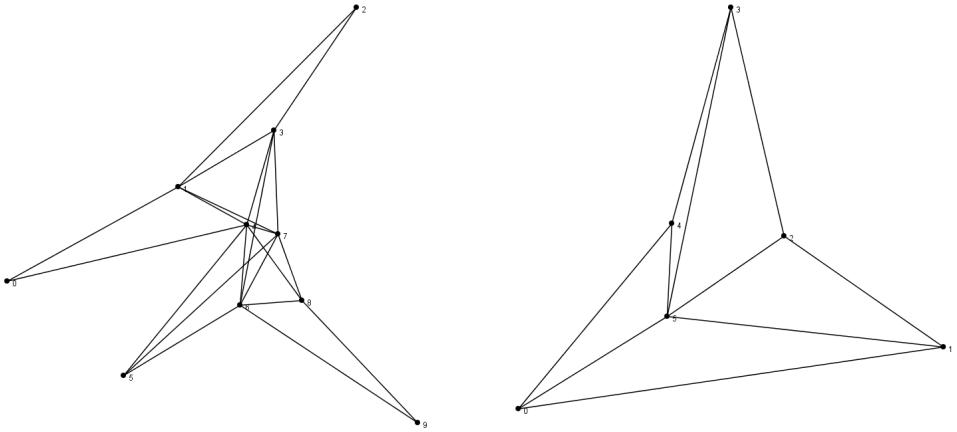
[8] Pseudo-kód byl psán česko-anglicky kvůli zjednodušení a lepší pochopitelnosti.

```
/*
 * Metoda vytvoří polygon spojením sousedních PT, které mají společnou
 * hranu – border.
 * vstup : Edge border – společná hrana sousedních PT.
 * výstup : Edge[] polygon – polygon vzniklý spojením sousední PT. Hrany
 * jsou orientovány proti směru hodinových ručiček.
 * Edge secondBorder – druha společná hrana sousedních PT, pokud existuje
 * (jinak null).
 */
getPolygon(Edge border) {
   ArrayList<Edge> polygon; // polygon vzniklý spojením sousedních PT
   PsTriangle pst;    // pseudo-trojúhelník, se kterým se aktuálně pracuje
   LinkedList<Edge> chain1, chain2, chain3;  // strany v sousedních PT
   // druhá hrana tvořící hranici, zpočátku null
   Edge secondBorder = null;

   for (oba sousední PT) {
      pst = první soused  // ve druhé iteraci druhý soused

      /*
       * Lokální přečíslování stran v pseudo-trojúhelníku.
       * Chain1 bude označena strana, ve které leží border. Chain2 a chain3
       * budou pak označeny následující dvě strany v orientaci proti směru
       * hodinových ručiček.
       */
      switch(podle strany, ve které leží border v pst) {
         case 1:
            chain1 = pst.strana1;
            chain2 = pst.strana2;
            chain3 = pst.strana3;

         obdobně pro případ 2 a 3
      }

      /*
       * jako první se do polygonu vloží hrany z chain1, jež se
       * v tomto řetězci nachází až za border
       */
      for(Edge e : chain1) {
         if (e se nachází v chain1 až za border)
            if (pst je první soused)
               polygon.add(e);
            // pst je druhý ze sousedních PT
            else {
               /*
                * v případě, že bych se pokoušel do polygonu vložit
                * jednu hranu podruhé – stává se při průchodu druhého
                * sousedního PT - vkládanou hranu vždy porovnávám s první a
                * poslední hranou v polygonu
                */
               if (e již byla jednou vložena)
                  // odstranění hrany z polygonu, e je druhá hranice
                  secondBorder = polygon.remove(e);
               else  // přidáme hranu do polygonu
                  polygon.add(e);
            }
         }
      }

      /*
       * stejný postup i pro chain2 a chain3
       * nakonec se vloží hrany, které leží v chain1 před border
       */
      …
```

```
    }

    /*
     * metoda vrací polygon jako pole hran – ArrayList<Edge> pouze při
     * naplňování polygonu
     * vrací též druhou hranu společnou sousedním PT, pokud existuje
     */
    Object[] ret = {polygon.toArray(), secondBorder};
    return ret;
}
```

Algoritmus 3.1. Vytvoření polygonu spojením dvou sousedních PT.

Získání diagonál polygonu: Dál potřebujeme získat všechny vnitřní diagonály polygonu, který vytvoří metoda `getPolygon()`. Jedná se o nejsložitější metodu z celého řešení (časově i implementačně). Metoda v prvních dvou cyklech vybírá všechny existující diagonály polygonu. Každou takovou potenciální vnitřní diagonálu polygonu je nutné podrobit testu na vnitřní diagonálu a testu na křížení hran. Obrázek 3.9 ukazuje příklad vytvoření vnitřních diagonál pro jednu nebo dvě společné hrany.



Obrázek 3.9. Vytvoření vnitřních diagonál polygonu Tento obrázek můžeme brát jako následovníka obrázku 3.7 v „obrázkové posloupnosti“ pro prohazování hran.

Jak již bylo zmíněno, vyhledávání vnitřních diagonál polygonu bude různé pro sousední PT s jednou společnou hranou a pro sousední PT se dvěmi společnými hranami.

Nejprve si ukážeme výběr vnitřních diagonál pro sousední PT, které mají jednu společnou hranu. Tento výběr je popsán pseudo-kódem v algoritmu 3.2.

Nechť je `polygon` pole hran vrácené metodou `getPolygon()`. Zavedeme si proměnnou `points`, do které uložíme posloupnost vrcholů vzniklou z `polygonu`. Body v `points` jsou orientované proti směru hodinových ručiček a reprezentují stejný polygon jako pole hran na vstupu. V prvních dvou cyklech algoritmu se vytvářejí všechny diagonály polygonu z bodů `[i,j]`. Nastavení proměnné `j` zaručuje, že se každá diagonála vytvoří pouze jednou (mohlo by dojít k vytvoření diagonál `[i,j]` i `[j,i]`). Dál se nevytváří diagonály se sousedním vrcholem v `points`, protože se jedná o hrany, které patří do `polygonu` a ty rozhodně vnitřní diagonály nejsou.

Každou nově vzniklou diagonálu `diag` je nutné podrobit testu na vnitřní diagonálu a testu na křížení hran. Test na vnitřní diagonálu se provádí pouze s hranami, které se nacházejí

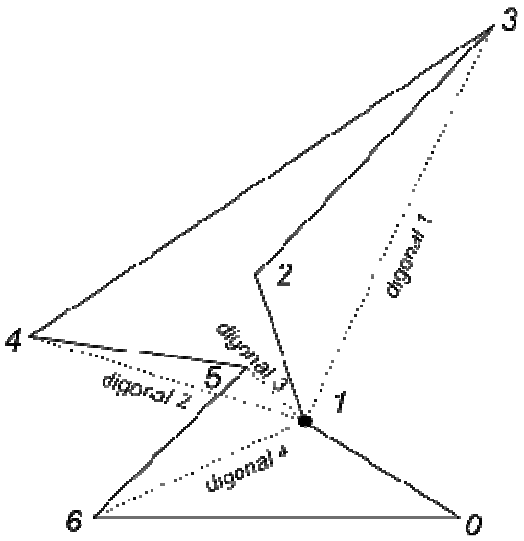
v `polygonu` a které mají společný vrchol `i`. V algoritmu jsou takové hrany označeny proměnnými `prev` a `next`. Test na křížení hran se naopak provádí se všemi hranami v `polygonu` kromě `prev` a `next`. V případě, že `diag` projde testem na vnitřní diagonálu i testem na křížení hran, bude vložena do pole `diagonals` a algoritmus bude pokračovat výběrem další diagonály.

```
/*
 * Metoda získá všechny vnitřní diagonály polygonu
 * polygon – polygon vzniklý spojením sousedních PT
 * vrací pole vnitřních diagonál
 */
getDiagonals(Edge[] polygon) {
      ArrayList<Edge> diagonals = new ArrayList<Edge>();
      // jednotlivé body v polygonu, orientované proti směru hodinových
      // ručiček
      int[] points = getPoints(polygon);
      Edge diag;  // aktuální diagonála
      boolean cross, in;   // příznaky pro testy

      /*
       * v prvních dvou cyklech jsou vytvořeny všechny potenciální
       * vnitřní diagonály polygonu
       */
      for (int i = 0 ; i < points.length - 1; i++) {
         // orientace proti směru hodinových ručiček
         Edge prev = hrana, jejíž koncový vrchol je i;
         Edge next = hrana, jejíž počáteční vrchol je i;

         for (int j = (i + 2); j < points.length; j++) {
            // nová diagonála, je nutné ji otestovat
            diag = new Edge(points[i], points[j]);

            // test na vnitřní diagonálu polygonu
            if ((in = isInnerDiagonal(prev, next) == false)
               // pokud se nejedná o diagonálu ležící uvnitř polygonu
               continue;

            // test na křížení hran
            cross = false;
            for (k : všechny hrany obalu kromě prev a next) {
               // pokud se diag kříží s některou hranou polygonu
               if ((cross = isCrossing(diag, polygon[k])))
                  break;
            }
            if ((!cross) && in) diagonals.add(diag);
         }
      }
      return diagonals.toArray();
   }
```

Algoritmus 3.2. Hledání vnitřních diagonál polygonu, jež byl vytvořen v algoritmu 3.1.

Na obrázku 3.10 je ukázka vytvoření diagonál pro `i` = 1 a `j` = 3, 4, 5, 6. Při testování diagonál z tohoto obrázku se dobereme následujících výsledků:

Diagonal 1 (vrcholy *1,3*) : Tato diagonála neprojde testem na vnitřní diagonálu.
Diagonal 2 (vrcholy *1,4*) : Tato diagonála se kříží s hranou *5,6*.
Diagonal 3 (vrcholy *1,5*) : Tato diagonála bude přijata jako vnitřní diagonála. Dokonce se jedná o diagonálu, která rozděluje polygon na dva pseudo-trojúhelníky.
Diagonal 4 (vrcholy *1,6*) : Tato diagonála bude také přijata jako vnitřní diagonála.



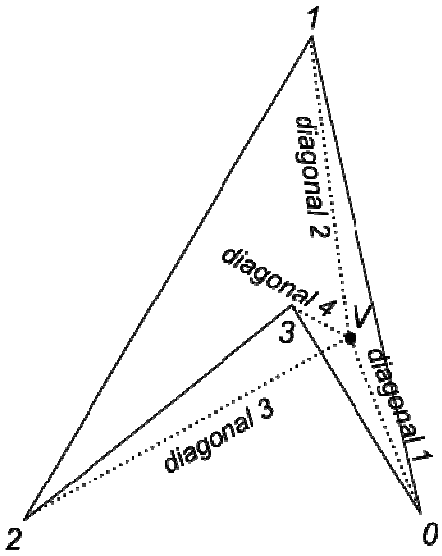
Obrázek 3.10. Ukázka vytvoření všech diagonál pro jeden bod polygonu.

V případě, že sousední PT mají dvě společné hrany, dochází ke zjednodušení výše popsané metody. V libovolném místě uvnitř polygonu se bude nacházet bod, který je společným bodem společných hran sousedních PT (viz obrázek 3.2 b)). Vnitřní diagonálou budeme v tomto případě nazývat každou hranu ležící uvnitř polygonu, která spojuje libovolný vrchol polygonu s tímto vnitřním bodem.

Je dán `polygon`, vzniklý spojením sousední PT se dvěma společnými hranami, jako vstupní parametr. Je dán vnitřní bod `inPoint`, společný bod společných hran sousedních PT, jako vstupní parametr. Algoritmus 3.2 bude možné zjednodušit následujícím způsobem: vnitřní diagonály budou hledány jako spojnice vrcholů `polygonu` a bodu `inPoint` (`diag = [inPoint, i]`), které leží uvnitř `polygonu`. Test, zda leží `diag` uvnitř polygonu bude v tomto případě představovat pouze test na křížení hran. Metoda `getDiagonals(polygon, inPoint)`[9], bude mít složitost $O(k^2)$.

Na obrázku 3.11 je znázorněn příklad vytváření diagonál pro polygon vzniklý spojením sousedních PT se dvěmi společnými hranami. Na *diagonal 3* (body *2,3*) se dá ukázat, že diagonálu není nutné podrobovat testu na vnitřní diagonálu. Toto tvrzení vychází z předpokladu, že bod *V* leží uvnitř polygonu a diagonála má krajní body *V* a vrchol polygonu. Pokud diagonála v okolí jednoho z koncových bodů neleží uvnitř polygonu (*diagonal 3* bod *2*) a v okolí druhého koncového bodu leží uvnitř polygonu (*diagonal 3* bod *V*), pak se musí v určitém místě protnout s hranami polygonu (*diagonal 3* se protíná s hranou *3,0*)[10].

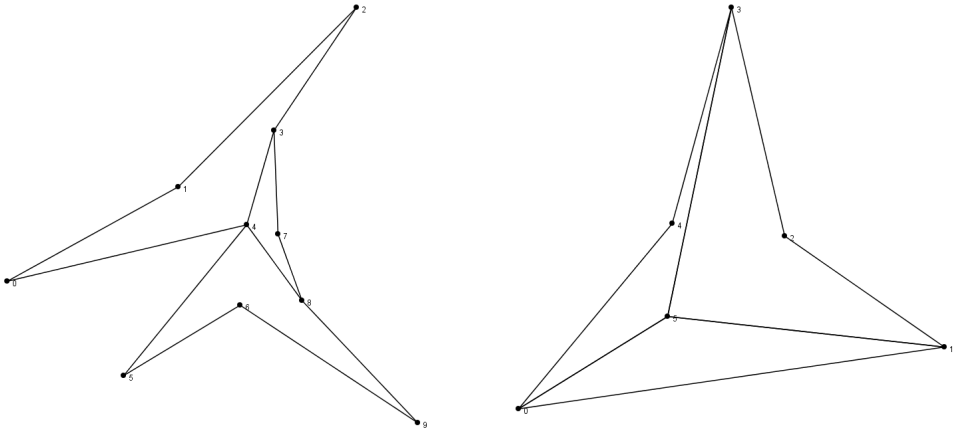
[9] Pro tuto metodu není uveden zápis pseudo-kódem, neboť algoritmus je zjednodušením algoritmu 3.1.
[10] Tímto jsem se snažil dokázat, že test na křížení hran je postačující podmínkou pro určení vnitřní diagonály.



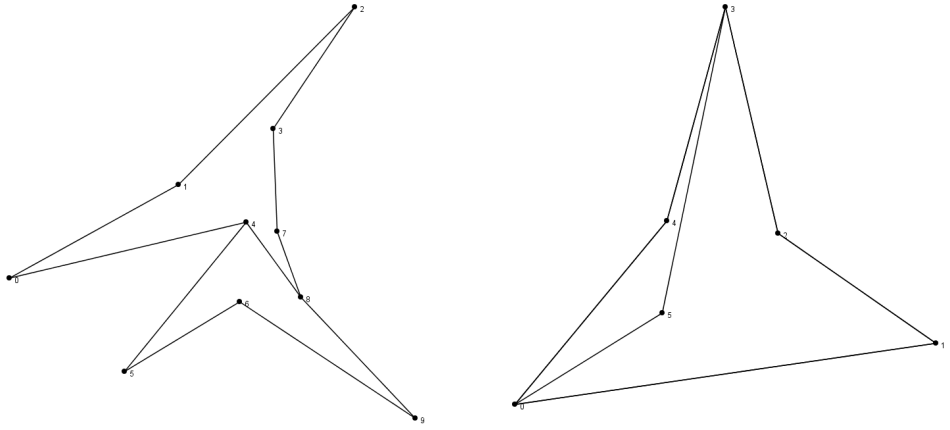
Obrázek 3.11. Ukázka hledání vnitřních diagonál pro sousední PT se dvěma společnými hranami.

Nalezení diagonál rozdělující polygon na dva pseudo-trojúhelníky, výběr jedné z nich a vytvoření nových dvou pseudo-trojúhelníků: Postupně se otestují všechny vnitřní diagonály, zda rozdělují polygon na dva pseudo-trojúhelníky. V případě sousedů se dvěmi společnými hranami se testují vždy dvojice diagonál. Prozatím se spokojíme s výběrem hrany (hran) s minimální délkou. V kapitole 3.3 Kritéria pro výběr nové hrany a v kapitole 4 Testy se budeme zabývat ještě jinými kritérii pro výběr diagonál.

Obrázek 3.12 je následovníkem obrázku 3.9. Ukazuje výběr diagonál rozdělující polygon na dva pseudo-trojúhelníky. Obrázek 3.13 ukazuje vytvoření nových sousedních PT. V tomto případě byly vybrány nejkratší společné hrany.



Obrázek 3.12. Výběr všech diagonál, jež rozdělují polygon na dva pseudo-trojúhelník.



Obrázek 3.13. Vytvoření nových pseudo-trojúhelníků.

Na závěr uvedeme celý algoritmus zobecněného prohazování hran. Vstupním parametrem bude libovolná hrana pseudo-triangulace `e`. Metoda `getPolygon(e)` uvedená výše vrátí `polygon` vzniklý spojením sousedních PT, jejichž společná hrana je `e`, orientovaný proti směru hodinových ručiček. Dále jsou v `polygonu` vyhledány všechny vnitřní diagonály - `diagonals` metodou `getDiagonals(polygon)`, resp. `getDiagonals(polygon, inPoint)`. V dalším kroku algoritmus vybírá z `diagonals` všechny vnitřní diagonály, které rozdělují polygon na dva pseudo-trojúhelníky – `flipDiagonals`. Z `flipDiagonals` je nutné podle zadaného kritéria (viz kapitola 3.3, prozatím se vybere nejkratší hrana/y) vybrat nejvhodnější diagonálu(y) – `newBorder[]`. Pokud se nové rozdělení polygonu odlišuje od starého rozdělení (hrana `e` je různá od `newBorder` nebo hrana `e` a `secondBorder` je různá od `newBorder[]`), budou vytvořeny nové dva sousední PT, které nahradí v pseudo-triangulaci původní sousední PT. Tento postup je zapsán pseudo-kódem jako algoritmus 3.3.

```
/*
 * Metoda spojí sousední PT, najde všechny vnitřní diagonály,
 * zjistí, které diagonály rozdělují polygon na pseudo-trojúhelníky,
 * vybere nejvhodnější diagonálu – zatím tu nejkratší a provede
 * potřebné změny ve struktuře.
 * Vstupní parametr je libovolná hrana pseudo-triangulace.
 * Návratová hodnota: true – byl proveden swap, false – nebyl proveden
 * swap.
 */
flip(Edge e) {
   // hrana se nachází v obalu struktury
   if (e.pst1 == null || e.pst2 == null)
      return false;

   // spojení sousedních PT do polygonu, nalezení druhé společné hrany
   Edge[] polygon = getPolygon(e)[0];
   Edge secondBorder = getPolygon(e)[1];
   Edge[] diagonals;    // vnitřní diagonály polygonu
   // diagonály rozdělující polygon na nové PT
   // potenciální nové hrany
   Edge[] flipDiagonals;
```

```
   if (jedna společná hrana, secondBorder == null) {
      // získáni diagonal
      diagonals = getDiagonals(polygon);
      // získání diagonál rozdělující polygon na dva pseudo-trojúhelníky
      flipDiagonals = getFlipableDiagonals(diagonals);
      // vyber optimální hranice podle zadaného kriteria
      Edge newBorder = chooseBest(flipDiagonals, criterium);
      // pokud se nová hranice nerovná původní hranici e
      if (e != newBorder) {
         vytvoření nových pseudo-trojúhelníků z polygonu a newBorder;
         úprava struktury;
      }
   }
   else {
      // hranice se skládá ze dvou hran, které mají společný jeden
      // bod, tento bod leží uvnitř polygonu
      Edge[] border = {e, secondBorder};
      // bod uvnitř polygonu
      int inPoint = společný bod e a secondBorder;
      // vytvoření vnitřních diagonál
      diagonals = getDiagonals(polygon, inPoint);
      // výběr diagonál, rozdělujících polygon na dva sousední PT
      flipDiagonals = getFlipableDiags(polygon, diagonals);
      // výběr optimálního rozdělení polygonu podle zadaného kritéria
      Edge[] newBorder = chooseBest(flipDiagonals, criterium);
      if (border != newBorder) {
         vytvoření nových pseudo-trojúhelníku z polygonu a newBorder;
         úprava struktury;
      }
   }
}
```

Algoritmus 3.3. Kompletní postup prováděný při prohazování hran.

3.2 Zjednodušení algoritmu

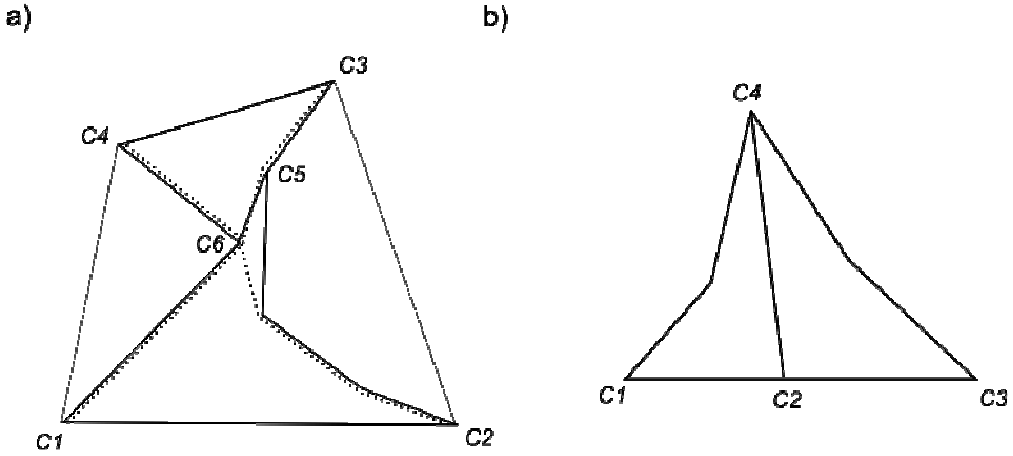
Je možné zrychlit postup hledání nové **hranice**[11]? Metoda *getDiagonals()* v předchozí podkapitole má, jak už bylo několikrát řečeno, časovou složitost $O(k^3)$, pro k rovno počtu vrcholů v sousedních PT. V této části práce se budeme snažit urychlit hledání nové hranice mezi sousedními PT.

Nechť máme `polygon` a `secondBorder` vrácené metodou `getPolygon(Edge border)`, viz minulá kapitola. Podle mých poznatků dosažených při návrhu a implementaci zobecněného swapu budou **diagonály, jež rozdělují `polygon` na nové pseudo-trojúhelníky v minimální pseudo-triangulaci, právě jedna, dvě nebo tři**. Počet diagonál závisí na počtu společných hran sousedních PT, tzn. pokud `secondBorder` bude existovat – nebude `null`. Pokusím se ukázat, že podle počtu společných hran sousedních PT je možné říci, mezi kterými dvěma vrcholy `polygonu` budu hledat novou hranici.

V případě **jedné společné hrany** obrázek 3.14 a), budou hledané diagonály dvě. První je již existující hranice *C6C5*. Dál použijme analogii se čtyřúhelníkem. Jako je možné čtyřúhelník rozdělit na dva trojúhelníky pouze dvěma způsoby, tak to samé bych tvrdil o polygonu vzniklém spojením sousedních PT v minimální pseudo-triangulaci, které mají jednou společnou hranu. Pokud by to byla pravda, mohl bych najít druhou společnou hranu jako spojnici protilehlých rohů (*C1C3* resp. *C2C4*), tj. geodetickou cestu mezi nimi, viz kap. 2.3.2.

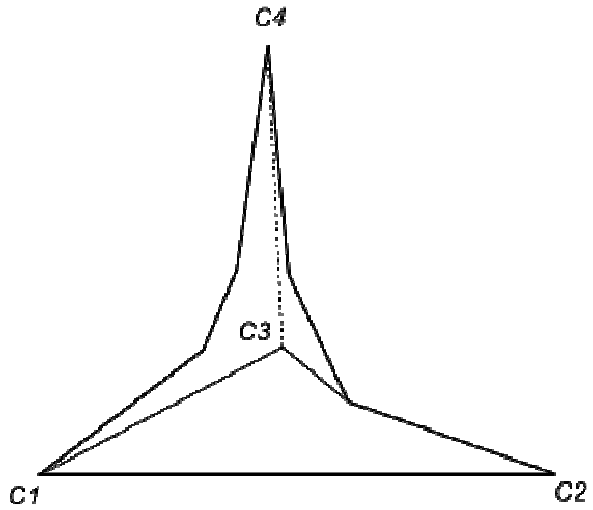
[11] Tj. diagonály, rozdělující dva sousedy na dva jiné pseudo-trojúhelníky.

Na obrázku 3.14 b) mají sousední PT (*C1C2C4* a *C2C3C4*) také jednu společnou hranu *C2C4*. V tomto případě však není možné tuto hranu nahradit[12] ze dvou důvodů: 1) Neexistuje jiná vnitřní diagonála polygonu vráceného metodou `getPolygon(`*C2C4*`)`, která by rozdělovala polygon na dva pseudo-trojúhelníky. 2) Pokud bychom hranu *C2C4* odstranili, vznikne na straně *C1C3* přímý úhel, což je v rozporu s definicí pseudo-trojúhelníku z kapitoly 2.1.



Obrázek 3.14. Sousední PT s jednou společnou hranou.

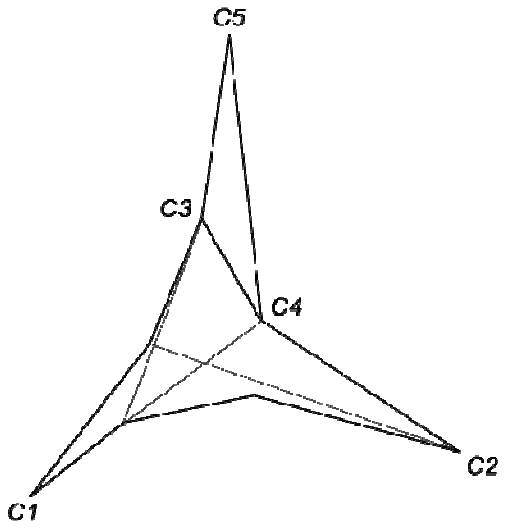
Pokud mají **sousední PT společné hrany dvě**, jedná se evidentně o analogii s vkládáním bodu do pseudotriangulace. Na obrázku 3.15 můžeme vidět jeden takový příklad. Polygon vzniklý spojením sousedních PT je pseudo-trojúhelník *C1C2C4*. Uvnitř polygonu se nachází jeden vnitřní bod *C3*. Společné hrany sousedních PT si můžeme představit jako dvě geodetické cesty z rohů polygonu do vnitřního bodu (*C1C3* resp. *C2C3*). Mohu tedy odkázat na kapitolu 2.3.2 Přidání bodu dovnitř pseudo-triangulace a říci, že jediná další diagonála, která rozděluje polygon na dva pseudo-trojúhelníky, je geodetická cesta mezi body *C3* a *C4*.



Obrázek 3.15. Sousední PT se dvěma společnými hranami.

[12] Tímto případem se nebudeme dále zabývat. Pokud se takový případ vyskytne v programu, algoritmus samozřejmě swap neprovede.

Tento postup bude fungovat pouze v případě, že pseudo-triangulace, se kterou pracujeme, **je minimální**. Na obrázku 3.16 je zobrazen příklad, kdy pseudo-triangulace není minimální, protože hrana *C3C4* mezi sousedními PT *C1C2C3* a *C3C4C5* je odebratelná. Na obrázku 3.16 je též vyznačeno několik možností, jak jinak lze rozdělit sousední PT. Pokud se pokusíme nalézt novou společnou hranu sousedních PT postupem uvedeným výše, tj. hledáním geodetické cesty v polygonu (vzniklého spojením sousedních PT) mezi vrcholem *C1* a *C5*, nebudeme úspěšní. Polygon je totiž také pseudo-trojúhelník s rohy *C1C2C5*. Geodetická cesta mezi vrcholy *C1* a *C5* tedy již existuje, resp. je to strana *C5C1*.

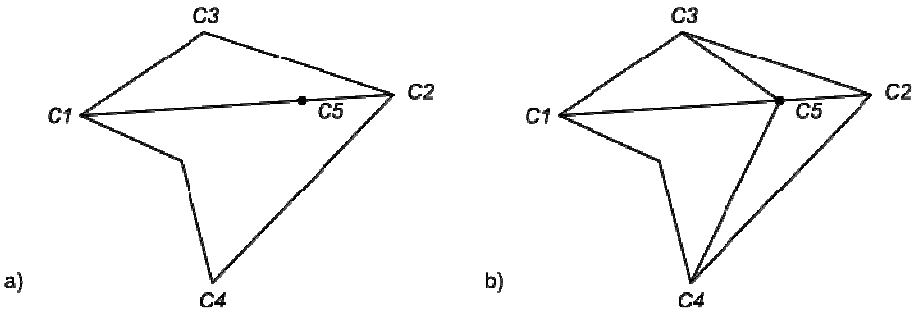


Obrázek 3.16 Případ sousedních PT v pseudo-triangulaci, která není minimální.

Pseudo-triangulace vytvořená inkrementálním vkládáním (viz kapitola 2.3) není minimální pseudo-triangulace, ačkoli se jí, podle testů v práci [Trčka07], velmi podobá. Důvod, proč tomu tak je, je uveden v kapitole 2.3.3 Přidání bodu na hraně pseudo-triangulace[13]:

Pokud přidáme bod na hranu pseudo-triangulace, rozdělí se tato hrana na dvě nové hrany a sousední PT, sousedící spolu na této hraně, se též rozdělí (každý na dva nové pseudo-trojúhelníky) [Trčka07]. Na obrázku 3.17 a) jsou znázorněny dva sousední PT *C1C2C3* a *C1C4C*2 a bod *C5* vložený na jejich společnou hranu. Pokud *C5* přidáme do pseudo-triangulace, rozdělí nám sousední PT podle obrázku 3.17 b). Z obrázku 3.17 b) je též patrné, že hrana *C5C2* (společná pro sousední PT *C2C3C5* a *C2C5C4*) je odebratelná, tudíž nově vzniklá pseudo-triangulace není minimální.

[13] Při studování práce [Trčka07] jsem si myslel, že struktura, vytvořená inkrementálním vkládáním je minimální PT.



Obrázek 3.17 Rozdělení pseudo-trojúhelníků po vložení bodu na společnou hranu.

Jak bylo řečeno na začátku této podkapitoly, urychlení hledání nové hranice mezi sousedními PT funguje pouze v případě, že tyto dva sousední PT jsou minimální pseudo-triangulace. V předešlém odstavci je ukázáno, že při konstrukci pseudo-triangulace inkrementálním vkládáním vznikají odebratelné hrany, proto v tomto případě hledání nové hranice výše popsaným postupem nefunguje a nová hranice se musí hledat pomalejším algoritmem popsaným v kapitole 3.1.

Implementace

Implementace zrychleného prohazování hran v pseudo-triangulaci je z velké části stejná jako implementace zobecněného swapu v kapitole 3.2.1 metoda `flip(Edge e)`. Jediná odlišnost je ve vyhledávání nových společných hran sousedních PT.

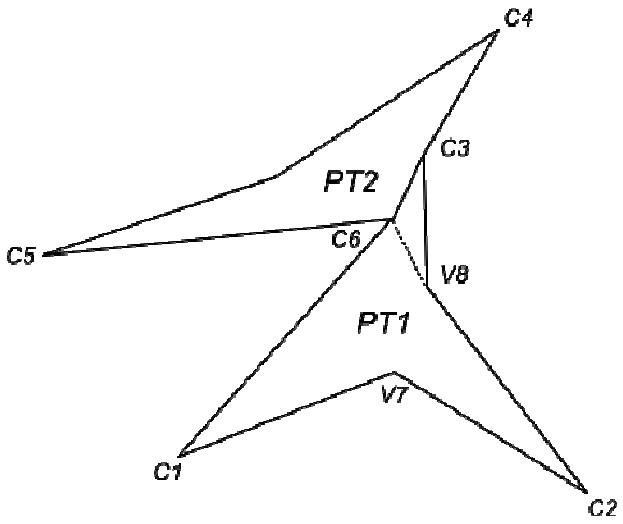
Vyhledávání nové hranice pro jednu společnou hranu sousedních PT je realizováno jako hledání geodetické cesty mezi dvěma rohy polygonu. Tyto rohy zjistíme snadno, díky implementaci datové struktury (kapitola 2.4.1). Každá hrana ve struktuře zná nejen sousední PT, ve kterých se nachází, ale i strany, ve kterých se nachází. Podle odkazu na stranu jsme schopni najít roh protilehlý straně v pseudo-trojúhelníku v čase O(1).

Vyhledávání nové hranice pro dvě společné hrany sousedních PT je téměř identické jako hledání geodetických cest při vkládání nového bodu do struktury v algoritmu konstrukce pseudo-triangulace inkrementálním vkládáním [Trčka07] (kapitola 2.3.2). Rozdíl je, že nám stačí nalézt jen jednu geodetickou cestu, další dvě už máme v podobě stávajících společných hran.

Popíšeme si nyní algoritmus pseudo-kódem zapsaným v alg. 3.4 a vysvětlíme jeho funkci. Metodu pro hledání nových společních hran budeme nazývat `findDiagonal(e, type, secondBorder, polygon)`, kde `e` je společná hrana sousedních PT, `type` je příznak, zda se jedná o jednu či dvě společné hrany sousedních PT, `secondBorder` je případná druhá společná hrana a `polygon` vzniklý spojením sousedních PT. V případě jedné společné hrany (`type` = 1, `secondBorder` = null) algoritmus zjistí protilehlé rohy k hraně `e`, `c1` a `c2`, a hledá geodetickou cestu mezi nimi. V prvním kroku je hledána geodetická cesta mezi vrcholy `c1` a `c2`. Vrchol, z něhož je z pohledu `c1` vrchol `c2` již viditelný, označme `v3`. Pokračuje se hledáním geodetické cesty z rohu `c2` do `c1`. Vrchol, z něhož je `c1` viditelný, označme `v4`. V dalších iteracích se budou hledat geodetické cesty mezi vrcholem `c1` a vrcholem `v4` (výsledný vrchol se uloží do `v3`), resp. mezi vrcholem `c2` a `v3` (výsledný vrchol se uloží do `v4`), dokud se budou proměnné `v3` a `v4` měnit. Diagonála v `polygonu` tvořená vrcholy `v3` a `v4` bude v průběhu iterací konvergovat k hledané hraně.

Na obrázku 3.18 si názorně ukážeme funkčnost:

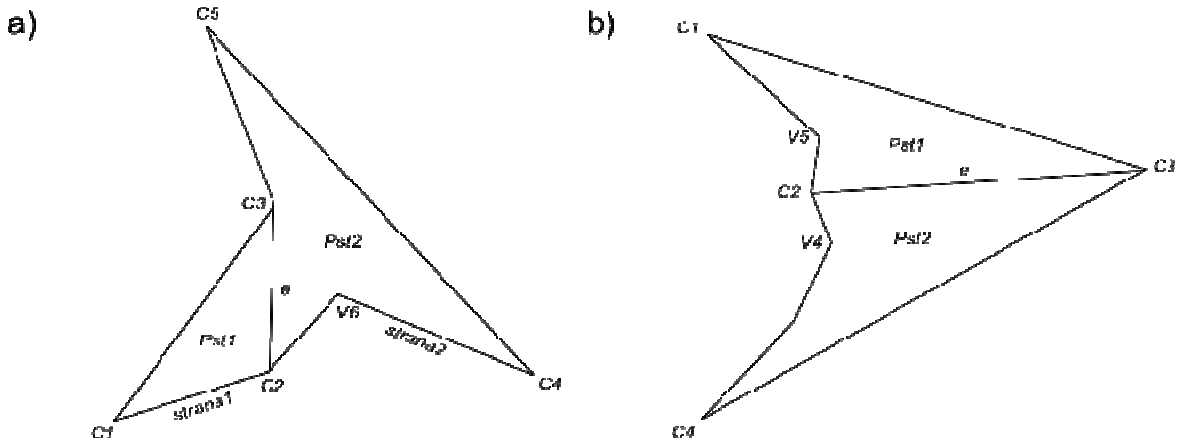
Sousední PT (*PT1 = C1C2C3* a *PT2 = C4C5C6*) mají společnou hranu *C3C6*. Proti této hraně se v *PT1* nachází roh `c1` = *C2* a v *PT2* roh c2 = *C5*. Při hledání geodetické cesty z `c1` do `c2` se dostaneme do vrcholu `v3` = *V7*. Naopak při hledání geodetické cesty z `c2` do `c1` se dostaneme do vrcholu `v4` = *C6*. Ve druhé iteraci se bude hledat geodetická cesta mezi vrcholy `c1` a `v4` (na obr. 3.18 vrcholy *C2* a *C6*) – výsledek `v3` = *V8* a geodetická cesta mezi vrcholy `c2` a `v3` (na obr. 3.18 vrcholy *C5* a *V8*) – výsledek `v4` = *C6*. V další iteraci se vrcholy `v3` a `v4` nezmění a bude vytvořena nová diagonála s vrcholy `v3`,`v4` (*V8*,C6).



Obrázek 3.18 Jedna společná hrana.

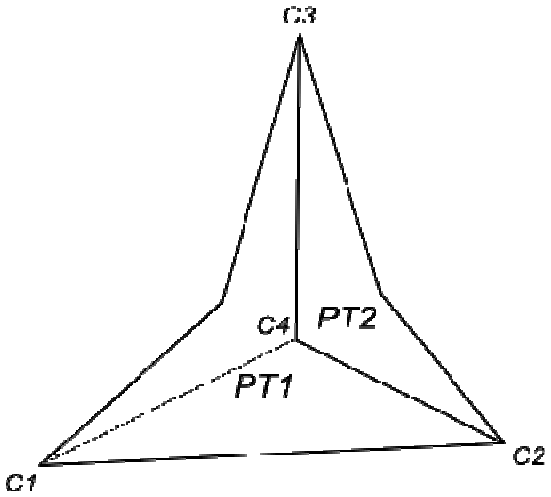
Během testování různých vstupních množin se ukázalo, že metoda popsaná výše nefunguje korektně u relativně velkého počtu sousedních PT[14]. Na obrázku 3.19 jsou ukázány dvě konfigurace sousedních PT, u kterých nebylo vráceno správné řešení. Pokud zkusíme postup popsaný výše na oba příklady z obrázku 3.19, dostaneme po několika iteracích `v1` = *C2*, `v2` = *C2*, `v3` = *C2* a `v4` = *C2*. To samozřejmě není správné řešení. Pro obrázek 3.19 a) by měla být nová společná hrana (*C1*, *V6*) a pro obrázek 3.19 b) hrana (*V4*, *V5*). Tento zjevný nedostatek byl dodatečně odstraněn jinou počáteční volbou vrcholů `v3` a `v4`. Apriorně byl vrchol `v3` vrcholem, z něhož byl z pohledu `c1` vrchol `c2` již viditelný. Po testování algoritmu byla počáteční hodnota `v3` nastavena na druhý vrchol strany zakrývající výhled z *C1* na *C4*, resp. ne předposlední vrchol strany zakrývající výhled z *C4* do *C1*, viz 3.19 b). Zda bude `v3/v4` nastaven na druhý nebo předposlední vrchol strany závisí na tom, jestli strana následuje nebo předchází roh `c1/c2`.

[14] Jednalo se o počty v řádu jednotek na testovaných vstupních množinách.



Obrázek 3.19. Konfigurace sousedních PT, pro než metoda findDiagonal() vrací chybný výsledek.

Ilustrace na obrázku 3.20 ukazuje případ dvou společných hran. Algoritmus vybere vrchol *C1* protilehlý společným hranám *PT1* a *PT2* a najde geodetickou cestu mezi tímto vrcholem a bodem *C4*, jež je společný pro obě společné hrany. V tomto případě bude nová hrana *C1C4*. Jedná se o téměř totožný postup jako při konstrukci pseudo-triangulace inkrementálním vkládáním v práci [Trčka07].



Obrázek 3.20. Dvě hrany sousedních PT.

Algoritmus zapsaný pseudo-kódem jako algoritmus 3.4.

```
/*
 * e – společná hrana sousedních PT
 * type : 1 – jedna společná hrana 2 – dvě společné hrany
 * secondBorder – druhá společná hrana, pokud existuje
 * vrací hledanou hranici, jedná se vždy o jednu hranu
 */
findDiagonal(Edge e, int type, Edge secondBorder, Edge[] polygon) {
   if (jedna společná hrana) {
      // protilehlé vrcholy c1 pro PT1, c2 pro PT2
```

```
      int c1, c2;
      // určení vrcholu protilehlého společné hraně
      switch (číslo strany v PT1 pro hranu e) {
         case 1 : c1 = vrchol 3 PT1;
         case 2 : c1 = vrchol 1 PT1;
         case 3 : c1 = vrchol 2 PT1;
      }
      switch (číslo strany v PT2 pro hranu e) {
         // to samé jako c1 pro c2
         …
      }

      // vrcholy na stranách přilehlých c1 a c2
      int v3, v4;
      // proměnné pro hledání geodetické cesty
      int[] path1, path2;
      // první hledání geodetické cesty je mezi c1 a c2, resp. c2 a c1
      path1 = findGeodesic(soused1, c1, c2);
      path2 = findGeodesic(soused2, c2, c1);

      /* apriorní návrh bodu v3 a v4 */
      // v3 = path1[bod, ze kterého je viditelný c2 z pohledu c1];
      // v4 = path2[bod, ze kterého je viditelný c1 z pohledu c2];

      /* aposteriorní návrh bodu v3 a v4 */
      v3 = první, resp. poslední vrchol ze strany path1[1];
      v4 = první, resp. poslední vrchol ze strany path2[1];

      // v cyklu se bude vyhledávat nová cesta
      while (dokud dochází ke změnám ve v3 a v4) {
         path1 = findGeodesic(e.soused1, c1, v4);
         path2 = findGeodesic(e.soused2, c2, v3);
         v3 = path1[bod, ze kterého je viditelný c2 z pohledu c1];
         v4 = path2[bod, ze kterého je viditelný c1 z pohledu c2];
      }
      return nová hrana s vrcholy v3 a v4;
   }

   else {   // dvě společné hrany
      int c = roh polygonu, ze kterého budeme hledat vnitřní bod;
      int inPoint = společný bod hrany e a secondBorder;
      // nalezení geodetické cesty mezi rohem c a inPoint
      path = findGeodesic(pst, c, inPoint);
      // vytvoření nové hrany
      newBorder = new Edge (path[bod, ze kterého je viditelný inPoint
                                 z rohu c], inPoint);
      return newBorder;
   }
}
```

Algoritmus 3.4. Nalezení nové hranice zrychlenou metodou.

