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

Bakalářská práce

Deformace terénu pro virtuální realitu – geometrická část modelu

Plzeň, 2007 Jan Kadlec

Abstract

This work attempts to find out whether irregular triangle meshes are applicable in virtual reality for real-time simulations of terrain deformation. In practice, regular square grids are most frequently utilized, since they are very easy to implement. However, when high visual details are required, the resolution of the grid has to be very large. That results in an enormous memory consumption. On the other hand, irregular triangle meshes can be smoothed up only in particular areas where it is necessary, e.g., due to abnormal terrain segmentation. The remaining areas of the mesh can be made up by a radically smaller number of triangles. Hence, the rendering quality is maintained, while memory consumption is reduced. Nevertheless, manipulation with irregular triangle meshes is much more demanding computationally.

Obsah

Prohlášení

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

V Plzni, dne ..... Jan Kadlec

1 Úvod

Tato práce je součástí mezinárodně řízeného projektu, zabývajícího se zkoumáním použitelnosti nepravidelných trojúhelníkových sítí pro účely modelování deformací na povrchových modelech v *real-time* aplikacích pro virtuální realitu.

Vedoucím vizualizační části je Bedřich Beneš, Ph.D. z Department of Computer Graphics Technology, Purdue University, USA. Vedoucí geometrické části je Doc. Dr. Ing. Ivana Kolingerová z katedry informatiky a výpočetní techniky Západočeské univerzity v Plzni.

Pro reprezentaci terénu se v praxi často používají čtvercové sítě, jelikož se snadno implementují. Pokud ale požadujeme velké detaily, musí být rozlišení této sítě veliké a tím neúměrně roste i spotřeba paměti. Naproti tomu nepravidelné trojúhelníkové sítě můžeme zjemňovat jen tam, kde je to zapotřebí. Zbytek sítě se může skládat z menšího počtu trojúhelníků. Manipulace s nimi je ale výpočetně náročnější.

Cílem projektu je ověřit, zda jsou nepravidelné trojúhelníkové sítě použitelné v *real-time* aplikacích pro deformaci terénu. Úkolem bylo vytvořit „virtuální pískoviště“, tj. model terénu s písečným povrchem, do kterého bude možno v reálném čase provádět změny pomocí virtuálního nástroje.

Na projektu se podílela trojice řešitelů. Na základě analýzy úlohy jsme po dohodě s vedoucí práci rozdělili na tři části. Všichni tři jsme měli za úkol své části spojit do výsledné aplikace. Spolupráce s kolegou V. Purchartem byla velmi těsná, kdežto s kolegou J. Sedmihradským poměrně volná.

Obsahem mé části bylo prostudovat existující metody modelování terénu, ve spolupráci s kolegou Purchartem navrhnout geometrický model terénu vhodný pro interaktivní zásahy, implementovat tento model a zhodnotit funkcionalitu řešení.

Obsahem bakalářské práce kolegy Purcharta bylo jednak ve spolupráci s autorem této práce navrhnout metodu vhodnou pro využití geometrického modelu při interakcích ve stylu virtuální reality a jednak navrhnout a realizovat vhodné rozhraní, které dovolí práci využít také pro modelování přírodních vlivů na terénu.

Obsahem diplomové práce kolegy Sedmihradského bylo navrhnout a realizovat fyzikální část modelu, tj. částicový model simulující chování písku (erozi) a výsledný model vizualizovat.

Obsah dalších kapitol je popsán v následujícím seznamu:

- Kapitola 2 – *Analýza* – popisuje existující přístupy k modelování terénů, zdůvodňuje volbu použitého přístupu, typu sítě a použité triangulační metody.
- Kapitola 3 – *Geometrie* – poskytuje teoretickou základnu pro pochopení problematiky a podrobně popisuje všechny hlavní geometrické algoritmy, včetně použitých znaménkových testů.
- Kapitola 4 – *Implementace* – popisuje strukturu aplikace, podrobně rozebírá realizaci hlavních algoritmů nad použitými datovými strukturami a ukazuje řešení problémů, které se během vývoje vyskytly.
- Kapitola 5 – *Experimenty a výsledky* – vysvětluje postup práce během vývoje aplikace, popisuje provedené experimenty a jejich výsledky, dává k dispozici výsledky měření a výkonnostních testů triangulačního jádra, zhodnocuje funkčnost a efektivitu použitého řešení a navrhuje možná vylepšení pro potenciální budoucí vývoj.
- Kapitola 6 – *Závěr* – rekapituluje stanovené cíle a zhodnocuje, nakolik byly splněny.

2 Analýza

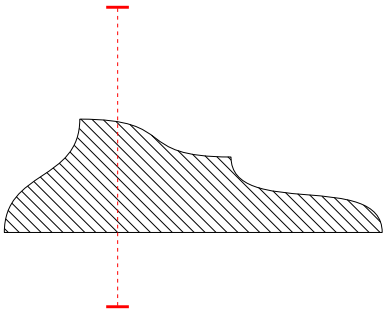
Text této kapitoly je převzat z naší dřívější práce [PRJ5].

2.1 Modelování terénů

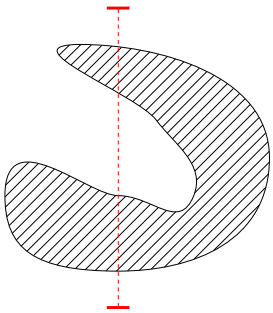
Pro počítačové modelování terénů ve 3D se používají nejčastěji povrchové modely. To jsou takové modely, které popisují pouze tvar povrchu terénu a nezabývají se tím, co se nachází pod ním.

Pro reprezentaci povrchového modelu se používají různé přístupy, nejčastější je použití 2D výškových map a plně 3D modelů. Plně 3D modely mají nesporné výhody oproti 2D modelům, totiž to, že umožňují namodelovat libovolný obecně konkávní objekt. Cenou za to je ovšem velmi vysoká časová výpočetní složitost používaných algoritmů.

Výškové mapy jsou zjednodušením 3D modelů. Vycházejí z myšlenky, že povrch můžeme uvažovat ve speciálním případě v podobě funkce dvou proměnných. Tím se jeden rozměr (výška) zredukuje na pouhý číselný údaj, který má význam vzdálenosti povrchu od nulové hladiny (myšlené podložky) v daném bodě. Dostáváme tak mapu, která uchovává pro vybrané body ve vodorovné rovině výškový údaj (viz obr. 2.1.1). Hlavní nevýhoda tohoto přístupu spočívá v tom, že nelze modelovat terén s přesahy, tzn. terén, na němž lze nalézt místo, v němž jeho povrch protíná svislou osu vícekrát než jednou (viz obr. 2.1.2). Takový povrch totiž není funkcí souřadnic ležících ve vodorovné rovině. Přicházíme tak o možnost podchytit skalní převisy, šikmé jámy a tunely, jeskyně apod. Na druhou stranu ale pro výškové mapy existují poměrně jednoduché a časově méně náročné algoritmy, a jsou proto vhodné pro aplikace, v nichž je kritickým faktorem čas. Takovými aplikacemi jsou zejména aplikace probíhající v reálném čase.



***Obr. 2.1.1**: Výšková mapa (boční pohled).*



***Obr. 2.1.2**: Terén s přesahem (boční pohled).*

Naše aplikace vyžaduje (v extrémních případech i poměrně drastické) změny modelu terénu, které musí probíhat v reálném čase. Proto byla jako model terénu zvolena výšková mapa.

Tento model uvažujeme jako model terénu s písečným povrchem. Písek má v suchém stavu tu vlastnost, že se sesypává a zahlazuje tak všechny přesahy a ostré přechody. V důsledku toho je v ustáleném stavu jeho povrch vyjádřitelný jako funkce dvou proměnných. Proto se můžeme spokojit s tím, že přesahy nebude možno modelovat.

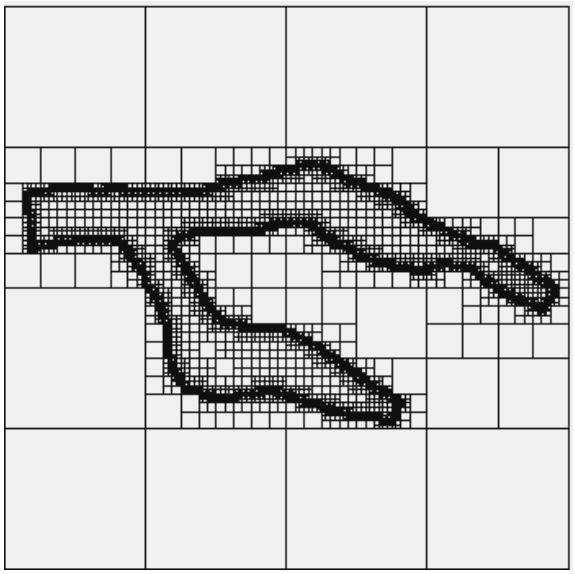
2.2 Volba sítě

K realizaci výškových map se používají vesměs dva různé přístupy: pravidelné sítě a nepravidelné sítě.

Pravidelné sítě jsou takové, jejichž buňky mají stejnou velikost. V praxi se nejčastěji používají čtvercové sítě, jelikož se velmi dobře implementují. Dále se používají pro různé účely např. i hexagonální sítě.

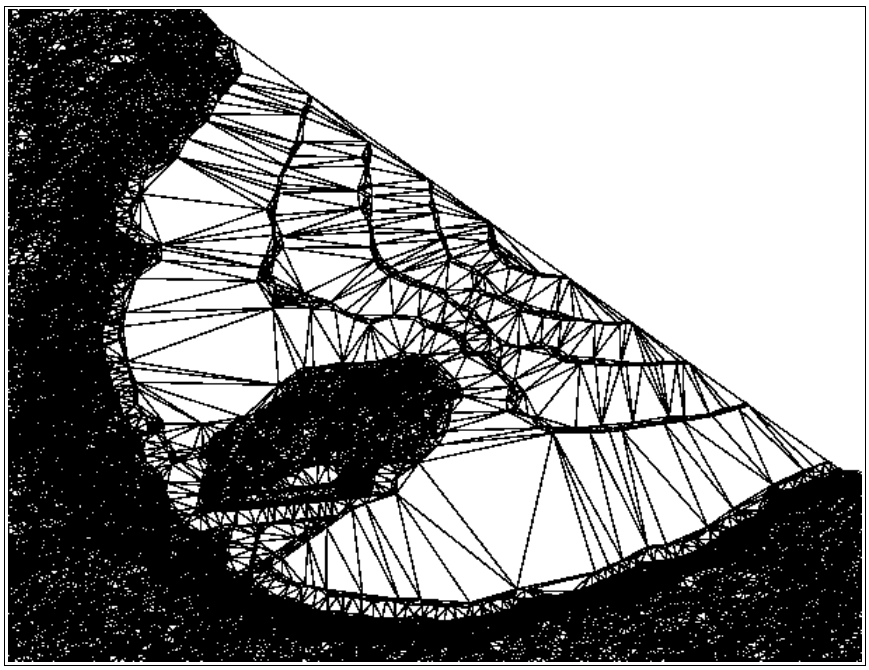
Nepravidelné sítě mohou být buď jednoduché a nebo rekurzivní.

Základem rekurzivních sítí je dělení jedné buňky na několik menších buněk stejného tvaru. Takovými sítěmi jsou např. *quad-tree* (viz obr. 2.2.1), která jako výchozí tvar buňky používá čtverec, nebo rekurzivní trojúhelníková síť, která jako tvar buňky volí rovnostranný trojúhelník.



***Obr. 2.2.1**: Síť* quad-tree *pro model bifurkace toku řeky (převzato z* [Liang1]*).*

Jednoduché nepravidelné sítě (dále jen nepravidelné) jsou představovány obecnou trojúhelníkovou sítí. Nepravidelné sítě nejsou, podobně jako rekurzivní sítě, svázány žádným globálním rozlišením. Různé části sítě tedy mohou mít různou úroveň rozlišení. Umožňují dosáhnout velkých detailů v místech, kde je terén vertikálně hodně členitý, a naopak minimalizovat úroveň detailů na téměř rovných plochách (viz obr. 2.2.2). Tím dostáváme model, který při zobrazení vypadá hladce, a to při nízkých paměťových nárocích.



Obr. 2.2.2*: Část modelu jezera Crater Lake vytvořeného pomocí nepravidelné trojúhelníkové sítě. (Vodní hladina je tvořena znatelně menším počtem trojúhelníků než okolní terén.)*

U pravidelných sítí je tento problém překonatelný jedině velkým zvýšením rozlišení, což má za následek velké nároky na paměť. (Např. u čtvercové sítě je celkový počet bodů v síti kvadraticky závislý na počtu bodů na jednotku vzdálenosti.)

Vlivy deformace a eroze má terén v naší aplikaci tendenci k velkým rozdílům členitosti v jednotlivých oblastech. Čerstvě deformovaný terén je hodně členitý, kdežto terén, na nějž působila eroze delší dobu, je charakteristický malými výškovými rozdíly. Proto pro nás pravidelné sítě nejsou vhodné.

Rekurzivní sítě sice odstraňují tento nedostatek, ale narážíme u nich na další problémy.

Do sítě nelze vnutit hrany, které neodpovídají hranám jednotlivých buněk. Proto je obtížné vynutit i základní geometrické obrazce jako např. kružnici. Navíc mají tyto sítě takovou strukturu, že dělení buněk obecně vede ke vzniku nepříjemných artefaktů na styku různě velikých buněk.

Z těchto důvodů jsme se rozhodli pro použití nepravidelné trojúhelníkové sítě.

2.3 Výběr triangulační metody

Obecná trojúhelníková síť je z uvedených sítí nejobecnější a přináší proto nejméně omezení. Na druhou stranu je z určitých hledisek absence omezení nevýhodná. Obecné trojúhelníkové sítě mj. dovolují vzniknout špatně podmíněným trojúhelníkům, které se mohou vlivem konečné přesnosti počítačové reprezentace čísel stát zdrojem nepříjemných numerických chyb. Proto se pro vytváření trojúhelníkových sítí používají triangulace (definice viz 3.1). Ty často používají kritéria, které tyto jevy omezují.

Existuje několik základních typů triangulace: Delaunayova triangulace (DT), žravá triangulace (GT), triangulace s minimální vahou (MWT), datově závislé triangulace (DDT), multikriteriálně optimalizované triangulace a triangulace s omezeními: Delaunayova triangulace s omezeními (CDT) a žravá triangulace s omezeními (CGT).

U triangulace s minimální vahou není známo, zda jde spočítat v polynomiálním čase, a je proto prakticky takřka nepoužitelná, zejména pro aplikace běžící v reálném čase. Multikriteriálně optimalizované triangulace jsou též velmi pomalé a není u nich zaručen úspěch, proto jsou též nepoužitelné. Přínos datově závislých triangulací (tj. respektování výškových rozdílů při vytváření trojúhelníků) pro nás nemá příliš velký význam, jelikož pracujeme s písečným terénem, který rychle eroduje a vyrovnává tak výškové rozdíly. Zbývají tedy na výběr Delaunayova triangulace s omezeními anebo žravá triangulace s omezeními. U žravé triangulace by však byla velmi problematická retriangulace části sítě. To je naopak velmi snadno proveditelné u Delaunayovy triangulace. Ta navíc maximalizuje minimální úhel a ze všech triangulací vytváří trojúhelníky nejbližší rovnostranným. To je obzvláště výhodné pro částicový systém. Při přesýpání písku je totiž nutné počítat s tím, že písek se přesouvá po hranách trojúhelníků v síti. Rychlost šíření hmoty je konečná a ta se tak musí přesouvat v čase postupně. Je tedy nevhodné mít v síti podlouhlé trojúhelníky s jednou stranou výrazně kratší než ostatní.

3 Geometrie

Tato sekce popisuje geometrickou teorii týkající se rovinných triangulací, včetně algoritmů pro jejich konstrukci, a vysvětluje, jak fungují potřebné znaménkové testy. Veškeré zmíněné geometrické objekty se předpokládají definované orientované proti směru hodinových ručiček (CCW), není-li uvedeno jinak.

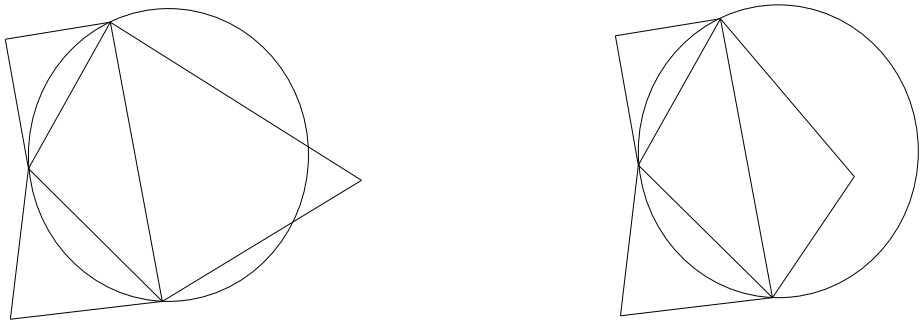
3.1 Rovinná triangulace

Triangulací v $\mathbb{R}^2$ neboli rovinnou triangulací na množině bodů P se rozumí libovolné rozdělení konvexní obálky množiny P na trojúhelníky (rovinné simplexy), které splňuje tyto dvě podmínky:

- každé dva trojúhelníky se buď neprotínají vůbec, nebo mají společný jeden bod, anebo jednu hranu,
- množina bodů R tvořená vrcholy všech trojúhelníků triangulace je shodná s množinou P.

3.2 Delaunayova triangulace

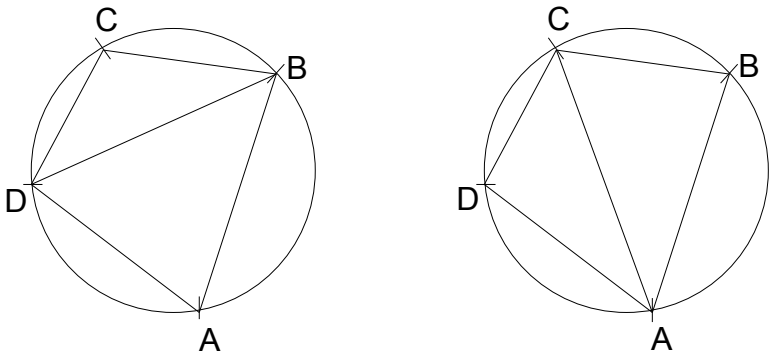
Základní Delaunayova triangulace v $\mathbb{R}^2$ na množině bodů P je taková triangulace $DT(P)$, v níž každý trojúhelník splňuje tzv. *Delaunayovo kritérium prázdné kružnice*. Trojúhelník ABC, kde $A, B, C \in P$ splňuje toho kritérium právě tehdy, když jemu opsaná kružnice neobsahuje žádný další bod z množiny P (viz obr. 3.2.1).



***Obr. 3.2.1**: Delaunayovo kritérium prázdné kružnice. (Vlevo: Trojúhelník splňující kritérium pro všechny hrany. Vpravo: Trojúhelník s jednou hranou nesplňující kritérium.)*

Za předpokladu, že žádné čtyři body z P neleží na prázdné kružnici, je dokázáno, že takto určená triangulace existuje na dané množině bodů P právě jedna.

