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

Bakalářská práce

Plzeň, 2007

Václav Purchart

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

Bakalářská práce

Deformace terénu pro virtuální realitu – datová, logická a vizualizační část modelu

Plzeň, 2007 Václav Purchart

Poděkování

Chtěl bych poděkovat všem, kteří se na projektu účastnili, za jejich nápady a pracovní nasazení. Můj dík patří těmto osobám: Jan Kadlec, Jiří Sedmihradský, Doc. Dr. Ing. Ivana Kolingerová a Ing. Bedřich Beneš, Ph.D.

Zadání

Deformace terénu pro virtuální realitu - datová, logická a vizualizační část modelu.

1. Prostudujte existující metody modelování terénu.
2. Ve spolupráci s kolegou řešícím geometrickou část projektu navrhněte metodu vhodnou pro využití geometrického modelu při interakcích ve stylu virtuální reality.
3. Navrhněte a realizujte vhodné rozhraní, které dovolí práci využít také pro modelování přírodních vlivů na terénu.
4. Implementujte a vyzkoušejte funkcionalitu svého řešení.
5. Zhodnoťte dosažené výsledky.

Prohlášení

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

V Plzni dne ..... Václav Purchart .....

Abstract

Terrain deformations for virtual reality – basic structures, control layer and model visualization.

The goal of this work is to find out whether an irregular triangular mesh is suitable for real-time terrain deformation applications. Document contains a recommendation how to create a generic interface for different physical processes that cause deformations to the terrain. It is also described how to force sudden changes made by a virtual tool to be present in the terrain model. The work also contains information about design of basic mesh structures.

Obsah

1. Úvod

Cílem této práce je zjistit, zda jsou nepravidelné trojúhelníkové sítě vhodné v aplikacích pro *real-time* deformaci terénu. Bylo tedy nutné vytvořit vhodný povrchový model, který umožní deformace způsobené různými jevy simulovat. Tento úkol byl časově i implementačně dosti náročný, proto jsme celý projekt realizovali v týmu. Po důkladném nastudování problematiky jsme celý projekt rozdělili na tři části. Každý člen měl na starosti jednu z těchto částí.

Úkolem kolegy J. Kadlece bylo najít a implementovat vhodný geometrický model terénu, který do něj umožní ve spolupráci se mnou provádět různé zásahy. To vše dostatečně rychle, aby mohla celá aplikace běžet v reálném čase.

Úkolem kolegy J. Sedmihradského bylo celý model efektivně – a efektně – vizualizovat, a také navrhnout vhodný částicový systém, který bude na geometrickém modelu simulovat erozi materiálu.

Mým úkolem bylo navrhnout základní struktury tvořící geometrický model. Ve spolupráci s kolegou J. Kadlecem navrhnout a realizovat metodu vhodnou pro zásahy do terénu ve stylu virtuální reality. Dále pak navrhnout rozhraní vhodné pro modelování přírodních vlivů na terénu a společně s J. Sedmihradským pak toto rozhraní využít pro modelování eroze materiálu.

Stručný přehled následujících kapitol:

- *Celkový přehled řešení* – obsahuje teorii a poznámky k jednotlivým částem projektu.
- *Celkový přehled mé části* – obsahuje obecně popsané problémy, řešené v rámci této práce.
- *Podrobnosti řešení* – popisuje řešení dříve nastíněných problémů na algoritmické úrovni.
- *Implementace* – postřehy z práce v týmu, vývoj aplikace.
- *Výsledky a zhodnocení* – obsahuje naměřená data, výsledky pozorování, zjištěné problémy a možná řešení těchto problémů.
- *Závěr* – obsahuje zhodnocení dosažených výsledků a nastiňuje budoucnost projektu.

2. Celkový přehled řešení

V této kapitole si obecně popíšeme různé metody pro modelování terénu. Dále se budeme zabývat jednotlivými částmi, které jsou v projektu řešeny.

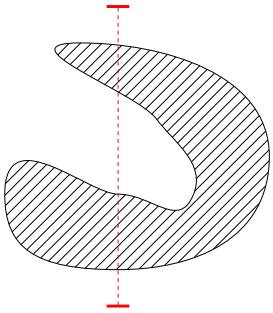
2.1 Modelování terénu

Nejdůležitější součástí celé práce je model terénu. Na něm závisí všechny použité algoritmy, ale i časové a paměťové nároky. Zvolený model také určuje některá omezení.

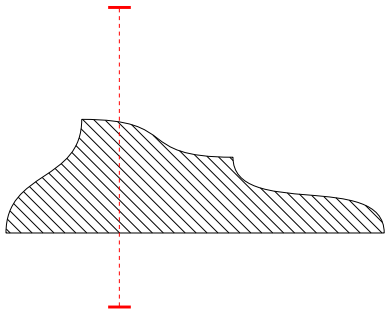
[Převzato z PK06] Pro počítačové modelování terénů ve 3D se používají nejčastěji povrchové modely. To jsou takové modely, které popisují pouze tvar povrchu terénu a nezabývají se tím, co se nachází pod ním.

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. 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. obrázek 2.1.1 a 2.1.2). Takový povrch totiž není funkcí souřadnic ležících ve vodorovné rovině. Přicházíme tak o možnost podchytit skalní převisy, šikmé jámy, 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.



Obrázek 2.1.1: Terén s přesahem (pohled z boku)



Obrázek 2.1.2: Výšková mapa (pohled z boku)

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 Druhy sítí

Nejjednodušší způsob, jak uchovat informace o tvaru terénu, je navzorkování jeho výšky např. do matice v pravidelných intervalech. Toto řešení je implementačně jednoduché a dostatečně rychlé. Rozlišovací schopnost toho modelu je ovšem omezena jemností vzorkování. Čím větší detaily chceme zobrazit, tím větší musí být vzorkovací matice. Tím *velmi* vzrostou nároky na paměť (u čtvercové sítě je celkový počet bodů v síti kvadraticky závislý na počtu bodů na jednotku vzdálenosti).

Tento problém řeší nepravidelné sítě – vzorkování neprobíhá v pravidelných intervalech, ale jeho jemnost měníme tak, jak je potřeba. Tím se sníží paměťové nároky, protože málo členité oblasti popíšeme jen několika buňkami. Pro manipulaci s terénem je nutné použít důmyslnější a náročnější algoritmy.

2.2.1 Pravidelné sítě

Pravidelné sítě mohou být například čtvercové, trojúhelníkové, hexagonální i jiné. Jejich největší výhodou je jejich jednoduchost. Deformací a erozí na těchto sítích se zabývají články [BDA06] a [Ono03]. Dá se říci, že se jedná o „probádanou“ oblast.

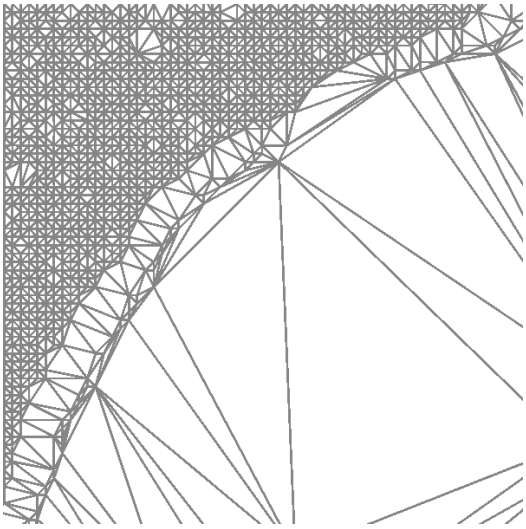
Další výhodou je, že nevznikají žádné singularity nebo špatně podmíněné trojúhelníky. To je výhodné pro simulaci různých fyzikálních dějů. Zjištění výšky v libovolném bodě sítě má časovou složitost *O(1)*, což velice urychlí všechny operace. Pro vizualizaci jsou pravidelné sítě také výhodnější, protože se dají snadněji pokrýt texturou.

V článku [Ono03] je popsán způsob, jak v těchto sítích docílit vytlačování materiálu po tažení nástrojem v písku. Stačí jen správně rozdistribuovat materiál do sousedních buněk výškové matice podle směru působení síly od nástroje.

2.2.2 Nepravidelné sítě

Nepravidelné sítě mohou být např. trojúhelníkové nebo čtvercové. Jemnost těchto sítí je možné měnit podle požadované úrovně detailů v různých místech. Procházení a úprava této struktury je ale komplikovanější. Výhodou toho přístupu jsou nižší nároky na paměť. Ukázka nepravidelné trojúhelníkové sítě (viz obrázky 2.2.1, 2.2.2). Všimněme si, jak je síť hustá v místech, kde je terén hodně členitý, a jak je naopak řídká na rovných plochách.

Obrázek 2.2.1: Vepředu velmi členitá trojúhelníková síť; v zadní části (na rovině) je členitost menší (pohled z boku)



Obrázek 2.2.2: Detailnější pohled na zadní část obrázku 2.2.1 (pohled shora)

Pro vytvoření trojúhelníkové sítě z množiny libovolně uspořádaných bodů použijeme triangulaci. Konkrétní, námi použitou triangulační metodou je Delaunayova triangulace, konstruovaná inkrementálním vkládáním, detailně popsaná ve [VAM] a [Kad07].

2.3 Vynucování hran v trojúhelníkové síti

Nezbytnou součástí úpravy terénu je vynucování změn v modelu. V nepravidelné trojúhelníkové síti je situace o dost složitější než v pravidelné výškové mapě. Nestačí jen

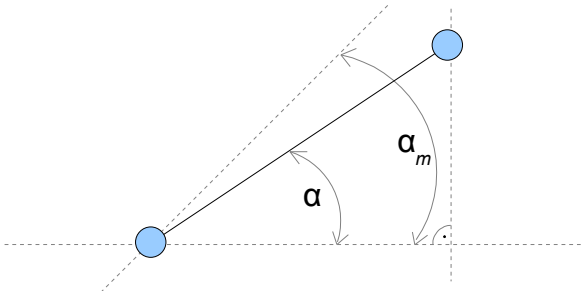
v daných místech nastavit výšku. Často se totiž na kýženém místě ani nenachází vrchol, který bychom mohli upravit. Při triangulaci se navíc mohou hrany trojúhelníků libovolně proházet, aby co nejlépe vyhověly triangulačnímu kritériu. Je tedy nutné dát síti nějak vědět, že určité hrany musí zůstat na svých místech (popřípadě je na požadované pozici musíme vytvořit), i když tím porušíme triangulační kritérium. Vynucováním hran můžeme vytvářet libovolné útvary, které lze do sítě napevno otisknout. Touto problematikou se zabývá Delaunayova triangulace s omezením (*Constrained Delaunay Triangulation*, dále jen CDT). Algoritmus pro výpočet CDT je popsán např. v publikaci [Slo93]. Vložení hrany do sítě je poměrně komplikované – může nastat velké množství singulárních případů, které je nutné ošetřit. Vynucováním hran v trojúhelníkové síti se ve své bakalářské práci zabývá kolega Jan Kadlec [Kad07]. V tomto textu nebudeme řešit problémy při vkládání *jedné* hrany nebo implementační detaily, ale spíše se zaměříme na „pohled shora“, tj. jak správně vložit skupinu hran do sítě, abychom docílili námi požadovaného efektu na terénu.

