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

Diplomová práce

Modelování eroze a deformací terénu

Plzeň, 2007 Jiří Sedmihradský

Poděkování

Chtěl bych poděkovat všem, kteří se na tomto projektu nějak podíleli, ať už svými nápady, či přímo prací na něm. Jmenovitě pak Doc. Dr. Ing. Ivaně Kolingerové za vedení mé diplomové práce a celého projektu, a také kolegům Janu Kadlecovi a Václavu Purchartovi za jejich spolupráci na tomto projektu.

Abstrakt

Simulation of erosion and deformation of terrain.

This paper deals with our solution of terrain simulation. In our work we try to test a new access to solve this task. Our solution consists in simulation of the terrain as a mesh. The mesh consists of points which represent the height of terrain. Points can be inserted and deleted from the mesh. We can deform the mesh by virtual tools and the mesh is influenced by gravitation erosion, which causes gradual recovery of the displaced material. Work on this task was divided to three persons. My part of work deals with visualization of the mesh and the design of algorithm for erosion.

Obsah

Poděkování....2
Abstrakt....3
Prohlášení:....5
1. Úvod....6
2.Teoretická část....8
2.1 Předchozí práce....8
2.2 Prostředky použité při realizaci....13
3. Základní struktury....22
3.1 Úvod....22
3.2 Realizace nepravidelné sítě a virtuálních nástrojů....22
3.3 Popis základních tříd....24
4. Algoritmy....25
4.1 Úvod....25
4.2 Vizualizace sítě....25
Zobrazování povrchu....25
Aplikace textur....28
Osvětlení....30
Zhodnocení:....32
4.3 Eroze....33
Úvod....33
Popis....33
Zhodnocení:....37
5. Realizace....38
5.1 Úvod....38
5.2 Vizualizace sítě....38
5.3 Eroze....43
5.4 Parametry....43
6. Výsledky....46
6.1 Vizualizace sítě....46
6.2 Zásahy do sítě a eroze....47
6.3 Výpočetní náročnost....52
7. Závěr....54
Literatura....55
Přílohy....56

Prohlášení:

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

V Plzni dne Jiří Sedmihradský

1. Úvod

Popisovaná práce se zabývá tvorbou a realizací počítačového modelu terénu tvořeného sypkými materiály, především pak pískem. Naším cílem bylo vytvořit model zobrazující povrch terénu, do něhož bude možné provádět zásahy virtuálními nástroji (tj. tělesy která se pohybují ve scéně, jsou ovládána uživatelem a definovaným způsobem mění tvar simulovaného povrchu). Terén bude na provedené zásahy reagovat pokud možno jako skutečný sypký materiál, tj. nástroj vytvoří v terénu prohlubeň odpovídajícího tvaru a v důsledku působení gravitace bude vzniklá prohlubeň zanesena okolním materiálem. Řešení celého problému musí být realizováno tak, aby umožňovalo případnému uživateli provádět veškeré změny a pozorování výsledků těchto změn interaktivně, tj. veškeré procesy musejí probíhat v reálném čase. Vzhledem k rozsahu celé práce daný úkol zpracovávali tito řešitelé: Jan Kadlec, Václav Purchart a Jiří Sedmihradský.

Rozdělení práce na řešení a výsledném programu bylo stanoveno následovně:

- Jiří Sedmihradský: simulace eroze terénu v důsledku působení gravitace na sypký materiál, vizualizace sítě představující povrch simulovaného terénu.
- Jan Kadlec a Václav Purchart: vytvoření a údržba sítě, triangulace nepravidelné sítě, vkládání a rušení bodů v síti a realizace zásahů nástroji do sítě představující simulovaný povrch.

S ohledem na již realizované práce v této oblasti, které budou popsány dále, a také s ohledem na zvažované pozdější rozšíření došlo k určitému upřesnění zadání. Především se jedná o způsob realizace povrchu terénu simulovaného materiálu. Ve většině dále uváděných prací byl povrch simulovaného materiálu realizován jako rozsáhlé dvourozměrné pole výškových souřadnic, které bylo následně za účelem zobrazení pokryto pravidelnou trojúhelníkovou sítí. Tento přístup přináší jistá pozitiva (jak bude zmíněno u popisu konkrétních prací), ale má i jeden významný nedostatek, a to značnou rozsáhlost pole reprezentujícího výšky (a následně tedy i trojúhelníkové sítě představující povrch), a to v závislosti na velikosti detailů povrchu, které chceme v modelu simulovat. To v konečném důsledku vede k nutnosti hledat rovnováhu mezi zachováním interaktivnosti celé aplikace a právě velikostí zobrazovaných detailů.

V našem případě byl proto předem zadán následující přístup. Povrch simulovaného terénu je modelován prostřednictvím nepravidelné sítě vrcholů (a trojúhelníků). Výhodou tohoto přístupu je to, že rozsáhlé plochy, které jsou rovné, je možno reprezentovat pomocí například pouze dvojice trojúhelníků a čtveřice vrcholů. Dojde-li v takové ploše k změně, například zásahem uživatelem ovládaného předmětu, dojde k zjemnění sítě v zasažených místech, tj. budou přidány vrcholy a trojúhelníky do sítě, a síť tak umožní simulovat dostatečně jemné detaily, aniž by vyžadovala rozsáhlé datové struktury. Navíc pokud například v důsledku působení eroze gravitací dojde k vyrovnání povrchu na určitém místě v simulovaném povrchu, lze přistoupit k odebrání některého z vrcholů. Tento postup samozřejmě má také své nevýhody a jeho použití vedlo k nutnosti řešit řadu problémů, jak bude uvedeno v dalších kapitolách.

Další upřesnění zadání se týkalo použitého programovacího jazyka a rozhraní pro samotnou realizaci. S ohledem na případné budoucí rozšiřování výsledného programu a požadavek na funkčnost na platformě linux/unix bylo rozhodnuto že vývoj bude probíhat v jazyku C++ a s využitím grafického rozhraní OpenGL. Vzhledem k tomu, že provádění

některých operací, jako například vytvoření základního okna aplikace, je značně platformně závislé a komplikované, byl navíc použit balík knihoven SDL a SDL_Image, díky čemuž bylo dosaženo výrazného zjednodušení některých částí kódu výsledného programu při zachování funkčnosti na požadovaných platformách.

Hlavní vlastnosti výsledného programu leze shrnout do těchto bodů:

1. Model terénu představující sypký materiál realizovaný prostřednictvím nepravidelné trojúhelníkové sítě s možností vkládat a rušit body (vrcholy sítě).
2. Do povrchu budou prováděny zásahy virtuálními nástroji.
3. Simulace eroze gravitací.
4. Programovací jazyk C++, rozhraní OpenGL, funkčnost na platformě unix/linux.

2.Teoretická část

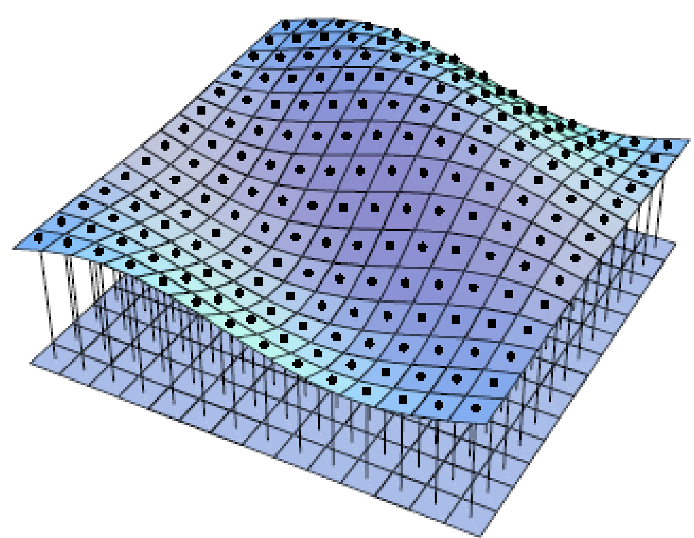
V této kapitole nejprve přiblížím v krátkém přehledu některé publikované práce, které se zabývaly tématicky podobnými náměty, a poté představím prostředky použité pro realizaci našeho řešení.

2.1 Předchozí práce

Popis

V oblasti počítačové grafiky bylo prezentováno již mnoho prací zabývajících se modelováním terénu a přidruženými problémy. A nemalá část těchto prací se zabývala také modelováním terénů tvořených sypkými materiály, jako je písek, sůl a podobně. Každá z těchto jednotlivých prací se pokoušela zaměřit podrobněji na některý z aspektů toho tématu a přinést nějaký nový poznatek. Z určitého úhlu pohledu lze tyto práce rozdělit do dvou hlavních kategorií, a to na simulace materiálů založené na fyzikálních modelech a na simulace kladoucí hlavní důraz na interaktivní průběh. Tyto kategorie se mohou částečně překrývat, přesto lze říci, že simulace založené na fyzikálně přesných modelech obvykle vylučují interaktivní simulaci počítanou v reálném čase.

Do první kategorie patří například práce [SUM98]. V tomto materiálu se autoři zaměřili na animace zásahů do různých materiálů, jako je písek, bahno či sníh. Základní datovou strukturou, která byla použita v této práci, je pravidelné dvourozměrné pole reprezentující výšky materiálu v jednotlivých bodech povrchu (viz Obr 2.1).



Obr 2.1: Dvourozměrné pole představující výšku povrchu (převzato z [SUM98])

Tento přístup k reprezentaci povrchu se poté opakuje i v dalších pracích popsaných dále, ale i u mnoha dalších. K výhodám této reprezentace patří velmi snadné získávání údajů o jednotlivých vrcholech sítě a jejich změna. Zjišťování sousedních prvků také nepředstavuje závažný problém a navíc vizualizace takto reprezentovaného povrchu není komplikovaná ani

výpočetně náročná. Pokud se rozhodneme na zobrazovaný povrch aplikovat texturu, je pravidelné pole opět vhodné a algoritmus relativně nenáročný. Lze najít jistě i mnohé další výhody. Zásadní nevýhoda tohoto přístupu je zmíněna právě například v [SUM98]. Je jí rozsáhlost datové struktury reprezentující povrch. Počet prvků pole musí totiž odpovídat nejmenším detailům, které chceme simulovat. Což například znamenalo, že v práci [SUM98] autoři použili datové struktury o rozměrech 2048x1024 prvků (ve výsledku více než 2 miliony prvků). Následně autoři do této datové struktury prováděli simulace zásahů různými předměty (například bota, kolo apod.) a simulace následných reakcí materiálu na tyto zásahy, jako je vytlačení materiálu v místě kontaktu s objektem a následná „eroze“ způsobená gravitací. Navíc model obsahoval i částicový systém, kterým bylo simulováno vymrštění části materiálu při prudkém kontaktu s předměty a jejich následný dopad zpět na povrch. Vzhledem k rozsahu zpracovávaných dat zde autoři přistoupili i k různým optimalizacím a k paralelizaci celého výpočtu. Výsledkem pak byly velmi kvalitní animace a obrázky (viz Obr 2.2).

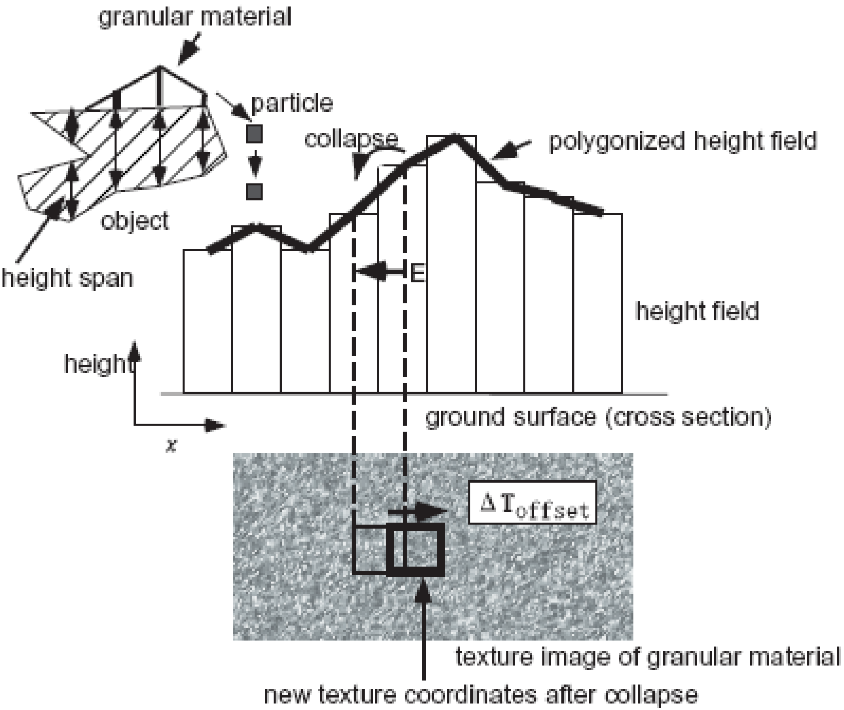
Obr. 2.2: Výsledek simulace (převzato z [SUM98])

Jako zástupce druhé kategorie lze uvést například práce [BEN06] a [ONO03]. Vyznačují se použitím přibližných modelů, které umožňují provádět výpočty v reálném čase. Podobně jako u práce popsané výše je základní datovou strukturou představující simulovaný povrch pravidelné výškové pole, kde jednotlivé elementy pole představují hodnotu výšky terénu v daném bodě. Což s sebou přináší i výše popsané výhody a nevýhody.

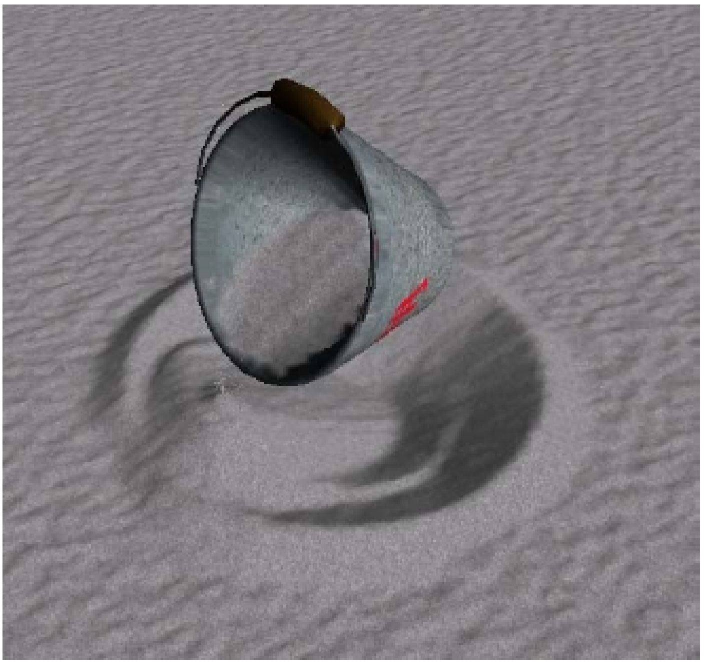
V práci [ONO03] se autoři zaměřili především na vizuální kvalitu simulace. Prezentované řešení opět umožňuje provádět zásahy do sítě prostřednictvím virtuálních nástrojů. Po kontaktu sítě objektem dochází k deformaci sítě na zasažených vrcholech sítě a také dochází k navýšení vrcholů ležících v bezprostředním okolí zasažených vrcholů, čímž je simulováno vytlačení části materiálu objektem. Objekt může být do simulovaného povrchu obtisknut nebo může být povrchem tažen, přičemž zanechává v povrchu příslušnou stopu. Po odstranění objektu pak dochází na povrchu k erozi způsobené gravitací. K urychlení výpočtů kolize objektů s povrchem se využívá tzv. bounding boxů a také takzvané HS (Height Spans) mapy, což je vlastně dvourozměrné pole určující výškové rozložení objektu. V práci byl také zpracován postup umožňující to, aby zasažený materiál zůstával na předmětu a případně z něho spadával zpět na povrch.

Materiál padající z povrchu objektů byl pak zobrazován prostřednictvím částic

(vykreslovány jako jednotlivé body), které po dopadu způsobovaly navýšení příslušné části povrchu. Jak jsem již zmínil, autoři se v této práci zaměřili především na vizuální stránku celé simulace, což se projevilo jednak v tom, že při vykreslování sítě představující povrch simulovaného terénu na tento povrch aplikovali texturu připomínající písek. Dále na tuto texturu aplikovali takzvaný „posun textur". Tato úprava byla použita pro zvěrohodnění eroze probíhající na terénu. Principem celého postupu bylo to, že při erozi způsobené gravitací docházelo k vyrovnání výšky mezi příliš vysoko umístěnými body a jejich níže položenými sousedy. Pokud byla během tohoto procesu aplikována na povrch textura „napevno", vykazovala výsledná simulace určitou nevěrohodnost. Autoři tedy využili znalosti o tom, k jakým přesunům došlo, a poté posunuli u příslušných vrcholů sítě také texturu (viz Obr. 2.3, v horní části obrázku je vidět HS mapa objektu na němž je navršeno určité množství materiálu, který spadává z okraje objektu. V spodní části obrázku je pak zobrazen již zmíněný posun textur na základě přesunu materiálu, podrobnosti viz [ONO03]). Výsledkem všech těchto technik bylo to, že se autorům podařilo dosáhnout velmi věrohodné simulace, a to i při použití celkem komplikovaných virtuálních nástrojů, které byly při simulacích použity (viz Obr. 2.4).



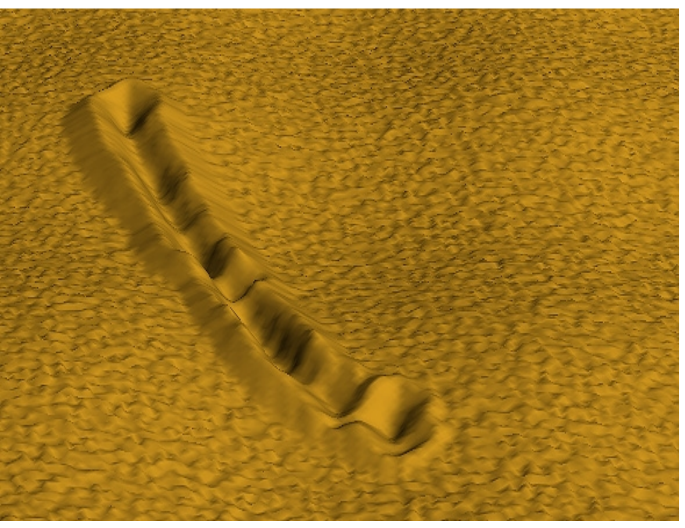
Obr. 2.3: Generování částic představujících písek padající z objektů a posun textur (převzato z [ONO03])



Obr. 2.4: Výsledek simulace (převzato z [ONO03])

Oproti předchozí uvedené práci se v materiálu [BEN06] autoři zaměřili spíše na využití tzv. haptických zařízení. Základní datovou strukturou bylo stejně jako v předchozích případech pravidelné výškové pole. Jako objekt provádějící zásahy do modelovaného terénu byla použita kulička. Objekt může opět v písku tažením zanechávat stopu a na vzniklé nerovnosti v terénu má vliv gravitační eroze, která je vyrovnává. K manipulaci s kuličkou uživatel používá takzvaná haptická zařízení (viz Obr. 2.5). Haptická zařízení neboli zařízení se silovou zpětnou vazbou slouží k navození pocitu, že uživatel manipuluje s předmětem v reálném světě. Principiálně se jedná o soustavu servomotorů působících na uživatele (ruku, prsty atd.) určitými silami, které spočte aplikace na základě pohybu uživatele v simulaci. V tomto konkrétním případě se jednalo o odpor simulovaného povrchu při „rýpání" kuličkou. Pro navození správného pocitu uživatele je tedy klíčový výpočet sil působících na kuličku při kontaktu se simulovaným terénem. Tato síla se skládá ze dvou složek, a to složky působící kolmo k zasaženému terénu (směrem od něj), která představuje odpor materiálu při pronikání kuličky, a ze složky působící vertikálně. Tato složka pak představuje tření kuličky o terén. Vzhledem k snaze autorů dosáhnou plynulosti změn sil v haptickém zařízení probíhaly výpočty těchto sil v jiné výpočetní smyčce než samotná vizualizace na obrazovku. Zatímco u vizualizační smyčky postačovalo 60 průběhů za sekundu k vnímání obrazu jako plynulého, u smyčky obsluhující haptické zařízení je potřeba asi 1000 aktualizací za sekundu k vnímání plynulosti. (viz Obr. 2.6).

Obr. 2.5: Haptická zařízení 6DOF Phantom (vlevo) a 3DOFOmega (převzato z [BEN06])



Obr. 2.6: Výsledky simulace dosažené v [BEN06] (převzato z [BEN06])

Pokud se teď vážený čtenář pozastavuje nad tím, proč jsou zde dopodrobna popisovány výše uvedené práce, které vycházejí z odlišného pojetí simulovaného terénu (respektive jeho povrchu), než jaký byl uveden v úvodní kapitole, tak je to především ze dvou důvodů. Tím prvním a podstatným je to, že výše popsané postupy jsou použitelné (s určitými úpravami) i pro nepravidelné sítě představující simulovaný povrch terénu. A tudíž jsme se jimi nechali při naší práci inspirovat a vycházeli jsme z nich při dalším vývoji (například způsob vykreslování sítě, eroze atd.), nebo se s jejich implementací počítá pro případná další rozšíření (například rozhraní pro haptická zařízení). Druhým důvodem bylo to, že se nám nepodařilo narazit na žádnou práci, která by pracovala s námi použitou datovou strukturou.

Zhodnocení předchozích prací

Každá z prací přinesla k realizaci daného tématu nové příspěvky a vylepšení. V případě práce prezentované v [BEN06] to bylo použití haptických zařízení, která vnášejí do celé aplikace vyšší míru realističnosti vnímání prováděných činností. V případě práce [ONO03] je celá řada zajímavých návrhů zlepšujících vizuální stránku simulace a výsledné animace působí skutečně nesmírně realistickým dojmem. Společným problémem všech tří popisovaných prací pak zůstává použití pravidelného pole představující simulovaný terén, které vede k značnému nárůstu výpočetní složitosti programů při zvětšujícím se počtu prvků, tj při snaze simulovat rozsáhlejší plochy nebo větší detaily. Možným řešením tohoto

problému je použití jiných datových struktur představujících povrch terénu. Jedná se o nepravidelné sítě a adaptivní pravidelné sítě. Adaptivní sítě představují vlastně pravidelné sítě, které na počátku obsahují malý počet prvků (velká pravidelná pole), která jsou v případě potřeby zjemňována pravidelným dělením. U nepravidelných trojúhelníkových sítí nejsou body vkládány do žádné pravidelné mřížky a tvoří vrcholy trojúhelníků. Nevýhodami adaptivních sítí jsou především ztráty základních vlastností pravidelných sítí, tj. jednoduchá adresovatelnost jednotlivých prvků a pravidelná vzdálenost mezi prvky. U nepravidelných trojúhelníkových sítí lze za hlavní nevýhodu označit především značnou programovou složitost.