Leží-li čtyři body $A, B, C, D \in P$ na kružnici a vytvářejí tak čtyřúhelník *ABCD*, pak je možné tento čtyřúhelník rozdělit na dva trojúhelníky dvěma různými způsoby: *ABC* a *ACD*, nebo *ABD* a *DBC*. Obě možnosti vyhovují Delaunayově kritériu (viz obr. 3.2.2). Triangulace *DT(P)* tedy není jednoznačná, ale je zaručeno, že alespoň jednu lze najít.



***Obr. 3.2.2**: Dvě alternativní konfigurace trojúhelníků pro čtyři body ležící na kružnici.*

Jednou z hlavních vlastností Delaunayovy triangulace je to, že maximalizuje minimální úhel trojúhelníků. V důsledku toho vytváří ze všech triangulací trojúhelníky nejbližší rovnostranným.

3.3 Konstrukce Delaunayovy triangulace

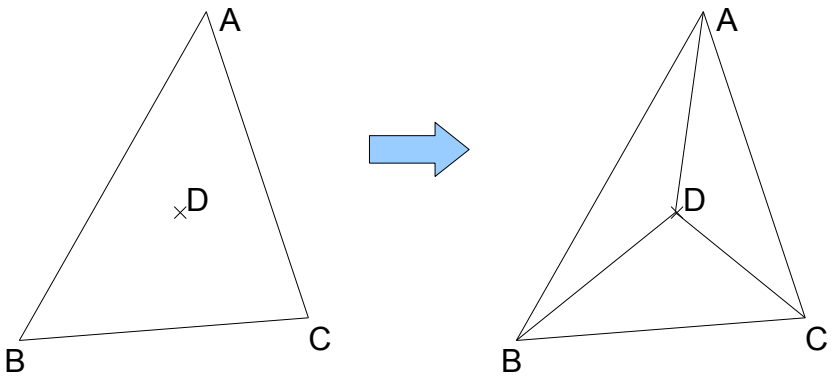
Existuje mnoho způsobů, jakými lze na dané množině bodů P vytvořit Delaunayovu triangulaci. V této práci je použito tzv. „inkrementální vkládání“. Jedná se o přímočarý algoritmus, pomocí něhož lze naprosto přirozenou cestou realizovat retriangulaci sítě v okolí nově vložených bodů.

Algoritmus je založený na postupném vkládání jednotlivých bodů. Po každém vložení bodu se síť rekonfiguruje, aby všechny trojúhelníky splňovaly Delaunayovo kritérium. Rekonfigurace vždy probíhá v okolí místa, kam byl nový bod vložen.

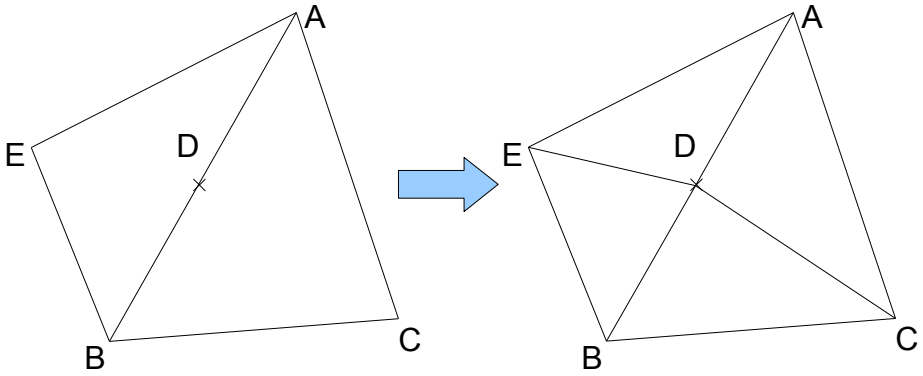
Pokud síť ještě neobsahuje žádné trojúhelníky, vytvoříme trojúhelník tak velký, aby pojal celou množinu vkládaných bodů. Pak postupně vložíme všechny body, které mají být v síti obsaženy. Vkládání bodu D pak probíhá následujícím způsobem:

Najdeme trojúhelník, v němž se bod *D* nachází, a označíme ho jako *ABC*. Tento trojúhelník pak rozdělíme na tři menší trojúhelníky. Každý z těchto trojúhelníků bude určen nově vloženým bodem *D* a dvěma body z původního trojúhelníku (viz obr. 3.3.1). Vzniknou tedy trojúhelníky *ABD*, *BCD* a *CAD*.

Pokud bod *D* leží na nějaké z hran trojúhelníka *ABC* (nebo blízko ní), vznikl by tímto rozdělením singulární (nebo jemu blízký) trojúhelník, což není žádoucí jev. Proto v takovém případě zahrneme do procesu i trojúhelník, který s *ABC* sousedí inkriminovanou hranou a místo třech trojúhelníků vytvoříme čtyři. Pokud *D* leží např. na hraně *AB*, vezmeme trojúhelník sousedící s *ABC* hranou *AB* a označíme ho *BAE*. Trojúhelníky nově vzniklé po rozdělení tedy budou *BCD*, *CAD*, *AED* a *EBD* (viz obr. 3.3.2).



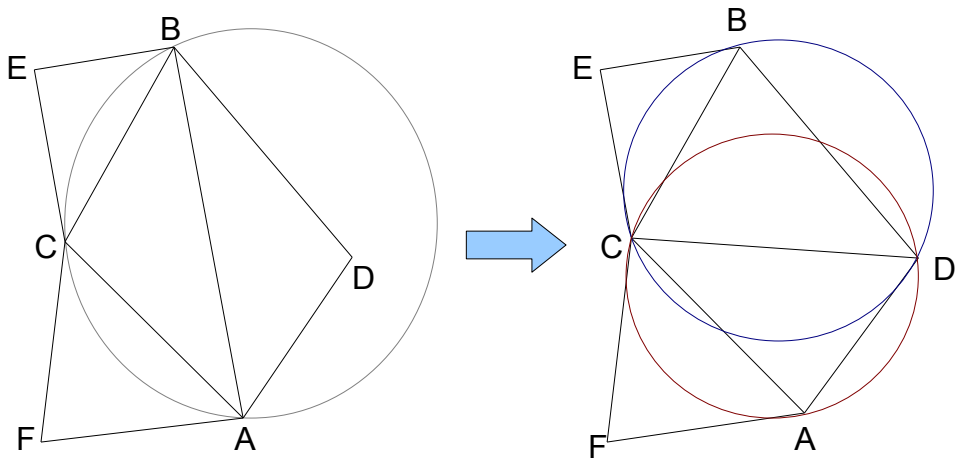
Obr. 3.3.1*: Vložení bodu do trojúhelníka (standardní případ – rozdělení na tři trojúhelníky).*



Obr. 3.3.2*: Vložení bodu do trojúhelníka (singulární případ – rozdělení na čtyři trojúhelníky).*

Pokud bod *D* leží přímo v některém z vrcholů *ABC* (nebo blízko něj), je nutné speciální zacházení, které záleží na konkrétní aplikaci. U obyčejné rovinné triangulace je možno nově vkládaný bod zcela ignorovat (jelikož se v síti již vlastně nachází). V této práci se nový bod sice také nevkládá, ale – jelikož je síť použitá jako výšková mapa a má tedy ještě třetí rozměr – starému bodu se přepíše údaj o výšce.

Krajní hrany (tj. v prvním případě hrany *AB*, *BC* a *CD* a v druhém případě hrany *AE*, *EB*, *BC*, *CA*) legalizujeme, tj. testujeme na Delaunayovo kritérium prázdné kružnice. Pokud je kritérium splněno, legalizace hrany končí. V opačném případě musíme hranu zrušit a vzniklý čtyřúhelník rozdělit na dva trojúhelníky tou úhlopříčkou, která nesplývá s právě zrušenou hranou (viz obr. 3.3.3). Poté lavinovitě šíříme legalizaci pro hrany obou nově vzniklých trojúhelníků. Nově vytvořenou hranu netestujeme, jelikož máme zaručeno, že kritérium splňuje.



***Obr. 3.3.3**: Legalizace hrany. (Vlevo: Počáteční stav. Nekorektní, jelikož bod D leží uvnitř kružnice opsané trojúhelníku* ABC. *Vpravo: Koncový stav. Korektní, jelikož obě opsané kružnice jsou prázdné.)*

3.4 Algoritmus procházky

Při konstrukci Delaunayovy triangulace často potřebujeme zjistit trojúhelník, v jakém leží zadaný bod, resp. jaký trojúhelník se nachází na zadaných souřadnicích. To lze zjistit různými způsoby. V této práci je použit tzv. „algoritmus procházky“. Jeho hlavní výhodou je, že není třeba uchovávat si žádné informace ve vedlejších datových strukturách, ani údaje o předchozích hledáních. Cenou za to je čas nutný k nalezení patřičného trojúhelníku.

Algoritmus funguje tak, že z nějakého trojúhelníku, který je zvolen jako startovní, se postupným „přelézáním“ z jednoho trojúhelníka do jiného přibližujeme k hledanému bodu a končíme v okamžiku, kdy hledaný bod leží v aktuálním trojúhelníku.

Startovní trojúhelník může být volen libovolně, ale v této práci jsou použity dva konkrétní přístupy. Tím prvním je výběr nejbližšího trojúhelníku z náhodně zvolené podmnožiny k trojúhelníků. Je odvozeno (viz [Mue1]), že nejvýhodnější volbou k je:

$$k = \left\lfloor \frac{\sqrt[3]{n+1}}{3} \right\rfloor, \tag{3.1}$$

kde hranaté závorky značí zaokrouhlení na celé číslo. Tento přístup je proti očekávání rychlejší, než prosté zvolení jednoho náhodného trojúhelníku.

Druhý přístup můžeme použít v případě, že přibližně víme, kde se hledaný trojúhelník nachází, resp. známe trojúhelník, který se nachází v jeho blízkosti. Potom je velice vhodné tento trojúhelník použít jako startovací. Při modelování deformací terénu se často provádí velké množství změn na jednom místě, takže je tento přístup velice dobře použitelný.

3.5 Delaunayova triangulace s ohraničeními

Delaunayova triangulace s ohraničeními (*constrained Delaunay triangulation*, dále jen CDT) je triangulační metodou, vycházející ze základní Delaunayovy triangulace. Oproti té se v CDT však zavádí tzv. „vynucené hrany“. To jsou hrany, na které je kladen požadavek, aby se v síti objevily i v případě, že nesplňují Delaunayovo kritérium prázdné kružnice.

Tento požadavek může v některých aplikacích vyplynout zcela přirozeně. Např. při modelování terénů je někdy nutné, aby model podchycoval takové rysy skutečného terénu, jako jsou ostré zlomy či příkré svahy. V některých aplikacích je nutné poměrně přesně podchytit vrstevnice terénu. V takových situacích se potom často přistupuje k použití triangulace s ohraničeními, která tyto rysy terénu v modelu vynutí, a tou bývá nejčastěji právě CDT.

3.6 Konstrukce Delaunayovy triangulace s ohraničeními

CDT lze konstruovat různými způsoby. V této práci je použit algoritmus vycházející ze [Sloan1]. Ten spočívá v tom, že se nejdříve vytvoří síť pomocí základní Delaunayovy triangulace, a posléze se v ní vynutí požadovaná množina hran. Algoritmus pro vynucení množiny hran R je

následující:

1. Pro každou hranu $E_i \in R$ opakujeme kroky 2–5.
2. Zjistíme, zda je hrana E_i již přítomna v triangulaci. Pokud ano, přejdeme na bod 1 a pokračujeme další hranou.
3. Prohledáme triangulaci a najdeme všechny hrany, které se protínají s vynucenou hranou E_i. Tyto hrany uložíme do seznamu protínajících se hran I.
4. Dokud není seznam I prázdný, tj. dokud je v triangulaci ještě přítomna nějaká hrana, která protíná vynucenou hranu, opakujeme kroky 4a)–4c).
   a) Odebereme první hranu ze seznamu I a označíme ji E_j.
   b) Pokud dva trojúhelníky, které sdílí hranu E_j, netvoří ostře konvexní čtyřúhelník, vrátíme hranu E_j na konec seznamu I a pokračujeme krokem 4a). V opačném případě v tomto čtyřúhelníku prohodíme diagonálu, tj. nahradíme staré dva trojúhelníky novými dvěma, které vzniknou rozdělením čtyřúhelníku podle diagonály, která nesplývá s původní hranou E_j. Novou diagonálu označíme jako E_k.
   c) Pokud nová diagonála E_k stále protíná vynucenou hranu E_i, přidáme E_k na konec seznamu protínajících se hran I. V opačném případě přidáme E_k do seznamu vytvořených hran C.
5. Legalizujeme všechny hrany v seznamu C kromě hrany E_j, tj. pokud nějaká hrana nesplňuje Delaunayovo kritérium a není vynucená, prohodíme ji tak, aby nově vzniklá hrana Delaunayovo kritérium splňovala.
   a) Pro každou hranu $E_l \in C$ provedeme kroky 5b) a 5c).
   b) Pokud hrana E_l je shodná s vynucenou hranou E_i, pak přejdeme na bod 5a) a pokračujeme další hranou.
   c) Pokud trojúhelníky, které sdílí hranu E_l, nesplňují Delaunayovo kritérium, tj. bod jednoho leží v kruhu vytyčeném opsanou kružnicí druhého, tak ve čtyřúhelníku, který vytvářejí, prohodíme diagonálu. Vznikne tak hrana E_m. V seznamu C musíme v rámci zachování konzistence dat hranu E_l nahradit hranou E_m.

Je zřejmé, že z důvodu omezení vzniklých ohraničeními nemusí některé trojúhelníky, které během procesu vzniknou, splňovat Delaunayovo kritérium. Přesto je však zaručeno, že trojúhelníky neobsahující žádný bod, z nějž vychází vynucená hrana, Delaunayovo kritérium splňují.

Jelikož v síti budou probíhat v čase změny, je nutné počítat s tím, že musí být možno za běhu přidávat a odebírat vynucené hrany. Popsaný algoritmus vyžaduje, aby do něj vstupovala množina hran k vynucení a konzistentní triangulace. To znamená, že budou existovat hrany vynucené z dřívější doby a hrany nově určené k vynucení. Situaci lze vyřešit tak, že síť bude mít uloženou informaci o tom, které hrany v ní jsou vynucené. Před každým přidáním vynucené hrany se zajistí, že síť obsahuje oba její body (tj. detekuje se, zda jsou již přítomny, a v případě, že nikoliv, se přidají). Potom se teprve začne vykonávat popsaný algoritmus.

Implementační detaily jsou značně závislé na použité datové reprezentaci a konkrétní implementace toho algoritmu je popsána v části 4. Je však nutno použít několik dílčích algoritmů, jejichž myšlenka je na implementaci nezávislá.

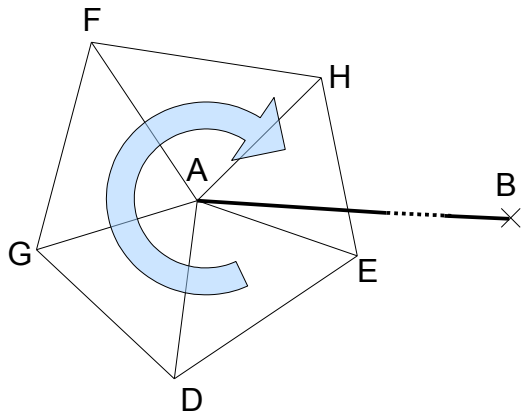
V kroku 3 je nutné nalézt všechny hrany, které se protínají s vynucenou hranou E_i. To lze provést následujícím způsobem:

1. Založíme prázdný seznam vytvořených hran *I*.
2. Označíme koncové body hrany E_i jako *A* a *B*.
3. Najdeme trojúhelník, v němž leží jeden koncový bod hrany E_i, např. *A*. Jelikož oba body této hrany jsou v tomto okamžiku již v síti přítomny, obsahuje tento trojúhelník bod *A* jako jeden ze svých vrcholů. Označíme ho tedy *ADE*.
4. Nyní se potřebujeme postupným „přelézáním“ z jednoho trojúhelníku do druhého dostat až do trojúhelníku, jenž obsahuje bod *B*. Prvním krokem je najít startovní trojúhelník, tj. trojúhelník, který obsahuje bod *A* a jeho hrana *DE* (hrana protilehlá vrcholu *A*) protíná vynucenou hranu E_i. Z geometrické analýzy situace vyplývá, že tento trojúhelník vždy existuje a sousedí s trojúhelníkem *ADE* jednou hranou a nebo se ho dotýká jedním bodem. Tento trojúhelník tak lze najít následujícím postupem:
   a) Nastav aktuální trojúhelník na *ADE*.

b) Opakuj krok 3c), dokud není aktuální trojúhelník (označíme ho *AFG*, jelikož o něm víme, že obsahuje bod *A*) startovním trojúhelníkem (viz obr. 3.6.1). To se pozná tak, že orientovaný znaménkový test proti úsečce určené hranou E_i vyjde pro bod *F* jinak nežli pro bod *G*. Znaménkový test je popsán v sekci 3.7.3.

c) Nastavíme aktuální trojúhelník na trojúhelník, který s trojúhelníkem *AFG* sousedí přes hranu *AF*.

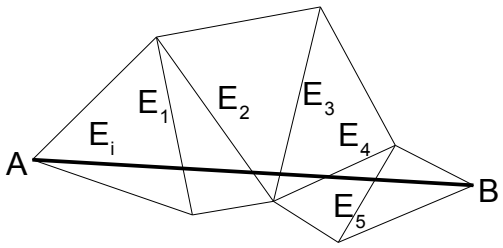
d) Tímto způsobem se vlastně pohybujeme dokola po trojúhelnících, které obsahují bod *A*, tak dlouho, dokud nenajdeme startovní trojúhelník. Za předpokladu, že trojúhelník *AFG* uvažujeme orientovaný proti směru hodinových ručiček, se jedná o pravotočivý pohyb, tj. po směru hodinových ručiček. Testovaných trojúhelníků je konečný počet, takže se nemůže stát, že bychom startovní trojúhelník nenašli.



Obr. 3.6.1*: Obcházení bodu* A *za účelem nalezení startovního trojúhelníku. (Vycházíme z trojúhelníku* ADE*. Nalezený startovní trojúhelník je* AEH*.)*

5. Nyní musíme postupně „přelézt“ ze startovního trojúhelníku do trojúhelníku obsahujícího bod *B*. Nastavíme aktuální trojúhelník na startovní trojúhelník.
6. Dokud neobsahuje aktuální trojúhelník koncový bod *B*, opakujeme kroky 7 a 8.
7. Pomocí znaménkových testů (viz 3.7.3) zjistíme, kterou ze hran aktuálního trojúhelníka protíná vynucená hrana E_i. Hranu, po níž jsme přišli, netestujeme. První protínající se hranu, kterou detekujeme, označíme jako E_n.

8. Nastav aktuální trojúhelník na trojúhelník, který sousedí se stávajícím aktuálním trojúhelníkem přes hranu E_n. Pokud hrana E_n neobsahuje bod A ani B, přidej ji do seznamu I.
9. Nyní jsme došli až do koncového trojúhelníku a v seznamu I máme všechny hrany protínající vynucenou hranu E_i, kromě těch hran, které přímo vycházejí z některého jejího koncového bodu.