2.4 Modelování eroze

V přírodě existuje mnoho různých druhů eroze – například vodní eroze, eroze způsobená roztahováním nebo smršťováním materiálu při teplotních změnách nebo třeba větrná eroze. K modelování těchto jednotlivých jevů je zapotřebí robustní matematický aparát a nejedná se o jednoduchou záležitost. Výsledkem všech těchto jevů dohromady je, že se materiál z vyvýšených míst přesouvá do míst nižších. Přesun je tím razantnější, čím je výškový rozdíl větší. Právě na tento typ eroze se zaměříme v naší aplikaci.

Celý model se skládá z trojúhelníků různých tvarů a velikostí. Tyto trojúhelníky jsou tvořeny vrcholy a každý vrchol má uloženou svou výšku. Erozi lze tedy namodelovat tak, že spočítáme převýšení (úhel hrany) mezi každými dvěma sousedními vrcholy. Pokud bude jeden z vrcholů výš, přesuneme část jeho „výšky“ do druhého vrcholu. Tím se sníží výškový rozdíl mezi těmito vrcholy tak, jak se to děje i v přírodě. V ideálním případě by se postupem času celá krajina vyrovnala do jedné roviny. Skutečný materiál se tak ale nechová. Přesýpání hmoty je tedy nutné v určitém okamžiku zastavit. Pokud bude spočítaný úhel menší než nějaký námi zvolený, změnu výšky neuskutečníme. Představme si písek na pláži, do kterého uděláme díru. Pokud je písek suchý, díra se za chvilku sama zbortí a písek z okraje a stěn zasype vnitřek díry. Pokud písek namočíme a vyhrabeme novou díru, zasype se jen část díry a zbytek zůstane. Mezní úhel pro přesýpání tedy závisí na vlastnostech modelovaného materiálu.

Prohledávání celé sítě a počítání úhlů mezi všemi body by bylo časově neúnosné. Proto je nutné dát nějakým způsobem algoritmu modelujícímu erozi vědět, které body jsou pro erozi zajímavé. Na počátku předpokládejme rovnou plochu. Do té postupně přidáváme vrcholy tím, jak terén upravujeme a potřebujeme v něm vynucovat změny. Řešení se nám tedy samo nabízí. Algoritmus necháme zpracovat jen ty body, které nějakým způsobem změní svoji výšku. Změněné body můžeme řadit do fronty, kde budou čekat na další zpracování. Když vrchol vůči všem svým sousedům klesne pod mezní úhel, můžeme ho z fronty vyloučit (viz obrázek 2.4.1). Při opětovném změně bodu – který už ve frontě je – je nutné ho v tomto seznamu aktualizovat.



Obrázek 2.4.1: Porovnání úhlu (α) mezi dvěma vrcholy s mezním úhlem pro daný materiál (α_m)

Fyzikální vrstva pracuje na principu částicového systému. Hmota se po hranách trojúhelníků „přelévá“ z jednoho vrcholu do druhého. Pro tento účel by bylo nejlepší, kdyby se síť skládala jen z rovnostranných trojúhelníků. To ovšem v nepravidelné síti není možné splnit. Proto se alespoň budeme snažit, aby se trojúhelníky v síti co nejvíce rovnostranným blížily. Kritérium Delaunayovy triangulace nám v síti bude vytvářet právě takové trojúhelníky.

Erozí se detailně zabývá kolega Jiří Sedmihradský ve své diplomové práci [Sed07].

2.5 Shrnutí

Stručně jsme si tedy určili, co k řešení problému použijeme a jakým směrem se přibližně budeme ubírat. V následující kapitole se podrobněji zaměříme na body v zadání této práce.

3. Celkový přehled mé části

V této kapitole se konkrétněji zaměříme na body v zadání této práce.

3.1 Základní struktury

Práce se skládá ze čtyř základních částí: trojúhelníková síť, virtuální nástroj, simulace fyzikálních procesů a zobrazování.

Tyto části na sobě musí být co nejméně závislé, aby je bylo možné snadno měnit a nahrazovat jinými implementacemi. Proto je důležité stanovit jednotné rozhraní mezi těmito částmi.

Veškeré simulace se odehrávají na nepravidelné trojúhelníkové síti (tato síť se skládá z bodů, které po výpočtu triangulace tvoří jednotlivé trojúhelníky). Pro snadný průchod sítí je nutné, aby si každý trojúhelník pamatoval své sousední trojúhelníky (přes hrany). To je nezbytné například pro algoritmus procházky [Kad07], ale i pro každé jiné „šíření“ sítí. Bez této informace by byly trojúhelníky nepoužitelné. Dále zavedeme pravidlo pro snadnější a rychlejší manipulaci s daty uloženými v trojúhelníku – veškerá data budou orientována proti směru hodinových ručiček. Tak snadno určíme následující vrchol nebo hranu, se kterou budeme pracovat. V síti budeme provádět vynucování hran (viz kapitola 2.3). Každá hrana si tedy musí uchovat, zda je v síti vynucená, nebo zda se jedná o obyčejnou hranu. Zavedeme tedy příznak „vynucenosti“ hrany. Vynucené hrany nebude možné při výpočtu triangulace prohazovat, ani kdyby tvořily špatně podmíněné trojúhelníky. Hrana (z bodu A do bodu B) je sdílena vždy mezi dvěma sousedními trojúhelníky. Každý z nich má hranu orientovanou jiným směrem. Trojúhelníky tedy budou u každé hrany uchovávat ještě příznak inverze této hrany (tj. zda je hrana pro tento trojúhelník opačně orientovaná, než je její skutečná orientace). Část simulující fyziku bude počítat úhel z rozdílu výšek sousedních bodů. Bude tedy dobré do každého bodu uložit, se kterými body sousedí, tj. ze kterého vrcholu vede do kterého hrana.

Touto úvahou jsme tedy zjistili, jaké základní struktury budeme pro funkční reprezentaci sítě potřebovat a jaké budou mít mezi sebou vzájemné vztahy.

3.1.1 Sousednost vrcholů

Nezbytnou součástí sítě je informace o sousednosti vrcholů (tzn. ze kterého vrcholu vede do kterého hrana). Při erozi se materiál přesýpá z jednoho vrcholu do druhého,

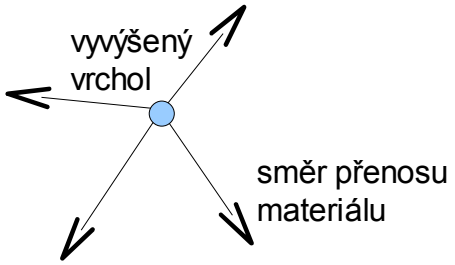
což se provádí pro všechny změněné vrcholy v každém časovém okamžiku. Sousedící body bychom mohli zjistit i z jiných uložených informací. Protože sousednosti potřebujeme opravdu často, je časově výhodnější je uložit.

Sousednost bodů je využívána i v algoritmu *CDT*.

3.2 Rozhraní pro fyzikální vrstvu

Toto rozhraní by mělo být co nejobecnější, aby bylo možné nad modelem simulovat libovolné fyzikální (a jiné) děje, aniž bychom tvůrce této vrstvy zbytečně obtěžovali vnitřními strukturami trojúhelníkové sítě. Než navrhneme obecné rozhraní, probereme si detailně požadavky konkrétního děje, implementovaného v rámci naší práce.

V našem případě simuluje fyzikální vrstva erozi. Celý algoritmus lze stručně popsat tak, že z bodů, které mají výrazně vyšší výšku než jejich sousední body, rozdistribuujeme část materiálu do okolí (viz obrázek 3.2.1). Záleží na vlastnostech materiálu, kdy tento proces zastavíme. Pokud je například materiál velice sypký, můžeme terén téměř vyhladit. Naproti tomu pevnější materiál si zachová většinu svých původních rysů.



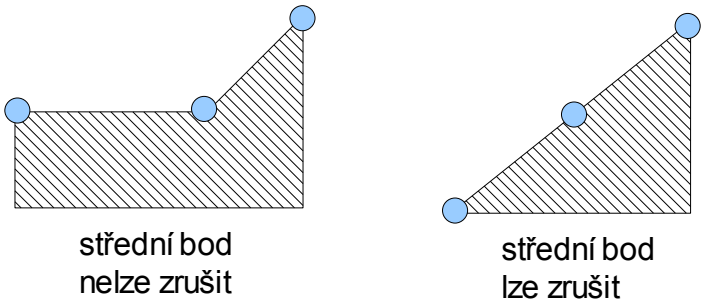
Obrázek 3.2.1: Směr šíření materiálu z vyvýšeného vrcholu

Tato vrstva ale nemůže v každém okamžiku prohledávat všechny body sítě, aby zjistila, které vrcholy mají erodovat, protože síť může být dosti rozsáhlá. Musíme tedy zavést zpětnou vazbu od sítě: pokaždé, když nějakému vrcholu změníme výšku nebo přidáme nový vrchol, vygenerujeme fyzikální vrstvě událost, že byl nějaký vrchol změněn (a pošleme informaci, který). Fyzikální vrstva si změněné body ukládá do fronty a v čase vyhrazeném pro erozi je pak zpracuje. Tento jednoduchý model nepotřebuje se sítí nijak pracovat. Zasaženým bodům stačí změnit výšku na novou hodnotu.

Pokud bychom chtěli simulovat složitější děje (například vodní erozi způsobenou deštěm nebo větrnou erozi na určitém místě terénu), potřebujeme nejen nastavovat výšku existujícím vrcholům v síti, ale i je přesouvat a vytvářet vrcholy nové – je nutné mít možnost

nastavit konkrétní výšku krajiny v libovolném místě. Je tedy nutná další vazba z fyzikální vrstvy do sítě.