Jak bylo řečeno dříve, tento algoritmus nefunguje, pokud sousední PT nejsou minimální pseudo-triangulace. Při hledání cesty se to projeví tak, že body `v3` a `v4` se sejdou v jednom vrcholu. Pokud se tak stane, nemůže dojít k vytvoření nové hrany, algoritmus vrátí „chybový stav“ a je nutné zavolat metodu pro vyhledávání všech diagonál v polygonu (kapitola 3.2).

3.3 Kritéria pro výběr nové hrany

Algoritmy uvedené výše se zabývají nalezením nových hranic sousedních PT. Nyní budeme předpokládat, že už známe všechna možná rozdělení sousedních PT, resp. rozdělení polygonu vzniklého spojením sousedních na dva pseudo-trojúhelníky, a podíváme se na kritéria pro výběr toho nejlepšího rozdělení tak, aby výsledná pseudo-triangulace byla optimální z hlediska zadaného kritéria. Kritéria pro výběr nové hrany budou dále popsána.

V našem programu jsme vyzkoušeli následující kritéria pro výběr nové hrany mezi sousedními PT. Všechna níže uvedená kritéria kromě kritéria *MinMaxAngle* byla již zahrnuta v programu [Trčka07].

- **Random kritérium** – Nové hranice jsou vybrány náhodně. Nebude docházet ke swapu. Na první pohled nebude poskytovat žádané výsledky.
- **MinLength** – Nové hranice jsou nejkratší možné diagonály.
- **MaxLength** – Nové hranice jsou nejdelší možné diagonály[15].
- **MaxMinAngle** – Je vybráno takové rozdělení sousedních PT, kde nejmenší úhel v rozích sousedních PT je větší nebo roven nejmenšímu úhlu v ostatních rozděleních sousedních PT[16].
- **MinMaxAngle** – Je vybráno takové rozdělení sousedních PT, kde největší úhel v rozích sousedních PT je menší nebo roven nejmenšímu úhlu v ostatních rozděleních sousedních PT.
- **MaxMinSumAngle** – Je vybráno takové rozdělení sousedních PT, kde menší ze součtu úhlů v rozích sousedních PT je větší nebo roven menšímu ze součtu úhlů v rozích v ostatních rozděleních sousedních PT.
- **MinMaxSumAngle** - Je vybráno takové rozdělení sousedních PT, kde větší ze součtu úhlů v rozích sousedních PT je menší nebo roven většímu ze součtu úhlů v rozích v ostatních rozděleních sousedních PT.
- **MaxMinAngleHull** – Je stejné jako kritérium *MaxMinAngle* s tím rozdílem, že úhel se měří rozích konvexního obalu pseudo-trojúhelníku. Více o tomto a následujícím kritériu v kapitole 4.2.2.
- **MinMaxAngleHull** – Je stejné jako *MinMaxAngle* s tím rozdílem, že úhel se měří v konvexním obalu pseudo-trojúhelníku.

V příloze B Kritéria pro výběr hranice jsou ke shlédnutí inkrementální pseudo-triangulace vytvořené pomocí těchto kritérií. Jak je vidět z přiložených obrázků, pouze kritéria *MinLength* a *MaxMinAngle* vytvářejí přijatelné struktury.

3.4 Swap v inkrementálním algoritmu

Aplikování swapu jsme v našem programu prováděli dvěma cestami:

1) Realizace swapu jako tzv. *post-processing*, kdy se swap prováděl dodatečně na hotové pseudo-triangulaci. Tento způsob byl o něco jednodušší na implementaci, ale jako vstupní parametr musela být již hotová pseudo-triangulace. Při tomto způsobu není možné provádět swap pro tzv. *online přidávání bodů*, kdy lze postupně přidávat nové body a přitom pořád udržovat strukturu optimální z hlediska zadaného kritéria.

[15] Funguje přesně opačně, než si přejeme, tj. vybírá ty nejhubenější trojúhelníky. Je zde uvedeno pouze pro úplnost.
[16] Celkový počet rozdělení je v minimální pseudo-triangulaci 2 nebo 3.

2) Prohazování hran v inkrementálním algoritmu, kdy je prohazování hran součástí algoritmu konstrukce inkrementální pseudo-triangulace. Tento postup umožňuje tzv. *online přidávání bodů* a prohazování hran zároveň.

Implementace inkrementálního swapu

[Kolinger02] uvádí algoritmus inkrementálního vkládání bodů do Delaunayovy triangulace[17]. Swap v algoritmu inkrementálního vkládání bodů do pseudo-triangulace je navržen do značné míry podobně jako legalizace hran po vložení bodu do Delaunayovy triangulace.

Inkrementální algoritmus navržený v původní práci [Trčka07] jsme obohatili o prohazování hran po každém vložení nového bodu následujícím způsobem:

1. Vložení nového bodu algoritmem [Trčka07].

2. Uložení pseudo-trojúhelníku, do kterého byl nový bod vložen, do lokální proměnné `pst`. Pokud byl bod vložen na hranu, uloží se polygon vzniklý spojením sousedních PT s hranicí, na kterou byl nový bod vložen, do proměnné `oldPol`. Poslední možnost je, že byl bod vložen mimo pseudo-triangulaci. Potom se do proměnné `oldPartHull` uloží ta část obalu pseudo-triangulace, která se nacházela ve starém obalu a nenachází se v novém.

3. Pro každou hranu v `pst`, `oldPol` nebo `oldPartHull` se provede swap. Pokud se hrana prohodí, pokračujeme rekurzivně v prohazování.

Algoritmus inkrementálního prohazování hran zapsaný v alg. 3.5 pracuje následujícím způsobem:

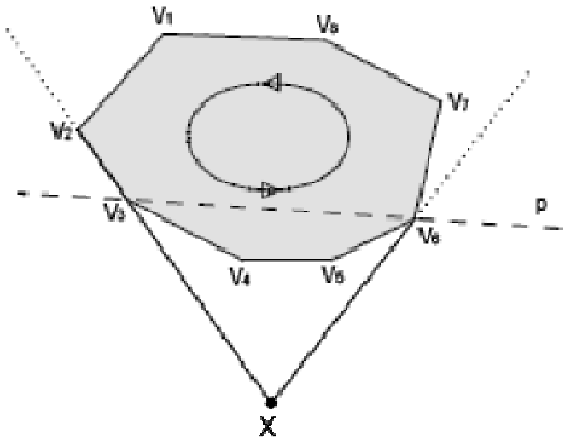
Nechť je vkládaný bod lokalizován uvnitř pseudo-trojúhelníku. Označme tento pseudo-trojúhelník `pst`. Trojúhelník `pst` bude v inkrementálním algoritmu v programu [Trčka07] rozdělen na dva nové pseudo-trojúhelníky podle zadaného kritéria. Původní pseudo-trojúhelník `pst` bude vyřazen ze struktury, ale ještě nebude smazán. Z `pst` se získá pole hran `polygon`, jež `pst` tvoří. Každá hrana `e` z `polygonu` bude testována `flip(e)`, zda ji je možné prohodit, popř. i prohozena. Pokud se tak stane (`flip(e) == true`), do `polygonu` v metodě `recursiveSwap()` se uloží pole hran sousedních PT pro hranu `e` (`getPolygon(e)` z kapitoly 3.1). Dál se budou rekurzivně znovu testovat všechny hrany v `polygonu`.

Nechť je vkládaný bod lokalizován na hraně pseudo-trojúhelníku *edge_localized*. Do proměnné `oldPol` se uloží polygon vzniklý spojením sousedních PT se společnou hranicí *edge_localized*. Do proměnné `polygon` se uloží `oldPol`. Dál se provádí rekurzivně swap jako v předešlém odstavci.

Nechť je vkládaný bod lokalizován mimo pseudo-triangulaci. Po přidání bodu do pseudo-triangulace se změní konvexní obal struktury, viz kapitol a 5.2.3 z práce [Trčka07]. Do proměnné `oldPartHull` se uloží ta část obalu, která bude po vložení bodu vypuštěna (viz následující odstavec). Do proměnné `polygon` se uloží `oldPartHull` a dál bude rekurzivní algoritmus probíhat jako v odstavci výše.

[17] V Delaunayově triangulaci se provádí prohazování hran v inkrementálním algoritmu.

Na obrázku 3.20 je ukázka „výměny“ části konvexního obalu – hran V_3V_4, V_4V_5 *a* V_5V_6 po vložení bodu X za hrany V_3X a XV_6. Obrázek byl převzat z práce [Trčka07]. V rekurzivním prohazování budou vypuštěné hrany uloženy do proměnné `oldPartHull`.



Obrázek 3.21. Ukázka úpravy konvexního obalu pseudo-triangulace po vložení bodu vně pseudo-triangulace. [Trčka07]

Na závěr této kapitoly je uveden algoritmus rekurzivního prohazovaní hran zapsaný pseudo-kódem jako algoritmus 3.5:

```
/*
 * Metoda provádí prohazování hran v pseudo-triangulaci po každém vložení
 * nového bodu.
 *
 * position = příznak, kam byl bod vložen: dovnitř pseudo-trojúhelníku,
 *            na hranu nebo mimo pseudo-triangulaci
 * criterium = příznak kritéria, podle kterého se bude provádět swap
 * oldPartHull = pokud byl bod vložen mimo pseudo-triangulaci, bude zde
 *               uložena část obalu, která byla přidáním nového bodu
 *               vypuštěna.
 * oldPol = pokud byl bod vložen na hranu, bude zde polygon vrácený metodou
 *          getPolygon(hrana, na níž byl nový bod vložen)
 * pst = pokud byl bod vložen do pseudo-trojúhelníku, je zde uložen ten
 *       pseudo-trojúhelník
 *
 * vrací počet flipů
 */
int incrementalInseriotnFlip(int position, int criterium, LinkedList<Edge>
                             oldPartHull, Edge[] oldPol, PsTriangle pst) {
   int count = 0;       // počet provedených swapů

   // přepínač podle pozice, kam byl nový vrchol vložen
   switch (position) {
```

```
   case IN_PSTRIANGLE:   // bod přidán dovnitř nějakého pseudo-trojúhelníku
      // pole hran, které budou prohazovány
      polygon = getPolygonFromPsTriangle(pst);
      for (e : polygon) {
         /*
          * e již mohla být v některém z předešlých rekurzivních flipů
          * odstraněna ze struktury
          */
         if (e již byla odebrána ze struktury)
            continue;

         else  // volání rekurzivního flipu
            count += recursiveSwap(e, criterium);
      }
      break;

   case ON_EDGE:        // bod byl přidán na hranu
      // pole hran, které budou prohazovány
      polygon = oldPol;
      for (e : polygon1) {
         stejný cyklus jako v předešlém případě
       }
       break;
   case OUTSIDE_HULL:         // bod byl přidán vně pseudo-triangulace
      // pole hran, které budou prohazovány
      polygon = oldPartHull;
      for (e : polygon) {
         stejný cyklus jako v prvním případě
      }
      break;
   }
   return count;
}


/*
 * rekurzivní flip
 * vrací počet flipů
 * e – hrana, která má být prohozena
 * criterium – kriterium, podle kterého se bude prohazovat
 */
int recursiveSwap(Edge e, int criterium) {
   int count;   // počet provedených flipů
   polygon = getPolygon(e);   // hrany, jež budou dál prohazovány

   /*
    * Pokud se hrana e prohodí, budou se prohazovat i hrany z polygonu
    * deklarovaného výše.
    */
   if (flip(e) == true) {   // e byla prohozena
      // budou se rekurzivně prohazovat hrany z polygonu
      for (edge : polygon) {
         if (e již byla odebrána ze struktury)
            continue;
         else
            count += recursiveSwap(e);

      }
   }
   return count;
}
```

Algoritmus 3.5. Rekurzivní prohazování hran v konstrukci pseudo-triangulace inkrementálním vkládáním.

4 Testy

4.1 Časová složitost prohazování hran

V této kapitole se budeme zabývat testováním algoritmu prohazování hran se zaměřením na časovou složitost. Budeme se snažit ukázat, že se algoritmus chová na různých množinách stejně (z hlediska spotřeby času). Dále porovnáme swap začleněný do inkrementálního algoritmu se swapem dodatečně provedeným na již hotové pseudo-triangulaci. Nakonec se podíváme, o kolik je rychlejší urychlený swap z kapitoly 3.2 oproti tzv. brute-force swapu z kapitoly 3.1.

Veškeré uvedené testy byly prováděny na počítači AMD Sempron 1.6 GHz, 512 MB, Windows XP SP3.

4.1.1 Časová složitost na různých množinách

Nyní budeme porovnávat časy potřebné na vytvoření inkrementální pseudo-triangulace na různých množinách pro různá kritéria. Budeme se snažit ukázat, že algoritmus prohazování hran se chová (z pohledu spotřeby času) na různých množinách stejně a že časová složitost je O(N logN), kde N je počet vrcholů ve struktuře. Jak se liší časy pro inkrementální swap a tzv. post-processing swap bude uvedeno na konci této kapitoly. Porovnání časů tzv. brute-force swapu a urychleného prohazování budeme zkoumat v následující kapitole.

Test bude prováděn na množinách s rozložením bodů: uniform – rovnoměrné, gaussovské, cluster – shluky bodů a arc – body rozmístěny do oblouku. Počet bodů v množinách: 1000, 5000, 10000, 30000, 50000 a 75000. Každé měření bude provedeno na třech různých množinách třikrát, tzn. jedno číslo v tabulce je průměr devíti hodnot. Veškerá vstupní data byla dodána vedoucí práce.

Vstupem bude pro všechny testy množina bodů a výstupem potom pseudo-triangulace, v níž bude proveden swap:

1) Swap je začleněn v inkrementálním algoritmu.
2) Dodatečné prohazování hran. Nejdříve se vytvoří pseudo-triangulace původním algoritmem [Trčka07], pak se provede swap. Čas uvedený v tabulkách níže bude tedy součtem dvou časů.

Testována budou kritéria: *MinLength*, *MaxMinAngle* a *MinMaxAngle*. Tato kritéria byla zvolena pozorováním. Zbylá kritéria uvedená v kapitole 3.3 neposkytovala už na první pohled optimální pseudo-trojúhelníky, viz příloha B.

Popis tabulek :

- V levém sloupci se nachází počet vrcholů v množině – *N*.
- Další sloupce zaznamenávají časy potřebné na vytvoření inkrementální pseudo-triangulace na různých množinách. Jednotky jsou milisekundy.

Jako první byl testován inkrementální swap v pseudo-triangulaci:

Z tabulek 4.1, 4.4, 4.7 a k nim přiložených grafů je vidět, že algoritmus prohazování hran se chová velmi podobně na množinách *uniform, gaus* a *cluster*. Na množině bodů *arc* dochází k výrazně větší spotřebě času. Dokonce jsem na této množině neprováděl testy pro 50000 a 75000 z důvodu enormní časové náročnosti.

Anomálie na množině *arc* je pravděpodobně způsobena pravidelností v rozložení bodů. Tato pravidelnost má přímé negativní následky. Body jsou rozloženy v půlkruhu relativně blízko u sebe. Při testu polohy bodu vůči přímce, viz kapitola 2.4.2, bude nutné často počítat s čísly typu *java.math.BigDecimal* V práci [Trčka07], příloze A se uvádí, že počítání s čísly *BigDecimal* je více jak 70krát pomalejší než počítání s čísly *double*[18]. Tento test se používá v algoritmu náhodné procházky nebo v algoritmu přidání bodu dovnitř pseudo-trojúhelníku v původním programu [Trčka07] a v testech na křížení hran nebo testu na vnitřní diagonálu v našem programu[19]. Dalším problémem by mohl být tvar pseudo-trojúhelníků. Pozorováním bylo zjištěno, že např. pro rozložení bodů *arc_10000* se průměrná hodnota min. úhlu pohybuje kolem 0,01° a max. úhlu kolem 150° a více pro kritérium *MinLength* (pro ostatní kritéria nevycházejí průměrné úhly o mnoho lépe). Více o úhlových testech v následující kapitole.

V tabulkách 4.2, 4.5 a 4.8 je uveden odhad časové složitosti pro inkrementální pseudo-triangulaci s prohazováním hran začleněným přímo v algoritmu konstrukce. Hodnoty v tabulkách jsou: *čas / f(N)* (*f(N)* je zadaná v názvu sloupce, *čas* je uveden v tab. 4.1, 4.4 a 4.7 ve sloupci *uniform*). Výpočet byl tedy proveden pouze pro rozdělení *uniform*. Vidíme, že tento algoritmus bude „mít nejblíže“ ke složitosti **O(N logN)**. Z přiložených grafů vyplývá, že pro rozložení bodů *gaus* a *cluster* bude časová složitost algoritmu stejná.

MinLength, inkrementální swap, zrychlené prohazování [ms]				
N	**uniform**	**gaus**	**cluster**	**arc**
1000	1364	1180	1266	12740
5000	7250	6418	6743	164001
10000	14617	13898	14972	67394
30000	45014	46120	45371	529006
50000	75854	78223	76490	
75000	115120	126993	122357	

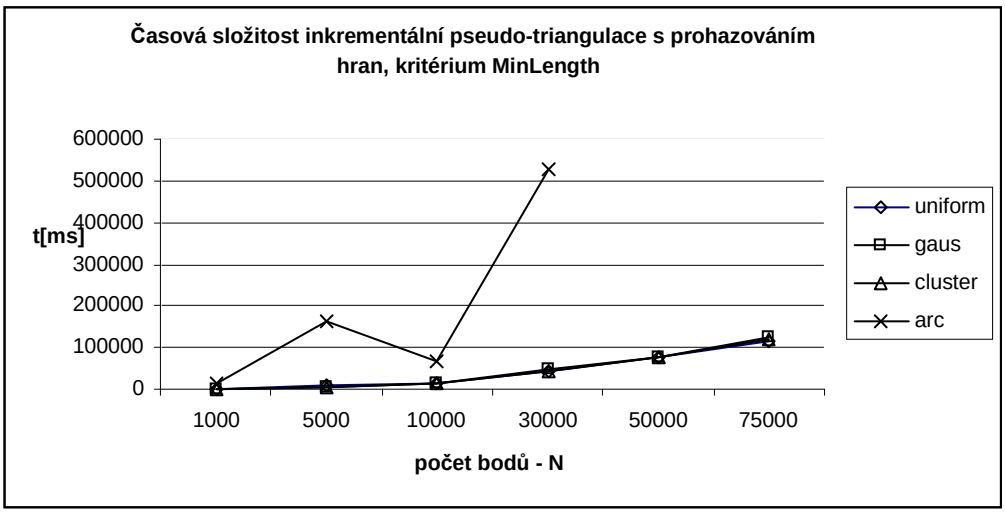
Tabulka 4.1.Tabulka pro určení časové složitosti pro kritérium MinLength.

složitost, uniform, MinLength			
N	**O(N)**	**O(N logN)**	**O(N^4/3)**
1000	1,364	0,4547	0,1364
5000	1,45	0,392	0,0848
10000	1,4617	0,3654	0,0678
30000	1,5005	0,3351	0,0483
50000	1,4574	0,3101	0,0396
75000	1,5349	0,3149	0,0364

Tabulka 4.2. Odhad asymptotické složitosti pro rozložení bodů uniform z tabulky 4.1.

[18] Počítač, na kterém byly prováděny testy v práci [Trčka07], byl výkonnostně podobný mému počítači. Proto bych očekával, že údaj „70krát pomalejší počítání testu na polohu bodu vůči přímce s čísly typu BigDecimal“ bude platit i v našich testech.

[19] Toto bych viděl asi jako hlavní důvod zpomalení algoritmu.



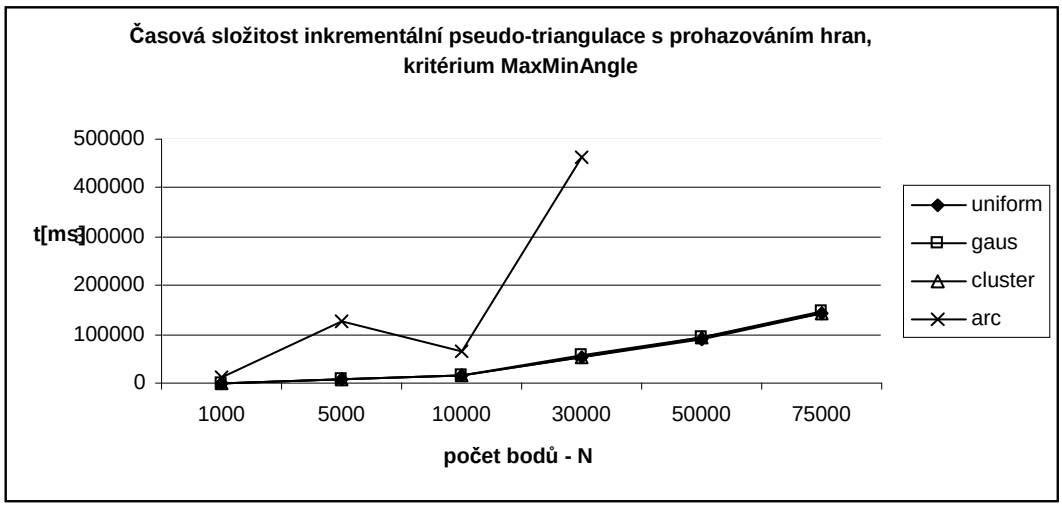
Graf 4.3. Grafické znázornění tabulky 4.1.

MaxMinAngle, inkrementální swap, zrychlené prohazování [ms]				
N	**uniform**	**gaus**	**cluster**	**arc**
1000	1767	1476	1550	12456
5000	8809	8044	8543	125235
10000	17782	17463	16843	65546
30000	54354	56516	54898	461658
50000	91504	94166	93573	
75000	142694	148667	142949	

Tabulka 4.4. Tabulka pro určení časové složitosti pro kritérium MaxMinAngle.

složitost, uniform, MaxMinAngle			
N	**O(N)**	**O(N logN)**	**O(N^4/3)**
1000	1,767	0,589	0,1767
5000	1,7618	0,4763	0,103
10000	1,7782	0,4446	0,0825
30000	1,7816	0,3979	0,0573
50000	1,8101	0,3852	0,0491
75000	1,8755	0,3847	0,0445

Tabulka 4.5. Odhad asymptotické složitosti pro rozložení bodů uniform z tabulky 4.4.



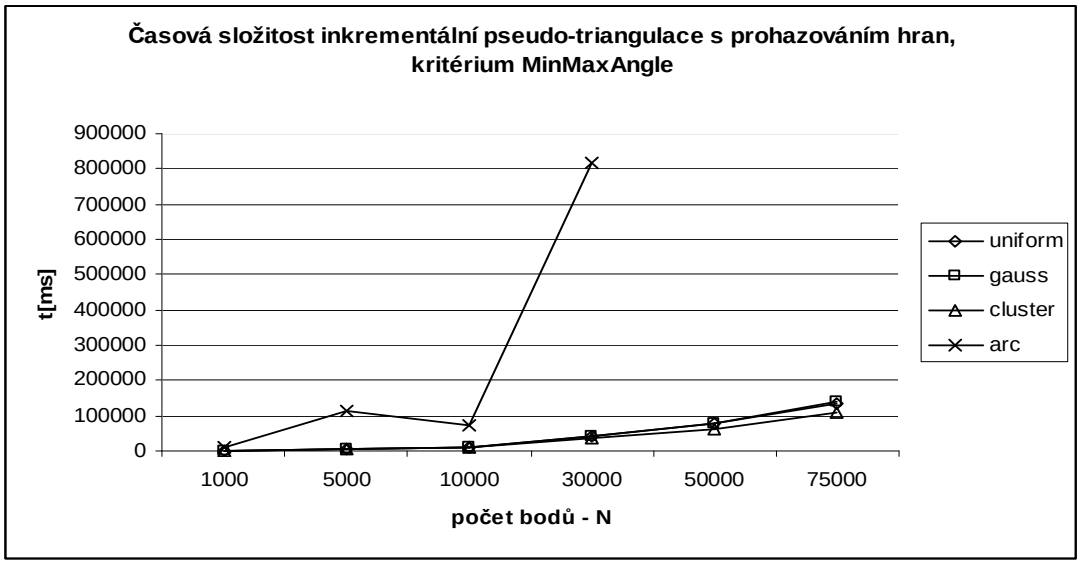
Graf 4.6. Grafické znázornění tabulky 4.4.

MinMaxAngle, inkrementální swap, zrychlené prohazování				
N	**uniform**	**gaus**	**cluster**	**arc**
1000	1119	961	1111	10520
5000	5638	5121	5362	115123
10000	11949	11619	10799	71979
30000	42415	41757	36898	816554
50000	79807	78577	60381	
75000	133694	138561	106867	

Tabulka 4.7. Tabulka pro určení časové složitosti pro kritérium MinMaxAngle.

složitost, uniform, MinMaxAngle			
N	**O(N)**	**O(N logN)**	**O(N^4/3)**
1000	1,119	0,373	0,1119
5000	1,1276	0,3048	0,0659
10000	1,1949	0,2987	0,0555
30000	1,3916	0,3108	0,0448
50000	1,5767	0,3356	0,0428
75000	1,7722	0,3635	0,042

Tabulka 4.8. Odhad asymptotické složitosti pro rozložení uniform z tabulky 4.7.



Graf 4.9. Grafické znázornění tabulky 4.7.

Stejná měření byla provedena i pro dodatečné prohazování hran:

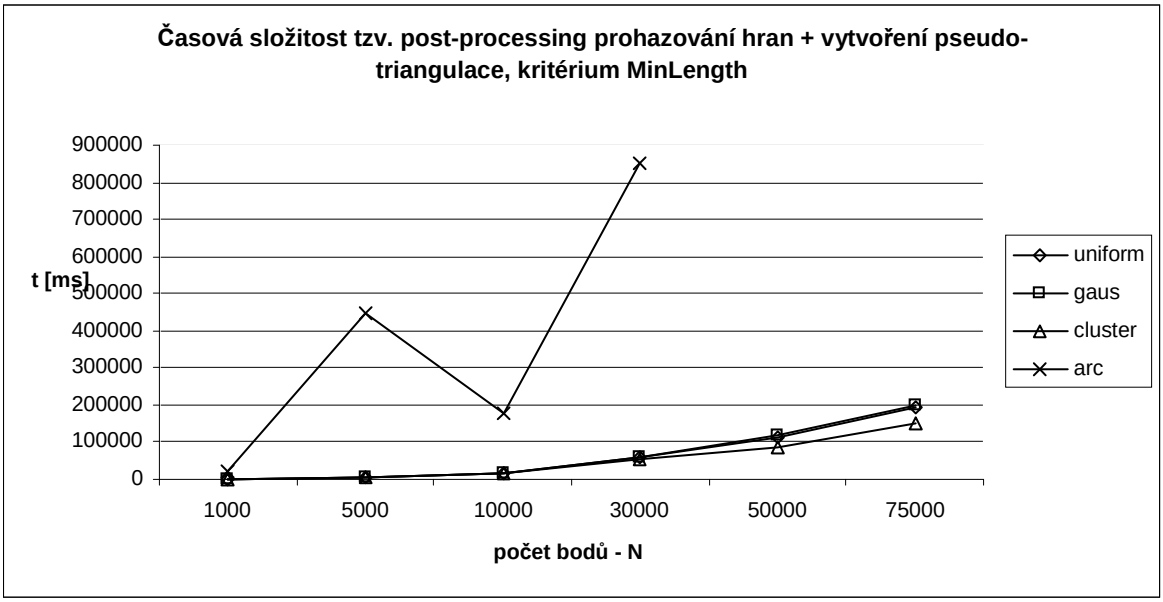
Časy v tabulkách 4.10, 4.13 a 4.16 jsou výsledky testů pro měření časové složitosti dodatečného prohazování hran. Jednotlivé hodnoty se skládají z času potřebného k vytvoření pseudo-triangulace inkrementálním vkládáním původním algoritmem [Trčka07] a z času potřebného k dodatečnému prohazování hran. Jako u inkrementálního algoritmu je z grafů 4.12, 4.15 a 4.18 patrné, že na množinách *uniform*, *gaus* a *cluster* se algoritmus dodatečného prohazování hran chová velmi podobně. Naopak na množině *arc* dochází k exponenciálnímu nárůstu spotřeby času. Asymptotická složitost byla stejně jako u inkrementálního prohazování vypočtena pouze pro rozložení *uniform*. Výsledky jsou uvedeny v tabulkách 4.11, 4.14 a 4.17. Je vidět, že asymptotická složitost je také nejblíže **O(N logN)**, ale je o trochu horší než pro inkrementální swap.

MinLength, post-processing, zrychlené prohazování				
N	**uniform**	**gaus**	**cluster**	**arc**
1000	1269	1252	1046	21818
5000	7527	7163	5771	447743
10000	17510	16528	15244	179125
30000	60863	60015	51895	853289
50000	115565	120998	85902	
75000	196468	200186	153286	

Tabulka 4.10. Výsledky testů pro tzv. post-processing swap. MinLength kritérium.

složitost, uniform, MinLength			
N	**O(N)**	**O(N logN)**	**O(N^4/3)**
1000	1,269	0,423	0,1269
5000	1,5054	0,407	0,088
10000	1,751	0,4378	0,0813
30000	2,0288	0,4531	0,0653
50000	2,3113	0,4919	0,0627
75000	2,6196	0,5373	0,0621

Tabulka 4.11. Odhad asymptotické složitosti pro rozložení uniform z tabulky 4.10.



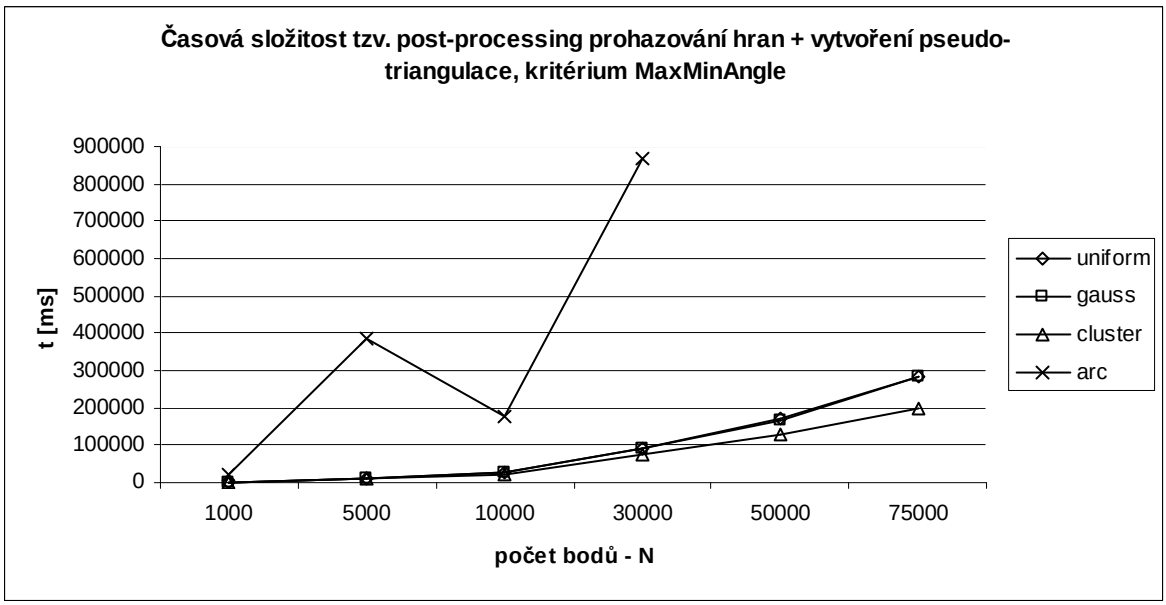
Graf 4.12. Grafické znázornění tabulky 4.10.

MaxMinAngle, post-processing, zrychlené prohazování				
N	**uniform**	**gaus**	**cluster**	**arc**
1000	1582	1642	1378	21337
5000	11243	11453	8607	388252
10000	28772	24914	21718	179281
30000	92222	90296	75699	867561
50000	168943	164073	130663	
75000	283174	283068	199546	

Tabulka 4.13. Výsledky testů pro tzv. post-processing swap.MaxMinAngle kritérium.

složitost, uniform, MaxMinAngle			
N	**O(N)**	**O(N logN)**	**O(N^4/3)**
1000	1,582	0,5273	0,1582
5000	2,2486	0,6079	0,1315
10000	2,8772	0,7193	0,1335
30000	3,0741	0,6866	0,0989
50000	3,3789	0,7191	0,0917
75000	3,7757	0,7745	0,0895

Tabulka 4.14. Odhad asymptotické složitosti pro rozložení uniform z tabulky 4.13.



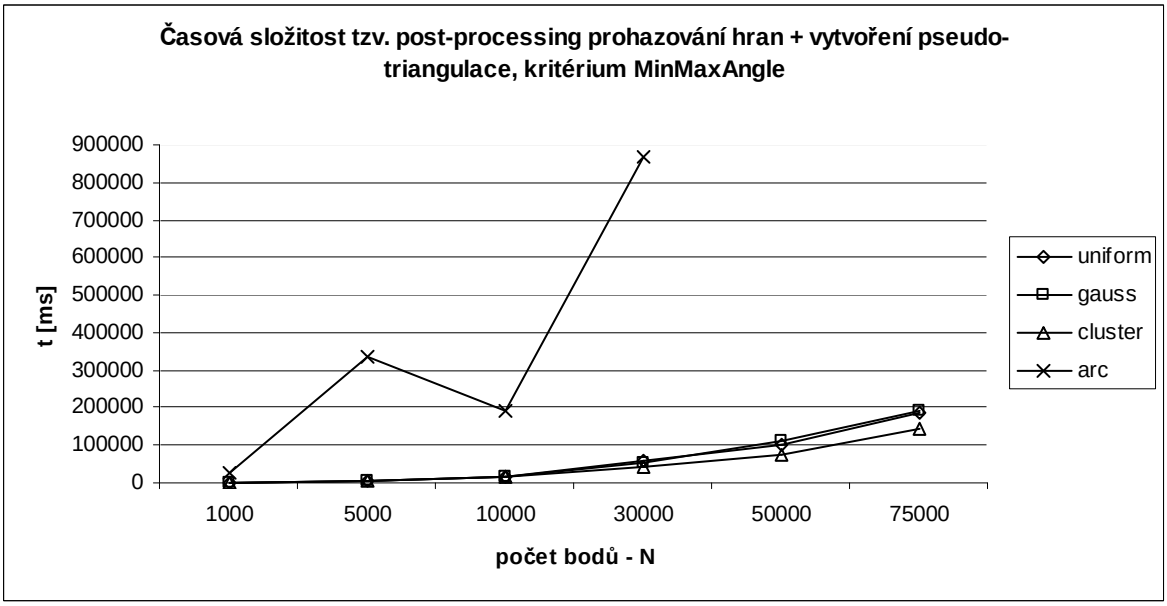
Graf 4.15. Grafické znázornění tabulky 4.13.

MinMaxAngle, post-processing, zrychlené prohazování				
N	**uniform**	**gaus**	**cluster**	**arc**
1000	1157	1184	1150	25337
5000	6805	7077	7222	337371
10000	15538	15559	13774	192741
30000	57555	54595	43170	868747
50000	99495	111079	76779	
75000	187651	192540	143850	

Tabulka 4.16. Výsledky testů pro tzv. post-processing swap.MinMaxAngle kritérium.

složitost, uniform, MinMaxAngle			
N	**O(N)**	**O(N logN)**	**O(N^4/3)**
1000	1,157	0,3857	0,1157
5000	1,361	0,3679	0,0796
10000	1,5538	0,3885	0,0721
30000	1,9185	0,4285	0,0617
50000	1,9899	0,4235	0,054
75000	2,502	0,5132	0,0593

Tabulka 4.17. Odhad asymptotické složitosti pro rozložení uniform z tabulky 4.16.



Graf 4.18. Grafické znázornění tabulky 4.16.

Na závěr je uvedeno porovnání časů potřebných pro vytvoření pseudo-triangulace s použitím inkrementálního swapu a post-processing swapu. V tabulkách 4.19, 4.20 a 4.21 je v procentech vyjádřeno o kolik je konstrukce pseudo-triangulace s použitím inkrementálního prohazování hran rychlejší než konstrukce pseudo-triangulace s post-processing swapem (základ je čas post-processing swapu). Z tabulek je patrné, že konstrukce s inkrementálním swapem byla ve většině případů na testovaných množinách až o desítky procent rychlejší než konstrukce s dodatečným prohazováním hran[20].

MinLength [%]				
N	**uniform**	**gaus**	**cluster**	**arc**
1000	-7	6	-21	42
5000	4	10	-17	68
10000	17	16	2	63
30000	26	25	13	53
50000	37	36	11	
75000	41	39	26	

Tabulka 4.19. Porovnání post-processing swapu s inkrementálním prohazováním. MinLength kritérium.

[20] Toto poslední měření berme pouze orientačně. Jak alg. post-processing swapu, tak alg. inkrementálního swapu, resp. rekurzivního swapu, lze ještě optimalizovat. Spokojme se zatím s tvrzením, že inkrementální swap je rychlejší než dodatečné prohazování.

MaxMinAngle [%]				
N	**uniform**	**gaus**	**cluster**	**arc**
1000	-12	10	-12	42
5000	22	30	1	69
10000	38	30	22	64
30000	42	40	27	54
50000	46	43	28	
75000	50	49	30	

Tabulka 4.20. Porovnání post-processing swapu s inkrementálním prohazováním. MaxMinAngle kritérium.

MinMaxAngle [%]				
N	**uniform**	**gaus**	**cluster**	**arc**
1000	3	19	3	58
5000	17	28	26	68
10000	23	25	22	64
30000	27	24	15	17
50000	21	30	21	
75000	29	29	28	

Tabulka 4.21. Porovnání post-processing swapu s inkrementálním prohazováním. MinMaxAngle kritérium.

4.1.2 Brute-force prohazování vs. zrychlené prohazování

V této kapitole porovnáme zrychlený swap z kapitoly 3.2 a původní tzv. brute-force swap z kapitoly 3.1. Stejně jako v předchozích testech, porovnávané časy zahrnují vytvoření pseudo-triangulace s prohazováním hran buď jako post-processing nebo začleněným přímo v inkrementálním algoritmu.

Pro následující měření bude použito pouze rovnoměrné rozložení bodů – *uniform*. Jak se v minulé kapitole ukázalo, swap se chová stejně na množinách s různým rozložením bodů, proto můžeme toto měření prohlásit za platné i na ostatních množinách (kromě rozložení *arc*).

Kritéria pro výběr nových hran budou použita stejná jako v minulé kapitole, tj. *MinLength*, *MaxMinAngle* a *MinMaxAngle*.

Test bude proveden pro inkrementální swap i pro tzv. post-processing swap.

Tabulka 4.16 zaznamenává výsledky testů pro inkrementální swap. Vlevo je zaznamenán počet bodů v množině – N. Dále jsou v tabulce uvedeny výsledky měření – časy v milisekundách pro výše uvedené kritéria. V levé části byl použit algoritmus tzv. brute force swap, v pravé části tabulky pak zrychlený swap – tyto výsledky jsou převzaté z minulé kapitoly. Tabulka 4.17 obsahuje výsledky pro tzv. post-processing prohazování hran.

Inkrementální swap						
	brute-force swap			zrychlený swap		
N	MinLength	MaxMinAngle	MinMaxAngle	MinLength	MaxMinAngle	MinMaxAngle
1000	2312	2280	1283	1364	1767	1119
5000	12953	12805	6563	7250	8809	5638
10000	27056	28582	14995	14617	17782	11949
30000	88562	88086	54221	45014	53448	41748
50000	154587	149766	99124	72869	90504	78837
75000	231471	232417	173106	115120	140662	132913

Tabulka 4.16. Výsledky měření urychlení swapu pro inkrementální swap.

Post-processing						
	brute-force swap			zrychlený swap		
N	MinLength	MaxMinAngle	MinMaxAngle	MinLength	MaxMinAngle	MinMaxAngle
1000	2392	2861	1774	1269	1582	1157
5000	12821	16584	8496	7527	11243	6805
10000	31715	45196	21516	17510	28772	15538
30000	96333	128627	75157	60863	92222	57555
50000	197598	243760	128669	115565	168943	99495
75000	311266	399189	225847	196468	283174	187651

Tabulka 4.17. Výsledky měření urychlení swapu pro post-processing swap.

V tabulce 4.18 je uvedeno procentuální urychlení zrychleného swapu proti tzv. brute-force swapu (základ je čas brute-force swapu) pro inkrementální prohazování hran. V tabulce 4.19 je uvedeno taktéž zrychlení swapu, ale při tzv. post-processing swapu. Z těchto dvou tabulek se ukázalo, že na testované množině poskytuje zrychlený swap lepší (rychlejší) výsledky řádově o desítky procent.