***Obr. 3.6.2**: Hledání protínajících se hran. (Vycházíme z bodu* A. *Cílovým bodem je* B. *Postupně detekujeme protínající se hrany* E_1–E_5.*)*

3.7 Znaménkové testy

V použitých algoritmech je na několika místech nutné rozhodnout o vzájemné poloze geometrických objektů, konkrétně o poloze bodu vůči kružnici definované třemi body a poloze bodu vůči úsečce. Tato rozhodnutí se provádí pomocí (elementárních) znaménkových testů a jsou součástí složitějších testů. Těmi jsou rozhodování o tom, zda jedna úsečka nebo přímka protíná druhou, a o tom, zda je daný čtyřúhelník konvexní.

Elementární znaménkové testy lze rychle provést pomocí výpočtu speciálních determinantů.

3.7.1 Test polohy bodu vůči přímce

Tento test určuje, zda bod leží na přímce, vlevo od ní, anebo vpravo od ní. Předpokládejme, že přímka p je definována dvěma různými body $X_1=[x_1, y_1]$, $X_2=[x_2, y_2]$ a testovaný bod je $P=[x_3, y_3]$. Potom o poloze bodu P vůči přímce p hovoří následující determinant:

$$d = \begin{vmatrix} y_2 - y_1 & y_3 - y_2 \\ x_2 - x_1 & x_3 - x_2 \end{vmatrix} . \qquad (3.2)$$

Pokud je $d < 0$, svírá polopřímka X_1P s polopřímkou X_1X_2 úhel $\alpha \in (0, \pi)$ radiánů. To znamená, že bod P leží v „levé“ polorovině, kterou vytíná přímka p (za předpokladu, že směr vektoru X_1X_2 označíme jako směr „nahoru“).

Pokud je $d > 0$, svírají polopřímky úhel $\alpha \in (\pi, 2\pi)$ radiánů. Bod P tudíž leží v „pravé“ polorovině.

Pokud je d = 0, svírají polopřímky úhel $\alpha = 0$ nebo $\alpha = \pi$ radiánů. Bod P tedy leží na přímce p.

3.7.2 Test polohy bodu vůči kružnici

Tento test slouží k určení, zda bod leží uvnitř kruhu, mimo něj, anebo na hraniční kružnici. Nechť $X_1 = [x_1, y_1]$, $X_2 = [x_2, y_2]$ a $X_3 = [x_3, y_3]$ jsou body definující kružnici k (seřazené po směru hodinových ručiček) a $P = [x_4, y_4]$ je testovaný bod.

Na polohu bodu P vůči kružnici k lze usuzovat ze znaménka následujícího determinantu:

$$d = \begin{vmatrix} x_1 - x_4 & y_1 - y_4 & (x_1 - x_4)^2 + (y_1 - y_4)^2 \\ x_2 - x_4 & y_2 - y_4 & (x_2 - x_4)^2 + (y_2 - y_4)^2 \\ x_3 - x_4 & y_3 - y_4 & (x_3 - x_4)^2 + (y_3 - y_4)^2 \end{vmatrix} . \qquad (3.3)$$

Když je $d < 0$, nachází se bod P mimo kruh ohraničený kružnicí k. Když je $d > 0$, leží bod P uvnitř kruhu ohraničeného kružnicí k. Když je $d = 0$, leží bod P na kružnici k, tj. body X_1, X_2, X_3 a P leží na stejné kružnici.

3.7.3 Test protnutí úsečky a přímky

Často potřebujeme zjistit, zda nějaká úsečka protíná danou přímku nebo úsečku. To lze poměrně elegantně určit za použití znaménkových testů polohy bodu vůči přímce (viz 3.7.1). Řekněme, že přímka p je definována dvěma různými body $X_1 = [x_1, y_1]$, $X_2 = [x_2, y_2]$ a zkoumaná úsečka je dána dvěma různými body $P = [x_3, y_3]$ a $Q = [x_4, y_4]$.

Provedeme znaménkový test polohy bodu P vůči přímce p a test polohy bodu Q vůči přímce p.

Pakliže vyjde u obou testů nenulová hodnota a znaménko obou hodnot je stejné, znamená to, že oba koncové body úsečky leží ve stejné polorovině vytyčené přímkou *p*, tj. úsečka tuto přímku neprotíná.

Vyjde-li u obou testů nenulová hodnota a znaménka hodnot jsou odlišná, leží každý bod úsečky v jiné polorovině vytyčené přímkou *p*, tj. jejich spojnice (úsečka *PQ*) nutně přímku *p* protíná.

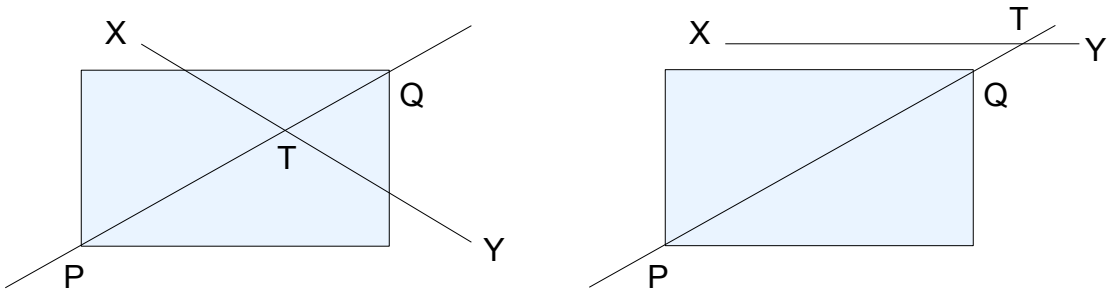
Pokud vyjde jedno znaménko rovno nule, znamená to, že jeden bod úsečky *PQ* leží na přímce *p*. Pokud vyjdou znaménka rovna nule obě, leží na přímce *p* celá úsečka *PQ*. Tyto případy jsou singulární a je nutné s nimi zacházet podle konkrétního kontextu.

Stejně jako tento neorientovaný test je možno zkonstruovat i orientovaný test. Ten se liší pouze v tom, že záleží na směru, v němž (orientovaná) úsečka *PQ* protíná přímku *p*. Sledujeme tedy konkrétní hodnoty znamének. Je-li znaménko testu pro bod záporné, leží bod v „levé“ polorovině. Je-li znaménko kladné, leží bod v „pravé“ polorovině. To znamená, že chceme-li, aby úsečka *PQ* protínala přímku *p* zleva doprava, musí být znaménko pro *P* kladné a pro *Q* záporné. A naopak úsečka *PQ* protíná přímku *p* zprava doleva, když je znaménko pro *P* záporné a pro *Q* kladné.

3.7.4 Test protnutí dvou úseček

Cílem tohoto testu je zjistit, zda úsečka určená body $X=[x_1, y_1]$, $Y=[x_2, y_2]$ protíná úsečku určenou body $P=[x_3, y_3]$ a $Q=[x_4, y_4]$. Jedná se o rozšířenou variantu testu protnutí úsečky a přímky (viz 3.7.3).

V případě, že je výsledek testu protnutí úsečky *XY* a přímky definované body *P* a *Q* pozitivní, musíme ověřit, že průsečík leží na úsečce *PQ*, tj. mezi body *P* a *Q*. Konkrétní průsečík $T=[x_5, y_5]$ nás sice nezajímá, ale spočítáme ho. To lze v obecném případě provést pomocí výpočtu uvedeného v sekci 3.7.6. Ve zvláštním případě, kdy bod jedné úsečky leží na druhé úsečce, nic počítat nemusíme, jelikož průsečík je právě zmiňovaný bod. Když známe průsečík, zkontrolujeme, zda leží v obdélníku definovaném body $[x_3, y_3]$, $[x_4, y_3]$, $[x_4, y_4]$ a $[x_3, y_4]$. V případě, že bod *T* v tomto obdélníku leží, je z obr. 3.7.1 patrné, že leží i na úsečce *PQ*, a tedy úsečka *XY* protíná úsečku *PQ*.



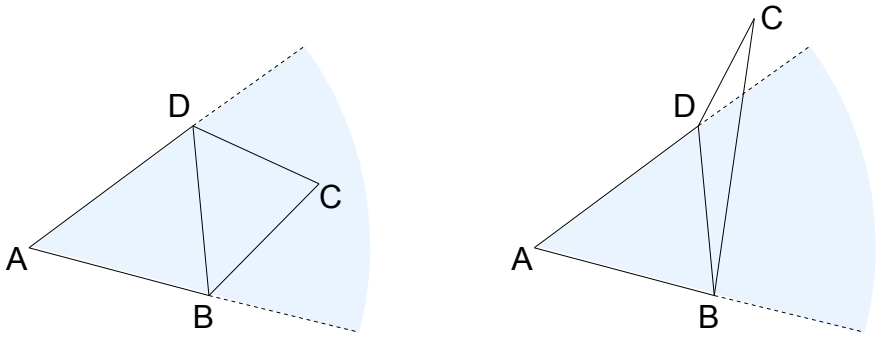
***Obr. 3.7.1**: Poloha úsečky proti úsečce. (Vlevo: Bod* T *leží v obdélníku vymezeném body* P *a* Q*, úsečky se protínají. Vpravo: Bod* T *leží mimo obdélník, úsečky se neprotínají.)*

3.7.5 Test konvexnosti čtyřúhelníku

Tento test umožňuje zjistit, zda dva trojúhelníky sdílející jednu hranu dohromady vytvářejí konvexní čtyřúhelník. Předpokládáme přitom, že kromě této hrany trojúhelníky nesdílejí žádné body, jinými slovy žádný bod jednoho trojúhelníku neleží uvnitř druhého. Označme tedy trojúhelníky *ABD* a *DBC*, přičemž *BD* je sdílená hrana.

Z geometrického nástinu na obr. 3.7.2 vyplývá, že čtyřúhelník *ABCD* bude konvexní právě tehdy, když žádný z krajních bodů *A* a *C* neopustí výseč, kterou vytínají hrany obsahující protější bod. To znamená, že bod *A* nesmí opustit výseč, kterou vytínají polopřímky *CB* a *CD*, a bod *C* nesmí opustit výseč, kterou vytínají polopřímky *AD* a *AB*.

To je možné rozpoznat za použití čtyř testů polohy bodu proti přímce. Bod *A* musí ležet „vpravo“ od polopřímky *CB* a „vlevo“ od polopřímky *CD*. Bod *C* musí ležet „vpravo“ od polopřímky *AD* a „vlevo“ od polopřímky *AB*. Determinanty v prvním a třetím testu tedy musí vyjít kladně a determinanty v druhém a čtvrtém testu musí vyjít záporně, aby byl čtyřúhelník *ABCD* konvexní. V případě, že alespoň jeden determinant vyjde s opačným znaménkem, znamená to, že je čtyřúhelník *ABCD* nekonvexní. Pokud žádný determinant nevyjde roven nule, znamená to, že je čtyřúhelník ostře konvexní/nekonvexní. Pokud je alespoň jeden determinant roven nule, značí to neostrou konvexnost, tj. tři body čtyřúhelníku *ABCD* leží na stejné přímce.



***Obr. 3.7.2**: Poloha krajního bodu vůči výseči vyťaté zbývajícími třemi body. (Vlevo: Bod* C *leží uvnitř výseče* BAD*, čtyřúhelník je konvexní. Vpravo: Bod* C *leží mimo výseč* BAD*, čtyřúhelník je nekonvexní.*

3.7.6 Průsečík dvou přímek

Mějme přímku *p*, na níž leží dva různé body $X_1=[x_1, y_1]$, $X_2=[x_2, y_2]$, a přímku *q*, na níž leží dva různé body $X_3=[x_3, y_3]$, $X_4=[x_4, y_4]$. Předpokládejme, že tyto přímky jsou různoběžné. Průsečík těchto přímek je potom v bodě $P=[x_p, y_p]$, kde hodnoty x_p a y_p lze spočítat pomocí následujících determinantů (převzato z [Wolf1]):

$$x_p=\frac{\begin{vmatrix}\begin{vmatrix}x_1 & y_1\\ x_2 & y_2\end{vmatrix} & x_1-x_2\\ \begin{vmatrix}x_3 & y_3\\ x_4 & y_4\end{vmatrix} & x_3-x_4\end{vmatrix}}{\begin{vmatrix}x_1-x_2 & y_1-y_2\\ x_3-x_4 & y_3-y_4\end{vmatrix}}, \qquad y_p=\frac{\begin{vmatrix}\begin{vmatrix}x_1 & y_1\\ x_2 & y_2\end{vmatrix} & y_1-y_2\\ \begin{vmatrix}x_3 & y_3\\ x_4 & y_4\end{vmatrix} & y_3-y_4\end{vmatrix}}{\begin{vmatrix}x_1-x_2 & y_1-y_2\\ x_3-x_4 & y_3-y_4\end{vmatrix}}. \tag{3.4, 3.5}$$

Vyjde-li jmenovatel zlomků roven nule, znamená to, že přímky jsou rovnoběžné a nemají tedy průsečík.

4 Implementace

4.1 Programové vybavení použité pro realizaci

Z důvodů přenositelnosti byl pro realizaci aplikace použit programovací jazyk C++.

Během vývoje aplikace bylo pro uživatelské rozhraní a vizualizace použito technologií GLU a GLUT, což je skupina knihoven pro aplikace využívající služeb OpenGL. Finální verze aplikace používá pro tyto účely technologii SDL.

Triangulační část aplikace byla kompletně vyvíjena ve vývojovém prostředí *Eclipse SDK 3.2*. Jelikož toto prostředí je primárně určeno pro programovací jazyk *Java*, bylo nutné použít zásuvný modul *Eclipse C/C++ Development Tooling* (CDT). Pro překlad aplikace pro operační systém *Microsoft Windows* bylo použito kompilátoru GCC obsaženého v softwarovém balíku *MinGW*. Pro překlad pro operační systém *Linux* byl rovněž použit kompilátor GCC.

4.2 Struktura aplikace

Zdrojový kód je rozčleněn do množství *.cpp* a *.h* souborů. Následující soubory tvoří geometrickou část aplikace.

- `Coord.h` – Definuje datový typ *Coord*, který je v síti použit pro reprezentaci veškerých reálných číselných údajů (souřadnic, vzdáleností). V současnosti je tento typ implementován jako *float* číslo.
- `Point.cpp`, `Point.h` – Obsahují definici třídy *Point*, která slouží k reprezentaci bodu v E^3, a metody pro práci s ním.
- `Edge.cpp`, `Edge.h` – Definuje datový typ *Edge*, který je v síti použit pro reprezentaci hran trojúhelníků.
- `Triangle.cpp`, `Triangle.h` – Obsahují definici třídy *Triangle*, která slouží k reprezentaci trojúhelníku, a metody pro práci s ním.
- `Grid.cpp`, `Grid.h` – Obsahují definici třídy *Grid*, která slouží k reprezentaci trojúhelníkové sítě, a metody pro práci s ní.
- `Delaunay.h` – Je součástí třídy *Grid*. Obsahuje metody pro konstrukci základní

Delaunayovy triangulace.

- `Constraints.h` – Je součástí třídy *Grid*. Obsahuje metody pro práci s vynucenými hranami.
- `TriangleWalking.h` – Je součástí třídy *Grid*. Obsahuje implementaci algoritmu procházky.
- `Geometry.cpp`, `Geometry.h` – Obsahuje funkce pro výpočty základních znaménkových testů, průsečíků apod.
- `Util.cpp`, `Util.h` – Obsahuje funkce pro pomocné matematické výpočty.
- `Constants.h` – Obsahuje definice globálních konstant, použitých na různých místech zdrojového kódu.

4.3 Datové struktury

Návrh datových struktur byl součástí práce kolegy Purcharta. Proto jsou podrobně popsány v [Purch1]. Jelikož je však většina implementačních detailů závislá na reprezentaci dat, je nezbytné použité struktury alespoň v krátkosti popsat.

4.3.1 Coord

Datový typ *Coord* představuje reálné číslo použitelné pro reprezentaci souřadnic, vzdáleností apod. V současné době je definovaná jako číslo *float*. Důvodem pro její zavedení bylo sjednocení použitých datových typů v rámci celé aplikace. Je tedy poměrně jednoduše možné celoplošně změnit rozsah používaných souřadnic např. za *double*.

4.3.2 Point

Třída *Point* představuje jeden bod či vrchol sítě. Jelikož je model terénu 3D, jedná se o bod se třemi souřadnicemi. Tyto souřadnice jsou uložené v poli tří hodnot *Coord* (tj. *float*). V dřívějších verzích aplikace byly tyto body uloženy jako tři samostatné proměnné. To však způsobilo značné problémy se škálovatelností v triangulačních algoritmech (viz 5.1). Proto bylo nutné časem reprezentaci změnit a hodnoty uložit do tříprvkového pole.

Dále tato třída obsahuje seznam hran vycházejících z daného vrcholu. Jelikož již od

počátku bylo zřejmé, že tato znalost je nezbytná pro fyzikální vrstvu aplikace, byla integrována přímo do třídy *Point*. Každý bod má tedy informaci o tom, které hrany z něj vycházejí. V průběhu změn sítě je nezbytné tuto informaci udržovat aktuální, aby síť zůstala konzistentní (viz 5.1). To bylo vyřešeno poměrně elegantně tím, že v konstruktoru třídy *Edge* se oběma jejím bodům tato hrana přidá do seznamu z něj vycházejících hran a v destruktoru se z nich zase odebere. Tento seznam se tak udržuje aktuální automaticky. Jediná nevýhoda tohoto přístupu spočívá v tom, že když uživatel zapomene zavolat destruktor hrany, zůstane tato hrana uložená v seznamech z bodů vycházejících hran, což vede k porušení konzistence sítě.

4.3.3 Edge

Třída *Edge* představuje orientovanou hranu v síti. Hrana je definována dvěma koncovými body *Point*. Body byly původně uloženy ve dvou samostatných proměnných, ale – podobně jako souřadnice bodu (viz 4.3.2) – musely být z důvodu zvýšení škálovatelnosti nahrazeny dvouprvkovým polem (viz 5.1).

Dále hrana obsahuje několik příznaků, z nichž nejdůležitější je `constrained`. Tento příznak udává, zda se jedná o vynucenou hranu či nikoliv. CDT tedy tento příznak používá pro rozlišení vynucených hran od obyčejných.

Při běžných implementacích Delaunayovy triangulace se hranová reprezentace vůbec nepoužívá a trojúhelníky se definují pouze pomocí tří bodů. V okamžiku návrhu datových struktur jsme však měli dobré důvody se domnívat, že bude výhodné zavést kromě bodové reprezentace i reprezentaci hranovou. Algoritmus CDT (oproti algoritmu základní DT) běžně operuje s termínem hrana. Je nutné uchovávat informaci o tom, které hrany jsou vynucené, uchovávat seznamy hran protínajících vynucenou hranu apod. Od fyzikální vrstvy jsme očekávali, že bude hrany jako samostatné objekty vyžadovat, jelikož veškeré přesýpání a eroze probíhá právě v bodech a na hranách trojúhelníků. Z těchto důvodů byly hrany do reprezentace trojúhelníků zavedeny.

Časem se ukázalo, že hranová reprezentace je poměrně nemotorná a velice náročná na údržbu, tzn. udržení v konzistentním stavu. Nejvíce je to patrné na implementaci algoritmu vynucování hran v CDT (viz 4.5.4).