Cílem naší práce pak bylo vyzkoušet použití nepravidelných trojúhelníkových sítí pro simulaci terénu. Trojúhelníkové sítě umožňují vkládat a ubírat vrcholy sítě, která představuje simulovaný povrch, čímž by mělo být možné dosáhnout simulace velmi malých detailů i u rozměrných ploch, při zachování relativně malého objemu dat.

2.2 Prostředky použité při realizaci

V úvodu byly zmíněny požadavky na použité grafické rozhraní a programovací jazyk. V této kapitole se tedy zaměřím především na rozhraní OpenGL a knihovny SDL, přičemž budu předpokládat, že s jazykem C++, je čtenář dostatečně obeznámen (budou-li však v dalším textu použity názvy některých méně obvyklých datových struktur či metod, pokusím se je krátce ozřejmit).

Knihovna OpenGL

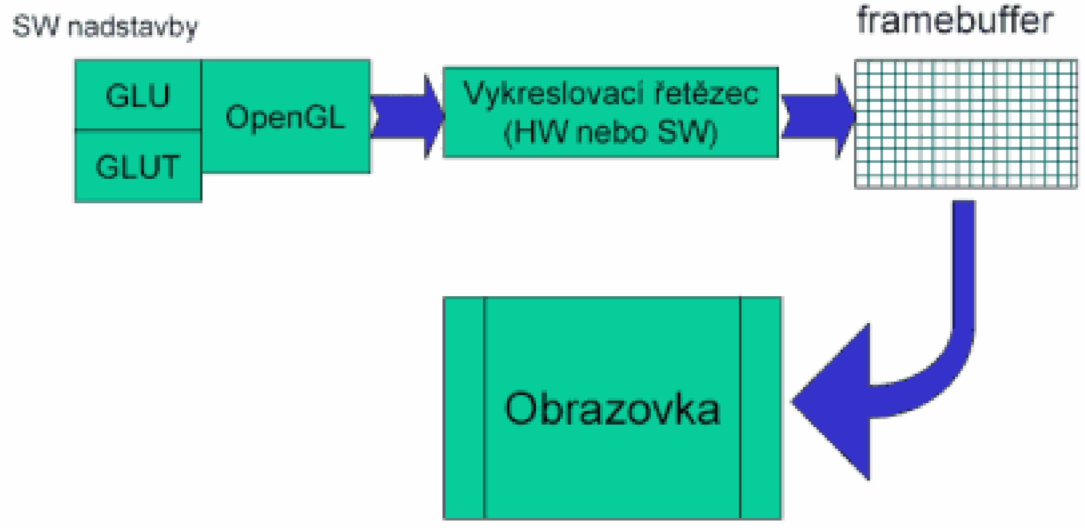
(části textu byly převzaty z [TIS03]).

Knihovna OpenGL™(Open Graphics Library) byla navržena firmou SGI (Silicon Graphics Inc.) jako aplikační programové rozhraní (Application Programming Interface - API) k akcelerovaným grafickým kartám, resp. celým grafickým subsystémům. Předchůdcem této knihovny byla programová knihovna IRIS GL (Silicon Graphics IRIS Graphics Library). OpenGL byla navržena s důrazem na to, aby byla použitelná na různých typech grafických akcelerátorů a aby ji bylo možno použít i v případě, že na určité platformě žádný grafický akcelerátor není nainstalován - v tom případě se použije softwarová simulace (která je samozřejmě výrazně pomalejší). V současné době lze knihovnu OpenGL použít na různých verzích unixových systémů (včetně Linuxu a samozřejmě IRIXu), OS/2 a na platformách Microsoft Windows.

Logo OpenGL a název OpenGL™ je registrovaná známka firmy Silicon Graphics Inc. Na některých platformách je možné rozdělení aplikace na dvě relativně samostatné části - serverovou a klientskou. Při vykreslování se potom jednotlivé příkazy (což jsou většinou parametry funkcí OpenGL) přenášejí přes síťové rozhraní. Knihovna OpenGL (narozdíl od IRIS GL nebo Direct 3D) byla vytvořena tak, aby byla nezávislá na použitém operačním systému, grafických ovladačích a správcích oken (Window Managers). Proto také neobsahuje žádné funkce pro práci s okny (otevírání, zrušení, změnu velikosti), pro vytváření grafického uživatelského rozhraní (Graphical User Interface - GUI) ani pro zpracování událostí. Tyto funkce lze zajistit buď přímo voláním funkcí příslušného správce oken, nebo lze použít některou z nadstaveb, například již zmiňované SDL, jehož popis bude uveden dále. Pro dosažení co největší nezávislosti na použité platformě zavádí knihovna OpenGL vlastní primitivní datové typy, například GLbyte, GLint nebo GLdouble.

Programátorské rozhraní knihovny OpenGL je vytvořeno tak, aby knihovna byla použitelná v téměř libovolném programovacím jazyce. Primárně je k dispozici hlavičkový soubor pro jazyky C a C++. V tomto souboru jsou deklarovány nové datové typy používané knihovnou, některé symbolické konstanty (např. GL_POINTS) a sada cca 120 funkcí tvořících vlastní rozhraní. Existují však i podobné soubory s deklaracemi pro další programovací jazyky, například Fortran, Object Pascal či Javu; tyto soubory jsou většinou automaticky vytvářeny z „Céčkovských" hlavičkových souborů.

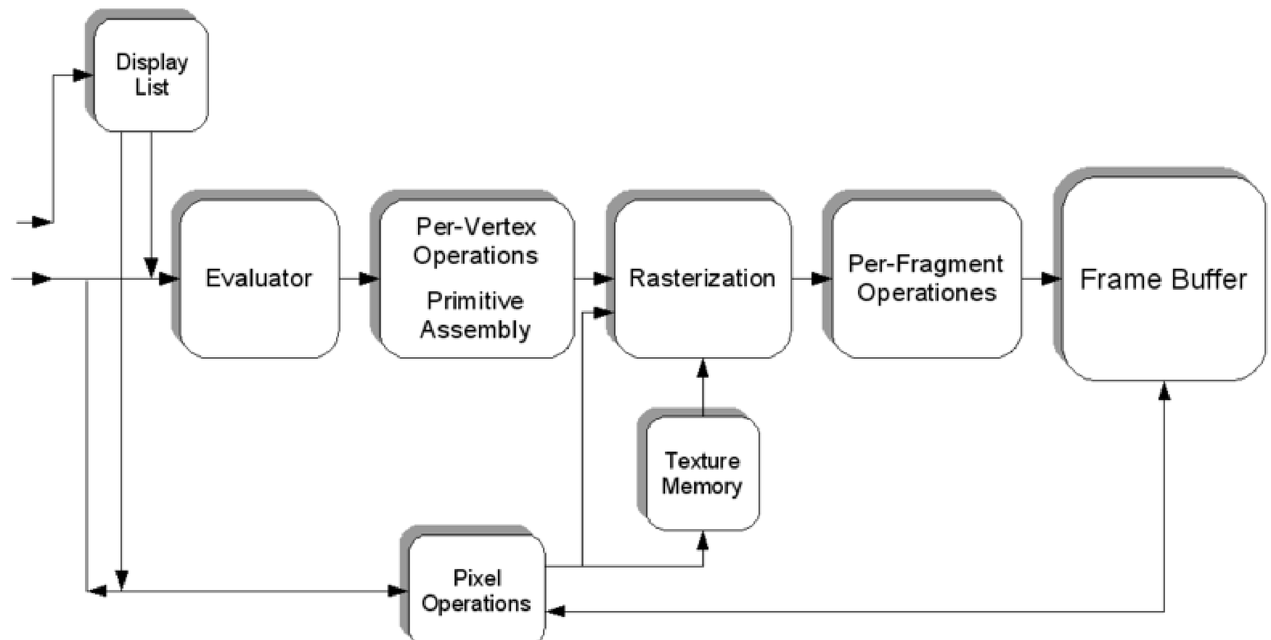
Z programátorského hlediska se OpenGL chová jako stavový automat (tato informace je nesmírně podstatná pro pochopení činnosti zdrojových kódů). To znamená, že během zadávání příkazů pro vykreslování lze průběžně měnit vlastnosti vykreslovaných primitiv (barva, průhlednost) nebo celé scény (volba způsobu vykreslování, transformace) a toto nastavení zůstane zachováno do té doby, než ho explicitně změníme. Výhoda tohoto přístupu spočívá především v tom, že funkce pro vykreslování mají menší počet parametrů a že jedním příkazem lze globálně změnit způsob vykreslení celé scény, například volbu drátového zobrazení modelu (wireframe model) nebo zobrazení pomocí vyplněných polygonů (filled model). Vykreslování scény se provádí procedurálně - voláním funkcí OpenGL se vykreslí výsledný rastrový obrázek. Výsledkem volání těchto funkcí je rastrový obrázek uložený v tzv. framebufferu, kde je každému pixelu přiřazena barva, hloubka, alfa složka popř. i další atributy. (viz Obr. 2.7 a Obr. 2.8).



Obr. 2.7 Postup vykreslování v OpenGL (převzato z [TIS03]).

Na Obr.2.8 jsou zjednodušeně schématicky znázorněny jednotlivé funkční bloky OpenGL. Příkazy OpenGL vstupují vlevo. Některé z nich mohou být uloženy v Display List a vyvolány později. Ostatní jsou poslány přímo ke zpracování. V prvním bloku Evaluator jsou aproximovány křivky a plochy vyčíslením polynomiálních funkcí. Jsou vytvořena primitiva, se kterými umí OpenGL pracovat. V dalším bloku jsou primitiva zpracována. Jsou transformovány vrcholy, vypočteny normály a osvětlení, je realizován clipping (ořezání částí scény ležících mimo obrazovku). V následujícím bloku je provedena rasterizace, kde je realizována vlastní projekce 3D obrazu do 2D. Každému bodu se přiřazuje hloubka a barva v závislosti na vzdálenosti a barvě objektu, zdrojích světla, modelech počítání světla, mapovaných texturách apod. Výsledkem rasterizace jsou tzv. fragmenty. K fragmentu jsou přidruženy informace o jednom pixelu výsledného obrazu. V předposledním bloku jsou s fragmenty provedeny různé operace, jako jsou např. scissor test, alpha test, depth test a

blending. A konečně jsou pixely zapsány do framebufferu, odkud se pak vykreslují na obrazovku. Pixely mohou být také z framebufferu čteny a kopírovány z jedné části framebufferu do druhé. Podobně může být přeskočena část pipeline pro zpracování vertexů, tj. můžeme poslat fragmenty přímo ke zpracování do fragmentové části pipeline. To vše nám naznačují šipky v obrázku. (text částečně převzat z [Simb06])



Obr. 2.8 Diagram grafické pipeline v OpenGL – vykreslovací řetězec (převzato z [Wiki])

OpenGL nezaručuje, že při spuštění identického programu používajícího knihovnu OpenGL na různých platformách nebo různých grafických akcelerátorech dostaneme vždy přesně stejný výsledek. Pokud bychom oba výsledné rastrové obrázky porovnali pixel po pixelu, mohli bychom zjistit mírné rozdíly v barvách. Může to být způsobeno například odlišnou přesností reprezentace čísel na grafické kartě, odlišnými algoritmy pro interpolaci barvy, normály a texturových souřadnic nebo jinou bitovou hloubkou Z-bufferu. Celkové geometrické a barevné podání scény by však mělo být zachováno.

Pomocí funkcí poskytovaných knihovnou OpenGL lze vykreslovat obrazce a tělesa složená ze základních geometrických prvků, které nazýváme *grafická primitiva*. Mezi tato primitiva patří bod, úsečka, trojúhelník, čtyřúhelník, plošný konvexní polygon, bitmapa (jednobarevný rastrový obraz) a pixmapa (barevný rastrový obraz). Existují i funkce, které podporují proudové vykreslování některých primitiv - lze například vykreslit polyčáru (line loop), pruh trojúhelníků (triangle strip), pruh čtyřúhelníků (quad strip) nebo trs trojúhelníků (triangle fan). Na vrcholy tvořící jednotlivá grafická primitiva lze aplikovat různé transformace (otočení, změna měřítka, posun, perspektivní projekce), pomocí kterých lze poměrně jednoduše vytvořit animace. Vykreslovaná primitiva mohou být osvětlena nebo pokryta texturou. Dále je možné celou vykreslovanou scénu "skrýt" v mlze.

Vzhledem k skutečnosti že slovo **display list** se bude v následujících odstavcích vyskytovat poměrně často, pokusím se tento prvek OpenGL popsat. *Display-listy* si můžeme představit jako makra, do kterých se nahraje několik příkazů OpenGL, a tato makra lze potom

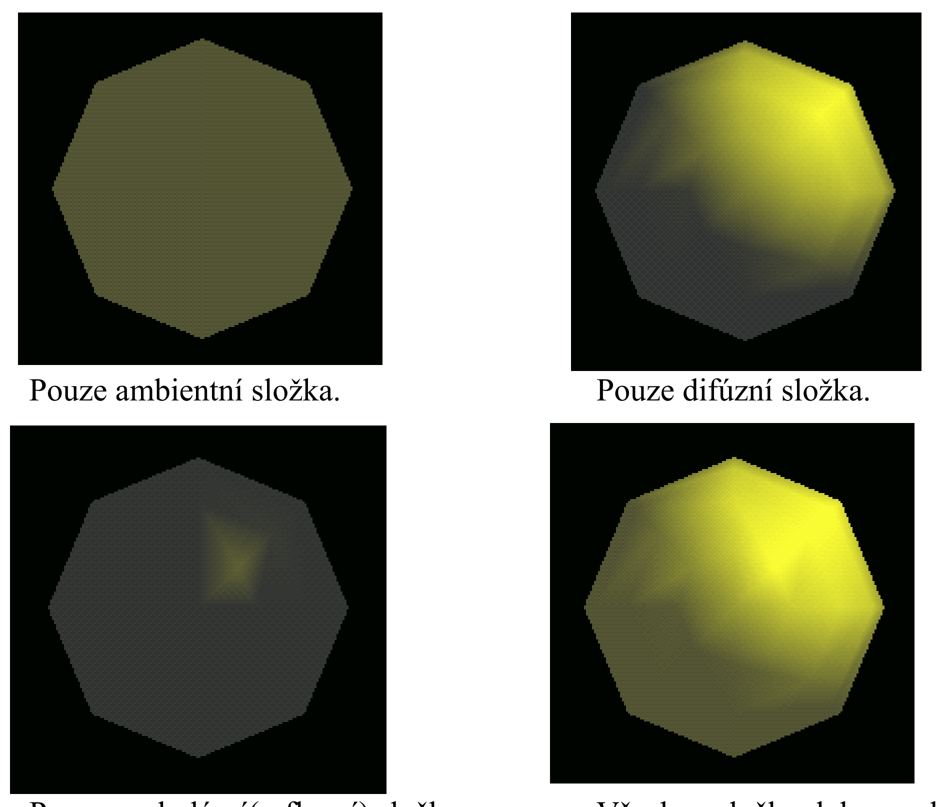
jedním příkazem "spustit". Výhodou display-listů je na jedné straně zvýšená rychlost vykreslování, protože display-listy jsou většinou uloženy přímo v paměti grafického akcelerátoru, na straně druhé také zjednodušení kódu pro vykreslování komplikovaných scén. Začátek záznamu do display-listu se povolí příkazem void glNewList(), ukončení záznamu příkazem void glEndList(). V příkazu void glNewList(GLuint list, GLenum mode) zadáváme dva parametry. První celočíselný parametr *list* představuje identifikátor vytvořeného display-listu. Pomocí tohoto identifikátoru můžeme display-list vyvolat. Druhý parametr *mode* určuje, zda se má display-list pouze vytvořit (hodnota GL_COMPILE), nebo vytvořit a přímo provést (hodnota GL_COMPILE_AND_EXECUTE). Většina volání příkazů OpenGL (jsou však výjimky) je do display-listu zaznamenána, ostatní příkazy se samozřejmě provedou. Provedení display-listu (tj. vyvolání příkazů uložených v display-listu) realizuje funkce void glCallList(GLuint list), v jejímž parametru *list* je uložen identifikátor dříve vytvořeného display-listu (viz Obr 2.9).

```
glNewList(lists[i][j], GL_COMPILE_AND_EXECUTE);
glBegin(GL_TRIANGLES);
//vrchol  A
        glTexCoord2d(0.5,0.5);
        glVertex3f( A->getX(), A->getY(), A->getZ() );
//vrchol B, c = velikost strany AB, TC = konstanta
        glTexCoord2d(0.5+(c*TC), 0.5);
        glVertex3f( B->getX(), B->getY(), B->getZ() );
//vrchol C, alfa = úhel při vrcholu A, b = velikost strany AC
        glTexCoord2d(0.5 + (sin(0.5*PI - alfa)*b*TC), 0.5 + (sin(alfa)*b*TC));
        glVertex3f( C->getX(), C->getY(), C->getZ() );
glEnd();
glEndList();
```

Obr. 2.9: Ukázka zdrojového kódu, který vytvoří display list (v display listu bude vykreslen jeden otexturovaný trojúhelník).

Dalším podstatným prvkem OpenGL, který bude v následujícím textu zmiňován, je tzv **Vertex Buffer Object (VBO)**. VBO je v podstatě datová struktura OpenGL, kerá umožňuje přímo zapisovat objekty určené k vykreslení do paměti příslušející GPU (graphics processing unit – grafický procesor). Paměť GPU se nachází buďto přímo na grafické kartě společně s GPU, nebo je takzvaně „sdílená“, což znamená, že GPU využívá části paměti RAM. Pokud data představující vykreslovaný objekt umístíme přímo na tuto paměť, mělo by teoreticky dojít k rychlejšímu zpracování těchto dat GPU, což znamená rychlejší vykreslování. VBO pod OpenGL je přístupný přes tzv Extension (rozšíření). V našem případě jsem použil knihovnu GLee, jejímž autorem je Ben Woodhouse (http://elf-stone.com), která umožňuje přístup ke všem potřebným funkcím o něco jednodušším způsobem než při použití samotných extensions. Před vytvořením samotného VBO je nejprve nutné vytvořit pole dat představujících příslušné parametry objektu (souřadnice vrcholů, normál atd...). Pole lze vytvořit pro daný objekt pouze jedno obsahující všechny potřebné údaje uspořádané postupně za sebou (vždy všechny údaje pro jeden vrchol, poté pro další atd.) anebo lze jednotlivé údaje uložit do samostatných polí. Poté se pomocí příkazu *glGenBuffers* vytvoří příslušný buffer (název funkce odpovídá použité knihovně, pokud použijeme jiný postup, jak využívat rozšíření OpenGL, může se mírně lišit). Poté je nutné pomocí příkazu *glBindBuffers* určit, že nadále budeme pracovat s tímto bufferem a příkazem *glBuffeData* specifikujeme data, která se odešlou do paměti GPU. Poté je ještě nutné určit OpenGL, které objekty představují který typ dat, což se provede příkazy *glNormalPointer* nebo *glVertexPointer* a podobně. V těchto

příkazech lze zároveň určit, jak jsou data uspořádána, a to za předpokladu, že jsme veškeré údaje uložili do jednoho pole. K vykreslení příslušných objektů poté použijeme například příkaz *glDrawArrays*. Před voláním této funkce je ještě nutné funkcí *glEnableKlientState()*, jejímž parametrem jsou symbolické konstanty určující typy zpracovávaných dat, povolit vykreslování příslušných dat.



Pouze ambientní složka. Pouze difúzní složka.

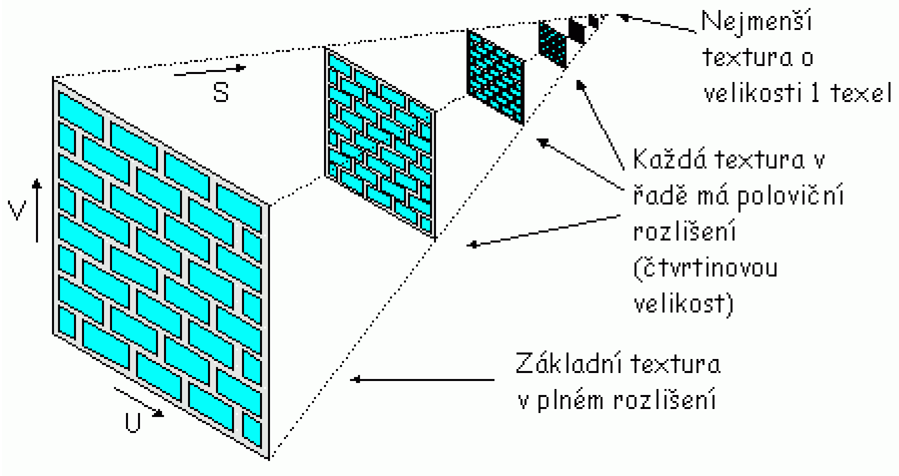
Pouze spekulární(reflexní) složka. Všechny složky dohromady.

Obr. 2.10: Složky světla v OpenGL (převzato z [CVUT02])

Další podstatnou vlastností všech grafických prostředí včetně OpenGL je osvětlení. Lze říci, že o tom, jakou barvu bude mít zobrazený objekt, rozhoduje několik faktorů. Zaprvé je to barva, respektive textura, aplikovaná na objekt, není-li v scéně zpracovávané OpenGL nastaveno žádné světlo, barva zobrazeného objektu odpovídá této barvě, respektive textuře. Druhým faktorem může být osvětlení scény, a to za předpokladu, že jej povolíme přepínačem glEnable(GL_LIGHTING). V takovém případě do „hry“ o výslednou barvu daného objektu začne vstupovat barva světla a pozice vykreslované plošky oproti pozici světla. OpenGL umožňuje požívat ve scéně nejméně 8 takzvaných hardwarových světel (tj. světel, u nichž výpočty osvětlení realizuje grafická karta). Každému z těchto světelných zdrojů lze nastavit typ (bodový zdroj, reflektor, atd.), barvu jednotlivých složek a pozici (případně i směr, útlum atd.). Světlo v OpenGL je tvořeno třemi složkami, a to ambientní, difúzní a spekulární. Význam jednotlivých složek lze pochopit při pohledu na obrázek 2.10.