Dále může tato vrstva při přesýpání hmoty vyhodnocovat, které vrcholy jsou ještě v síti potřeba a které už můžeme zrušit. Pokud se například tři body vyrovnají (přibližně) do přímky, je možné prostřední bod zrušit, a tak síť zjednodušit (obrázek 3.2.2). Je tedy nutné zavést možnost odebrání některých bodů ze sítě.



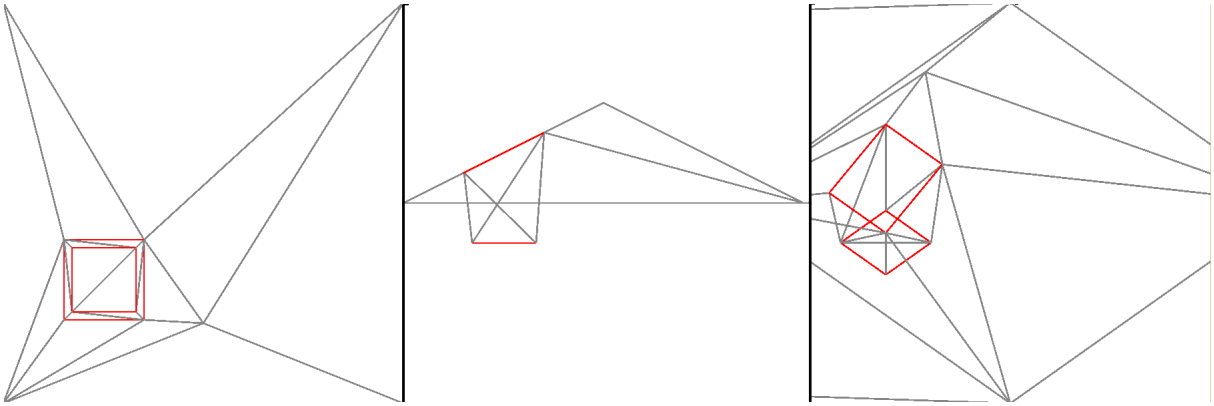
Obrázek 3.2.2: Kdy může fyzikální vrstva vydat pokyn k odstranění bodu. Vlevo – nelze odstranit, vpravo – lze odstranit.

3.3 Virtuální nástroj

Virtuální nástroj (razítko) je nástroj, se kterým uživatel ovlivňuje terén a vynucuje v něm požadované změny. Tyto změny provádíme vynucováním vhodných hran v síti. Pro vynucování hran je použit algoritmus *CDT*.

Program obsahuje dva typy razítek. Obecné razítko, které slouží jen k vynucovaní hran v síti, a hloubící razítko pro vytváření různých důlků a děr.

Hloubící razítko (resp. jeho otisk) se dělí na dvě důležité části – vnější a vnitřní část. Tyto části jsou od sebe nepatrně vzdáleny, aby se nepřekrývaly (aby platilo, že lze povrch terénu vyjádřit jako funkci). Ukázka hloubícího razítka je vidět na obrázku (3.3.1).

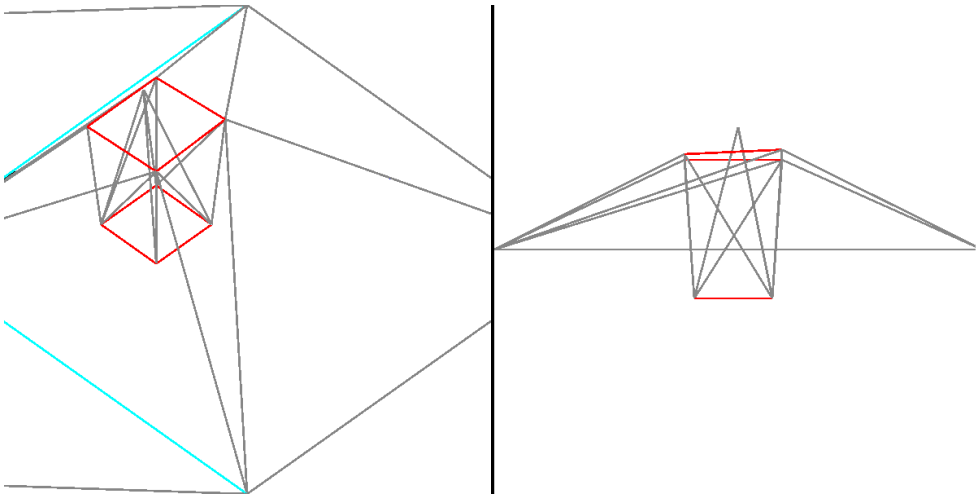


Obrázek 3.3.1: Hloubící razítko (vlevo – půdorys, uprostřed – bokorys, vpravo – pohled ze strany). Všimněme si rozestupů mezi vnějším a vnitřním okrajem razítka. Ty jsou důležité pro zachování konvexnosti terénu (viz dále).

3.3.1 Vnitřní část hloubícího virtuálního nástroje

Vnitřní část hloubícího razítka mění podle nastavení výšku terénu. V síti pomocí CDT vynutíme hrany. Vzniklým bodům pak nastavíme výšku podle požadované hloubky.

Pokud je terén příliš členitý nebo je razítko moc velké, mohou ve vnitřní části zůstat různé „kopečky“. Ty je nutné eliminovat, protože jinak výsledný efekt neodpovídá hrabání. Vzniklou chybu vidíme na obrázku (3.3.2).



Obrázek 3.3.2: Ve vnitřní části razítka se nachází kopeček. Všechny části s výškou větší než dno razítka je nutné eliminovat. Vlevo – pohled ze strany, vpravo – pohled z boku.

Eliminaci je možné provést tak, že postupně projdeme body uvnitř plochy ohraničené vynucenými hranami a nastavíme jim správnou výšku – ovšem jen v případě, že je hloubka těchto bodů menší než hloubka vynucovaná razítkem (kopeček).

Pro zmenšení složitosti sítě lze odstranit všechny body z vnitřní oblasti razítka, pokud splňují následující vlastnosti:

1. hloubka bodu je (přibližně) stejná jako hloubka razítka,
2. odstraňovaný bod není bodem, tvořícím vnitřní hraniční oblast,
3. bod nesousedí s bodem, jehož hloubka je větší než hloubka razítka.

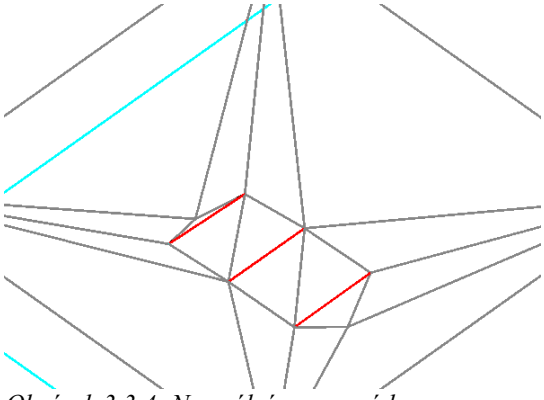
3.3.2 Vnější část hloubícího virtuálního nástroje

Vnější část hloubícího razítka tvoří hranici mezi terénem a hloubenou dírou. V ideálním případě tedy musí přesně kopírovat terén (všechny prohlubně i kopečky).

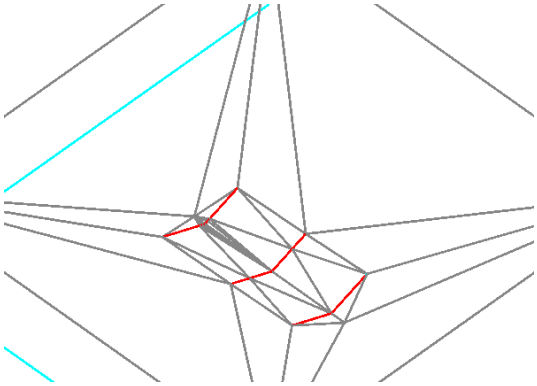
Přibližného výsledku dosáhneme tím, že pro krajní body jednotlivých hran razítka zjistíme přesnou výšku v daném místě před přidáním těchto hran do sítě. Pokud tento postup aplikujeme na všechny krajní body vkládaných vynucených hran, dostaneme – ve většině případů – velice dobrý výsledek (obrázek 3.3.1).

Co když ale vynucenou hranu v dostatečně velké vzdálenosti od jejích krajních bodů protíná jiná hrana, která se svou výškou v průsečíku s vynucenou hranou výrazně liší od výšky krajních bodů vynucené hrany? *CDT* dá přednost vynucené hraně a původní hranu zruší. Tím se ztratí i skutečná výška terénu v tomto bodě a zbude jen interpolovaná výška krajních bodů vynucené hrany (obrázek 3.3.3). Tento případ nastane, pokud je v krajině podlouhlá rokle, nebo horský hřeben tvořený jedinou hranou.

V místech, kde se vynucená hrana protíná s jinými hranami, tedy potřebujeme zachovat výšku terénu. To lze udělat například tak, že před přidáním vynucené hrany do sítě zjistíme interpolací výšku v místě budoucích průsečíků s hranami, které už v síti jsou. Razítko potom nepřidává jednu vynucenou hranu, ale sérii hran. Každá z těchto vkládaných hran je ohraničená buď jedním z krajních bodů původní hrany, anebo některým ze zjištěných průsečíků (obrázek 3.3.4).



Obrázek 3.3.4: Normální vynucená hrana. Nepřizpůsobuje se struktuře sítě, ale napevno se vynutí pomocí CDT.



Obrázek 3.3.3:Dělená hrana s předem zjištěnými průsečíky s původními hranami v síti. U každého průsečíku je interpolací zjištěna výška.

Pokud vynucujeme vždy jen jednu hranu, je výsledek výborný. Pokud ale chceme vynutit skupinu hran, které spolu nějak souvisí (například razítko), vynutí se správně jen první hrana a zbylé hrany se už nemusí správně přizpůsobit. Po vložení první hrany totiž některé trojúhelníky nemusí splňovat triangulační kritérium a některé jejich hrany mohou být prohozeny. Tím opět ztratíme informaci o výšce v daném místě. Výše uvedený postup je tedy potřeba nějak zdokonalit.

Ztrátě informace zabráníme například tak, že do sítě nic nepřidáme. Což je sice v rozporu s naší snahou, ale nijak to nevadí. Před přidáváním jednotlivých hran si připravíme frontu, do které pak budeme dávat všechny zjištěné průsečíky (s interpolovanou výškou) a počáteční nebo koncové body jednotlivých částí razítka. Po nalezení všech průsečíků pro jednu hranu ji nepřidáme do sítě, ale jednoduše začneme počítat průsečíky další hrany. Takto budeme pokračovat, dokud nezjistíme všechny průsečíky celého vynucovaného objektu. Poté už postupně přidáváme kratší vynucené hrany tvořené jednotlivými body ve frontě. Teprve teď se razítko za všech okolností správně přizpůsobí místu v terénu, kam ho vkládáme.