4.3.4 Triangle

Třída *Triangle* představuje trojúhelník v síti. Trojúhelník je definován třemi body a třemi hranami. Dále udržuje informaci o tom, s jakými trojúhelníky přes kterou hranu sousedí. Tato informace je naprosto nezbytná, jinak by se nejednalo o trojúhelníkovou síť, ale o množinu trojúhelníků, u nichž neznáme vzájemné vztahy. Všechny tyto údaje jsou z důvodů škálovatelnosti uloženy v tříprvkových polích (viz 5.1).

První problém, který přinesla hranová reprezentace, byl následující: Hrana *Edge* je má počáteční a koncový bod a je tedy orientovaná. Trojúhelník – označme jej *ABC* – je uvažován pravotočivý, tzn. jeho hrany jsou orientovány následovně: *AB*, *BC* a *CA*. Aby měla hranová reprezentace vůbec smysl, sdílejí dva sousední trojúhelníky objekt hrany, přes níž spolu sousedí. Řekněme tedy, že přes hranu *AB* sousedí s trojúhelníkem *ABC* trojúhelník *ADB*. Tento trojúhelník má však hrany orientovány následovně: *AD*, *DB* a *BA*. V prvním trojúhelníku byla hrana orientována jako *AB* a ve druhém jako *BA*. Je tedy zřejmé, že tatáž hrana je v každém z trojúhelníků, které ji vlastní, orientována jinak. Proto byl zaveden příznak inverze hrany. Každý trojúhelník má pro každou svoji hranu uložený příznak, zda se má tato hrana uvažovat s opačnou orientací, než s jakou je uložena v objektu *Edge*. Toto řešení je celkem přímočaré a kromě komplikací na mnoha místech kódu (všude, kde se operuje s body v hranách, je nutné kontrolovat, zda není hrana invertovaná, a pokud ano, body se uvažují prohozené) s sebou nepřináší žádné další problémy.

4.3.5 Reprezentace polem a indexace modulo

Jak již bylo naznačeno v přecházejících odstavcích, byly problémy se škálovatelností vyřešeny zavedením reprezentace údajů *m*-prvkovým polem (u *Point* a *Triangle* je $m=2$ a u *Edge* je $m=1$). Např. u trojúhelníku (označme ho *ABC*) to znamená, že body *A*, *B* a *C* nejsou uloženy jako samostatné proměnné, ale v tříprvkovém poli, tj. nultým prvkem (s indexem 0) je bod *A*, první prvek je *B* a druhý prvek je *C*. Hrany jsou číslovány obdobně s tím, že hrana protilehlá bodu s indexem *i* má taktéž index *i*. K těmto údajům lze tedy přistupovat přes index.

Když tedy např. zjistíme, že hledaná hrana je *CA*, čili hrana s indexem 1, a potřebujeme získat hranu vpravo od ní, tj. hranu s indexem o 1 větším, stačí získaný index zvětšit o 1 a dostáváme hranu *AB* s indexem 2. Pokud bychom však totéž provedli s hranou *AB*, dostali

bychom index 3, který již není platný. Následující hrana je totiž *BC* a ta má index 0.

Jelikož platné indexy jsou pouze čísla od 0 do $m-1$, musel by uživatel při každém přístupu kontrolovat, zda hodnotu $m-1$ nepřekročil. To by značně znepřehlednilo kód, proto byla zavedena tzv. „modulo indexace“. Ta spočívá v tom, že objekty tříd *Point*, *Edge* a *Triangle* interně na požadovaném indexu provedou operaci modulo *m*. V důsledku toho může uživatel beze strachu vzít získaný index např. hrany, zvětšit ho o tolik, o kolik hran se potřebuje posunout, a modulo indexace zajistí, že dostane správnou hranu. Jediné, na co je potřeba dávat pozor, je to, že operace modulo funguje požadovaným způsobem pouze na oboru celých nezáporných čísel. Nechť i je celé číslo od 0 do $m-1$ a a je číslo od 0 do m. Požaduje-li uživatel hranu s indexem o a menším než i, musí místo indexu $i-a$ použít index $i-a+m$. V důsledku omezení lze psát odhad:

$$i-a+m \geq 0-m+m \geq 0 .$$

To znamená, že pokud nepožadujeme „oběhnout“ všechny hrany více jak jednou dokola, stačí k požadovanému indexu přičíst hodnotu *m*. Zjednodušeně řečeno, k získaným indexům je nutné vždy přičítat a nikdy ne od nich odčítat. Tento požadavek však není žádným způsobem omezující, pouze převádí pohyb po indexech doleva na pohyb doprava.

4.4 Uložení trojúhelníkové sítě v paměti

Třída *Grid* obsahuje dvě pole: `Triangle **triangles` a `Point **points`. Pole `triangles` slouží k uchování ukazatelů na veškeré trojúhelníky tvořící síť a pole `points` slouží k uchování všech bodů. K přidávání objektů do těchto polí slouží metody `void addPoint(Point *p)` a `void addTriangle(Triangle *triangle)`. Pro objekty třídy *Edge* žádný takový seznam zaveden není, jelikož nikdy není zapotřebí procházet všechny hrany. Objekty *Edge* jsou tedy umístěny volně v paměti a přistupuje se k nim přes jejich vlastnící trojúhelníky.

4.5 Numerické nepřesnosti

V použitých algoritmech je často nutné porovnávat číslo získané z výpočtu (např. determinant nějaké matice) s jiným číslem, např. s nulou. Kvůli nepřesné reprezentaci čísel s pohyblivou řádovou čárkou v počítači není možno tato čísla porovnávat přímo. Jelikož pro

vývoj aplikace nebyla použita žádná knihovna pro přesnou aritmetiku, bylo nutné pro veškerá taková porovnání použít ε-testy, tj. uvažovat, že rozdíl dvou čísel, který je v absolutní hodnotě menší než nějaké malé číslo ε, indikuje rovnost.

4.6 Implementace hlavních algoritmů

Tato sekce popisuje vlastní programovou realizaci algoritmů uvedených v sekci 3. Důrazem je kladen na implementační detaily, odvíjející se zejména od datové reprezentace, a komplikace, které v praxi nastaly a bylo je nutno vyřešit.

4.6.1 Algoritmus procházky

Cílem procházky je najít trojúhelník, v němž (geometricky) leží zadaný bod. Algoritmus je implementován v metodě `Triangle *recursiveGetTriangleAtPoint(Triangle *current, Point *p, Edge *parentEdge)` třídy `Grid`. Volání zvnějšku se však provádí pomocí metody `Triangle *getTriangleAtPoint(Point *p)` třídy `Grid`. Tato metoda obaluje vlastní rekurzivní algoritmus a před jeho vykonáním zajistí nastavení startovacího trojúhelníku.

Metoda `getTriangleAtPoint()` nejdříve zkontroluje, zda uživatel zvnějšku nastavil proměnnou `triangleWalkingStartingTriangle` na hodnotu různou od *NULL*. V případě, že ano, použije se tento trojúhelník jako startovní. V případě, že ne, provede se volba startovního trojúhelníku tak, že se z náhodné podmnožiny trojúhelníků sítě vybere ten nejbližší hledanému bodu (viz 3.4). Startovní trojúhelník se uloží do proměnné `startingTriangle`. Potom se zavolá vlastní rekurzivní algoritmus s následujícími parametry: `recursiveGetTriangleAtPoint(startingTriangle, p, NULL)` a vrácený trojúhelník se uloží do proměnné `triangle`. Potom se nastaví `triangleWalkingStartingTriangle` na výchozí hodnotu *NULL* a metoda vrátí trojúhelník `triangle`.

Rekurzivní metoda `Triangle *recursiveGetTriangleAtPoint(Triangle *current, Point *p, Edge *parentEdge)` se v každém zanoření přibližuje ze zadaného trojúhelníku `current` k cílovému, takže v konečném důsledku do cílového

trojúhelníku „dojde“ a vrátí jej. Bod `p` udává pozici, na níž leží hledaný trojúhelník. Hrana `parentEdge` je hrana, přes níž jsme se do trojúhelníku `current` dostali. Algoritmus probíhá následujícím způsobem:

1. Zkontrolujeme, zda `current` není *NULL*. Pokud ano, znamenalo by to, že jsme nějakým nedopatřením opustili síť. Něco takového by se však stát nikdy nemělo.
2. Do proměnné `parentEdgeIndex` uložíme index hrany `parentEdge` v trojúhelníku `current`.
3. Náhodně vybereme směr pohybu doleva nebo doprava.
4. Nastavíme `resultsOnEdge` na 0, `collidingEdgeIndexTmp` na –1 a `collidingPointIndex` na –1.
5. V cyklu projdeme všechny hrany trojúhelníka (kromě té s indexem `parentEdgeIndex`) a pro každou provedeme kroky 5a)–5d).
   a) Proměnnou `i` označíme index právě zkoumané hrany.
   b) Do proměnné `result` uložíme výsledek znaménkového testu polohy bodu `p` proti hraně s indexem `i`, přenásobený číslem –1 v případě, že je tato hrana invertovaná.
   c) Pokud je `result` rovný nebo blízký nule, provedeme kroky 5ci.–5civ.
      i. Inkrementujeme proměnnou `resultsOnEdge`.
      ii. Pokud `resultsOnEdge` je rovno dvěma, znamená to, že dva znaménkové testy vyšly rovné nebo blízké nule, tj. bod `p` leží v jednom z vrcholů aktuálního trojúhelníku `current`. Nastavíme tedy hodnotu collidingPointIndex na index tohoto vrcholu. To je číslo od 0 do 2, které se nerovná ani jednomu z indexů hran, pro něž vyšly testy rovné nebo blízké nule. Jelikož v proměnné `collidingEdgeIndexTmp` máme uložen index předchozí hrany, pro níž test vyšel rovný nebo blízký nule, lze psát, že hodnota `collidingPointIndex` je rovna číslu `3 - (collidingEdgeIndexTmp + i)`.
      iii. Nastavíme hodnotu `collidingEdgeIndexTmp` na `i`.

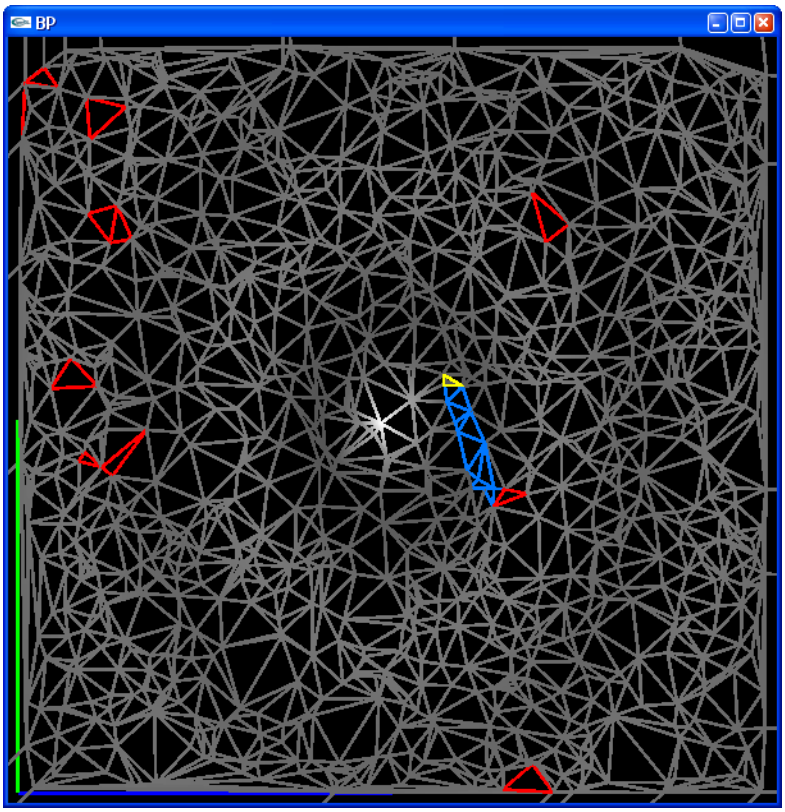
iv. Pokračujeme další iterací cyklu.

d) Pokud je `result` větší než nula, leží hledaný bod za hranou s indexem `i`. Odejdeme-li přes ni, přiblížíme se tak k hledanému bodu. To provedeme v krocích 5di. a 5dii.

i. Zavoláme metodu rekurzivně s následujícími parametry: `recursiveGetTriangleAtPoint(current->getNeighbour(i), p, current->getEdge(i))`. První parametr značí souseda trojúhelníku `current` přes hranu s indexem `i`. Druhým parametrem je opět hledaný bod. Třetí parametr říká, přes kterou hranu jsme se do trojúhelníku udaného jako první parametr dostali. Do `newTriangle` uložíme trojúhelník, který metoda po proběhnutí rekurze vrátí.

ii. Pokud trojúhelník `newTriangle` není roven *NULL*, ukončíme metodu a vrátíme jej jako návratovou hodnotu.

6. Pokud `resultsOnEdge` je rovno dvěma, znamená to, že bod `p` je identický s existujícím bodem v síti. V tom případě do veřejné proměnné `collidingPoint` nastavíme bod trojúhelníku `current` s indexem `collidingPointIndex`.
7. Pokud `collidingEdgeIndexTmp` je nezáporné číslo, znamená to, že přidávaný bod `p` leží na hraně trojúhelníka `current` s tímto indexem. Uložíme tedy tento index do veřejné proměnné `collidingEdgeIndex`. Tyto informace jsou potom dostupné např. metodě pro přidání bodu pomocí inkrementálního vkládání.
8. Pokud algoritmus proběhl až do tohoto kroku, znamená to, že všechny tři znaménkové testy vyšly záporné a bod `p` tedy leží uvnitř trojúhelníku `current`. Vrátíme tedy trojúhelník `current` a skončíme.



***Obr. 4.6.1**: Procházka a volba startovního trojúhelníku. (Červeně zvýrazněné trojúhelníky tvoří podmnožinu, z níž se vybíral startovní trojúhelník. Žlutý trojúhelník je cílový. Modré trojúhelníky jsou ty, které byly v průběhu procházky postupně navštíveny.)*

4.6.2 Konstrukce Delaunayovy triangulace

Algoritmus inkrementálního vkládání se poměrně dobře realizoval v praxi. Je implementován v metodě `Point *addPointAndTriangularizeGrid(Point *p)`.

Název metody je trochu matoucí, jelikož narozdíl od metody `addPoint()` nepřidává bod do pole `points`, avšak přidává bod, který v tomto poli již je, do triangulace. Tato metoda tedy vezme zadaný bod a pokud už na tomto místě nějaký bod je, přepíše mu výšku, jinak nový bod přidá do sítě a v jeho okolí síť ztriangularizuje. Metoda vrací bod, který se nachází na požadované pozici. To znamená, že v případě přidání nového bodu vrací metoda zadaný objekt *Point*, a v případě přepsání výškového údaje vrací objekt *Point*, jemuž byla výška přepsána.

Algoritmus je implementován následovně:

1. Nejdříve se pomocí metody `getTriangleAt()` zjistí, v jakém trojúhelníku

leží přidávaný bod `p`. Tento trojúhelník se uloží do `triangle`.

2. Metoda `getTriangleAt()` nastavuje veřejnou proměnnou `collidingPoint` na hodnotu různou od *NULL* v případě, že přidávaný bod koliduje s nějakým bodem v síti již přítomným. Pokud je v `collidingPoint` nastaven nějaký bod, přepíšeme jeho výšku na výšku bodu `p` a opustíme metodu, přičemž vrátíme bod `collidingPoint`.
3. Metoda `getTriangleAt()` nastavuje veřejnou proměnnou `collidingEdgeIndex` na nezápornou hodnotu v případě, že přidávaný bod leží na nějaké hraně. Tato hodnota udává index hrany v trojúhelníku `triangle`, na které bod `p` leží. V případě, že `(collidingEdgeIndex >= 0)`, je nutné rozdělit čtyřúhelník sdílející tuto hranu na čtyři trojúhelníky, provedeme tedy body 3a)–3n).
   a) Uložíme `triangle` do `X` a jeho souseda přes hranu s indexem `collidingEdgeIndex` do `Y`.
   b) Uložíme do `A`, `B` a `C` body trojúhelníka `X` (tak, že `B` je bod s indexem `collidingEdgeIndex`) a do `D` bod trojúhelníka `Y`, který není ani jedním z bodů `A`, `C`.
   c) Z bodů `A`, `B`, `C` a `p` vytvoříme nové trojúhelníky `q`, `r`, `s`, `t`.
   d) Vytvoříme hrany z bodů `A`, `B`, `C` a `p` hrany `j`, `k`, `l`, `m`.
   e) Nastavíme inverze hran s indexem 1 v trojúhelnících `q`, `r`, `s`, `t` na *true*.
   f) Pokud je (`isUsingCDT() == true`), zjistíme, zda některá z hran `j`, `k`, `l`, `m` odpovídá vyžadované vynucené hraně, a v případě, že ano, nastavíme příslušné hraně příznak `constrained` na *true*.
   g) Do proměnné `edge` uložíme hranu trojúhelníka `X` s indexem `collidingEdgeIndex`.
   h) Trojúhelníkům `q`, `r`, `s`, `t` postupně nastavíme hrany s indexy 0 a 1 na příslušné hrany z množiny `j`, `k`, `l`, `m` a hranu 2 získáme od sousedů. Hraně s indexem 2 musíme nastavit inverzi podle toho, jak byla nastavena

u souseda, od níž jsme ji získali.

i) Sousedům trojúhelníků přes hrany, které tvořily původní čtyřúhelník nastavíme nové trojúhelníky jako sousedy.

j) Zvýšíme `pointCount`, tj. počet bodů v síti, o 1.

k) Trojúhelníky `X` a `Y` nahradíme v seznamu `triangles` za trojúhelníky `q` a `r`. Přidáme trojúhelníky `s` a `t` za konec seznamu `triangles`.

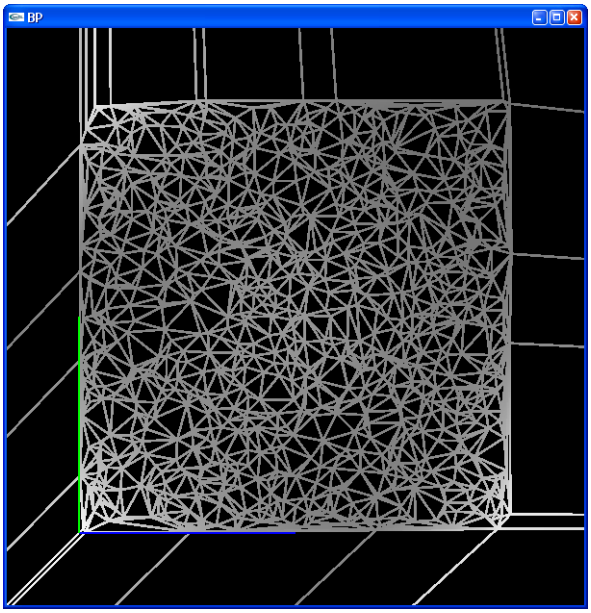
l) Smažeme trojúhelníky `X`, `Y` a hranu `edge`.