Inkrementální swap [%]			
N	MinLength	MaxMinAngle	MinMaxAngle
1000	41	23	13
5000	44	31	14
10000	46	38	20
30000	49	39	23
50000	53	40	20
75000	50	39	23

Tabulka 4.18. Procentuální vyjádření zrychlení swap pro inkrementální prohazování hran – tabulka 4.16.

Post-processing [%]			
N	MinLength	MaxMinAngle	MinMaxAngle
1000	47	45	35
5000	41	32	20
10000	45	36	28
30000	37	28	23
50000	42	31	23
75000	37	29	17

Tabulka 4.20. Procentuální vyjádření zrychlení swapu pro post-processing prohazování hran – tabulka 4.17.

4.2 Kvalita pseudo-triangulace

V minulých kapitolách jsme používali výraz optimální pseudo-triangulace, aniž bychom jakýmkoli způsobem definovali tento pojem. Velice jednoduchá definice se nachází v práci [Trčka07]. Po konzultaci s vedoucí práce jsme zvolili jako měřítko optimality pseudo-trojúhelníků test porovnávající vnitřní úhly v rozích pseudo-trojúhelníků[21].

V této kapitole budou testovány následující geometrické vlastnosti pseudo-trojúhelníků:

Maximální úhel pseudo-trojúhelníku je největší z vnitřních úhlů počítaných v rozích pseudo-trojúhelníku. Může nabývat hodnot z intervalu (0^o;180^o), na rozdíl od maximálního vnitřního úhlu v trojúhelníku, který nabývá hodnot z intervalu <60^o;180^o).

Minimální úhel pseudo-trojúhelníku je nejmenší z vnitřních úhlů počítaných v rozích pseudo-trojúhelníku. Nabývá hodnot z intervalu (0^o;60^o>, stejně jako minimální vnitřní úhel v trojúhelníku.

Optimalitu pseudo-triangulace ověřuje počítání úhlů v rozích pseudo-trojúhelníků. Kvalita pseudo-trojúhelníku stoupá, pokud se zvětšuje jeho minimální úhel, resp. klesá jeho maximální úhel.

Pro měření byla vybrána jedna konkrétní množina – *uniform* rozložení 10000 bodů.

Testovat se budou kritéria z předcházejících kapitol. Pro každé kritérium bude provedeno měření:

1) Pro pseudo-triangulaci vytvořenou původním inkrementálním algoritmem bez swapu [Trčka07].
2) Pro dodatečné prohazování hran na pseudo-triangulaci z bodu 1).
3) Pro pseudo-triangulaci vytvořenou inkrementálním vkládáním s prohazováním hran.

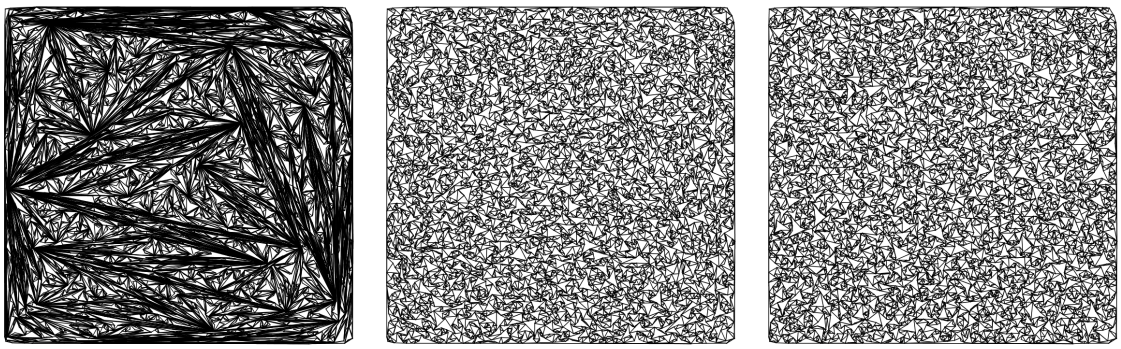
Výsledky testů se skládají z částí:

1) Histogram četností minimálních úhlů v pseudo-triangulaci[22].
2) Histogram četností maximálních úhlů v pseudo-triangulaci.
3) Graf výskytu kombinace maximálního a minimálního úhlu v pseudo-triangulaci.

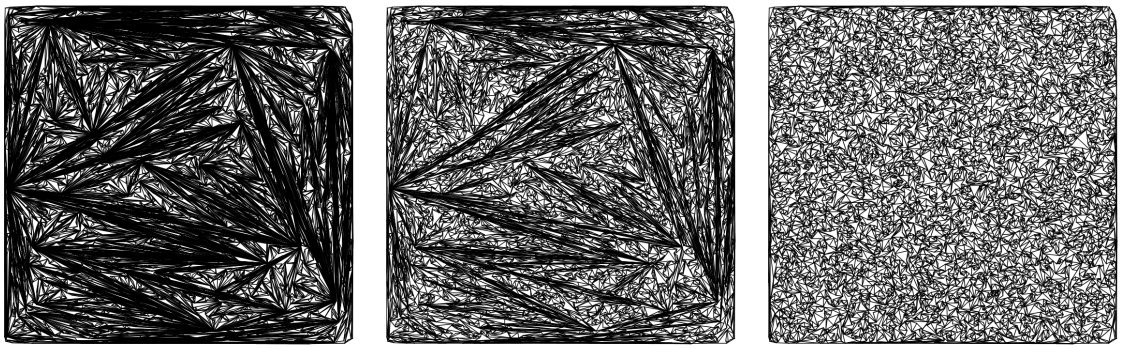
Obrázek 4.1 ukazuje pseudo-triangulace vytvořené pomocí kritéria *MinLength*. První zleva je pseudo-triangulace vytvořená v původním programu [Trčka07], druhá je pseudo-triangulace s dodatečným prohazováním a třetí je pseudo-triangulace se swapem začleněným přímo v inkrementálním algoritmu. Obrázek 4.2 je stejná posloupnost obrázků pro kritérium *MaxMinAngle* a obrázek 4.3 pro kritérium *MinMaxAngle*.

Na první pohled lze z obrázků vyčíst, že kritérium *MinLength* bude poskytovat optimální výsledky pro dodatečné prohazování hran i pro inkrementální prohazování. Kritérium *MaxMinAngle* bude poskytovat optimální výsledky jen pro inkrementální swap a kritérium *MinMaxAngle* nebude poskytovat optimální výsledky vůbec.

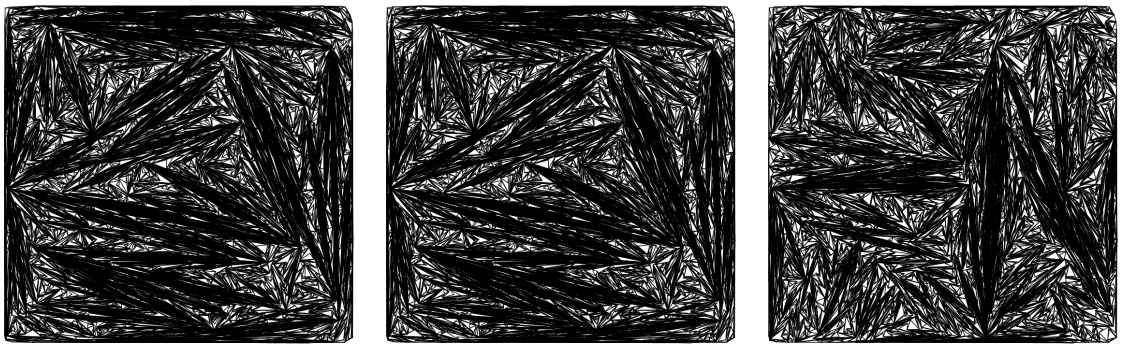
[21] Původně se tímto testem měřila kvalita triangulací. Ale i na pseudo-trojúhelník se můžeme dívat jako na trojúhelník, vezmeme-li pouze jeho konvexní obal.
[22] Tj. kolikrát se minimální úhel vyskytl v pseudo-triangulaci. Na vodorovné ose jsou hodnoty úhlu ve stupních, na svislé ose počet výskytů. U maximálních úhlů to samé.



Obrázek 4.1. Ukázka pseudo-triangulací pro kritérium MinLength. Původní inkrementální pseudo-triangulace [Trčka07], swap jako post-processing a inkrementální swap.



Obrázek 4.2. Ukázka pseudo-triangulací pro kritérium MaxMinAngle. Původní inkrementální pseudo-triangulace [Trčka07], swap jako post-processing a inkrementální swap.



Obrázek 4.3. Ukázka pseudo-triangulací pro kritérium MinMaxAngle. Původní inkrementální pseudo-triangulace [Trčka07], swap jako post-processing a inkrementální swap.

Vyhodnocení testů kvality pro kritérium **MinLength**:

Grafy 4.4, 4.7 a 4.10 – původní algoritmus [Trčka07], post-processing swap a inkrementální swap.

Hodnota průměrného minimálního úhlu se zvýšila z 5° na 22° pro dodatečné prohazování i pro inkrementální swap a hodnota maximálního úhlu se snížila ze 121° na 89° pro dodatečné prohazování hran a na 90° pro inkrementální swap. Můžeme tvrdit, že kritérium *MinLength* funguje a dodatečné prohazování i inkrementální swap poskytují kvalitativně velmi podobné pseudo-trojúhelníky.

Vyhodnocení testů kvality pro kritérium **MaxMinAngle**:

Grafy 4.5, 4.8 a 4.11 – původní algoritmus [Trčka07], post-processing swap a inkrementální swap.

Hodnota průměrného minimálního úhlu se zvýšila z 3° na 12° pro dodatečné prohazování a na 22 ° pro inkrementální swap. Hodnota maximálního úhlu se snížila ze 111° na 99° pro dodatečné prohazování hran a na 81° pro inkrementální swap. Můžeme tvrdit, že kritérium *MaxMinAngle* funguje a inkrementální swap poskytuje kvalitativně nejlepší pseudo-trojúhelníky[23].

Vyhodnocení testů kvality pro kritérium **MinMaxAngle**:

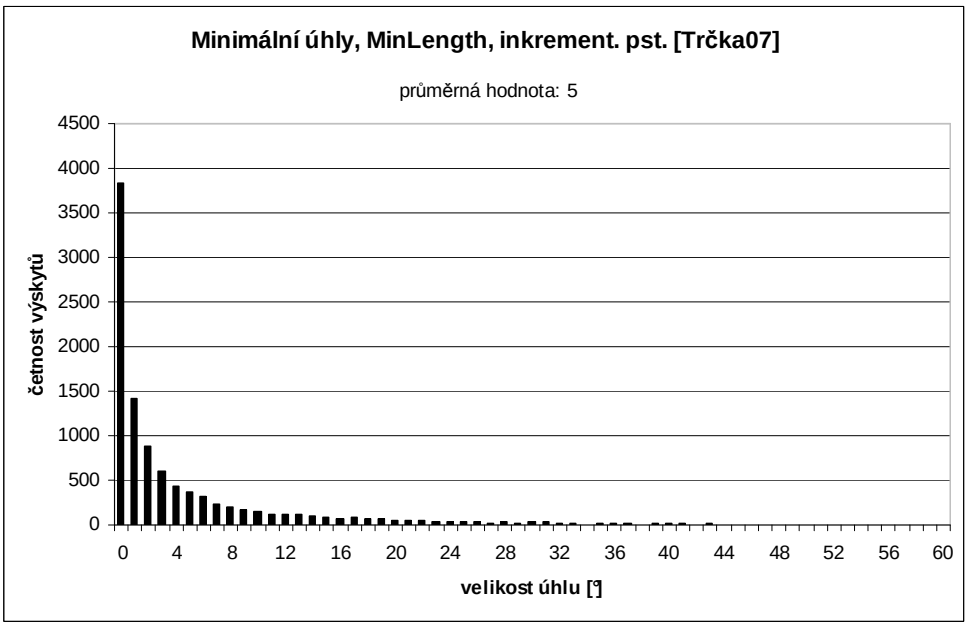
Grafy 4.6, 4.9 a 4.12 – původní algoritmus [Trčka07], post-processing swap a inkrementální swap.

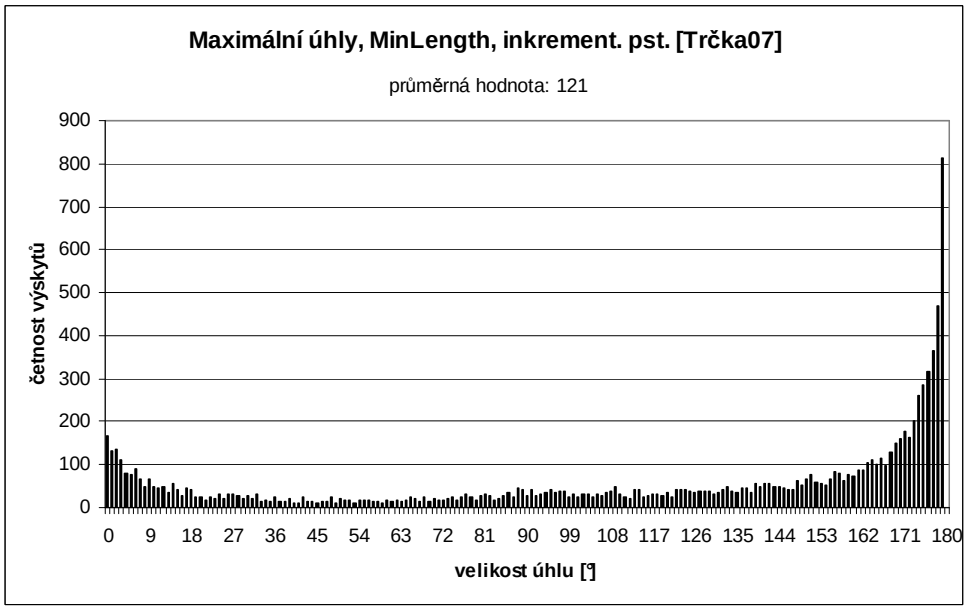
Jak již obrázek 4.3 ukázal, dalo se předpokládat, že toto kritérium nebude poskytovat optimální pseudo-triangulaci. Tento předpoklad se měřením potvrdil. Průměrný minimální úhel byl pro všechny měření stejný: 4°. Průměrný maximální úhel pro původní algoritmus [Trčka07], dodatečné prohazování a inkrementální swap vyšel postupně: 39°, 34°, 32°.

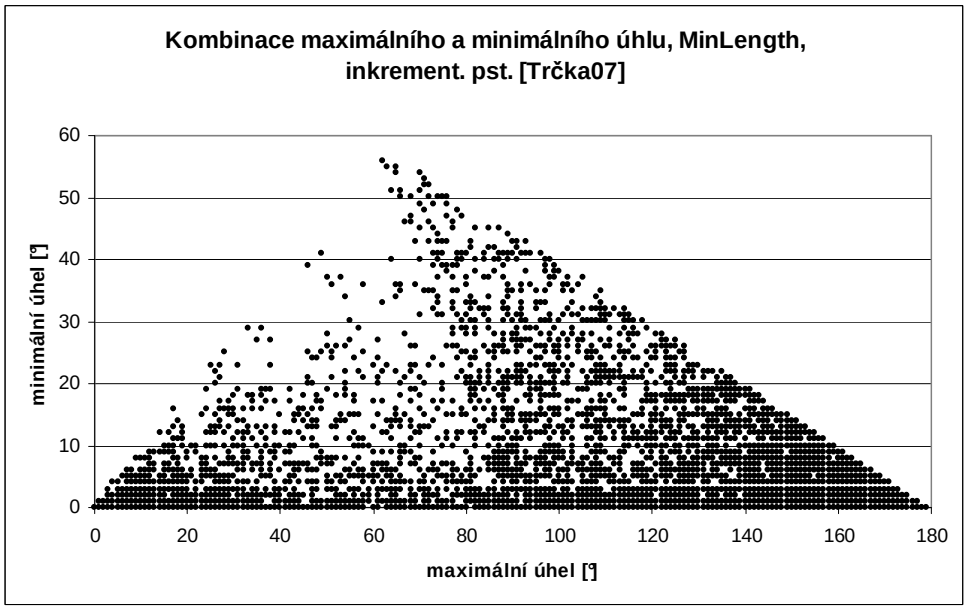
Z výsledků měření je vidět, že nedochází ke zvětšování minimálního úhlu. Maximální úhel se sice zmenšuje, ale jeho hodnoty jsou mnohem menší než požadovaných 60°. Prakticky to znamená, že struktura vytvořená s použitím tohoto kritéria bude obsahovat stále „hubené pseudo-trojúhelníky“.

V příloze C jsou uvedena stejná měření pro množiny *gaus 10000* a *cluster 10000*. Výsledky měření na těchto dvou množinách jsou podobné jako na množině *uniform 10000*.

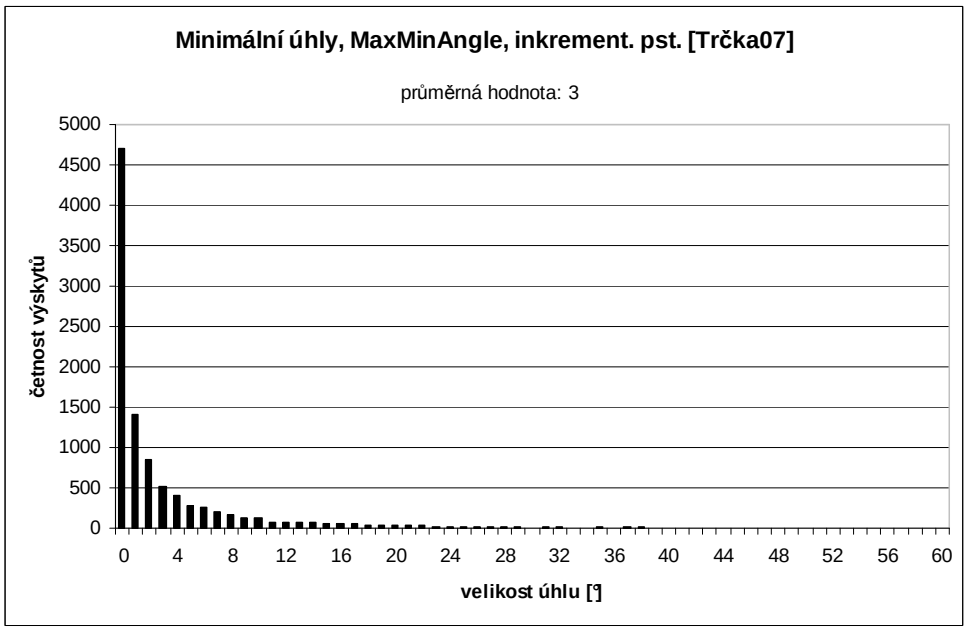
[23] Nejlepší pseudo-trojúhelníky podle těchto testů. V další kapitole se pokusím ukázat, že používání úhlových kritérií tak, jak jsme je definovali my, s sebou přináší určité problémy.

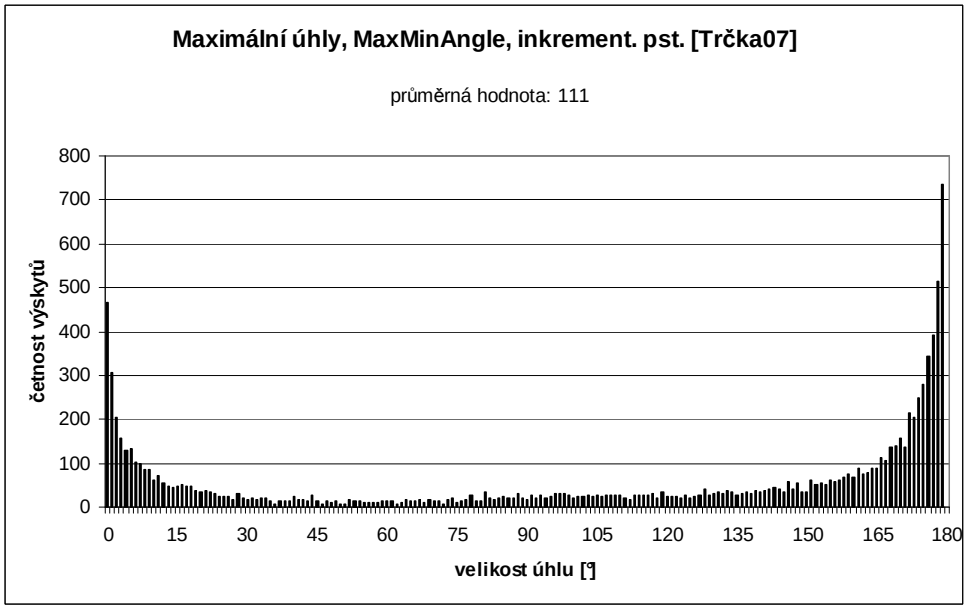


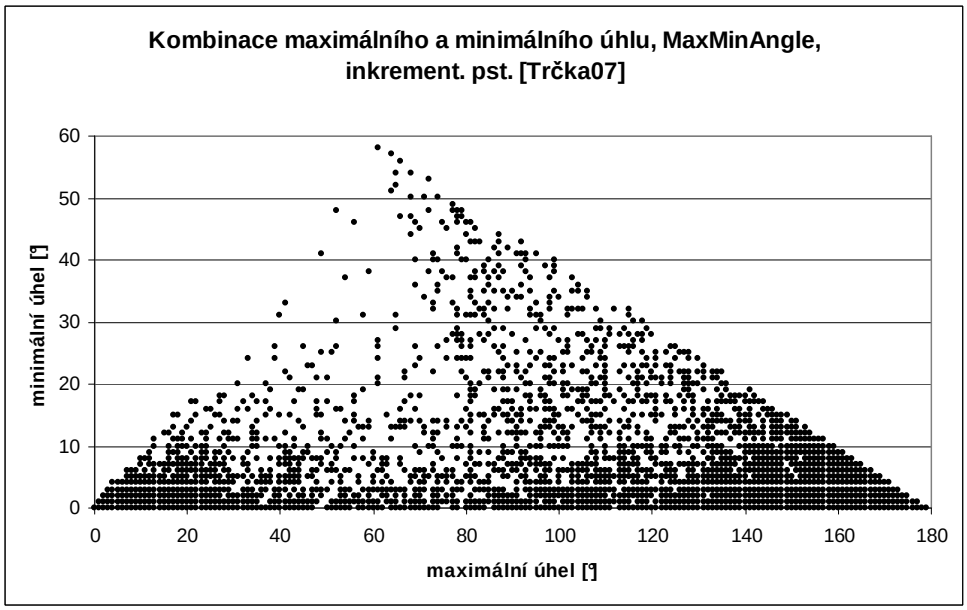




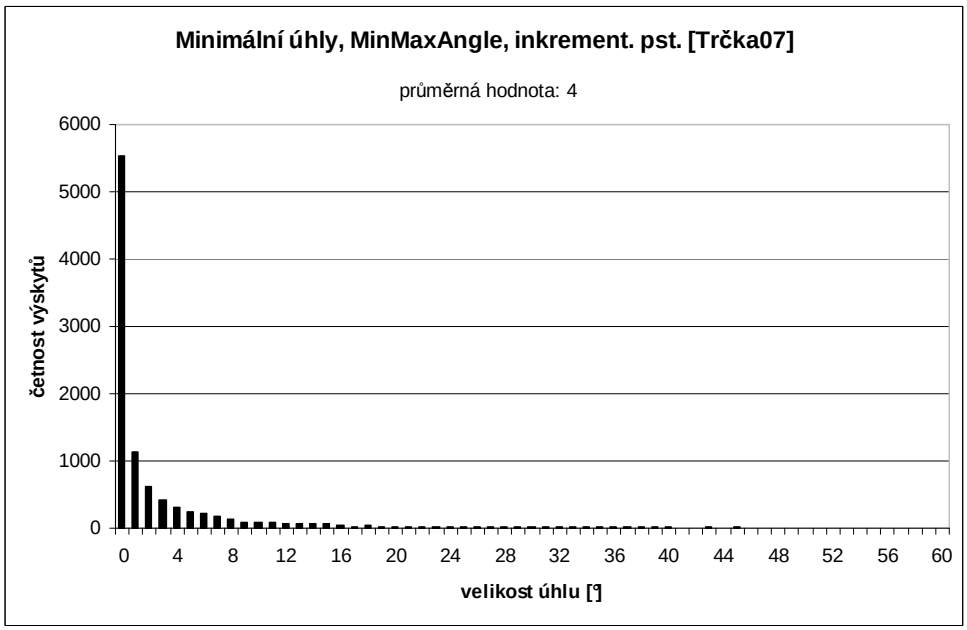
Graf 4.4. Výsledky testů pro MinLength, původní algoritmus [Trčka07].

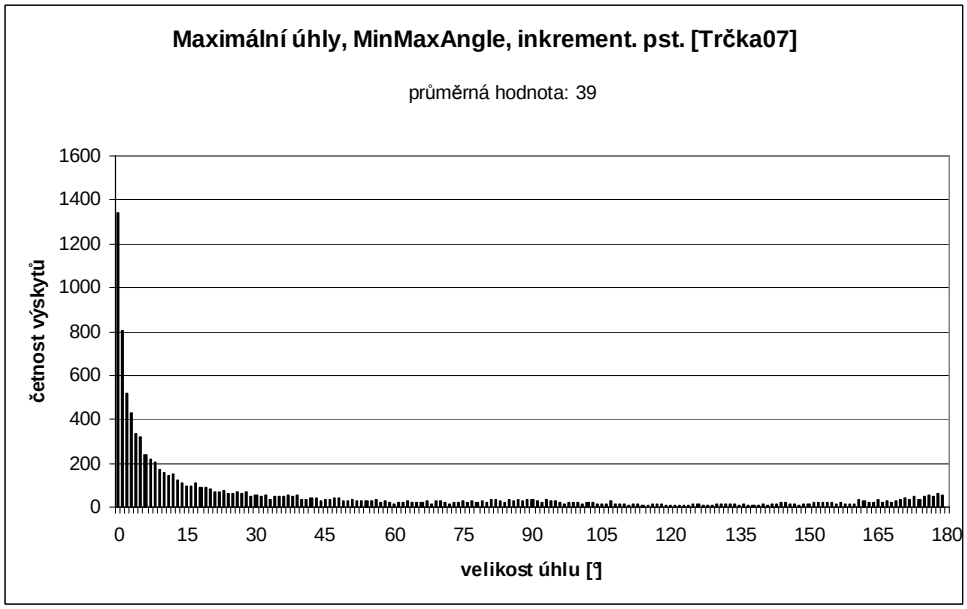


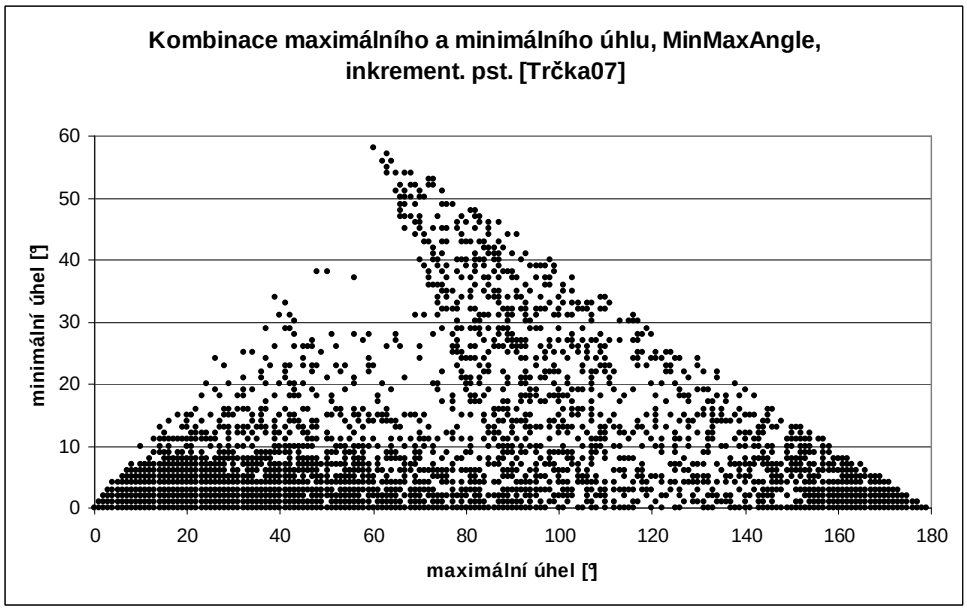




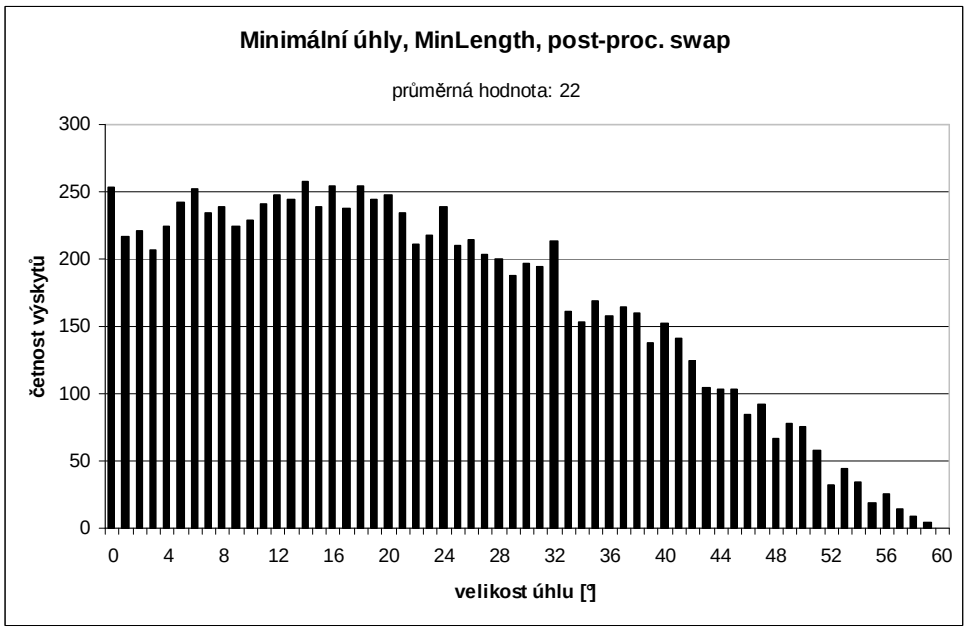
Graf 4.5. Výsledky testů kritérium MaxMinAngle, původní algoritmus [Trčka07].

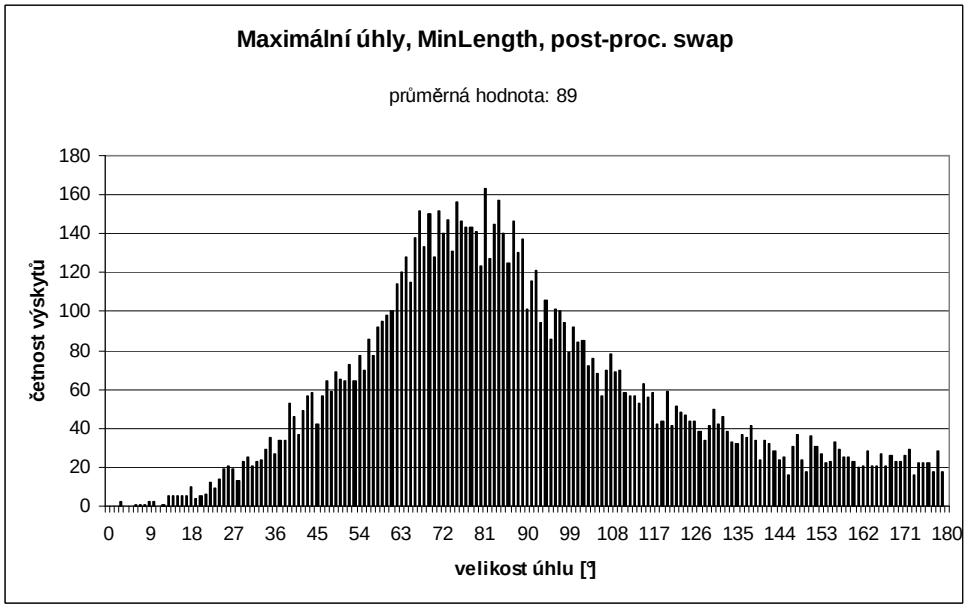


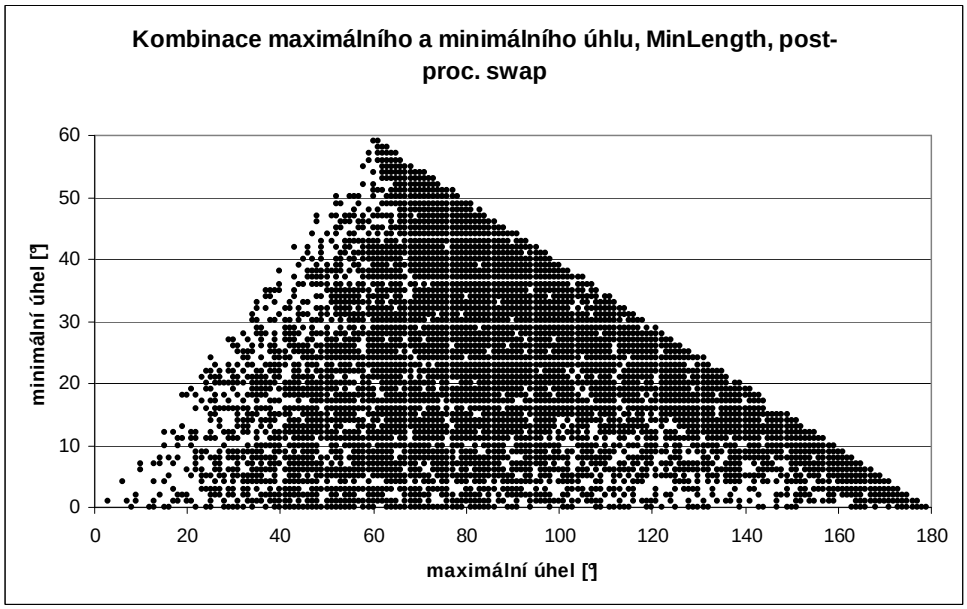




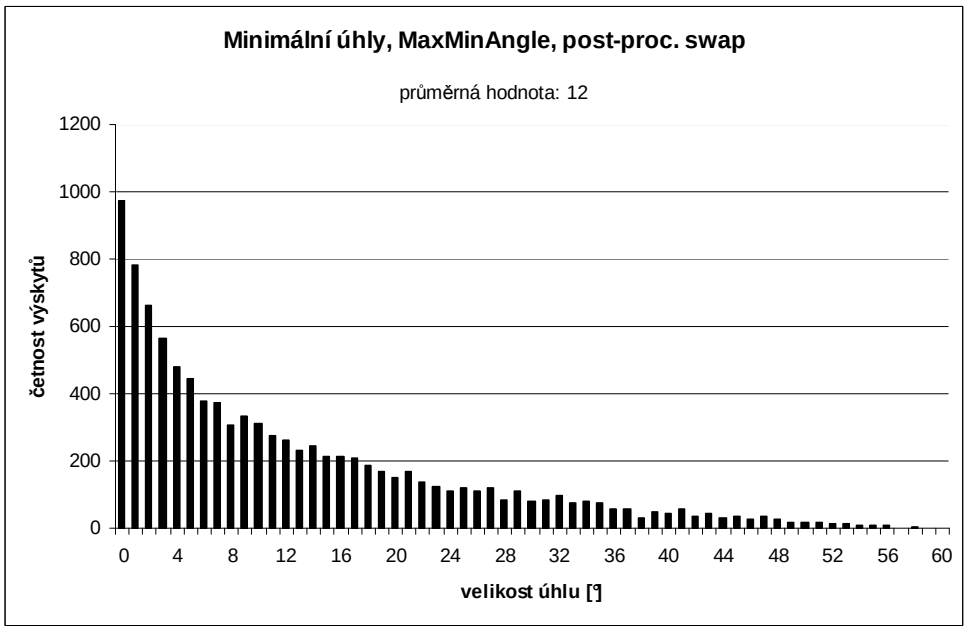
Graf 4.6. Výsledky testů pro kritérium MinMaxAngle, původní algoritmus [Trčka07].

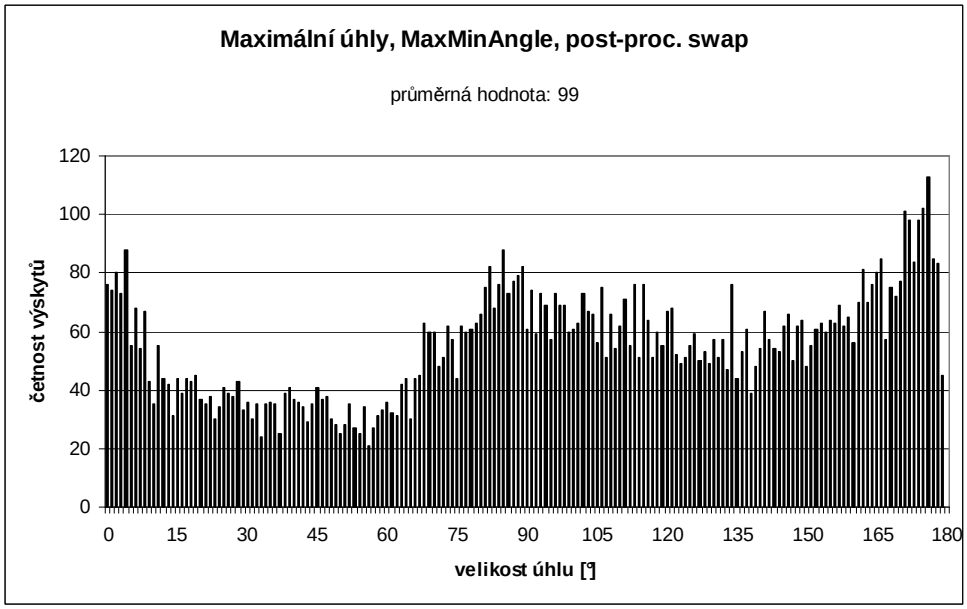


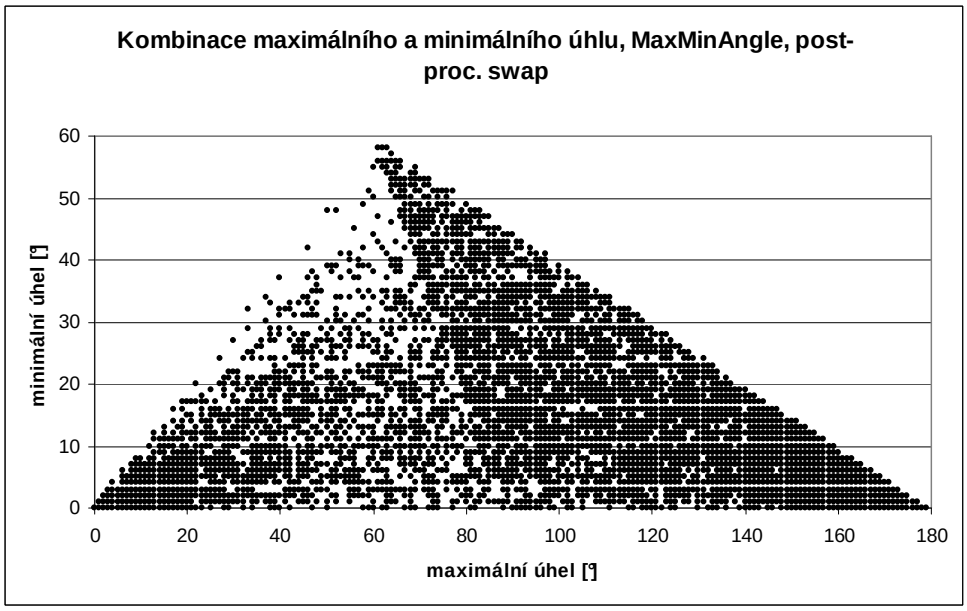




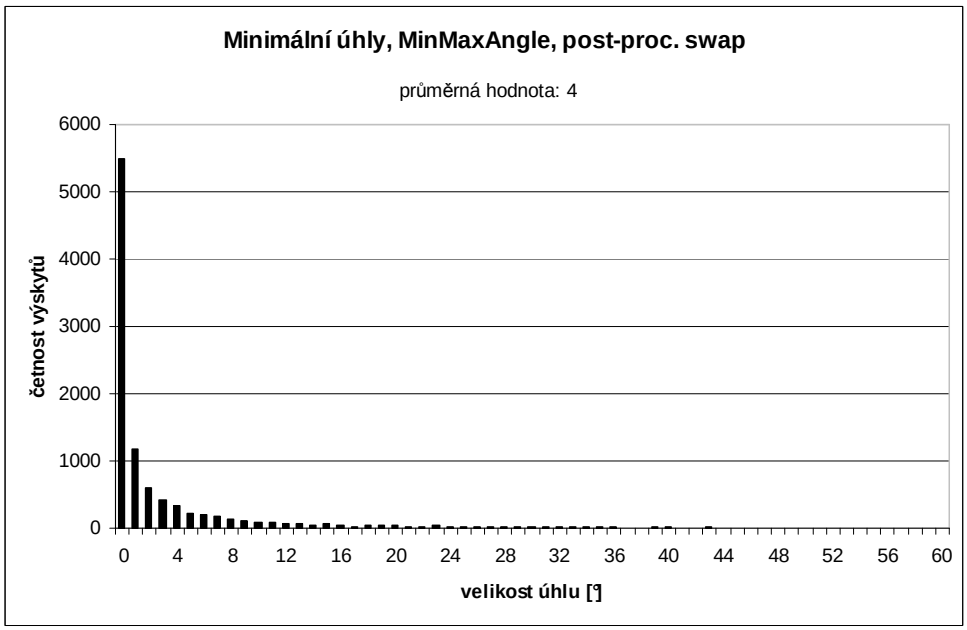
Graf 4.7. Výsledky testů pro kritérium MinLength, dodatečné prohazování.