Pro výpočet osvětlení jednotlivých plošek (trojúhelníků,...), z nichž se skládají objekty scény, jsou podstatné takzvané normály. Normála je vektor kolmý k dané plošce a lze ji zadat pomocí příkazu glNormal3{fidsb}(nx, ny, nz). Normála pro výpočet osvětlení musí být jednotkovým vektorem, to znamená že její velikost musí být rovna jedné. Toto lze zařídit buď provedením příslušného výpočtu u zadávaných údajů nebo tuto činnost můžeme přenechat samotnému OpenGL prostřednictvím přepínače glEnable(GL_NORMALIZE).

Významným prvkem, určujícím vnímání povrchu zobrazovaného tělesa, je textura aplikovaná na tento povrch. V jedné z dalších kapitol je popsáno, jakým způsobem je v našem případě aplikována textura na nepravidelnou trojúhelníkovou síť. Při zobrazování textury na povrchu tělesa může nastat několik problémů souvisejících s tím, že zobrazovaná textura může být v scéně umístěna různě vzhledem k pozorovateli (kameře), a tudíž rozlišení textury a rozlišení kamery pozorující scénu mohou být různá. Rozlišení zobrazované textury je tedy nutné prostřednictvím nějakého filtru „přepočítat“ na odpovídající rozlišení. Problémem ovšem je, že ani nejlepší filtr nevytvoří kvalitní obraz, pokud je nutno texturu přepočítat na vetší rozlišení, než je její původní (pokud se kamera přiblíží k objektu). Řešením je tedy použít dostatečně detailní(rozsáhlou) texturu, respektive v případě vizualizace povrchů, jako je písek (což je náš případ), aplikovat „dostatečně velkou část“ textury na jednotku povrchu. Nedostatkem tohoto řešení je to, že při pohledu z velké vzdálenosti na takovýto model, dochází při pohybu kamery k výraznému problikávání jednotlivých pixelů. Příčinou je to, že při pohybu kamery či otexturovaného tělesa, dochází mezi jednotlivými snímky k výběru texelů (texel = texture element), které jsou v rovině textury velmi vzdálené a běžné filtry, pracující na okolí několika málo texelů, nejsou schopny tyto rozdíly eliminovat. Nejčastějším řešením tohoto problému je takzvaný mipmapping, což je technika, při níž je textura uchovávána ve více rozlišeních (úrovních detailu), a při vykreslování otexturovaného povrchu se nejdříve zjistí relativní velikost povrchu vůči celé textuře a poté se vybere vhodné rozlišení textury, která se posléze nanese na povrch. Tento postup má nevýhodu v tom, že při postupném zmenšování objektu by docházelo ke skokové změně textury (přešlo by se k menšímu rozlišení textury). Proto se zavádí další úroveň interpolace („první“ úroveň interpolace může být provedena při samotné aplikaci mipmapy, stejně jako u běžné textury), kdy se vybere nejbližší větší a nejbližší menší textura a barva pixelů se vypočte interpolací mezi těmito dvěma texturami. Interpolovány jsou tedy jak texely v rámci jedné textury, tak i hodnoty mezi jednotlivými dvojicemi textur. Mipmapové textury si můžeme vytvořit sami nebo jejich tvorbu můžeme přenechat specializovaným funkcím implementovaným do grafických rozhraní. Podstatné je, že rozlišení mipmap by se mělo u každé další úrovně snižovat na polovinu v horizontálním i vertikálním směru (především z důvodu rychlejšího přístupu GPU k texturám), a to až do dosažení velikosti 1 x 1 pixel (viz. Obr. 2.11).

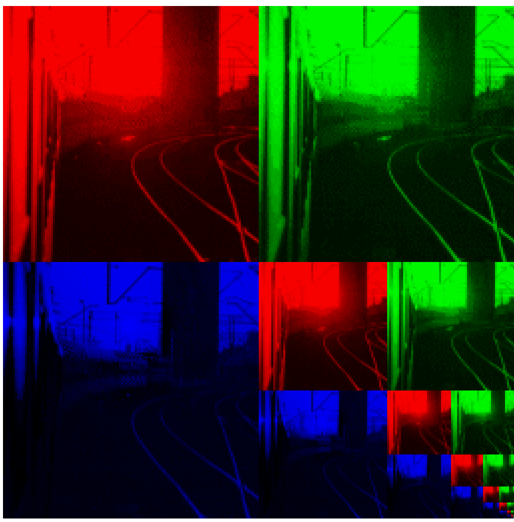


Obr. 2.11: Princip tvorby mipmapy (převzato z [TIS03]).

V OpenGL jsou samozřejmě mipmapy podporovány. Pokud jsme si mipmapy vytvořili sami, je nutné specifikovat funkcí *glTexImage2D(),* které textury tvoří jednotlivé úrovně

mipmapy. Další možností při tvorbě mipmap je vytvoření pomocí speciálních funkcí. Ty nabízí jednak nadstavbová knihovna GLU a jednak prostřednictvím příslušné extension (rozšíření nad vlastnosti dané specifikace OpenGL) i samo rozhraní OpenGL. Na Obr 2.12 je ukázka toho, jak je textura mipmapy uložena v paměti GPU. V obou případech dojde k automatickému vytvoření mipmap ze specifikované základní textury. I při použití mipmap dochází k interpolaci a je tedy možné nastavit, jaký filtr bude při interpolaci použit. Je možné použít následující filtry (převzato z [TIS03]):

1. Při nastaveném filtru *GL_NEAREST_MIPMAP_NEAREST* je pro obarvení pixelu vybrán nejbližší texel z nejbližší větší nebo menší textury. Tento filtr poskytuje vizuálně nejhorší výsledky, vykreslování je však nejrychlejší.
2. *GL_NEAREST_MIPMAP_LINEAR* - vybere dva nejbližší texely z obou textur a provede mezi nimi lineární interpolaci. Tímto filtrem se dají jednoduše odstranit nepříjemné skokové změny v obraze, které nastávají v případě, že se zobrazované těleso příliš zvětší nebo zmenší a provede se tak výběr jiné dvojice textur z mipmapy.
3. *GL_LINEAR_MIPMAP_NEAREST* - provádí billineární interpolaci nejbližších texelů v jedné textuře. Při zmenšování nebo zvětšování tělesa mohou nastat skokové změny ve vykreslované textuře.
4. *GL_LINEAR_MIPMAP_LINEAR* - nejprve se použije interpolace pro výpočet barev texelů v obou texturách a poté se výsledná barva spočte další interpolací mezi dvěma předešlými (ve skutečnosti se tedy provádí trilineární interpolace). Tento filtr je sice nejpomalejší, ale poskytuje nejlepší vizuální výsledky.



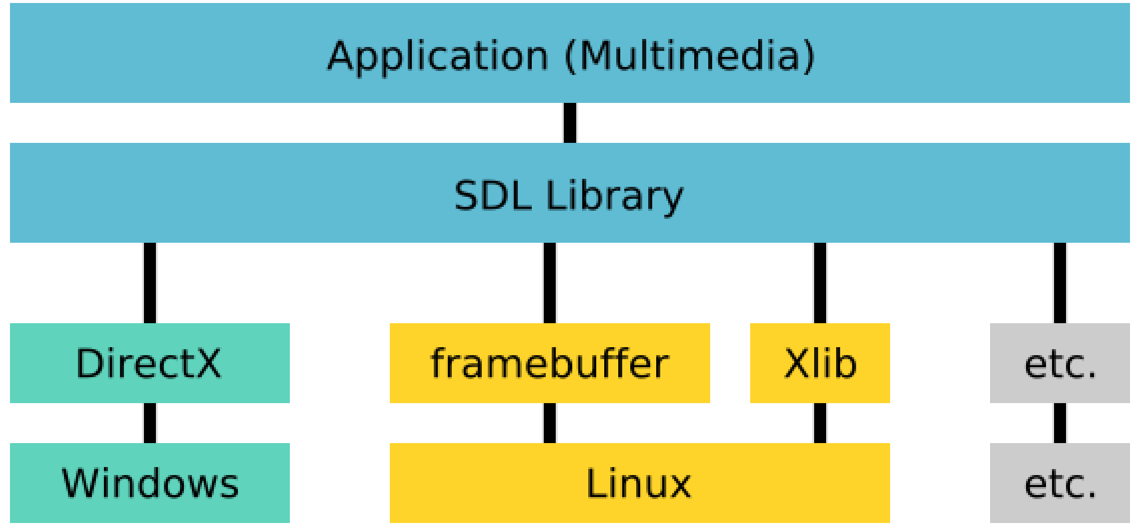
Obr. 2.12: Ukázka interní reprezentace mipmapy uložené v texturovací paměti grafického akcelerátoru (převzato z [TIS03]).

Knihovny SDL a SDL_image

(části textu převzaty z [Tur05])

SDL neboli Simple DirectMedia Layer je multiplatformní multimediální knihovna. SDL je k dispozici pod licencí GNU Lesser General Public License (GNU LGPL). Počátky knihovny SDL ukazují ke společnosti Loki Entertainment Software, která se zabývá přenosem her do operačního systému GNU/Linux, a jejímu hlavnímu programátoru Samu Lantingovi. Byla navržena jako obecné nízkoúrovňové API (aplikační programové rozhraní)

pro tvorbu her a obecně multimediálních aplikací. Z velké části zastřešuje funkce operačních systémů, a tím umožňuje téměř stoprocentní přenositelnost zdrojového kódu. SDL obsahuje funkce pro vytvoření okna (včetně tzv. fullscreenu) a správu událostí. Dvourozměrná grafika je zahrnuta přímo, přičemž jsou využívány prostředky poskytované daným operačním systémem, 3D grafika je realizována pomocí OpenGL, které má přímou podporu. SDL dále umožňuje práci s audiem, CD-ROM, časovači, obsahuje také podporu vícevláknového programování a obsluhu periferních zařízení, jako je klávesnice, myš či joystick. Knihovna SDL, jak již plyne z názvu (Simple...), obsahuje pouze „základní“ funkce. Vše "navíc" poskytují různé nadstavby, např. SDL_image pro nahrávání obrázků (samotné SDL umí nahrát pouze formát BMP), SDL_sound a SDL_mixer pro zvuky, SDL_ttf pro truetype fonty, SDL_net pro využití sítí a další. V současnosti SDL existuje pro celou řadu operačních systémů: GNU/Linux, MS Windows, BeOS, MacOS Classic, MacOS X, FreeBSD, OpenBSD, BSD/OS, Solaris, IRIX, a QNX. Dále je ho možno najít na Windows CE, AmigaOS, Dreamcast, Atari, NetBSD, AIX, OSF/Tru64 a SymbianOS, ale u těchto systémů neexistuje oficiální podpora. SDL bylo původně napsáno v jazyce C, a je tedy plně využitelné i v jazyce C++. V současnosti ale již existuje podpora i pro mnoho ostatních rozšířených programovacích jazyků. SDL má ve svém názvu slovo „Layer“, což značí, že SDL tvoří jakousi vrstvu mezi aplikací a na operačním systému závislými funkcemi (viz. Obr. 2.11). Pro každý jednotlivý podporovaný operační systém tedy existují příslušné SDL knihovny. Hlavičky funkcí a jejich parametry jsou však vždy totožné.



Obr. 2.11: SDL tvoří přechod mezi aplikací a různými platformami.(převzato z [Wiki]).

Velkou výhodou používání SDL je tedy to, že aplikace nepsaná pod jedním operačním systémem by měla bez větších zásahů být funkční i pod ostatními podporovanými systémy. Například v našem případě jsme s výhodou použili výše zmiňovanou schopnost SDL vytvořit a spravovat „okno“ aplikace a spravovat události vznikající za běhu aplikace. Pokud bychom přistoupili k realizaci bez použití této knihovny, byli bychom nuceni provádět tyto činnosti s využitím prostředků poskytovaných daným operačním systémem s tou nevýhodou, že výsledná aplikace by byla jen obtížně přenositelná na jiný operační systém.

Jak jsem již zmínil knihovna SDL nepodporuje jako vstupní obrazový formát jiný než formát BMP. Proto se společně s knihovnou SDL často používá také knihovna SDL_image,

které značně rozšiřuje počet možných vstupních obrazových formátů o tyto: PRM, XPM, PCX, GIF, JPEG, PNG a TGA. Knihovna SDL_image pak přidává funkci IMG_Load(const char*), která vrací vrací ukazatel na SDL_Surface, kde je načtený obrázek uložen. V našem případě jsme tuto knihovnu využili k načítání obrázků použitých následně jako textury.

V této kapitole byly popsány některé z předchozích realizovaných prací v oblasti simulací terénu a také základní programové prostředky, které byly použité při tvorbě programové části práce a o nichž se budou také zmiňovat následující kapitoly. Ty popisují naši podobu simulace terénu, jednotlivá programová řešení a jejich vlastnosti.

3. Základní struktury

3.1 Úvod

Tato kapitola se zabývá popisem základních datových struktur programu, realizovaných kolegy Kadlecem a Purchartem. Jejich práce se zaměřila především na realizaci nepravidelné sítě, která představuje simulovaný povrch, a na virtuální nástroje, umožňující uživateli provádět zásahy do tohoto povrchu. Dále vytvořili jednoduché vizualizační rozhraní, sloužící k zobrazení sítě jako drátového modelu. Tato kapitola je zde zařazena především z toho důvodu, že všechny jimi vytvořené části byly použity bez výrazných změn i do výsledného programu a mnou vytvořené a dále popisované algoritmy staví na jich práci.

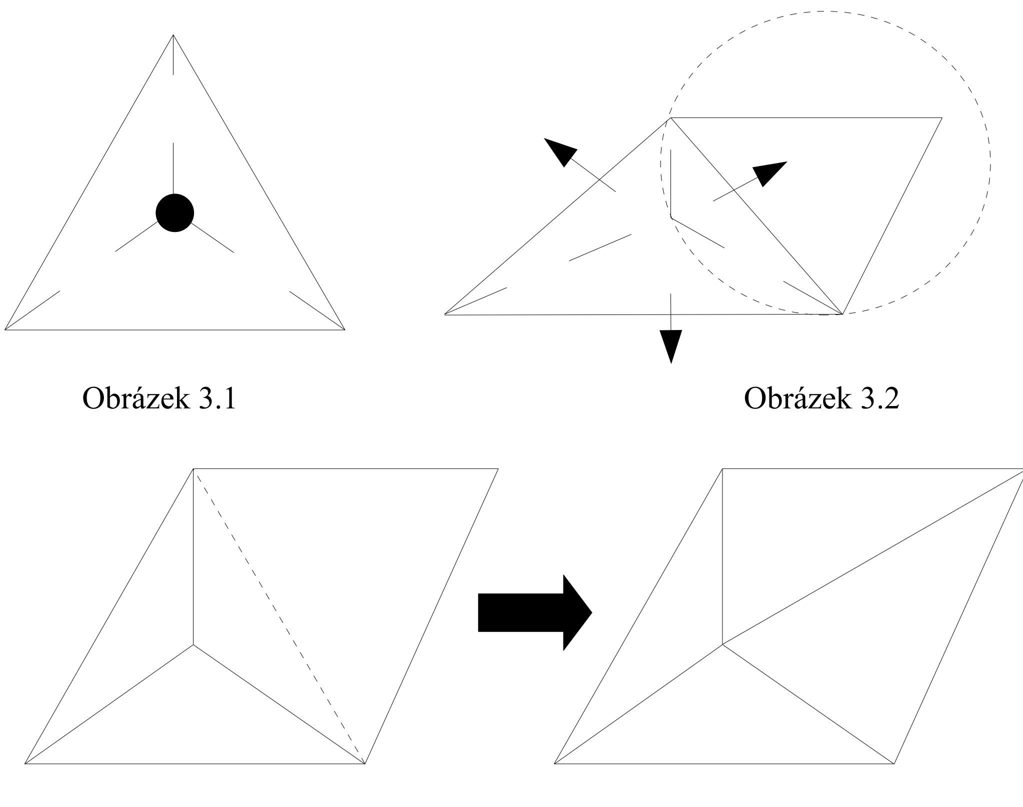
3.2 Realizace nepravidelné sítě a virtuálních nástrojů

Jak bylo již zmíněno v předchozích kapitolách, je simulovaný povrch tvořen nepravidelnou sítí bodů, které určují výšku simulovaného terénu v daném bodě sítě. Mezi těmito body je nutné zavést spojnice, a to z důvodu určení sousednosti vrcholů pro realizaci eroze povrchu, a také za účelem vizualizace sítě, respektive simulovaného povrchu. Pro vizualizaci je vhodné, aby jednotlivé vrcholy sítě tvořily vrcholy neprotínajících se trojúhelníků. Existuje značný počet postupů realizujících triangulaci sítě tvořené body, přičemž jednotlivé postupy se liší algoritmickou a realizační náročností a také vlastnostmi výsledných trojúhelníků. Kolegové v tomto případě zvolili takzvanou Delaunayovu triangulaci (DT), mezi jejíž přednosti lze zařadit existenci relativně rychlého a jednoduchého algoritmu a také to, že výsledné trojúhelníky se blíží trojúhelníkům rovnostranným. Přibližně rovnostranné trojúhelníky jsou výhodné nejenom z hlediska vizualizačního, ale také z hlediska pozdějšího zpracování, neboť snižují rizika různých numerických chyb v důsledku nepřesností. S ohledem na to, že do sítě budou prováděny zásahy různými předměty, bylo nutné použít verzi Delaunayovy triangulace, která umožňuje „vnutit" do sítě určitou hranu, která by v důsledku vlastností DT nebyla původně možná. Tato varianta DT bývá označována jako Constrained Delaunay triangulation (CDT). Následující odstavec, který popisuje použité algoritmy je převzat z [Kad07] (a obrázky taktéž).

Triangulace sítě

Triangulaci provádíme inkrementálním vkládáním nových bodů. Pokud síť neobsahuje žádné trojúhelníky, vytvoříme trojúhelník tak velký, aby pojal celou množinu vkládaných bodů. Po vložení bodu pak pomocí algoritmu procházky najdeme trojúhelník, v němž se daný bod nachází, a rozdělíme ho na tři menší trojúhelníky. Každý z těchto trojúhelníků bude určen nově vloženým bodem a dvěma body z původního trojúhelníku (viz obrázek 3.1). Hrany původního trojúhelníku testujeme na Delaunayovo kritérium prázdné kružnice (viz obrázek 3.2). 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 obrázek 3.3). Samozřejmě musíme opravit informace o sousedech, vrcholech, inverzích apod. Poté lavinovitě šíříme legalizaci pro všechny hrany obou trojúhelníků. Nově vytvořenou hranu netestujeme, jelikož máme

zaručeno, že kritérium splňuje.



Obrázek 3.1 Obrázek 3.2

Obrázek 3.3

Přidávání vynucených hran do sítě

- přidáme všechny body, které jsou ve vynucených hranách
- do sítě přidáme vynucené hrany, které už v síti jsou
- pomocí procházky najdeme trojúhelník obsahující první bod vynucené hrany
- dokola obejdeme startovní bod a najdeme takový trojúhelník, kterým prochází vynucená hrana
- z něj putujeme po trojúhelnících, jejichž hranu protíná vynucená hrana a dáváme je do seznamu
- projdeme seznam hran, které se protínají s vynucenou hranou, a pokud je to možné, prohodíme každou hranu ze seznamu s hranou, která v sousedních trojúhelnících tvoří diagonálu, s opačnou diagonálou. Tuto hranu si přidáme do seznamu nových hran, které pak legalizujeme. Pokud to možné není vybereme ji ze seznamu a dáme ji na druhý konec seznamu.

Virtuální nástroje

Virtuální nástroje umožňují uživateli provádět zásahy do simulovaného povrchu. Nástroje provádějí v podstatě změnu výšky terénu v daných místech, a to tak, že na určená místa do sítě jsou vloženy body, spojené tzv. vynucenými hranami, mající příslušnou výšku (dle tvaru nástroje). Bližší podrobnosti lze nalézt v bakalářské práci kolegy Purcharta [Pur07]. Mým příspěvkem do této části bylo pouze vytvoření nového nástroje (vznikl modifikací z již existujícího nástroje vytvořeného kolegy), který se snaží simulovat vytlačení materiálu při provádění zásahů nástrojem do terénu. Vytlačený materiál je simulován přidáním dalších

dvou řad vnucených hran do okolí „vtlačeného otvoru“ (ty jsou pak propojeny trojúhelníky tak, že vytvoří na síti tvar, přibližně odpovídající vytlačenému materiálu), které jsou umístěny do určité výšky nad terén. Umístění hran je řízeno parametry nástroje (hloubkou a průměrem).

3.3 Popis základních tříd

Realizovaný program využívá kromě značného počtu různých pomocných proměnných základní datové struktury, jejichž uml diagram je v příloze (viz Příloha 1). Tyto datové struktury slouží především k uložení údajů o síti, která představuje povrch simulovaného terénu. V přiloženém diagramu a i v následujícím popisu jsou zmíněny pouze podstatné datové struktury a jejich prvky. Výsledný program pak obsahuje i mnohé další, které jsou případně zmíněny v některých jiných kapitolách, k nimž náležejí.

Třída *Grid*: je základní třídou představující simulovaný povrch. Uchovává seznam (respektive dynamické pole) vrcholů sítě a trojúhelníků, které spojují jednotlivé vrcholy. Obsahuje metody umožňující přístup k jejím prvkům a jejich editaci.

Třída *Point*: představuje jeden vrchol sítě simulující povrch. Obsahuje údaje o souřadnicích vrcholu. Dále údaje o hranách, jejichž koncový bod tvoří, což je využíváno například při realizaci eroze k určení toho, kam přemístit erodovaný materiál. Dále obsahuje seznam indexů trojúhelníků, jejichž je vrcholem, což je využíváno při výpočtu normál pro vizualizaci.

Třída *Triangle*: představuje jeden trojúhelník ze sítě. Obsahuje odkazy na trojici vrcholů *(Point)*, na hrany, kterými je tvořen a na tří sousedy ležící „přes“ dané hrany. Sousedi jsou ukládáni, jak je zvykem, tj. soused označený jako *neighbour_a* leží proti vrcholu značenému jako *pointA*.