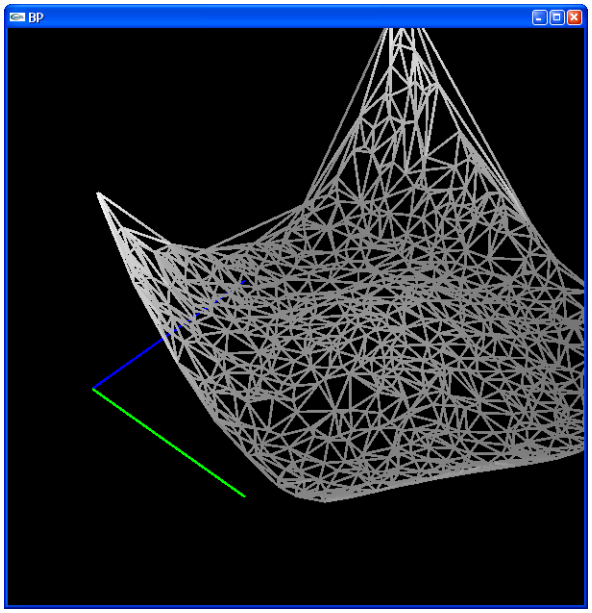
m) Na všechny vnější hrany nově vzniklých trojúhelníků, tj. na hrany s indexem 2 v trojúhelnících `q`, `r`, `s`, `t` zavoláme legalizaci.

n) Opustíme metodu a vrátíme bod `p`.

4. Pokud jsme se dostali sem, znamená to, že `(collidingEdgeIndex < 0)`. Musíme tedy trojúhelník rozdělit na tři nové. To se provede ekvivalentně jako v bodech 3a)–3n), s jediným rozdílem, že místo čtyř trojúhelníků vzniknou tři.

Na obr. 4.6.1 a obr. 4.6.2 je vidět výsledek Delaunayovy triangulace terénu.





***Obr. 4.6.1 a 4.6.2**: Hotová Delaunayova triangulace terénu při pohledu shora a tatáž triangulace ve 3D zobrazení zešikma. (Barva čar indikuje výšku terénu v daném místě.)*

4.6.3 Legalizace hrany

Cílem algoritmu je zajistit, aby všechny trojúhelníky v síti splňovaly Delaunayovo kritérium, kromě těch u nichž to není možné kvůli CDT.

Jedná se o rekurzivní algoritmus, který se lavinovitě šíří, až do okamžiku, kdy všechny konfrontované trojúhelníky splňují Delaunayovo kritérium (a nebo ho nesplňují, ale prohození patřičné hrany není možné, jelikož je vynucená CDT). Algoritmus je implementovaný v metodě `bool legalizeEdge(Edge *edge, Triangle *owner)`.

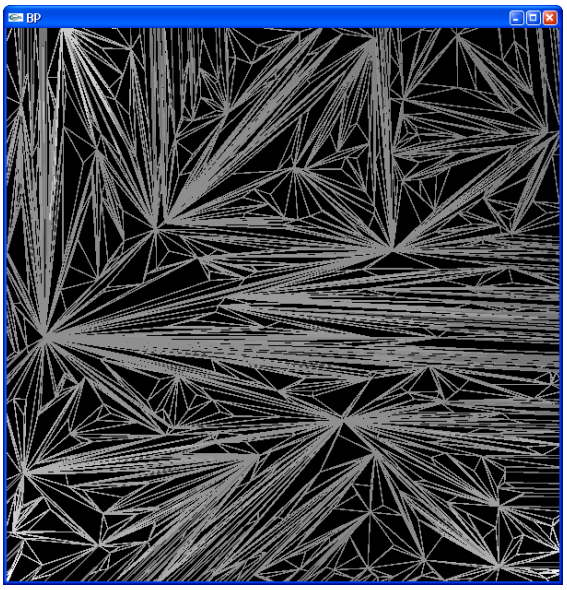
Principem algoritmu je provedení znaménkových testů, podle jejichž výsledků se rozhodne o tom, zda zkoumaná hrana vyhovuje Delaunayovu kritériu v kontextu dvou trojúhelníků, které ji sdílí. Pokud nevyhovuje, prohodíme v čtyřúhelníku tvořeném těmito trojúhelníky diagonálu, tj. inkriminovanou hranu smažeme a vytvoříme místo ní novou.

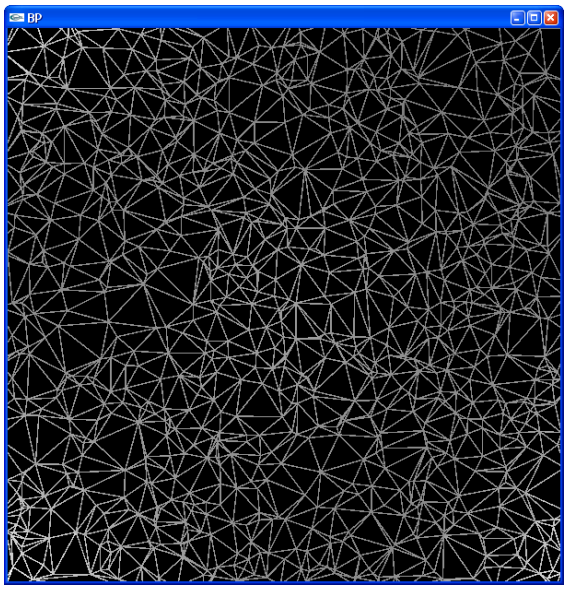
1. Zkontrolujeme, zda legalizovaná hrana nemá příznak `constrained` nastavený na *true*. V případě, že ano, ihned opouštíme metodu a vyskakujeme o úroveň z rekurze.
2. Do `edgeAtIndex` uložíme index hrany `edge` v trojúhelníku `owner`.
3. Pro test prázdné kružnice potřebujeme body `c1`, `c2`, `c3`. Uložíme do nich příslušné body trojúhelníku `owner` v odpovídajícím pořadí.
4. Do `neighbor` uložíme souseda trojúhelníku `owner` na indexu `edgeAtIndex`, tj. souseda přes hranu `edge`.
5. Do `s12` uložíme souseda trojúhelníku `owner` na indexu `edgeAtIndex + 1` a do `s23` souseda na indexu `edgeAtIndex + 2`.
6. Do `swapped1` uložíme hranu trojúhelníku `owner` na indexu `edgeAtIndex + 2`.
7. Do `pointAtIndex` uložíme index bodu `c3` v trojúhelníku `neighbor`.
8. Do `point` uložíme bod trojúhelníku `neighbor`, který není bodem trojúhelníka `owner`. Pro jeho nalezení využijeme orientace trojúhelníka jako obyčejně. Jedná se tedy o bod na indexu `pointAtIndex + 1`.
9. Do `s3p` uložíme souseda trojúhelníku `neighbor` na indexu

`pointAtIndex + 2` a do `sp1` souseda na indexu `pointAtIndex`.

10. Do proměnné `signum` uložíme výsledek znaménkového testu polohy bodu `point` vůči kružnici `c1`, `c2`, `c3`.
11. Pokud je `(signum <= 0)`, leží bod vně kružnice a nebo na ní. Ukončujeme tedy běh metody a vracíme *false*.
12. Jestliže je `(signum > 0)`, leží bod uvnitř kružnice a proto musíme hranu `edge` zrušit a místo ní rozdělit čtyřúhelník tvořený trojúhelníky `owner` a `neighbor` diagonálou, která nesplývá s `edge`. To se provede ve zbývajících krocích metody.
13. Vytvoříme novou hranu z bodu `point` do bodu `c2` a uložíme ji do `created`.
14. Pokud je `(isUsingCDT() == true)`, zjistíme, zda hrana `created` odpovídá vyžadované vynucené hraně, a v případě, že ano, nastavíme jí příznak `constrained` na *true*.
15. Změníme přímo vrcholy v trojúhelnících: vrchol `c3` nahradíme vrcholem `point` a vrchol `c1` vrcholem `c2`.
16. Do `edgeAtIndex` uložíme index hrany `edge` v trojúhelníku `neighbor`.
17. Zjistíme, která hrana v `neighbor` bude prohozena s hranou v `owner`. Je to hrana s indexem `edgeAtIndex + 2`. Uložíme ji do `swapped2`.
18. Pokud není `s23` rovno *NULL*, nastavíme tomuto trojúhelníku nového souseda na trojúhelník `neighbor`.
19. Pokud není `sp1` rovno *NULL*, nastavíme tomuto trojúhelníku nového souseda na trojúhelník `owner`.
20. Nastavíme inverzi sdílené hrany v trojúhelníku `owner` (to je ta s indexem o 1 větším než `swapped1`) na hodnotu, kterou má trojúhelník `neighbor` uloženou pro hranu `swapped2`.
21. Obdobně nastavíme inverzi sdílené hrany v trojúhelníku `neighbor` (ta s indexem o 1 větším než `swapped2`) na hodnotu, kterou má trojúhelník `owner` uloženou pro hranu `swapped1`.

22. Nastavíme `edgeAtIndex` na index `swapped1` v trojúhelníku `owner`.
23. Nastavíme trojúhelníku `owner` hrany na správné místo, tj. hranu `swapped2` na index `edgeAtIndex + 1` a hranu `created` na `edgeAtIndex`.
24. Nastavíme inverzi hrany `created` v trojúhelníku `owner` na *true*.
25. Nastavíme trojúhelníku `owner` správně sousedy, tj. souseda na indexu `edgeAtIndex + 1` na `sp1` a na indexu `edgeAtIndex` na `neighbor`.
26. Provedeme kroky 22–25 ekvivalentně pro trojúhelník `neighbor`, tj. pouze s tím rozdílem, že místo `owner` uvažujeme `neighbor`, místo `swapped1` uvažujeme `swapped2`, `swapped1` místo `swapped2` a `s23` místo `sp1`.
27. Označíme hranu `edge` jako smazanou, tj. nastavíme `deleted` na *true*.
28. Pokud může být hrana smazána rovnou (příznak `canBeDeleted`, který se nastavuje v CDT), smažeme ji. V opačném případě se hrana smaže až později manuálně. To se však nesmí zapomenout provést, jelikož by to způsobilo nekonzistenci dat (viz 4.3.2).
29. Posledním krokem je lavinovité rekurzivní zavolání legalizace na všechny hrany trojúhelníků kromě té, jež byla legalizací právě vytvořena.





***Obr. 4.6.3 a 4.6.4**: Vliv legalizace na kvalitu sítě. (Vlevo: triangulace vytvořená bez pomoci legalizace. Vpravo: triangulace stejné množiny bodů vytvořená za pomoci legalizace.)*

Již na tomto algoritmu je vidět, jak hranová reprezentace komplikuje implementaci. Z teoretického pohledu přitom algoritmus pouze otestuje úhlopříčku na Delaunayovo kritérium a případně ji prohodí.

Na obr. 4.6.3 a obr. 4.6.4 je vidět rozdíl mezi obecnou triangulací a Delaunayovou triangulací (konstruovaných pomocí inkrementálního vkládání bodů). Z obrázků je zřejmé, že bez použití legalizace by vznikaly velice špatně podmíněné trojúhelníky, které by v potenciálních aplikacích mohly způsobovat velké numerické problémy. Síť vytvořená Delaunayovou triangulací naopak obsahuje velmi dobře podmíněné trojúhelníky, takže je z numerického hlediska velice výhodná.

4.6.4 Konstrukce Delaunayovy triangulace s ohraničeními

Metoda pro vynucení hrany z bodu *A* do bodu *B* je implementována ve třídě *Grid* jako `void addConstrainedEdge(Point *a, Point *b)`.

Nastává zde však problém, zejména u rozlehlejších sítí. Spojíme-li totiž body *A* a *B* hranou přímo, může se stát, že na této hraně (nebo velmi blízko ní) bude ležet jeden nebo dokonce několik bodů. To není přípustné, proto není možné vynutit tak dlouhou hranu přímo. Je však možné ji rozdělit na fragmenty mezi jednotlivými body. Na těchto fragmentech již žádné mezilehlé body neleží a proto je vynutit lze.

Z toho důvodu je metoda `addConstrainedEdge()` vlastně jen obal metody `void addConstrainedEdgeFragment(Point *a, Point *b)`. Případné rozdělování hrany na fragmenty zajišťuje metoda `addConstrainedEdge()`, zatímco metoda `addConstrainedEdgeFragment()` se stará o vlastní vynucení jednotlivých fragmentů.

Metoda `void addConstrainedEdge(Point *a, Point *b)` tedy funguje následovně:

1. Nastavíme bod `lastPoint` na `a`.
2. Dokud bod `lastPoint` nesplyne s bodem `b`, opakujeme krok 3.
3. Zavoláme `addConstrainedEdgeFragment(lastPoint, b)` a potom nastavíme `lastPoint` na bod, který tato metoda vrátí.

Bod, který metoda `addConstrainedEdgeFragment()` vrací, udává místo, kam až se podařilo hranu vynutit. Z toho plyne, že je poté nutné zavolat metodu `addConstrainedEdgeFragment()` pro úsek od tohoto bodu až do bodu `b`.

Metoda `addConstrainedEdgeFragment()` je oproti teoretickému algoritmu popsanému v sekci 3.6 relativně komplikovaná, jelikož bylo nutné ošetřit množství speciálních a singulárních případů. Metoda je poměrně dlouhá a složitá, proto nebude popsána detailně, ale budou zdůrazněny její hlavní myšlenky.

1. Nastavíme veřejnou proměnnou `CDT_EdgeCreated` na *false*. (Tato proměnná značí, zda v průběhu CDT již byla vytvořena požadovaná vynucená hrana. Na *true* je nastavována pouze zvnějšku jako vedlejší efekt funkce `edgeIsConstrainedEdge()`. Tato funkce se používá během v DT během vkládání bodu a v CDT ke zjištění, zda hrana s níž právě pracujeme, je hledaná vynucená hrana. Toto řešení bylo zvoleno proto, že přímá komunikace mezi DT a CDT není uspokojivě realizovatelná. Jeho výhoda je v tom, že je velmi rychlé.)
2. Nastavíme proměnnou `collidingPoints` na 0.
3. Přidáme oba body vynucované hrany do sítě. Při vkládání bodů se vytvoří nezbytné hrany a v případě, že některá z vytvořených hran je hledaná vynucená hrana, se nastaví `CDT_EdgeCreated` na *true*. Po každém přidání bodu do sítě inkrementujeme `collidingPoints`, pokud `collidingPoint` není *NULL*.
4. Pokud `collidingPoints` je rovno dvěma, znamená to, že oba body již v síti byly. Proto otestujeme, zda jsou spojeny hranou, a v případě, že ano, nastavíme ji jako vynucenou, příznak `CDT_EdgeCreated` vrátíme na *false*, skončíme a vrátíme hodnotu `b`, čili očekávaný koncový bod.
5. Otestujeme hodnotu `CDT_EdgeCreated`. Pokud je *true*, byla hrana v průběhu vkládání bodů vytvořena, a byla tedy již také nastavena jako vynucená, a proto lze vrátit příznak `CDT_EdgeCreated` na *false*, skončit a vrátit hodnotu `b`.
6. V případě, že `CDT_EdgeCreated` je *false*, hrana v síti přítomna nebyla ani nově nevznikla. Musíme ji tedy vynutit. Začneme tím, že vytvoříme nový objekt

Edge, začínající v bodě `a` a končící v bodě `b`. Vytvoříme ji však pomocí konstruktoru, který nemění seznam hran vycházející z koncových bodů. Tato hrana totiž v síti vůbec nezůstane. Výsledná vynucená hrana se totiž vytvoří během prohazování diagonál. Tento objekt však potřebujeme používat jako parametr znaménkových testů apod.

7. Pomocí procházky zjistíme, v jakém trojúhelníku leží počáteční bod `a` vynucené hrany. Tento trojúhelník uložíme do proměnné `currentTriangle`.
8. Pomocí algoritmu „obcházení bodu“ popsaného v sekci 3.6 nalezneme startovní trojúhelník (začneme v trojúhelníku `currentTriangle` a obcházíme bod `a`) a uložíme jej do `currentTriangle`.
9. Pokud během hledání startovního trojúhelníku zjistíme, že na vynucené hraně leží testovaná hrana, nejsme schopni nalézt použitelný startovní trojúhelník. Ukončíme tedy fragment vynucené hrany tím, že skončíme a vrátíme ten bod testované hrany, který nesplývá s `a`. Metoda `addConstrainedEdge()` zvenku zajistí, že bude vynucen i zbytek hrany.
10. Vytvoříme prázdný seznam protínajících se hran `intersecting`. K jeho implementaci použijeme `std::list<Edge*>`.
11. Vytvoříme prázdnou datovou strukturu pro uchovávání trojúhelníků vlastnících hrany. Pojmenujeme ji `owners` a k její implementaci použijeme mapu `std::map<Edge*, Triangle*, gr_edge>`, kde `gr_edge` je struktura umožňující porovnat dva ukazatele na objekt *Edge* podle jejich numerické hodnoty. Jedná se o stromovou strukturu, ve níž je klíčem ukazatel na objekt *Edge* a data jsou představována trojúhelníkem vlastnícím tento objekt hrany.
12. Pomocí algoritmu „přelézání“ po trojúhelnících popsaného v sekci 3.6 nalezneme hrany, které protínají vynucenou hranu a nesdílí žádný koncový bod s žádným z jejích koncových bodů. Začínáme ve startovním trojúhelníku `currentTriangle`.
13. Každou nalezenou hranu ukládáme na začátek seznamu `intersecting`. Pro každou uloženou hranu musíme uložit i informaci o tom, jaký trojúhelník tuto

hranu vlastní. Kdybychom tuto informaci neměli explicitně uloženou, nebyli bychom později schopni zjistit, v jakém trojúhelníku se hrana nachází, aniž bychom museli použít prohledávání pomocí procházky.