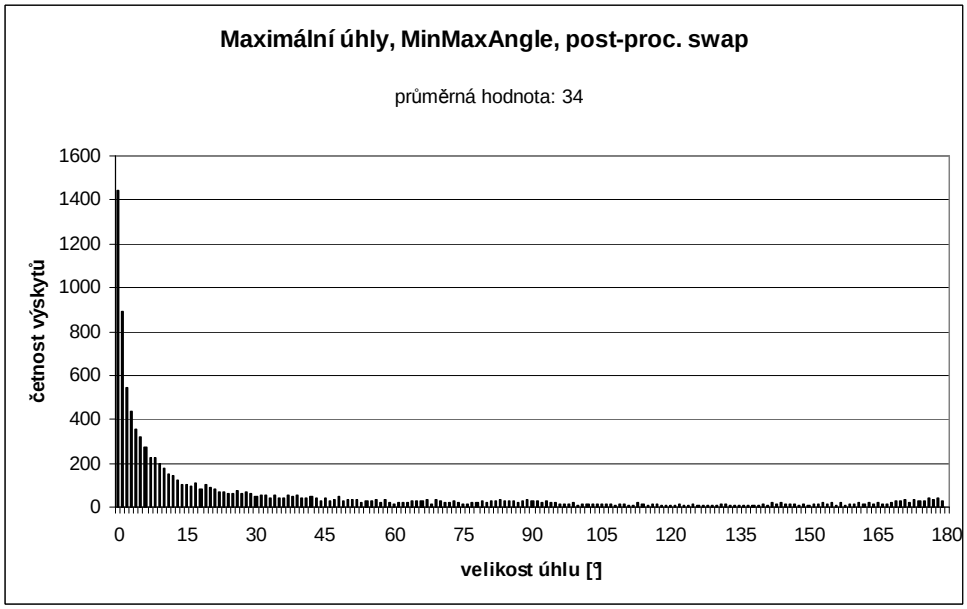


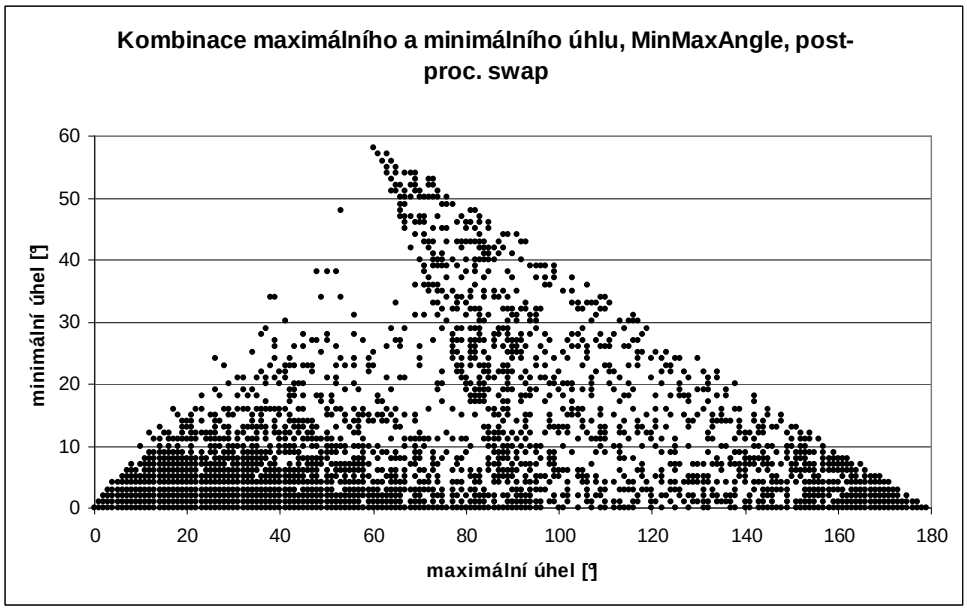




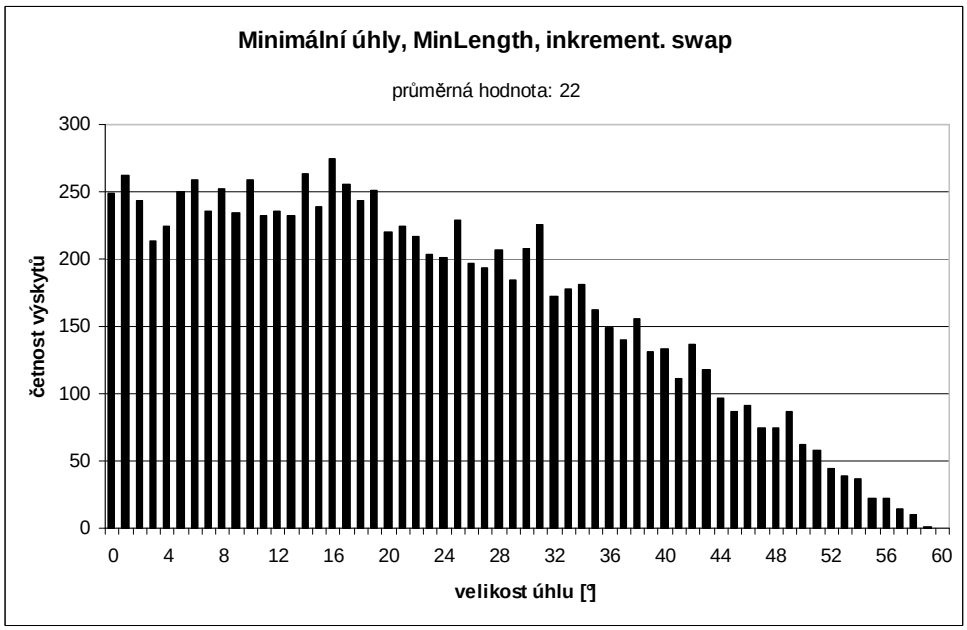
Graf 4.8. Výsledky testů pro kritérium MaxMinAngle, dodatečné prohazování.

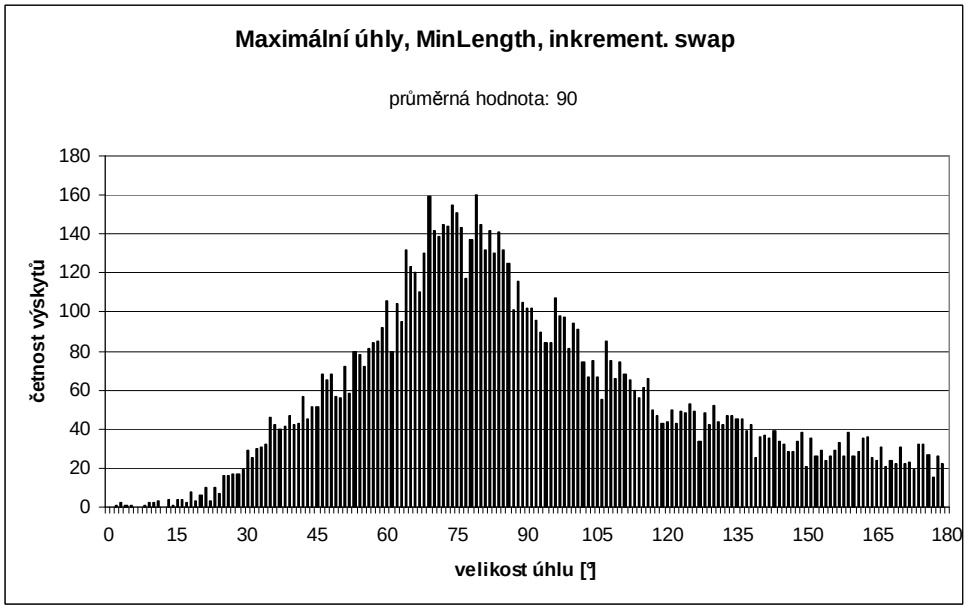


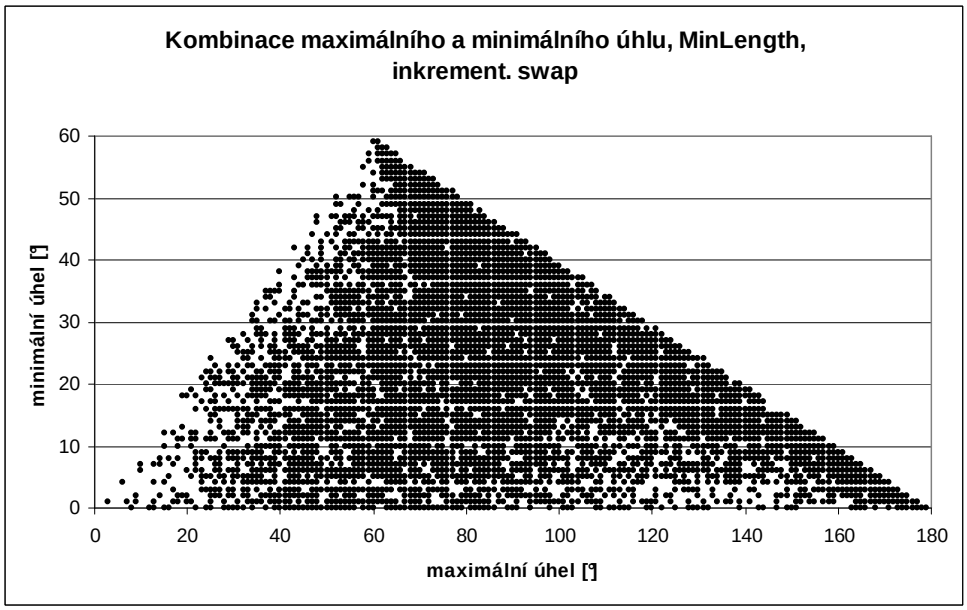




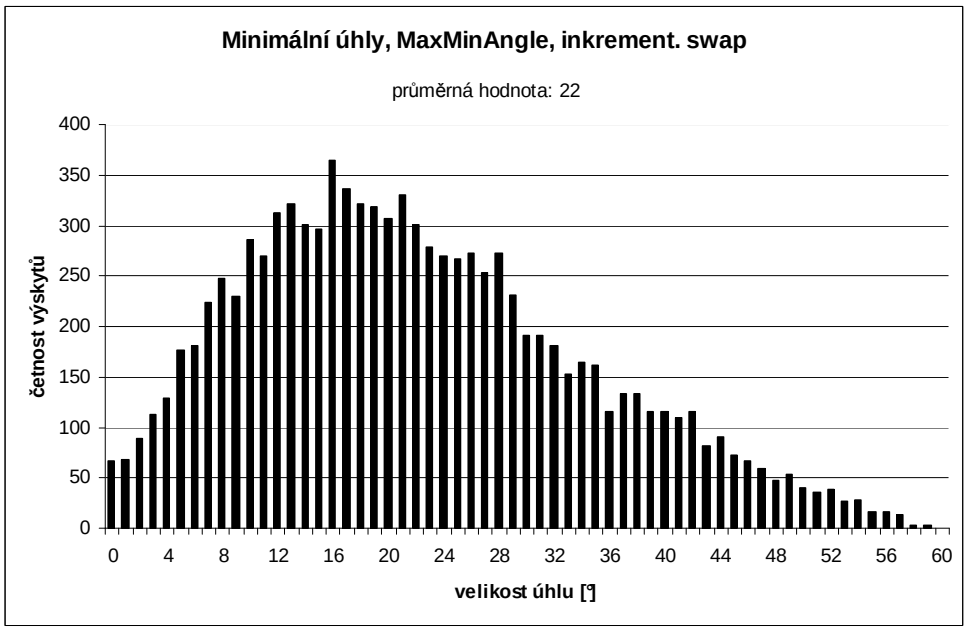
Graf 4.9. Výsledky testů pro kritérium MinMaxAngle, dodatečné prohazování.

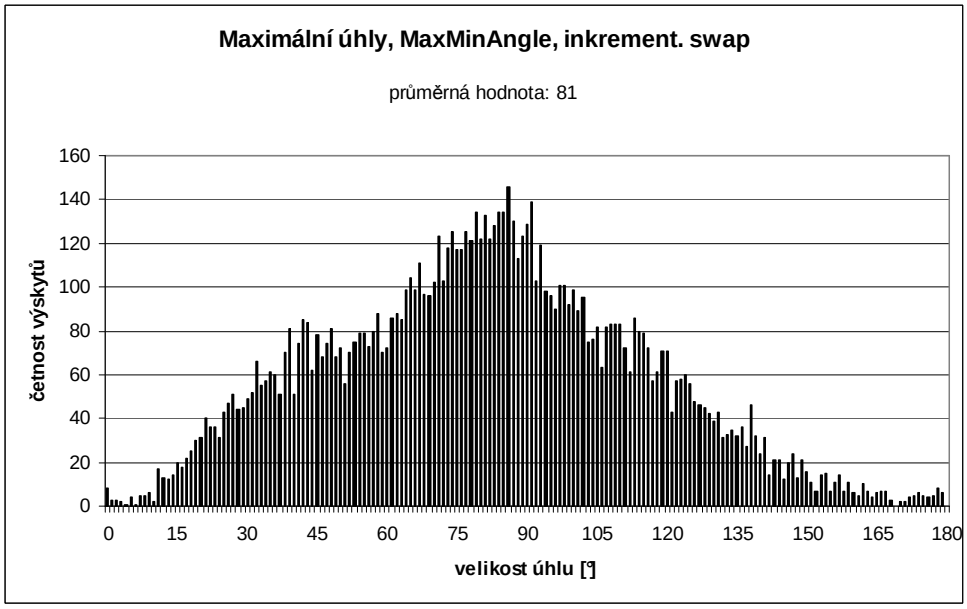


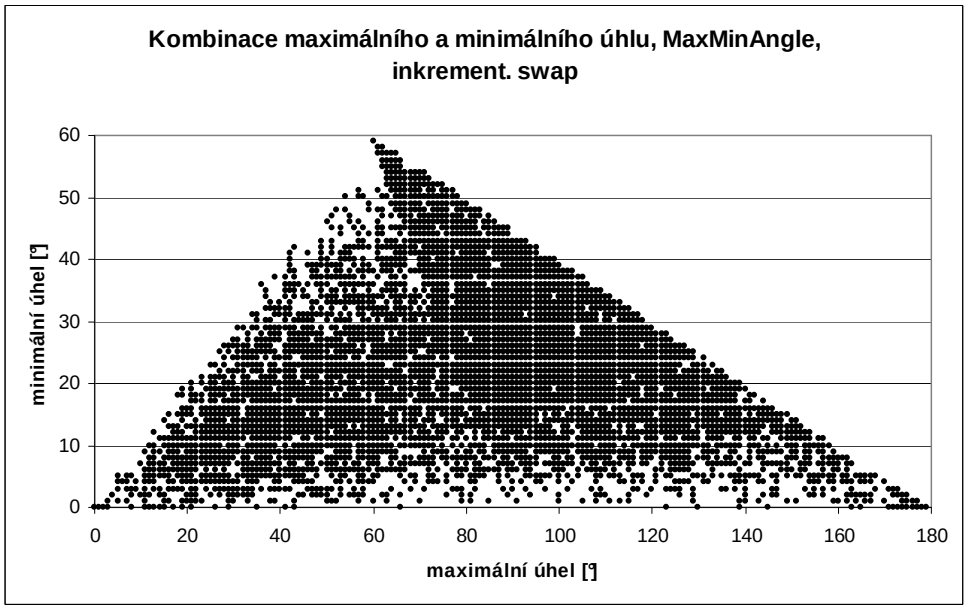




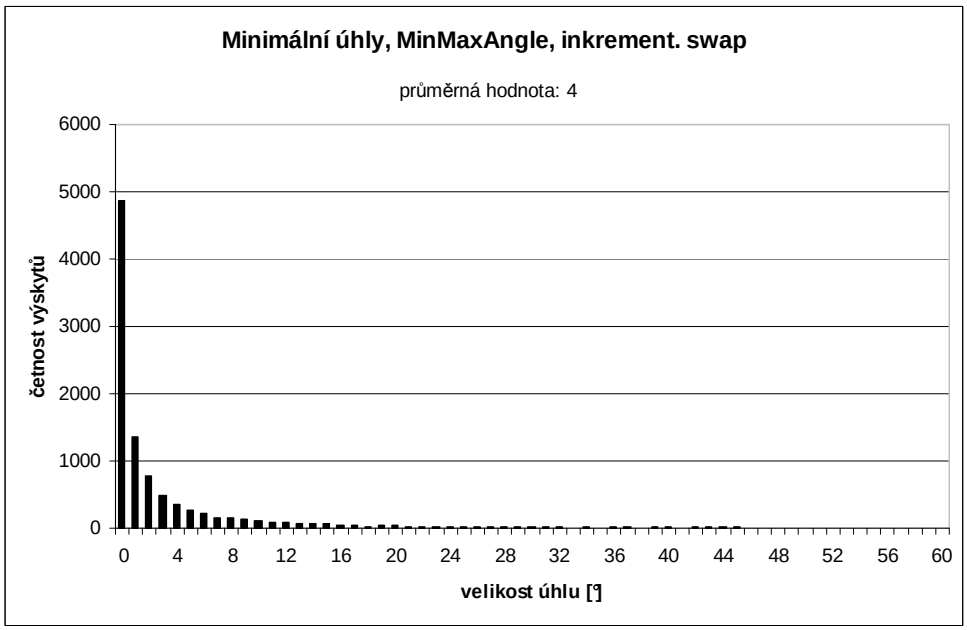
Graf 4.10. Výsledky testů pro kritérium MinLength, inkrementální swap.

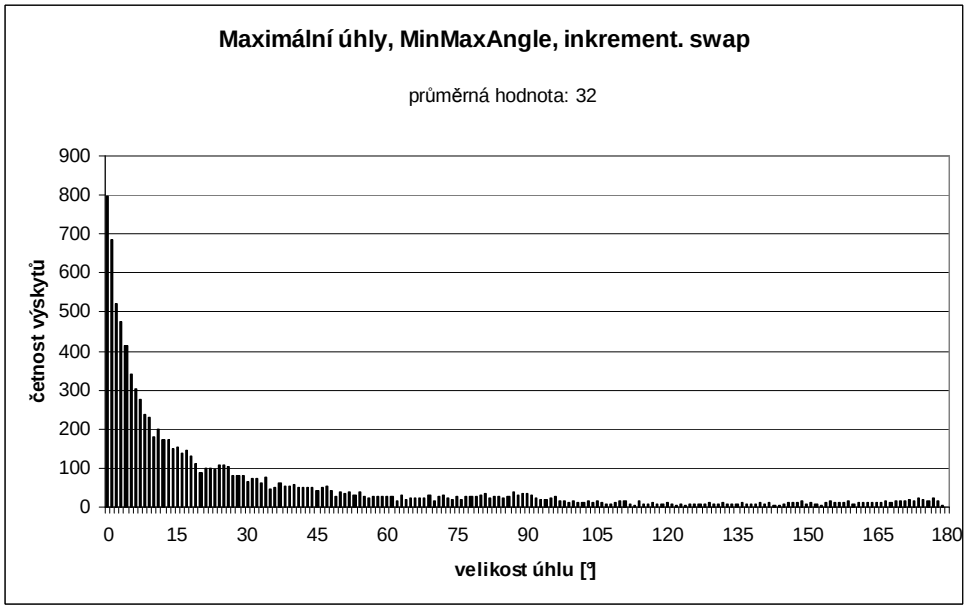


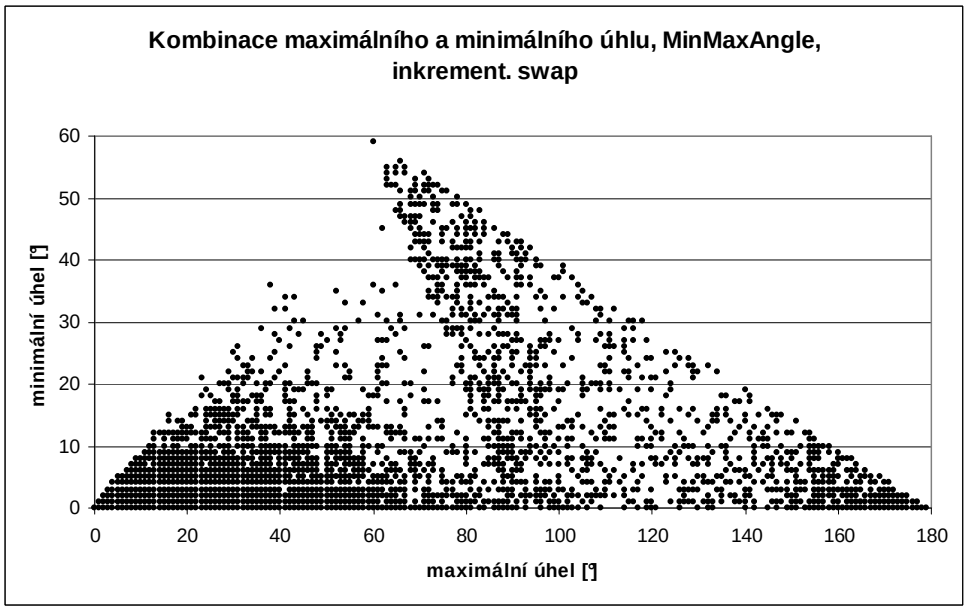




Graf 4.11. Výsledky testů pro kritérium MaxMinAngle, inkrementální swap.





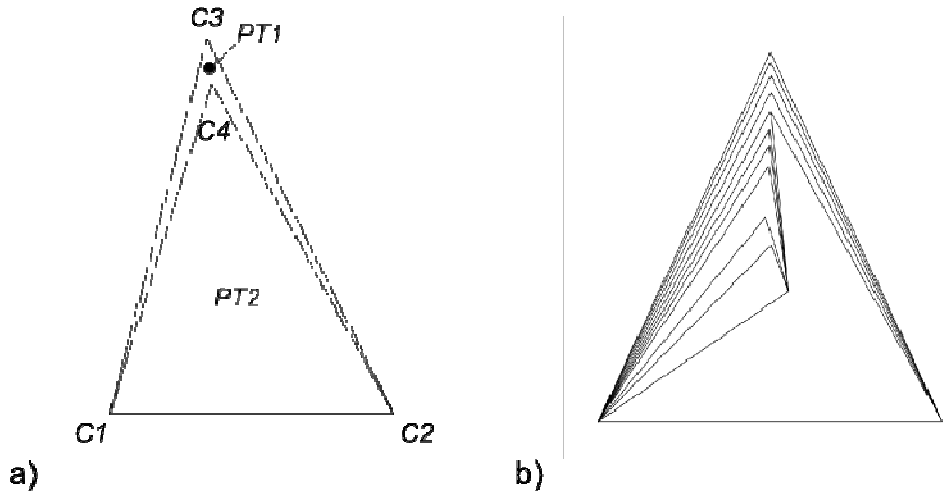


Graf 4.12. Výsledky testů pro kritérium MinMaxAngle, inkrementální swap.

4.2.1 Defekt úhlových kritérií

Mohlo by se zdát trochu zarážející, proč kritérium *MinMaxAngle* na první pohled neposkytuje kvalitní výsledky a kritérium *MaxMinAngle* poskytuje přijatelné výsledky pouze pro inkrementální swap. Podle mého názoru je to zapříčeno poněkud odlišnými vlastnostmi trojúhelníků a pseudo-trojúhelníků, pro které neplatí některé zákonitosti, např.: Součet vnitřních úhlů v rozích pseudo-trojúhelníku nemusí být právě 180° - obvykle bude menší. Maximální úhel v rozích pseudo-trojúhelníku nemusí být >= 60° - může být i menší.

Podívejme se nyní konkrétně na kritérium *MinMaxAngle*. Obrázek 4.13 a) ukazuje dvojici pseudo-trojúhelníků *PT1 – C1C2C3* a *PT2 – C1C2C4*. Pseudo-trojúhelník *PT1* je typickou ukázkou pseudo-trojúhelníku, jejichž vznik kritérium *MinMaxAngle* „podporuje“. Vhodnější by se zdálo, kdyby *PT1* byl tvořen vrcholy *C1C4C3* (tedy prohodit hranu *C4C2* za *C4C3*, což by se stalo při použití např. *MinLength* kritéria). Nicméně z hlediska probíraného kritéria je *PT1* optimální – maximální úhel je opravdu v *PT1* minimální. Na obrázku 4.13 b) je ukázka, jak může vypadat pseudo-triangulace vytvořená inkrementálním vkládáním s prohazováním hran při použití tohoto kritéria. Z obrázku 4.13 b) je jasně patrné, že kritérium *MinMaxAngle* použité na pseudo-triangulaci nebude vytvářet kvalitní pseudo-trojúhelníky. To také potvrdily testy z minulé kapitoly.



Obrázek 4.13. Ukázka pseudo-trojúhelníků, které jsou optimální pro kritérium MinMaxAngle.

Přejděme nyní na kritérium *MaxMinAngle*.

Triangulace T(P) na množině bodů P v rovině je Delaunayova triangulace tehdy a jen tehdy, když žádný bod z množiny P neleží uvnitř kružnice opsané libovolnému trojúhelníku z T(P). Delaunayova triangulace zajišťuje maximalizaci minimálního úhlu pro každý trojúhelník, jakož i pro celou triangulaci[24]. [Kolinger02]

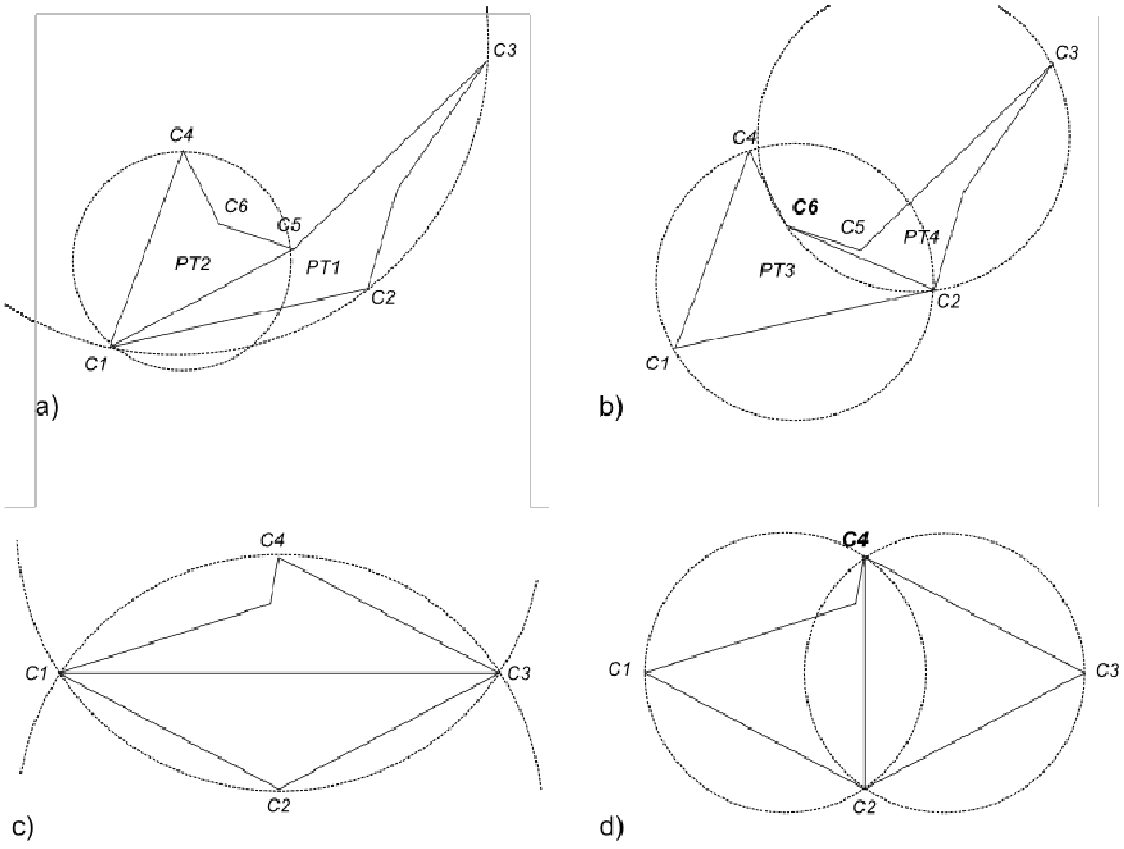
Virtuální trojúhelník PT budeme nazývat konvexní obal daného pseudo-trojúhelníku. [Trčka07]

Kružnice opsanou pseudo-trojúhelníku PT budeme nazývat kružnici opsanou virtuálnímu trojúhelníku PT.

[24] Nemám v úmyslu úplně srovnávat triangulaci s pseudo-triangulací, pouze chci na tomto příkladu ukázat, proč MaxMinAngle kritérium pro triangulace funguje a pro pseudo-triangulace ne.

Na příkladech z obrázku 4.14 bych chtěl ukázat, co nastává za problém při aplikaci *MaxMinAngle* kritéria na pseudo-triangulaci. Obrázek 4.14 a) ukazuje dvojici sousedních PT, *PT1 – C1C2C3* a *PT2 – C1C5C4*, a kružnice jim opsané. Tyto sousední PT jsou optimální pro kritérium *MaxMinAngle*. Vidíme však, že kružnice opsaná *PT1* bude obsahovat vrchol *C4* a pravděpodobně mnoho dalších vrcholů z případných dalších pseudo-trojúhelníků. Z tohoto důvodu se mi jeví rozdělení z obrázku 4.14 b) jako vhodnější: prohození hrany *C1C5* z obr. 4.14 a) za hranu *C2C6* z obr. 4.14 b). Je vidět, že i kružnice opsané sousedním PT z obr. 4.14 b) budou pravděpodobně obsahovat méně vrcholů dalších pseudo-trojúhelníků. Takového rozdělení lze dosáhnout použitím kritéria *MinLength* nebo v případě, že budeme kritérium *MaxMinAngle* aplikovat na virtuální trojúhelníky sousedních PT. Úhel u zvýrazněného vrcholu *C6* v pseudo-trojúhelníku *PT4* je nejmenší úhel v rozích sousedních PT z obrázků 4.14 b) i 4.14 a). Stejný problém je potom znázorněn na obrázcích 4.14 c) a 4.14 d).

> *Poznámka 3:* Z obrázku 4.14 b) je také patrné, že kružnice opsaná pseudo-trojúhelníku, který má více než tři vrcholy, tj. aspoň jedna jeho strana je složená z více než jedné hrany, bude obsahovat roh jiného pseudo-trojúhelníku (kružnice opsaná pseudo-trojúhelníku *PT3* obsahuje vrcholy *C6* a *C5* z *PT4*).



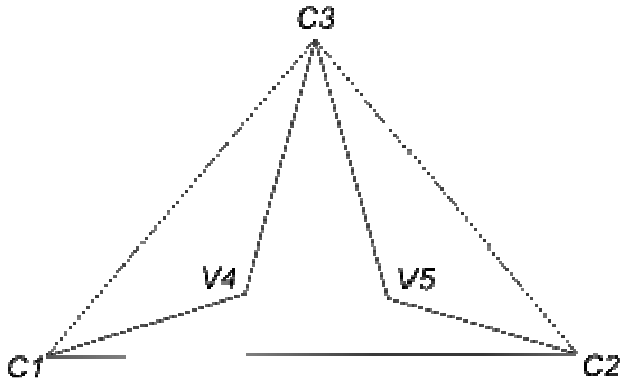
Obrázek 4.14. Příklady, kde nefunguje kritérium MaxMinAngle podle principu převzatého z Delaunayovy triangulace .

Příklady z obrázku 4.14 mají tu společnou vlastnost, že v rozdělení sousedních PT, které se nám podle principu opsaných kružnic zdá být vhodnější, se vyskytuje vrchol blízko jednoho z rohů sousedních PT (jde o zvýrazněné rohy *C6* a *C4*), který v daném rohu minimalizuje úhel. Kritérium *MaxMinAngle* potom ve snaze maximalizovat minimální úhel v rozích vybere takové rozdělení, kde sice bude minimální úhel v rozích větší, ale celkový tvar sousedních PT bude horší. Tímto jsem chtěl říci, že úhel v rozích pseudo-trojúhelníků nemusí úplně přesně vypovídat o jejich kvalitě (což je zásadní rozdíl oproti trojúhelníkům). Pokud bychom se podívali na PT vytvořenou inkrementálním vkládáním s prohazováním hran s použitím kritéria *MaxMinAngle* více zblízka, třeba v přiloženém programu, uvidíme, že ačkoli podle testů z kapitoly 4.2 by měla takto vytvořená struktura obsahovat nejkvalitnější pseudo-trojúhelníky, struktura vytvořená stejným způsobem za použití kritéria *MinLength* bude vytvářet „o malinko lepší“ trojúhelníky.

V této kapitole jsem se pokusil ukázat, že použití kritérií *MaxMinAngle* a *MinMaxAngle* na pseudo-triangulace nemusí poskytovat optimální výsledky, i když pro triangulace je poskytuje. V další kapitole se budeme zabývat otázkou: Nebyly by výsledné pseudo-trojúhelníky při použití těchto kritérií lepší, pokud bychom výše probíraná kritéria aplikovali na virtuální trojúhelníky daných pseudo-trojúhelníků?

4.2.2 Aplikace úhlových kritérií na konvexní obal pseudo-trojúhelníku

V této kapitole budeme testovat kritéria *MaxMinAngle* a *MinMaxAngle* aplikovaná na virtuální trojúhelník pseudo-trojúhelníku. Na obrázku 4.15 je vidět rozdíl v použití kritéria. Zatímco doposud byl minimální, resp. maximální úhel vypočten mezi dvěma po sobě jdoucími hranami, jež měly jako společný vrchol jeden z rohů pseudo-trojúhelníku, nyní budeme úhel počítat mezi hranami virtuálního trojúhelníku – úhel u *C1* mezi hranami *C1C3* a *C1C2*, analogicky úhel u *C2* a *C3*.



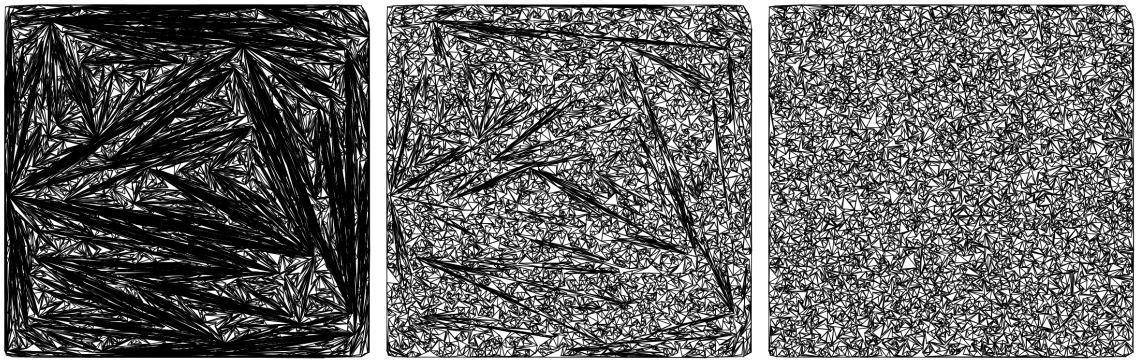
Obrázek 4.15. Pseudo-trojúhelník a jeho virtuální trojúhelník.

Takto upravená kritéria jsme pojmenovali *MaxMinAngleHull* a *MinMaxAngleHull*.

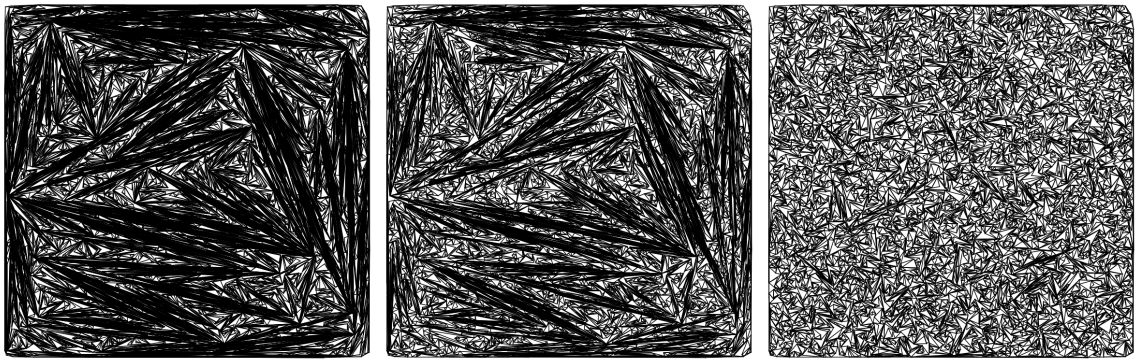
Pro tyto nová kritéria jsme provedli stejné testy jako v kapitole 4.2[25]. Na obrázku 4.16 jsou vidět výsledky těchto testů pro kritérium *MaxMinAngleHull*. První zleva je pseudo-triangulace vytvoření původním algoritmem [Trčka07], následuje pseudo-triangulace s dodatečným prohazováním hran a pseudo-triangulace s inkrementálním swapem.

[25] V těchto testech se maximální, resp. minimální úhel měří stále v rozích pseudo-trojúhelníku, ne v rozích virtuálního trojúhelníku.

Na obrázku 4.17 je stejná posloupnost pseudo-triangulací pro kritérium *MinMaxAngleHull*. Již z obrázků je vidět, že pro kritérium *MaxMinAngleHull* nedošlo k zjevnému zlepšení. Naopak pro kritérium *MinMaxAngleHull* k jistému zlepšení došlo v pseudo-triangulaci s inkrementálním prohazováním hran (porovnáváno s obrázkem 4.3 z kapitoly 4.2).



Obrázek 4.16. Ukázka pseudo-triangulací pro kritérium MaxMinAngleHull. Původní inkrementální algoritmus [Trčka07], post-processing swap a inkrementální swap.



Obrázek 4.17. Ukázka pseudo-triangulací pro kritérium MinMaxAngleHull. Původní inkrementální algoritmus [Trčka07], post-processing swap a inkrementální swap.

Vyhodnocení testů kvality pseudo-triangulace pro kritérium ***MaxMinAngleHull***:

Grafy 4.18, 4.20 a 4.21 – původní algoritmus [Trčka07], post-processing swap a inkrementální swap.

Hodnota průměrného minimálního úhlu se zvedla ze 4° na 17° pro post-processing i inkrementální swap v porovnání s původním algoritmem [Trčka07]. Maximální úhel se zvýšil z 54° na 68° pro post-processing a na 66° pro inkrementální swap v porovnání s původním algoritmem [Trčka07].

Srovnejme tyto výsledky s výsledky pro kritérium *MaxMinAngle* z kapitoly 4.2. Můžeme říci, že se velikost maximálního úhlu poměrně výrazně přiblížila 60°. Naopak minimální úhel zůstal podobný (pro post-processing swap mírně stoupl, pro inkrementální swap mírně klesl).

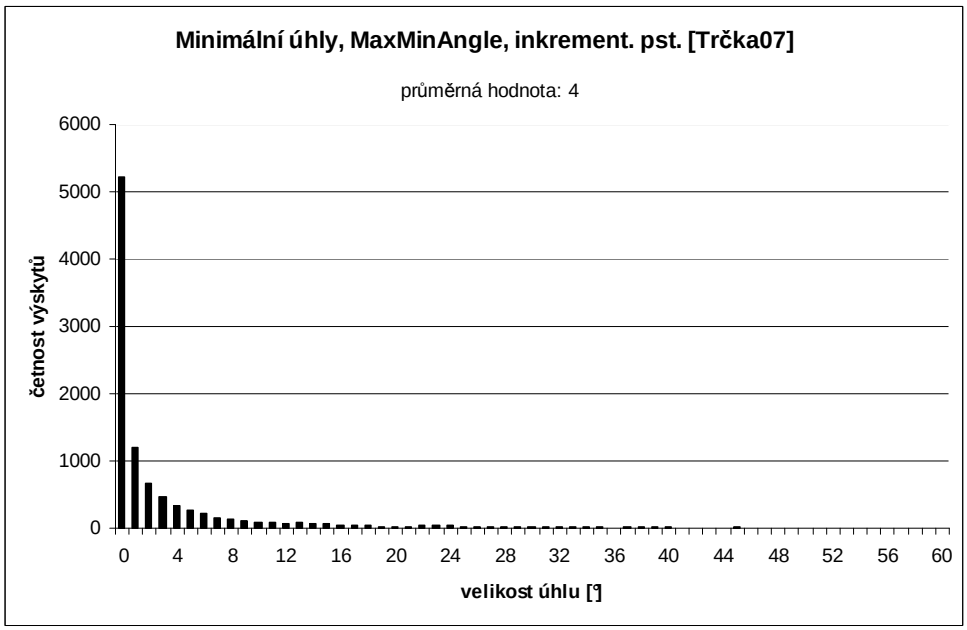
Vyhodnocení testů kvality pseudo-triangulace pro kritérium ***MinMaxAngleHull***:

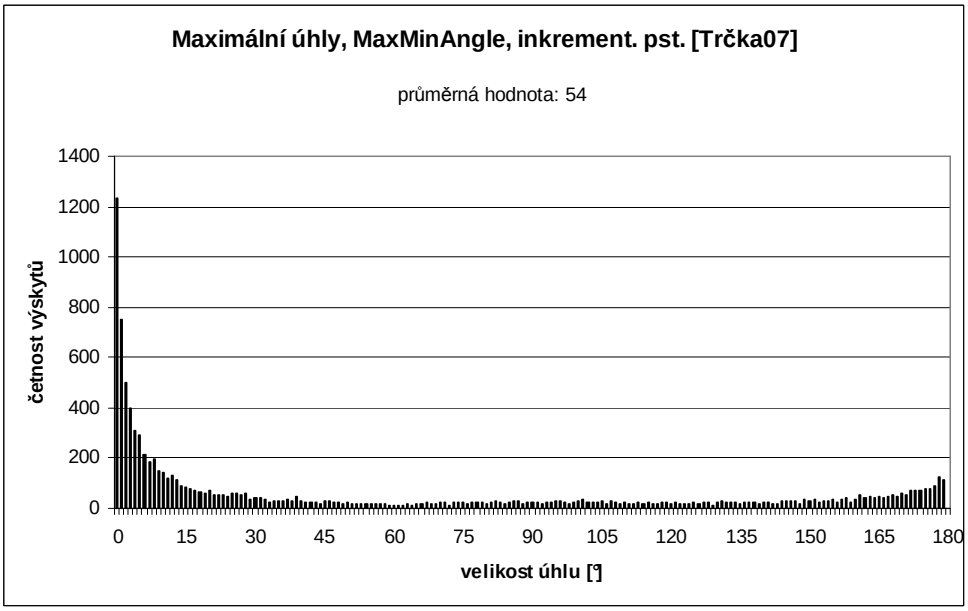
Grafy 4.19, 4.21 a 4.23 – původní algoritmus [Trčka07], post-processing swap a inkrementální swap.

