VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

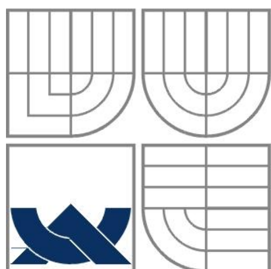
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HERNÍ PROJEKT SE ZAMĚŘENÍM NA SVĚTELNÉ EFEKTY

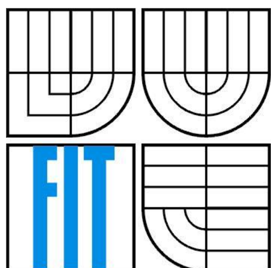
BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE DUŠAN HUPKA
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HERNÍ PROJEKT SE ZAMĚŘENÍM NA SVĚTELNÉ EFEKTY

COMPUTER GAME PROJECT FOCUSED ON LIGHT EFFECTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
Dušan Hupka

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. Jan Pečiva, Ph. D.

BRNO 2012

Abstrakt

Cílem této práce je představit a implementovat metody pro simulaci nerovností povrchu. Na začátku je vysvětlená potřebná teorie k Phongovmu osvětlení a následně teorie nejznámějších metod. Zaměření práce je především na Normal mapping a množství typů Parallax mappingů. Následně je přistoupené k implementaci těchto metod a k demonstrační aplikaci. Závěrem je uvedené krátké srovnání výkonů implementovaných metod.

Abstract

This bachelor thesis aims at introduction into methods for simulation of undulation of surface. There will be explained Phong reflection model and other best known theories at the beginning. The text deals with Normal mapping and various types of Parallax mapping. Then there will be demonstration application. There will be short comparison performances of implemented methods.

Klíčová slova

Delta3D, Vertex Shader, Fragment Shader, Phongova interpolace, Phongovo osvětlení, Normal mapping, Parallax mapping, Parallax Occlusion Mapping, Iterative Parallax Mapping, Parallax Occlusion Mapping With Soft Shadows, Parallax Occlusion Mapping se siluetami

Keywords

Delta3D, Vertex Shader, Fragment Shader, Phong Interpolation, Phong Reflection Model, Normal mapping, Parallax mapping, Parallax Occlusion Mapping, Iterative Parallax Mapping, Parallax Occlusion Mapping With Soft Shadows, Parallax Occlusion Mapping With Silhouettes

Citace

Hupka Dušan: Herný projekt so zameraním na svetelné efekty, bakalářská práce, Brno, FIT VUT v Brně, 2012

Herný projekt so zameraním na svetelné efekty

Prohlášení

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Jana Pečivu, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……………………
Dušan Hupka
16. mája 2012

Poděkování

Rád by som poďakoval pánovi Ing. Janovi Pečivovi, Ph.D. za vedenie, smerovanie a ochotu podeliť sa o svoje poznatky a pomoc pri tvorbe tejto práce. Takisto veľká vďaka patrí pánovi Ing. Tomášovi Starkovi za množstvo odborných konzultácií, ktoré bol ochotný poskytnúť.

© Dušan Hupka, 2012
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1
1 Úvod 2
2 Metódy simulácie zakrivenia povrchu 3
2.1 Phongovo osvetlenie 3
2.1.1 História 3
2.1.2 Phongova interpolácia 3
2.1.3 Phongov osvetľovací model 4
2.1.4 Blinn-Phong osvetľovací model 7
2.2 Bump mapping a normal mapping 8
2.2.1 História 8
2.2.2 Princíp metód 8
2.2.3 Textúra nerovnosti povrchu 9
2.2.4 TBN matica 11
2.3 Parallax mapping 13
2.3.1 História 13
2.3.2 Princíp metódy 13
2.3.3 Výpočet textúrovacích súradníc 14
2.3.4 Metóda offset limiting 15
2.4 Parallax occlusion mapping 17
2.4.1 História 17
2.4.2 Princíp metódy 17
2.4.3 Výpočet textúrovacích súradníc 18
2.4.4 Hard shadows a Soft shadows 21
2.5 Iterative parallax mapping 23
2.5.1 História 23
2.5.2 Princíp metódy 23
2.5.3 Výpočet textúrovacích súradníc 24
3 Implementácia 25
3.1 Implementácia metód 25
3.1.1 Phongovo osvetlenie 25
3.1.2 Normal mapping 26
3.1.3 Parallax mapping a Iterative Parallax mapping 28
3.1.4 Parallax Occlusion Mapping so soft shadows 28
3.2 Vlastná metóda siluet 31
3.2.1 Účel metódy 31
3.2.2 Návrh metódy 31
3.3 Graficko-herná aplikácia 33
3.3.1 Vytvorenie modelov a animácií 33
3.3.2 Implementácia hráča a kostlivca 34
3.3.3 Editor STAGE 35
4 Porovnanie metód 37
5 Záver 39
Literatúra 40

1 Úvod

Grafická disciplína sa už od počiatku svojej existencie snaží čo najdetailnejšie zobraziť predmety okolo nás. Roky vývoja rôznych technológií priniesli schopnosť zobrazenia okolitého sveta do virtuálneho 3D priestoru. Keďže ide o zložitý proces, preto sa začal vývoj uberať aj smerom zrýchľovania týchto algoritmov, aby sa scéna dala zobraziť v „reálnom čase“ (to znamená minimálne 25 krát za sekundu). Začali vznikať technológie, ktoré sa snažia nájsť najlepší kompromis medzi zobrazením objektu čo najreálnejšie a pritom čo najrýchlejšie. Jedna vetva týchto technológií sa zaoberá simuláciou nerovností povrchu jednoduchého plochého objektu, bez zmeny geometrie.

Táto práca poskytuje prehľad najznámejších takýchto algoritmov a popisuje ich fungovanie a implementáciu predovšetkým v hernom priemysle. Konkrétne sa zaoberá hlavne metódam bump mappingu, parallax mappingu a príbuznými technológiami. Demonštračné predvedenie týchto metód je uskutočnené v jednoduchom hernom projekte a na koniec sa práca venuje aj rýchlostnému porovnaniu implementovaných metód.

2 Metódy simulácie zakrivenia povrchu

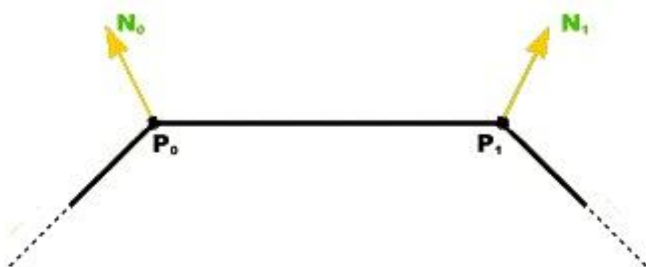
2.1 Phongovo osvetlenie

2.1.1 História

Táto metóda nepatrí úplne medzi simulácie zakrivenia povrchu, i keď ponúka isté náznaky takejto techniky. Predovšetkým ale ponúka vysvetlenie ako sa v 3D priestore so svetlom pracuje, čo je dôležité pre výklad ďalších kapitol. Základná idea akéhokoľvek osvetlenia je stmaviť od zdroja svetla odklonené časti modelu a naopak zosvetliť časti, ktoré sú priklonené. Veľmi realistické osvetlenie priniesol roku 1973 pán Bui Tuong Phong [Phong 1973]. Metóda Phongovho osvetlenia bola publikovaná ako dizertačná práca pána Phonga na univerzite v Utahu, ktorá na svoju dobu bola považovaná verejnosťou za radikálnu. Dnes však táto metóda (a jej ďalšie modifikácie) sa považuje skôr za jednu z hlavných metód a používa ju väčšina moderných grafických aplikácií. Phongove osvetlenie je vylepšenie staršieho osvetlenia menom Gouraudovo osvetlenie [Gouraud 1971]. Phongova metóda osvetlenia spočíva v dvoch jeho ideách a to vo Phongovej interpolácii a vo Phongovom osvetľovacom modely [Phong 1973].