Třída *Edge*: představuje hranu spojující dvojici vrcholů, respektive hranu trojúhelníka. Obsahuje ukazatele na dvojici vrcholů které ji tvoří.

Třída *ErrosionSystem* (není uvedena v diagramu): realizuje výpočet eroze nad sítí tvořenou třídou *Grid*.

4. Algoritmy

4.1 Úvod

V předchozí kapitole byly popsány základní datové struktury a realizace nepravidelné sítě, která má představovat simulovaný povrch tvořený sypkým materiálem. K tomu, aby uživatel získal představu, že povrch, s nímž v programu pracuje, představuje skutečný materiál, je nutné danou síť zobrazovat tak, aby pokud možno co nejvíce připomínala reálný písek. Dále je nutné převést na síť chování skutečného materiálu, tj. pokud je do povrchu například vyryta rýha, okolní materiál ji částečně zaplní. Řešením těchto problémů, tedy vizualizace sítě jako povrchu sypkého materiálu a gravitační eroze realizující přesuny materiálu v síti se budu zabývat v této kapitole. Kapitola obsahuje popis problémů, na něž jsem při zpracovávání těchto problémů narazil a nástin jednotlivých zkoušených řešení.

4.2 Vizualizace sítě

Vizualizace sítě představující modelovaný povrch se skládá z několika samostatných podproblémů, které bylo zapotřebí řešit. Prvním problémem, kterému bude věnována pozornost, je to, jak zobrazit síť vrcholů a je spojujících trojúhelníků co nejefektivněji. Druhým problémem je aplikace textury na tuto síť. A konečně posledním je aplikace osvětlení.

Zobrazování povrchu

Úvod

Naším požadavkem bylo zobrazovat trojúhelníkovou síť, která představovala povrch sypkého materiálu (například písek). Zobrazovaná síť by měla být velmi rozsáhlá (řádově tisíce vrcholů trojúhelníků, které představují výšku terénu). Do sítě budou prováděny časté zásahy, a to nejen změnou souřadnic jednotlivých vrcholů sítě (především změna souřadnice představující „výšku“ terénu), ale také přidáváním a odebíráním bodů a prohozením hran sousedních trojúhelníků. Vizualizace by měla být dostatečně rychlá, aby bylo možno splnit požadavek na interaktivnost celé aplikace, a přitom by neměla být výpočetně příliš náročná.

Popis

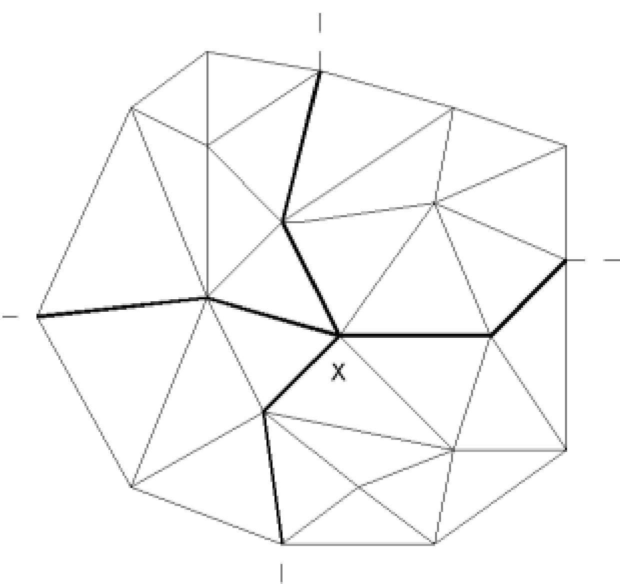
Prvním problémem, o němž jsem se v úvodu k této části zmínil, je zobrazování velmi rozsáhlé sítě, v níž mohou probíhat časté změny. S ohledem na zadané programové prostředky (rozhraní OpenGL) je možno najít mnoho řešení, přičemž já jsem v rámci této práce vyzkoušel dvě různé varianty. V obou variantách je nepravidelná trojúhelníková síť vizualizována jako jednotlivé trojúhelníky.

Display listy

První varianta vychází z postupu popsaného již například v [BEN06]. Daná síť je

zobrazována prostřednictvím dvourozměrného pole tvořeného OpenGL display listy, tj. síť je rozdělena na pravidelné celky v osách x, y a každý trojúhelník sítě náleží do jednoho z display listů. Pokud tedy dojde v síti ke změně, je aktualizován pouze příslušný d-list. Samotná realizace tohoto přístupu je ovšem komplikována možností přidání, respektive odebrání bodu v síti a také možností prohazování hran trojúhelníků. Jsou-li totiž trojúhelníky, u nichž tyto případy nastanou, „uvnitř" display listu, nedochází k výrazným komplikacím. Budeme-li však například provádět prohození hran u trojúhelníků náležejících do dvou různých display listů, nastává problém, jenž je nutno řešit. Zdrojem další komplikace je způsob, jímž jsou samotné trojúhelníky, ale i vrcholy sítě uloženy. Trojúhelníky nejsou nikterak seřazeny, například dle souřadnic (s ohledem na způsoby manipulace se sítí by to ani nebylo rozumně realizovatelné), a jsou uloženy v dynamickém poli. Tudíž pokud chceme provést aktualizaci pouze jednoho z display listů, je nutné nějakým způsobem určit, které trojúhelníky jsou vykreslovány v kterém display listu (viz Obr. 4.1 – nepravidelná trojúhelníková síť rozdělená do pole 2x2 display listů. Tučná čára představuje dělení, bod X představuje bod, při jehož změně je nutné překreslit všechny čtyři display listy).

Řešení výše popsaných problémů si vyžádalo zavedení specializované datové struktury a několika dalších zásahů do stávajících struktur. Pro určení, které trojúhelníky jsou zobrazovány v kterém display listu, jsem vytvořil pole obsahující seznamy indexů trojúhelníků. Pole se seznamy má stejné rozměry jako pole obsahující display listy, a tedy příslušný seznam určuje trojúhelníky vykreslované příslušným display listem. K vytvoření seznamů dochází při průběhu první renderovací smyčky, která stejně musí všechny trojúhelníky projít (a vytvořit display listy, které se zobrazí na začátku aplikace). Dojde-li tedy k výše uvedeným změnám v síti (přidání bodu, prohození hran), je nutné tyto seznamy aktualizovat. V případě přidání bodu se jedná (s ohledem na použitou techniku triangulace sítě) o odstranění trojúhelníku, v němž přidávaný bod leží, a přidání nově vzniklých trojúhelníků do stejného seznamu. Jedná-li se o prohození hran mezi dvěma trojúhelníky, je nutné rozlišovat, zda oba náležejí do stejného seznamu či nikoli.



Obrázek 4.1: Dělení sítě pro d-listy

V prvním případě (náleží do stejného seznamu) dojde k odebrání obou trojúhelníků ze seznamu a přidání obou nově vzniklých do stejného.

Druhý případ (trojúhelníky náleží do různých seznamů) by bylo možno řešit několika způsoby. Tím nejjednodušším je přidání obou nově vzniklých trojúhelníků do jednoho (náhodně vybraného) ze seznamů. Tím se sice ztrácí geometrické rozdělení sítě do display listů, ale s ohledem na to, že této vlastnosti stejně nevyužíváme, to nevadí. Další v předchozím odstavci popsaným problémem je, jak určit, který display list je vlastně potřeba aktualizovat. Protože změny mohou probíhat i na úrovni vrcholů sítě (výškové změny - eroze terénu, zásahy nástroji do terénu), je nutné, aby vrcholy obsahovaly údaj o tom, do kterých display listů zasahují. Vzhledem k tomu, že vrcholy sítě mohou tvořit vrcholy více trojúhelníků, může změna jednoho vrcholu znamenat nutnost změnit až čtyři display listy, a tedy také jednotlivé vrcholy musí být připraveny obsáhnout souřadnice až čtyř display listů (souřadnice do pole display listů jsou uchovávána jako jedno celé číslo získané dle vzorce $(i*x)+j$, kde i a j jsou souřadnice display listu a x je rozměr pole display listů). Proběhnou-li výše uvedené změny na úrovni trojúhelníků (prohození hran, přidání bodu – trojúhelníků), je nutné provést příslušné změny také v těchto údajích.

Vertex Buffer Objects

Druhá varianta, k níž jsme dospěli, spočívá v použití Vertex Buffer Objects (VBO – viz kapitola 2.2). Zobrazovaná síť není rozdělena jako v případě použití display listů, ale je zobrazována v celku. Jsou používány tři VBO, které obsahují informace o souřadnicích vrcholů, normálách ve vrcholech a texturovacích koordinátech ve vrcholech. Jak již bylo zmíněno, celá síť je zobrazována najednou, a proto jakákoli změna v síti znamená vytvoření všech příslušných VBO znovu. Funkce vytvářející VBO (a také pole obsahující zobrazovaná data) je volána v případě, že na síti dojde k změnám, a to ve všech třech případech změn, jež mohou nastat, tj. první vytvoření VBO, změna obsahu VBO se zachováním jejich rozsahů a změna VBO v důsledku změny počtu prvků sítě. To znamená, že na počátku funkce je nutné nejprve rozhodnout na základě toho, zda se změnil počet prvků sítě, o tom, který z případů nastal. Jedná-li se o první vytvoření VBO, je nutné nejprve vyhradit paměť *(malloc)* pro dynamická pole, která budou obsahovat zobrazovaná data. V případě, kdy došlo k přidání či odebrání prvků ze sítě, je potřeba změnit velikost dynamických polí, do nichž budou následně ukládána data *(*realloc), a také smazat předchozí VBO *(glDeleteBuffers*). Pokud došlo pouze k změně parametrů prvků sítě (změna výšky vrcholů sítě), není nutné provádět v této části žádné změny (pole i VBO již existují a mají správné rozměry). Poté následuje rozsáhlý cyklus, jenž projde všechny trojúhelníky sítě a dle algoritmů popsaných dále naplní trojici polí daty určujícími souřadnice normál v jednotlivých vrcholech trojúhelníků, texturové souřadnice a souřadnice samotných vrcholů. Jednotlivé souřadnice jsou do příslušných polí ukládány postupně za sebou, a to jako datový typ GLfloat. Poté, co jsou příslušná pole naplněna údaji, dojde v závislosti na tom, o který z výše uvedených případů se jedná, buďto k vytvoření tří nových VBO a naplnění jejich obsahu daty (v případě prvního volání funkce nebo změny počtu prvků sítě), anebo k přepsání obsahů stávajících VBO novými daty (v případě změny parametrů vrcholů sítě), viz Obr. 4.2.

```
glGenBuffers(1,vbufs);//generuj jeden VBO index
glBindBuffer(GL_ARRAY_BUFFER, vbufs[0]);//nastav ho aktivní
glBufferData(GL_ARRAY_BUFFER, (9*gridgetTriangleCount())*sizeof(GLfloat),vert_array,
                GL_STATIC_DRAW_ARB); //odešli ho do paměti GPU
```

Obr 4.2: Ukázka vytvoření Vertex Buffer Object.