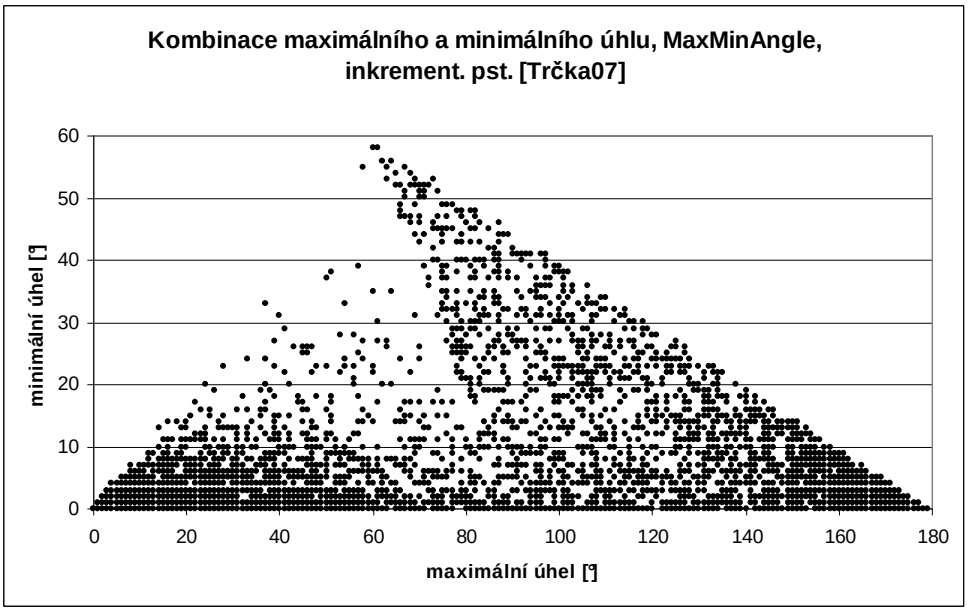
Hodnota průměrného minimálního úhlu se zvedla ze 4° na 8° pro post-processing a na 14° pro inkrementální swap v porovnání s původním algoritmem [Trčka07]. Maximální úhel se zvýšil z 56° na 69° pro post-processing a na 62° pro inkrementální swap v porovnání s původním algoritmem [Trčka07].

Srovnejme opět tyto výsledky z výsledky pro kritérium *MinMaxAngle* z kapitoly 4.2. Zde došlo k zlepšení výraznému při použití kritéria *MinMaxAngleHull*, kdy se maximální úhel velmi přiblížil 60° a minimální úhel se pro inkrementální swap také mnohem zvýšil (ze 4° - kap. 4.2, na 14°). I z obrázku 4.17 je vidět, že by inkrementální swap mohl vytvářet kvalitní pseudo-trojúhelníky.

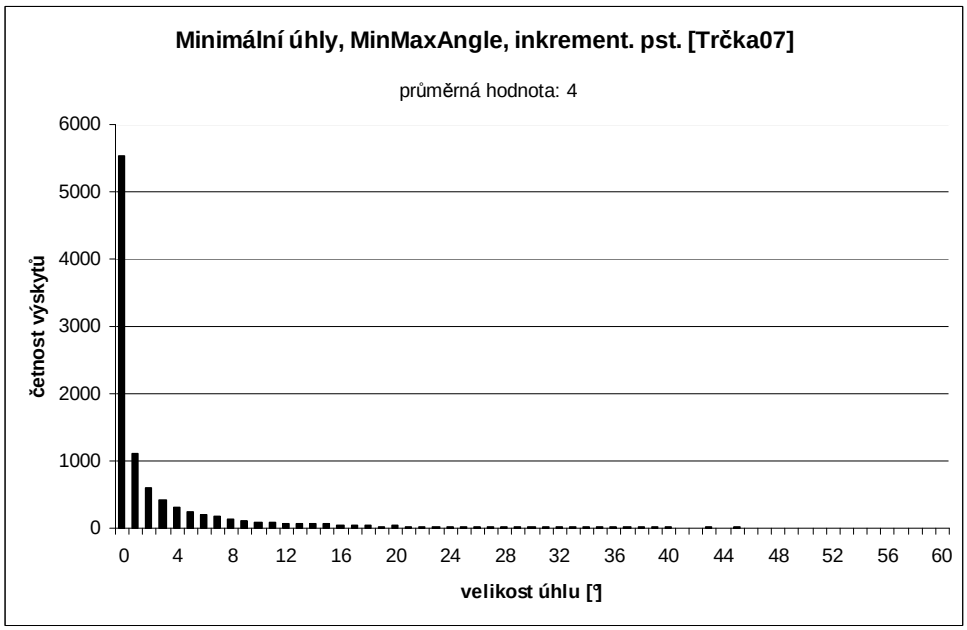
Z výše uvedených testů by se zdálo, že nová kritéria *MaxMinAngleHull* a *MinMaxAngleHull* by mohla poskytovat kvalitnější pseudo-trojúhelníky než kritéria *MinLength* nebo *MaxMinAngle* (inkrementální swap). Pokud si ovšem výsledné struktury lépe prohlédneme, třeba v přiloženém programu, zjistíme, že opět dochází ke vzniku „hubených trojúhelníků“ a ke kumulaci hran v určitých vrcholech ve větší míře, než tomu bylo u zmiňovaných kritérií *MinLength* nebo *MaxMinAngle*. Jeden z možných důvodů relativního neúspěchu těchto kritérií je vidět na obrázku 4.13 a), pseudo-trojúhelník *PT1*. Minimální úhly virtuálního trojúhelníku *PT1* jsou mnohem větší než úhly v rozích pseudo-trojúhelníku *PT1*.

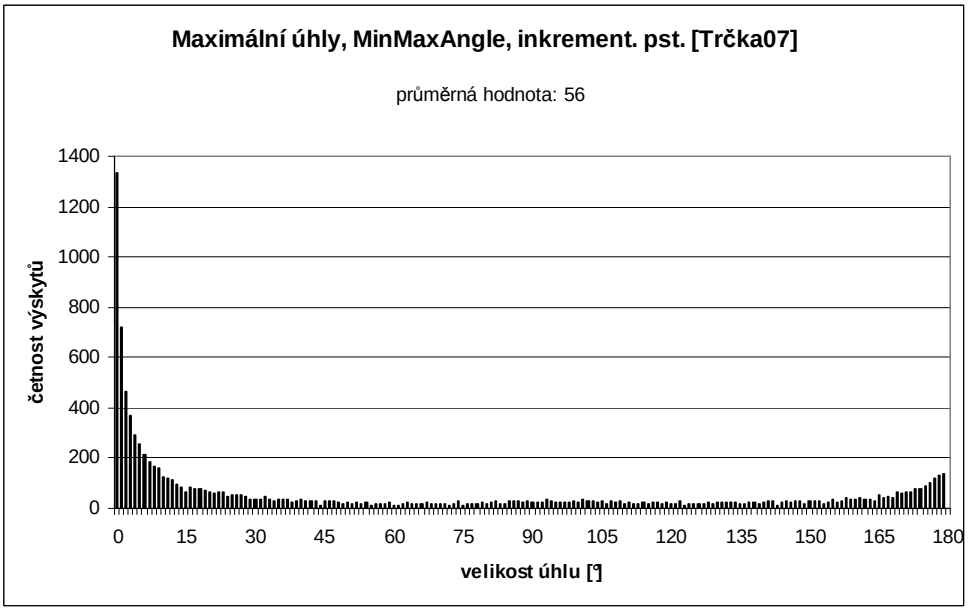


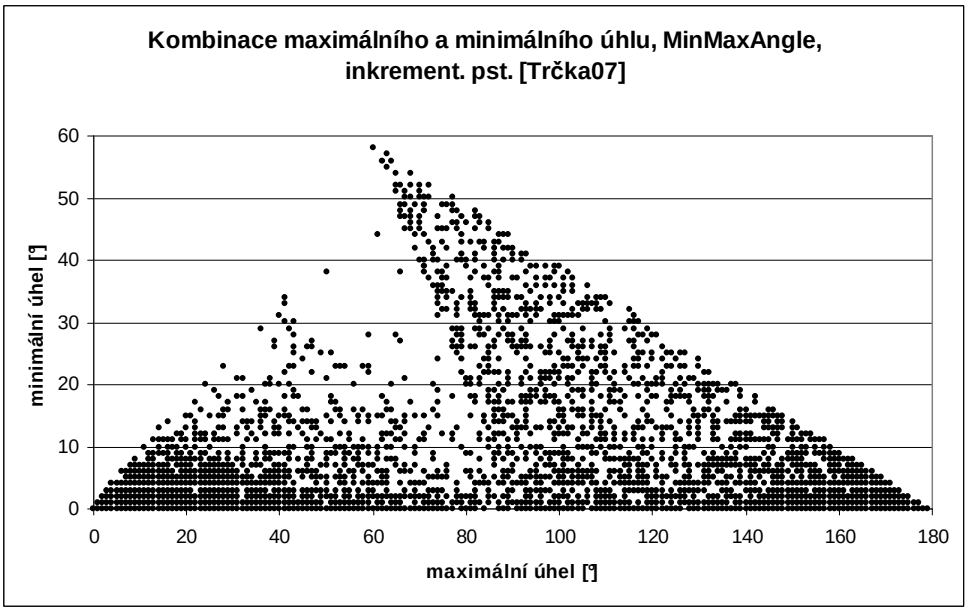




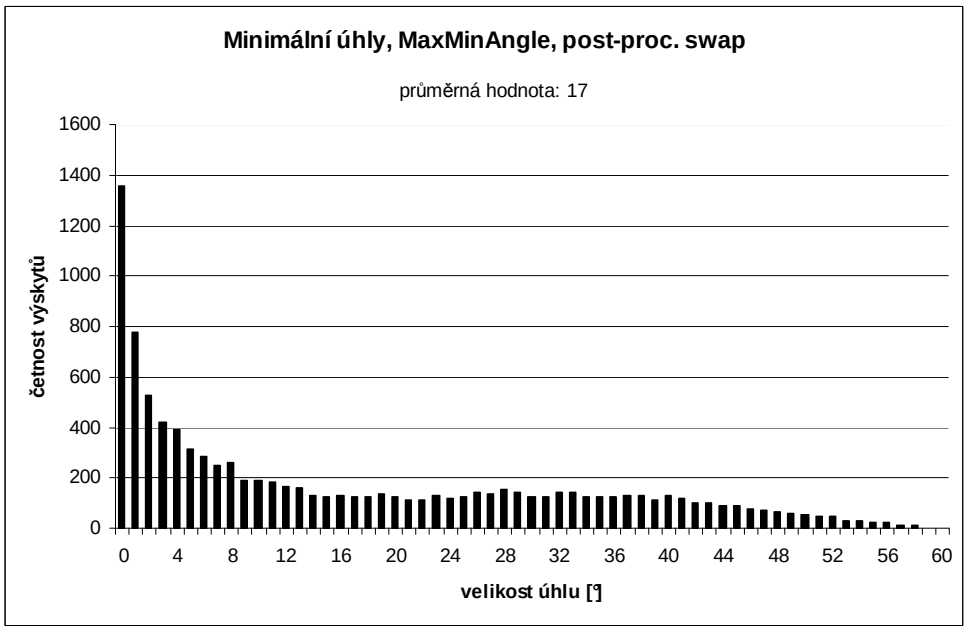
Graf 4.18. Výsledky testů pro kritérium MaxMinAngle, původní algoritmus [Trčka07].

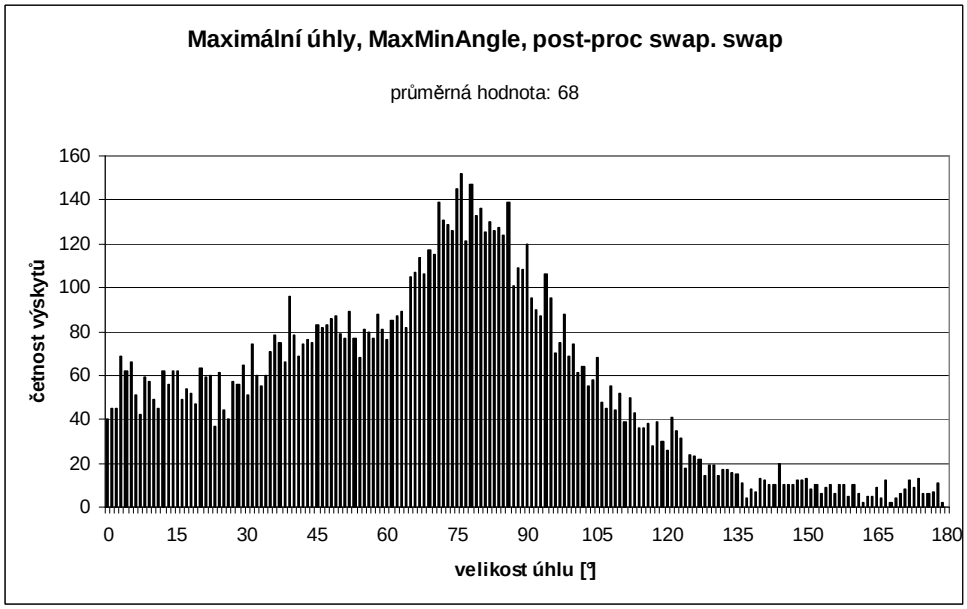


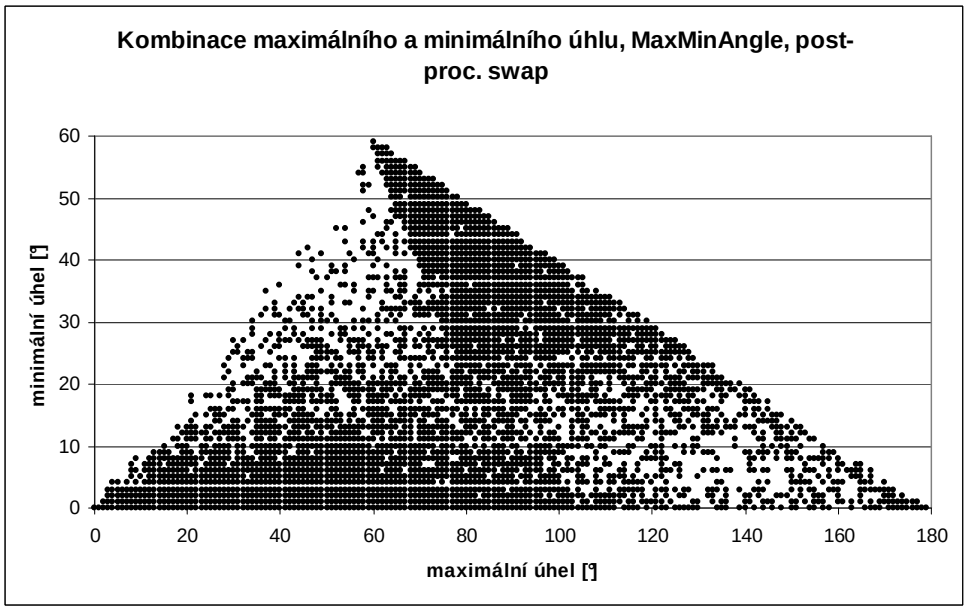




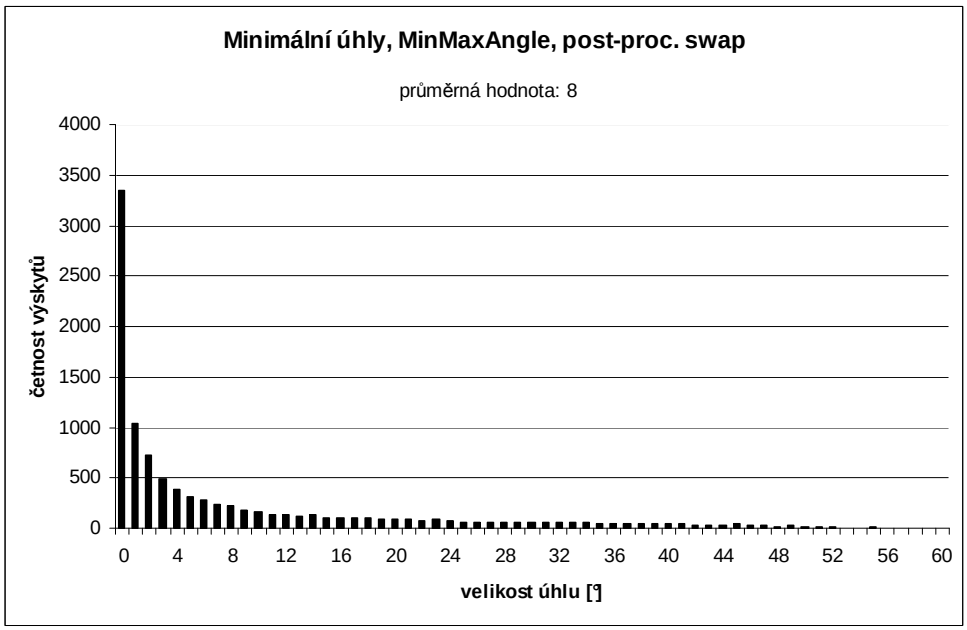
Graf 4.19. Výsledky testů pro kritérium MinMaxAngle, původní algoritmus [Trčka07].

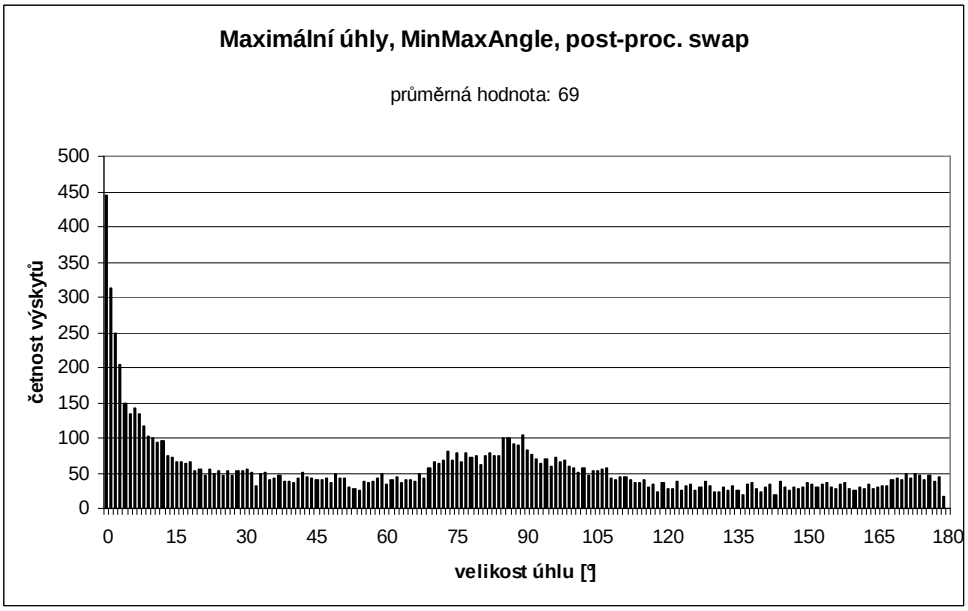


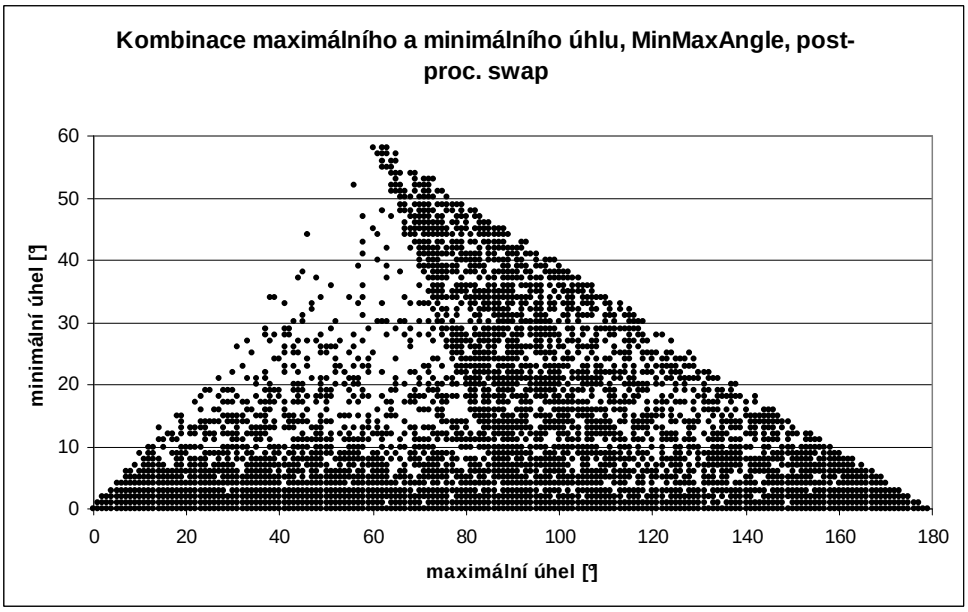




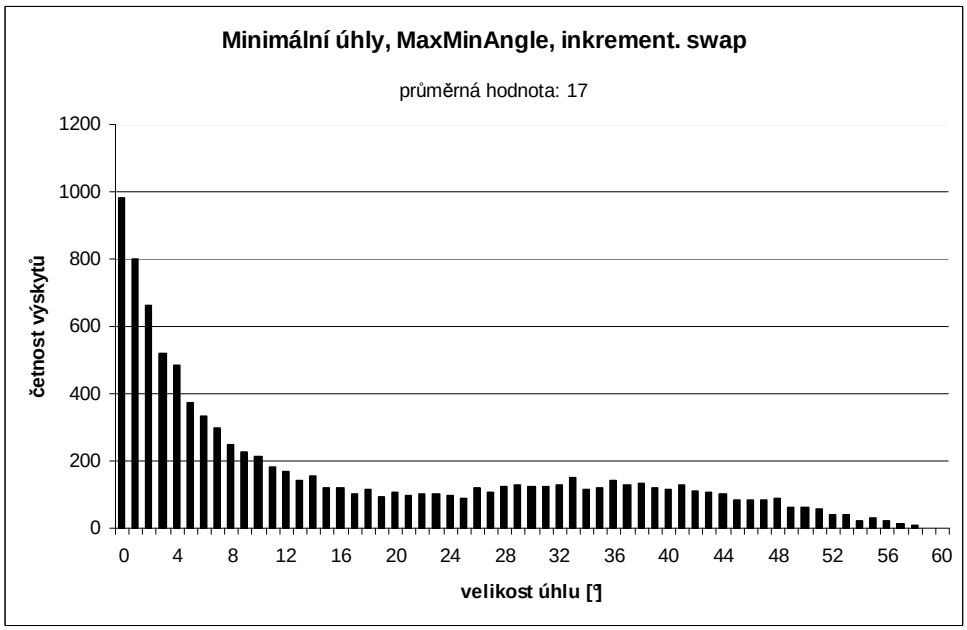
Graf 4.20. Výsledky testů pro kritérium MaxMinAngle, post-processing swap.

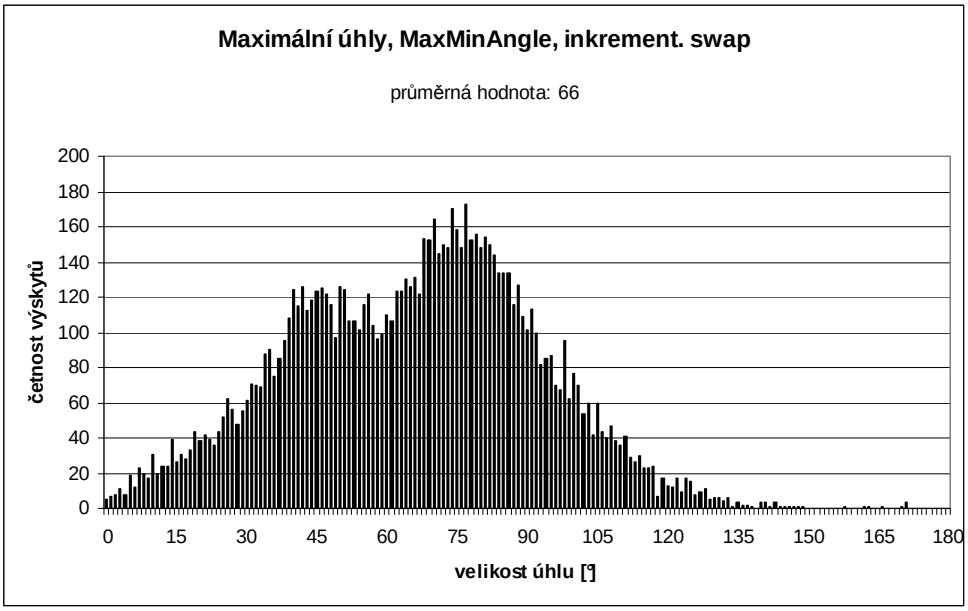


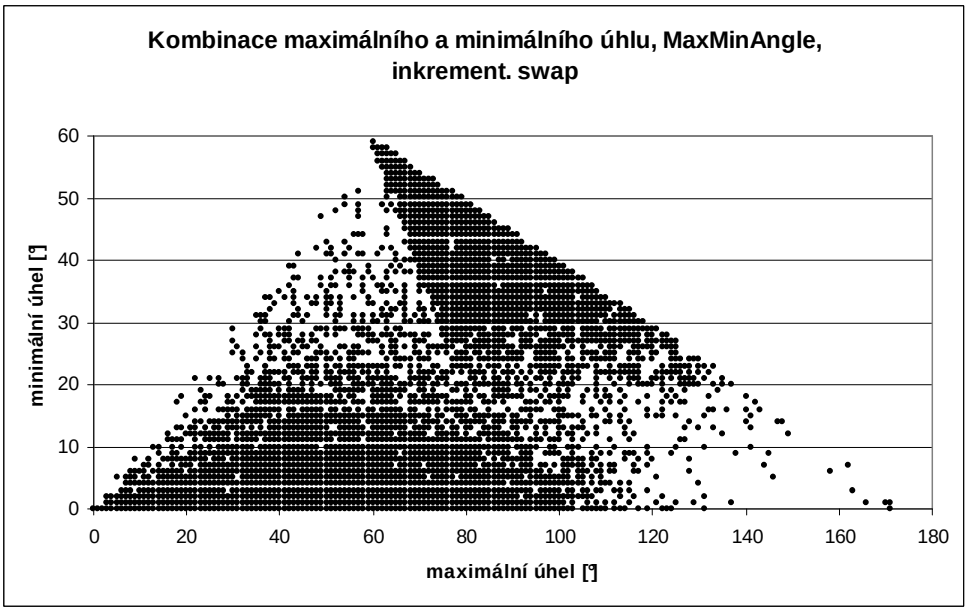




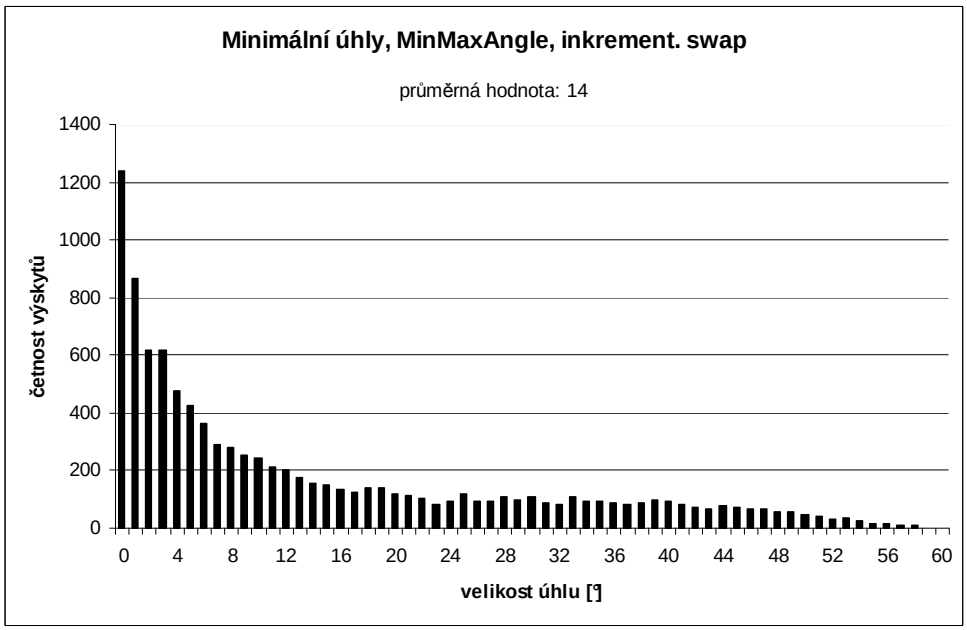
Graf 4.21. Výsledky testů pro kritérium MinMaxAngle, post-processing swap.

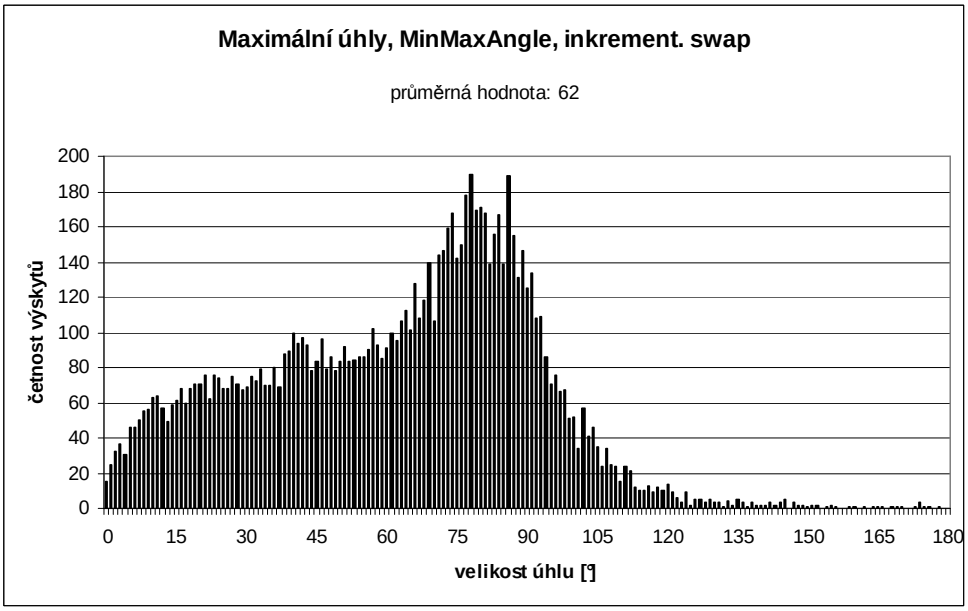


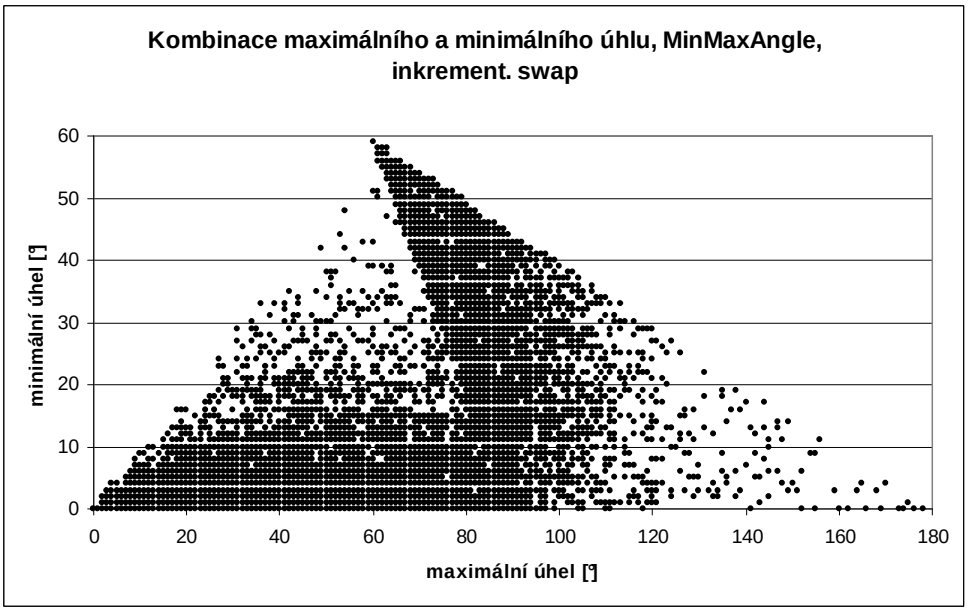




Graf 4.22. Výsledky testů pro kritérium MaxMinAngle, inkrementální swap.







Graf 4.23. Výsledky testů pro kritérium MinMaxAngle, inkrementální swap.

4.2.3 Závěr pro používání úhlových kritérií

Závěrem k testům výsledných pseudo-triangulací. V kapitolách 4.2.1 a 4.2.2 jsem se snažil ukázat, že používání úhlových kritérií tak, jak se používají pro triangulace, viz Delaunayova triangulace, kapitola 4.2.1, nemusí nutně znamenat stejně kvalitní výsledky pro pseudo-triangulace. Dokonce bych tvrdil, že používání úhlových kritérií námi definovaných, popř. definovaných v kapitole 5.3 v původní práci [Trčka07], nepovede k přijatelným výsledkům. Moje osobní domněnka je, že jediné námi testované kritérium, které poskytuje relativně dobré výsledky, je kritérium *MinLength*.

Tímto závěrem jsem nechtěl říci, že přes úhlová kritéria „cesta nevede“. Myslím, že by bylo vhodné navrhnout kritéria jinak, třeba aby zohledňovala tvar celého pseudo-trojúhelníku, nejen blízkého okolí jeho rohů. Možná by stálo za uvážení zkusit nějaké kritérium, které počítá i se stupněm vrcholu ve struktuře.

Závěr

Tato práce byla zaměřena na prohazování hran v pseudo-triangulaci. Byly postupně navrhnuty a implementovány dva algoritmy prohazování hran. První algoritmus, v textu označovaný jako *brute-force* algoritmus, dosahoval požadovaných výsledků, ale časová složitost tohoto algoritmu byla $O(k^3)$, pro k rovno počtu vrcholů v sousedních pseudo-trojúhelnících. Proto bylo navrženo zjednodušení, jež urychlovalo hledání nové hranice v sousedních pseudo-trojúhelnících. Pro vykonání operace prohazování hran v existujícím programu [Trčka07] byly zvoleny dvě cesty: 1) Prohazování hran na již vytvořené struktuře, tzv. *post-processing*. 2) Prohazování hran bylo začleněno do inkrementálního algoritmu konstrukce pseudo-triangulace navrženého v práci [Trčka07].

Podařilo se ukázat, že po prohazování hran v pseudo-triangulaci jsou výsledné struktury kvalitnější jak „na oko“, tak i podle úhlových měření v pseudo-trojúhelnících (v porovnání s pseudo-triangulacemi vytvářenými v původní práci [Trčka07]). Přičemž volba kritéria pro výběr nových hranic měla zásadní vliv na kvalitu pseudo-trojúhelníků.

Během implementace řešení se také přišlo na nedostatky navrhovaných algoritmů. Hlavním nedostatkem je omezená funkčnost zjednodušeného algoritmu prohazování hran, pokud pseudo-triangulace není minimální.

Literatura

[Kolinger02] KOLINGEROVÁ, I. – ŽALIK, B. Improvements to randomized incremental Delaunay insertion. In *Computer & Graphics 26(2)*, Delaunay triangulation and incremental insertion, s. 477-490.

[MathWorld] Eric Weisstein. *MathWorld* [online]. <http//www.mathworld.com>

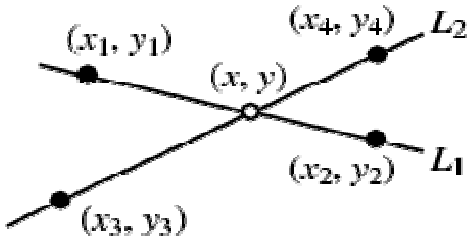
[Trčka07] TRČKA, J. *Pseudo-triangulace a jejich využití v aplikované výpočetní geometrii*. Praha, 2007. 71s. Diplomová práce na Matematicko-fyzikální fakultě Univerzity Karlovy, Kabinet software a výuky informatiky. Vedoucí diplomové práce Doc. Dr. Ing. Ivana Kolingerová.

Přílohy

A Počítání průsečíku hran

V této příloze je uveden postup pro test na křížení hran, který byl původně použit v algoritmu pro výběr vnitřní diagonály. Po konzultaci s vedoucí práce byl nahrazen znaménkovým testem – kapitola 2.4.2.

Obrázek A.1 je ilustrací k výpočtu průsečíku dvou hran. Nechť přímky L_1 a L_2 jsou prodloužením testované diagonály a jedné z hran polygonu. Nechť body (x_1, y_1), (x_2, y_2), (x_3, y_3) a (x_4, y_4) jsou vrcholy polygonu. Potom souřadnice průsečíku (x, y) se vypočítají podle vzorce A.2, který je k dispozici na stránkách [MathWorld].

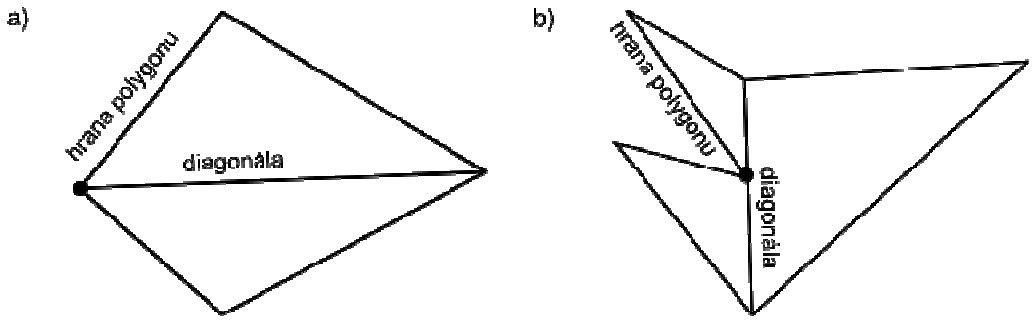


Obrázek A.1. Ilustrace k výpočtu průsečíku dvou přímek.

$$x = \frac{\begin{vmatrix} \begin{vmatrix} x1 & y1 \\ x2 & y2 \end{vmatrix} & x1-x2 \\ \begin{vmatrix} x3 & y3 \\ x4 & y4 \end{vmatrix} & x3-x4 \end{vmatrix}}{\begin{vmatrix} x1-x2 & y1-y2 \\ x3-x4 & y3-y4 \end{vmatrix}} \qquad y = \frac{\begin{vmatrix} \begin{vmatrix} x1 & y1 \\ x2 & y2 \end{vmatrix} & y1-y2 \\ \begin{vmatrix} x3 & y3 \\ x4 & y4 \end{vmatrix} & y3-y4 \end{vmatrix}}{\begin{vmatrix} x1-x2 & y1-y2 \\ x3-x4 & y3-y4 \end{vmatrix}}$$

Vzorec A.2.Souřadnice průsečíku dvou přímek.

Pokud se průsečík (x,y) nachází mimo úsečku $(x_1, y_1)(x_2, y_2)$ nebo mimo úsečku $(x_3, y_3)(x_4, y_4)$, hrana polygonu se s diagonálou nekříží. V případě, že průsečík leží uvnitř úsečky $(x_1, y_1)(x_2, y_2)$ a zároveň uvnitř úsečky $(x_3, y_3)(x_4, y_4)$, hrana polygonu se s diagonálou kříží, a diagonála bude zamítnuta. Poslední možnost, kdy průsečík vyjde v jednom z vrcholů (xi, yi). Pokud se tak stane, testuje se, zda je tento vrchol společný pro obě úsečky. Pokud ano (obrázek A.2 a)), diagonála je schválena, pokud ne (obrázek A.2 b)), diagonála je zamítnuta.