Jednodušší možností, jak vnější okraj razítka přibližně přizpůsobit terénu, je rozdělit jednotlivé vkládané hrany na více částí. Každá tato část pak interpolací zjistí výšku sítě na svém začátku a konci a při dostatečně jemném dělení se razítko také přizpůsobí. Tímto způsobem však zbytečně narůstá složitost sítě, protože vkládáme body i tam, kde to není potřeba.

4. Podrobnosti řešení

V tato kapitola obsahuje přesný popis použitých algoritmů a řešení.

4.1 Základní struktury

V kapitole (3.1) jsme si definovali základní struktury, které jsou pro funkci geometrického modelu nezbytné. V následujícím textu si popíšeme detaily jejich implementace, tj. jednotlivé třídy a jejich metody, které poskytují.

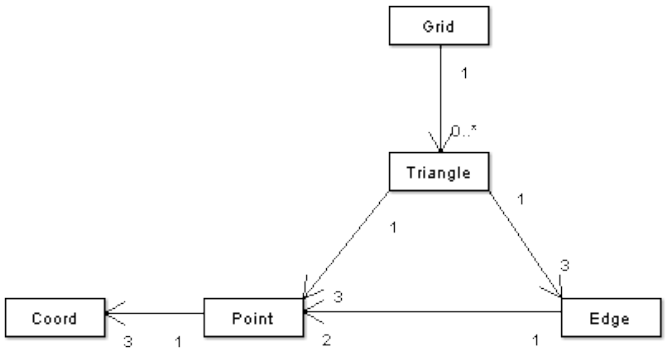
Seznam tříd:

[struktura – odpovídající třída v kódu]

- trojúhelníková síť – *Grid*
- trojúhelník – *Triangle*
- hrana – *Edge*
- bod – *Point*
- souřadnice – *Coord*

4.1.1 Vzájemné vztahy mezi strukturami

Následuje popis vzájemných vztahů. Pro názorné zobrazení závislostí jsem použil UML diagram (obrázek 4.1.1).



Obrázek 4.1.1: Vzájemné vztahy mezi základními strukturami trojúhelníkové sítě.

4.1.2 Důležité vlastnosti jednotlivých struktur

Dále uvádím popis důležitých vlastností v syntaxi jazyka C++:

(K nastavovacím funkcím typu *get* samozřejmě existují i nastavovací funkce typu *set*, ty zde pro jejich nadbytečnost neuvádím.)

Point :

- `Point *getNeighbour(int index)` – vrátí n-tého souseda aktuálního vrcholu,
- `Coord& operator[] (int i)` – přístup k souřadnicím (index modulo 3),
- tato třída dále obsahuje operace pro vektorový součet, rozdíl, součin, inverzi vektoru a také skalární součin.

Edge :

- `Point* getVertex(int index)` – jeden ze dvou vrcholů hrany (index modulo 2),
- `void setConstrained(bool b, int order)` – nastavení „vynucenosti“ hrany a pořadové číslo „generace“ vynucené hrany – toto číslo pak pomáhá určovat stáří vynucených hran při jejich mazání.

Triangle :

- `Point* getPoint(int index)` – jeden ze tří vrcholů trojúhelníka (index modulo 3),
- `Edge* getEdge(int index)` – jedna ze tří hran (modulo 3),
- `bool isInvertedEdge(int index)` – zjištění příznaku inverze hrany (modulo 3),
- `Triangle* getNeighbour(int index)` – vrátí souseda přes n-tou hranu (index modulo tři).

Grid :

- `void addConstrainedEdge(Point *a, Point *b)` – vložení vynucené hrany (CDT),
- `bool load(char *path)` – načtení kompletní sítě ze souboru (body, trojúhelníky, sousedé),
- `bool loadPointsOnly(char *path)` – načtení vrcholů ze souboru, následná triangularizace,

- `void generatePointsOnly(int count)` – vygenerování *n* náhodných bodů, triangulace,
- `Triangle *getTriangleAtPoint(Point *p)` – algoritmus procházky,
- `Point* addPointAndTriangulizeGrid(Point *p)` – inkrementální DT, přidané body jsou posílány fyzikální vrstvě pro kontrolu,
- `Coord getElevationAtPoint(Coord x, Coord y)` – pro fyziku, interpolace výšky,
- `Point *setElevationAtPoint(Point *p)` – pro fyziku, nastavení výšky (kdekoli), přidané body se už neposílají fyzikální vrstvě, aby nedocházelo k vzájemnému přeposílání bodů,
- `Point *addConstrainedEdge_Divide(Point *a, Point *b)` – vložení přizpůsobivé vynucené hrany (hrana neprohazuje stávající hrany v síti, ale dělí je průsečíky s interpolovanou výškou),
- `void startConstrainedEdge_Divide()` – zapnout shromažďování průsečíků pro razítko,
- `void endConstrainedEdge_Divide()` – přidat všechny shromážděné průsečíky do sítě.

Modulo aritmetika byla zavedena až v průběhu vytváření programu jako reakce na vzrůstající složitost kódu. Díky tomu se *velmi* zredukovalo množství rozhodovacích bloků v kódu a došlo tím k celkovému zpřehlednění programu.

4.1.3 Sousednost bodů

Pro tento problém je velice užitečné zavedení *hran*. Hrany totiž reprezentují právě tato spojení mezi vrcholy. Pokud tedy vytvoříme hranu mezi nějakými body *A* a *B*, upravíme sousednosti těchto bodů. Pokud hranu zrušíme, zrušíme i sousednost mezi body *A*, *B*. Naskýtá se tedy velice jednoduché řešení. V konstruktoru objektu *Edge* navzájem provážeme sousednostmi body, tvořící novou hranu. V destruktoru objektu pak tuto sousednost zase zrušíme. To je vše! V celé síti teď máme automaticky spravované sousednosti všech bodů.

4.2 Implementace fyzikální vrstvy

Přejděme ke konkrétnímu řešení části pro modelování fyzikálních procesů. Fyzikální modely spravuje třída *Physics*, která zároveň definuje rozhraní pro tyto modely. Použitý fyzikální model se musí automaticky (nejlépe v konstruktoru třídy) zaregistrovat metodou *registerInstance()* u správce. Tím se prováže se sítí a při změně vrcholů pak automaticky dostává informace o těchto vrcholech. Aktivní může být vždy jen jeden fyzikální model, což ovšem nebrání použití více modelů. Stačí vytvořit jeden „hlavní" model, který bude podřízeným modelům generovat podle potřeby události. Tento přístup je pružnější, než kdybychom registrovali všechny modely napevno. Události totiž můžeme posílat podle potřeby a ne *vždy*. Pokud není registrován žádný fyzikální model, události jsou zahozeny.

Událost generovaná sítí zavolá při změně vrcholu metodu *pointChanged()*. Tuto metodu může každý model přetížit dle libosti a se získanou informací naložit podle svých potřeb.

Změnu výšky vrcholů v síti je možné v jednodušších případech provádět přímo nastavením výšky bodu v bufferu. Pro složitější procesy pak můžeme využít funkci sítě *setElevationAtPoint()*. Tímto způsobem můžeme nastavit libovolnou výšku v libovolném místě v síti. Protože je triangulace realizována inkrementálním vkládáním jednotlivých bodů, je implementace této funkce velice jednoduchá. Prostě jen pokračujeme v triangulaci. Procházka najde trojúhelník, ve kterém nový bod leží, popřípadě určí singularity. Tento trojúhelník rozdělíme na více menších trojúhelníků, které nový bod obsahují, a hrany přeházíme tak, aby splňovaly triangulační kritérium. Pokud už se v blízkosti nového bodu nachází jiný bod (při *epsilon testu* vyjdou oba body totožné), vrátí tato funkce původní bod, který se už v síti nachází, a nový bod nebude přidán. Původnímu bodu v síti ale nastavíme výšku na novou hodnotu.

Další věcí, která je nutná pro simulaci, je zjištění výšky v libovolném bodě sítě; pokud například na terén dopadne dešťová kapka, je nutné v okolí stávající výšku o něco zmenšit (udělat důlek), ať se v daném místě nachází vrchol nebo ne. Pro tento účel slouží funkce *getElevationAtPoint()*. I pro různé jiné děje je tato funkce nezbytná. Výšku v libovolném místě zjistíme lineární interpolací podle vzorce (4.3.6).

Původně jsme zamýšleli udělat program jako dvouvláknový. V jednom vlákně by běžela hlavní smyčka, která by sbírala události od uživatele, prováděla vykreslování a triangulaci sítě a ve druhém vlákně by běžely fyzikální procesy. Nakonec jsme zvolili jednodušší jednovláknové řešení. Pro toto jednovláknové řešení slouží funkce *tick()*. Tato funkce se volá

před každým překreslením scény. Fyzikální vrstva v něm může provádět postupnou erozi bodů nashromážděných v bufferu. Postupnou erozi modelu zachycují obrázky v příloze (10.7 a 10.8).

4.3 Virtuální nástroj

V kapitole (3.3) jsme probrali základní problémy spojené s virtuálním nástrojem (razítkem). Mimo jiné také rozdělení virtuálních nástrojů na obecné, které jen vkládají skupinu hran do sítě, a na hloubící virtuální nástroje.

Obecné razítko definuje základní funkce každého virtuálního nástroje, jako je pohyb, vykreslování a reakce na události od uživatele. Toto razítko je reprezentováno třídou *VirtualTool* a je základem pro všechny ostatní virtuální nástroje.

Základní funkce rozhraní pro virtuální nástroje:

- `virtual void draw(Graphics *g)` – vykreslení nástroje,
- `virtual void setPosition(Coord x, Coord y)` – nastavení pozice v terénu,
- `virtual void setEvalationChange(Coord z)` – změna výšky (hloubky) razítka,
- `virtual void action()` – vynucení hran / vyhloubení díry na aktuální pozici.

Dále se budeme zabývat jen hloubícími nástroji. Hloubící razítko se dělí na dvě části – vnější a vnitřní část.

4.3.1 Vnitřní část hloubícího virtuálního nástroje

Úkolem vnitřní části nástroje je srazit výšku všech vrcholů nacházejících se uvnitř této části na úroveň hloubky razítka (jak je uvedeno v kapitole 3.3.1). To provedeme pomocí rekurzivní „laviny“, kterou projdeme všechny trojúhelníky vnitřní část razítka. Než budeme moci šíření uskutečnit, potřebujeme najít startovní trojúhelník. Spočítáme tedy střed razítka (vzorec 4.3.1) a použijeme trojúhelník, kterému tento bod náleží (jak je popsáno dále).