Připravené VBO jsou poté vykresleny v rendrovací smyčce. Vykreslení probíhá postupem popsaným již v teoretické části, tj. jsou nastaveny ukazatele na jednotlivé typy dat (normály, vrcholy atd.) a poté příkazem *glDrawArrays* vykresleny (viz Obr 4.3).

```
glEnableClientState(GL_NORMAL_ARRAY);// Zapne vertex arrays
glEnableClientState(GL_TEXTURE_COORD_ARRAY);// Zapne texture coord arrays
glEnableClientState(GL_VERTEX_ARRAY);// Zapne vertex arrays

glBindBuffer(GL_ARRAY_BUFFER, nbufs[0]);//nastav ho aktivní
glNormalPointer(GL_FLOAT, 0, (char *) NULL);//údaje v něm jsou normály
glBindBuffer(GL_ARRAY_BUFFER, tbufs[0]);
glTexCoordPointer(2, GL_FLOAT, 0, (char *) NULL);
glBindBuffer(GL_ARRAY_BUFFER, vbufs[0]);
glVertexPointer(3, GL_FLOAT, 0, (char *) NULL);

glDrawArrays(GL_TRIANGLES, 0, (3*grid->getTriangleCount()));//vykresli

glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

Obr 4.3: Ukázka vykreslení Vertex Buffer Objects

Aplikace textur

Úvod

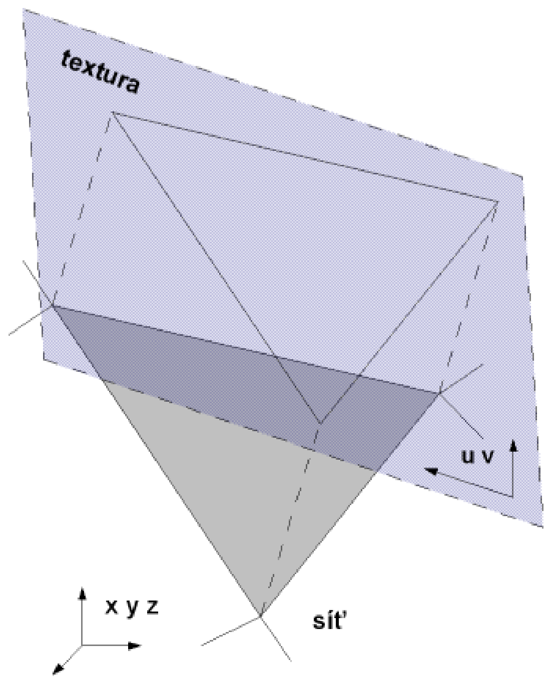
Pokud chceme, aby síť připomínala nějaký povrch, musíme prostor mezi stranami jednotlivých trojúhelníků „něčím“ vyplnit. Touto výplní může být barva interpolovaná mezi jeho vrcholy nebo také textura. Barva představuje velmi jednoduché řešení, které může v případě určitých povrchů vyhovovat. Textura oproti tomu umožňuje svou obrazovou informací nasimulovat na povrchu různé detaily. V našem případě jsem se rozhodl použít textury s ohledem na to, že v nepravidelné síti se mohou vyskytovat rozsáhlé plochy tvořené jedním trojúhelníkem, které by v případě použití barvy působily jednotvárně.

Popis

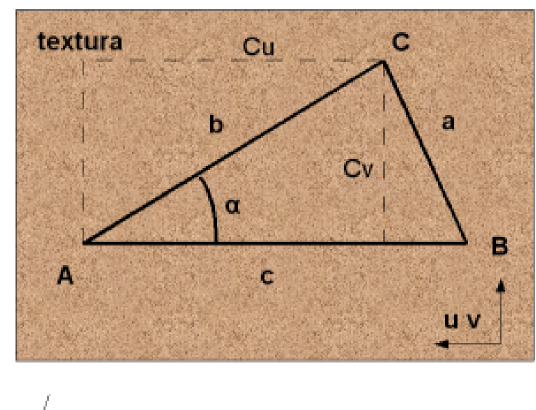
Aplikace textury na povrch s sebou nese určité komplikace, přičemž největší z nich je určení texturových souřadnic (koordinát). Výpočet texturových koordinát, které určují, jak se bude textura na daný trojúhelník mapovat, je značně problematický. Trojúhelník je tvořen třemi body, které jsou umístěny v prostoru, a tudíž jejich souřadnice mají tři hodnoty obvykle označované jako x,y,z. Texturové koordináty jsou obvykle tvořeny jednou, dvěma, nebo třemi souřadnicemi, přičemž pro naše účely (texturu aplikujeme na plochu představující povrch) použijeme takzvanou 2D texturu, jejíž texturové koordináty jsou označovány u,v. Z tohoto popisu je již vidět zřejmý nesouhlas. Souřadnice vrcholů trojúhelníka tvoří tři údaje, naproti tomu texturové koordináty tvoří dvojice údajů (textura je dvojrozměrná). Řešením je tedy ztotožnit rovinu textury s rovinou texturovaného trojúhelníka a provést výpočet texturových koordinát, přičemž je především podstatné zachovat vzdálenosti mezi vrcholy, aby nedošlo k „roztažení“ textur v některém směru (viz Obrázek 4.4).

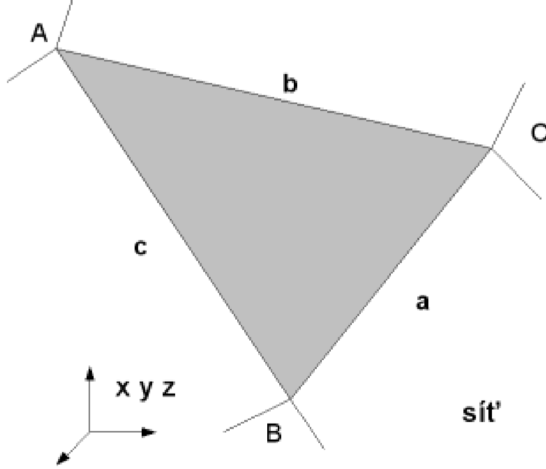
Samotný výpočet texturových souřadnic lze provést mnoha způsoby. V našem případě jsem zvolil postup využívající velikost stran texturovaného trojúhelníka, které získám ze souřadnic vrcholů. Z důvodu snadnější realizace je v průběhu výpočtu strana AB trojúhelníka ztotožněna s osou u texturových souřadnic, což usnadní výpočet samotných koordinát (souřadnice pro druhý vrchol je spočtena pouhým přičtením velikosti strany AB k souřadnicím pro první vrchol), ale může přinést jiné komplikace, jak bude zmíněno dále. Texturové souřadnice prvního vrcholu trojúhelníka pevně zvolím a souřadnice pro druhý bod získám výše uvedeným způsobem. K výpočtu souřadnic pro třetí vrchol lze poté využít znalostí velikostí hran trojúhelníka, s jejichž pomocí vypočítám úhel sevřený stranami *b* (CA) a *c* (BC) (kosinová věta), a z něho následně s využitím goniometrických funkcí souřadnice třetího bodu texturových koordinát (viz Obr 4.5 a Obr. 2.9, kde je v ukázce kódu uveden příklad otexturování jednoho trojúhelníka).

Jak vyplývá z výše uvedeného popisu, je aplikace textury provedena pro každý trojúhelník celé sítě samostatně a textura není po ploše sítě souvislá. Tento způsob lze akceptovat z toho důvodu, že síť představuje písek či jiný sypký materiál a použitá textura obsahuje pouze velmi drobné detaily. Při „pohybu materiálu“, tj. při změnách výšky bodů by však mohlo docházet u rozměrných trojúhelníků představujících souvislé plochy k nepříjemným jevům (textury na sousedních trojúhelnících by se mohly pohybovat různým směrem), což je dáno aplikací textury, kdy první bod trojúhelníku má pevnou souřadnici v textuře. K eliminaci tohoto nežádoucího jevu jsem provedl úpravu spočívající v přeorganizování vrcholů vykreslovaných trojúhelníků „stejně“ – tj. například tak, že bude první vrchol vždy nejvýše (největší hodnota souřadnice z). Lze totiž předpokládat, že nejvyšším vrcholem bude, ve velké většině případů, stále stejný vrchol. A tedy u sousedních trojúhelníků se bude, při změně jejich velikosti v důsledku eroze, textura „posouvat“ totožným směrem.



Obrázek 4.4: Správná aplikace textury na trojúhelník.



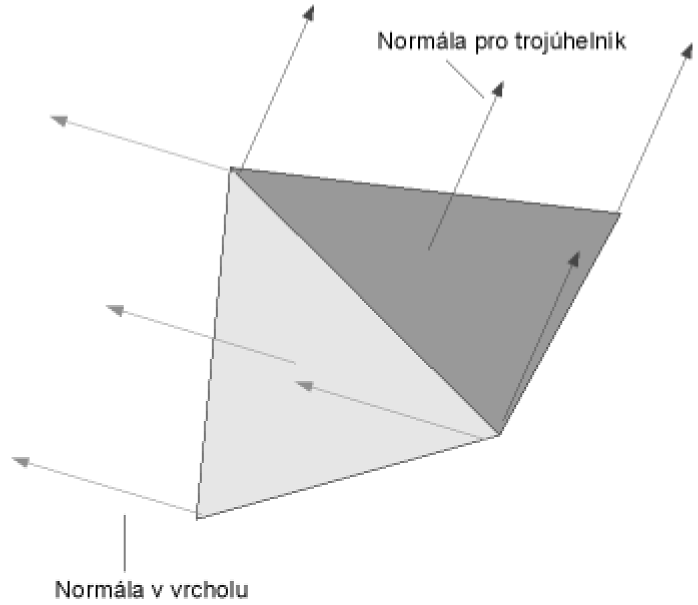


Obrázek 4.5: Výpočet texturových souřadnic vrcholů trojúhelníka.

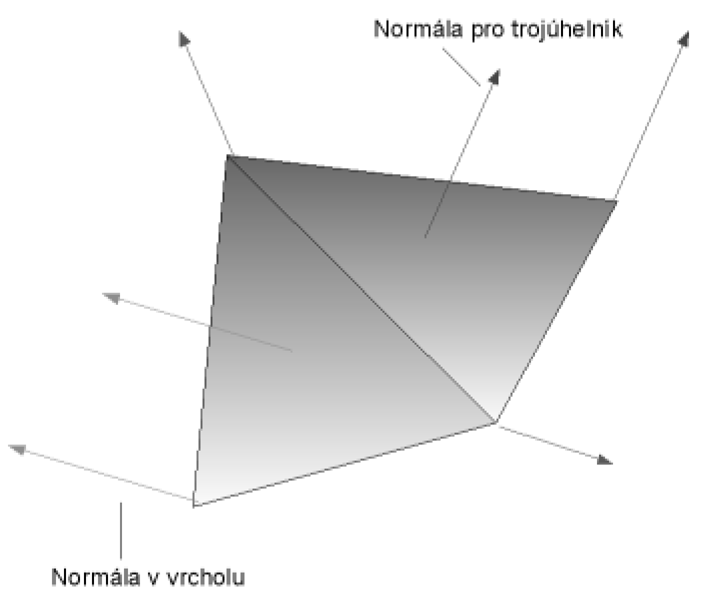
Osvětlení

Dalším prvkem, který má vliv na výsledný obraz sítě, je osvětlení, respektive hodnoty normál vrcholů sítě. Na tomto místě je vhodné upozornit na to, že nepravidelná trojúhelníková síť je vizualizována jako jednotlivé trojúhelníky. U každého z trojúhelníků je tedy nutné určit normály v jeho třech vrcholech. Pokud budou tyto normály získány jako normála trojúhelníku, jemuž dané vrcholy náležejí, půjde o takzvané konstantní stínování (viz Obr 4.6). Nevýhodou je to, že u zobrazované sítě jsou v tomto případě jasně zřetelné jednotlivé trojúhelníky. Jednotlivé vrcholy však tvoří vrchol několika trojúhelníků. Pokud tedy bude normála v jednotlivých vrcholech tvořena interpolací normál všech trojúhelníků, které v daném vrcholu sousedí, dostaneme tzv. Gouraudovo stínování (viz Obr 4.7). Výsledkem je vizuální vjem spojité sítě.

V programu jsou tedy nejprve spočteny normály v jednotlivých trojúhelnících (jako vektorový součin dvou stran trojúhelníka). A následně jsou pro jednotlivé vrcholy spočteny normály jako průměr z normál příslušných trojúhelníků. Výsledný vektor je poté nutno ještě znormalizovat (to znamená vydělit jednotlivé složky velikostí normály). Na takto získané normály je poté aplikováno standardní OpenGL světlo.



Obrázek 4.6 : normály ve vrcholech jsou stejné jako u příslušných trojúhelníků.



Obrázek 4.7: normály ve vrcholech jsou interpolovány z normál u trojúhelníků.

Zhodnocení:

V části zabývající se vizualizací sítě z hlediska možností, které nám dává OpenGL, byly popsány dva přístupy. První varianta využívající OpenGL display listů s rozdělením sítě je výhodná především pro rozsáhlé sítě, jako byly například pravidelné sítě použité v některých popisovaných pracích, a to především proto, že v případě změn je měněn pouze určitý z display listů, a také z toho důvodu že v případě pravidelné sítě lze velmi snadno určit hranice pro dělení. To je také největším nedostatkem při aplikaci této metody na nepravidelné sítě. Nutnost komplikovaně rozhodovat, které prvky sítě patří do kterého display listu (především, pokud v síti proběhnou nějaké změny), nejenom snižuje rychlost výsledného algoritmu, ale především snižuje přehlednost zdrojových kódů.

Druhé metoda využívající VBO (Vertex Buffer Objects) tímto nedostatkem netrpí, neboť v případě změny v síti dochází k jejímu celkovému překreslení. Tato metoda tedy nikterak výrazně nezasahuje do ostatních částí algoritmu a nevyžaduje žádné rozsáhlé pomocné datové struktury. Nedostatkem pak může být právě to, že při změně v síti dochází k vytvoření nových VBO, což v případě, že síť dosáhne větších rozměrů může znamenat značnou výpočetní zátěž. Dalším méně zřejmým nedostatkem je to, že na rozdíl od display listů VBO je relativně „pokročilá“ technika, a tedy výsledný algoritmus využívající VBO bude „citlivý“ na schopnosti grafické karty, na níž bude pracovat.

Po zvážení všech zmíněných přínosů a nedostatků jsme se rozhodli, že ve výsledném algoritmu budou použity Vertex Buffer Objects. A to ze dvou důvodů. Prvním bylo zjednodušení výsledného zdrojového kódu. Vzhledem k tomu, že na výsledném programu se podílí trojice lidí a je případně plánován další rozvoj, je příliš složitý kód zasahující do mnoha částí programu (jak tomu bylo v případě použití display listů) nevhodný. Druhým důvodem byl předpoklad, že zpracovávaná síť nebude příliš rozsáhlá, což by měla být hlavní výhoda nepravidelné sítě.

Dále byl zmíněn postup texturování sítě, který má několik negativ. Tím prvním je to, že textura na síti není souvislá. Při pohledu z velké a střední vzdálenosti to není díky vlastnostem použité textury rozpoznatelné, ale při větším přiblížení je již tento nedostatek viditelný. Možným řešením by bylo upravit výše uvedený postup pro souvislé texturování alespoň v rámci konkrétního display listu či VBO, což by ale znamenalo nejen nutnost uchovávat u každého vrcholu texturovací souřadnice z důvodu návaznosti, ale především zvolit vhodný postup zpracovávání trojúhelníků.

Druhý problém navazuje na výše uvedený. Jde o to, že v [ONO03] je uvedena velmi zajímavá technika, která s využitím posunu textur simuluje přesýpání písku po povrchu při změně výšky bodů sítě. Tuto techniku je samozřejmě možné realizovat při použití výše uvedeného postupu, ale výsledný efekt není příliš významný a navíc použitý postup texturování trojúhelníků, kdy v podstatě dochází k otočení souřadného systému uv (textury) oproti systému xyz (vrcholy) vede ke komplikacím při výpočtu posunů textur. Tento nedostatek lze pravděpodobně vyřešit pouze při vyřešení souvislosti textury na síti.

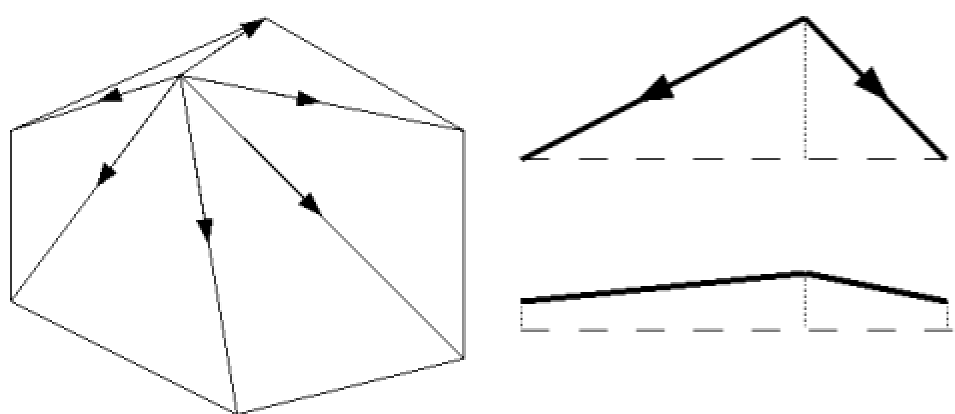
4.3 Eroze

Úvod

Druhým řešeným problémem byla simulace působení gravitace na síť představující sypký materiál. Simulace vychází z principů uvedených v článcích [BEN06] a [ONO03]. Simulace neprobíhá na základě fyzikálně přesných modelů, a to především z důvodu důrazu na interaktivnost celého procesu.

Popis

I přesto, že články zmíněné v úvodu využívaly pravidelných sítí, je základní princip simulace eroze způsobené gravitací totožný. Tj. pro každý vrchol sítě (označím jej jako „středový") prohledáme jeho nejbližší sousední vrcholy (vrcholy s „přímou" viditelností přes hranu) a zjistíme, zda není některý z těchto vrcholů níže o předem daný rozdíl. Tento rozdíl je dán úhlem mezi vrcholy (je měřen mezi přímkou procházející výše položeným vrcholem a spojnicí vrcholů), jenž je rozdílný v závislosti na vlastnostech simulovaného materiálu (pro suchý písek je tento úhel roven přibližně 30°). Po projití všech sousedních vrcholů proběhne druhá fáze procesu, kdy je ze „středového" vrcholu odebráno určité množství materiálu (tj. snížena výška), které je rozdistribuováno (přidáno) do těch sousedních vrcholů, u nichž bylo v první fázi splněno úhlové kritérium (tj. byly níže než „středový" vrchol). Tento proces by se opakoval pro všechny vrcholy sítě. (Obr. 4.8)

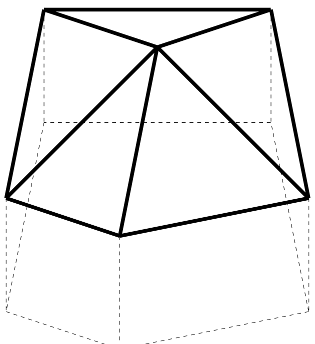


Obrázek 4.8: Naznačení průběhu eroze.

Konkrétní realizace si vyžádala několik drobných úprav výše uvedeného procesu. Především proces eroze neprochází všechny vrcholy sítě, ale pouze ty vrcholy, jež jsou uvedeny ve zvláštním seznamu. Vrcholy jsou do tohoto seznamu přidávány jednak při realizaci deformací sítě prostřednictvím nástrojů (vrcholy, jež jsou nástrojem zasaženy a u nichž dochází ke změně výšky) a také samotným procesem eroze, a to ty vrcholy, u nichž v důsledku eroze došlo ke změně výšky. Ze seznamu je nutné také vrcholy odebírat, a to v okamžiku, kdy u nich opakovaně nedošlo v důsledku eroze ke změnám (to, že podmínkou je, že ke změnám nedojde opakovaně, je z důvodu zamezení chyb, kdy pro některé konfigurace sítě by mohly některé vrcholy „zamrznout").

Metody

Jak jsem již napsal, v druhé fázi algoritmu dochází k změně výšky vrcholů. Jednomu vrcholu je výška snížena a některým jeho sousedům naopak zvýšena. Podstatným údajem, z hlediska věrohodnosti průběhu eroze je právě poměr těchto hodnot. Z fyzikálního hlediska je podstatnou vlastností eroze zachovávání objemu a naším cílem by tedy mělo být určení daného poměru výšek tak, aby „objem sítě" před a po erozi zůstal stejný. Trojúhelníková síť samozřejmě nemá objem, ale lze si ji představit jako jednu ze „stěn" tělesa, které vznikne pokud z okrajů sítě spustíme svislé stěny a ty uzavřeme v určité vzdálenosti rovinou (obr 4.9).



Obr. 4.9: Těleso, silně vytažená část představuje trojúhelníkovou síť.

Podobným způsobem si lze představit vznik těles z jednotlivých trojúhelníků tvořících síť. Pokud se vrátíme k našemu problému spojenému s poměrem výšek vrcholů sítě, zjistíme, že změna výšky jednoho vrcholu sítě změní objem všech těles vzniklých z trojúhelníků obsahujících tento vrchol. Chceme-li tedy změnit výšku u dvojice vrcholů v síti tak, aby zůstal zachován objem, je nutné, aby poměr výšek (výšky odebrané jednomu vrcholu, ku výšce přidané druhému) byl přímo úměrný poměru objemů změnou zasažených těles.
V případě algoritmu eroze dochází k snížení jednoho bodu a zvýšení několika jeho sousedů. Ve výsledném vzorci, který řídí poměr výšek, se tedy vyskytuje jednak objem tělesa tvořeného trojúhelníky, které obsahují vrchol, z něhož bude výška odebrána, a jednak součet objemů těles, jež tvoří trojúhelníky obsahující vrcholy, jejichž výška bude navýšena (součet je skutečně součtem objemů těles, a pokud jsou některé vrcholy sousedy, je objem započten v podstatě dvakrát). Vzorec pak vypadá následovně:

$$Z = Za * Va / V$$

kde *Za* je výška, kterou odeberu vrcholu, *Z* je výška, kterou přidám vrcholům sousedním, *Va* je objem tělesa náležejícího k vrcholu, z něhož odebírám a *V* je součet objemů těles ostatních vrcholů.

K realizaci algoritmu dle uvedeného vzorce potřebujeme znát postup, jak vypočítat objem tělesa, jehož dvě stěny tvoří trojúhelníky, které nemusejí být rovnoběžné, a jedna z těchto stěn je kolmá na zbylé obdélníkové stěny. Objem takto vypadajícího hranolu se spočte jako součin plochy kolmé podstavy hranolu (neboli plochy řezu kolmého na obdélníkové stěny) a průměru z výšek (hran hranolu kolmých k ploše řezu) hranolu.

Výsledný algoritmus tedy vypadá následovně: Jsou procházeny jednotlivé vrcholy ze seznamu vrcholů a je zjišťováno, zda je úhel mezi zpracovávaným vrcholem („dárce") a jeho sousedy ležícími níže („příjemci") větší než daný úhel. V kladném případě je spočten objem hranolů, jejichž vrchol tvoří daný soused, a tato hodnota je pro všechny sousedy zpracovávaného vrcholu sčítána. Po projití všech sousedů je spočten i objem hranolů

zpracovávaného vrcholu. V druhé fázi dojde k navýšení „příjemců“ o určitou „vhodně malou“ hodnotu a o tuto hodnotu vynásobenou poměrem objemů, je snížen vrchol „dárce“ (jedná se o inverzi výše popsaného vzorce). Konstanta, o jejíž hodnotu jsou navýšeny vrcholy, určuje rychlost průběhu eroze. Je-li ovšem tato hodnota příliš velká, dojde k „rozkmitání“ sítě a algoritmus eroze nikdy nedojde do ustáleného stavu, anebo budou na síti v průběhu eroze velmi zřetelné změny výšek jednotlivých vrcholů.

Takto realizovaná metoda eroze dosahuje celkem dobrých výsledků (jak bude uvedeno v dalších kapitolách), ale již z jejího popisu je zřejmý její největší nedostatek, jímž je výpočetní náročnost. Ta je způsobena především výpočtem velkého počtu objemů hranolů pro každý zpracovávaný vrchol. Výpočet objemů je nutný k určení poměru hodnot odebíraných, resp. přidávaných výšek vrcholům. Pokud budeme vycházet z předpokladu, že „objemy trojúhelníků“ (hranolů) jsou podobné, lze přejít k technikám, které nevyžadují výpočet objemu. Použijeme tedy určitých aproximací a metody „vyladíme“ tak, aby se pokud možno na většině sítí chovaly, jako by zachovávaly objem.

Těchto aproximací lze nalézt celou řadu, přičemž základní algoritmus je stejný jako v případě výše popsané metody využívající objem. Při procházení jednotlivých vrcholů ze seznamu (vrcholů určených k zpracování erozí), je zjišťováno, zda je úhel mezi právě zpracovávaným vrcholem a jeho níže položeným sousedem větší než daná hodnota. V kladném případě jsou spočteny určité parametry použité v další fázi (vzdálenost mezi body, atd.). V druhé fázi se pak zpracovávanému vrcholu odebere určitá výška a příslušným sousedům je výška přidána.

První ze zkoušených aproximací spočívá v tom, že hodnota výšky odebrané výše položenému vrcholu byla vydělena konstantou a výsledný podíl byl přičten níže položeným sousedům. Zkoušeno bylo několik hodnot dělitele : 1.5, 2, 3, 4 a byly porovnávány rozdíly v objemech před a po erozi. Porovnávání probíhalo prostřednictvím modelovacího nástroje 3d Studio MAX, v němž byly vytvořeny malé úseky povrchu s několika vrcholy, jejichž výšky byly měněny, jako by s nimi pracoval zkoušený algoritmus. Poté byly pomocí nástroje zjišťující objem porovnávány rozdíly v objemech sítí. Výsledky těchto pokusů (stejně jako stejné pokusy prováděné u další uvedené metody) ukázaly, že rozdíly v objemech před a po aplikaci eroze na síti nejsou příliš velké a také že velmi závisí na rozložení vrcholů v síti.

Druhá z aproximací vychází z myšlenky nerovnoměrného rozdělení distribuovaného materiálu, tj. aby těm vrcholům, které leží výrazně níže, přibyla větší část než těm výše položeným. Je tedy zjišťován rozdíl výšky mezi vrcholem, z něhož bude probíhat distribuce, a vrcholy, do nichž bude materiál distribuován. Tato hodnota je uložena jednak zvlášť pro každý z vrcholů a jednak v podobě součtu. Při distribuci materiálu (výšky) je hodnota odebraná výše položenému vrcholu vynásobena podílem celkového rozdílu výšek a výšky konkrétního vrcholu. Zapsáno vzorcem vše vypadá následovně:

zB = zB +*deltaM*(differenceToI[j]/sum)*

kde *zB* je z-tová souřadnice vrcholu do něhož přidávám, *deltaM* je výška odebraná vrcholu, *differenceToI[j]* je rozdíl výšek mezi právě zpracovávaným vrcholem a vrcholem, z něhož je výška odebírána, a *sum* je součet rozdílů výšek. Algoritmus je opět aplikován pouze na ty vrcholy, které splňují „úhlové“ kritérium. Výsledkem algoritmu by mělo být přirozenější chování eroze, kdy níže položené vrcholy jsou navýšeny více než ty, jejichž rozdíl není tak výrazný.

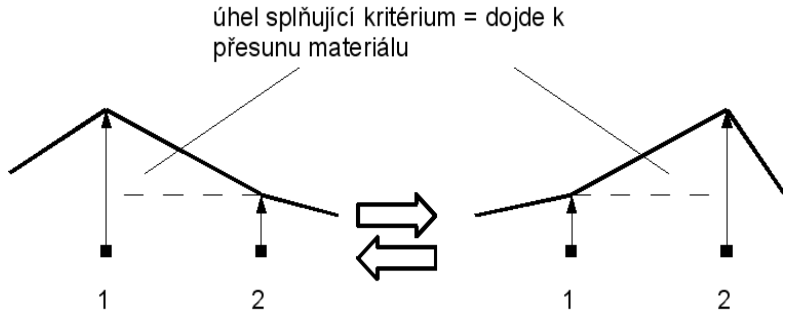
Třetí z metod pracuje na mírně odlišném principu. V první fázi algoritmu je vyhledán takový soused, který má největší rozdíl výšky oproti zpracovávanému vrcholu (a splňuje

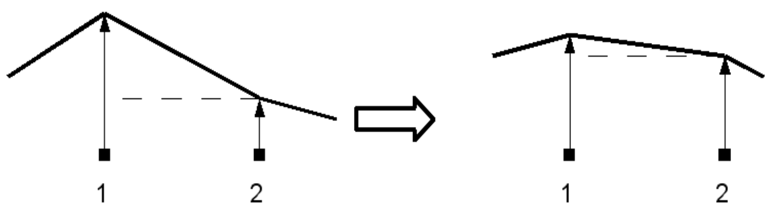
kritérium s úhlem). V druhé fázi je pak hodnota výšky (daná konstantou) odebraná zpracovávanému vrcholu přičtena sousedovi nalezenému v první fázi. Toto řešení přináší výhodu především v rychlejším dosažení ustáleného stavu.

Úpravy metod

Po otestování v programu byly zjištěny určité nedostatky, které se určitou měrou objevovaly u všech uvedených metod. Odstranění těchto nedostatků vedlo k vzniku dvojice úprav, které byly aplikovány na všechny zkoušené metody.

První úprava modifikuje hodnotu konstanty určující, jaké množství materiálu (výška), bude odebráno, respektive přidáno vrcholům. U druhé skupiny metod vycházejících z přesunů výšek bez ohledu na objemy docházelo k „rozkmitání" dvojic vrcholů. Tento jev byl způsoben nepravidelností sítě, tj. tím, že některé dvojice vrcholů mohou být mnohem blíže u sebe než ostatní. V takovém případě mohlo dojít k tomu, že sousední níže položený vrchol byl navýšen natolik, že se dostal výrazně výše, než jeho soused, z něhož „získal materiál", což vedlo k opakování celého jevu (viz Obrázek 4.10). Úprava tedy spočívá v tom, že v první fázi algoritmu je nalezena nejmenší vzdálenost mezi sousedy (mezi nimiž bude probíhat přesun) a v druhé fázi je „konstanta" určující přenesený materiál (výšku) vynásobena touto vzdáleností. Tato úprava zabraňuje téměř všem případům popisovaného jevu a vizuálně mírně zlepšuje průběh eroze. Aby se zabránilo i těm případům, které tato úprava nepodchytí, je v metodách jednoduchá podmínka, která kontroluje to, zda se vrchol z něhož odebírám nedostane níže než některý z jeho sousedů, do nichž přidávám. Pokud by k tomu mělo dojít, dojde k úpravě velikosti odebírané výšky tak, aby se oba zmíněné vrcholy dostaly do roviny.





Obrázek 4.10 Přenos materiálu mezi vrcholy.

Druhá úprava má opět souvislost s konstantou určující množství přenášeného materiálu. Jedná se o techniku, která v rámci jednoho volání metody realizující erozi v hlavní smyčce programu volá tuto metodu vícekrát, přičemž hodnotu konstanty (či spíše proměnné) určující přenos materiálu vydělí počtem volání metody. Přínosem této metody je vizuálně lepší průběh eroze (v závislosti na tom, kolikrát dojde k volání metody realizující erozi). Pravdou ovšem je, že tato technika vlastně „upřednostní" průběh eroze na úkor ostatních částí programu a v případě nutnosti zpracovávat větší počet bodů vede k horší celkové odezvě programu. Řešením by mohlo být svázání počtu opakování například s fps (počtem snímků za sekundu).

Zjemňování sítě

V rámci metod realizujících gravitační erozi je prováděno také zjemňování trojúhelníkové sítě. K zjemňování sítě slouží metoda, jejímiž parametry je vkládaný bod, která zajistí vytvoření příslušných trojúhelníků v síti (podrobnosti viz [Pur07]). Metrikou pro vložení bodů je vzdálenost mezi dvojicí erozí zpracovávaných vrcholů, mezi nimiž má dojít k přenosu materiálu. Překročí-li tato vzdálenost určenou hodnotu, dojde k vložení nových vrcholů do trojúhelníků, jejichž hranu tvoří zmiňovaná dvojice vrcholů. Nový vrchol je do trojúhelníku vložen na souřadnice získané lineární interpolací souřadnic jednoho z vrcholů trojúhelníku se souřadnicemi bodu, získanými opět lineární interpolací zbylé dvojice vrcholů.

Zhodnocení:

Každá z popisovaných metod má svá pozitiva a negativa. Je tedy nutno zvážit jaké vlastnosti výsledné simulace požadujeme, a dle toho zvolit příslušné metody. Je také nutné podotknout, že průběh i výsledek eroze dle jednotlivých metod je velmi závislý na celé řadě jejich parametrů. Tyto parametry a jejich vliv na průběh eroze budou zmíněny v dalších kapitolách. Celkově lze říci, že nejlepších výsledků dosahuje metoda založená na zachování objemů. Ostatní uvedené metody se jejím výsledkům více či méně blíží, v závislosti na konkrétních podmínkách (parametry, uspořádání sítě). Obecně platí, že k dosažení lepších výsledků dojde při takovém nastavení parametrů, které znamená větší výpočetní náročnost.

5. Realizace

5.1 Úvod

V předchozí kapitole byly popsány algoritmy, zkoušené pro řešení jednotlivých problémů, především z hlediska teoretického návrhu a vlastností. V této kapitole pak bude popsána jejich implementace v rámci samotného programu.

5.2 Vizualizace sítě

I přesto, že v zhodnocení v předchozích kapitolách byly popsány zápory jednotlivých způsobů řešení a také to, kterým metodám jsem v výsledném programu dal přednost, byly všechny popisované metody řešení postupně realizovány a vyzkoušeny. Bylo to dáno jednak potřebou vyzkoušet vlastnosti jednotlivých metod při skutečné implementaci a jednak také tím, že pokud se některý z postupů neosvědčil, bylo hledáno a zkoušeno jiné řešení.

Inicializace

Před tím, než popíšu, jak byly realizovány stěžejní části popisované v předchozích kapitolách, zmíním se krátce o těch částech programu, které zajišťují různé počáteční inicializace. Jedná se především o načtení textur a nastavení okna, do něhož bude probíhat vykreslování výsledků simulace. V teoretické části byla představena knihovna SDL, a jak jsem uvedl, vytvoření okna aplikace je záležitost, která není jednoduchá a je závislá na použitém operačním systému. Právě použitím knihovny SDL a jejích funkcí lze tento problém elegantně odstranit.

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_NOPARACHUTE);

SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 16);

SDL_SetVideoMode(OUTPUT_WIDTH, OUTPUT_HEIGHT, 0, SDL_OPENGL);
```

Obr 5.1: Ukázka kódu zajišťující vytvoření okna v SDL.