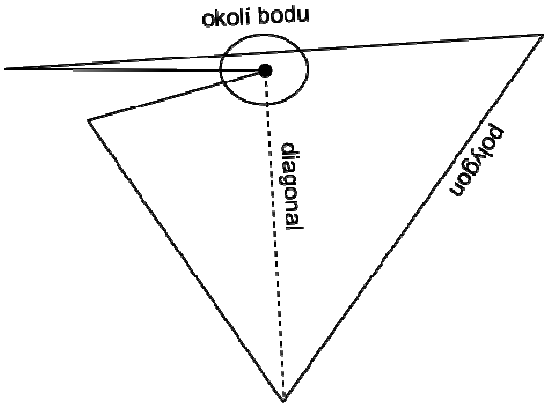


Obrázek A.3. Průsečík přímek leží ležící v krajním bodě jedné z nich

Porovnávání vrcholů

Jak známo, počítání s čísly typu *Double* není úplně přesné. V programu nám to moc nevadí, až na jednu výjimku - porovnávání vrcholů.

Dva vrcholy jsou vyhodnoceny jako totožné, pokud jsou od sebe vzdáleny méně než je hodnota určité konstanty, říkejme jí *e*. Důležité je nastavení velikosti *e*. Pokud bychom nastavili *e* příliš velké, budou se body vyhodnocovat jako totožné, i když tomu ve skutečnosti tak nebude. Naopak, pokud bude *e* příliš malé, budou se body vyhodnocovat jako různé, i když budou ve skutečnosti stejné. V našem programu je porovnávání vrcholu implementováno ve třídě *flip.VertexComparator2*. Konstanta *e* je nastavena na *10^(-d-3)*, kde *d* je největší počet desetinných míst v souřadnicích vstupních bodů[26]. Na obrázku A3. je příklad, kdy bude diagonála algoritmem zamítnuta, neboť průsečík s jednou z hran polygonu bude, díky nepřesnostem ve výpočtu ležet i na diagonále[27].



Obrázek A.4. Chyba při porovnávání bodů.

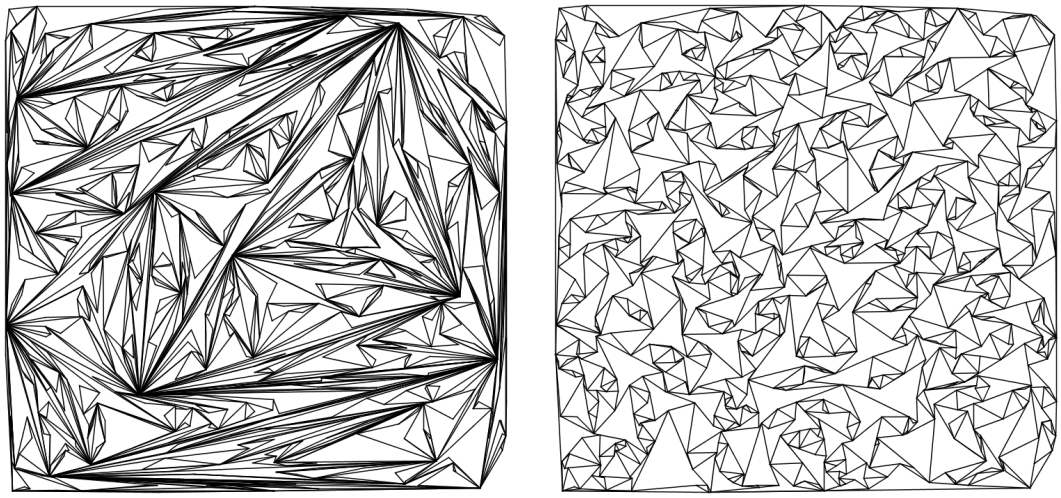
Pro přesné počítání s čísly nabízí knihovny jazyka Java třídu *java.math.BigDecimal*. Bohužel počítání s čísly *BigDecimal* je časově mnohem náročnější než počítání s čísly *Double*. Náš algoritmus navíc pracuje s již zmiňovanou složitostí $O(k^3)$, proto jsme zvolili rychlejší a méně přesné počítání s čísly typu *Double*.

[26] Tento postup bude fungovat pouze za předpokladu, že budou hodnoty mít vstupních dat stejný počet desetinných míst.
[27] Toto „selhání výpočtu“ se projevuje velmi zřídka. Algoritmus prohodí v okolí takového pseudo-trojúhelníku jiné hrany a tento problém „zmizí“.

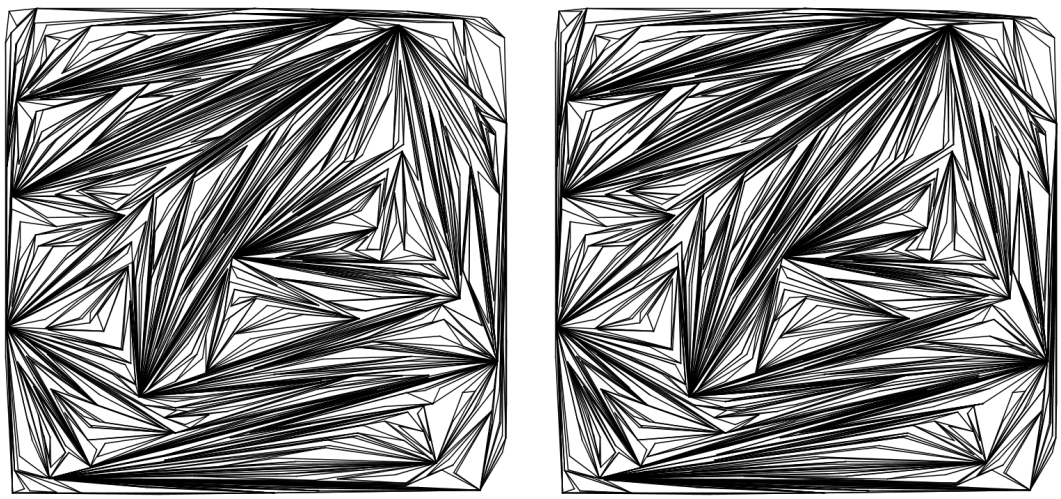
B Kritéria pro výběr hranice

Dvojice pseudo-triangulací vytvářených ze vstupního souboru *./data/original/1000.txt*

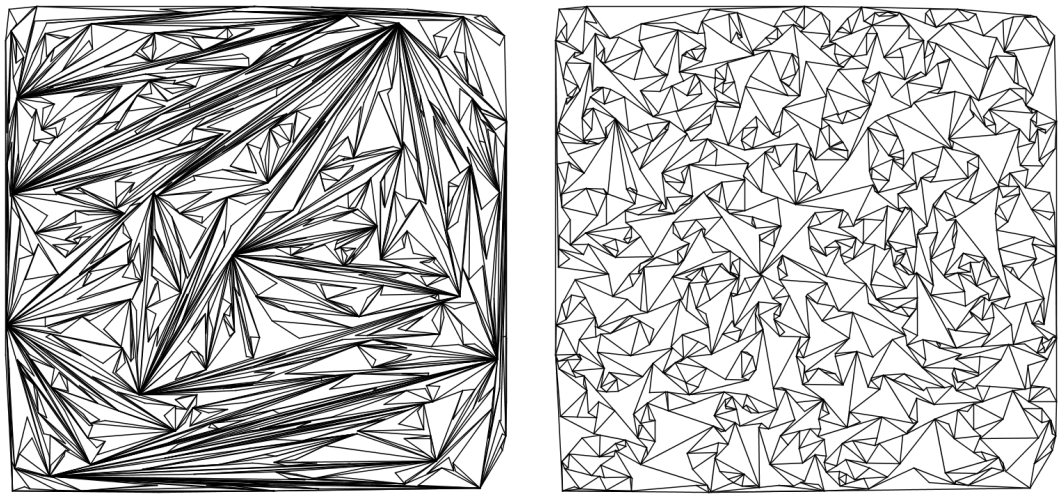
- Nalevo v obrázcích je inkrementální pseudo-triangulace vytvořená bez prohazování hran pro různá kritéria. [Trčka07]
- Napravo v obrázku je inkrementální pseudo-triangulace s prohazováním hran, tj. pseudo-triangulace byla vytvořena algoritmem inkrementálního vkládání s prohazováním hran pro různá kritéria.



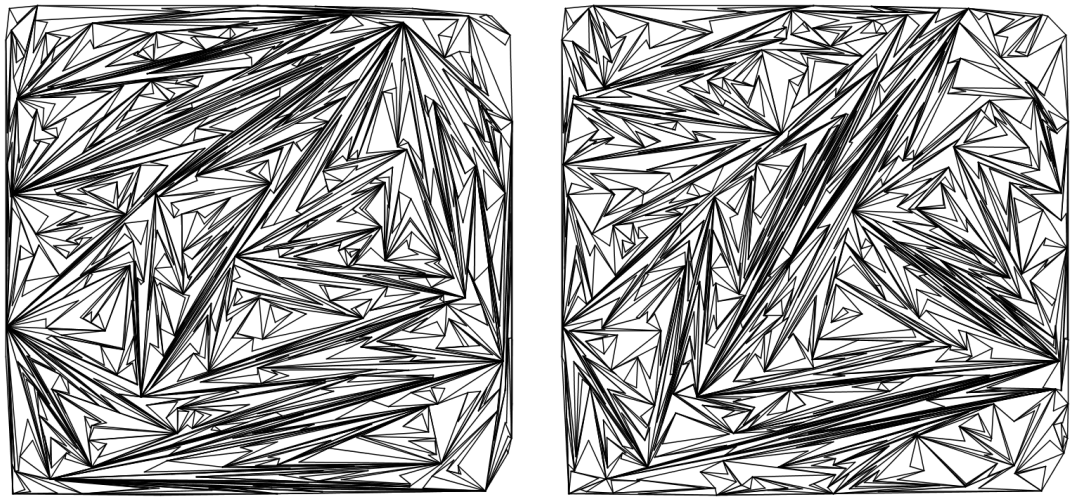
B.1. MinLength.



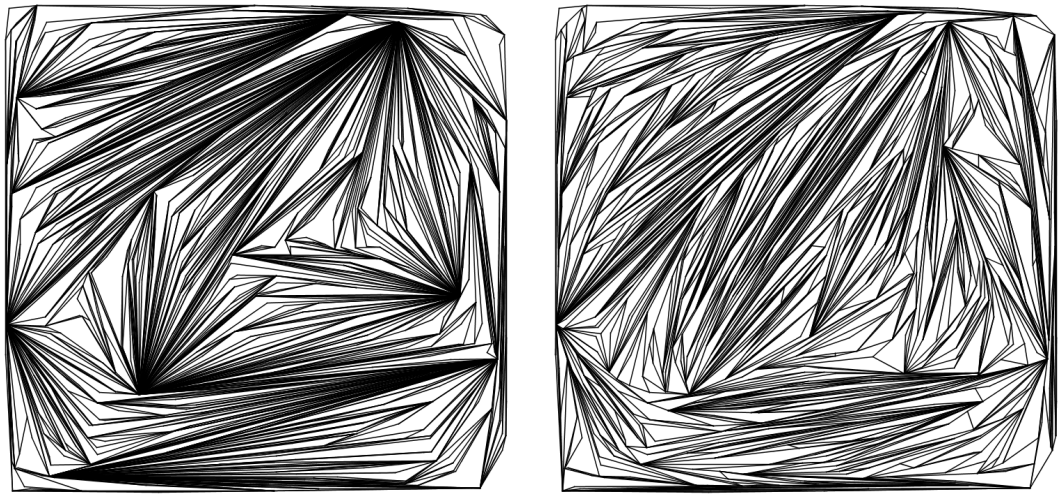
B.2. MaxLength.



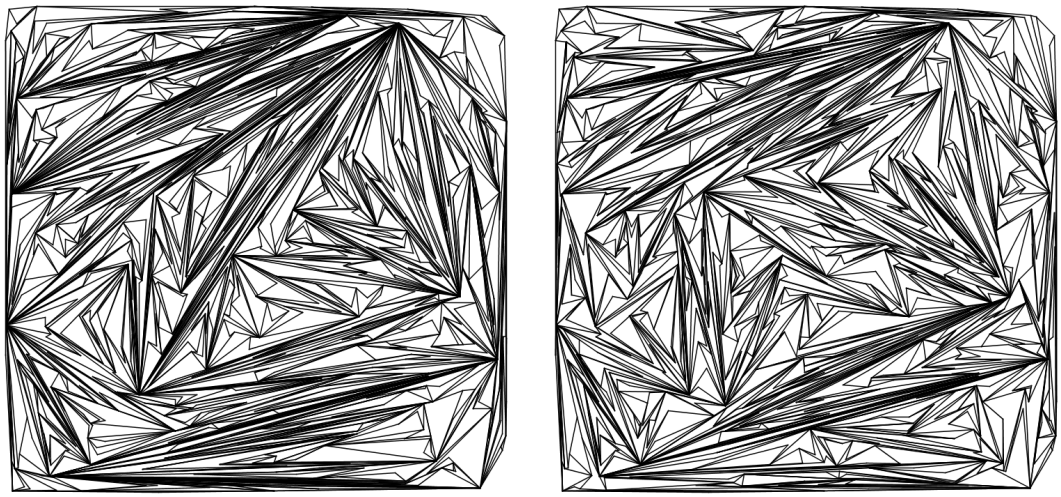
B.3. MaxMinAnlge.



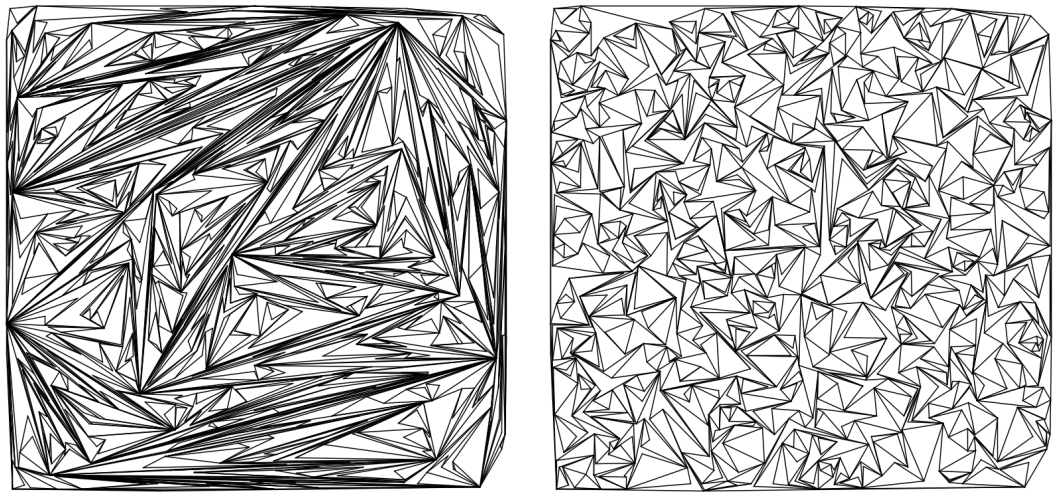
B.4 MinMaxAngle.



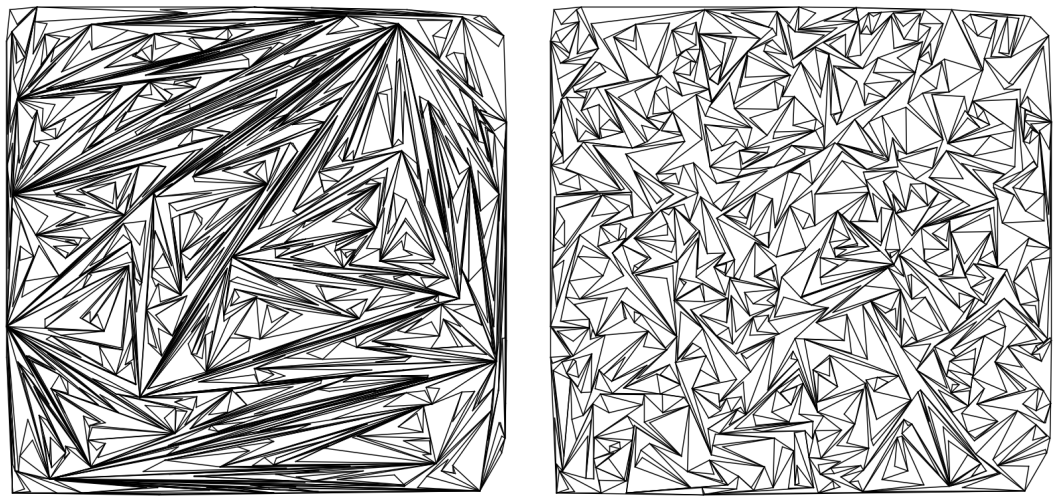
B.5 MaxMinSumAngle.



B.6. MinMaxSumAngle.



B.7. MaxMinAngleHull.



B.8. MinMaxAngleHull.

C Úhlové testy pro rozložení bodů gaus a cluster

V této příloze jsou uvedeny úhlové testy z kapitoly 4.2 pro množiny *gaus 10000* a *cluster 10000*.

Výsledky testů na kvalitu pseudo-trojúhelníků na množinách *gaus 10000* a *cluster 10000* jsou podobné jako na množině *uniform 10000*, viz kapitola 4.2.

Tabulka C.1 zaznamenává informace o velikosti maximálního a minimálního úhlu ve stupních pro množinu *gaus 10000*. Můžeme zde vidět, že průměrné velikosti min. a max. úhlů vychází velmi podobně jako pro množinu bodů s rovnoměrným rozložením *uniform*, viz kapitola 4.2.

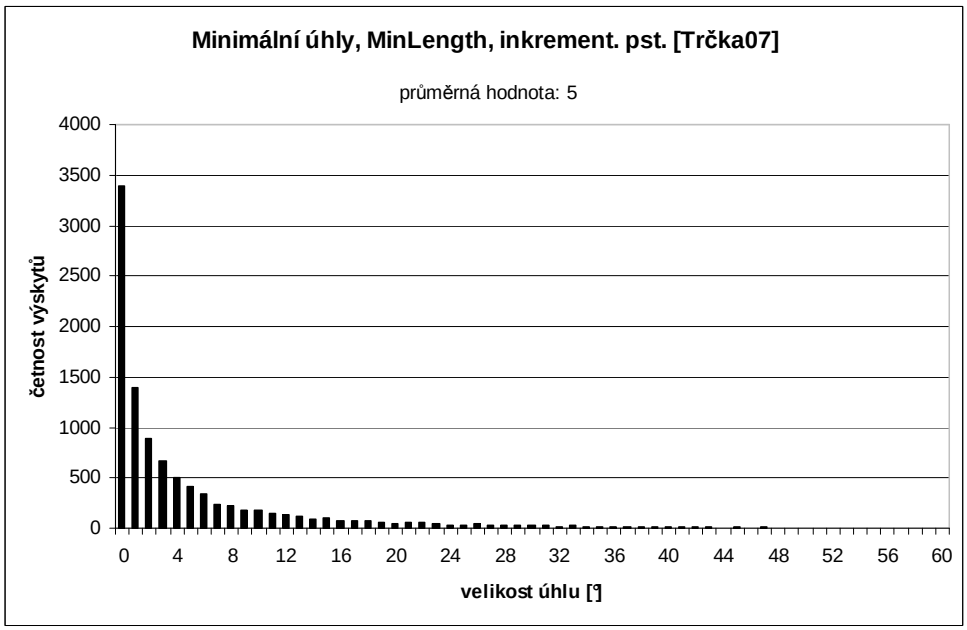
Tabulka C.2 zaznamenává informace o velikosti maximálního a minimálního úhlu ve stupních pro množinu *cluster 10000*.

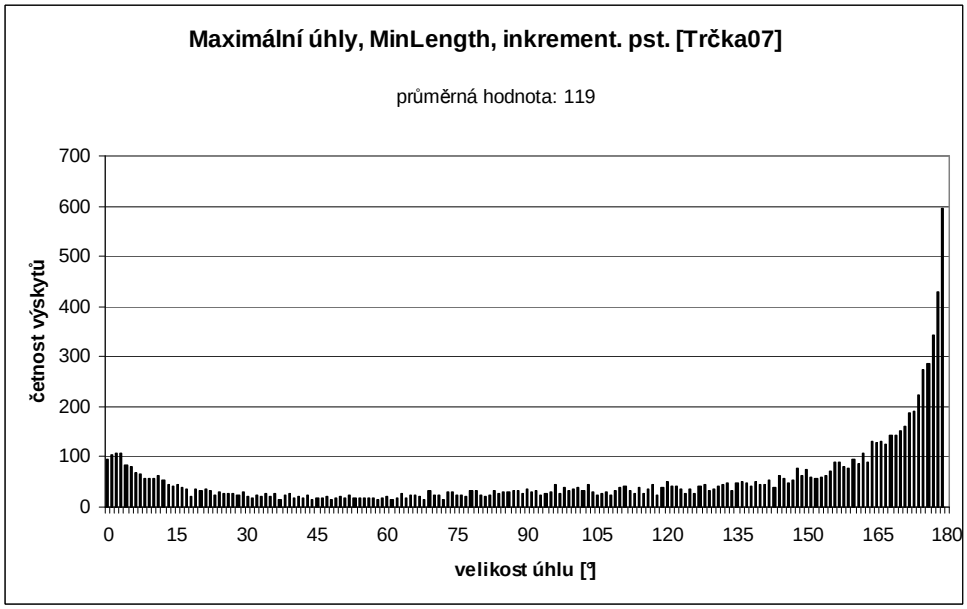
Průměrné hodnoty min. a max. úhlu, gaus 10000 [°]						
kritérium	bez swapu		post-proc. swap		inkr. swap	
	min.	max.	min.	max.	min.	max.
MinLength	5	119	22	88	22	89
MaxMinAngle	4	108	14	96	23	80
MinMaxAngle	4	39	4	39	4	32

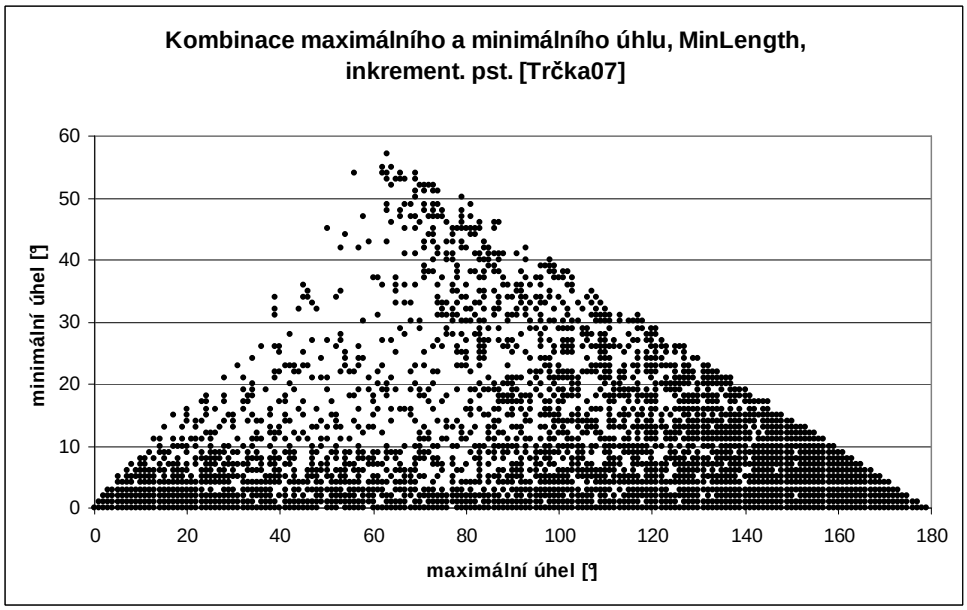
Tabulka C.1. Průměrné hodnoty max. a min. úhlu v mn. gaus 10000.

Průměrné hodnoty min. a max. úhlu, cluster 10000 [°]						
kritérium	bez swapu		post-proc. swap		inkr. swap	
	min.	max.	min.	max.	min.	max.
MinLength	5	117	22	89	22	89
MaxMinAngle	5	105	16	91	23	80
MinMaxAngle	5	41	5	35	5	34

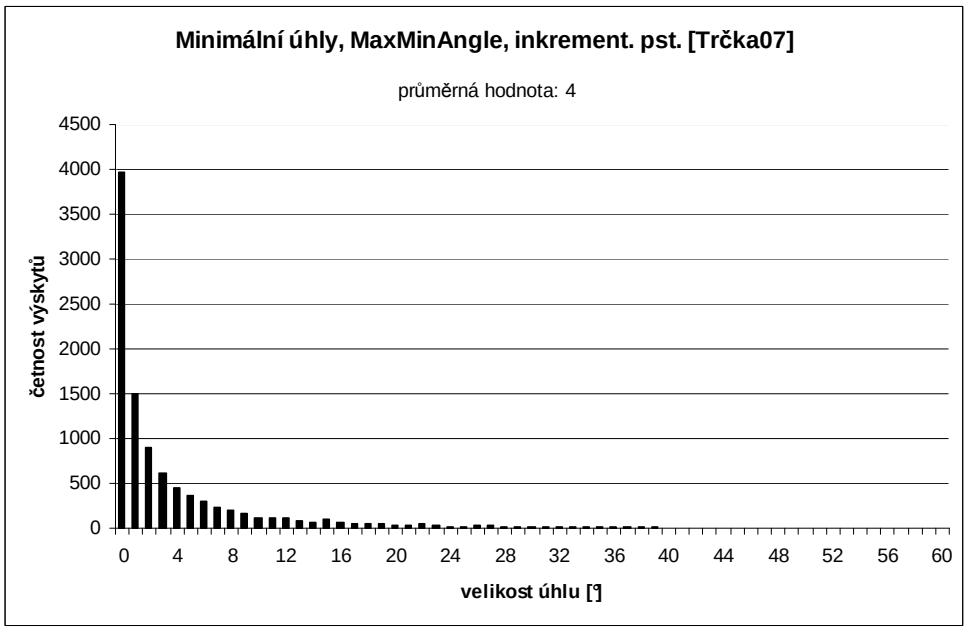
Tabulka C.2. Průměrná hodnoty max. a min. úhlu v mn. cluster 10000.

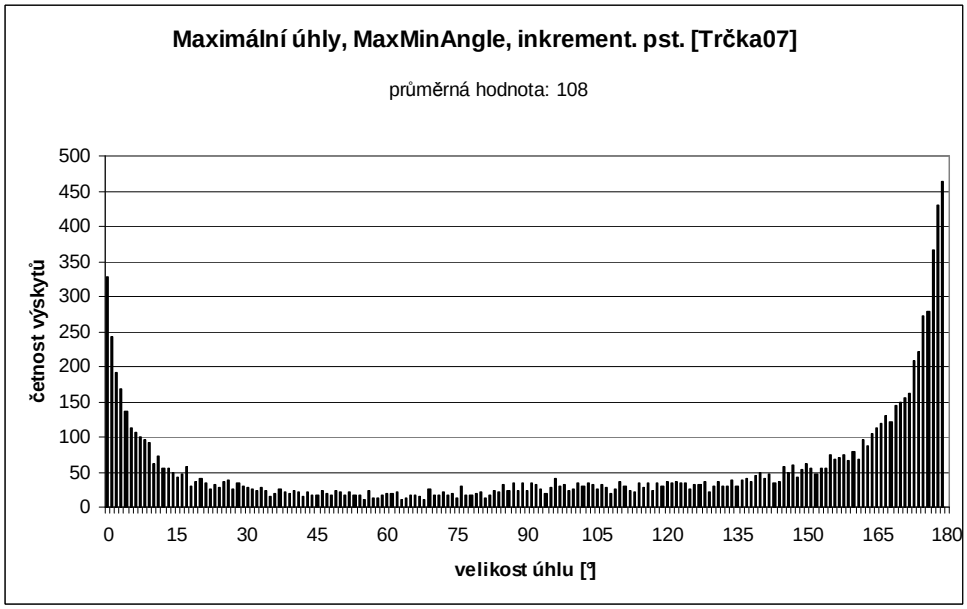


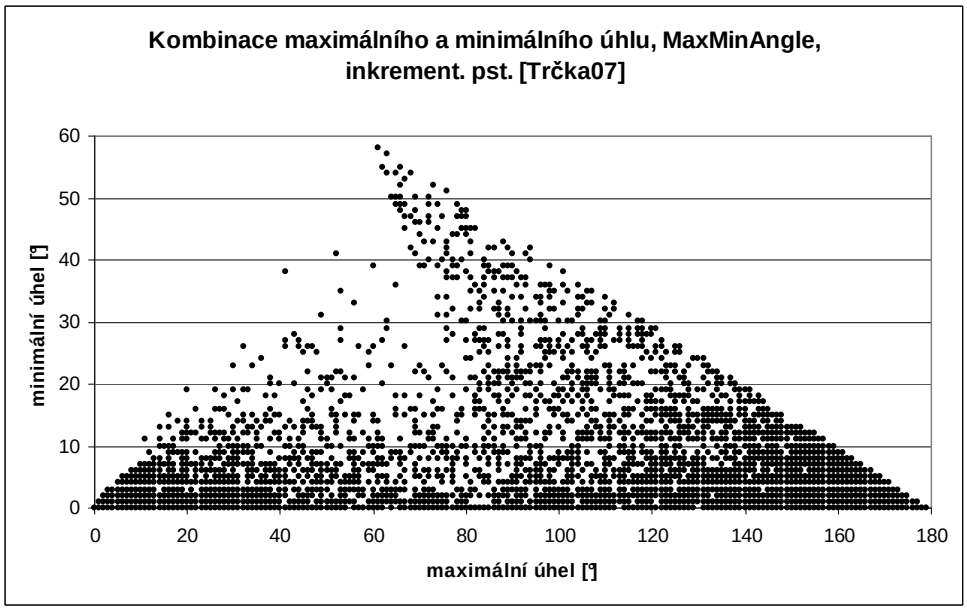




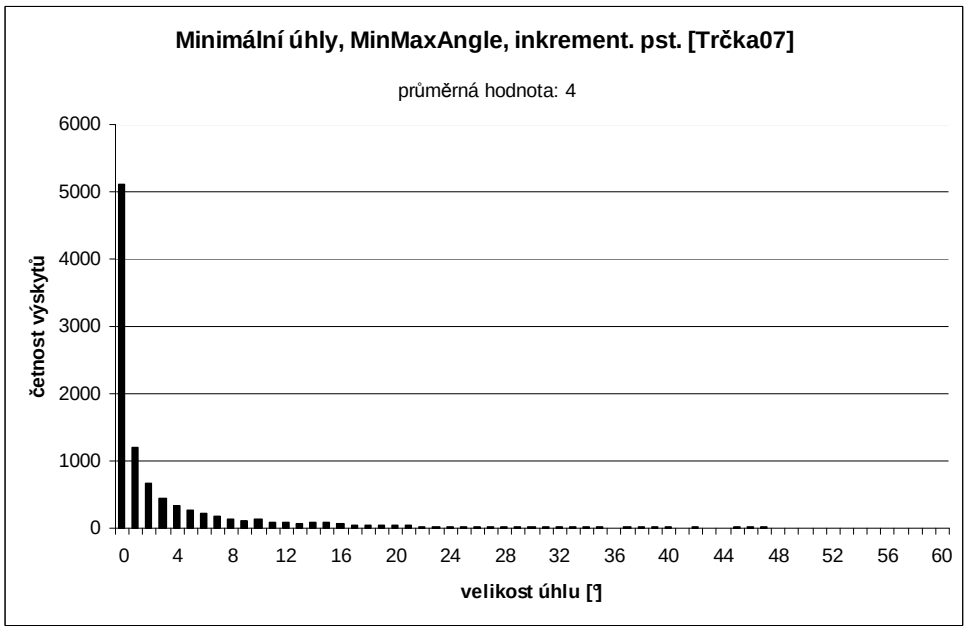
Graf C.1. Gaus 10000, MinLength, bez prohazování hran.

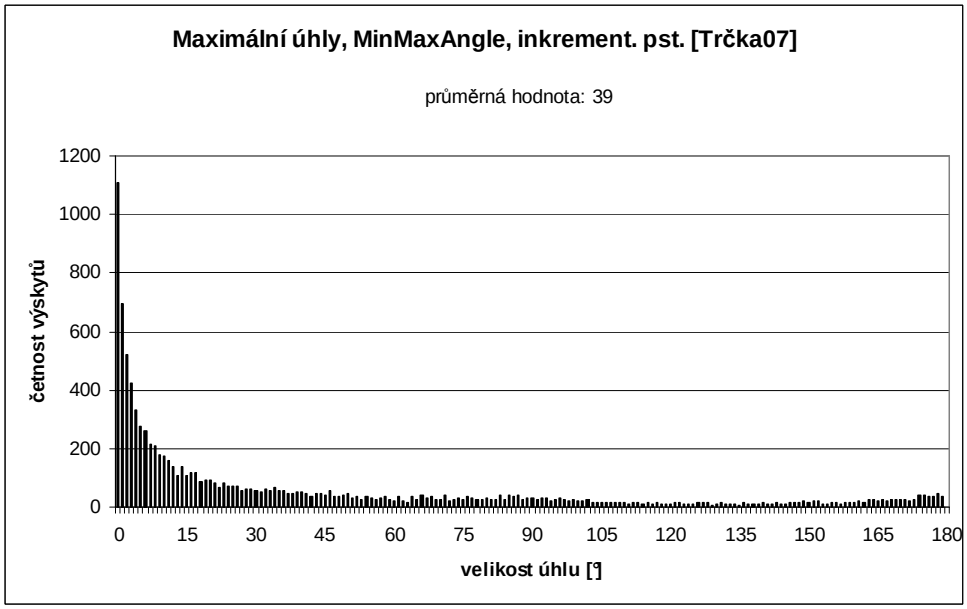


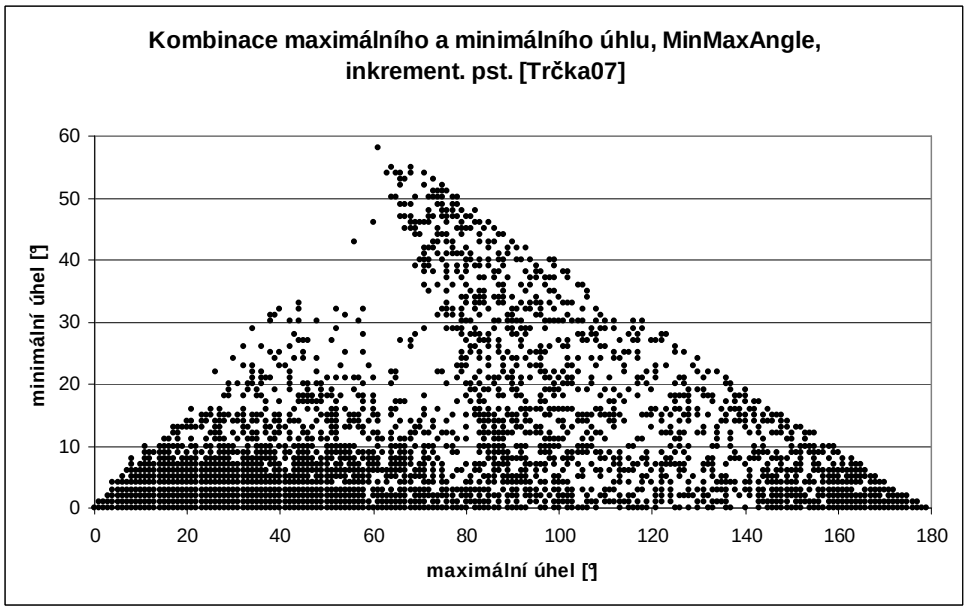




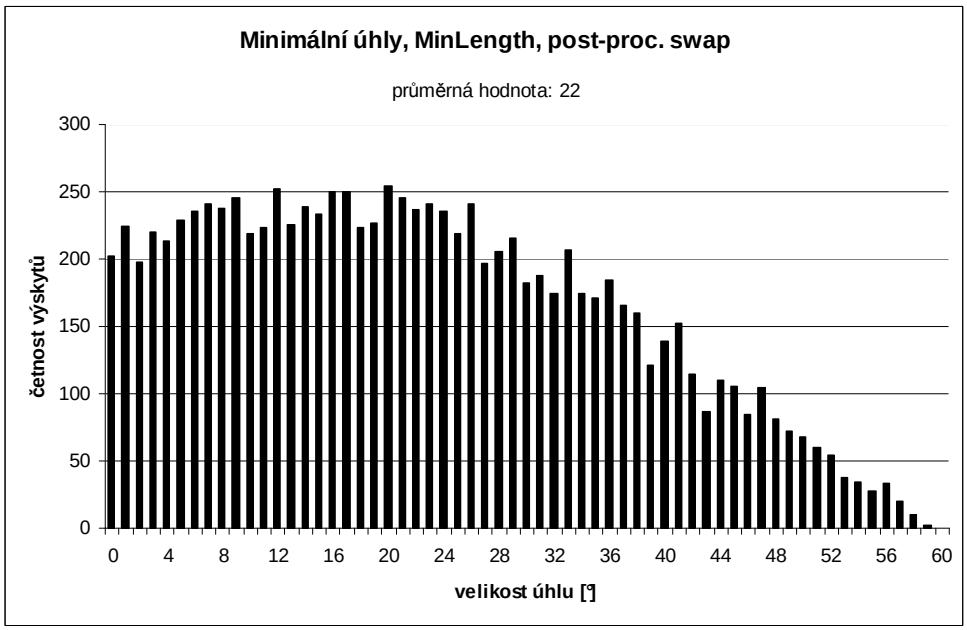
Graf C.2. Gaus 10000, MaxMinAngle, bez prohazování hran.

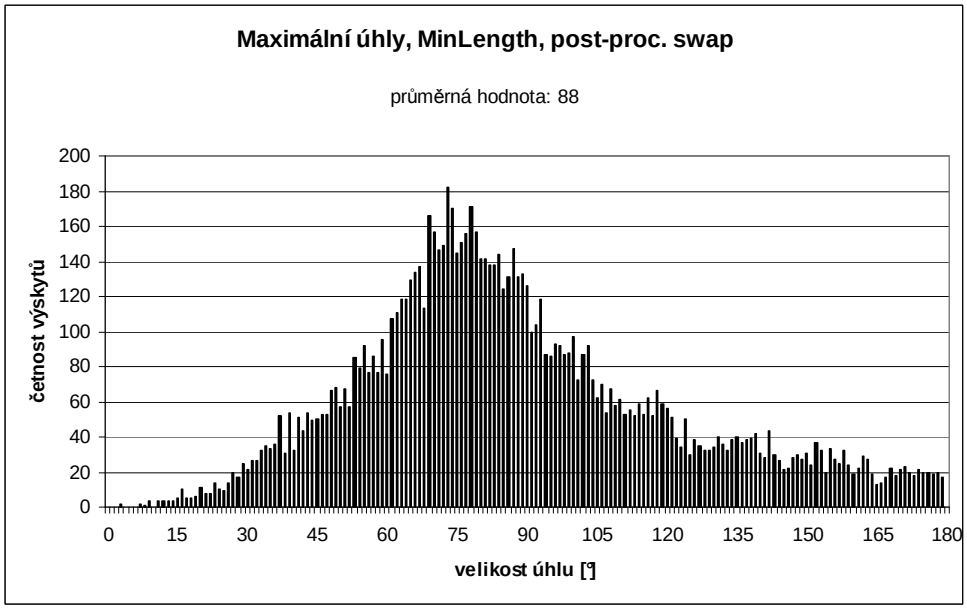


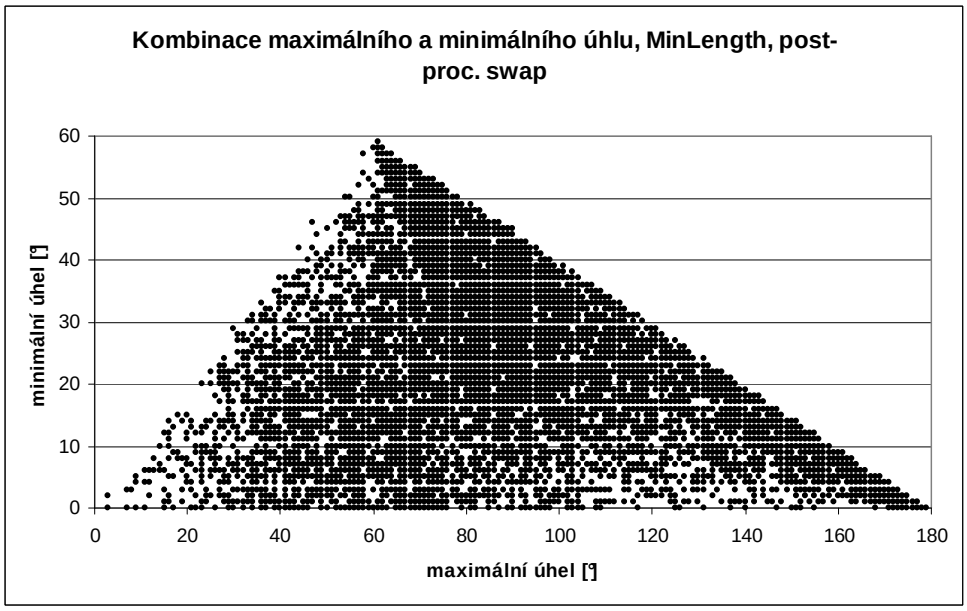




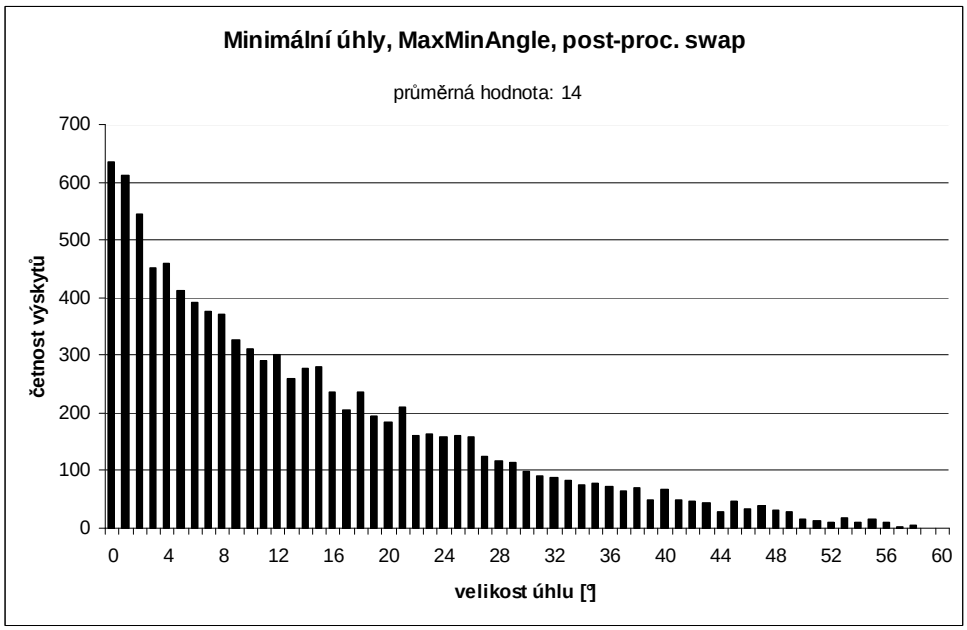
Graf C.3. Gaus 10000, MinMaxAngle, bez prohazování hran.