Střed vnitřní části razítka spočítáme takto:

oblast, ohraničující vnitřní část, je tvořena body $P_{1,} P_{2,} \ldots P_{n}$,

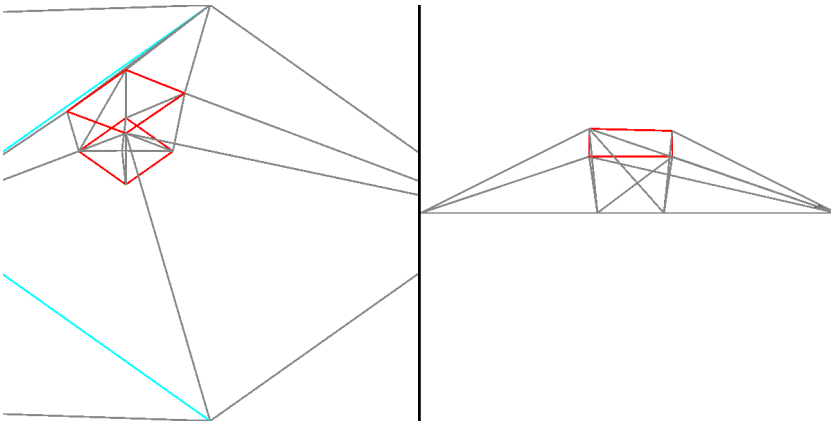
zjistíme minimální a maximální hodnotu z těchto bodů pro obě souřadnice (x, y),

$$P_{min} = [\, min(P_{ix}), min(P_{iy})\,], \quad i = 1..n,$$
$$P_{max} = [\, max(P_{ix}), max(P_{iy})\,], \quad i = 1..n,$$

střední bod pak spočítáme,

$$P_{center} = \frac{P_{min} + P_{max}}{2}$$ (vzorec 4.3.1, střed razítka).

Pomocí algoritmu procházky najdeme trojúhelník ležící v tomto místě. Všechny jeho body nastavíme na požadovanou hloubku, pokud splňují to, že jejich hloubka je menší než je hloubka vnitřní části razítka. Nazvěme toto kritérium „kritériem menší hloubky“. Tuto akci provedeme u všech sousedících trojúhelníků, které nesousedí přes vynucenou hranu – to nám zaručí, že při tomto lavinovitém šíření neopustíme vnitřní část razítka. Tímto postupem bychom ale brzy přeplnili rekurzivním voláním zásobník. Musíme nějak trojúhelníky, které už kritérium menší hloubky splňují, označit. Každému navštívenému trojúhelníku tedy přiřadíme unikátní číslo, specifické právě pro aktuální lavinovité šíření ve vnitřní části *(explodeGenerationIndex)*. Proměnnou určující toto šíření budeme při každém dalším šíření (dalším otisku razítka) zvyšovat o jedna (možná radši číslovkou). Při navštívení nějakého trojúhelníku teď stačí ověřit, že poslední na něm probíhající lavinovité šíření neodpovídá aktuálnímu šíření. Pokud se tedy čísla v trojúhelníku a počítadle šíření nerovnají, nastavíme hloubku podle kritéria menší hloubky. Pokud se čísla rovnají (trojúhelník byl navštíven), neprovedeme žádnou operaci. Tímto postupem tedy vždy dosáhneme dokonalého efektu pro vnitřní část razítka. Šíření je ilustrováno v obrázku 4.3.3. Výsledek po dokončení šíření viz obrázek 4.3.2.



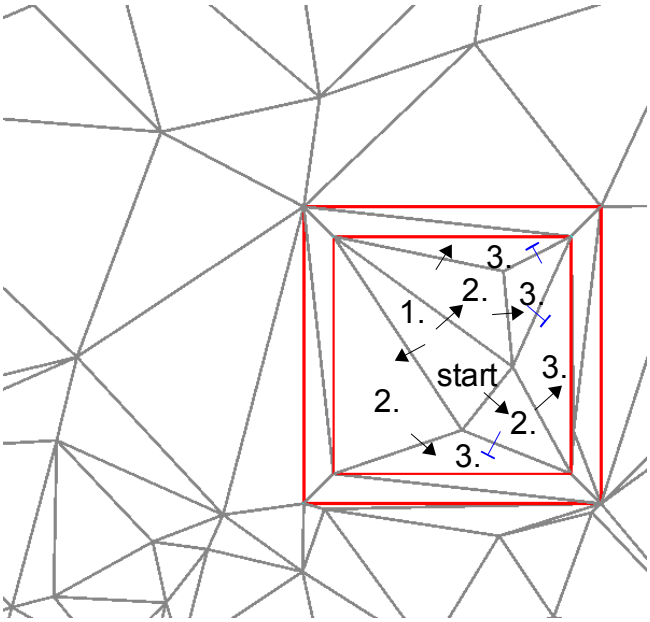
Obrázek 4.3.2: Po vyrovnání všech kopečků ve vnitřní části razítka už otisk vypadá tak, jak bychom předpokládali. Vlevo – pohled ze strany, vpravo – pohled z boku.

Popis algoritmu pro vytvoření vnitřní části razítka:

1. Podle vzorce (4.3.1) určíme bod, ve kterém je střed razítka.
2. Algoritmem procházky určíme trojúhelník, ve kterém spočtený bod leží – tento

trojúhelník je startovní. Nastavíme startovní trojúhelník jako aktuální.

3. Zvýšíme globální počítadlo vnitřních šíření o jedna.
4. Pokud má aktuální hodnotu posledního šíření shodnou s globálním počítadlem šíření, pak ukončíme tuto větev prohledávání sítě.
5. Pokud se hodnoty počítadel nerovnají, potom pro všechny body trojúhelníku provedeme následující kroky:
   a) porovnáme výšku bodu s hloubkou razítka. Pokud je hloubka bodu menší než hloubka razítka, nastavíme vrcholu hloubku na hloubku razítka,
   b) pokud je hloubka menší než hloubka razítka, neděláme s bodem nic.
6. Aktuálnímu trojúhelníku nastavíme interní počítadlo šíření na hodnotu globálního počítadla
7. Pro všechny trojúhelníky, se kterými aktuální trojúhelník sousedí:
   a) pokud trojúhelník sousedí přes vynucenou hranu, neprovádíme nic,
   b) v opačném případě nastavíme sousední trojúhelník jako aktuální a pokračujeme v bodě 4.



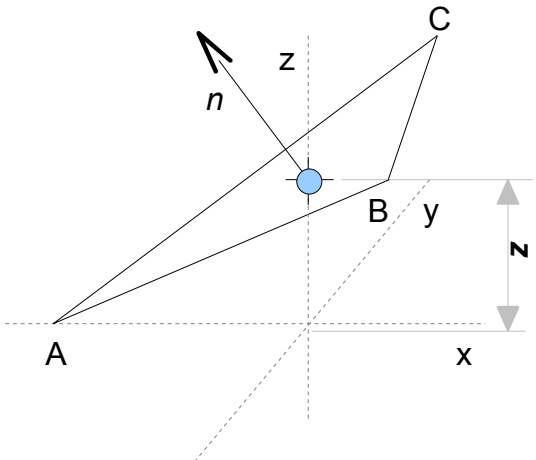
Obrázek 4.3.3: Postupné rekurzivní šíření „laviny“, která vyhlazuje vnitřní plochu razítka. Černé šipky reprezentují další rozvětvení při procházení. Čáry s plochým zakončením rekurzi zastavují.

4.3.2 Vnější část hloubícího virtuálního nástroje

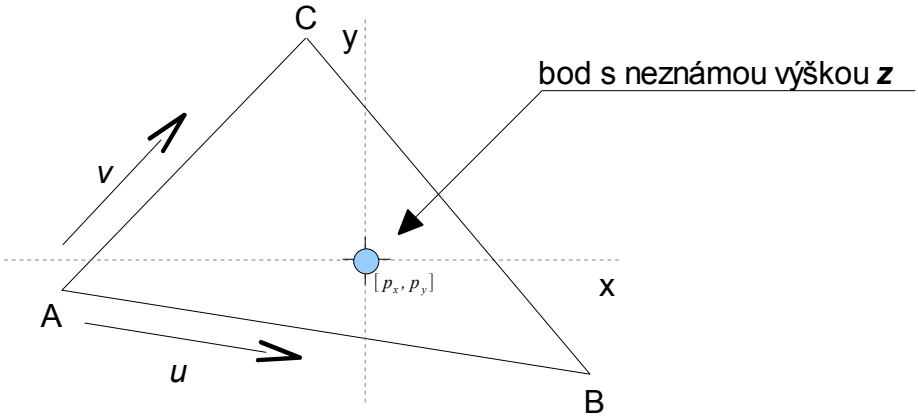
Jak je uvedeno v kapitole (3.3.2), vnější část nástroje se pokud možno snaží přizpůsobit původnímu tvaru terénu před vytvořením díry. Pro tento účel je důležité, abychom uměli zjistit výšku terénu v libovolném bodě – i tam, kde síť neobsahuje žádný vrchol.

Výšku v neznámém bodě $[p_x, p_y, ?]$ snadno určíme lineární interpolací výšky

okolních bodů. Algoritmem procházky určíme trojúhelník, který bude nový bod obsahovat. Jeho tři body tvoří v prostoru rovinu (obrázky 4.3.4 a 4.3.5). Potřebujeme tedy zjistit hodnotu výšky bodu, ležícím v této rovině na souřadnicích $[p_x, p_y]$. K tomuto účelu lze využít například barycentrické souřadnice. V programu jsem použil jednoduchý vzorec, odvozený pomocí analytické geometrie (vzorec 4.3.6).



Obrázek 4.3.4: Pohled na interpolační rovinu tvořenou body trojúhelníka – ze strany



Obrázek 4.3.5: Pohled na interpolační rovinu tvořenou body trojúhelníka – shora

Určíme tedy směrové vektory (u, v) roviny definované body trojúhelníka (A, B, C):

$u = B - A$,

$v = C - A$.

Z nich určíme vektorovým součinem normálový vektor roviny (n):

$n = u \times v$.