V uvedené ukázce kódu (Obr 5.1) můžeme vidět nejprve příkaz zajišťující inicializaci samotného SDL *(SDL_Init()),* který musí být zavolán jako první před tím, než začneme využívat jakékoli jiné funkce SDL. Poté následují některá základní nastavení nově vytvořeného prostředí a posledním příkazem v ukázce kódu je příkaz zajišťující vytvoření samotného okna. Tato sekvence příkazů bude (s drobnými odlišnostmi) stejně fungovat pod jakýmkoli podporovaným operačním systémem a zajistí nám vytvoření okna, do něhož lze kreslit prostředky OpenGL.

Po vytvoření nového okna přijde na řadu další činnost, při níž je opět využívána knihovna SDL, respektive knihovna SDL_image. Načtení textury je další činnost, kterou při využití rozhraní OpenGL musí programátor řešit sám, respektive se spolehnout na některé rozšiřující knihovny, které mu tuto činnost ulehčí. Jak jsem se již zmínil v teoretické kapitole, je knihovna SDL sama o sobě schopna načítat pouze formát .bmp . I přesto, že by

v našem případě mohla tato vlastnost postačovat, rozhodl jsem se nakonec z důvodu zachování větší variability pro knihovnu SDL_image, která podporuje větší výběr formátů. Načtení textury je v případě použití SDL vyřešené jediným příkazem *IMG_load,* který načte obrazový soubor, jehož cesta je parametrem příkazu a vrátí jej v podobě datového typu *SDL_surface*. Z tohoto datového typu by bylo možné několika málo příkazy vytvořit OpenGL texturu. Důvodem, proč to není tak úplně jednoduché, je způsob uspořádání textury v případě SDL a v případě OpenGL. OpenGL totiž vyžaduje, aby se první pixel nacházel v levém dolním rohu, zatímco SDL má první pixel v levém horním. Je tedy nutné pixely přeházet (v našem případě by to nutné nebylo vzhledem k vlastnostem textury, ale vzhledem k tomu, že tuto operaci provádíme pouze na začátku, není to na škodu). Nyní máme texturu „správně“ a je potřeba ji z datové struktury SDL převést na texturu OpenGL. V teoretické části byly zmíněny některé potíže s vizualizací otexturovaných objektů a postup, jak je řešit pomocí tzv. mipmapingu. A právě vytvoření mipmap spadá do této části. K vytvoření mipmap použijeme funkci obsaženou v extension GL SGIS generate mipmap a její použití stejně jako celý postup vytvoření textury ukazuje následující část kódu (Obr 5.2).

```
SDL_Surface *surface;// Obrázek
surface = load_texture(filename);// Načtení obrázku
if(surface == NULL) return 0;

GLuint texture;// OpenGL textura
glGenTextures(1, &texture);// Generování jedné textury
glBindTexture(GL_TEXTURE_2D, texture);// Nastavení textury

if (mipmaps)// Mipmapovaná textura
{
	// Nastavení požadovaných filtrů
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, mag_filter);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, min_filter);
	glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
	glTexImage2D(GL_TEXTURE_2D, 0, 3, surface->w, surface->h, 0, GL_RGB,
			GL_UNSIGNED_BYTE, surface->pixels);

}
else// Obyčejná textura
{
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexImage2D(GL_TEXTURE_2D, 0, 3, surface->w, surface->h, 0, GL_RGB,
			GL_UNSIGNED_BYTE, surface->pixels);
}
glBindTexture(GL_TEXTURE_2D, 0);
SDL_FreeSurface(surface);// Smazání SDL_Surface
surface = NULL;// Nastavení ukazatele na NULL
return texture;// Vrátí texturu
```

Obr 5.2: Ukázka kódu vytvoření textury.

Podstatnými parametry z hlediska vizualizace jsou typy použitých filtrů. Použitý filtr se nastavuje v funkci *glTexParametr* (viz ukázka kódu). Funkce má tři parametry, přičemž prvním určujeme, že budeme pracovat s 2D texturou (konstanta *GL_TEXTURE_2D*), druhým pak, jaký z parametrů textury budeme měnit (*GL_TEXTURE_MAG_FILTER* filtr použitý při zvětšování textury, *GL_TEXTURE_MIN_FILTER* filtr použitý při zmenšování textury) a posledním je samotný parametr. V našem případě jsem použil pro zvětšování

(*GL_TEXTURE_MAG_FILTER*) filtr *GL_LINEAR* (lineární interpolace) a pro zmenšování (*GL_TEXTURE_MIN_FILTER*) filtr *GL_NEAREST_MIPMAP_LINEAR* (lineární filtrování mezi mipmapami, v rámci mipmapy je vybírán nejbližší pixel).

Posledním využívaným prostředkem knihovny SDL je obsluha událostí. Událostí v tomto smyslu existuje celá řada. V našem případě událostí míníme především stisk kláves na klávesnici či pohyb myši a stisk tlačítek na myši. Obsloužení událostí pak znamená provedení nějakého kódu v závislosti na typu vzniklé události. Obsloužení události SDL pak probíhá prostřednictvím funkce *SDL_PollEvent,* která ve svém parametru vrací strukturu specifikující typ události. Následující ukázka kódu přibližuje postup při obsluze událostí (Obr 5.3).

```
SDL_Event event; /* Event structure */
.
.
.
/* Check for events */
while(SDL_PollEvent(&event)){  /* Loop until there are no events left on the queue */
  switch(event.type){  /* Process the appropiate event type */
case SDL_KEYDOWN:  /* Handle a KEYDOWN event */
     printf("Oh! Key press\n");
     break;
   case SDL_MOUSEMOTION:
     .
     .
     .
   default: /* Report an unhandled event */
     printf("I don't know what this event is!\n");
  }
}
```

Obr 5.3: Ukázka kódu obsluhujícího zachycené události (převzato z [SDL]).

Základem je cyklus *while,* v němž kontrolujeme návratovou hodnotu funkce *SDL_PollEvent,* která určuje, zda existují ještě nějaké nezpracované události. Poté se přepínačem určí typ události *(event.type)* a na jeho základě je provedena příslušná reakce. V našem případě bychom si nevystačili s pouhým rozhodnutím typu události, tj. zda se jedná o stisk klávesy či pohyb myši, ale potřebujeme také určit typ události podrobněji (jaká klávesa atd.). K tomuto rozhodnutí nám poslouží další prvky struktury *SDL_Event*, které umožňují podrobněji určit, k jaké události došlo (Obr 5.4).

Pro nás jsou důležité zejména prvky *button, key* a *motion*. Opět se jedná o struktury nesoucí značná množství informací. *Button* určuje stisk tlačítka na myši, podstatným prvkem struktury je prvek označený *button,* který určuje, které tlačítko bylo stisknuto. Prvek *key* určuje stisknutou klávesu, přičemž se opět jedná o strukturu, jejíž podstatným prvkem je *keysim.sim* (jedná se vlastně opět o strukturu – zapsal jsem tedy přímo notaci přístupu k použité proměnné), který obsahuje symbolickou konstantu určující stisklou klávesu. Poslední zmiňovaný prvek *motion* pak obsahuje údaje o souřadnicích (normálních a relativních) kurzoru. Bližší informace včetně symbolických konstant lze nalézt v [SDL].

```
typedef union{
  Uint8 type;
  SDL_ActiveEvent active;
  SDL_KeyboardEvent key;
  SDL_MouseMotionEvent motion;
  SDL_MouseButtonEvent button;
  SDL_JoyAxisEvent jaxis;
  SDL_JoyBallEvent jball;
  SDL_JoyHatEvent jhat;
  SDL_JoyButtonEvent jbutton;
  SDL_ResizeEvent resize;
  SDL_ExposeEvent expose;
  SDL_QuitEvent quit;
  SDL_UserEvent user;
  SDL_SywWMEvent syswm;
} SDL_Event;
```

Obr 5.4: Definice struktury SDL_Event (převzato z [SDL]).

Podstatným prvkem výsledné vizualizace sítě bylo také osvětlení. V programu bylo použito standardní OpenGL světlo. OpenGL si v takovém případě řídí veškeré výpočty na grafické kartě samo a naší starostí je jednak to, aby zobrazovaný model obsahoval správné normály, a jednak nastavení parametrů světla. Normály jsou probrány v jiné části textu. Nastavení světla je v případě OpenGL vcelku jednoduchou záležitostí. Nejprve je vhodné nastavit světlu barvy jednotlivých složek (viz teoretická část) a povolit jeho používání (Obr 5.5).

```
glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);// Nastavení okolního světla
glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);// Nastavení přímého světla
glEnable(GL_LIGHT1);// Zapne světlo
```

Obr 5.5: Ukázka kódu: povolení světla a nastavení barev.

Poté pouze stačí povolit OpenGL, aby požívalo nastavená světla (*glEnable(GL_LIGHTING)*) a nastavit pozici světla. V našem případě jsem použil jedno statické světlo nacházející se dvě jednotky nad počátkem scény ve směru z (to jest nad povrchem). Nastavení pozice světla, má-li být statické, je nutné provádět po provedení všech transformací, které mění pozici kamery.

D-Listy

První realizované řešení využívalo display listů (d-listů). Vytvoření display listů probíhalo v dvojici funkcí. První funkce byla volána na začátku činnosti programu a zajišťovala jednak vytvoření display listu, který byl zobrazován před provedením první změny v síti, a pak také určení, který trojúhelník bude následně zobrazován kterým d-listem. Tato informace byla uložena v dvojí podobě. Jednak každý trojúhelník znal „číslo“ d-listu, v němž byl zobrazován, a jednak existoval seznam všech trojúhelníků, které se zobrazují v příslušném d-listu. Informace o tom, do kterého d-listu náleží příslušný trojúhelník, nesly i vrcholy trojúhelníků, tj. vrcholy sítě. Tato informace byla podstatná především z hlediska rozhodování, který z d-listů je nutné přepočítat. Druhá zmiňovaná funkce pak měla na starosti opětovné vytvoření příslušného d-listu, pokud došlo k nějakým změnám. K indikaci změn

sloužilo pole proměnných typu *bool*, které mělo stejné rozměry jako pole d-listů. Funkce realizující „překreslení“ d-listů využívala výše zmíněné seznamy trojúhelníků a při vytváření d-listů procházela tedy pouze trojúhelníky náležející do daného d-listu. V rámci těchto funkcí také probíhaly výpočty texturových souřadnic a normál pro jednotlivé vrcholy. Tato realizace se ukázala jako celkem rychlá a funkční. Nedostatky tohoto řešení se projevily především v okamžiku, kdy bylo nutné provádět zásahy do struktury sítě, jako například vkládání a odebírání bodů. Pokud totiž dojde k vložení trojúhelníku do sítě (například při vložení bodu do sítě vzniknou nové trojúhelníky), je nutné určit, v jakém d-listu se bude zobrazovat, přidat jej do příslušného seznamu a nastavit mu číslo d-listu. Podobná procedura následuje při odebírání či prohození hran u sousedních trojúhelníků. Ošetřit správně tyto činnosti v programu je značnou komplikací, nehledě na to, že těmto částem jsem se nevěnoval já, ale kolegové. K těmto operacím navíc dochází relativně často a tato údržba rozdělení trojúhelníků do d-listů je tudíž i značně výpočetně náročná.

Vertex Buffer Object

Druhým přístupem k vizualizaci nepravidelné trojúhelníkové sítě bylo použití Vertex Buffer Objects. Použití VBO se jevilo jako schůdnější metoda především z toho důvodu, že jsme se rozhodli, že síť bude zobrazována v celku, a tudíž odpadly různé komplikace s dělením. V předchozí části byl popsán základní postup, který jsem použil při zobrazování sítě prostřednictvím VBO, k tomu přibyl ještě algoritmus výpočtu normál ve vrcholech interpolací z normál v trojúhelnících. Při použití VBO je nutné nejprve VBO vytvořit a poté je v vykreslovací smyčce programu, která je pravidelně volána, zobrazovat. Jak bylo již zmíněno v teoretické části, vykreslení VBO je, na rozdíl od jejich vytvoření, operací velmi rychlou. Naší snahou by tedy mělo být, aby vytváření VBO probíhalo pouze v tehdy, je-li to nezbytné, tj. v případě, kdy dojde k nějakým změnám v síti. K rozpoznání toho případu slouží metoda *rebuild_arrays()*, která kontroluje proměnnou typu *bool* nazvanou *rebuild*, jejíž hodnota *true* určuje, že v síti došlo k změně a je nutné aktualizovat VBO (podobně jako v případě d-listů existovalo pole *bool* proměnných určujících změny v síti). Samotné vytvoření VBO realizuje metoda *build_array()*. V této metodě probíhá jak vytvoření polí obsahujících zobrazovaná data, tak i naplnění vytvořených VBO. Před tím je ale nutné spočítat normály ve všech trojúhelnících v síti. Před započetím vytváření polí s daty je tedy zavolána příslušná funkce, která projde všechny trojúhelníky v síti a popsaným postupem vypočte jejich normály (tyto normály nejsou normalizované). Poté jsou vytvořena příslušná pole, respektive je změněna jejich velikost v případě, že se změnil počet prvků sítě. Pokud došlo pouze k změně parametrů prvků sítě, jsou pouze přepsány obsahy dříve použitých polí. V takovém případě jsou použity také stávající VBO, jejichž obsah je pouze přepsán. Byla-li vytvářena nová pole, tj. jedná se o první běh programu, a nebo byl-li měněn počet prvků polí, je nutné vytvořit nové VBO, přičemž je důležité smazat případné stávající VBO. Pole zobrazovaná prostřednictví VBO obsahují tři typy informací. Prvním údajem jsou souřadnice normály daného vrcholu. Tu získáme voláním funkce *intepolace_nirmal()*, která projde pro daný vrchol seznam trojúhelníků a interpoluje jejich normály. Výsledná normála je ještě poté normalizována, tj. jednotlivé složky jsou vyděleny velikostí normály získané interpolací tak, aby výsledkem byl jednotkový vektor. Druhým údajem ukládaným do pole jsou texturové souřadnice, které jsou spočteny ze souřadnic vrcholů trojúhelníka výše popsaným způsobem. Třetím údajem jsou samotné souřadnice vrcholů sítě získané z objektu třídy *Grid*.
Vytvořené VBO jsou následně zobrazeny postupem, který byl již popsán.

5.3 Eroze

Eroze je realizována jako samostatná třída *(ErosionSystem),* jejímiž podstatnými prvky je seznam vrcholů určených k zpracování a metody, které realizují erozi samotnou. Seznam vrcholů určených k zpracování je tvořen datovým typem *vector* a metodou zajišťující přidání vrcholu sítě do seznamu. Datový typ *vector* je v podstatě spojový seznam umožňující libovolné přidávání a odebírání prvků. Metoda *AddPoint* sloužící k přidávání vrcholu sítě do seznamu zjistí, zda není daný vrchol již v seznamu (na základě bool proměnné, kterou má každý objekt představující vrchol), v případě, že tomu tak není, vloží jej na konec seznamu. Metod vykonávajících samotný výpočet je více (viz jejich popis v předchozí kapitole), rozhodnutí o tom, která bude volána, provádí na základě uživatelova nastavení třída *Physics*, která slouží jako rozhraní pro erozi v programu. Přes toto rozhraní současně přicházejí do metody realizující erozi také všechny parametry od uživatele, které řídí průběh eroze.

Metoda realizující samotnou erozi je volána v pravidelných intervalech (volání probíhá ve stejném cyklu jako volání renderovací metody a zajišťuje jej *rozhraní Physics).* Metoda prochází všechny vrcholy uložené v seznamu. Pro každý vrchol projde jeho sousedy a zjišťuje, zda je úhel mezi vrcholem a jeho sousedem větší než stanovená hodnota (jako úhel je brán úhel mezi spojnicí bodů a přímkou procházející vyšším z nich, tj. atan(len_z / len_xy), kde len_z je vzdálenost bodů v ose z a len_xy je vzdálenost bodů v osách x,y). V kladném případě jsou určeny potřebné údaje a pokračuje se dalším sousedem. Po projití všech sousedů přichází na řadu druhá fáze, v níž jsou změněny výšky vrcholů již popsaným způsobem. Pokud nebyla u zkoumaného vrcholu prováděna jakákoli změna, přejde algoritmus do třetí části. Tato část zajišťuje odstranění vrcholů ze seznamu. Každý vrchol má čítač, který určuje, kolikrát vstoupil do této části. Pokud tento čítač překročí určitou hodnotu, je vrchol ze seznamu odebrán, pokud ale vrchol vstoupí do části, v níž se provádějí změny, je mu tento čítač vynulován. Tento přístup zabraňuje předčasnému vyhození určitých vrcholů, k němuž docházelo při určitých konfiguracích sítě (například se jedná o vrcholy při okrajích sítě).

5.4 Parametry

Činnost celého programu lze ovlivňovat celou řadou nastavení a parametrů. Vstupem parametrů může být buďto textový soubor daného formátu, anebo klávesnice, jíž lze také některé parametry měnit. Ovládání prostřednictvím klávesnice je popsáno v kapitole Přílohy.

Nejprve jsou uvedeny dva nejpodstatnější parametry určující typ virtuálního nástroje a metodu eroze. Dále následuje popis formátu vstupního souboru s stručným popisem všech parametrů.

Hodnoty parametrů

Parametr „tool“ označuje jednotlivé virtuální nástroje. V tabulce (Tabulka 5.1) jsou uvedeny jejich číselné kódy tak, jak jsou použity v souboru s parametry (a klávesy s těmito čísly slouží i k přepínání mezi nástroji prostřednictvím klávesnice). Dále je uveden krátký popis a snímek jednoho zásahu nástrojem (drátový a stínovaný model). Podrobnější informace o nástrojích lze nalézt v kapitole 3 a v materiálu [Pur07].

Tabulka 5.1: Virtuální nástroje

Číselný kód	Popis nástroje	Obrázek
7	Nástroj simuluje vyhrnutí materiálu z oblasti stopy nástroje do okolí. U nástroje je možné měnit jeho hloubku, průměr a počet svislých stěn.	
8	Nástroj provede odebrání, respektive přidání materiálu. Je-li dno nástroje pod terénem odebírá nástroj materiál, je-li „dno“ nástroje nad terénem nástroj přidává materiál. U nástroje lze měnit hloubku, respektive výšku, průměr a počet svislých stěn.	
9	Základní nástroj od něhož je odvozen nástroj č. 8. Má stejné vlastnosti, ale počet stěn a jejich rozměry nelze měnit.	

Parametr „erosion“ označuje jednotlivé metody eroze. V následující tabulce jsou uvedeny číselné kódy jednotlivých metod tak, jak jsou použity v souboru s parametry, a jejich krátký popis (podrobnosti viz kapitola 4.3).

Tabulka 5.2: Metody eroze

Číselný kód	Popis metody
1	Přenos materiálu probíhá mezi sousedy s nejvyšším rozdílem výšek.
2	Rozděluje přenášený materiál nepřímo úměrně rozdílu výšek mezi vrcholy.
3	Pro úhel větší než 80° použita metoda č. 1, pro menší metoda č. 2.
4	Přenos materiálu na základě poměru objemů ovlivněných těles.

Soubor

Název (cesta) souboru obsahujícího nastavení programu může být zadán jako parametr při spouštění programu. Formát souboru je v podobě názvu parametru a jeho hodnoty, přičemž pořadí parametrů je libovolné a názvy jsou následující:

- -p číslo - síť vznikne z náhodných bodů o počtu *číslo* (podrobnosti viz [Pur07])
- -f cesta - síť vznikne z bodů v souboru *cesta*
- -a cesta - síť načtena z souboru *cesta* (načtena celá síť)
- graphics 0-1 – 1 = plná síť, 0 = drátěný model
- angle hodnota - hodnota úhlu pro erozi
- erosion číslo_metody(1-4) - určuje metodu použitou pro erozi
- tool číslo_nástroje(7-9) - určuje typ „nástroje“
- add_point číslo - vzdálenost pro přidávání bodů v erozi (0 = nepřidávat)
- radius číslo - průměr „razítka“ (musí být použita desetinná tečka)
- height číslo - hloubka „razítka“
- erosion_param číslo - určuje počet opakování metody eroze
- erosion_bit číslo - hodnota *z* souřadnice, která je předávána mezi vrcholy při algoritmu eroze.

Pokud není zadán soubor s parametry, nebo tento soubor obsahuje nesprávná data, jsou parametrům přiřazeny přednastavené hodnoty.

O tom, jak hodnoty jednotlivých parametrů ovlivňují činnost programu, (mimo jiné) pojednává následující kapitola.

6. Výsledky

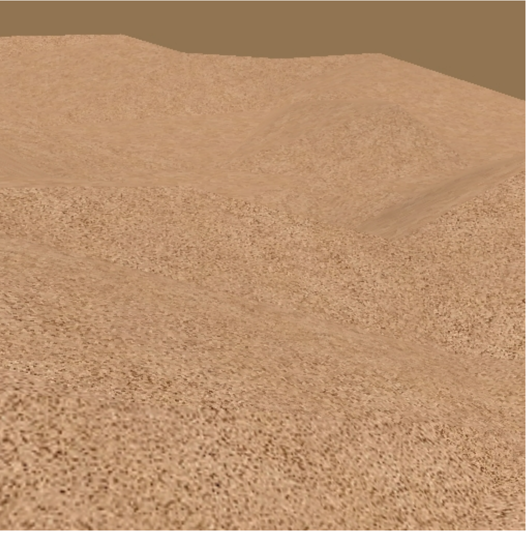
V této kapitole budou popsány a zobrazeny dosažené výsledky. Budou zde uvedeny příklady chování popisovaných metod a vliv jednotlivých parametrů na dosažené výsledky. Vzhledem k vlastnostem celé práce bude mít většina prezentovaných výsledků podobu snímků, zobrazujících dění v okně programu. Veškeré uváděné výsledky byly dosaženy na počítači s těmito parametry: AMD Sempron 2800+ (1,6GHz), 512 MB RAM, graf. karta NVIDIA GF6600.