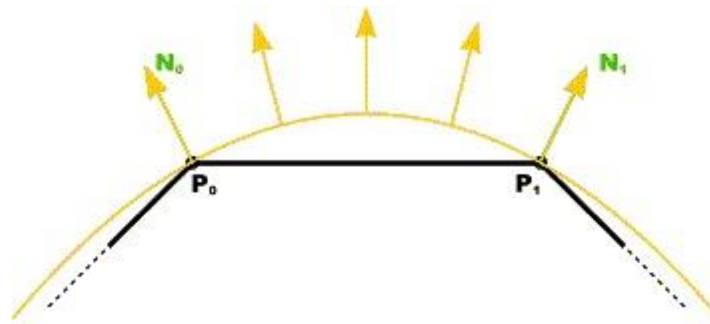
2.1.2 Phongova interpolácia

Ako je vyššie uvedené, množstvo dopadajúceho svetla je úmerné nakloneniu časti modelu k zdroju svetla. Geometrické vlastnosti modelu umožňujú pre každý jeho vrchol vypočítať jeho normálový vektor, ktorý udáva kolmý smer od vrcholu vzhľadom na povrch modelu.



Obrázok 2.1: Normálové vektory vo vrcholoch modelu [csee.umbc.edu]

Normálovými vektormi sa dá popísať sklon každého vrcholu voči súradnicovým osám a tým pádom aj voči svetlu. Na základe tohto faktu, je možné vypočítať dopadajúce svetlo na každý vrchol a tým je udaný základ pre Gouaraudovo osvetlenie. Na rozdiel od Gouaraudova osvetlenia, ktoré pri rasterizácii interpoluje osvetlenú farbu vrcholov, Phongova metóda vôbec nepočíta svetlo pre vrchol ale interpoluje z vrcholov normálové vektory. Touto operáciou sa dostane lineárne interpolovaná normála v danom rasterizovanom bode na povrchu polygónu, ktorá sa následne normalizuje [Phong 1973], [Gouraud 1971].



Obrázok 2.2: Interpolované normály [csee.umbc.edu]

Ako je na obrázku vidno, pre rovný povrch polygónu je udané množstvo rôznych normálových vektorov, ktoré popisujú virtuálne zaoblenie polygónu. Pri rasterizácii sa práve takto vypočítaný normálový vektor stáva základom pre Phongov osvetľovací model.

2.1.3 Phongov osvetľovací model

Phongov osvetľovací model popisuje výpočet svetla za pomoci interpolovanej normály. Základným princípom Phongovho osvetlenia je pri rasterizácii polygónu zosvetliť (alebo stmavnúť) a pridať (alebo ubrať) lesk do výslednej farby pixelu. Phongov osvetľovací model sa teda skladá dokopy z troch hlavných zložiek [Phong 1973].

Prvou je **ambient** zložka, ktorá udáva konštantné minimálne osvetlenie bodu a nie je závislá na množstve priameho dopadajúceho svetla. Výsledná intenzita bude teda mať minimálne ambient osvetlenie, či už je polygón odklonený alebo priklonený k svetlu. Táto zložka slúži na simulovanie viacnásobného odrazu svetla tam, kde priame svetlo nedopadá [Phong 1973].

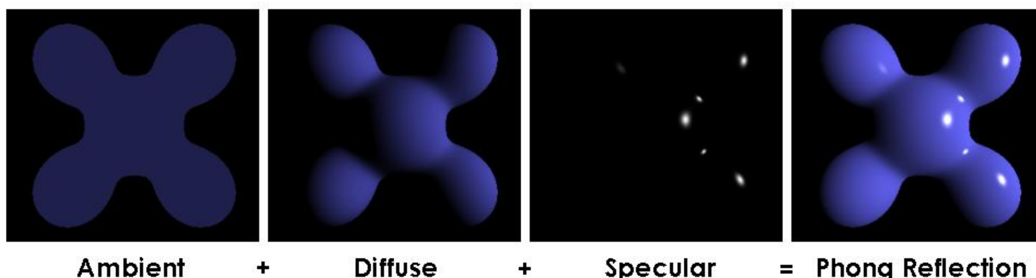
Ďalšou je **diffuse** zložka, ktorá vyjadruje na základe sklonu povrchu (normálového vektoru) množstvo dopadajúceho svetla na rasterizovaný bod a jeho rovnomerné rozptýlenie všade do priestoru. Touto zložkou sa teda vo Phongovom modelu vypočíta zväčšenie alebo zmenšenie výslednej intenzity [Phong 1973].

Treťou, poslednou zložkou je **specular**. Táto zložka jediná pracuje s vektorom k pozorovateľovi a hovorí o tom, koľko sa priamo týmto smerom odrazí svetla od povrchu. Zložka tým pádom popisuje lesklosť rasterizovaného bodu na polygóne [Phong 1973].

Na záver sa podľa nasledujúceho vzorca vypočíta výsledná intenzita **I** svetla v danom rasterizovanom bode [Phong 1973]:

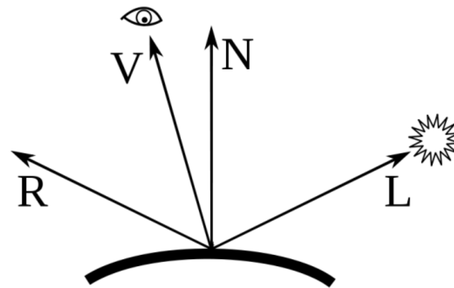
$$I = ambient + diffuse + specular$$

Nasledujúci obrázok ilustruje všetky zložky Phongovho osvetľovacieho modelu a konečný výpočet intenzity svetla:



Obrázok 2.3: Zložky Phongovho osvetľovacieho modelu a výpočet intenzity [wikipedia.org]

Ako bolo vyššie naznačené, pre výpočty niektorých zložiek je nutnosť vedieť vektory, s ktorými Phongov osvetľovací model počíta. Obrázok nižšie znázorňuje všetky potrebné normalizované vektory pre korektný výpočet osvetlenia (obrázok je upravený a prevzatý z wikipedia.org).



Obrázok 2.4: Vektory potrebné pre Phongove osvetlenie

$\vec{N}$ – Normálový vektor k povrchu
$\vec{L}$ – Vektor, ktorý udáva smer z bodu na povrchu k zdroju svetla
$\vec{R}$ – Vektor priameho odrazu svetla od povrchu (lesk)
$\vec{V}$ – Vektor smerujúci z bodu na povrchu k pozorovateľovi

Vektory $\vec{L}$ a $\vec{V}$ sa dajú vypočítať ľahko a to odpočítaním súradníc bodu na povrchu polygónu od súradníc pozície svetla (pre vektor $\vec{L}$) alebo od súradníc pozície kamery (pre vektor $\vec{V}$). Na rozdiel od toho vektor $\vec{N}$ je výsledkom výpočtu Phongovej interpolácie. Vektor $\vec{R}$ je vyjadrený operáciou, ktorá je uvedená na nasledujúcom vzorci, pričom znak " · " značí skalárny súčin [Phong 1973]:

$$\vec{R} = 2(\vec{L} \cdot \vec{N})\vec{N} - \vec{L}$$

Pred samotným výpočtom jednotlivých zložiek je ešte nutné zaviesť definíciu materiálu **M**, ktorý je na rasterizovanom polygóne.

M_a – ambient konštanta materiálu udáva farbu v nepriamom svetle
M_d – diffuse konštanta materiálu popisuje farbu v priamom svetle
M_s – specular konštanta materiálu značí farbu odlesku
M_e – specular exponent materiálu (tzv. Shininess) určuje lesklosť materiálu. Pri vysokej hodnote exponentu sú odlesky menšie a naopak pri nízkej hodnote odlesky rastú.

Keďže ambient zložka nesúvisí nijako s priamym svetlom a ani s pozorovateľom, na jej výpočet nie je nutný žiaden z vyššie uvedených vektorov. Vypočíta sa vynásobením M_a konštanty materiálu s L_a konštantou, ktorá udáva ambient zložku svetla [Phong 1973] [Stehlík 2007].

$$ambient = M_a * L_a$$

Ďalšia zložka, diffuse, je závislá na vektoroch $\vec{N}$ a $\vec{L}$. Intenzita tejto zložky sa vypočíta vynásobením kosínusu uhlu týchto dvoch vektorov s konštantami M_d a L_d, kde L_d je konštantou pre vyjadrenie farby priameho svetla [Phong 1973] [Stehlík 2007]..

$$diffuse = M_d * L_d * \vec{N} \cdot \vec{L}$$

Skalárny súčin dvoch normalizovaných vektorov (v tomto prípade $\vec{N}$ a $\vec{L}$) vyjadruje kosínusovú hodnotu uhla medzi nimi. Teda čím uhol medzi dopadajúcim svetlom a normálou povrchu je menší (povrch je viac naklonený ku svetlu), tým kosínus uhla rastie a zložka diffuse dosiahne väčšej intenzity.