14. Pokud během hledání protínajících se hran kdykoli zjistíme, že nějaký bod aktuálního trojúhelníku leží na vynucené hraně, nastavíme `b` na tento bod, čímž aktuální fragment vynucené hrany předčasně ukončíme. To způsobí, že později metoda `addConstrainedEdge()` zvnějšku zavolá i pro zbývající část fragmentu metodu `addConstrainedEdgeFragment()`.
15. Všem hranám v seznamu `intersecting` zrušíme příznak `constrained`, aby nezpůsobovaly problémy při prohazování. Výsledný efekt je ten, že když nová vynucená hrana protíná starší, u staré hrany se zruší vynucení a ta se podřídí nové hraně. Nemusíme se tedy explicitně starat o křížící se vynucené hrany, což je výhodné protože zjišťování intersekcí obecně vede na soustavy rovnic a ty jsou výpočetně pomalé.
16. Vytvoříme prázdný seznam nově vytvořených hran `created`. K jeho implementaci taktéž použijeme `std::list<Edge*>`.
17. Dokud není seznam `intersecting` prázdný, opakujeme kroky 17a)–17d).
    a) Z konce seznamu `intersecting` vybereme hranu a uložíme ji do proměnné `diagonal`.
    b) Z mapy `owners` zjistíme trojuhelník, v němž leží hrana `diagonal`, uložíme ho do `owner` a souseda tohoto trojúhelníku přes hranu `diagonal` označíme `neighbor`.
    c) Pokud je čtyřúhelník tvořený trojúhelníky `owner` a `neighbor` konvexní, provedeme kroky 17ci.–17cix.
        i. Prohledáme seznam `created` a pokud obsahuje hranu `diagonal`, smažeme ji z něj.
        ii. Prohodíme diagonálu ve čtyřúhelníku tvořeném trojúhelníky `owner` a `neighbor`. To se provede obdobně jako v algoritmu legalizace. Novou

diagonálu uložíme do proměnné `newEdge`.

iii. Po prohození smažeme objekt `diagonal`.

iv. Z mapy `owners` smažeme položku s klíčem `diagonal`.

v. Pomocí metody `edgeIsConstrainedEdge()` otestujeme hranu newEdge, zda není vynucenou hranou. Pokud je, zavoláme metodu `updateMap(owner, neighbor)`, která obnoví konzistenci mapy. Toho se dosáhne tím, že projdeme všechny položky mapy a u těch, jejichž trojúhelník je nastaven na `owner` nebo `neighbor`, zkontrolujeme, zda je tato informace pro danou hranu platná. Pokud není, nastavíme této hraně jako vlastníka správný trojúhelník. Tím je `neighbor` v případě, že původně byl v mapě uložen `owner`, a naopak. Po obnovení konzistence mapy můžeme cyklus z kroku 17 přerušit, jelikož hrana již byla vynucena.

vi. Otestujeme `newEdge`, zda pořád neprotíná vynucenou hranu. Pokud ano (a nesdílí žádný koncový bod s vynucenou hranou), vrátíme ji zpět do seznamu `intersecting` a do mapy `owners` uložíme pro hranu `newEdge` trojúhelník `owner`.

vii. Zamkneme `newEdge` proti smazání tím, že jí nastavíme příznak `canBeDeleted` na *false*. V lavinovitě se šířící legalizaci si totiž nemůžeme dovolit ji smazat. To by okamžitě způsobilo porušení konzistence dat, jelikož bychom v seznamech `intersecting` a `created` mohli mít ukazatele na již neexistující objekty. Z toho důvodu je nutné hranu zamknout. Legalizace zamčenou hranu místo zrušení jenom označí jako smazanou nastavením příznaku `deleted` na *true*. Vlastní mazání provedeme později manuálně.

viii. Přidáme hranu `newEdge` do seznamu `created` a nastavíme ji v mapě `owners` vlastníka na trojúhelník `owner`.

ix. Opravíme mapu pomocí `updateMap(owner, neighbor)`, stejně

jako v kroku 17cv.

d) Jestliže je čtyřúhelník tvořený trojúhelníky `owner` a `neighbor` nekonvexní, vrátíme hranu `diagonal` na začátek seznamu `intersecting`.

18. Nyní musíme provést legalizaci nově vytvořených hran.
19. Vytvoříme prázdný seznam hran určených pro odemknutí `edgesToReset`. K jeho implementaci použijeme opět `std::list<Edge*>`.
20. Dokud není seznam `created` prázdný, provádíme kroky 20a)–20g).
    a) Vyber hranu `e` z konce seznamu `created`.
    b) Vlož tuto hranu na začátek seznamu `edgesToReset`.
    c) Pokud má hrana nastaven příznak `deleted`, přeskakujeme ji. Není možno ji legalizovat, jelikož již není součástí sítě, a vzápětí ji smažeme z paměti. Přeskočení provedeme skokem na bod 20a) a pokračováním další hranou.
    d) Zjistíme z mapy trojúhelník vlastnící hranu `e` a uložíme ho do `t`.
    e) Zavoláme legalizaci na hranu `e` v trojúhelníku `t`.
    f) Důsledky legalizace nemůžeme předem odhadnout a můžou se týkat jakékoli hrany v seznamu `created`. Na tomto místě je tedy nezbytně nutné překontrolovat mapu. Nemáme však (a ani efektivně nemůžeme mít) uložené veškeré informace, které bychom pro tuto akci potřebovali. Proto musíme tuto obnovu provést přímo pomocí hledání patřičných trojúhelníků. Zdálo by se, že to bude velmi výpočetně náročné, ovšem je nutné si uvědomit, že se hledání provádí jenom pro několik málo hran a hledané trojúhelníky jsou velmi blízko startovním. Pro každou hranu v mapě tedy provedeme kroky 20fi.–20fiii.
        i. Zkontrolujeme, zda trojúhelník uložený v mapě skutečně vlastní danou hranu. Pokud ano, je vše v pořádku a můžeme pokračovat další hranou v kroku 20fi. Pokud ne, pokračujeme následujícím krokem.
        ii. Pomocí procházky najdeme trojúhelník, který obsahuje počáteční bod hrany. Tento trojúhelník může, ale také nemusí obsahovat i koncový bod

této hrany. Proto musíme „obejít“ trojúhelníky kolem počátečního bodu (podobně jako v kroku 8) a najít trojúhelník, který hranu vlastní.

iii. Nalezený trojúhelník nastavíme do mapy jako vlastníka hrany.

g) Nyní musíme odemknout všechny hrany v seznamu `edgesToReset`, aby mohly být normálně mazány jako dříve. Dokud tedy není seznam `edgesToReset` prázdný, provádíme kroky 20gi.–20giii.

i. Z konce seznamu `edgesToReset` vybereme hranu `e`.

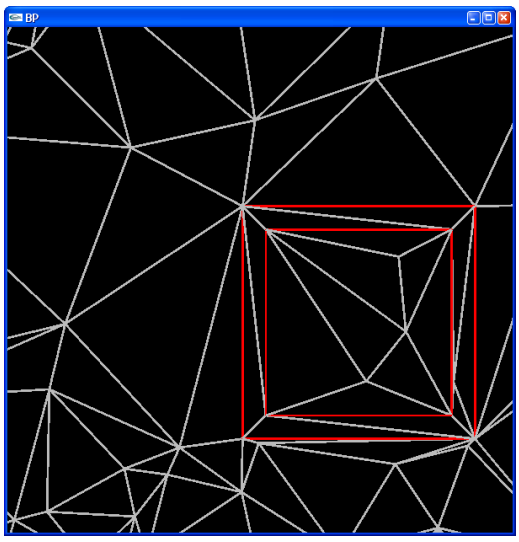
ii. Odemkneme hranu `e` nastavením příznaku `canBeDeleted` na *true*.

iii. Pokud má hrana `e` nastaven příznak `deleted` na *true*, můžeme ji na tomto místě konečně fyzicky smazat z paměti.

21. Nastavíme `CDT_EdgeCreated` na *false* a vrátíme koncový bod fragmentu, tj. bod `b`.

Je zřejmé, že udržet síť a všechny pomocné datové struktury (seznamy a mapu) v konzistentním stavu je při současné datové reprezentaci velmi obtížné.

Na obr. 4.6.5 je vidět, jak lze použít vynucování hran v CDT např. pro vytvoření otisku razítka. Touto problematikou se podrobně zabývá kolega Purchart v [Purch1].



Obr. 4.6.5: Vynucení otisku razítka pomocí CDT. (Červeně vyznačené hrany představují hrany vynucené v triangulaci.)

5 Experimenty a výsledky

5.1 Postup práce během vývoje

Aplikace byla vyvíjena týmem tří lidí. To přineslo v první řadě velkou režii způsobenou nutností komunikace mezi členy týmu a dohadování se navzájem. Bylo nutné pořádat společné konzultace, pro něž se vždy musel dohodnout termín v průniku volných časů jednotlivých členů týmu. Dále způsobila práce v týmu komplikace při slučování dílčích částí kódů od různých členů, jelikož během vývoje nebyl použit žádný systém jako SVN, který by toto zautomatizoval. Na druhou stranu byla spolupráce v některých situacích velmi přínosná, zejména během implementace kritických částí aplikace.

Práce na geometrické části aplikace (tj. triangulačním jádru) byla z velké míry záležitostí implementace známých algoritmů. Tyto algoritmy však byly implementovány nad nepříliš často používanou datovou reprezentací, což přineslo znatelné komplikace. Nezřídka se stávalo, že algoritmus byl implementován během poměrně krátké doby a pak mnohem delší dobu zabralo ladění. Bylo totiž nutno ošetřit velké množství speciálních a singulárních případů, z nichž mnohé byly značně netriviální a předem nepředpokládané. Velké množství chyb v aplikaci bylo způsobeno tím, že při nějaké speciální konfiguraci dat došlo k porušení jejich konzistence. Z toho důvodu byla i implementace známých algoritmů poměrně průkopnickou záležitostí.

Během vývoje geometrické části aplikace se nevyskytlo mnoho slepých uliček. Jedním z nepříjemných problémů byla nízká škálovatelnost přístupu k datovým strukturám. Původně nebylo možno přistupovat např. k hranám trojúhelníka pomocí indexů, ale musela se pro každou hranu volat speciální metoda. Zpočátku se nezdálo, že by to mohlo způsobit problémy. Avšak při implementaci algoritmů jako legalizace hran se zjistilo, že na identifikaci hrany v trojúhelníku je třeba použít tři podmínky, pro nalezení hrany v sousedním trojúhelníku, která je shodná s nějakou hranou v původním trojúhelníku, je potřeba použít podmínek devět. Pokud bychom chtěli přistupovat ještě k hranám sousedního trojúhelníku sousedního trojúhelníku, potřebovali bychom podmínek již dvacet sedm. Jedná se tedy o exponenciální explozi, která byla pro další vývoj aplikace netolerovatelná. Navíc, pokud již jednou byla hledaná hrana nalezena, nebylo možno si zapamatovat, na které pozici se nacházela, a tak se tytéž podmínky opakovaly několikrát za

sebou. Proto bylo nutno změnit interní reprezentaci datových struktur. Veškeré mnohonásobné prvky byly oindexovány a byl zaveden modulo přístup (viz 4.3.5). Např. zdrojový kód zmíněné legalizace se tím zkrátil na méně než jednu čtvrtinu původní délky a výrazně se tak zpřehlednil.

Triangulační jádro bylo původně vyvíjeno jako implementace základní Delaunayovy triangulace, bez znalosti principu vynucování hran v CDT. Podpora pro vynucování hran byla doimplementována později, až v okamžiku, kdy byla konstrukce základní Delaunayovy triangulace plně odladěna. To má za následek, že některé akce jsou prováděny lehce „přes ruku“. Např. inkrementální vkládání s CDT komunikuje prostřednictvím veřejných proměnných. V CDT se pro zjištění, zda hrana již v síti vznikla, používá metoda s vedlejším efektem. To lehce znepřehledňuje určité části kódu.

5.2 Experimenty s předzpracováním množiny bodů

Experimentálně bylo zjištěno, že doba potřebná k vytvoření triangulace je závislá na pořadí, v němž body vkládáme. To je způsobeno tím, že při některých konfiguracích bodů je potřeba během legalizace prohazovat více diagonál.

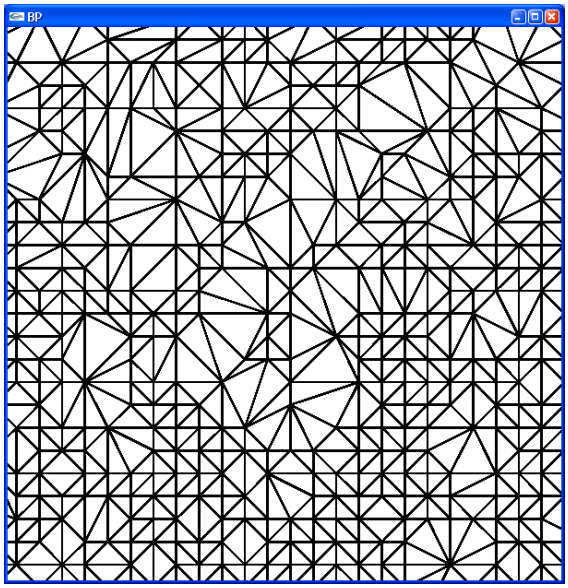
Z experimentů vyplynulo pozorování, že jsou-li body seřazené podle polohy (tj. vkládáme je postupně vedle sebe), je nutné prohazovat mnohem více diagonál, než když se vkládají body „napřeskáčku“. Je to patrně způsobené tím, že každý nový bod se vkládá do prázdného prostoru (myšleno do prostoru bez bodů), který je pokryt velkým množstvím podlouhlých trojúhelníků, které se stýkají v jednom z bodů velkého trojúhelníka (*supertriangle*) obepínajícího celou množinu bodů (viz obr. A.1). Vložením nového bodu do této oblasti vznikne několik trojúhelníků, které s největší pravděpodobností nesplňují Delaunayovo kritérium prázdné kružnice. To má za následek mohutnou vlnu legalizace.

Tato vlna je vlastně způsobena vkládáním bodu do prázdného prostoru (viz obr. A.2), což mě přivedlo na myšlenku vkládat body v pořadí klesající vzdálenosti od středu. Tím by se hned na začátku vytvořila kolem množiny bodů jakási konvexní obálka (viz obr. A.3 a A.4). Zamezilo by se tak vzniku velkých dlouhých trojúhelníků, jejichž jeden bod je bodem všeobjímajícího trojúhelníka. K měření vzdáleností by bylo vhodné použít první vektorovou normu.

V praxi se tato teorie potvrdila, jelikož se ukázalo, že toto seřazení skutečně vede na

výrazné snížení počtu prohazovaných hran při legalizacích. Takto seřazená množina však stále vykazuje rysy prostorového seřazení. Vytváření obálky zvnějšku má nevýhodu v tom, že uvnitř obálky zůstává volný prostor bez bodů, v němž opět vznikají dlouhé velké trojúhelníky. Tento problém lze minimalizovat tím, že se body seřadí podle klesající vzdálenosti a následně lehce náhodně promíchají. Hned od počátku se tak začínají přidávat body i do oblasti uvnitř obálky, čímž se do značné míry zamezí vzniku velkých dlouhých trojúhelníků (viz obr. A.5 a A.6). Dosažený výsledek je tak ještě lepší než u pouhého seřazení.

Porovnání časů nutných k vytvoření triangulace množiny bodů seřazené různými způsoby přineslo zajímavé výsledky. Jak je vidět z tabulek 5.2.1 a 5.2.2, dílčí časy pro procházky a zbytek triangulačního algoritmu jsou pro všechny metody velmi podobné. Časy pro prohazování hran v legalizacích se však výrazně liší. Jednoznačně nejhůře test dopadl pro prostorově seřazenou množinu. Množina seřazená podle vzdálenosti dopadla mnohem lépe, avšak nejlépe dopadla metoda náhodného zamíchání (viz obr. A.7 a A.8) a metoda seřazení podle vzdálenosti a následného lehkého promíchání. Pro datovou množinu „Crater Lake“ dopadla jednoznačně nejlépe sofistikovaná metoda, jak v čase nutném pro legalizace, tak v celkovém čase. Naproti tomu u náhodných dat vyšel čas nutný pro legalizace u obou metod přibližně nastejno, avšak z hlediska celkového času byla nejrychlejší náhodná metoda.



***Obr. 5.2.1**: Triangulace téměř pravidelně rozmístěných bodů. (Část modelu „Crater Lake“. Rysy připomínající pravidelnou síť jsou poměrně nápadné.)*

Pořadí bodů	Celkový čas [s]	Procházky [s]	Legalizace [s]	Zbytek [s]
Původní (seřazené podle polohy)	9,77	1,77	7,21	0,76
Seřazené podle vzdálenosti	3,94	1,71	1,40	0,79
Seřazené podle vzdálenosti a lehce promíchané	2,97	1,64	0,60	0,70
Náhodné	3,27	1,79	0,69	0,73

***Tabulka 5.2.1**: Časy nutné k vykonání jednotlivých částí triangulačního algoritmu pro prvních 30000 bodů z datové množiny „Crater Lake“ (100000 bodů) vkládané v různém pořadí. (Pořadí bodů, celkový čas pro vytvoření triangulace a dílčí časy – čas spotřebovaný algoritmem procházky, čas spotřebovaný na legalizace hran, čas spotřebovaný zbytkem triangulačního algoritmu. Veškeré hodnoty byly měřeny 5x a následně zprůměrovány. Hodnoty byly naměřeny na osobním počítači s procesorem AMD Athlon 1.20GHz a 768MB RAM.)*

Pořadí bodů	Celkový čas [s]	Procházky [s]	Legalizace [s]	Zbytek [s]
Seřazené podle vzdálenosti	0,97	0,39	0,41	0,16
Seřazené podle vzdálenosti a lehce promíchané	0,79	0,38	0,24	0,14
Náhodné	0,74	0,33	0,24	0,15

***Tabulka 5.2.2**: Časy nutné k vykonání jednotlivých částí triangulačního algoritmu pro prvních 10000 z 25000 náhodně vygenerovaných bodů vkládaných v různém pořadí. (Pořadí bodů, celkový čas pro vytvoření triangulace a dílčí časy – čas spotřebovaný algoritmem procházky, čas spotřebovaný na legalizace hran, čas spotřebovaný zbytkem triangulačního algoritmu. Veškeré hodnoty byly měřeny 5x a následně zprůměrovány. Hodnoty byly naměřeny na osobním počítači s procesorem AMD Athlon 1.20GHz a 768MB RAM.)*

Patrně tedy jsou časy závislé na charakteru datové množiny (míra pravidelnosti sítě apod.), počtu bodů v datové množině, míra promíchanosti sítě (viz obr. 5.2.1). Nastavit konstantu promíchanosti sítě fixně není jednoduché. Navíc pro některé datové množiny bude pravděpodobně náhodná metoda fungovat lépe než sofistikovaná. V obecném případě je tak patrně dobrou volbou volit pořadí zcela náhodné.

Ve fázi předzpracování tak provádíme pouhé náhodné zamíchaní bodů (*randomization*).

5.3 Výkonnostní testy

Časy nutné k vytvoření triangulace na množině náhodně vygenerovaných bodů a paměťová náročnost aplikace jsou zachyceny v tab. 5.3.1. Časy nutné k vynucení 100 náhodně vygenerovaných hran pomocí algoritmu CDT jsou zachyceny v tab. 5.3.2.