Připomeňme si tvar obecné rovnice roviny v prostoru:

$ax + by + cz + d = 0$,

kde $n = (a, b, c)$,

a libovolný bod $F=(x, y, z)$,

koeficient (*d*) získáme snadno skalárním součinem normálového vektoru s některým z bodů trojúhelníka:

$$n \cdot A + d = 0,$$
$$d = -n \cdot A.$$

Dosazením souřadnic (*x*, *y*) interpolovaného bodu do rovnice teď můžeme snadno získat jeho výšku:

$$a \cdot p_x + b \cdot p_y + c \cdot z + d = 0,$$
$$c \cdot z = -(a \cdot p_x + b \cdot p_y + d),$$

$$z = -\frac{(a \cdot p_x + b \cdot p_y + d)}{c}$$

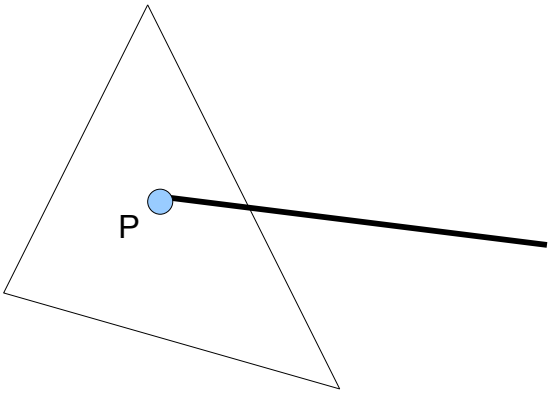
(vzorec 4.3.6, kde p_x a p_y jsou souřadnice neznámého bodu, koeficienty a, b, c jsou složky normálového vektoru určeného vrcholy trojúhelníka, a koeficient d je dán skalárním součinem normálového vektoru s libovolným bodem trojúhelníka).

Takto tedy zjistíme přesnou výšku v libovolném bodě sítě. Koeficient *c* nemůže být v našem modelu nikdy nulový, protože používáme povrchový model, kde nelze vytvořit kolmou stěnu.

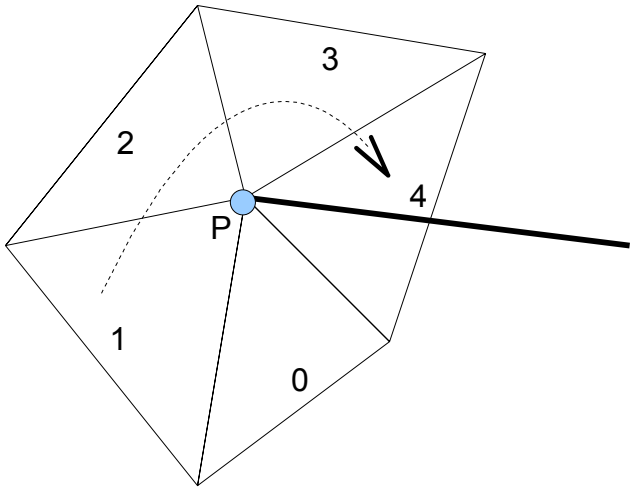
Jak zjistíme průsečíky se všemi hranami, které se protínají s ještě neexistující vynucenou hranou? Nejdříve musíme najít startovní trojúhelník, tj. trojúhelník obsahující první bod vkládané vynucené hrany. Navíc se také jedna z hran startovního trojúhelníka musí protínat s vynucenou hranou. Opět nám pomůže algoritmus procházky, který nám vrátí trojúhelník obsahující první bod vynucené hrany.

Mohou nastat dva případy:

- bod leží někde v trojúhelníku (obrázek 4.3.7) – v tom případě je trojúhelník startovní,
- bod je totožný s bodem vráceného trojúhelníku (obrázek 4.3.6) – tento trojúhelník nemusí být startovní. Ověříme, jestli se hrana ležící proti bodu protíná s vynucenou hranou. Pokud ano, máme startovní trojúhelník. V opačném případě se vydáme po sousedech toho trojúhelníku proti směru hodinových ručiček kolem vkládaného (středového) bodu. U každého trojúhelníku testujeme, jestli se hrana, ležící proti středovému bodu, protíná s vkládanou vynucenou hranou. Dříve nebo později takový trojúhelník najdeme, protože všechny testované trojúhelníky mají společný první bod vynucené hrany.



Obrázek 4.3.7: Bod vynucené hrany leží uvnitř startovního trojúhelníku



Obrázek 4.3.6: Bod vynucené hrany je vrcholem několika trojúhelníků

Máme tedy startovní trojúhelník a víme, která jeho hrana se protíná s budoucí vynucenou hranou. Spočítáme průsečík obou hran, interpolací zjistíme výšku v daném místě a průsečík si uložíme do seznamu průsečíků. Přes protínající se hranu se vydáme k sousednímu trojúhelníku. Pokud tento trojúhelník obsahuje koncový bod vynucené hrany, můžeme skončit a přidat jednotlivé vynucené hrany do sítě. Pokud nejsme na konci této „procházky“, otestujeme zbylé dvě hrany na průsečík s vynucenou hranou. Jedna vždy průsečík má. Opět průsečík spočítáme a přes tuto hranu se vydáme k dalšímu sousedovi.

Uvedeme si vzorec, pomocí kterého snadno a rychle spočítáme průsečík dvou přímek. První přímka je určena body $[x_1, y_1], [x_2, y_2]$, druhá přímka je určena body $[x_3, y_3], [x_4, y_4]$. Průsečík obou přímek na souřadnicích $[x, y]$ pak spočítáme podle vzorce (4.3.8):

$$x = \frac{\begin{vmatrix} \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} & y_1 - y_2 \\ \begin{vmatrix} x_3 & y_3 \\ x_4 & y_4 \end{vmatrix} & y_3 - y_4 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & y_1 - y_2 \\ x_3 - x_4 & y_3 - y_4 \end{vmatrix}} \qquad y = \frac{\begin{vmatrix} \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} & x_1 - x_2 \\ \begin{vmatrix} x_3 & y_3 \\ x_4 & y_4 \end{vmatrix} & x_3 - x_4 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & y_1 - y_2 \\ x_3 - x_4 & y_3 - y_4 \end{vmatrix}}$$

Vzorec 4.3.8: Vzorec pro výpočet průsečíku dvou přímek zadaných dvěma body. Tento vzorec byl převzat z [WRE99].

Pokud je jmenovatel ve vzorci 4.3.8 nulový, jsou přímky rovnoběžné nebo totožné a nemají průsečík.

Máme tedy zjištěné všechny průsečíky budoucí vynucené hrany s ostatními hranami, které už v síti jsou. Postupně teď stačí vynutit kratší hrany mezi jednotlivými průsečíky. Původní hrana se teď dokonale přizpůsobí výšce sítě ve všech zlomech.

Popis algoritmu pro dokonalé otisknutí vnějšího okraje razítka:

1. Pro každou hranu, kterou chce nástroj vynutit, provedeme následující operace:
   a) pokud jsme už spočítali průsečík se všemi hranami, přejdeme na bod 2, jinak:
   b) nalezme startovní trojúhelník pomocí algoritmu procházky,
      i. pokud v procházce nenastal singulární případ (hledaný bod není totožný s žádným bodem v síti), prohlásíme nalezený trojúhelník za startovní,
      ii. v případě, že nastala singularita postupně obcházíme singulární bod proti směru hodinových ručiček (obrázek 4.3.6) přičemž u každého trojúhelníku testujeme hranu protilehlou proti tomuto bodu na průsečík s vynucenou hranou. Použité znaménkové testy implementoval kolega Kadlec a jsou popsány v jeho bakalářské práci [Kad07]. Dříve nebo později musíme takový trojúhelník najít. Tento trojúhelník prohlásíme za startovní,
   c) startovní trojúhelník prohlásíme za aktuální a uložíme index hrany, která se protíná s vynucenou hranou,
   d) obdobně jako v bodě *b)* nalezneme trojúhelník, ve kterém hrana končí,
   e) do fronty průsečíků uložíme první bod původní vynucené hrany,
   f) pomocí vzorce 4.3.8 určíme průsečík s hranou, jejíž index jsme si uložili. Průsečík uložíme do fronty průsečíků,
   g) přes tuto hranu přejdeme na sousední trojúhelník a otestujeme, jestli se nacházíme v koncovém trojúhelníku. Pokud ano, přejdeme na bod *i)*,
   h) trojúhelník, ve kterém se nacházíme, nastavíme jako aktuální. Spočítáme, která ze zbývajících dvou hran se protíná s vynucenou hranou. Její index si uložíme. Trojúhelník, ve kterém se nacházíme, prohlásíme za aktuální. Přejdeme na bod *f)*,

i) do fronty průsečíků uložíme koncový bod vynucené hrany a přejdeme na bod *a)*.

2. Postupně procházíme frontu průsečíků a pomocí CDT vynutíme jednotlivé hrany. Tyto hrany jsou vždy mezi dvěma body náležících k *jedné* původní hraně.

Algoritmus pro přibližné otisknutí vnějšího okraje razítka:

(Při vytvoření razítka rozdělíme každou jeho hranu na *n* částí)

1. Pro každou hranu:
   a) pro každou část hrany:
      i. interpolací (podle vzorce 4.3.6) zjistíme výšku v počátečním a koncovém bodě vynucené hrany *A, B*,
      ii. bodům *A, B* nastavíme výšku na výsledek interpolace,
      iii. vynutíme hranu mezi body *A, B*.

4.3.3 Pořadí vynucování částí razítka

Důležité je, v jakém pořadí v síti vynutíme vnitřní a vnější část. Pro správné otisknutí razítka je nutné nejdříve vynutit vnější část a až poté část vnitřní. Vnější část se totiž správně přizpůsobí tvaru terénu a teprve potom je „násilně“ vložena vnitřní část. V opačném případě by se totiž některé hrany mohly při výpočtu triangulace po vložení vnitřní části zrušit. Vnější okraj razítka by se pak sice přizpůsobil tvaru terénu, ale ten už by neodpovídal situaci před otiskem razítka.

4.3.4 Tažení virtuálním nástrojem

Docílení efektu tažení virtuálním nástrojem je realizováno poměrně jednoduše. Pokud uživatel drží tlačítko myši (chce táhnout nástrojem), po určité době provádíme pravidelně otisknutí razítka. Naše implementace CDT dá přednost novým vynuceným hranám před starými. Staré hrany jsou zrušeny. Pokud je vzorkovací frekvence pro otisk razítka příliš vysoká, dochází k prudkému nárůstu složitosti sítě. Pokud je frekvence nízká, může se stát, že na sebe jednotlivé otisky nebudou správně navazovat. Pro dosažení co nejlepšího efektu tažení se v programu razítko otiskuje v každém překreslení obrazovky. Průběh tažení je ilustrován obrázky (10.1 až 10.6 a 10.9 až 10.17).