Specular zložka pracuje s vektormi $\vec{V}$ a $\vec{R}$ a opäť práve s ich skalárnym súčinom. Podobne ako pri diffuse zložke sa kosínus násobí s M_s a L_s, pričom L_s konštanta popisuje farbu lesku svetla. Avšak do výpočtu je potrebné zahrnúť aj M_e a tým pádom sa vzorec pre výpočet skalárnej zložky trochu odlišuje od predchádzajúceho výpočtu [Phong 1973] [Stehlík 2007]:

$$specular = M_s * L_s * (\vec{V} \cdot \vec{R})^{M_e}$$

Doposiaľ bol vysvetlený Phongov osvetľovací model práve pre jedno svetlo a počítalo sa s tým, že svetlo je nekonečné (jeho intenzita je v každej vzdialenosti rovnaká). Pri väčšom množstve svetiel sa základný Phongov vzorec musí trochu modifikovať:

$$I = ambient + \sum_{t \in P} (diffuse_t + specular_t)$$

Keďže diffuse a specular zložka počítajú s nejakou vlastnosťou svetla, je potrebné tieto zložky vypočítať pre všetky svetlá. Naopak ambient zložka sa počíta iba raz. V uvedenom vzorci symbol *P* znázorňuje počet všetkých svetiel v scéne.

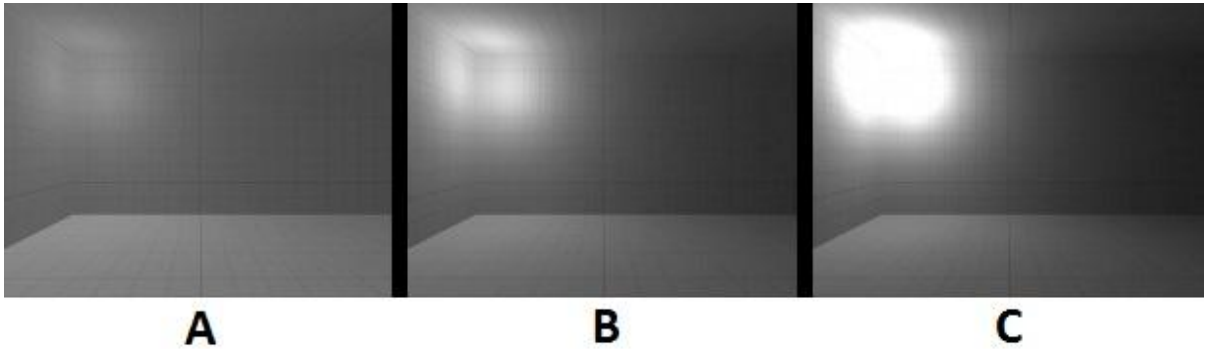
Treba však zmieniť, že existujú aj iné ako nekonečné svetlá a tým pádom je prirodzené, že s narastajúcou vzdialenosťou od svetla sa intenzita zmenšuje. Tým je zavedená nová konštanta *attenuation,* ktorá popisuje útlm svetla a ďalej upravuje Phongov vzorec [Phong 1973]:

$$I = ambient + \sum_{t \in P} (attenuation_t * (diffuse_t + specular_t))$$

Existuje veľa rôznych metód výpočtu attenuation, ale najčastejšie sa používa kombinácia takzvaného konštantného, lineárneho a štvorcového útlmu. Táto metóda poskytuje široké spektrum možností pre nastavenie útlmu svetla [Phong 1973].

$$attenuation = \frac{1}{const + linear * dist + quadratic * dist^2}$$

Kde *dist* je vypočítaná vzdialenosť od bodu na povrchu polygónu k zdroju svetla. Ďalej *const* udáva konštantný útlm svetla, ktorý je jediný nezávislý od vzdialenosti *dist*. Premenná *linear* popisuje lineárny útlm svetla a následne *quadratic* vyjadruje štvorcový útlm. Tieto konštanty sa udávajú pre každé svetlo v scéne. Nasledujúci obrázok demonštruje porovnanie, keď je nastavený iba konštantný, lineárny alebo štvorcový útlm (obrázok je upravený a prevzatý z developer.valvesoftware.com).

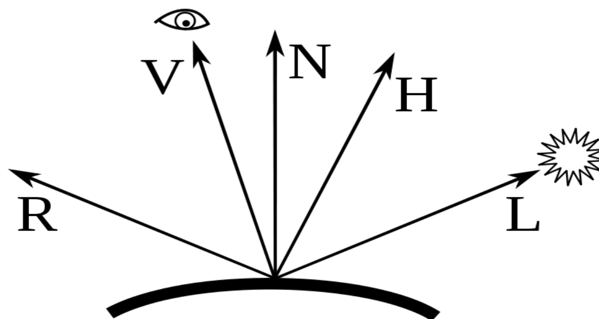


Obrázok 2.5: Porovnanie konštantného (A), lineárneho (B) a štvorcového (C) útlmu svetla

2.1.4 Blinn-Phong osvetľovací model

V dnešnej dobe asi najpoužívanejšou modifikáciu Phongova osvetlenia je Blinn-Phong osvetľovací model. Tento model navrhol pán James F. Blinn [Blinn 1977] a v roku 1977 na konferencii SIGGRAPH '77 ho prvý krát predstavil verejnosti.

Implementačný rozdiel Phongova modelu a modelu pána Blinna je vo výpočte spekulárnej zložky osvetlenia. Phongov model spekulárnu zložku počítal na základe vektora priameho odrazového svetla $\vec{R}$. Keďže výpočet tohto vektora je pomerne zložitý a náročný na čas, pán Blinn došiel s myšlienkou tento vektor vôbec nepočítať. Namiesto neho sa v Blinnovom modelu počíta vektor $\vec{H}$ [Blinn 1977] (obrázok je upravený a prevzatý z wikipedia.org).



Obrázok 2.6: Vektory použité v Blinn-Phongovom osvetľovacom modelu

Výpočet vektoru $\vec{H}$ je podstatne jednoduchší ako výpočet vektoru $\vec{R}$. Vektor $\vec{H}$ je udaný tak, že polí vektory $\vec{L}$ a $\vec{V}$, a teda sa vypočíta nasledujúcim vzorcom [Blinn 1977]:

$$\vec{H} = \frac{\vec{L} + \vec{V}}{|\vec{L} + \vec{V}|}$$

Výsledná zložka specular teda nie je závislá od uhlu medzi vektormi $\vec{R}$ a $\vec{V}$, ale od uhlu medzi vektormi $\vec{N}$ a $\vec{H}$ [Blinn 1977].

$$specular = M_s * L_s * (\vec{N} \cdot \vec{H})^{M_e}$$

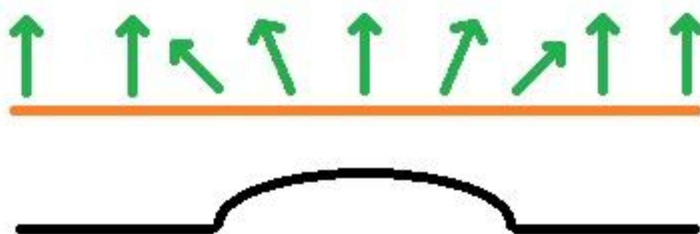
2.2 Bump mapping a normal mapping

2.2.1 História

Bump mapping je spomedzi techník simulujúcich úpravu povrchu historicky najstaršia. Jej implementácia a myšlienky priniesli prevrat v zobrazovaní 3D objektov. Technológia bola predvedená po prvý krát už v roku 1978 na konferencii SIGGRAPH, pánom James F. Blinnom [Blinn 1978]. V hernom priemysle sa ale začala používať až približne o 20 rokov neskôr. Dôvodom bol nedostatočne výkonný a programátorovi uzavretý hardvér v grafických kartách, aby zvládol a zároveň umožnil prepočet v reálnom čase na bežných PC. Niektorí vývojári stavali špeciálny hardvér pre výpočet Bump mappingu, iní keď po čase bol dostupný viacpriechodový hardvér (multipass), realizovali metódu na bežne dostupných grafických kartách. V roku 1984 pán R. L. Cook [Cook 1984] definoval takzvanú normál mapu a tým spravil základy pre normal mapping, ktorého koncepcia vychádza z Bump mappingu a používa sa dodnes. V druhej polovici 90-tých rokoch sa táto technológia vyskytla ako rozšírenie pre OpenGL pod názvom Dot3 Bump mapping a masívne používanie začalo s príchodmi programovateľných shaderov.