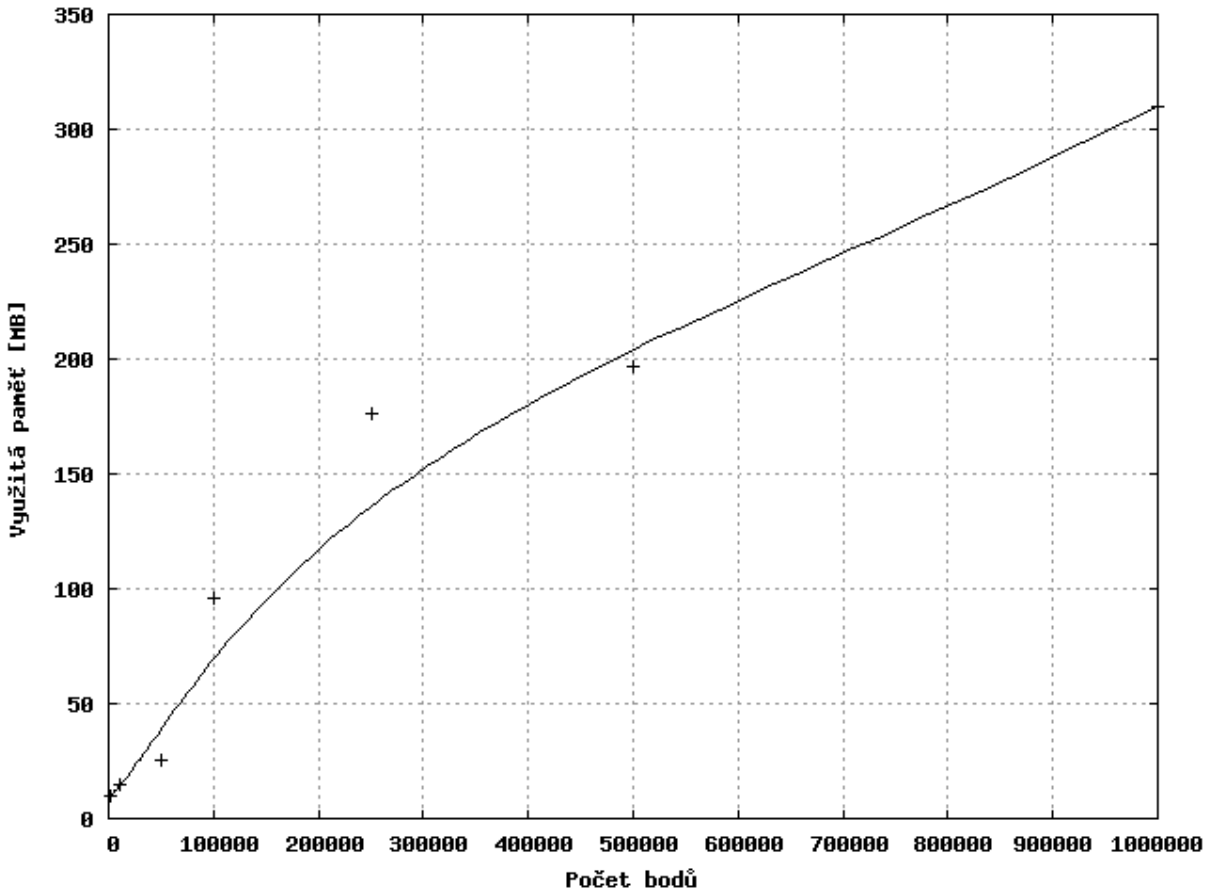
Počet bodů	Celkový čas [s]	Procházky [s]	Legalizace [s]	Zbytek [s]	Využitá paměť [MB]
1000	0,02	0,00	0,02	0,00	9,50
10000	0,23	0,16	0,05	0,03	15,00
50000	2,45	1,39	0,31	0,72	25,00
100000	6,19	4,03	0,58	1,53	96,00
250000	18,91	15,12	0,86	2,82	176,00
500000	51,11	42,27	0,93	7,66	197,00
1000000	139,09	118,02	1,07	19,66	310,00

Tabulka 5.3.1: *Časy nutné k vykonání jednotlivých částí triangulačního algoritmu a využití paměti pro různé počty vkládaných náhodně vygenerovaných bodů. (Počet vkládaných bodů, celkový čas pro vytvoření triangulace a dílčí časy – čas spotřebovaný algoritmem procházky, čas spotřebovaný na legalizace hran, čas spotřebovaný zbytkem triangulačního algoritmu, celkové množství paměti alokované aplikací. Veškeré hodnoty byly měřeny 3x a následně zprůměrovány. Hodnoty byly naměřeny na osobním počítači s procesorem Intel Core 2 Duo 2x1.66GHz a 1GB RAM.)*

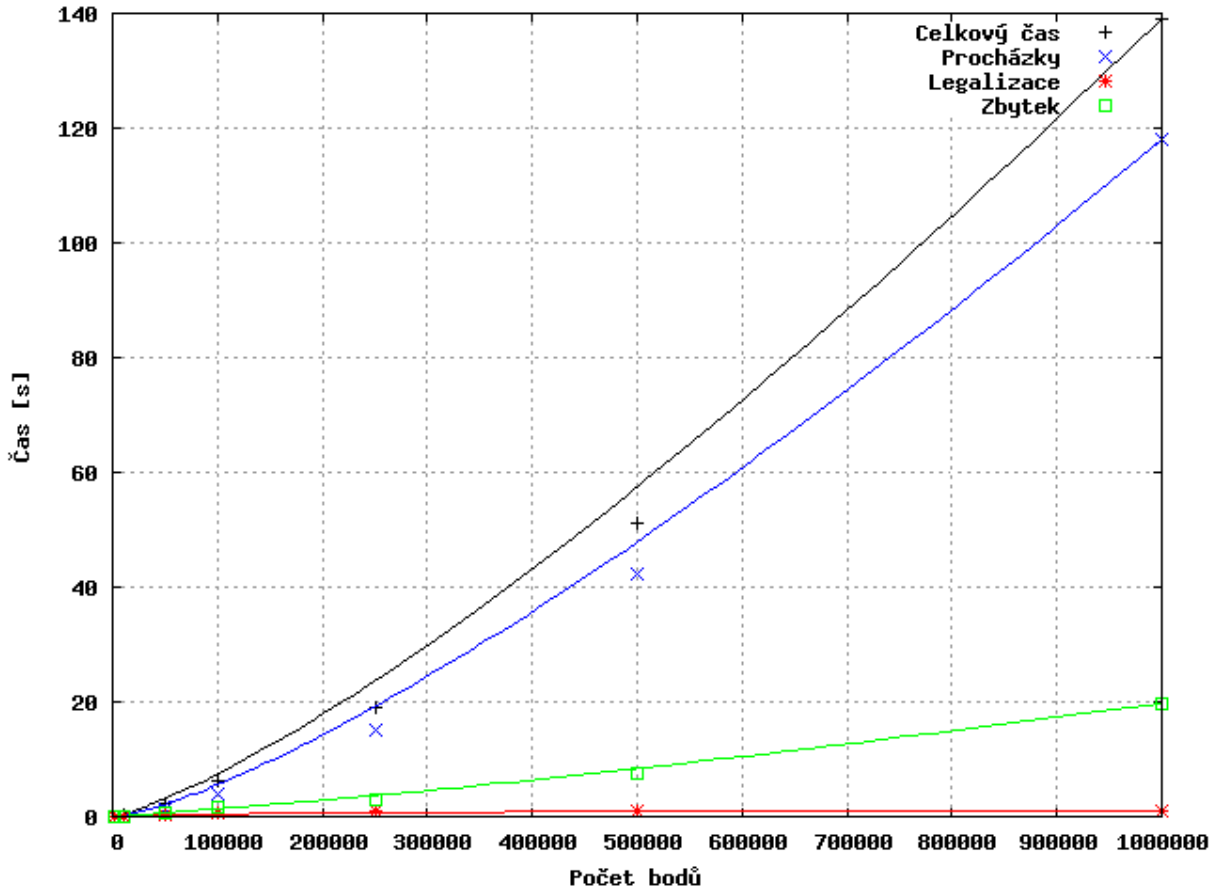
Počet bodů	Čas pro vynucení 100 hran [s]
1000	0,11
10000	0,44
50000	0,99
100000	5,78
250000	12,70
500000	28,55
1000000	37,70

Tabulka 5.3.2: *Časy nutné k vynucení 100 náhodně vygenerovaných hran. (Veškeré hodnoty byly měřeny 3x a následně zprůměrovány. Hodnoty byly naměřeny na osobním počítači s procesorem Intel Core 2 Duo 2x1.66GHz a 1GB RAM.)*

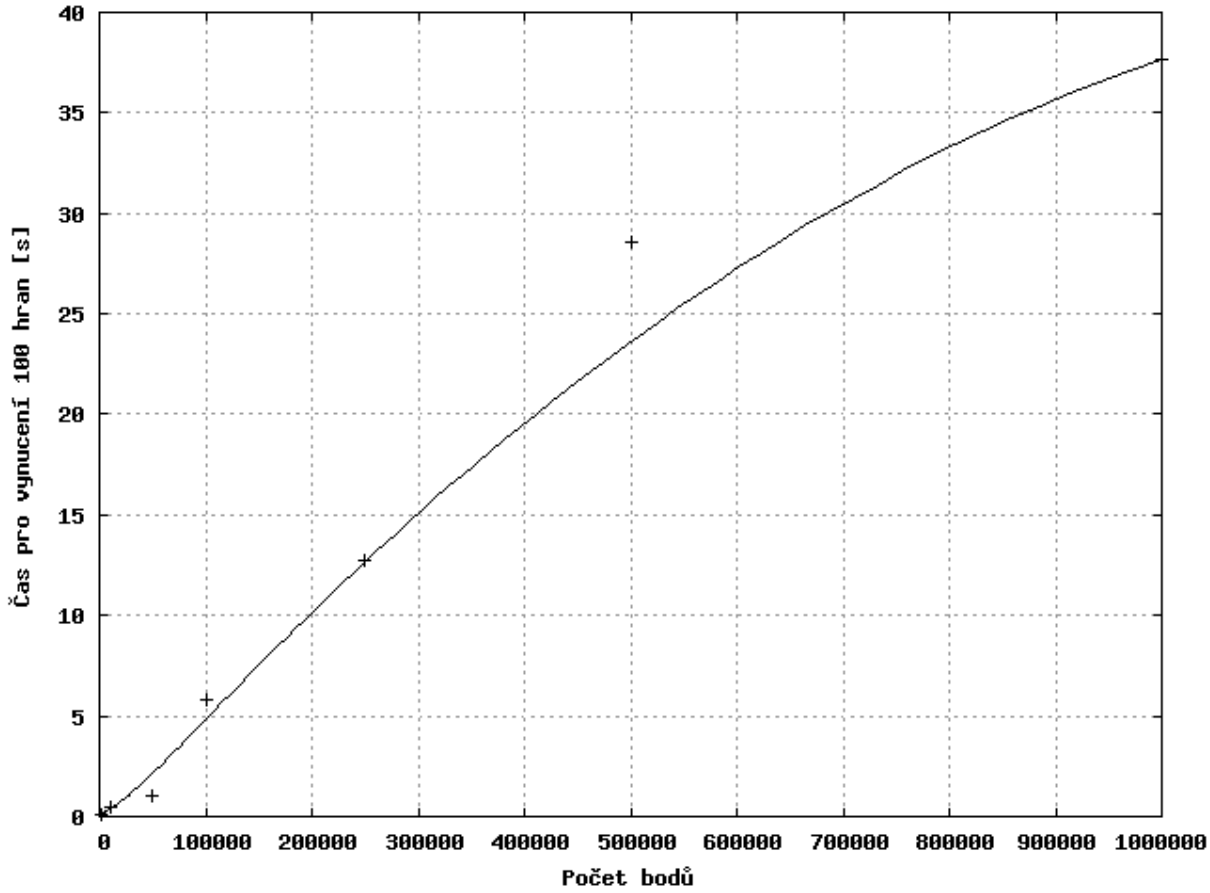
Celkový algoritmus je poměrně složitý, takže není snadné určit jeho složitost analyticky. Z grafu 5.3.2 je však vidět, že časová složitost algoritmu DT je o něco vyšší než lineární, ale mnohem nižší než kvadratická. Asymptotická paměťová složitost je přibližně lineární (začátek křivky je vychýlený, jelikož aplikace alokuje paměť i na jiné účely než pro datové struktury sítě).



Graf 5.3.1: *Využití paměti pro různé počty vkládaných náhodně vygenerovaných bodů. (Celkové množství paměti alokované aplikací. Hodnoty byly naměřeny na osobním počítači s procesorem Intel Core 2 Duo 2x1.66GHz a 1GB RAM.)*



***Graf 5.3.2**: Časy nutné k vykonání jednotlivých částí triangulačního algoritmu pro různé počty vkládaných náhodně vygenerovaných bodů. (Počet vkládaných bodů, celkový čas pro vytvoření triangulace a dílčí časy – čas spotřebovaný algoritmem procházky, čas spotřebovaný na legalizace hran, čas spotřebovaný zbytkem triangulačního algoritmu. Hodnoty byly naměřeny na osobním počítači s procesorem Intel Core 2 Duo 2x1.66GHz a 1GB RAM.)*



Graf 5.3.3: *Časy nutné k vynucení 100 náhodně vygenerovaných hran. (Hodnoty byly naměřeny na osobním počítači s procesorem Intel Core 2 Duo 2x1.66GHz a 1GB RAM.)*

5.4 Zhodnocení výsledků

Konstrukce základní Delaunayovy triangulace pomocí inkrementálního vkládání funguje naprosto spolehlivě. I pro rozsáhlé množiny bodů je čas nutný k jejich triangularizaci přijatelný. Z tabulek v sekci 5.3 je zřejmé, že velkou většinu času nutného k vytvoření triangulace zabírá hledání trojúhelníků pomocí procházky. Zbývající část algoritmu je poměrně rychlejší. Paměťové nároky algoritmu jsou v souladu s předpokladem relativně nízké.

Z důvodu absence hodnot naměřených na aplikacích používajících pravidelné čtvercové sítě nebylo možné provést přesné porovnání výkonu.

Vynucování hran pro CDT funguje velice uspokojivě. Z časových důvodů nebylo možné algoritmus dokonale odladit, takže na velkých sítích není při vynucování složitých útvarů

obsahujících velké množství singularit zcela stabilní. K nestabilitám však dochází pouze výjimečně, takže za normálních okolností je algoritmus použitelný.

Z pozdějších úvah vyplynulo, že i bez hranové reprezentace by mohla být fyzikální vrstva plně funkční. Implementace algoritmu CDT by bez ní byla dosti nesoudržná, přesto je pravděpodobné, že by byla realizovatelná, ačkoliv by se mnoho operací muselo provádět „přes ruku“. Odstranilo by se tím však s velkou pravděpodobností velké množství problémů spojených s udržováním sítě a veškerých pomocných datových struktur v konzistentním stavu. Není však zřejmé, zda by výsledné triangulační jádro byl vhodné pro komunikaci s fyzikální vrstvou.

5.5 Možná vylepšení a návrhy pro další postup

Z předchozího odstavce je zřejmé, že nejslabším článkem triangulačního algoritmu je právě mechanismus vyhledávání trojúhelníků pomocí procházky. Vyhledávání trvá relativně dlouho, zejména v případě, kdy se startovní trojúhelník vybírá náhodně. Procházka je poměrně jednoduchý algoritmus a mohl by být nahrazen něčím efektivnějším, například algoritmem hierarchického vyhledávání. Bylo by patrně vhodné použít dělení prostoru (*Space Subdivision*).

Bylo by též možno uvažovat o změně datové reprezentace, potažmo o úplném zrušení hranové reprezentace. To by však obnášelo nutnost přepsat takřka kompletně implementaci veškerých základních algoritmů. Navíc by mohly nastat komplikace při spolupráci s fyzikální vrstvou. Přesto existuje možnost, že by takovýto krok byl aplikaci ku prospěchu.

5.6 Poděkování

Na tomto místě by autor práce rád poděkoval všem lidem, kteří se nějakým způsobem projektu účastnili. Jmenovitě to jsou: Václav Purchart, Jiří Sedmihradský, Doc. Dr. Ing. Ivana Kolingerová a Bedřich Beneš, Ph.D.

6 Závěr

Cílem projektu bylo vytvořit aplikaci simulující chování písečného terénu a jeho reakce na změny, vyvolané v něm pomocí virtuálního nástroje. Tato aplikace měla posloužit jako podklad pro zvážení, zda jsou nepravidelné trojúhelníkové sítě použitelné v *real-time* aplikacích pro deformaci terénu.

V současné době je aplikace až na drobné detaily funkční a splňuje nároky kladené na rychlost odezvy a vytížení paměti. Aplikaci lze spustit pod operačními systémy *Microsoft Windows* a *Linux*.

Implementace algoritmu pro konstrukci DT je plně funkční. Metoda pro vynucování hran pomocí CDT je uspokojivě funkční, při použití rozlehlých sítí však není zcela stabilní. Z časových důvodů a z důvodů příliš složité reprezentace dat ji nebylo možné dokonale odladit. K nestabilitám však dochází výjimečně a tak je metoda pro potřeby aplikace použitelná.

Použitý typ sítě se osvědčil a nebyly s ním žádné problémy. Volba triangulační metody se také projevila jako vhodná a fyzikální vrstva neměla s vytvářenými trojúhelníky žádné zásadní problémy.

Volba reprezentace dat poměrně značně zkomplikovala implementaci a ladění. Udržení sítě a použitých pomocných datových struktur bylo velmi obtížné. Pro fyzikální vrstvu však tato datová reprezentace naprosto přirozenou cestou poskytla požadované rozhraní, takže v důsledku lze její volbu považovat za adekvátní.

Práce v týmu byla místy poměrně velkou přítěží, jelikož nutnost spolupráce znatelně zpomalovala vývoj. Ve výsledku se však ukázala z mnoha důvodů jako výhodná.

Vývojový tým se shodl na tom, že nepravidelné trojúhelníkové sítě pro *real-time* aplikace pro modelování deformací terénu použitelné jsou a mají svoje opodstatnění. Přesto jsou však výpočetní nároky esenciálních algoritmů značné. U složitějších aplikací se tedy může stát, že kvůli vysoké algoritmické složitosti nebude výkon aplikace dostačující.

Možným vylepšením geometrické části aplikace by bylo použití rychlejšího algoritmu pro vyhledávání trojúhelníků v síti. To by výrazně urychlilo triangularizaci sítě. Důvodem užití stávajícího řešení byla relativní jednoduchost implementace a paměťová nenáročnost. Patrně by bylo též možno optimalizovat datovou reprezentaci.

Výsledky projektu byly v podobě animací také prezentovány vedoucí práce Doc. Dr. Ing. Ivanou Kolingerovou v rámci zvané přednášky o triangulacích na konferenci CESCG 2007 na Slovensku.

Na základě výsledků experimentů a zpětné vazby zahraničních partnerů se předpokládá další zdokonalování algoritmů a implementace v rámci oborových projektů a diplomových prací.

7 Přehled zkratek

- CCW proti směru hodinových ručiček (*counter-clockwise*)
- CDT (1) Delaunayova triangulace s ohraničeními (*constrained Delaunay triangulation*),
- CDT (2) *Eclipse C/C++ Development Tooling*,
- CGT žravá triangulace s ohraničeními (*constrained greedy triangulation*),
- DDT datově závislá triangulace (*data dependent triangulation*),
- DT Delaunayova triangulace (*Delaunay triangulation*),
- GCC *GNU Compiler Collection*,
- GLU *OpenGL Utility Library*,
- GLUT *OpenGL Utility Toolkit*,
- GT žravá triangulace (*greedy triangulation*),
- MinGW *Minimalist GNU for Windows*,
- MWT triangulace s minimální vahou (*minimal weight triangulation*),
- OpenGL *Open Graphics Library*,
- SDL *Simple DirectMedia Layer*,
- SVN *Subversion*.