4.3.5 Razítko jako fyzikální děj

Razítko využívá stejných metod jako fyzikální vrstva. Jen s tím rozdílem, že nás nezajímají body v síti, které se změnily, ale přímo vynucujeme změnu bodů na pozici určené uživatelem. Chybí tedy zpětná vazba, kterou mají ostatní fyzikální děje. Pro razítko není nutné přidávat další nové vlastnosti do modelu.

5. Implementace

Programovací jazyk, knihovny, spolupráce.

Pro maximální přenositelnost a budoucí využití programu jsme zvolili programovací jazyk *C++*.

Použití *C++*, podle mého názoru, trochu brzdilo vývoj programu, protože programování v něm není tak „komfortní“ a rychlé jako v některém z novějších jazyků (např. *C#*, *D*). Pro *real-time* aplikace je ale důležitá rychlost programu. Tady má *C++* jednoznačně navrch.

Při dostatku času by asi bylo nejlepší řešení odladit a vyřešit celý problém v některém z *RAD* jazyků a hotovou věc pak přepsat do *C++*.

Pro vizualizaci dat v OpenGL jsme zpočátku používali knihovny GLU a GLUT. Později jsme přešli ke knihovně SDL.

Protože někteří členové našeho týmu neměli pravidelný přístup k internetu, byla práce v týmu obtížná. Čas od času bylo nutné synchronizovat implementaci základních tříd, podle toho, jak se vyvíjelo řešení problému. Při tom vznikala řada nových problémů. Celou věc by snadno vyřešil například SVN repozitář, přes který by šlo snadno distribuovat aktuální kód.

Program jsem vyvíjel v prostředí *Eclipse* s pluginem *CDT*, za použití kompilátoru *gcc* na platformách Windows XP (32bit) a Fedora Core 6 (64bit). V systému Windows jsem používal balík *MinGW*.

Gcc nebyl jediný používaný kompilátor. J. Sedmihradský používal MS Visual Studio. Často jsme naráželi na nekompatibilitu mezi těmito kompilátory.

S úspěchem jsme použili programovací techniku Design by Contract Programming popsanou v [DCP06]. Díky vestavěným validacím v kódu lze téměř zamezit samovolnému „padání“ programu. Také snadno nalezneme nekonzistentní objekty, a tak se chyby v programu nešíří dál.

6. Výsledky a zhodnocení

V této části shrneme dosažené výsledky a pozorování. Také zde uvádím možná řešení různých objevených problémů, které je nutné pro opravdové nasazení programu vyřešit.

6.1 Pozorování

Model se chová dobře a trojúhelníková síť je tedy použitelná pro *real-time* aplikace na úpravu terénu. Všechny zjištěné nedostatky lze nějakým způsobem odstranit.

Rychlost triangulace pro *náhodně* vkládané body je uvedena v tabulce 6.1.1.

Počet bodů	Čas celkem [s]	Procházka [s]	Legalizace [s]	Ostatní [s]	Využití paměti [MiB]
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 6.1.1: Rychlost jednotlivých částí triangulace pro různé počty náhodně vkládaných bodů a paměťová náročnost programu: počet vkládaných bodů, celkový čas triangulace, čas spotřebovaný algoritmem procházka, legalizace nových hran (přizpůsobení sítě Delaunayovu kritériu), ostatní části triangulace (konstruktory / destruktory / obslužný kód), celkové paměťové nárok programu

časy jsou vztaženy k mému počítači s procesorem Intel Core 2 Duo (2x1.66GHz), 1GB RAM, Windows XP

Vidíme, že pro velké množiny bodů spotřebuje většinu času algoritmus procházky. Při nasazení programu na tak obsáhlých datech by bylo nutné procházku nahradit nějakým efektivnějším algoritmem. Pro naše účely ale procházka stačí.

Časy vložení razítka pro různě rozsáhlé sítě s náhodně umístěnými body jsou uvedeny v tabulce 6.1.2.

Počet bodů	Celkový čas otisknutí VR nástroje [s]
100	0.000
1000	0.000
10000	0.016
50000	0.172
100000	0.375
500000	5.156

Tabulka 6.1.2: Čas vložení jednoduchého hloubícího razítka do různě složitých sítí. Razítko se skládá z osmi vynucených hran. Adaptivní vnější okraj není zapnutý. Časy jsou vztaženy ke stejnému počítači jako v tabulce 4.1.1.

Pro rozsáhlejší sítě opět spotřebuje většinu času algoritmus procházky. Síť obsahující 500 000 vrcholů už popisuje velice komplikovaný terén.

6.2 Nedostatky modelu

Pokud vynucujeme v síti hrany poblíž jiné hrany, která je nějakým výrazným rysem terénu, může se stát, že vzniklé trojúhelníky neodpovídají *Delaunayovu kritériu* a tato hrana je prohozena s jinou. Pokud nastane tato situace, terén se dost výrazně změní a výsledek už neodpovídá našemu očekávání. To nastane jen v případě, že načítáme model ze souboru. Při nahrání sítě totiž nejsou výrazné rysy terénu vynucené a tak je může legalizace hran zrušit. Při úpravě rovného terénu pomocí nástroje jsou všechny rysy tvořené vynucenými hranami.

Současná implementace virtuálního nástroje postupně ubírá z terénu při hloubení děr hmotu. Vnitřní část razítka totiž při vyrovnávání terénu odstraňuje všechny kopečky, a hmota z nich se tak ztrácí. Po určité době tedy v krajině „zmizí“ všechen písek a plocha je rovná.

V aplikaci zatím není zabudován žádný mechanismus na odstraňování nepotřebných detailů. Síť se tedy postupem času stává čím dál více složitou. Řešení je popsáno v kapitole 3.2.

Pro síť o milionu vrcholů je už vynucování hran razítkem nestabilní. Toto množství však není v praxi použitelné. Algoritmus procházky spotřebuje spoustu času a stávající vykreslování tak rozsáhlé sítě není pro *real-time* aplikace dostatečně rychlé.

V současné době není aktivní adaptivní vnější okraj virtuálního nástroje, protože se mi zatím nepodařilo úplně odladit algoritmus pro dělení hran. Místo toho je zatím použito druhé výše popsané řešení – automatické dělení vkládaných hran po pevných krocích.

Omezení pro tvar otisku nástroje vyplývá z omezení použitého modelu. Je možné simulovat jen nástroje, jejichž otisk lze popsat pomocí funkce dvou proměnných (tj. koule, jehlany, kužely apod.).

6.3 Možná řešení a vylepšení

Pokud bychom výrazné rysy terénu při načítání ze souboru předem detekovali a vynutili, nedocházelo by k jejich občasnému zrušení virtuálním nástrojem (jak je uvedeno výše).

Mizení hmoty z krajiny lze zabránit tak, že budeme zohledňovat sílu, kterou virtuálním nástrojem na terén působíme, a také hloubku vrypu. Podle těchto parametrů pak v příslušném směru „vytlačíme“ všechnu hmotu o objemu vyhloubené díry. Jedná se jen o rozšíření virtuálního nástroje. Díky použitému modelu bude toto rozšíření velmi implementačně náročné.

Pokud bych měl možnost psát tento program úplně znovu, zřejmě bych neimplementoval objekt typu hrana (*Edge*). Jeho zavedením se stalo každé prohození hrany v síti (ale i další operace) implementačně dost náročné. Na druhou stranu je nutné poznamenat, že hrany zjednodušily například vynucování hran v síti nebo správu sousednosti bodů.

V současné verzi vytváří program z jednoduché sítě postupně síť čím dál složitější. Pro opravdové nasazení je nutné přidat mechanismus, který bude kontrolovat, zda je možné nepotřebné body ze sítě vyloučit. To lze poměrně snadno realizovat například ve fyzikální vrstvě simulující erozi. Mezi změněnými body je totiž nutné vyhodnocovat jejich vzájemnou polohu. Nic tedy nebrání přidání události, která dá síti vědět, že daný bod už není potřeba a je ho možné odstranit.

Pro velké množiny dat tráví program většinu času prováděním algoritmu procházky. Pro nasazení tohoto programu na rozsáhlé sítě by bylo vhodné nahradit procházku efektivnějším algoritmem.

7. Závěr

Model se chová dobře, a nepravidelná trojúhelníková síť tedy je použitelná pro *real-time* aplikace na úpravu terénu. Implementačně je však celá záležitost dosti složitá. Všechny zjištěné nedostatky jdou nějakým způsobem odstranit. Pro dokonalou simulaci písku je potřeba ještě hodně věcí vylepšit.

Rád bych zdůraznil, že byl tento projekt mezinárodně řízen. Vedoucí geometrické části je Doc. Dr. Ing. Ivana Kolingerová (ČR). S vizualizací nám svými zkušenostmi pomohl Bedřich Beneš, Ph.D (USA). Animace ukazující výsledky práce byly prezentovány I. Kolingerovou v rámci přednášky o triangulacích na konferenci CESCG 2007 na Slovensku.

Podle výsledků experimentů a zpětné vazby zahraničních partnerů se předpokládají další úpravy práce v rámci oborových a diplomových prací.