2.2.2 Princíp metód

Ako už sám názov napovedá, ide o mapovanie vypuklín. Jedná sa o simuláciu bežných nerovností telesa (ako sú napríklad rôzne ryhy, vypukliny alebo diery) na rovný povrch polygónu. V predchádzajúcej kapitole bolo uvedené ako sa so svetlom pracuje v 3D priestore a definoval sa pojem normálový vektor, ktorý popisuje sklon povrchu. Pri aplikácii Phongovho osvetľovacieho modelu [Phong 1973] sa normála z vrcholov interpolovala, čiže bola závislá od geometrie telesa. Princíp metód bump mappingu a normal mappingu spočíva v myšlienke možnosti upraviť normálový vektor v danom rasterizovanom bode tak, aby popisoval simulovaný povrch (ryhy, vypukliny,...), ktorý by mal polygón predstavovať. Tento proces ilustruje nasledujúci obrázok:



Obrázok 2.7: Normálové vektory simulujúce nerovnosť povrchu

Na obrázku je možné pozorovať oranžovou farbou rovný povrch polygónu. Ďalej čiernou farbou nerovnosť, ktorú chceme simulovať na povrchu a zelenou farbou vypočítané výsledné normály znázorňujúce nerovnosť povrchu. Za pomoci takto upravených normál sa pri osvetľovaní vypočíta iná intenzita svetla na rôznych častiach povrchu polygónu. Záleží na tom, či je normála odklonená alebo priklonená k svetlu a od pozície pozorovateľa (viď kapitolu 2.1.3). Tým pozorovateľ pri výslednom zobrazení nadobúda dojem, ako keby že sa vypuklina skutočne geometricky nachádza na telese, pričom ku žiadnej zmene geometrie nedošlo [Blinn 1978].

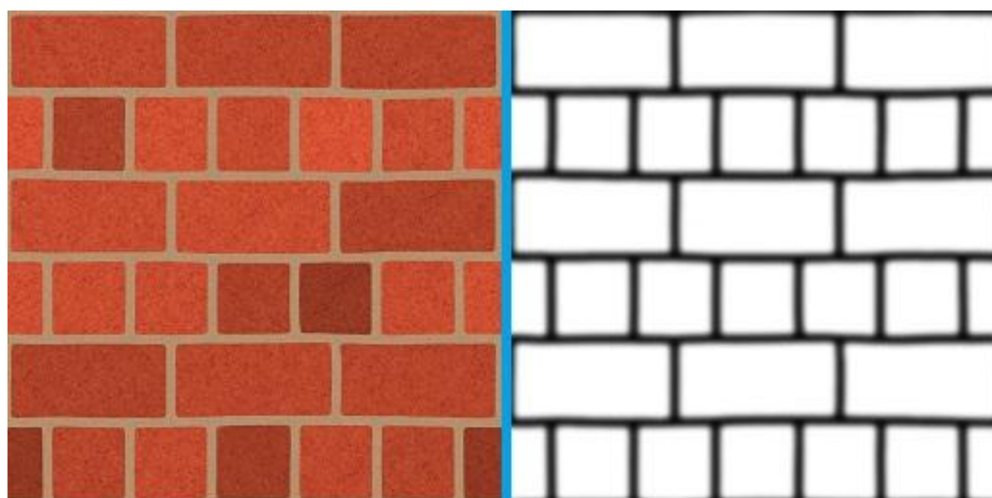
Obrázok 2.8: Porovnanie Phongova osvetlenia a bump mappingu [chromesphere.com]

Vyššie uvedený obrázok predstavuje aplikovanie bump mappingu na guľový povrch telesa. Guľa vykreslená len Phongovým osvetlením (vľavo) pôsobí geometricky podstatne jednoduchšie ako guľa vykreslená s pridaním bump mappingu (vpravo), pričom počet polygónov telesa sa nezmenil. Touto technológiou sa dá docieliť aby niektoré modely, ktoré majú malý počet polygónov, sa zobrazovali približne v rovnakej kvalite ako geometricky prepracované modely, ktoré znázorňujú to isté teleso.

Spôsob ako definovať pre polygón nerovnosti na simuláciu jeho povrchu sa uskutočňuje za pomoci špeciálnej textúry. Na jej základe sa modifikuje v danom rasterizovanom bode normála, ktorá následne ovplyvní výpočet intenzity svetla v danom bode. Textúra nerovnosti povrchu a následný výpočet normály je práve to, v čom sa tieto dve metódy pre simuláciu zakrivenia povrchu líšia.

2.2.3 Textúra nerovnosti povrchu

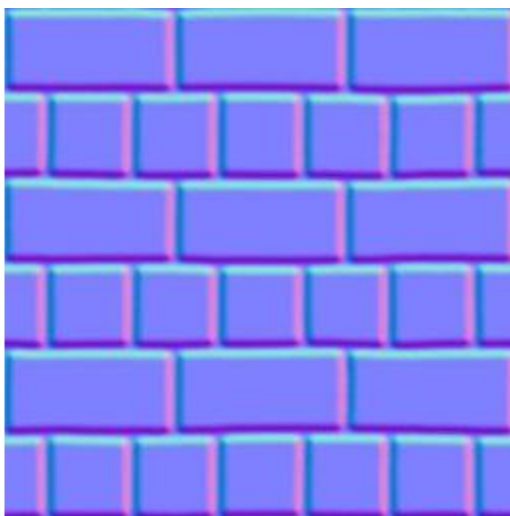
Počas histórie sa najčastejšie používali dva typy textúr pre záznam nerovnosti povrchu. Prvý typ textúry uviedol pán Blinn [Blinn 1978] vo svojom bump mappingu a vtedy bol tento typ nazvaný podľa názvu technológie, a teda bump mapa. Dnes sa však už častejšie používa názov height mapa, ktorý je len synonymum pre bump mapu. Tento typ textúry sa dá definovať ako jednokanálová textúra, kde obvyklá veľkosť kanálu je štandardná a to 1B, teda 256 kombinácií. V súčasnosti sa ale implementuje častejšie do obyčajnej trojkanálovej textúry, ktorá má všetky kanály rovnaké. Je to neefektívne na pamäť, avšak takéto textúry sú ľahšie editovateľné a sú kompatibilné s ostatnými textúrami. Prípadne sa použije štvorkanálová textúra, ktorá využije prvé tri kanály na iný účel a height mapa sa vloží do 4. kanálu. Pri zobrazení height mapy, ako obrázku, sa najčastejšie používa trojkanálové zobrazenie, a teda textúra je v odtieni šedej farby.



Obrázok 2.9: Príklad farebnej textúry a k nej prislúchajúcej bump mapy [encelo.netsons.org]

Na obrázku je možné vidieť klasickú farebnú textúru (vľavo), ktorá sa aplikuje na model steny a jej bump mapu (vpravo), ktorá udáva simulovanú hĺbku rýh medzi tehličkami. Konkrétny bod bump textúry znázorňuje výšku na danej pozícii na povrchu polygónu. Zvyčajne, čím je hodnota bodu textúry (R, G alebo B) vyššia, tým je aj simulovaný povrch v danom bode „vzdialenejší" od polygónu. Z toho vyplýva, že belšie časti textúry budú predstavovať vyššie položené miesto na povrchu. Vypočítanie normálového vektora z bump mapy prebieha za pomoci gradientu mapy v danom bode. Keďže tento výpočet normály sa v dnešnej dobe už moc nepoužíva, v tejto práci nie je uvedený. Kompletný výpočet sa dá prečítať v [Blinn 1978].