6.1 Vizualizace sítě

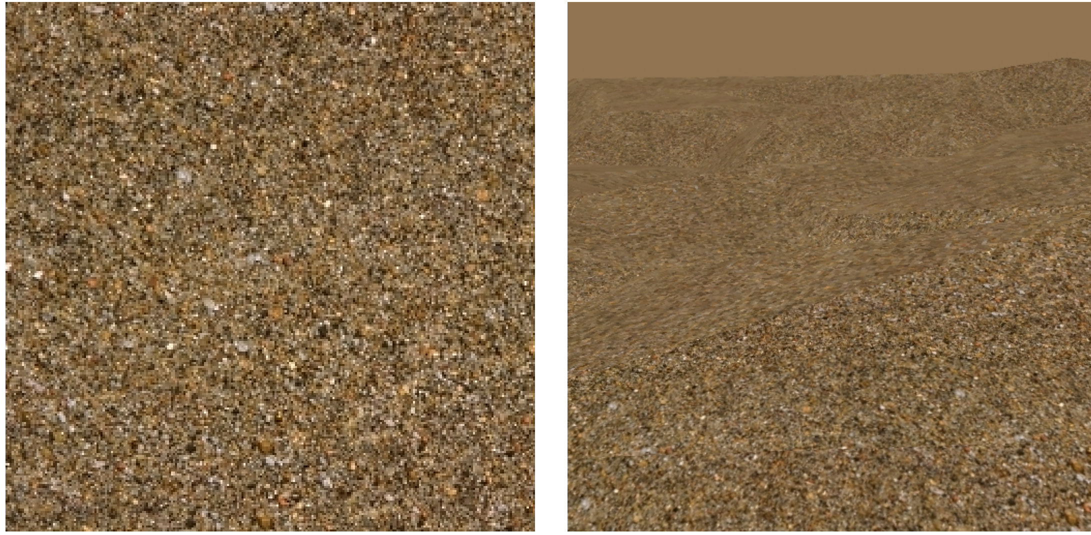
V této části bude probrána vizualizace trojúhelníkové sítě jakožto povrch s aplikovanou texturou a osvětlením. Kromě této varianty umožňuje program také vizualizaci sítě jako „drátový“ model (viz kapitola 5.4 Parametry), autory této části jsou kolegové Jan Kadlec a Václav Purchart.

V předchozích kapitolách bylo popsáno, že k vizualizaci sítě byly použity tzv. Vertex Buffer Objects, jimž byla dána přednost před řešením s display-listy. Tento výběr nemá na výsledný vzhled sítě žádný vliv. Oproti tomu výběr textury a použití mip-map, stejně jako normály povrch a použité osvětlení, tvoří podstatnou část vzhledu povrchu. Na následujících snímcích je pak vidět vliv jednotlivých použitých textur (obr. 6.1 , 6.2 , 6.3).

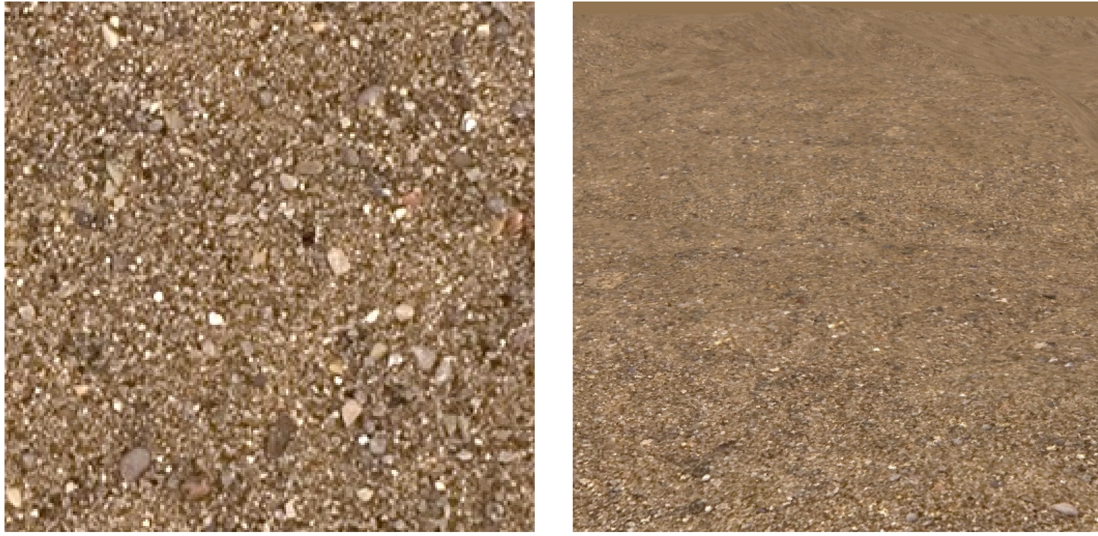
Byla vyzkoušena celá řada textur, od vytvořených v grafických programech (Gimp, Adobe Photoshop) až po fotografie skutečného písku (získané z volně dostupných databází na internetu). Skutečné fotografie se ukázaly jako vhodnější, ale bylo nutné, aby splňovaly několik základních vlastností, především stejnoměrnost osvícení snímku a dostatečnou míru detailů. Je také nutné, aby použitá textura měla rozměry 2^k, kde k je celé číslo (moderní grafické karty si ale poradí i s libovolným rozměrem).



Obr. 6.1: Textura (vlevo) a povrch (vpravo), textura vytvořena v programu Gimp.



Obr. 6.2: Textura (vlevo) a povrch (vpravo), textura je fotografie písku.

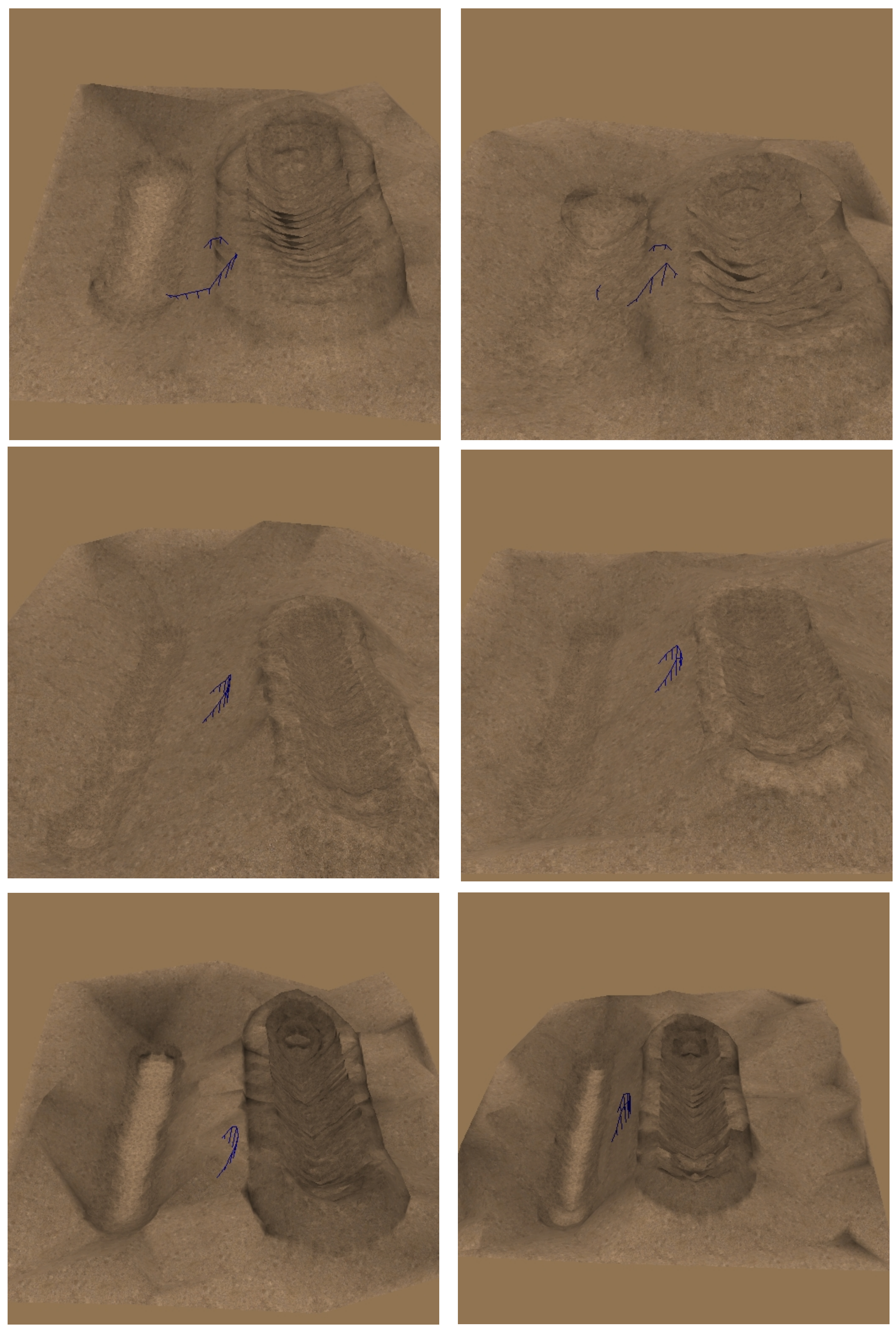


Obr. 6.3: Výřez z textury (vlevo) a povrch (vpravo), textura je fotografie písku.

Ve výsledném programu je pak použita textura z obr. 6.3 (autor Emil Persson, http://www.humus.ca). Vliv použití mip-map není příliš zřejmý ze statických snímků. Jejich hlavní význam je především v umožnění namapování textury tak, aby byla dostatečně detailní i při pohledu z malé vzdálenosti a přitom nedocházelo při pohledu z větších vzdáleností k problikávání jednotlivých pixelů.

6.2 Zásahy do sítě a eroze

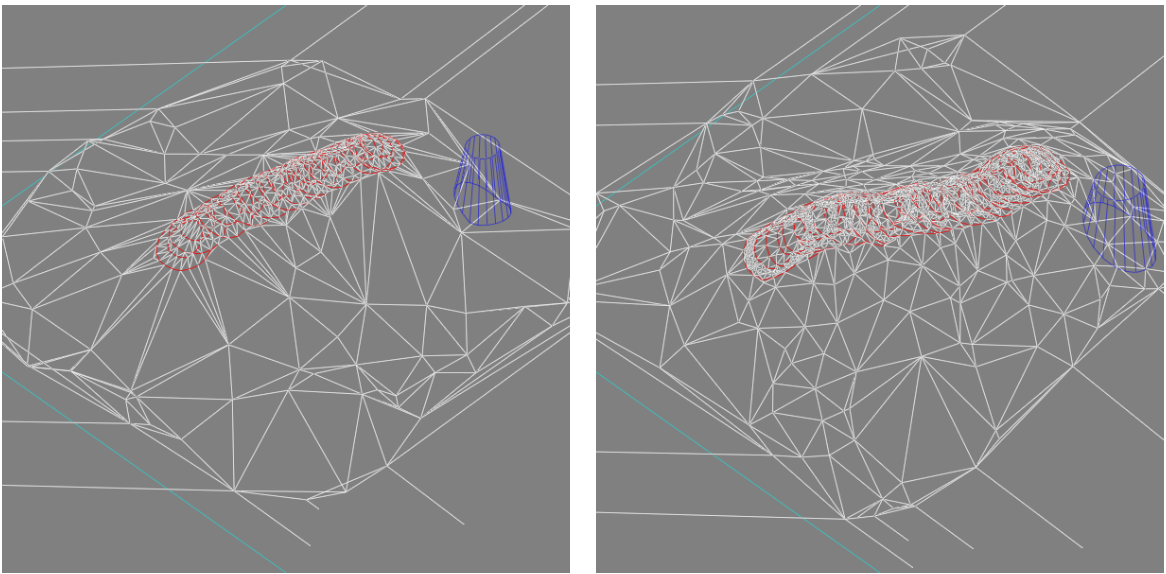
K provádění zásahů do sítě slouží uživateli virtuální nástroje, na jejichž změny v síti reaguje algoritmus gravitační eroze, který provádí změny výšek vrcholů sítě. V této kapitole budou předvedeny dva virtuální nástroje a čtveřice algoritmů eroze, s ohledem na různé nastavení jejich parametrů (Obr 6.4). Dále pak zjemňování sítě při algoritmu eroze a několik dalších testů.



Obr. 6.4: Snímky vlevo algoritmus č. 3, vpravo algoritmus č. 4.

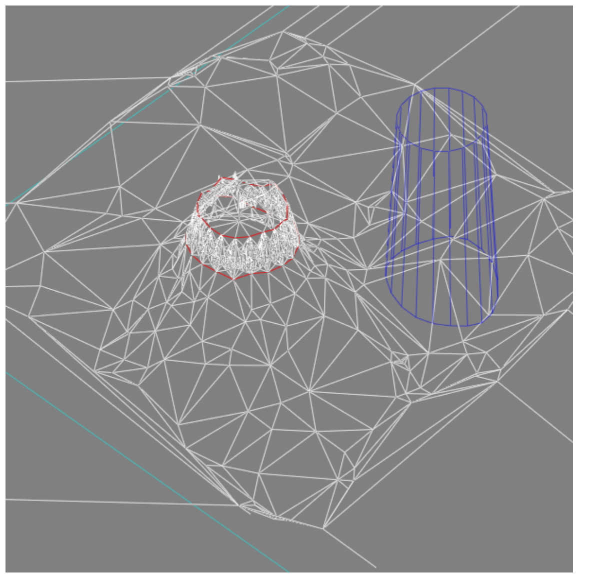
Na Obr 6.4 je série snímků sítě po reakci na zásahy dvěma typy nástrojů. Obrázky ukazují výsledek po vyrytí rýhy při aplikaci dvojice metod eroze (viz Tabulka 5.2), metoda č. 3 je reprezentantem skupiny metod založenou na přenosu materiálu bez znalosti objemů, metoda č. 4 je založena na přesné znalosti objemů. Obrázky ukazují síť poté, co algoritmus eroze dosáhl ustáleného stavu. Rýha vlevo byla způsobena nástrojem odebírajícím materiál (nástroje č. 8 – viz Tabulka 5.1), rýha vpravo nástrojem simulujícím vytlačení materiálu (nástroje č. 7 – viz Tabulka 5.1). Modrý objekt na snímcích představuje vizualizaci virtuálního nástroje. Úhel po jehož překročení je realizována eroze byl v případě prvních dvou řad snímků 30° a v případě poslední třetí řady 45°. V průběhu eroze nedocházelo k přidávání bodů. Nástroj v druhé a třetí řadě snímků měl přibližně poloviční průměr než nástroj v první řadě snímků. Je dobře patrné, že úhel 45° vede ke vzniku ostrých hran mezi jednotlivými trojúhelníky sítě a síť přestává budit dojem písku. Zvyšování úhlu by mělo budit dojem větší soudržnosti materiálu (například mokrý písek), ale zároveň vede k zhoršení vizuálního vjemu výsledného povrchu.

Další z prezentovaných vlastností je zjemňování sítě, tj. přidávání vrcholů a trojúhelníků do sítě. Připomenu, že přidání vrcholů proběhne do těch trojúhelníků, jejichž hrana je zpracovávána v erozi a je delší než daná mezní hodnota. Na následující dvojici snímků je vidět síť po zásahu nástrojem bez aktivního přidávání vrcholů a s přidáváním vrcholů (obr 6.5).



Obr. 6.5: Přidání materiálu nástrojem, vlevo bez přidávání bodů, vpravo s přidáváním bodů, pokud délka hrany překročí hodnotu 0,1 .

Na obrázcích je zřetelné určité zjemnění sítě, to je ovšem v režimu zobrazení povrchu (otexturované sítě) jen velmi málo patrné. Podstatným problémem tohoto režimu je, že dochází k přidání bodů i do oblasti zásahu nástroje, které ale později vlivem eroze leží velmi blízko u sebe (obr. 6.6), ze stejného důvodu je tento postup téměř nepoužitelný v kombinaci s nástrojem číslo 7 (vyhrnutí materiálu – dojde k vložení ještě většího počtu vrcholů).



Obr. 6.6: Přidávání vrcholů do stopy nástroje.

Obr. 6.7: Stopa nástroje ve tvaru šipky.

Obr. 6.8: Síť bez aplikované textury

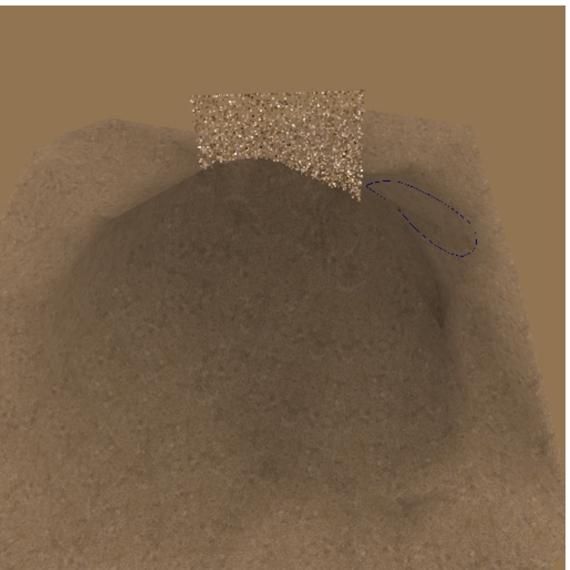
Obr. 6.9: Pravidelné pole [Bene06].

Na obrázku 6.7 je vidět pokus o vytvoření tvaru šipky nástrojem č. 7. Úhel eroze byl nastaven na hodnotu 35°. Ze snímku je také patrný problém s osvětlením. Oblast uvnitř stopy nástroje je velmi tmavá a hrana vytlačeného materiálu v okolí stopy je velmi špatně zřetelná (obojí je dobře patrné i na obrázku 6.8, kde je stejná scéna jako na obr. 6.7, ale bez namapované textury). Obojí je pravděpodobně důsledkem způsobu výpočtu normál v jednotlivých vrcholech. Při interpolaci jsou pro daný vrchol normály spočteny z normál všech trojúhelníků, tvořených tímto vrcholem. Pokud je tedy některý z těchto trojúhelníků odlišně natočen k světlu, může značně ovlivnit směr výsledné normály. Výpočet vlastně vyhlazuje normály. Při použití konstantního stínování by byly normály příliš rozdílné a změny příliš viditelné, řešením by tedy mohlo být použití pouze „některých" trojúhelníků pro výpočet normály v daném vrcholu. Problém je samozřejmě méně zřetelný při použití intenzivnějšího osvětlení, což má ale za následek ještě větší ztrátu hran v okolí stopy nástroje.

Obrázek 6.9 představuje podobný „tvar“ vytvořený v programu založeném na pravidelném poli (snímek převzat z práce [BEN06]). Je vidět, že povrch terénu na obrázku 6.9 nepůsobí příliš realisticky (to ale také nebylo hlavním záměrem autorů), ale zato výsledek „rytí“ je věrohodnější než v našem případě. Stopa po nástroji je hladká, oproti tomu v našem případě jsou ve stopě patrné jednotlivé zásahy nástroje (viz obr. 6.8)

Na dalším obrázku (obr. 6.10) je další z testů chování vlastností eroze. Velmi jednoduchým způsobem je simulován „částicový systém“, tj. „částice“ vypadávají z bodů na přímce ležící nad sítí a do sítě jsou přidány vrcholy, ležící na průmětu bodů, z nichž vypadávají částice. Tyto vrcholy v síti jsou pravidelně navyšovány o určitou hodnotu a současně jsou přidány do seznamu vrcholů zpracovávaných erozí. Výsledkem je pomalu se zvyšující hromada a odhalení některých dalších vlastností použitých metod.

Na snímcích je dobře patrný vliv mezního úhlu, zatímco hromada na snímku vlevo je mnohem nižší a přesto již téměř dosahuje okraje povrchu, hromada vpravo je výrazně vyšší a užší (ostatní parametry jsou v obou případech stejné). Při těchto simulacích byl parametr „erosion_bit“, určující množství předávaného materiálu, nastaven na hodnotu 0,1, což vede k menšímu kmitání sítě, které se v těchto případech objevuje.



Obr. 6.10: Částice. Vlevo úhel 30°, vpravo úhel 45°. Metoda eroze č. 4.

Zhodnocení

Výrazný vliv na výsledek i průběh eroze mají parametry. A to jak ty, jejichž vliv je zřejmý na uvedených snímcích (úhel), tak i ty, jejichž vliv je viditelný při chodu programu (parametry označované jako „erosion_param“, „erosion_bit“ - viz kapitola 5.4). S výsledky eroze úzce souvisí i vlastnosti virtuálních nástrojů. Zatímco nástroj odebírající (přidávající) materiál (č. 8) pracuje celkem bez problémů, nástroj simulující vytlačení materiálu trpí značnými problémy. Je to jednak vizuální stránka v průběhu zásahu nástrojem i po „zerodování“ stropy nástroje, kdy výsledek příliš neodpovídá skutečnosti. Příčinou těchto problémů je především nutnost „šetřit“ s množstvím vnucených hran, což následně vede k nepříliš ideálnímu tvaru vytlačené části stopy. Všechny nástroje pak trpí problémy při činnosti na okrajích sítě, které jsou pravděpodobně způsobeny úzkými trojúhelníky, které zde občas vznikají při triangulaci (a vše samozřejmě závisí na konkrétní konfiguraci sítě).

6.3 Výpočetní náročnost

K určitému naznačení výpočetní náročnosti výsledného programu jsem zvolil postup, kdy je program pouštěn v různých situacích a je měřen počet snímků za sekundu (fps). Jako nejvíce vypovídající jsem se rozhodl zvolit hodnotu nejnižší a průměrnou (parametry počítače uvedeny na začátku kapitoly). K měření byl použit nástroj Fraps (www.fraps.com), který disponuje funkcí umožňující měřit v daném časovém úseku počet snímků za sekundu.