8 Seznam použité literatury a zdrojů

8.1 Literatura

[PRJ5] *Kadlec, J. – Purchart, V.*, **Modelování eroze a deformací terénu na povrchových modelech**, Dokumentace k Projektu 5, 2007

[Purch1] *Purchart V.*, **Deformace terénu pro virtuální realitu – datová, logická a vizualizační část modelu**, Bakalářská práce KIV/ZČU, 2007

[Sloan1] *Sloan, S. W.*, **A Fast Algorithm for Generating Constrained Delaunay Triangulations**, *Computers and Structures*, Pergammon Press Ltd., Vol 47, Num 3, pp 441–450, 1993

[Kallm1] *Kallmann, M. – Bieri, H. – Thalmann, D.*, **Fully Dynamic Constrained Delaunay Triangulations**, *Geometric Modelling for Scientific Visualization,* Heidelberg, Germany, 2003

[Ono1] *Onoue, K. – Nishita, T.*, **Virtual Sandbox**, *11th Pacific Conference on Computer Graphics and Applications* (PG'03), pp 252–, 2003

[Ben1] *Beneš, B. – Dorjgotov, E. – Arns, L. – Bertoline, G.*, **Granular Material Interactive Manipulation: Touching Sand with Haptic Feedback**, Winter School of Computer Graphics 2006 Plzeň, Česká republika, 2006

[Mue1] *Mücke, E.P. – Saias, I. – Zhu, B.*, **Fast Randomized Point Location Without Preprocessing in Two- and Three-dimensional Delaunay Triangulations**, *Proceedings of the 12th Annual Symposium on Computational Geometry 1996*, pp 274–283, 1996

8.2 Elektronické zdroje

[VAM7] *Kolingerová, I.*, **Rovinné triangulace a jejich aplikace**, přednášky KIV/VAM http://iason.zcu.cz/~kolinger/VAM/VAM7.zip

[Wolf1] *Weisstein, Eric W.*, **Line-Line Intersection**, MathWorld – A Wolfram Web http://mathworld.wolfram.com/Line-LineIntersection.html

[Wiki1] **Delaunay Triangulation** http://en.wikipedia.org/wiki/Delaunay_triangulation

[Wiki2] **Triangulation (Advanced Geometry)**
http://en.wikipedia.org/wiki/Triangulation_%28advanced_geometry%29

[Wiki3] **Digital Elevation Model**
http://en.wikipedia.org/wiki/Digital_elevation_model

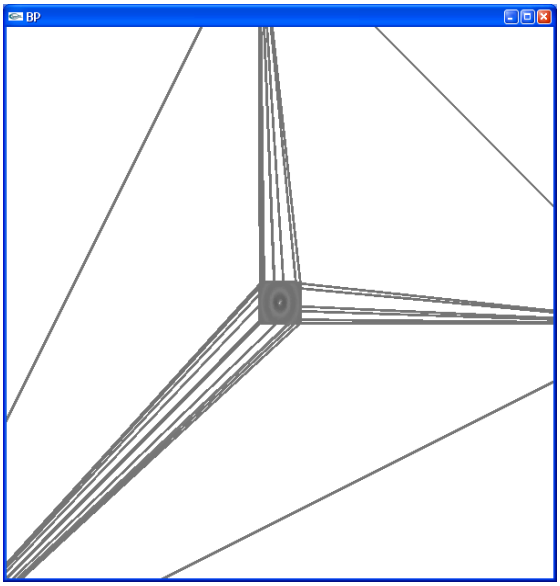
[Wiki4] **Heightmap**
http://en.wikipedia.org/wiki/Heightmap

[Liang1] *Liang, Q.*, **Adaptive Quadtree Grid and Cartesian Cut-Cell Method**
http://www.staff.ncl.ac.uk/qiuhua.liang/Research/grid_generation.html

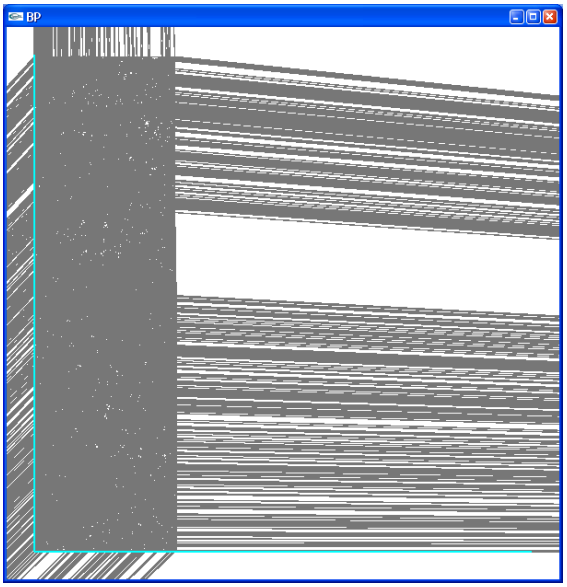
[Contr1] **Design by Contract Programming in C++**
http://www.eventhelix.com/RealtimeMantra/Object_Oriented/design_by_contract.htm

9 Přílohy

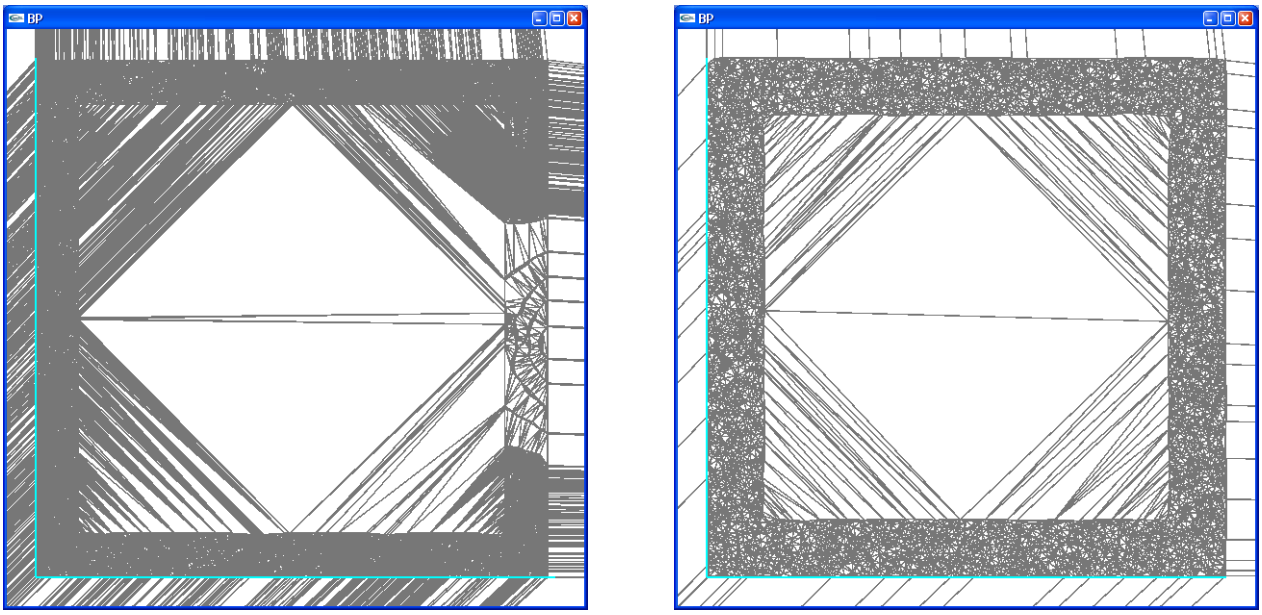
Příloha A – Grafické výstupy aplikace



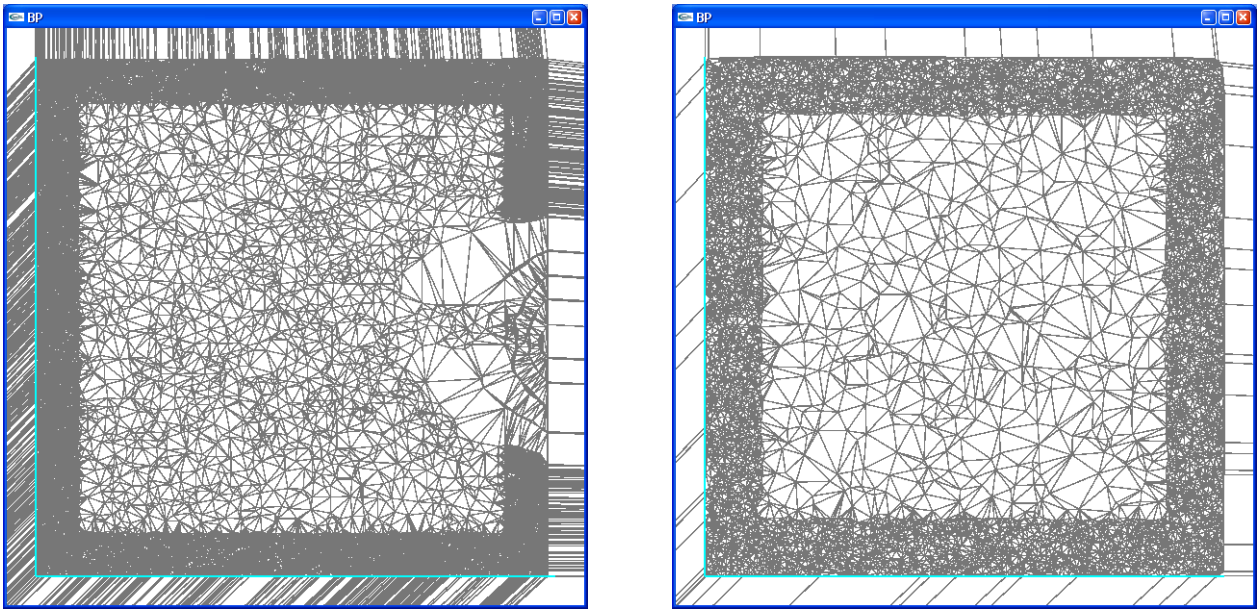
***Obr. A.1**: Všeobjímající trojúhelník. (Vyplněný čtverec uprostřed představuje vlastní triangulovanou množinu.)*



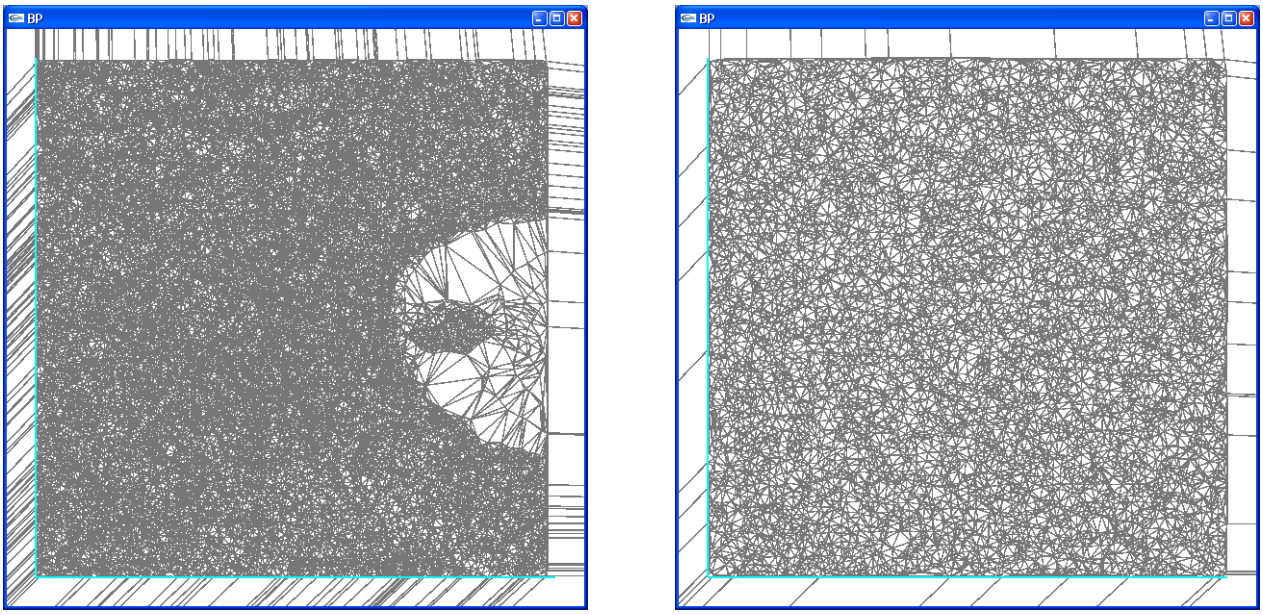
***Obr. A.2**: Průběh vkládání bodů seřazených podle polohy. (Model „Crater Lake“. Levá část je již triangulovaná podmnožina bodů. Pravá část je tvořena prázdným prostorem bez bodů.)*



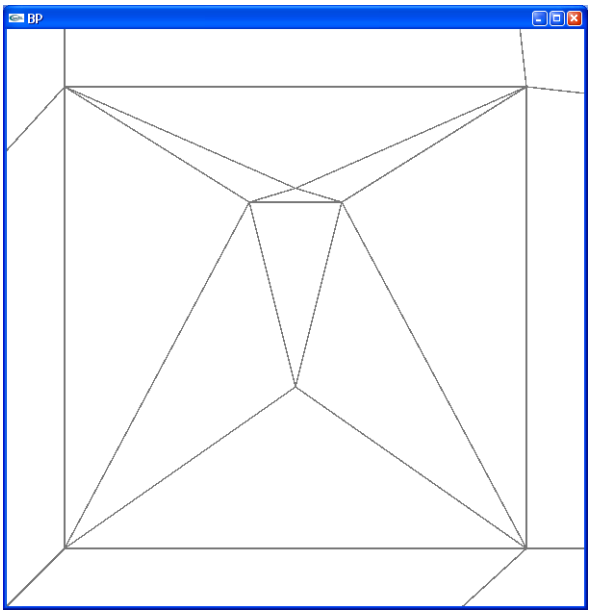
***Obr. A.3 a A.4**: Průběh vkládání bodů seřazených podle klesající vzdálenosti od středu (Vlevo model „Crater Lake“, vpravo náhodná množina bodů.)*

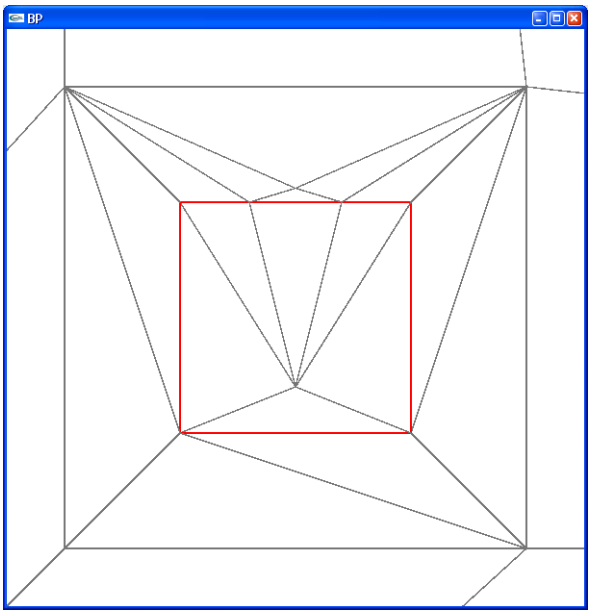


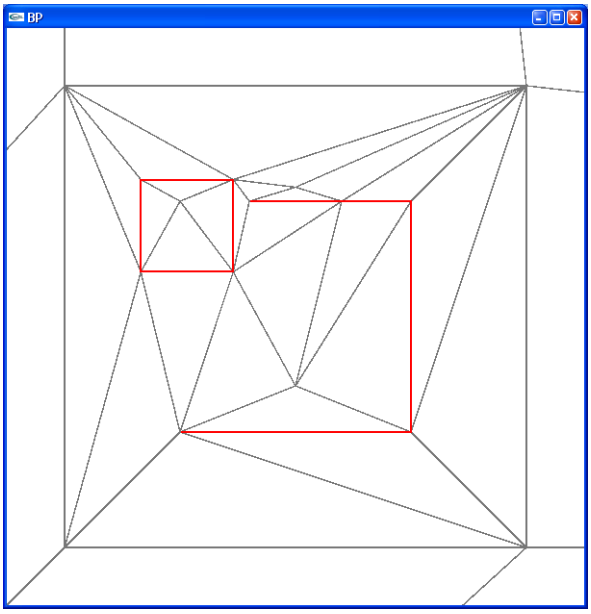
***Obr. A.5 a A.6**: Průběh vkládání bodů seřazených podle klesající vzdálenosti od středu a následně lehce promíchaných. (Vlevo model „Crater Lake“, vpravo náhodná množina bodů.)*



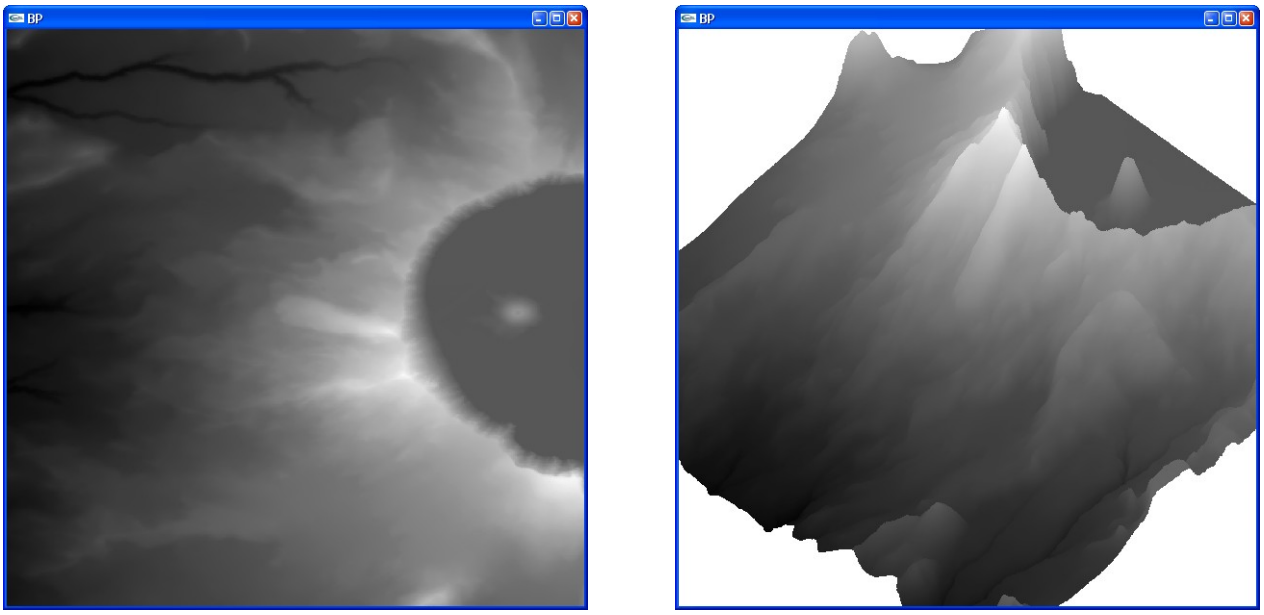
***Obr. A.7 a A.8**: Průběh vkládání bodů v náhodném pořadí. (Vlevo model „Crater Lake“, vpravo náhodná množina bodů.)*







Obr. A.9, A.10 a A.11*: Demonstrace vynucení čtyř hran, tvořících čtverec, do sítě. (Vlevo nahoře: Síť před vynucením hran. Vpravo nahoře: Síť po vynucení hran. Z obrázku je zřejmé, že horní vynucená hrana částečně splývá s hranou, která je v síti již obsažena. Horní vynucená hrana je tedy rozdělena na tři fragmenty. Vpravo dole: Vynucení dalšího čtverce přes roh původního. Je vidět, že svislá hrana původního čtverce byla zrušena celá, kdežto z vodorovné hrany byla odstraněna pouze levá třetina. To dokazuje, že horní hrana byla skutečně rozdělena na fragmenty.*



***Obr. A.12 a A.13**: Model „Crater Lake“ s naznačením výškových poměrů. (Vlevo: Pohled shora. Vpravo: 3D zobrazení zešikma.)*

***Obr. A.14**: Otexturovaný a nasvícený model „Crater Lake“ shora ve 3D perspektivním zobrazení.*

***Obr. A.15**: Otexturovaný a nasvícený model písečného terénu ve 3D zobrazení.*