Druhý typ textúry na záznam nerovnosti povrchu bol uvedený pánom Cookom [Cook 1984] a začal sa používať v novšom normal mappingu. Ten používa namiesto height mapy takzvanú normál mapu a jej definícia spočíva v 3 kanáloch namiesto jedného kanálu. Všetky tri kanály zvyčajne bývajú 1B, teda 256 kombinácií. Princíp normál mapy nie je v rôznych výškach povrchu, ako to bolo pri height mape, ale každý bod textúry prezentuje už vypočítaný normálový vektor v danom bode na povrchu. Hodnoty RGB udávajú súradnice XYZ normály v polygónovom tečnom priestore, ktorý predstavuje súradnicový systém vzťahujúci sa k povrchu. Takto definovanou textúrou teda nie je nutné počítať normálu podľa gradientu, ako to bolo pri bump mappingu [Blinn 1978].



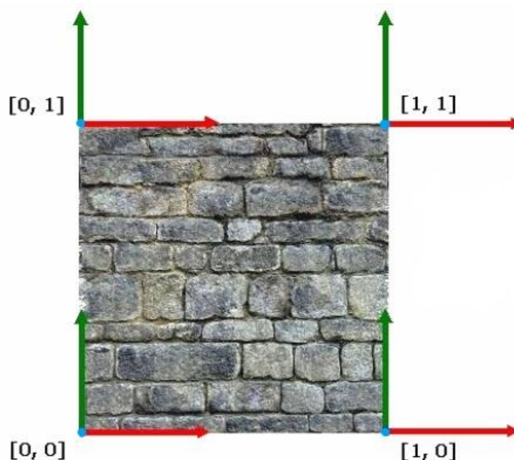
Obrázok 2.10: Príklad normál mapy [flickr.com]

Ako je na obrázku vidno, normál mapa obsahuje hlavne modrú farbu. Dôvod je taký, že väčšina normálových vektorov smeruje od povrchu polygónu a teda je udaná hlavne hodnota Z (pre textúru hodnota B). Uvedená normál mapa vyjadruje rovnaké zakrivenie povrchu ako bump mapa z obrázku číslo 2.9.

Avšak pri takomto zadaní normál vzniká problém, pretože zadávaný normálový vektor cez normál mapu má pozíciu v tečnom priestore. Na rozdiel od neho vektor pohľadu a svetla sú vypočítané v modelovom priestore (súradnicový systém vzťahujúci sa k pozícii modelu). To znamená, že výpočet osvetlenia za pomoci takto vyjadrených vektorov by bol nekorektný a viedol by k nesprávnemu zobrazeniu nerovnosti povrchu. Pre riešenie tohto problému sa používa prevod vektorov medzi modelovým priestorom a tečným priestorom, ktorý prebieha za pomoci takzvanej TBN matice [Blinn 1978] [Stehlík 2007].

2.2.4 TBN matica

Tečný priestor predstavuje súradnicový systém vzťahujúci sa k povrchu polygónu. Tento priestor je potrebné matematicky popísať, aby sa dali s ním uskutočniť adekvátne operácie. Preto pre každý vrchol polygónu sa vypočítajú 3 potrebné vektory, a to normálový vektor, tečna a binormála.



Obrázok 2.11: Vektory tečného priestoru [Stehlík 2007]

Uvedené vektory sú na obrázku vidieť pod modrou (normála N), zelenou (binormála B) a červenou (tečna T) farbou. Následne v hranatých zátvorkách sa nachádzajú textúrovacie súradnice v danom vrchole polygónu. Obrázok znázorňuje, že všetky tri vektory sú na seba kolmé a každý z nich predstavuje jednu os v tečnom priestore. V tomto rozpoložení teda tvoria bázu tohto priestoru pre daný vrchol polygónu. Jednotlivé vektory sú vypočítané podľa nasledujúcich vzorcov (pre jeden trojuholník) [Gath 2006] [Stehlík 2007]:

$$\vec{T} = \frac{1}{M} * (\Delta t3t1_{(v)} * \Delta v2v1 - \Delta t2t1_{(v)} * \Delta v3v1)$$

$$\vec{B} = \frac{1}{M} * (-\Delta t3t1_{(u)} * \Delta v2v1 + \Delta t2t1_{(u)} * \Delta v3v1)$$

Uvedené premenné vo vzorcoch znamenajú:

$\Delta t3t1_{(v)}$ – rozdiel textúrovacích súradníc V medzi vrcholom 3 a vrcholom 1
$\Delta v2v1$ – vektor smerujúci z vrcholu 1 do vrcholu 2
$\Delta t2t1_{(v)}$ – rozdiel textúrovacích súradníc V medzi vrcholom 2 a vrcholom 1
$\Delta v3v1$ – vektor smerujúci z vrcholu 1 do vrcholu 3
$\Delta t3t1_{(u)}$ – rozdiel textúrovacích súradníc U medzi vrcholom 3 a vrcholom 1
$\Delta t2t1_{(u)}$ – rozdiel textúrovacích súradníc U medzi vrcholom 2 a vrcholom 1
$M = \Delta t2t1_{(u)} * \Delta t3t1_{(v)} + \Delta t3t1_{(u)} * \Delta t2t1_{(u)}$

Navzájom kolmosť všetkých troch vektorov tečného priestoru poskytuje výpočet tretieho vektoru, v tomto prípade normálového vektoru, jednoduchším spôsobom. Stačí medzi $\vec{T}$ a $\vec{B}$ previesť operáciu vektorového súčinu [Gath 2006] [Stehlík 2007]:

$$\vec{N} = \vec{T} \times \vec{B}$$

Po vykonaní výpočtov sa dá z týchto troch vektorov vyjadriť TBN Matica, ktorej stĺpce sú tvorené v poradí tečnou, binormálou a normálou:

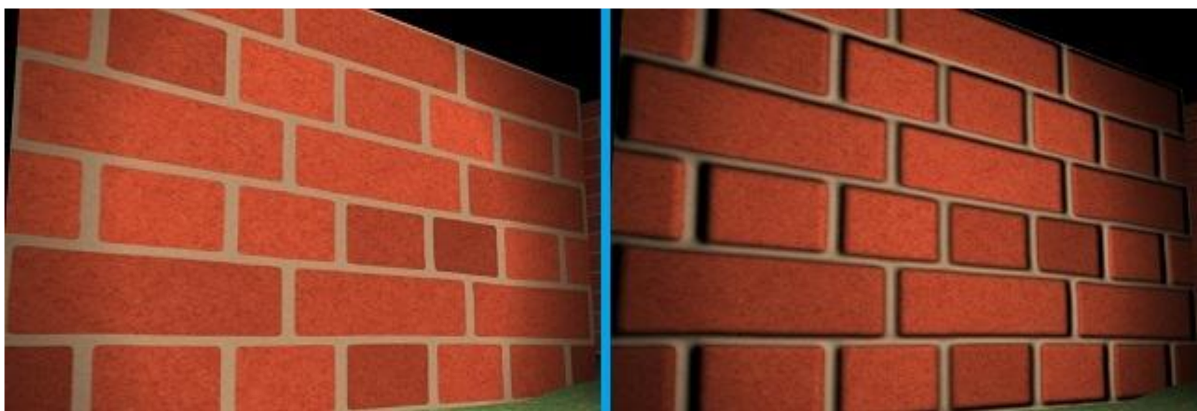
$$\begin{pmatrix} T_{(x)} & B_{(x)} & N_{(x)} \\ T_{(y)} & B_{(y)} & N_{(y)} \\ T_{(z)} & B_{(z)} & N_{(z)} \end{pmatrix}$$

Veľká výhoda je, že prepočítanie zložiek TBN matice sa nemusí uskutočniť pri každom cykle renderovania. Vo veľa prípadoch je možné normály, tečny alebo binormály prepočítať na začiatku programu a ďalší krát už výpočet nie je nutný. Či TBN maticu prepočítať, alebo nie, určuje ako sa s modelom manipulovalo. Pri statických objektoch, ktoré nemenia svoju geometriu a prípadne sa len pohybujú, maticu stačí vypočítať len raz. Avšak pri rotácii modelu alebo zmene jeho geometrie, je nutné po každej takejto činnosti znova prepočítať zložky matice [Gath 2006] [Stehlík 2007].