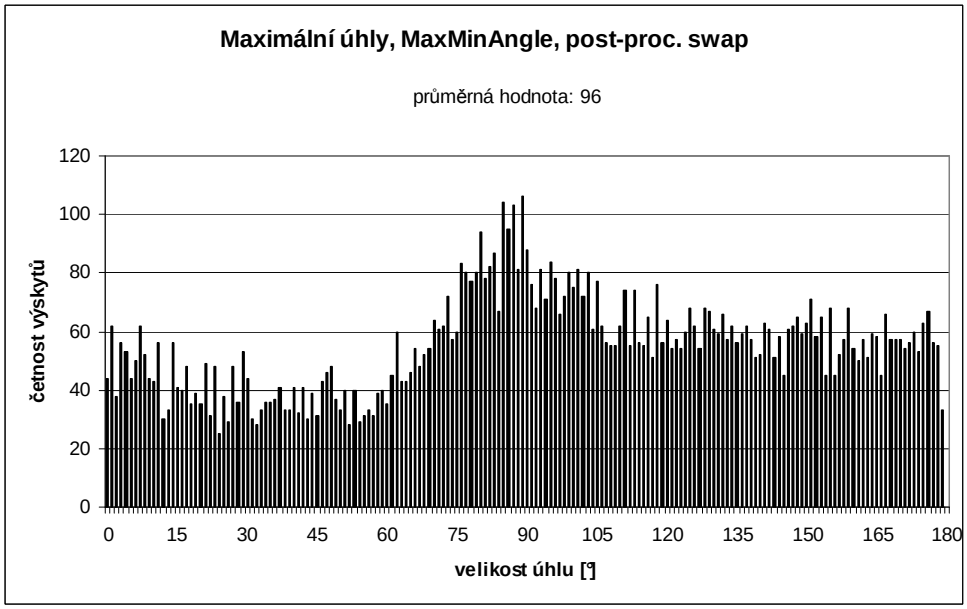


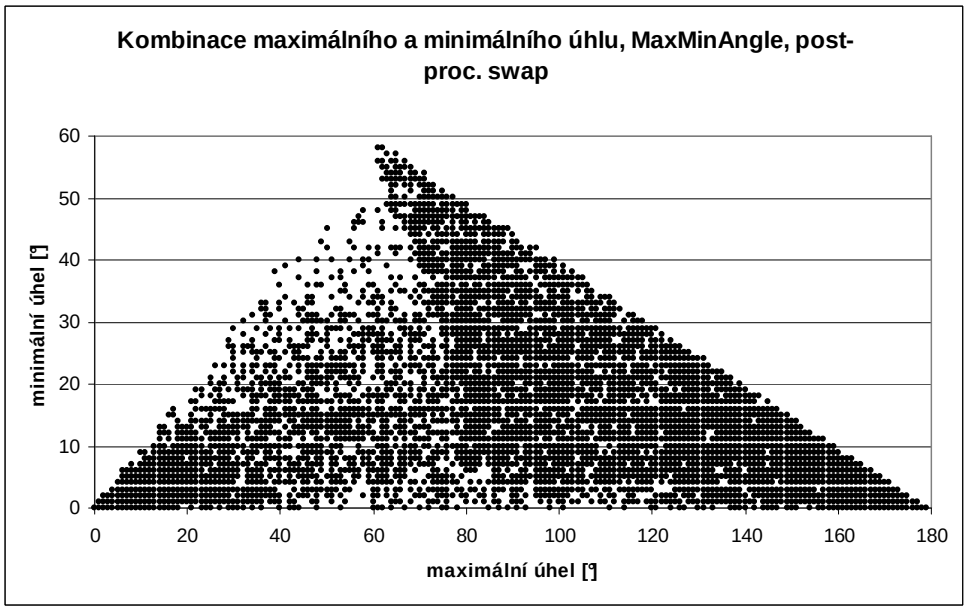




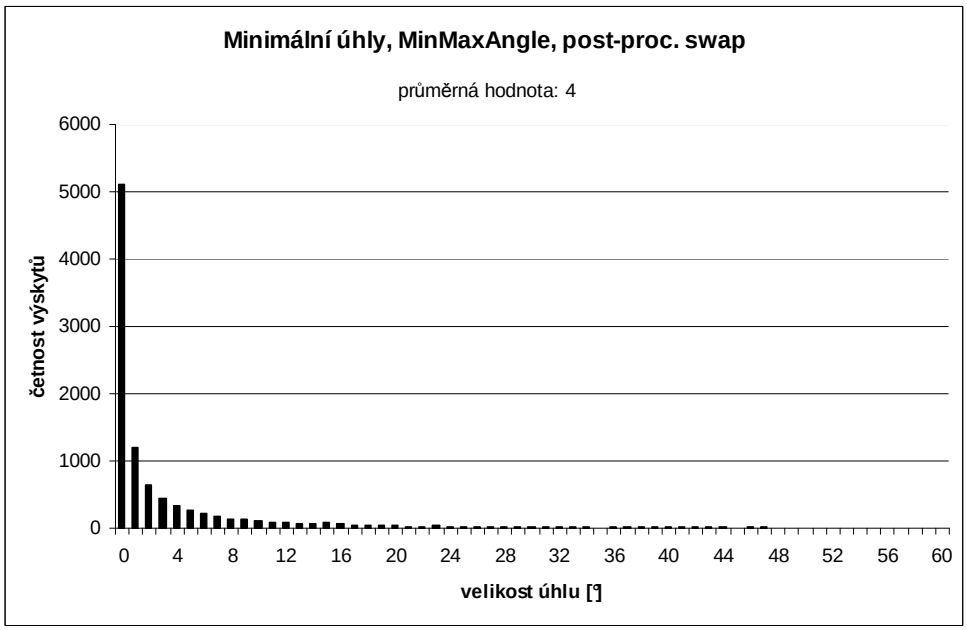
Graf C.4. Gaus 10000, MinLength, post-processing swap.

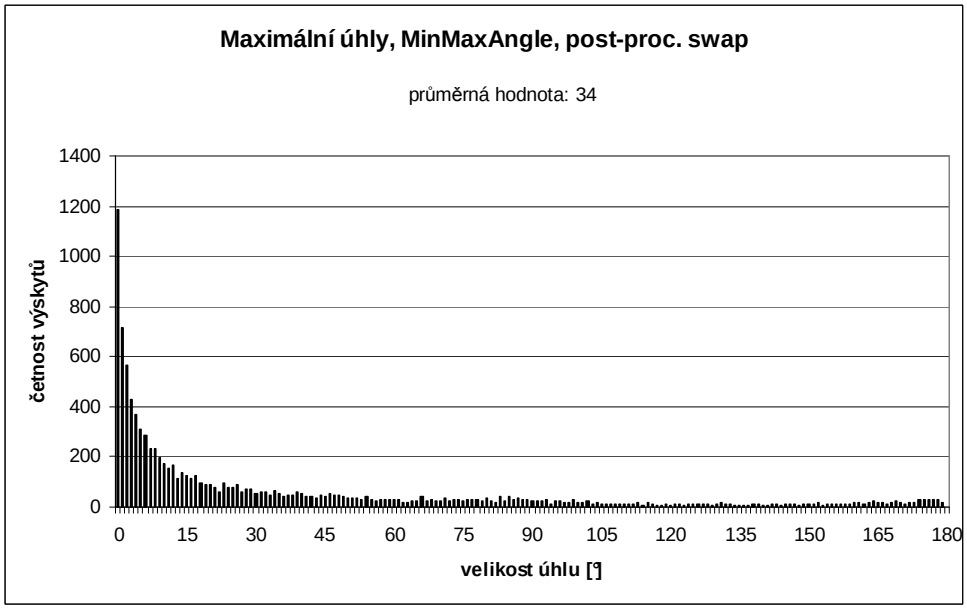


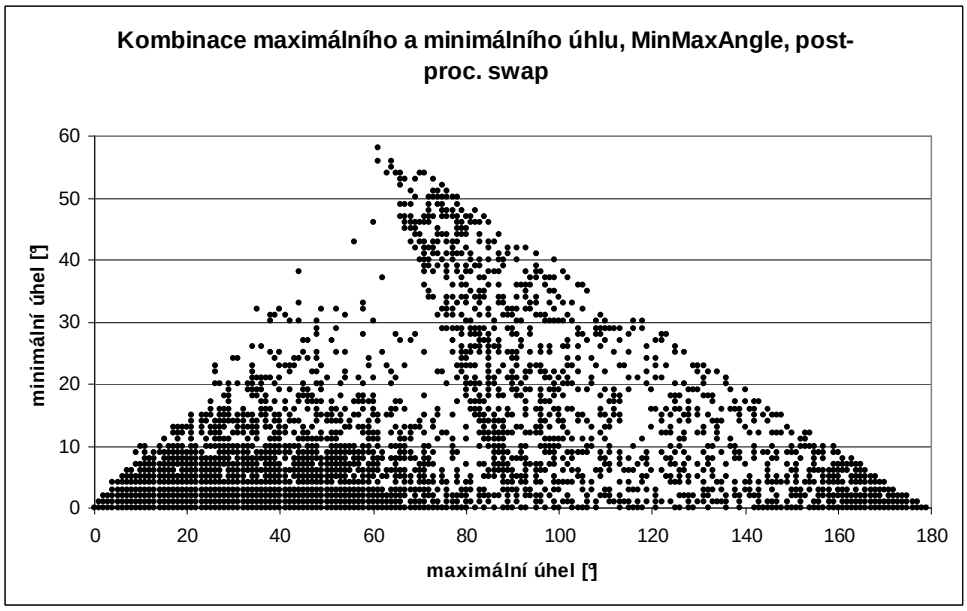




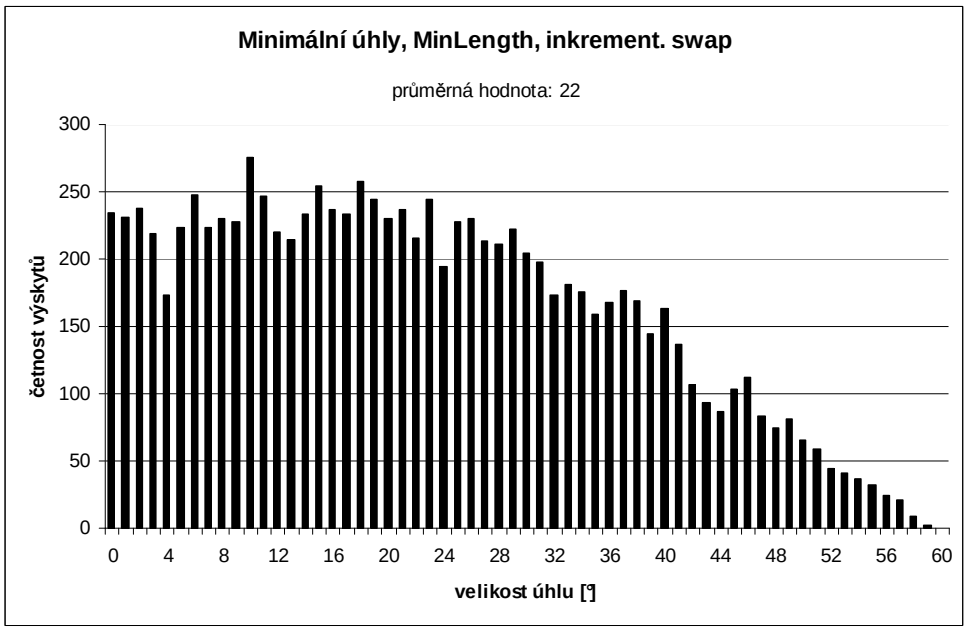
Graf C.5. Gaus 10000, MaxMinAngle, post-processing swap.

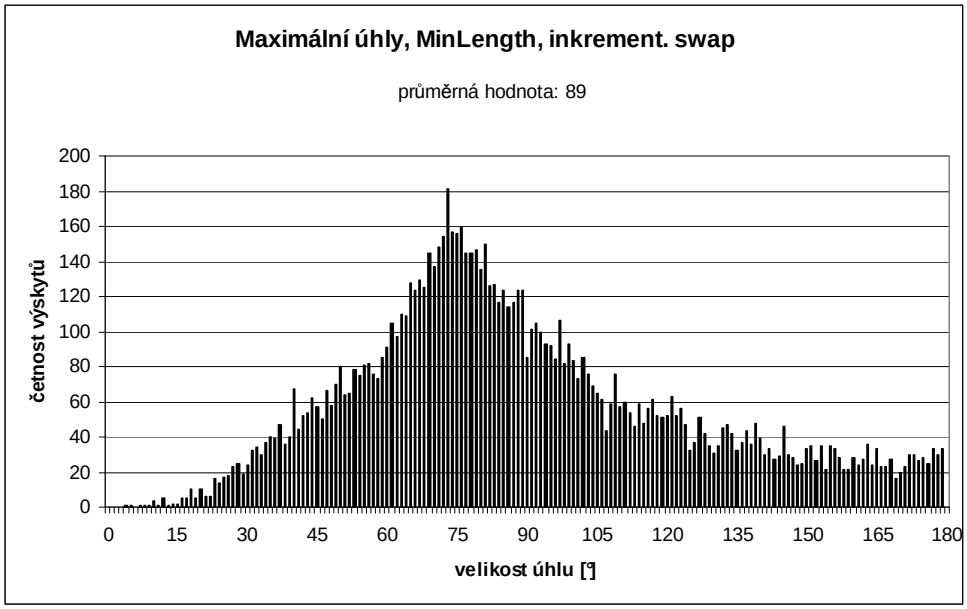


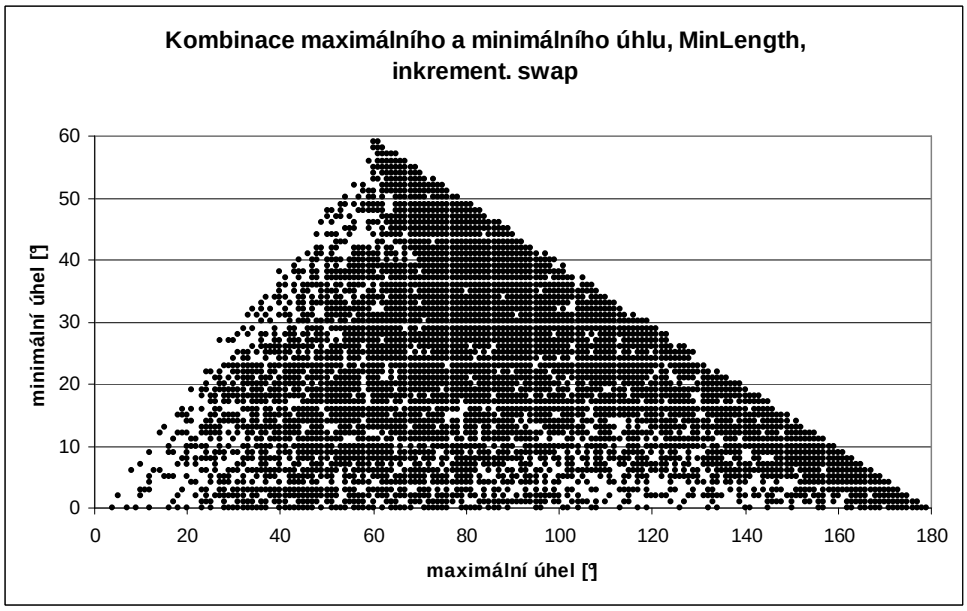




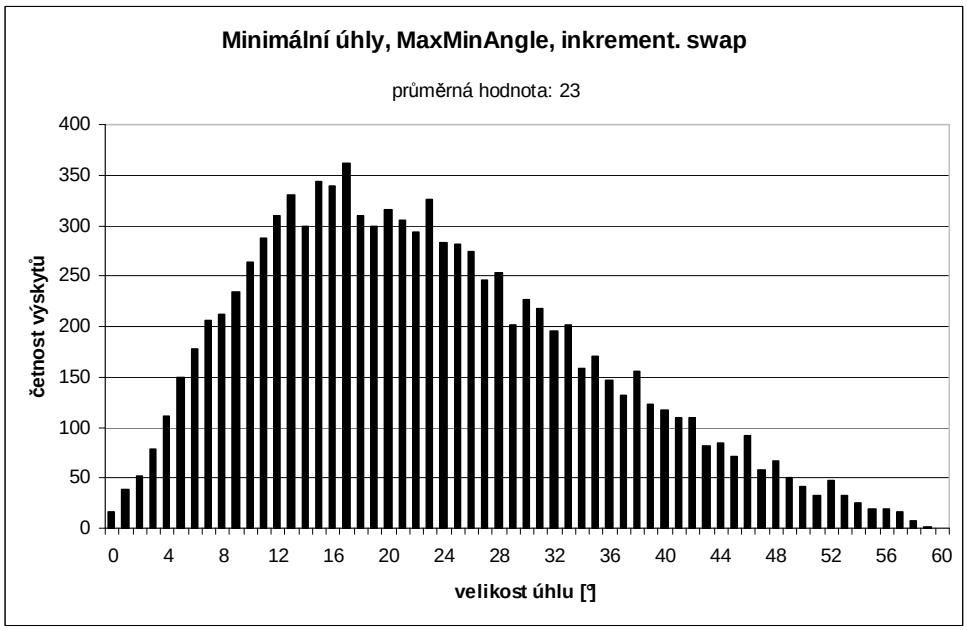
Graf C.6. Gaus 10000, MinMaxAngle, post-processing swap.

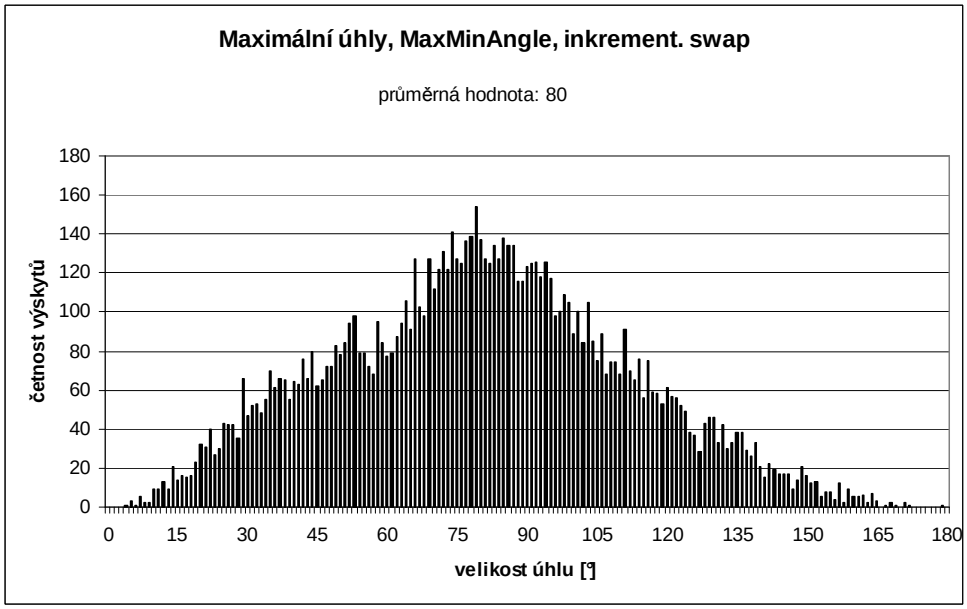


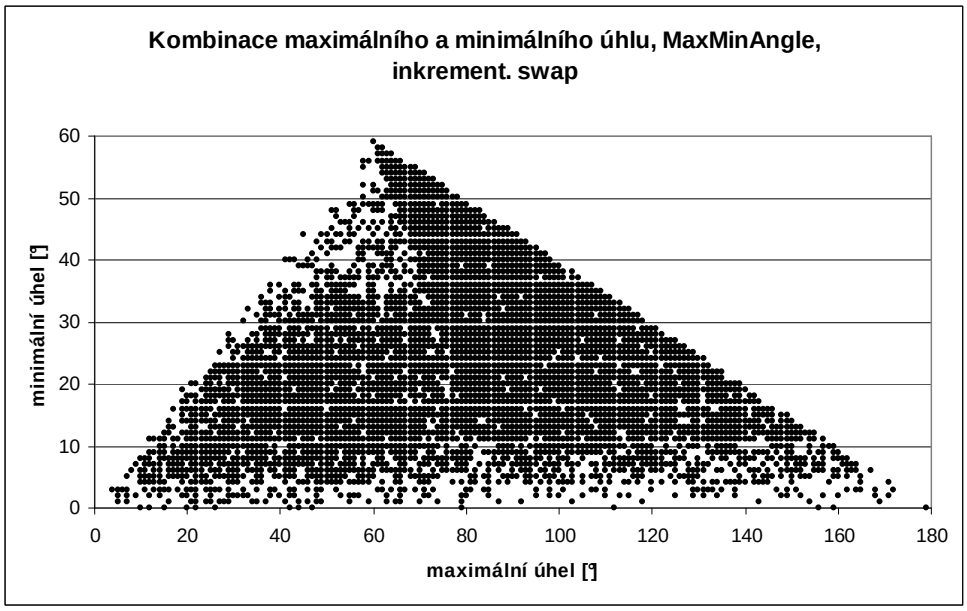




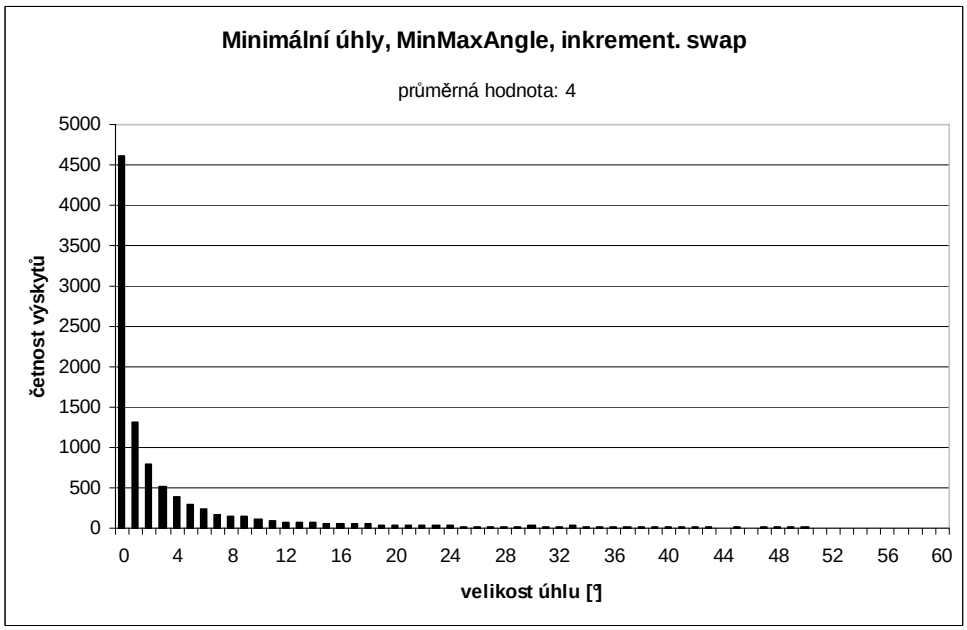
Graf C.7. Gaus 10000, MinLength, inkrementální swap.

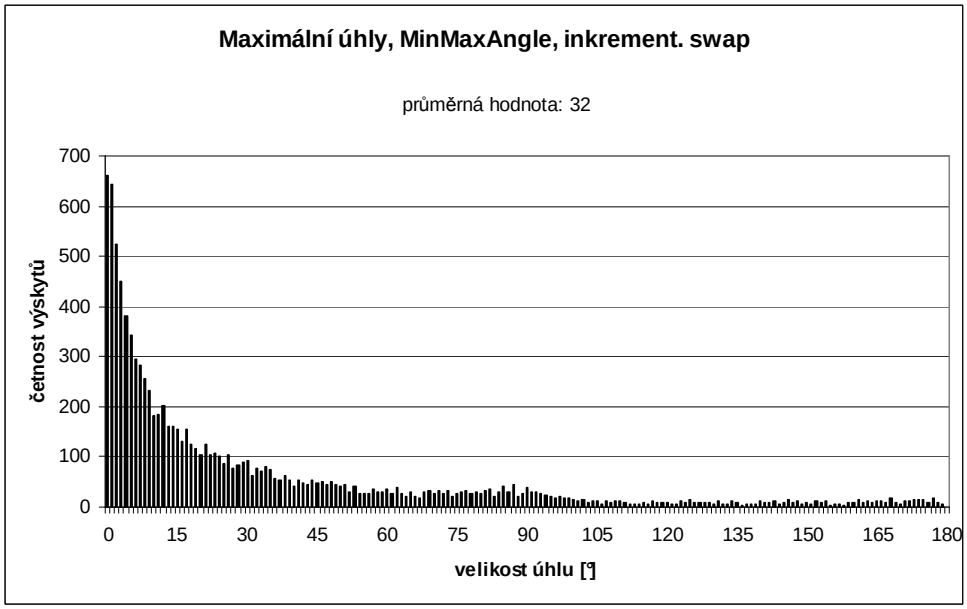


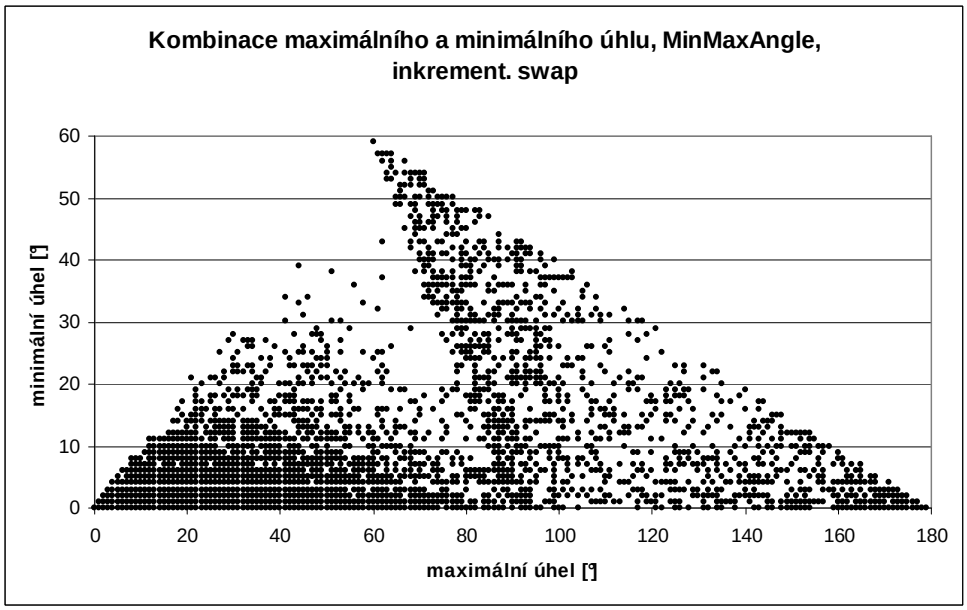




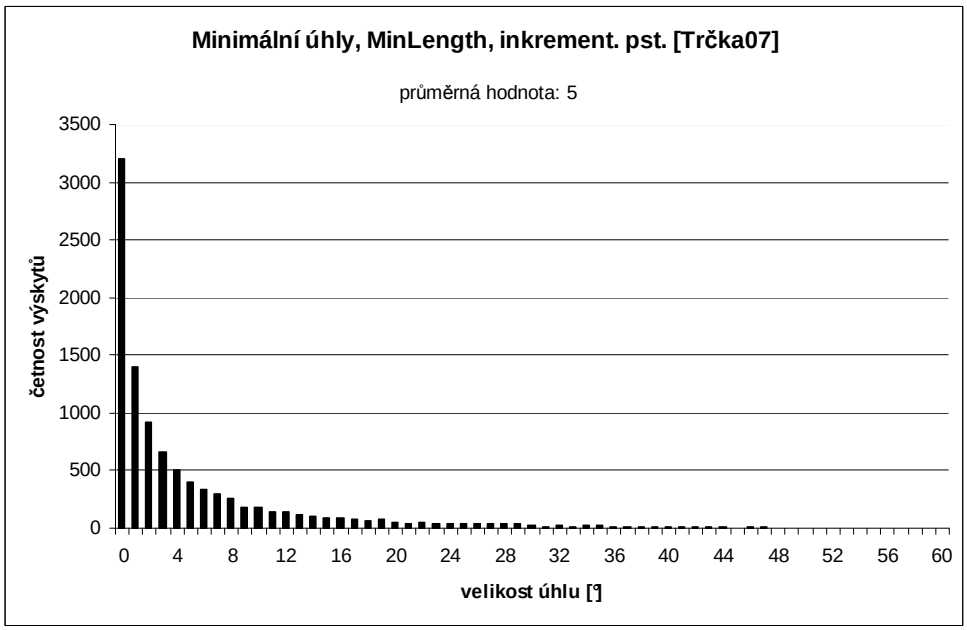
Graf C.8. Gaus 10000, MaxMinAngle, inkrementální swap.

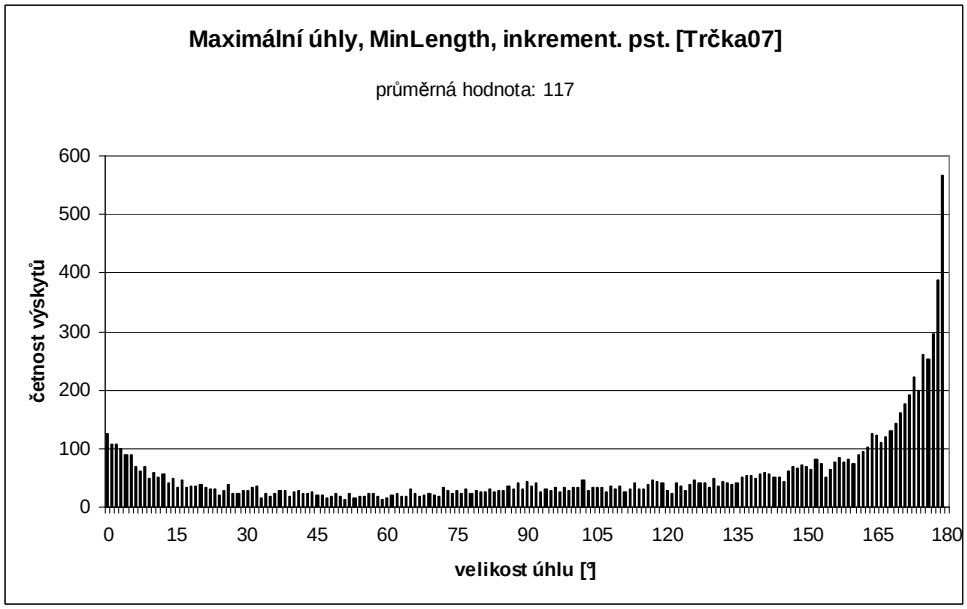


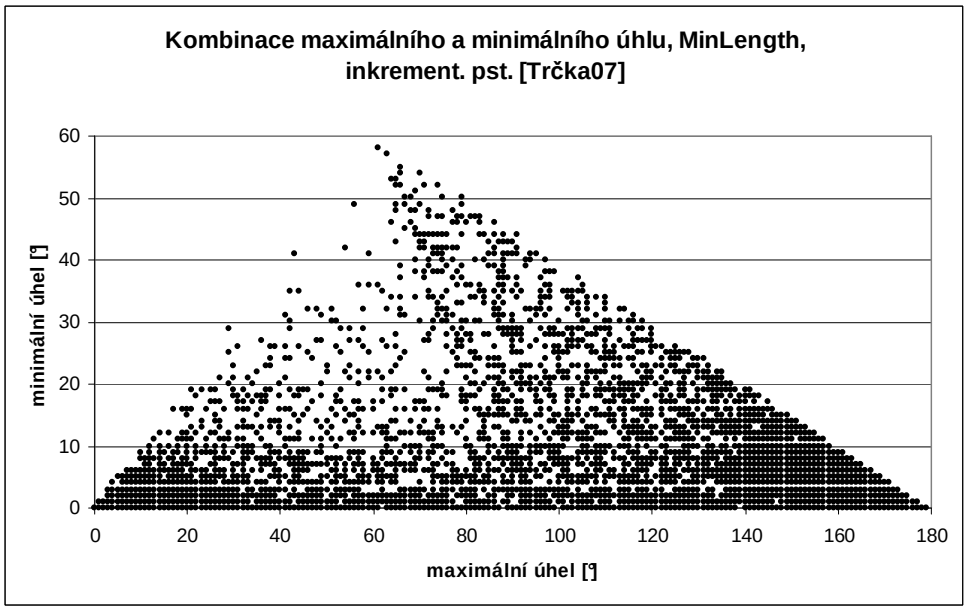




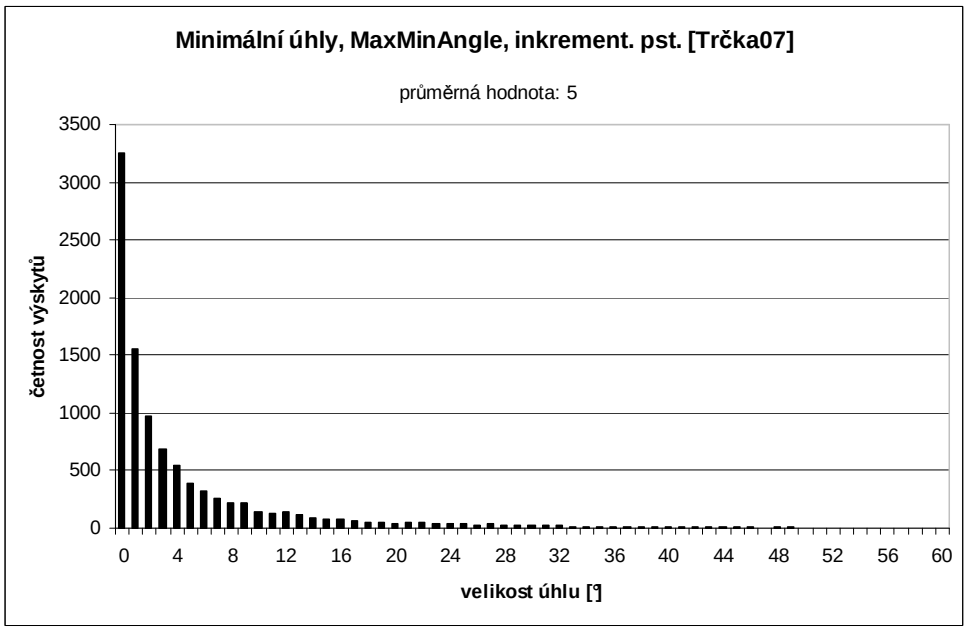
Graf C.9. Gaus 10000, MinMaxAngle, inkrementální swap.

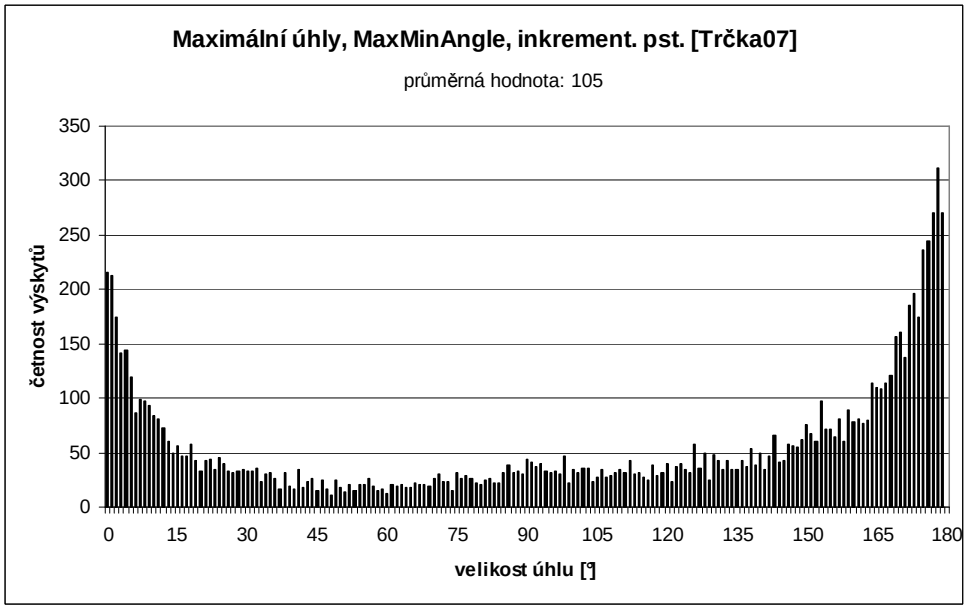


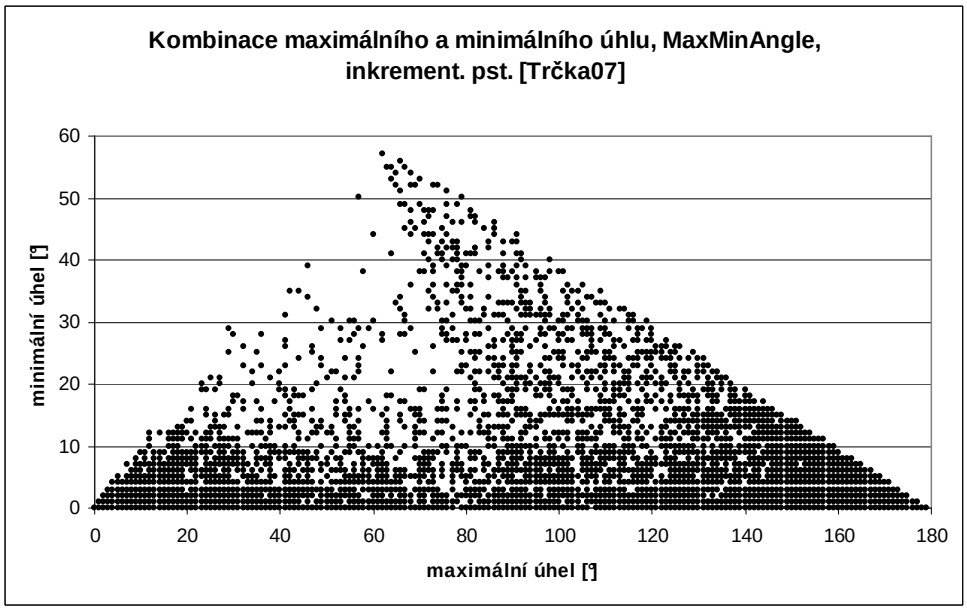




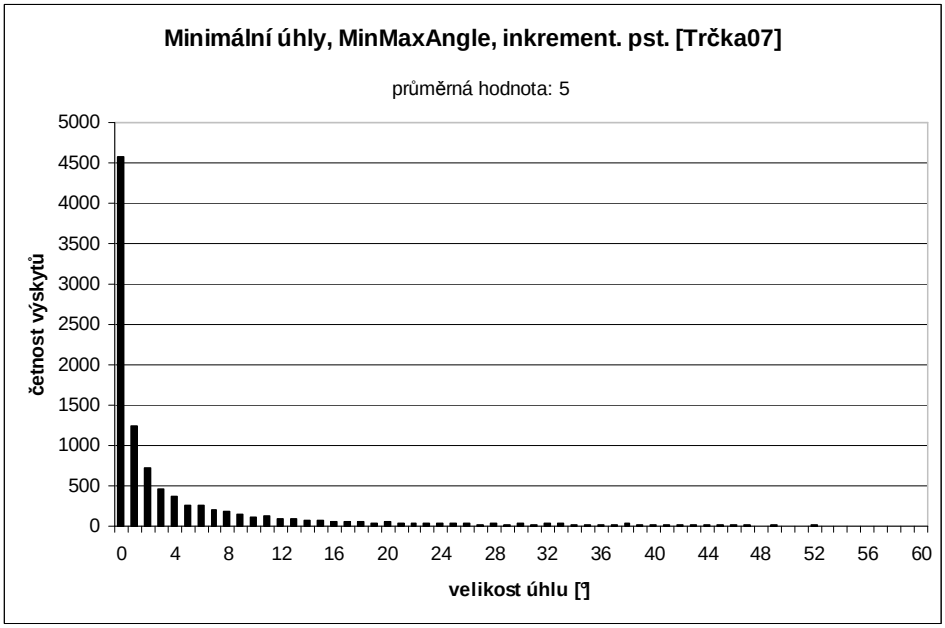
Graf C.10. Cluster 10000, MinLength, bez prohazování hran.