8. Použité pojmy a zkratky

DT – Delaunay Triangulation

CDT – Constrained Delaunay Triangulation

RAD – Rapid Application Development

OpenGL – Open Graphics Library

SDL – Simple DirectMedia Layer

GLU – OpenGL Utility Library

GLUT – The OpenGL Utility Toolkit

SVN – Subversion

MinGW – Minimalist GNU for Windows

Eclipse CDT – Eclipse C++ Development Tools

UML – Unified Modeling Language

legalizace hran – přizpůsobení trojúhelníků *Delaunayovu kritériu* prohazováním jejich hran

VR nástroj – Virtual-Reality Tool (neboli nástroj, kterým v terénu vynucujeme změny)

9. Literatura a zdroje

9.1 Literatura

[BDA06] 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

[Kad07] KADLEC, J., *Deformace terénu pro virtuální realitu – geometrická část modelu,* Bakalářská práce KIV/ZČU, 2007

[Ono03] Onoue, K. – Nishita, T., *Virtual Sandbox*, 11th Pacific Conference on Computer Graphics and Applications (PG'03), pp 252–, 2003

[PK06] PURCHART, V. – KADLEC, J., *Modelování eroze a deformací terénu na povrchových modelech*, Dokumentace k Projektu 5, 2006

[Sed07] SEDMIHRADSKÝ, J., *Modelování eroze a deformací terénu*, Diplomová práce KIV/ZČU, 2007

[Slo93] 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

9.2 Elektronické zdroje

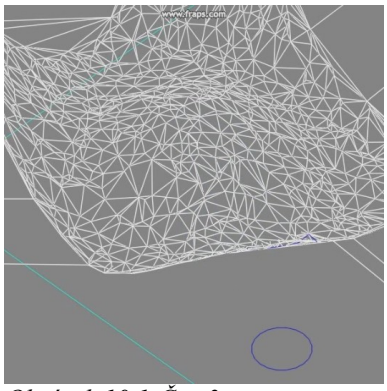
[DCP06] *Design by Contract Programming in C++*
URL: <http://www.eventhelix.com/RealtimeMantra/Object_Oriented/design_by_contract.htm>

[VAM07] KOLINGEROVÁ, I., *Rovinné triangulace a jejich aplikace*, přednášky KIV/VAM
URL: <http://iason.zcu.cz/~kolinger/VAM/VAM7.zip>

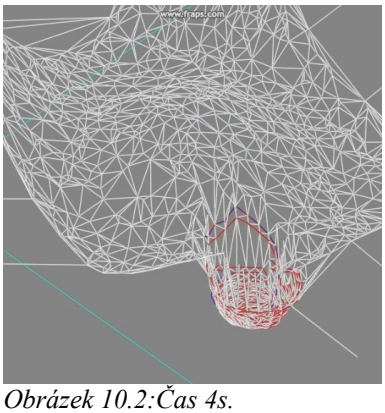
[WRE99] Weisstein, Eric W., *Line-Line Intersection*, MathWorld – A Wolfram Web
URL: <http://mathworld.wolfram.com/Line-LineIntersection.html>

10. Přílohy

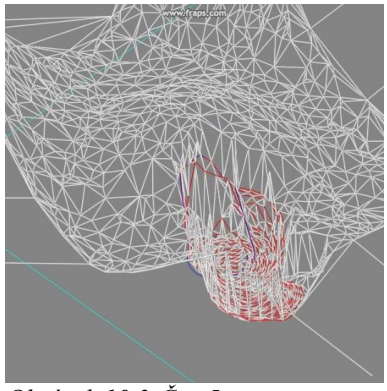
Postupné tažení kruhovým nástrojem s fyzikální vrstvou (erozí) od J. Sedmihradského.



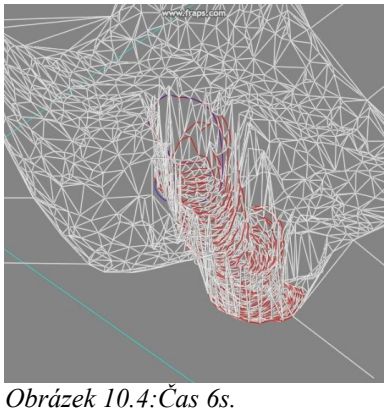
Obrázek 10.1:Čas 3s.



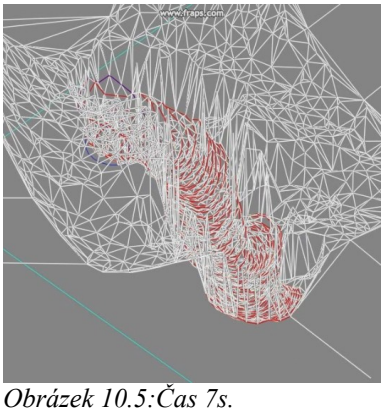
Obrázek 10.2:Čas 4s.



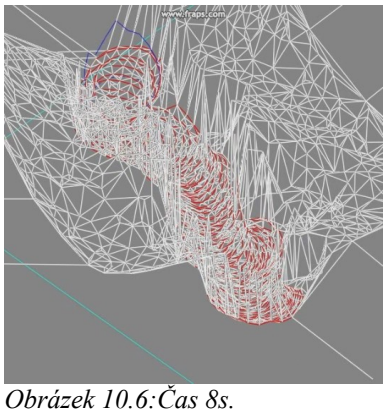
Obrázek 10.3:Čas 5s.



Obrázek 10.4:Čas 6s.

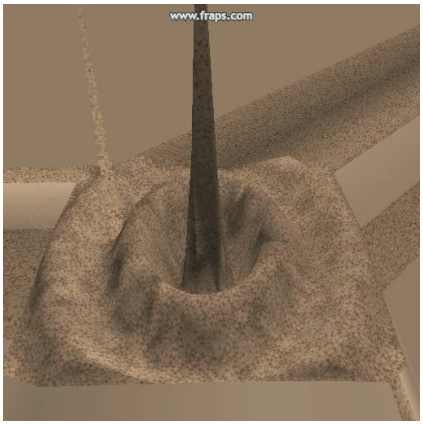


Obrázek 10.5:Čas 7s.

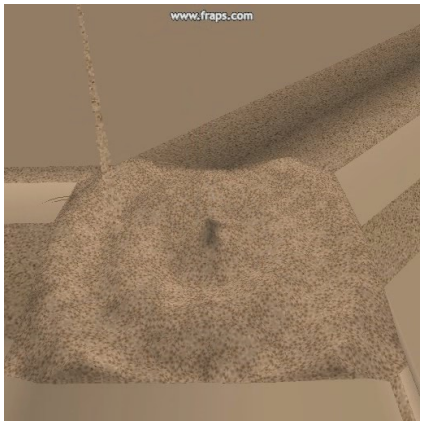


Obrázek 10.6:Čas 8s.

Otexturovaný model (vizualizace od J. Sedmihradského).

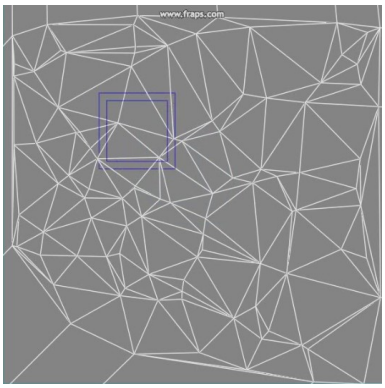


Obrázek 10.8:Otexturovaný model.

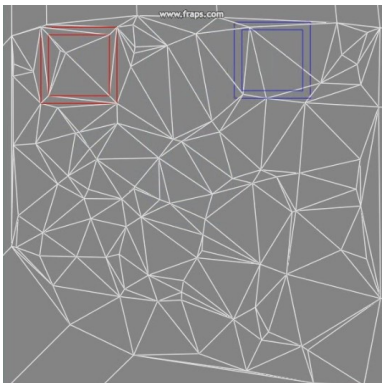


Obrázek 10.7:Erodovaný model.

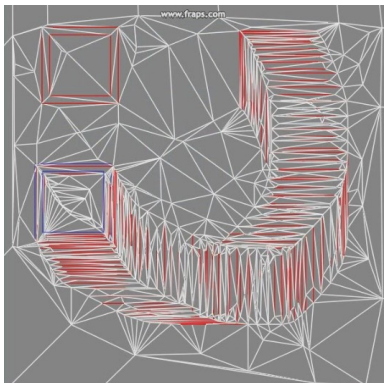
Vynucování změn v síti pomocí nástroje s otiskem ve tvaru čtverce.



Obrázek 10.9:Původní síť.

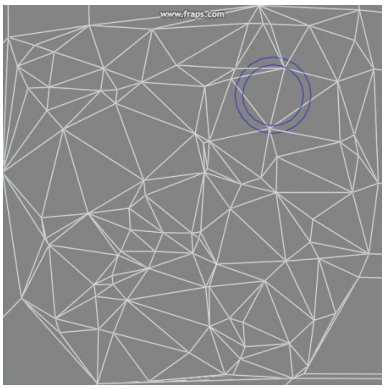


Obrázek 10.10:Jeden otisk.

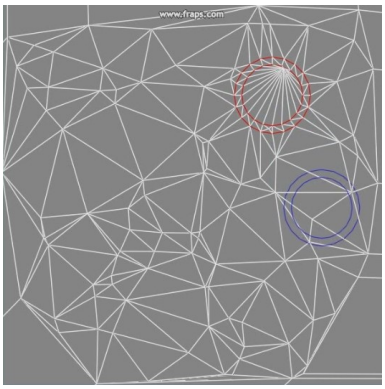


Obrázek 10.11:Tažení nástrojem.

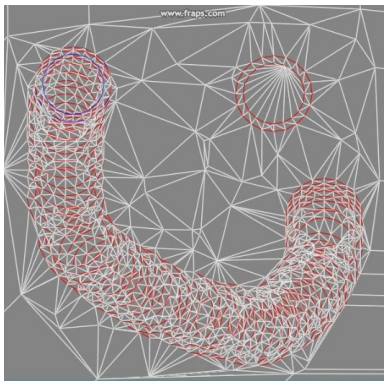
Vynucování změn v síti pomocí nástroje s kruhovým otiskem.



Obrázek 10.12:Původní síť.

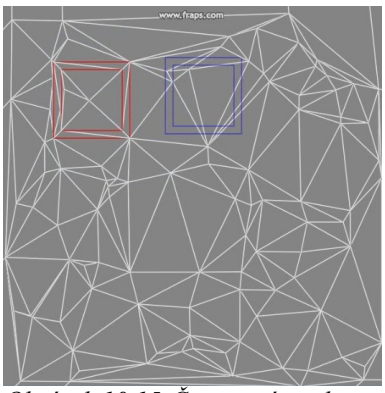


Obrázek 10.13:Jeden otisk.

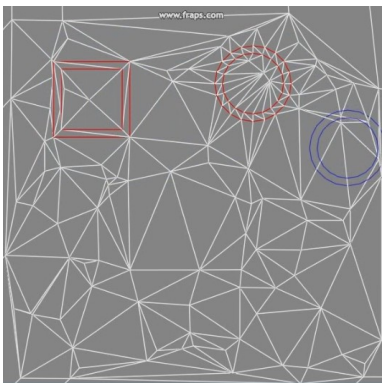


Obrázek 10.14:Tažení nástrojem.

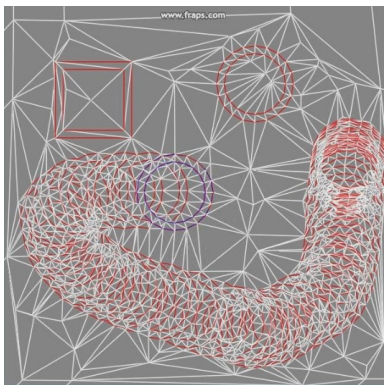
Použití více typů nástrojů současně (přepnutí nástroje).



Obrázek 10.15:Čtvercový otisk.



Obrázek 10.16:Kruhový otisk.



Obrázek 10.17:Tažení nástrojem.