TBN matica kompletne popisuje tečný priestor pre vrchol polygónu. Ako bolo vyššie spomenuté, táto matica má slúžiť na prevod vektora z modelového priestoru do tečného priestoru. Prevod sa uskutoční jednoduchým násobením prevádzaného vektora s TBN maticou. Teraz je možné pri normal mapingu previesť vektor pohľadu a vektor svetla do rovnakého priestoru ako je uvedený normálový vektor z normál mapy. Po prevode je možné uskutočniť korektné osvetlenie povrchu napríklad s Phongovým osvetlením [Phong 1973] a správne realizovať efekt.

Dá sa teda povedať, že hlavný rozdiel medzi Bump mappingom a Normal mappingom je množstvo výpočtov a pamäťové nároky. V dnešnej dobe sa používa prednostne normal mapping. Hlavne kvôli hernému priemyslu sa veľmi dbá na rýchlosť a efektivitu výpočtov pri zobrazovaní 3D priestoru, ale na úkor väčšieho množstva pamäte. Dôraz sa predovšetkým kladie na vykreslenie scény v reálnom čase.

Na záver tejto kapitoly si ešte ukážeme aplikovanie normal mappingu na textúru tehličiek uvedenú v obrázku číslo 2.9. Obrázok vľavo je vykreslený len Phongovým osvetlením a na pravo je možné vidieť efekt normál mapingu:



Obrázok 2.12: Ukážka aplikovania normal mapingu

2.3 Parallax mapping

2.3.1 História

Normal mapping sa od jeho publikácie stal ohromne populárnym a masívne sa začal používať. Je to spôsobené hlavne aj rýchlym technickým pokrokom v tom období. Technika rýchlo napredovala a onedlho po normal mappingu sa objavila ďalšia zaujímavá myšlienka ako ešte zdokonaliť simuláciu nerovností povrchu. Na konferencii ICAT 2001 v Tokiu s ňou prišiel pán Tomomichi Kaneko [Kaneko 2001] a uviedol ju s názvom parallax mapping. Táto technika poňala výpočet krivosti z úplne iného uhlu pohľadu ako tomu bolo doposiaľ. Parallax technológia však nie je náhrada za normál mapping, ale ho len doplňuje. Táto metóda neskôr dostala aj dve ďalšie mená a to offset texture mapping a virtual displacement mapping. Neskôr metóda pána Kaneka bola upravená pánom T. Welshom [Welsh 2004]. Táto upravená metóda nesie názov parallax mapping with offset limiting a jej princípy sú vysvetlené taktiež v tejto kapitole.

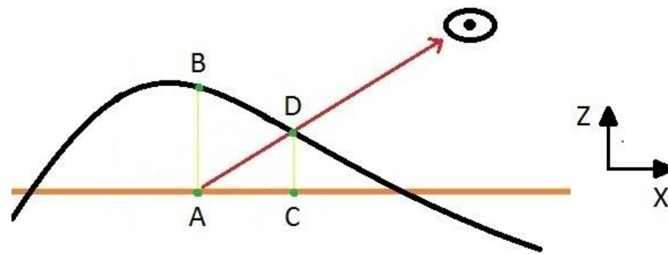
2.3.2 Princíp metódy

Za pomoci normal mappingu a osvetlenia je možné dodať objektu veľmi kvalitný a reálny vzhľad. Avšak je to technológia, ktorá pracuje len s pridávaním a uberaním intenzity svetla v bodoch. Za predpokladu zobrazovania modelu bez lesku je možné povedať, že s použitím normal mappingu sa pri pohľadoch na model z rôznych uhlov výsledný povrch vôbec nemení. Tento fakt je pri predstave napríklad pohľadu na stenu s vypuklými tehličkami nereálny. Na takejto stene by sa pri rôznych pohľadoch mala nejaká strana tehličky „ukázať“ a naopak iná „schovať“. Takýto jav je za pomoci normal mappingu nemožný, avšak parallax mapping ponúka elegantné riešenie.

2.13: Porovnanie: parallax mapping (vľavo) a Phongove osvetlenie (vpravo) [hardwareheaven.com]

Táto metóda používa na zadávanie nerovností povrchu height mapy [Blinn 1978] (viď kapitola 2.2.3). Býva však z pravidla kombinovaný aj s normal mappingom, takže pri implementácií sa nachádza aj height mapa a aj normál mapa [Cook 1984]. Parallax síce používa rovnakú reprezentáciu nerovnosti povrchu ako bump mapping, ale výpočty s touto height mapou sú absolútne rozdielne. Keďže táto technológia počíta nielen s normálamy, ale berie v úvahu aj výškové rozdiely na povrchu (vypuklosť tehličiek), je kombinácia normal mapy a height mapy pre uloženie nerovností ideálna. Preto sa v mnoho prípadoch implementuje jedna štvorkanálová textúra, ktorej prvé tri kanály obsahujú informácie pre normal mapping a zvyšný štvrtý kanál slúží pre reprezentáciu height mapy.

Základná idea tejto technológie je za pomoci zmeny textúrovacích súradníc pri rasterizácii docieliť efekt nerovnosti povrchu polygónu aj pri rôznych uhloch pohľadu. Túto myšlienku znázorňuje nasledujúci obrázok čiastočne prevzatý z [Stehlík 2007]:

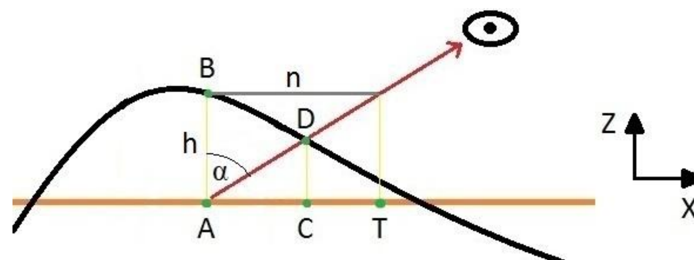


2.14: Základný princíp parallax mappingu

Treba opomenúť, že takto uvedený obrázok zobrazuje princíp parallax mappingu len v dvoch súradniciach (X a Z) a keďže princíp pre Y a Z je úplne totožný, je uvedený len jeden obrázok. Vzápätí treba dodať, že všetky uvedené vektory sú v tečnom priestore polygónu. Na obrázku je možné pozorovať rovný povrch polygónu oranžovou čiarou a simulované zakrivenie čiernou vlnou. Červenou farbou je znázornený smer pohľadu pozorovateľa, ktorý hľadí na bod A. Pri normal mappingu sa teda pozorovateľovi vypočíta výsledná intenzita svetla z bodu B pri aplikovaní textúrovacích súradníc z bodu A. Po správnosti a podľa reálneho vzhľadu by však mal byť zobrazený bod D miesto bodu B. Pre tento účel je nutnosť posunúť textúrovacie súradnice z bodu A do bodu C a až potom previesť výpočet osvetlenia a určenie výslednej farby. Technológia parallax mappingu sa zaoberá práve výpočtom výsledných textúrovacích súradníc, tak aby boli čo najbližšie k bodu C [Kaneko 2001] [Stehlík 2007].