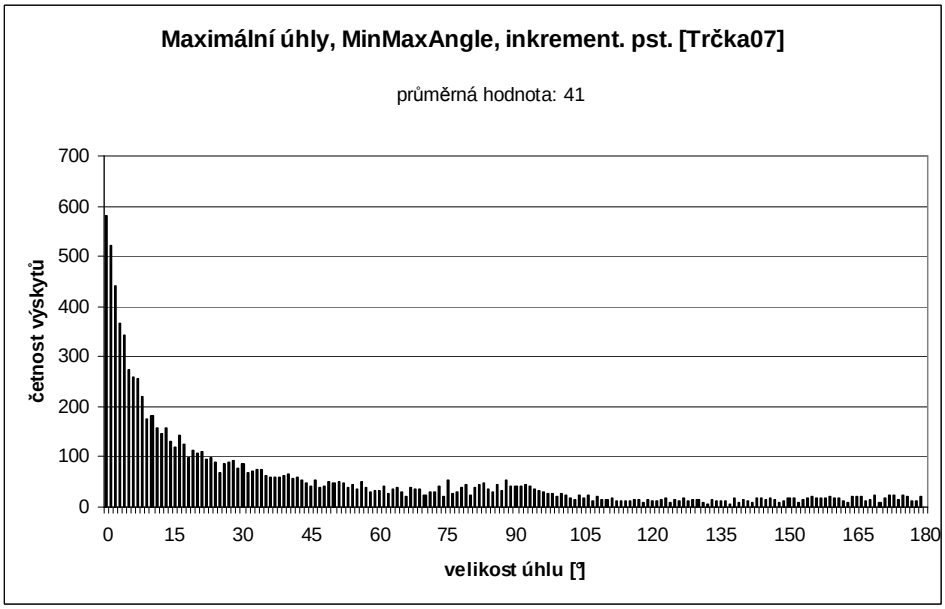


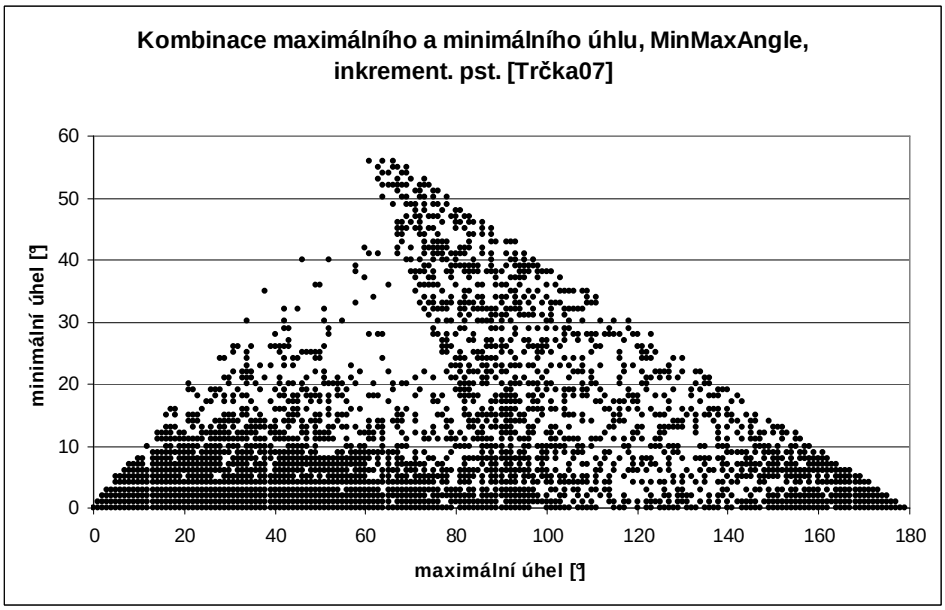




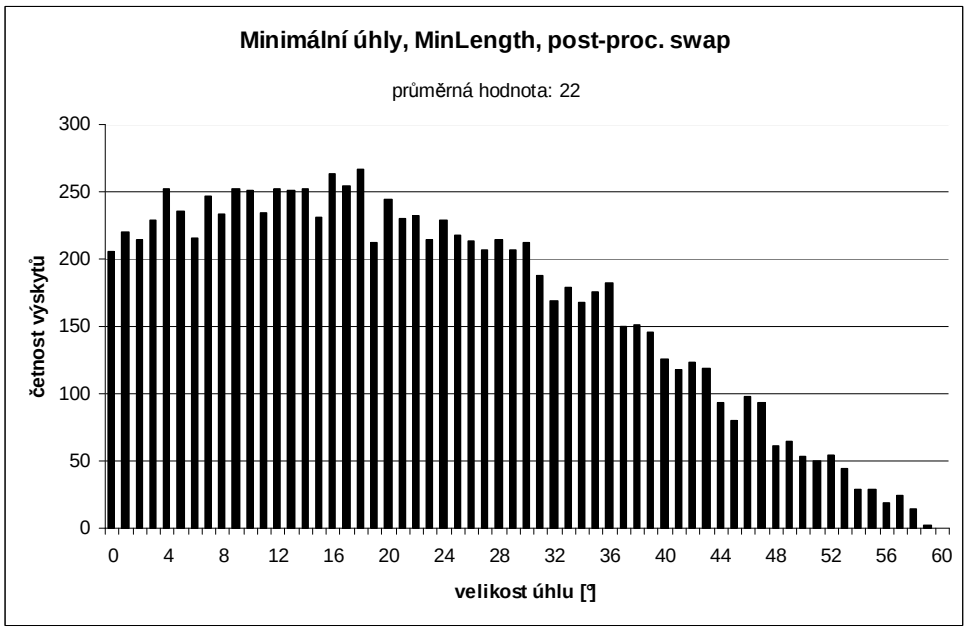
Graf C.11. Cluster 10000, MaxMinAngle, bez prohazování hran.

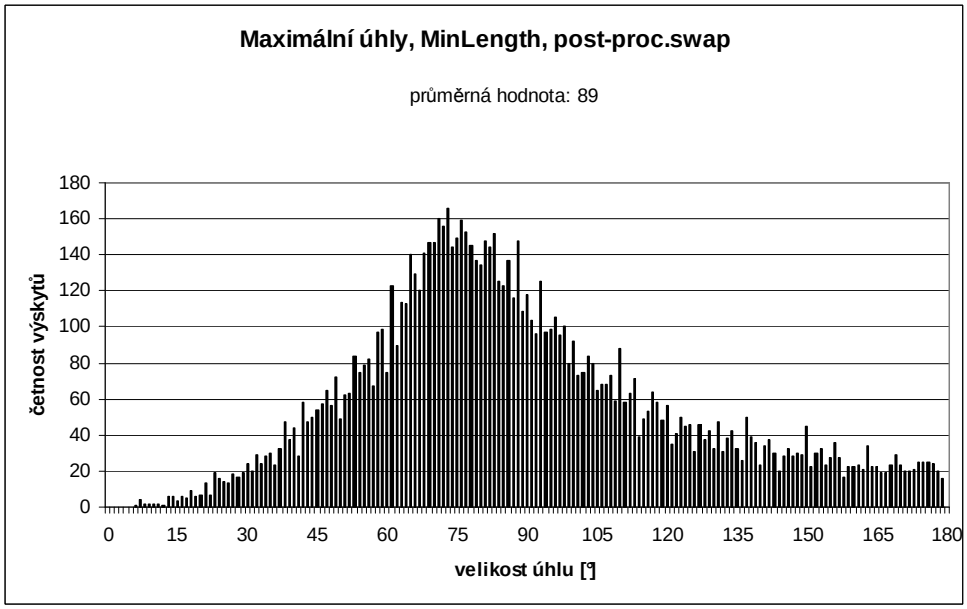


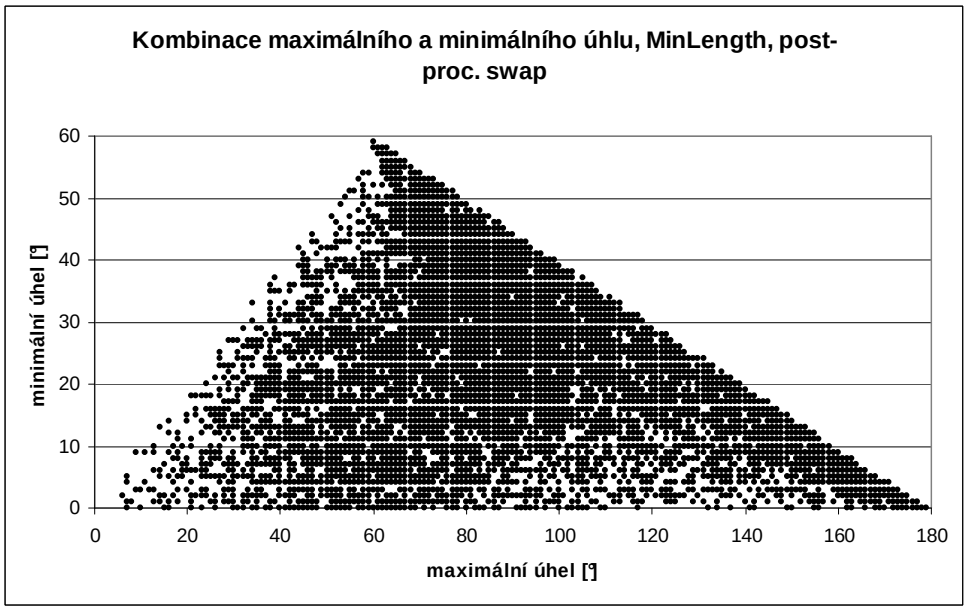




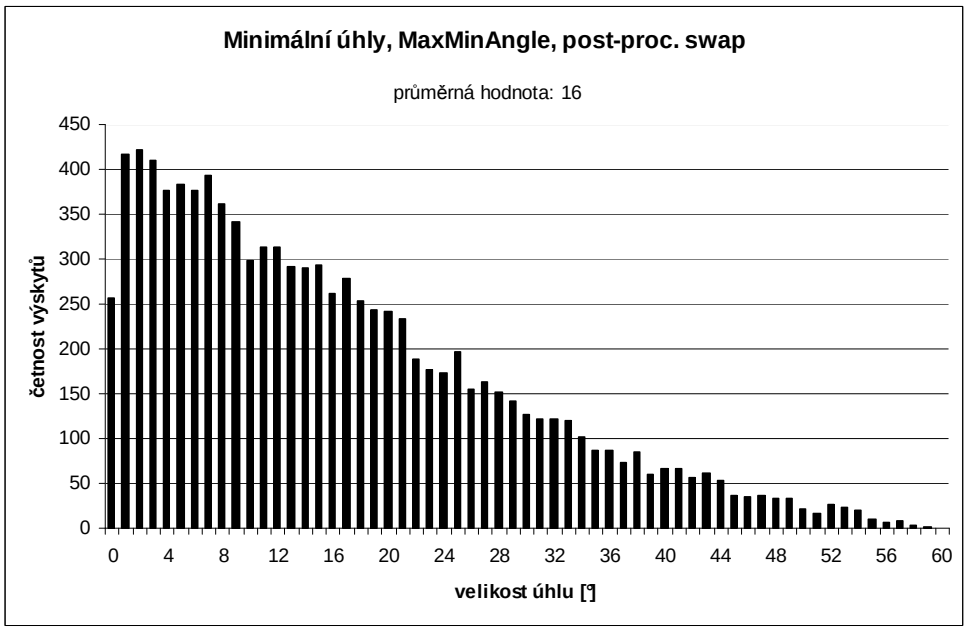
Graf C.12. Cluster 10000, MinMaxAngle, bez prohazování hran.

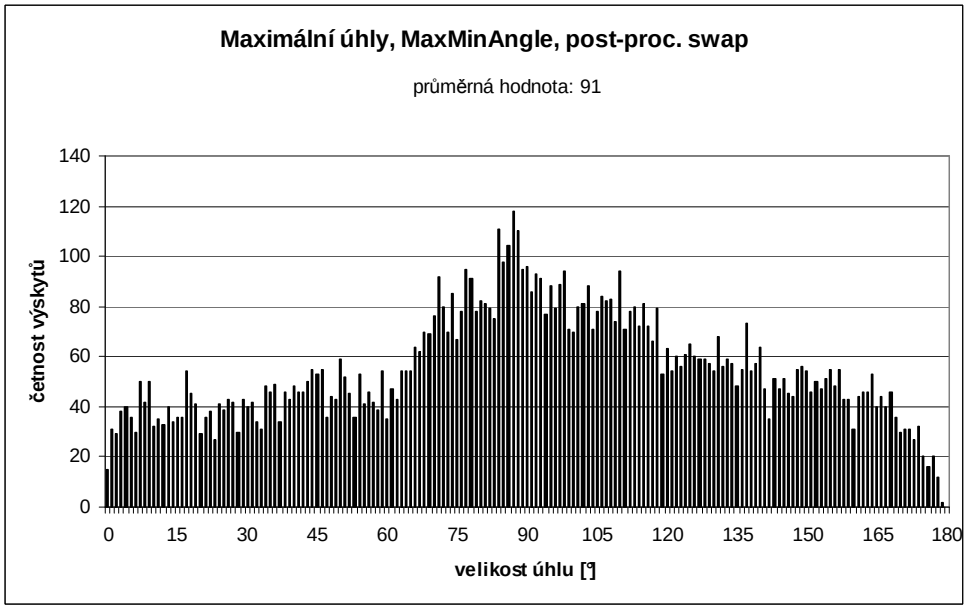


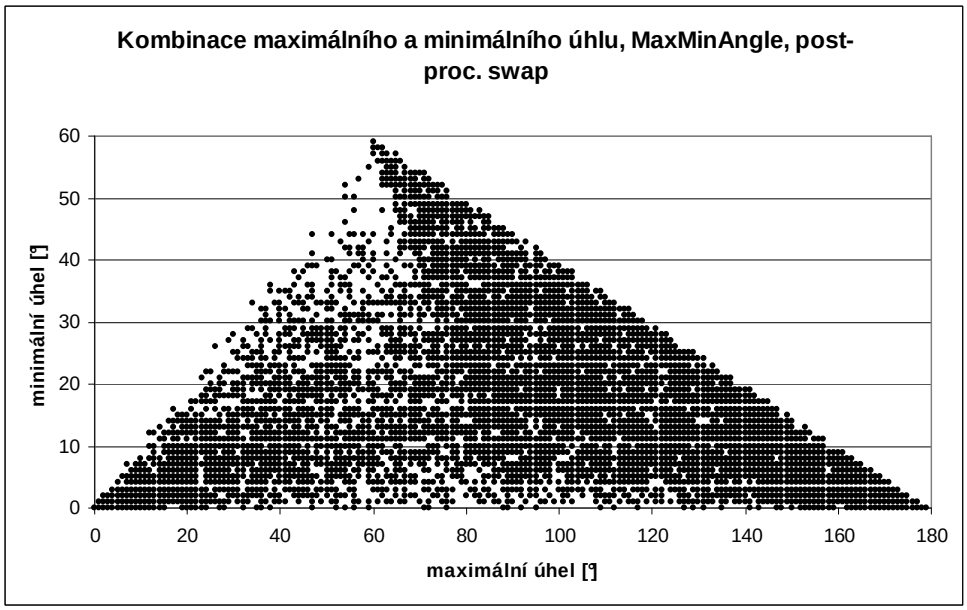




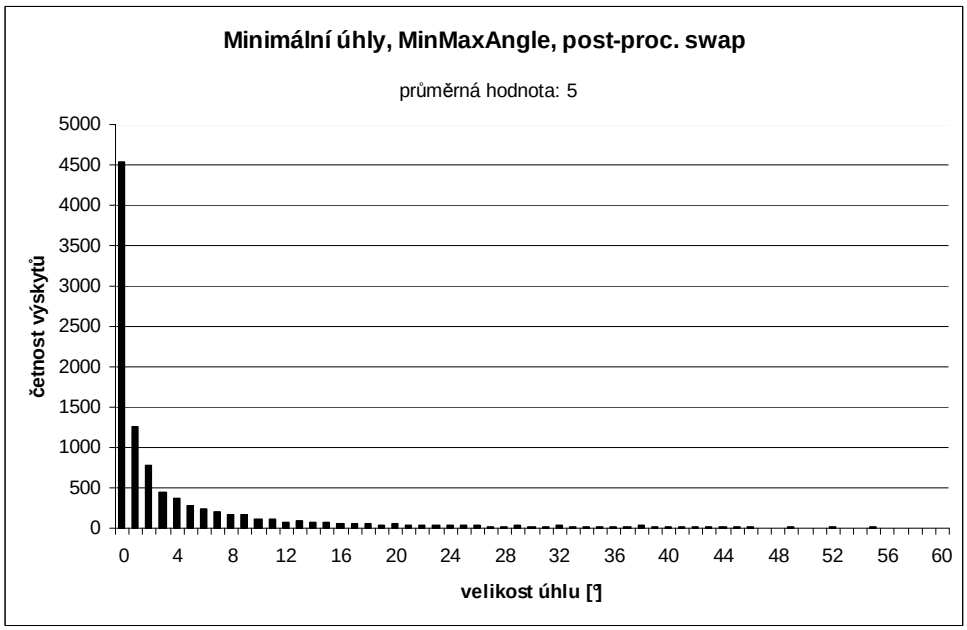
Graf C.13. Cluster 10000, MinLength, post-processing swap.

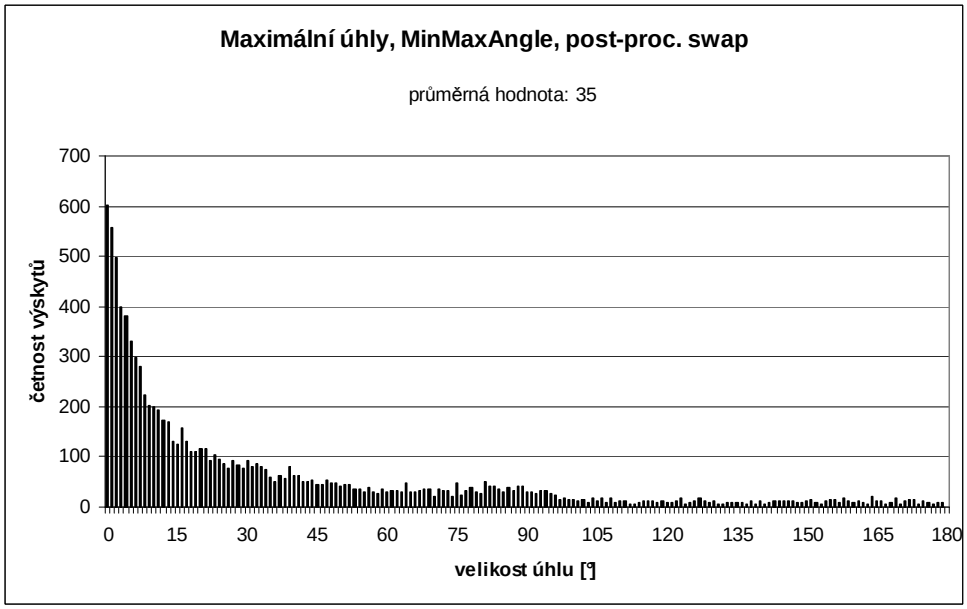


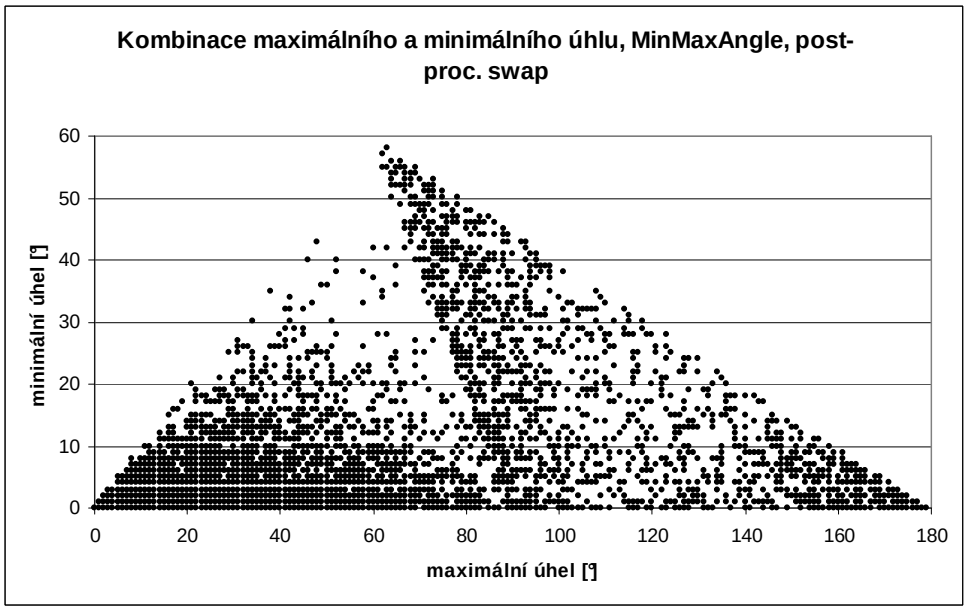




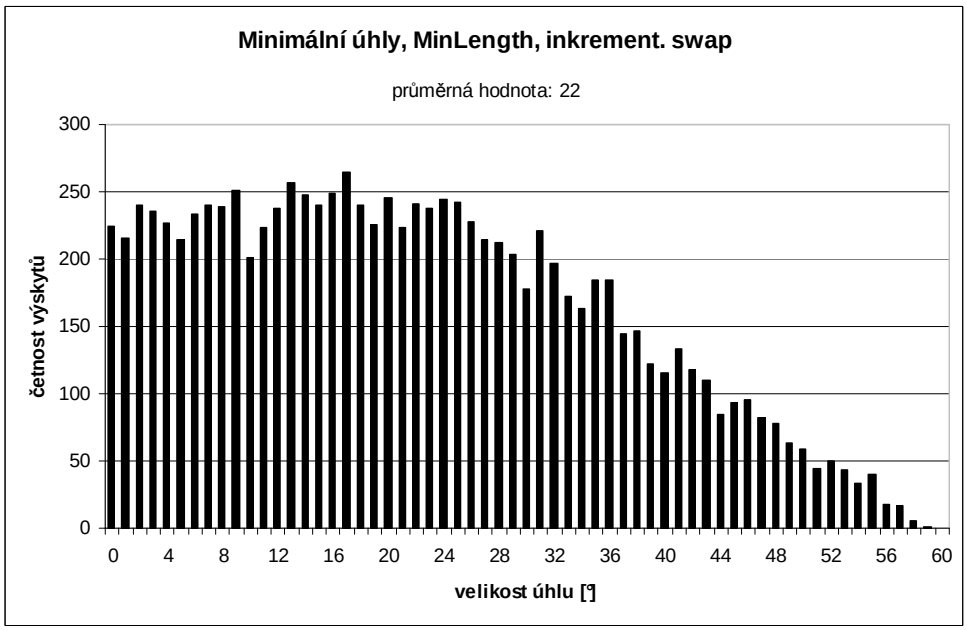
Graf C.14. Cluster 10000, MaxMinAngle, post-processing swap.

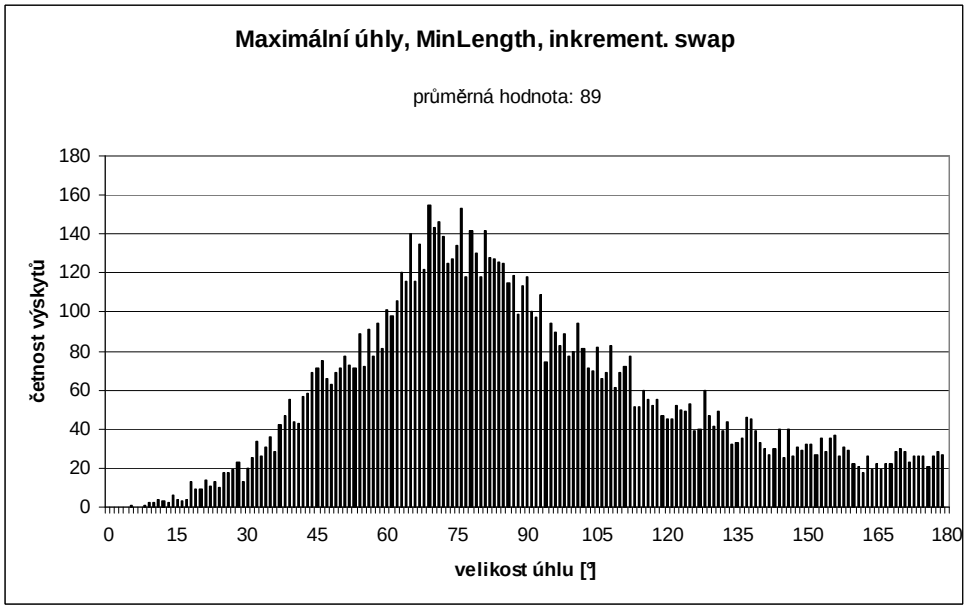


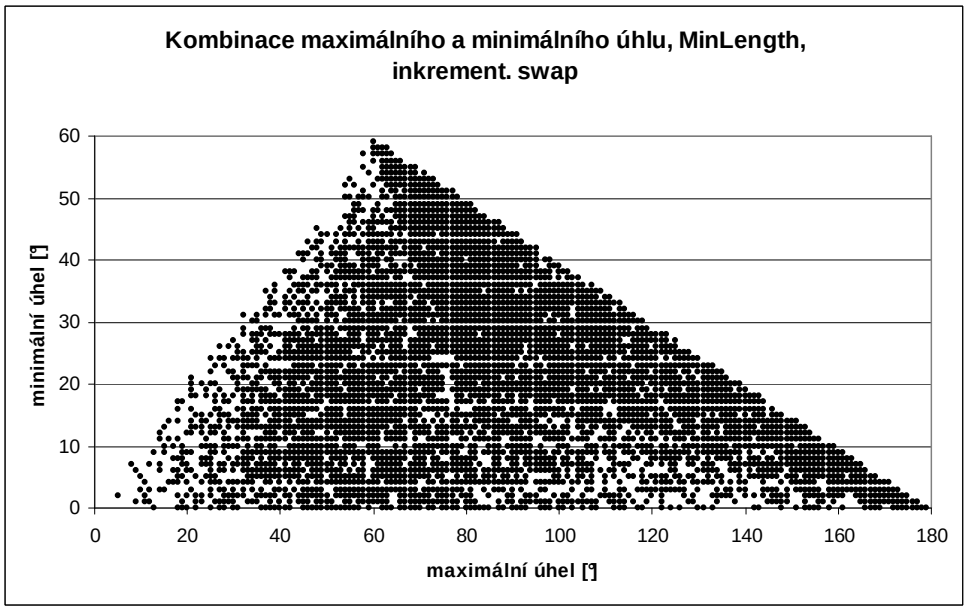




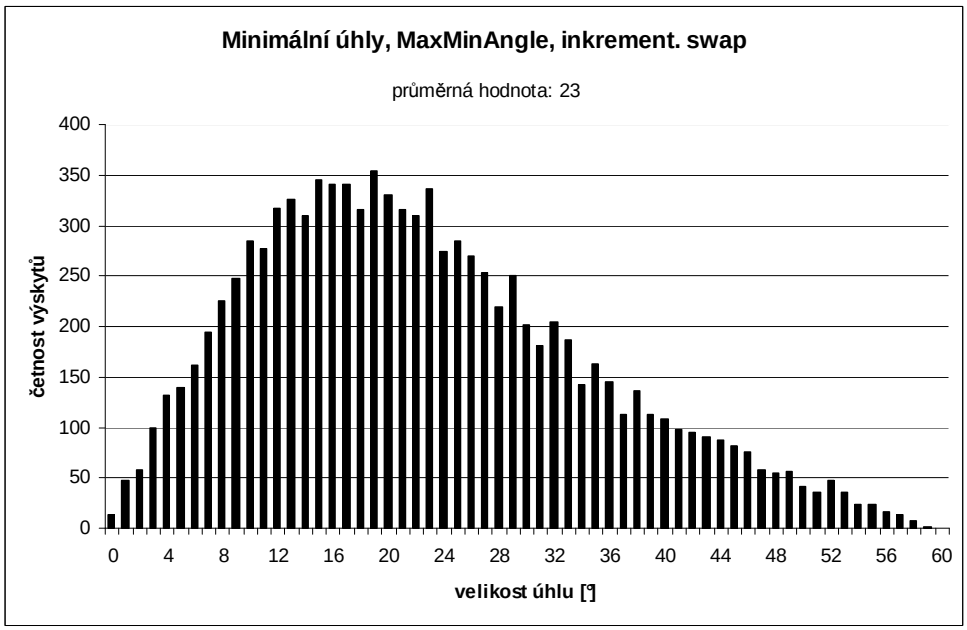
Graf C.15. Cluster 10000, MinMaxAngle, post-processing swap.

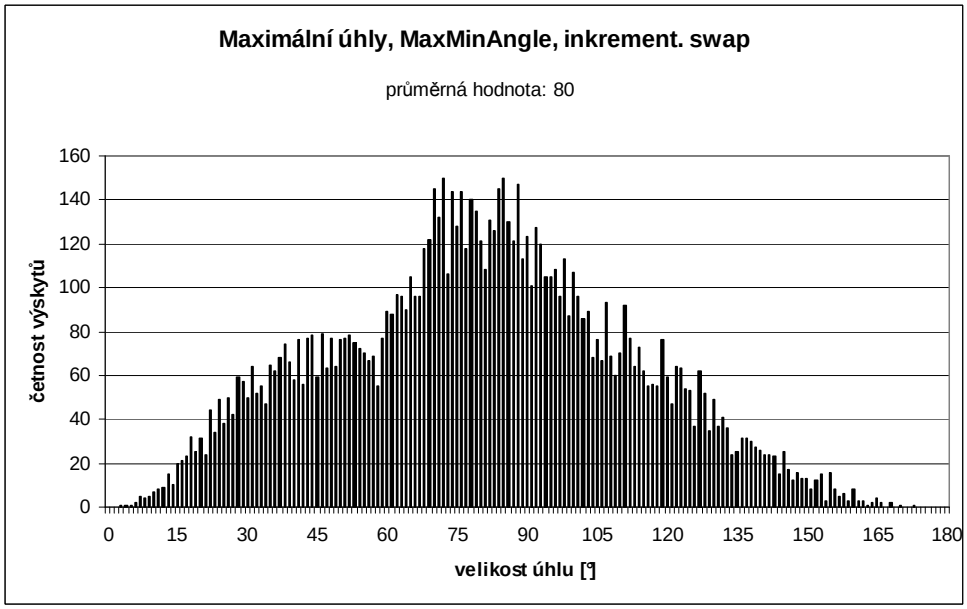


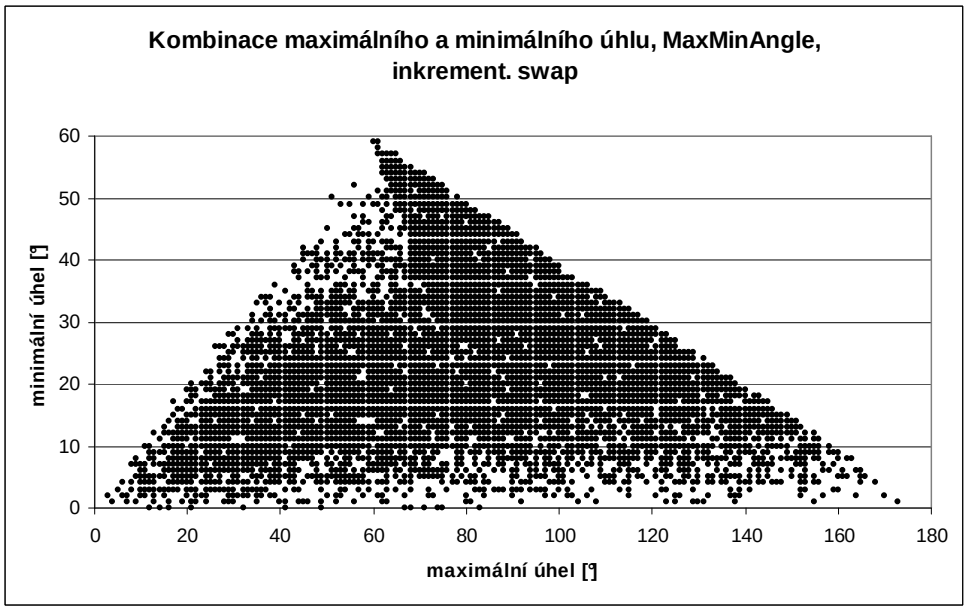




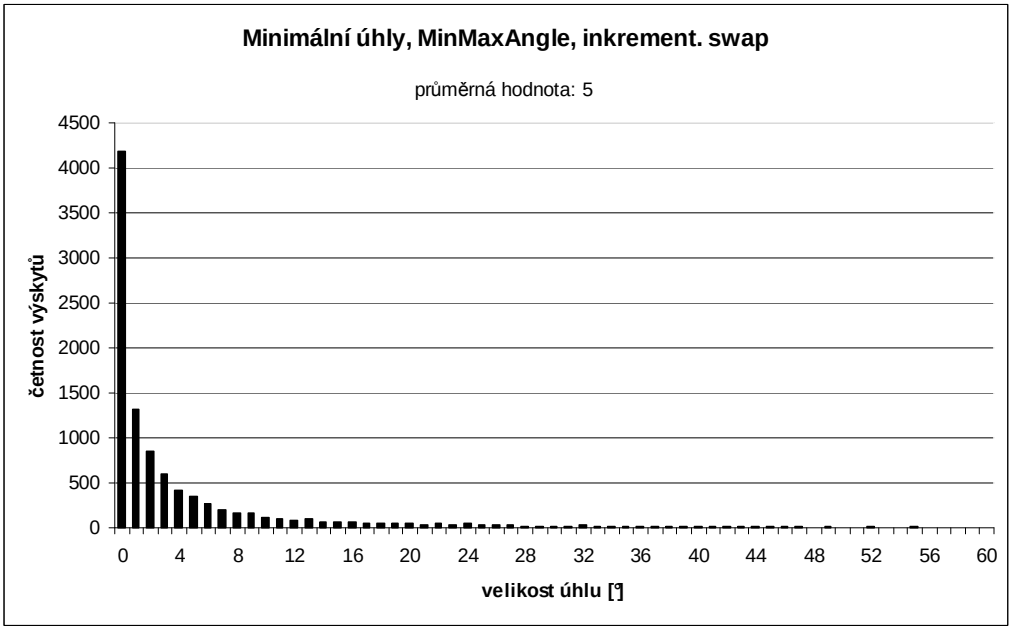
Graf C.16. Cluster 10000, MinLength, inkrementální swap.

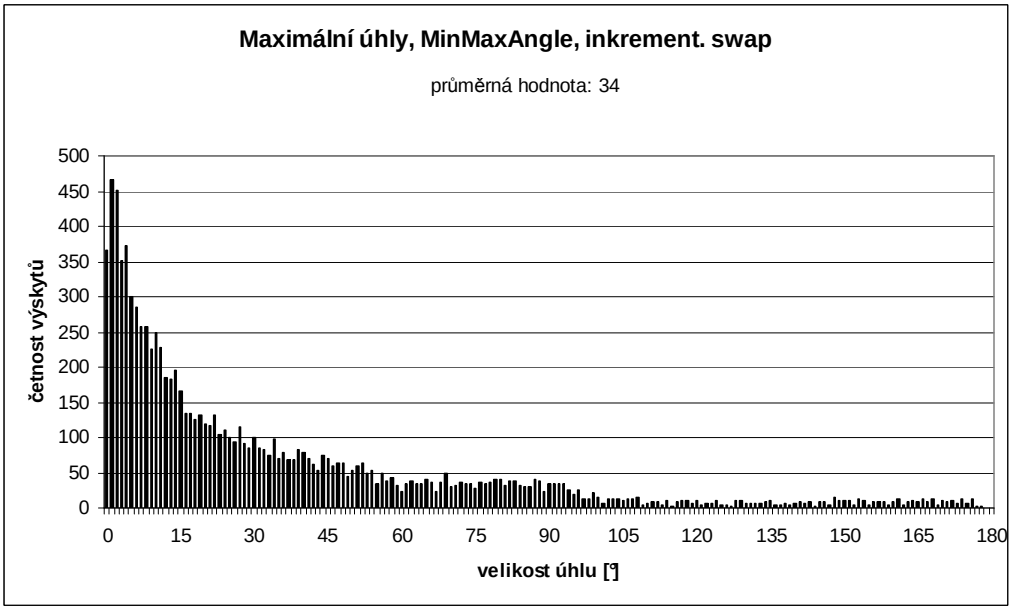


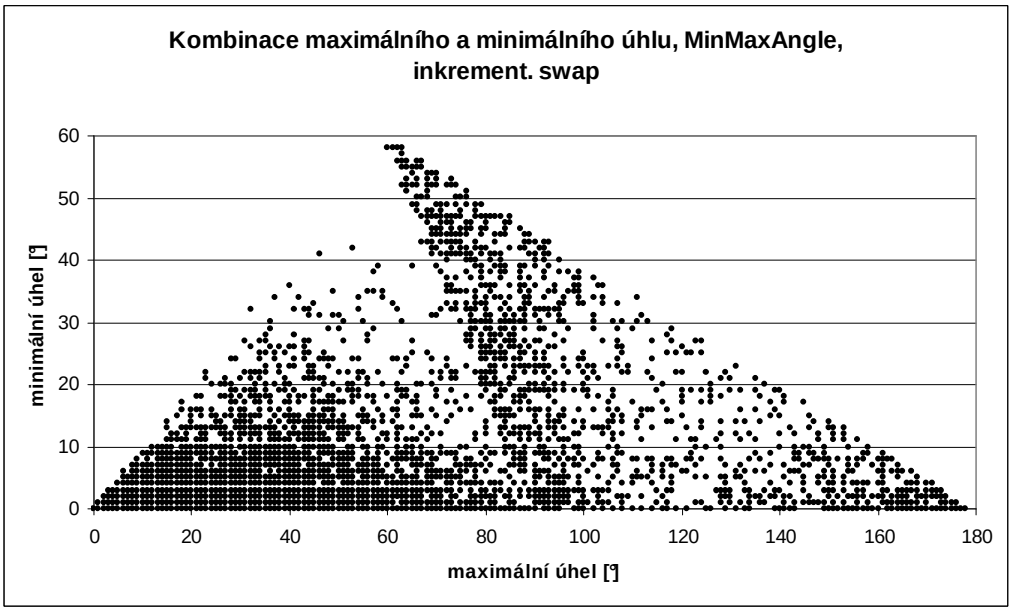




Graf C.17. Cluster 10000, MaxMinAngle, inkrementální swap







Graf C.18. Cluster 10000, MinMaxAngle, inkrementální swap..

D Program

Existující program, který je detailně popsaný v práci [Trčka07], byl doplněn pouze o několik funkčních tlačítek a dokumentace k tomuto programu byla doplněna o několik bodů.

D.1 Požadavky

Celý program byl napsán v jazyce Java. Pro spuštění programu je tedy nutné mít nainstalováno JRE[28] (verze 6.0 a víc). Program je možné spustit z příkazové řádky napsáním *java –jar pseudo.jar*, popř. pro přesměrování textového výstupu do souboru *java –jar pseudo.jar > output.txt*.

D.2 Doplněné ovládací prvky

Program byl doplněn o následující ovládací prvky:

Menu Settings

- **incremental swap** – Při zaškrtnutí této volby se při vytváření PT inkrementálním vkládáním bude provádět swap podle zvoleného kritéria.
- **brute-force swap** – Při zaškrtnutí se swap bude provádět brute-force algoritmem.
- **improved swap** – Při zaškrtnutí se swap bude provádět rychlejším algoritmem.
- **MaxMinAngleHull** – Při zaškrtnutí se bude používat toto kritérium.
- **MinMaxAngleHull** – Při zaškrtnutí se bude používat toto kritérium.

Menu Tools

- **swap –** Provede swap se na zadané pseudo-triangulaci Toto tlačítko umožňuje provádět swap jako post-processing.
- **angle measurement –** Vypočítá minimální a maximální úhly pseudo-trojúhelníků. Používá se pro testování kvality pseudo-triangulace. Výsledky uloží ve formě textového souboru do adresáře, v němž se nachází program.
- **print structure –** Vytiskne základní informace o zadané pseudo-triangulaci na obrazovku.

D.3 Použití nových ovládacích prvků

V této kapitole jsou uvedeny dva základní mnou využívané způsoby použití programu:

1) **Prohazování hran jako post-processing.** Ze souboru načtu množinu bodů (ctrl + O). Zvolím kritérium (v menu *settings*, zkratka crtl + alt + 1..7). V menu *Tools* zvolím *Pseudo-triangulate by inserting points* (ctrl + I). Tímto způsobem vytvořím v programu inkrementální pseudo-triangulaci bez prohazování hran. Dál zvolím v menu *Tools* možnost *Swap* (F7) a program provede swap jako tzv. post-processing.

[28] JRE (Java Runtime Environment)

2) **Vytvoření inkrementální pseudo-triangulace s prohazováním hran.** Postup je zprvu stejný jako v bodě 1). Načtu množinu bodů, zvolím kritérium. Navíc ale zaškrtnu v menu *Settings* volbu *incremental swap* (alt + C). Potom opět zvolím v menu *Tools* volbu *Pseudo-triangulate by inserting points* (ctrl + I). Program vytvoří pseudo-triangulaci inkrementálním vkládáním se začleněným prohazováním hran.

Pokud bych chtěl provést měření úhlů na pseudo-triangulaci (vytvořené libovolným způsobem) zvolím volbu *angle measurement* z menu *Tools*. Naměřené hodnoty budou zapsány do souborů *measure_max_angs.txt*, *measure_min_angs.txt* a *measure_comb.txt* v aktuálním adresáři ve formátu, který se snadno importuje např. do MS Excel.