Úhel pro erozi byl ve všech případech uvedených v tabulce 6.1 totožných 35°. Tento parametr má samozřejmě také určitý vliv na časovou náročnost výpočtů, ale vzhledem k tomu, že se jedná o parametr, který nelze nastavovat v příliš velkém rozsahu (při zachování reálného chování sítě), jsem jeho vliv do měření nezařadil. Stejně tak byla použita jediná metoda eroze založená na poměru objemů (č. 4), ostatní metody jsou výpočetně méně náročné. Měření probíhalo tak, že po zásahu nástrojem do sítě jsem počkal na dosažení ustáleného stavu (seznam vrcholů pro erozi je prázdný, což se projeví nárůstem fps) a měření jsem ukončil, aby nedocházelo k zkreslení výsledků.

Tabulka 6.1:

Popis situace, parametry programu, atd.	*Minimální fps*	*Průměrné fps*
Pohyb nad povrchem, 10000 vrcholů, bez zásahů nástroji.	84	84,7
Vytvoření scény z obr 6.10 *.	4	36,8
„Sypání písku“, 100 vrcholů, „kousek“ 0,1.**	30	34,9
„Sypání písku“, 1000 vrcholů, „kousek“ 0,1.**	18	22,1
Přidávání nástrojem č.8, 1000 vrcholů.***	3	9,3
Rytí nástrojem č.8 tenkým, 1000 vrcholů ***	4	18,7
Rytí nástrojem č.8 širokým, 1000 vrcholů ***	3	13,3

*,** nástroj č. 7, parametry dle názvů z kap. 5.4 .
, * snímky z programu pro tyto případy viz příloha.

Z prvního příkladu uvedeného v tabulce je vidět, že pokud není potřeba v programu znovu vytvářet VBO (tj. nedochází ke změnám v síti), je vizualizace i na rozsáhlejších datech dostatečně rychlá (což jsme předpokládali – viz kapitola 4.2). Hlavní výpočetní zátěž tedy připadá na algoritmy spojené s tvorbou VBO a s prací v síti (eroze, nástroje). Příklad číslo dvě ukazuje, že nástroj č. 7 je výpočetně velmi náročný. Program při jeho použití dosahuje podobných výsledků (časových) jako při mnohem delším (časově i početně) rytí jednodušším nástrojem č. 8 (příklady 6 a 7). „Sypání“ písku je v tomto případě mnohem méně náročné, což je dáno malým počtem bodů, které jsou ovlivněny (v tomto případě přímo 10 bodů a další v závislosti na erozi). Je ale vidět rozdíl v náročnosti v závislosti na velikosti sítě. Ta totiž ovlivňuje počet bodů zasažených erozí. Rozdíly v poslední dvojici případů jsou malé a mohou být způsobeny jednak tím, že tenčí nástroj zasáhne méně trojúhelníků, které je nutné dělit, jednak možná nestejným počtem provedených zásahů do sítě nástroji (přičemž oba nástroje měly stejné parametry jako je počet stěn, atd.).

Rychlost a výpočetní náročnost úzce souvisí s nastavením jednotlivých parametrů. Pokud zvýšíme parametr „uhel“, který určuje mezní úhel, při němž začíná, respektive končí

eroze, dojde k mírně rychlejšímu dosažení ustáleného stavu. To samé platí pro parametr „kousek“ (určuje množství přenášeného materiálu v jednom kroku eroze), jehož vyšší hodnota se může projevit kmitáním v síti anebo dokonce rozkmitáním celé sítě bez možnosti dosažení ustáleného stavu. Parametr „fyz_param“, který určuje počet opakování algoritmu eroze v rámci jednoho cyklu programu opět může výrazně ovlivnit časovou náročnost celého programu, ale i vzhled a průběh eroze. Je tedy nutné tyto parametry vybalancovat tak, abychom získali co nejlepší výsledky při rozumné výpočetní náročnosti.

7. Závěr

Naší snahou bylo především otestovat použití nepravidelné sítě k realizaci tohoto typu simulací a vytvořit program, který umožní nad nepravidelnou sítí realizovat zásahy předměty, erozi gravitací a tuto síť zobrazit jako povrch simulovaného materiálu.

Podařilo se nám vytvořit program realizující nad nepravidelnou sítí výše zmíněné operace, tj. „rýpání“ předmětem do sítě a následnou erozi nad sítí. Nepravidelná síť bodů, které představují výšku terénu, je triangulována prostřednictvím Delaunayovy triangulace (DT) respektující vnucené hrany. Do této sítě lze provádět zásahy jednoduchými předměty. Nad sítí operuje algoritmus eroze, který reaguje na prováděné zásahy předmětem do sítě a příslušným způsobem upravuje síť. Trojúhelníková sít získaná DT je zobrazována prostřednictvím OpenGL, je otexturována a nasvícena.

Jedním z cílů této práce bylo zjistit vlastnosti simulace terénu využívající nepravidelné sítě. Vzhledem k tomu, že výsledný program neobsahuje odebírání již nepodstatných bodů z sítě, je poněkud obtížné vyvodit z výsledků této práce přesné závěry pro tuto techniku obecně. Přesto lze konstatovat, že se nám alespoň částečně podařilo prokázat použitelnost tohoto přístupu pro simulace terénu.

Na tomto projektu lze najít mnoho směrů dalšího rozvoje. Jednak je to již zmiňované rozšíření pro využití haptických zařízení, ale jedná se i o věci týkající se úprav již hotových částí. Například lepší zobrazení předmětu, pomocí něhož jsou prováděny zásahy do sítě, spojité otexturování sítě, vylepšení ovládání a změna algoritmu eroze anebo již zmiňované odebírání vrcholů z sítě. Také samotné virtuální nástroje, jimiž jsou prováděny zásahy do sítě, jsou velmi jednoduché a v souvislosti s odebíráním bodů z sítě se nabízí možnost jejich výrazného zlepšení. Také optimalizace a zefektivnění některých částí stávajících kódů je částí, jíž by bylo vhodné se dále věnovat. Program může v určitých konfiguracích vykazovat nestabilitu končící pádem programu či vznikem chyb v simulovaném terénu. Nalezení míst v kódu, v nichž k těmto chybám dochází, je také nesnadný úkol, který ale bude muset být v případě dalšího rozvoje vyřešen. Dalším problémem zmíněným v předchozí kapitole je chování eroze na okrajích sítě, které je způsobeno přítomností dlouhých úzkých trojúhelníků při některých konfiguracích sítě. Zde by mohlo pomoci přidání bodů na okraje sítě, což by mělo zabránit vzniku úzkých trojúhelníků na okrajích sítě.

Klíčovým problémem, na nějž je potřeba zaměřit další práci, je tedy odebírání vrcholů ze sítě. S tím pak souvisí další rozvoj v ostatních částech a možnosti dalšího rozšiřování programu.

Literatura

[BEN06] Beneš B., Dorjgotov E., Arns L.,Bertoline G.: Granular Material Interactive Manipulation:Touching Sand with Haptic Feedback. WSCG'2006, http://iason.zcu.cz/wscg2006/Papers_2006/Full/D05-full.pdf .

[CVUT02] Stínování, světla a materiály, http://netra.felk.cvut.cz/~zpg/cviceni7/ , 2002

[Fou06] Fousek M. : Simulace vzniku a vývoje písečných dun. Diplomová práce, ČVUT 2006.

[Ger06] Geršl V. : Modelování a vykreslování trávy pro herní aplikace. Bakalářská práce, ZČU 2007, herakles.zcu.cz/~skala/MSc/Diploma_Data/2007_BP_Gersl_Vladimir.pdf.

[Kad07] Kadlec J., Purchart V.: Dokumentace k projektu 5, Modelování eroze a deformací terénu na povrchových modelech. KIV/ZČU 2007.

[ONO03] Onoue K., Nishita T.: Virtual Sandbox. 11th Pacific Conference on Computer Graphics and Applications. 2003, nis-lab.is.s.u-tokyo.ac.jp/~onoue/research/lib/PG2003Onoue.pdf.

[Pur07] Purchart V.: Deformace terénu pro virtuální realitu – datová, logická a vizualizační část modelu. Bakalářská práce, KIV/ZCU 2007, herakles.zcu.cz/~skala/MSc/Diploma_Data/2007_BP_Purchart_Vaclav.pdf

[SDL] SDL library documentation.

[Simb06] Simbartl M., Programujeme v OpenGL aplikace v C++. www.pcsvet.cz . 2006

[Sta05] S. Stachniak, W. Stuerzlinger, An Algorithm for Automated Fractal Terrain Deformation, Computer Graphics and Artificial Intelligence 2005, http://www.cse.yorku.ca/~wolfgang/papers/fractaldeform.pdf.

[SUM98] R. W. Sumner, J. F. O'Brien, and J. K. Hodgins. Animating Sand, Mud, and Snow. Graphics Interface '98, http://www.gvu.gatech.edu/animation/Papers/sumner:1999:ASM.pdf.

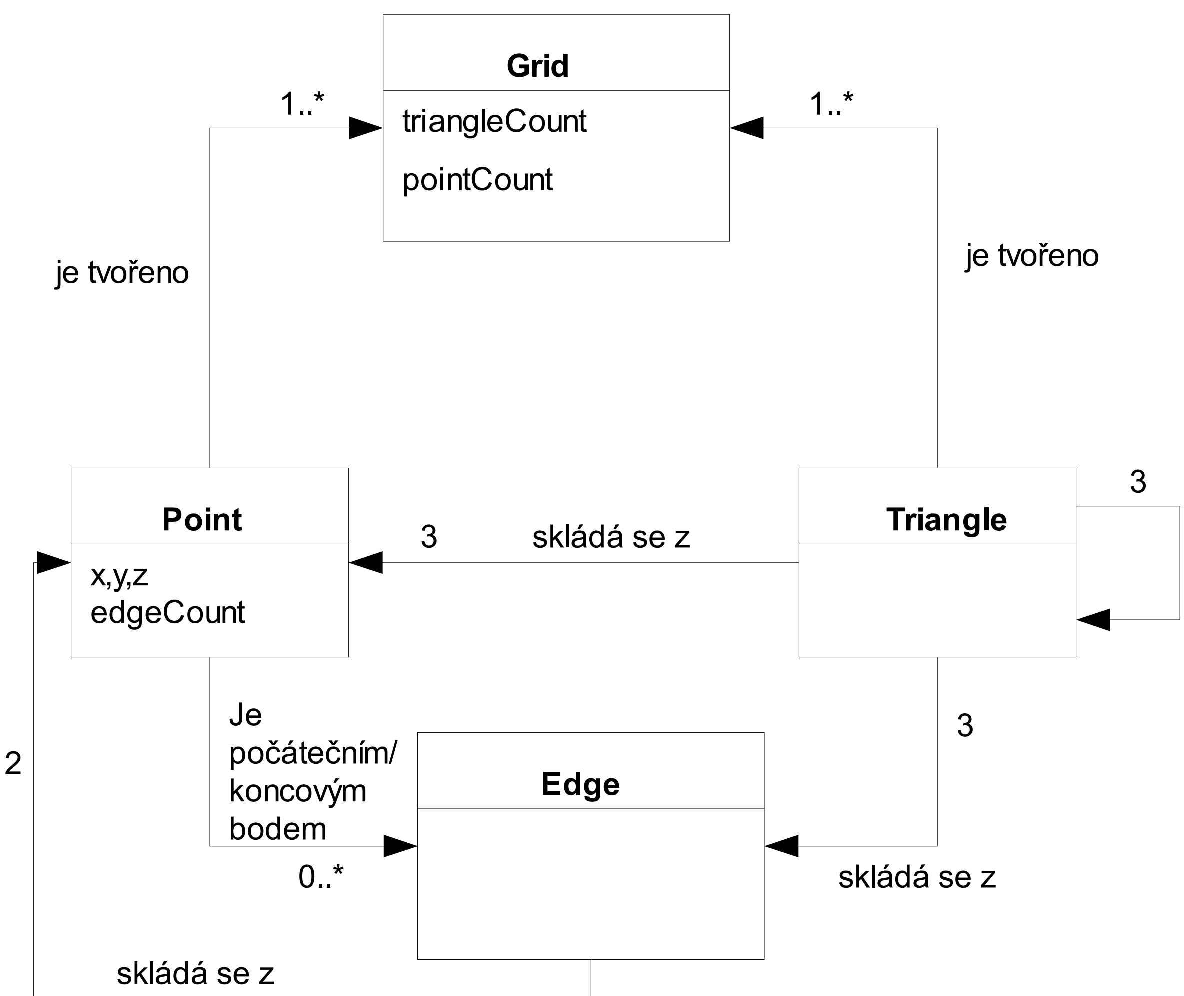
[Tes06] Beneš B., Těšínský V., Hornyš J., Sanjiv K. Bhatia : Hydraulic erosion. Computer animation and virtual worlds, 2006.

[TIS03] Tišnovský P., Grafická knihovna OpenGL (1), (25). www.root.cz . 2003.

[Tur05] Turek M., SDL hry nejen pro linux. www.root.cz . 2005.

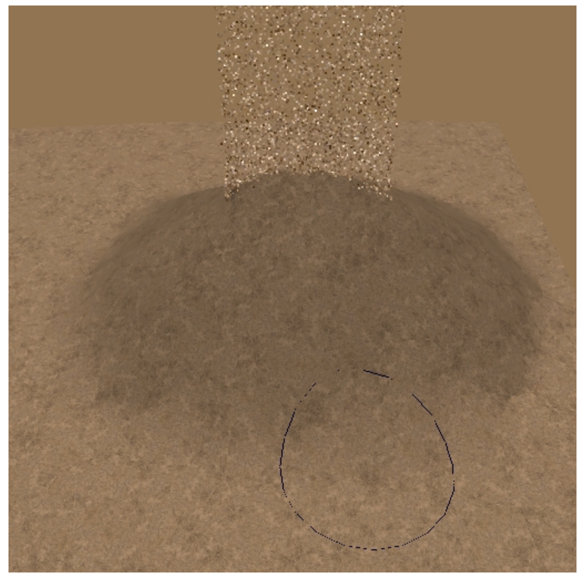
[Wiki] OpenGL. http://en.wikipedia.org/wiki/ .

Přílohy

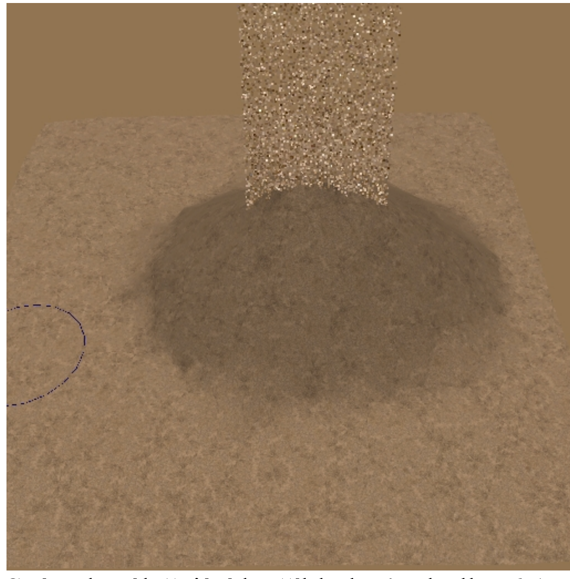


Příloha 1: UML diagram hlavních tříd programu.

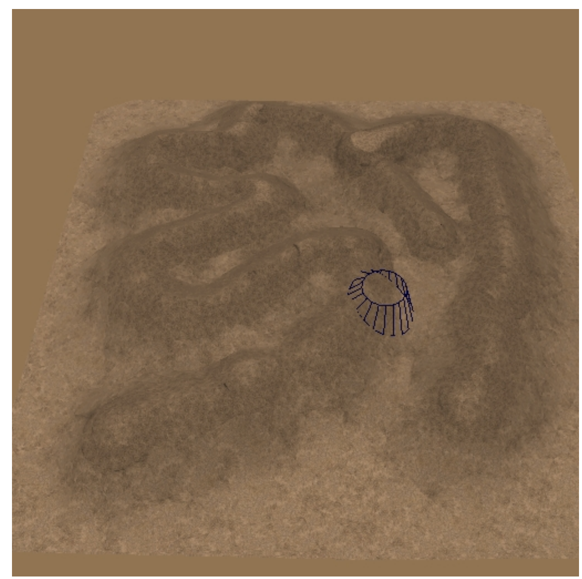
Příloha 2 : Snímky programu, podrobnosti viz popisky.



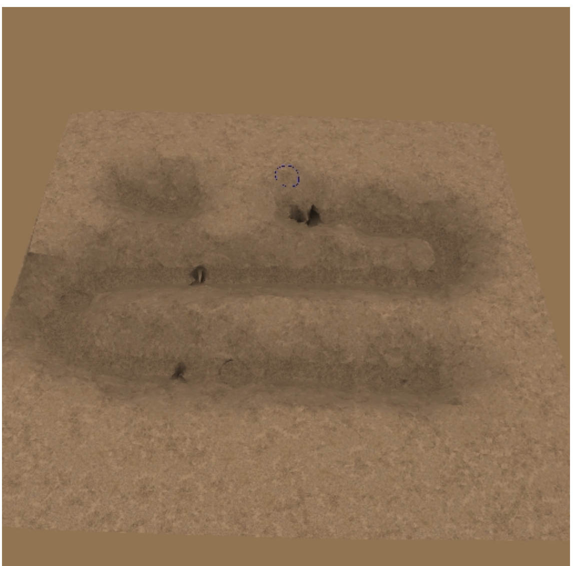
Snímek náležející k příkladu 3 tabulky 6.1



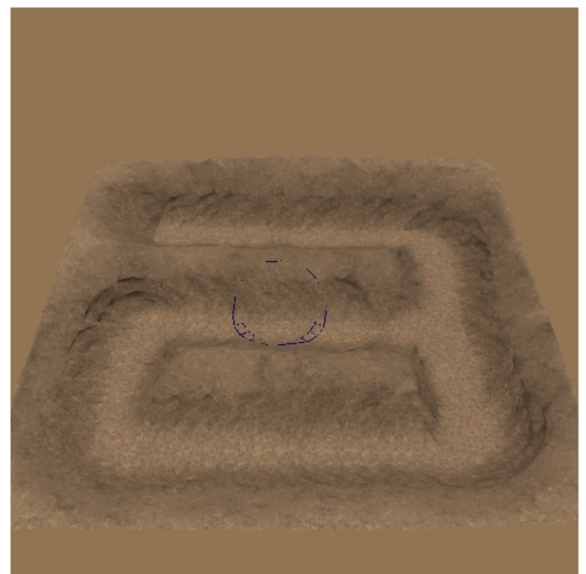
Snímek náležející k příkladu 4 tabulky 6.1



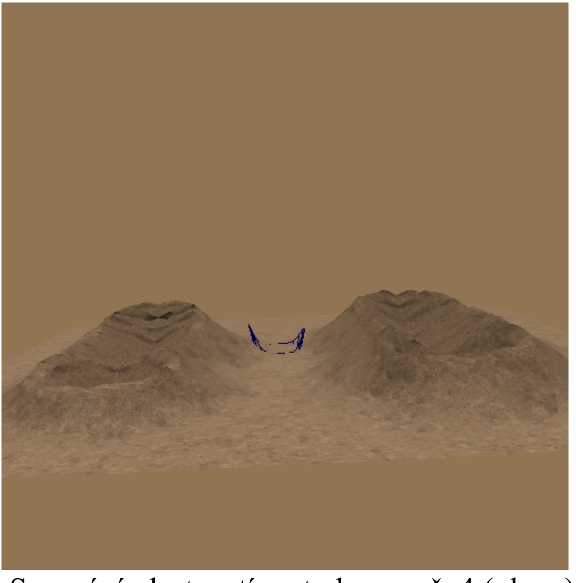
Snímek náležející k příkladu 5 tabulky 6.1



Snímek náležející k příkladu 6 tabulky 6.1



Snímek náležející k příkladu 7, tabulky 6.1



Srovnání vlastností metod eroze č. 4 (vlevo) a č. 3 (vpravo) – rozdíl v objemech je silně závislý na konkrétním nastavení.(síť 1000 vrcholů, úhel 35°)

Příloha 3: Ovládání programu

Program se ovládá prostřednictvím klávesnice a myši. Ovládání se mírně liší v závislosti na zvolené vizualizaci. Verzi vizualizace lze měnit pouze prostřednictvím konfiguračního souboru (možnost změny za běhu programu by si vyžádalo značné zásahy do obou částí vizualizace).

U obou vizualizací je pohybem myši bez stisknutých tlačítek řízen pohyb virtuálního nástroje nad plochou terénu. Stisknutím levého tlačítka se provede zásah nástroje do terénu. Stisknutí pravého tlačítka a současný pohyb myši slouží k změně směru pohledu kamery.

K pohybu kamery v prostoru slouží šipky na klávesnici (je nutné použít samotné šipky, nikoli jejich variantu na numerické klávesnici). Pohyb s kamerou je možný pouze v případě zobrazování sítě jakožto otexturovaného povrchu (parametr „graphics 1“). Šipky dopředu a dozadu slouží k pohybu v ose pohledu kamery (dopředu a zpět). Šipky doprava a doleva slouží k posunu ve směru osy kolmé na osu pohledu kamery.

Další klávesy jsou funkční v obou případech vizualizace a slouží k nastavení parametrů eroze a virtuálních nástrojů. Číslice 1 – 4 určují typ metody eroze (viz kapitola 5.4 Parametry) a číslice 7 – 9 typ virtuálního nástroje (v obou případech se jedná o klávesy s čísly umístěnými v alfanumerické části klávesnice, knihovna SDL je totiž odlišuje od kláves v oddělené numerické části).

Další klávesy:

- *v / b* - zvětšení / zmenšení úhlu eroze
- *e* / *d* - změna hloubky, respektive výšky nástroje (e = -0,1 / d = +0,1)
- *f* / *r* - zmenšení / zvětšení průměru nástroje
- *g* / *t* - zmenšení / zvětšení počtu hran nástroje
- *q* / *a* – přičtení / odečtení jedničky od parametru určujícího počet opakování eroze
- o / p – zapnutí / vypnutí vykreslování nástroje.
- h / j – zapnutí / vypnutí zobrazení nápovědy v okně programu.
- escape – ukončení programu.