2.3.3 Výpočet textúrovacích súradníc

Ako sa na prvý pohľad nemusí zdať, výpočet výsledných textúrovacích súradníc nie je elementárny úkon. Existuje mnoho metód výpočtov (viď ďalšie kapitoly). Niektoré sú rýchle a nepresné, iné sú pomalé, ale za to veľmi reálne. Prvú a teda základnú metódu pochopiteľne predstavil pán Kaneko [Kaneko 2001] a je najlepšie ilustrovateľná na nasledujúcom obrázku (čiastočne prevzatý z [Stehlík 2007]):



2.15: Výpočet textúrovacích súradníc

Tento obrázok má na rozdiel od obrázka 2.14 niekoľko zmien. Uhol alfa popisuje uhol medzi normálou a pohľadom pozorovateľa. V tomto prípade, keďže sú všetky vektory udané v tečnom priestore, je normála polygónu zvyčajne (0, 0, 1). Následne je možné vidieť premennú *h*, ktorá udáva výšku simulovaného povrchu v danom bode, ktorá je získaná z height. Ďalej sa na obrázku nachádza

šedou farbou vzdialenosť n a bod na povrchu T. Vzdialenosť n udáva veľkosť posunu textúrovacích súradníc od bodu A smerom k pozorovateľovi a tým sa tvorí bod T. Ten udáva výsledný bod parallax mappingu s textúrovacími súradnicami pre ďalšie výpočty ako je napríklad normal mapping alebo Phongovo osvetlenie [Phong 1973]. Na základe týchto informácií a obrázku je možné napísať všeobecný vzorec pre výpočet posunu súradníc. (uvedené vzorce sú upravené z [Kaneko 2001], [Welsh 2004] a [Stehlík 2007]):

$$n = \tan(\alpha) * h$$

Ako bolo vyššie zmienené, premenná h je výška z height mapy. Táto výška je však uvedená v klasickom intervale pre pracovanie s textúrami a to v <0, 1>. V takomto istom intervale sú uvedené aj textúrovacie súradnice a aj všetky vektory v tečnom priestore z obrázku 2.15. Tým pádom height mapa udáva voči celej textúre príliš veľké hodnoty (pre ilustráciu: z obrázka 2.9 by jedna tehlička bola tak vypuklá ako šírka celej textúry) a teda je nutné ich adekvátne upraviť. Túto úpravu vyjadruje následný vzorec:

$$h' = h * scale + bias$$

Na vzorci sú uvedené tri nové premenné. *Scale* označuje koeficient zmenšenia výšky z height mapy a následne *bias* vyjadruje dodatočné konštantné zvýšenie alebo zníženie. Nová výsledná výška je vo vzorci reprezentovaná premennou h'. Za pomoci tohto vzorca je možné akokoľvek doupravovať výšku z height mapy, tak aby vizuálne presne popísala ľubovoľný povrch. Ďalej je možné vyjadriť konečný vzorec pre výpočet posunu textúrovacích súradníc:

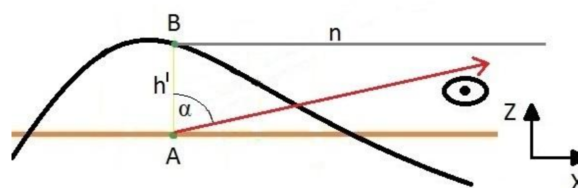
$$n_{(u,v)} = \frac{V_{(x,y)}}{V_{(z)}} * h'$$

kde $V_{(x,y)}$ je X a Y súradnica a $V_{(z)}$ je Z súradnica vektora smerujúceho k pozorovateľovi od rasterizovaného bodu. Napokon stačí vypočítať finálny bod $T_{(u,v)}$ na povrchu, ktorý sa bude používať pre ostatné výpočty po parallax mappingu (normal mapping, Phongovo osvetlenie, atď):

$$T_{(u,v)} = A_{(u,v)} + n_{(u,v)}$$

2.3.4 Metóda offset limiting

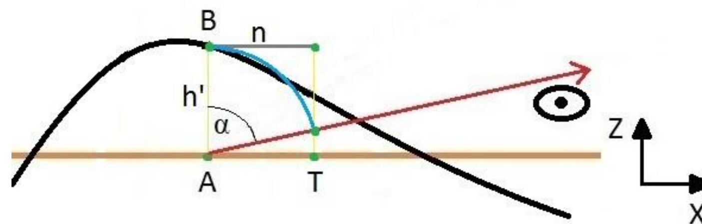
Parallax mapping sám o sebe vykazuje kvalitné výsledky. Avšak po hlbšej analýze sa ukazujú isté nedostatky alebo nepresnosti. Napríklad, pri veľmi veľkej hĺbke simulovaného povrchu by dochádzalo ku nereálnemu posunu súradníc, a teda deformácii výsledného zobrazenia povrchu. Iným z týchto nedostatkov je fakt, že ak by uhol pohľadu k povrchu bol malý a zároveň výška z height mapy veľká, výsledná vzdialenosť nových súradníc by bola príliš vzdialená:



2.16: Ilustrácia problému parallax mappingu (prerobený zo [Stehlík 2007])

Ako je na obrázku vidno, vzdialenosť n sa v žiadnom prípade nepribližuje výslednej hodnote aká by pre tieto vektory mala byť. Pri takomto probléme sa hypotetický posun textúrovacích súradníc v niektorých prípadoch môže dokonca blížiť nekonečnu. Tým pádom pri istých uhloch sa povrch zobrazí absolútne nerealisticky.

Tento problém rieši metóda offset limitingu pána Welsha [Welsh 2004]. Ten pre výsledný posun textúrovacích súradníc zaviedol takzvaný limit, ktorý určuje maximálny možný posun. Hodnotu tohto limitu určil pritom na číslo zodpovedajúce výške v danom bode, čiže premennej h' (kapitola 2.3.3). Aplikovanie limitu je znázornené na nasledujúcom obrázku [Welsh 2004][Stehlík 2007]:

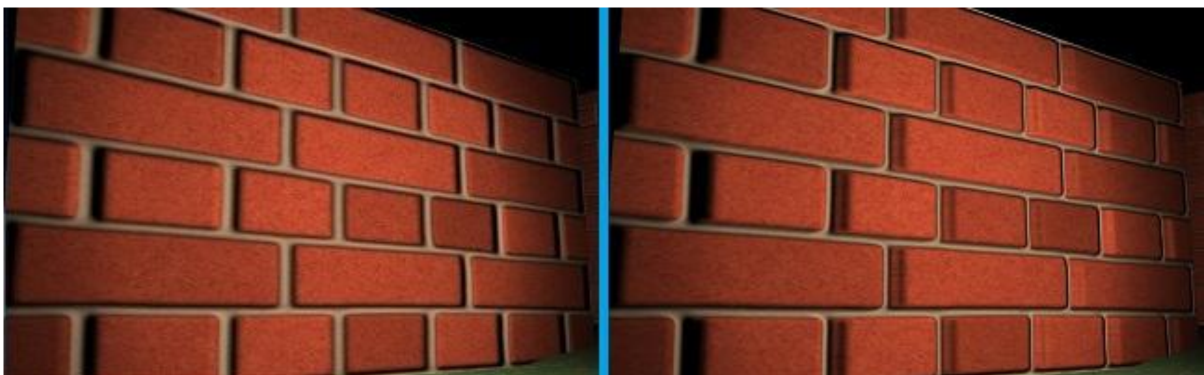


2.17: Aplikácia offset limiting na výpočet posunu súradníc

Na obrázku sa okrem už vysvetlených hodnôt (viď obrázok 2.15 a 2.16) nachádza modrou farbou znázornená aplikácia offset limitingu. Ako je vidno, offset limiting úspešne rieši tento problém parallax mappingu. Pán Welsh určil takúto hodnotu limitu preto, lebo pri čísle h' výsledky boli veľmi uspokojivé a zároveň sa nemusela žiadna ďalšia hodnota počítať, keďže hodnota h' už je v čase aplikovania limitu vypočítaná. Je možné teda aplikovanie limitu a výpočet výsledného posunu popísať matematicky [Welsh 2004] a to úpravou už vyššie uvedenej rovnice (kapitola 2.3.3) pre výpočet posunu textúrovacích súradníc:

$$n'_{(u,v)} = V_{(x,y)} * h'$$

V dnešnej dobe sa z týchto dvoch metód viacej používa modifikácia parallax mapping with offset limiting, avšak hlavná myšlienka parallax mappingu sa nezmenila. V akejkoľvek inej modifikácii parallaxu (niektoré sú uvedené ďalej v tejto práci), ide o zmenu textúrovacích súradníc tak isto ako v metóde pána Kaneka [Kaneko 2001]. Na záver kapitoly si uvedieme ukážku porovnávajúcu vykreslenie tehličiek z obrázka 2.9 s použitím iba normal mappingu (na ľavo) a s pridaním parallax efektu (na pravo):



2.18: Porovnanie vykreslenia s normal mappingom a s pridaním parallax efektu

2.4 Parallax occlusion mapping

2.4.1 História

V roku 2004 priniesla kniha ShaderX3 [Wolfgang 2004] publikáciu technológie s názvom Parallax occlusion mapping (ďalej len POM) od autorov Zoe Brawley a Natalya Tatarchuk. Túto technológiu v zapätí v roku 2005 predstavila Tatarchuk na konferencii SIGGRAPH [Tatarchuk 2005]. V tomto roku boli publikované aj iné podobné technológie a to Steep parallax mapping od Morgan McGuire z Brownovej univerzity [McGuire 2005] a takzvaný Real-time relief mapping od Fábia Policarpa, M. M. Oliveira a J. L. D. Comba [Policarpo 2005], ktorý vychádzal z myšlienok Relief mappingu z roku 2000 od M. M. Oliveira [Oliveira 2000]. Tieto dve nové technológie fungovali v zásade na podobnom princípe ako POM a nie sú veľmi rozdielne. Preto sa táto práca podrobne zaoberá iba POM. Veľkou novinkou v týchto technológiach bola možnosť generovať na povrchu tiene. Výhodou POM bola možnosť vykresľovať tiene za pomoci metódy Soft Shadows (viď kapitola 2.4.4), pričom ostatné metódy používali hlavne Hard Shadows (viď kapitola 2.4.4).

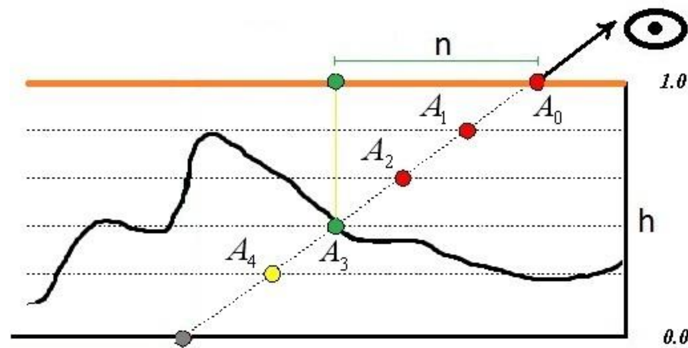
2.4.2 Princíp metódy

Tak ako v obyčajnom parallax mappingu, tak aj v POM je najpodstatnejší výpočet posunu textúrovacích súradníc. V oboch metódach sa výšková nerovnosť povrchu popisuje height mapou. Pri dôkladnejšej analýze obyčajného parallaxu zistíme, že okrem problému, ktorý rieši offset limiting (kapitola 2.3.4) má Kanekova metóda ešte ďalší problém a to presnosť vypočítaného posunu súradníc. Na obrázkoch 2.15 až 2.17 je možné túto nepresnosť pozorovať. POM prichádza s metódou veľmi presného výpočtu textúrovacích súradníc, čím sa dosahuje omnoho kvalitnejších výsledkov.

2.19: Porovnanie POM s klasickým parallaxom a Phongovým osvetlením [cs.utah.edu]

Na obrázku je možné pozorovať kocku vykreslenú len Phongovým osvetlením (vľavo). Ďalej obrázok ponúka porovnanie Kanekovho parallax mappingu [Kaneko 2001] (v strede) s POM (vpravo). Pravý obrázok pôsobí najreálnejšie a demonštruje schopnosti POM vrátane vrhania tieňov.

POM dokáže nové súradnice vypočítať veľmi presne, avšak na úkor času potrebného pre túto operáciu. Pri obyčajnom parallax mappingu stačil na výpočet jeden náhľad do height mapy, pri POM ich je potrebných podstatne viac. Za náhľad sa považuje zistenie farby textúry (mapy) v bode udanom textúrovacími súradnicami a treba upozorniť, že je to jedna z najnáročnejších operácií na čas. V dnešnej dobe s pomerne výkonným hardvérom je POM použiteľný aj v real-time aplikáciách. Princíp samotného výpočtu textúrovacích súradníc za pomoci POM je znázornený na nasledujúcom obrázku (čiastočne prerobený z [Tatarchuk 2006]):



2.20: Princíp Parallax occlusion mappingu

Na obrázku je znázornený princíp POM, ktorý spočíva v postupnom trasovaní vektora pohľadu, a ten je tradične uvedený v tečnom priestore polygónu. Oranžová farba znázorňuje povrch polygónu a čierna vlna stvárňuje nerovnosť povrchu. Symbol *h* predstavuje výšku načítanú z height mapy, ako bolo uvedené v predchádzajúcich kapitolách, načítaná výška je v rozsahu 0.0 až 1.0. Bod A_0 je prvý bod trasovania vektora a zároveň je to rasterizovaný bod na povrchu polygónu. Šedou farbou je určený posledný bod trasovania vo výške 0.0. Úsečka medzi A_0 a šedým bodom sa rozdelí počtom krokov trasovania a tým je vyjadrená množina bodov, ktoré sa budú prechádzať, kým sa nenájde bod prieniku so simulovaným povrchom. Premenná *n* symbolizuje výsledný posun súradníc.

Takto navrhnutá metóda poskytuje využitie nových možností. Keďže sa vektor pohľadu postupne po krokoch prechádza, trasovanie odhalí prvý bod prieniku pri akýchkoľvek uhloch medzi pozorovateľom a povrchom. Ďalej je možné docieliť napríklad, že pod istým uhlom pohľadu sa málo vypuklá tehlička môže „schovať" za viac vypuklú tehličku. Táto možnosť ponúka vykreslenie veľmi podobajúce sa skutočnosti. Rovnako sa touto metódou dá simulovať podstatne väčšia hĺbka ako pri obyčajnom parallax mappingu, a to bez deformácii výsledného vzhľadu. Následne sa v POM veľmi často využíva technológia Level of detail, ktorou sa dá výpočet urýchliť a vzdialenejšie objekty sa budú vykresľovať menej kvalitne. Taktiež dôležitou novinkou pri tejto technológii je možnosť vykresľovania tieňov, čo dodáva výslednému efektu reálnosť.

2.4.3 Výpočet textúrovacích súradníc

Výpočet parallax vektora

Tak ako bolo v predchádzajúcej kapitole zmienené, základný princíp výpočtu textúrovacích súradníc spočíva v trasovaní vektora pohľadu. Tento vektor sa pred použitím musí previesť s pomocou TBN matice (kapitola 2.2.4) do tečného priestoru polygónu. Následne sa podľa tohto vektora určí takzvaný parallax vektor, ktorého súradnice X a Y sú odvodené od vektora pohľadu (len sú v istom pomere zmenšené) a zároveň jeho súradnica Z musí byť maximálna možná výška simulovanej nerovnosti povrchu, čiže 1.0. To sa docieli týmto vzorcom (uvedené vzorce v celej tejto kapitole 2.4.3 sú prevzaté a upravené z [Tatarchuk 2006], [Policarpo 2005], [McGuire 2005]):

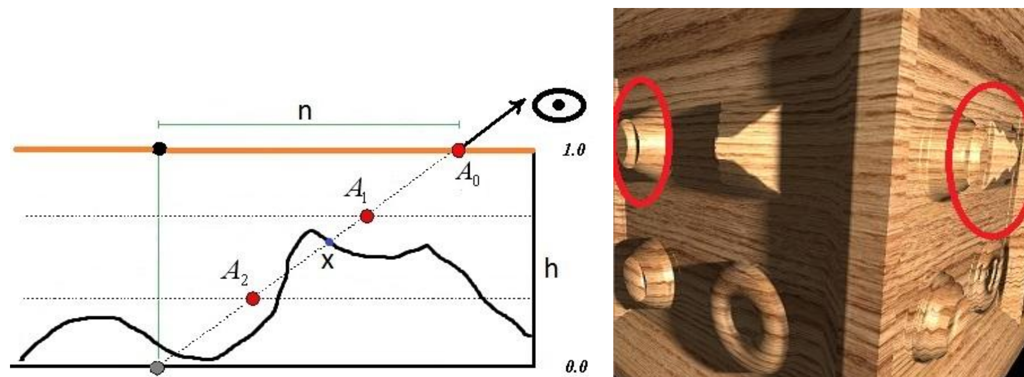
$$\overrightarrow{PV}_{(x,y,z)} = vec3\left(\frac{\vec{V}_{(x,y)}}{\vec{V}_{(z)}} * scale, 1.0\right)$$

Vo vzorci $\vec{V}$ predstavuje vektor pohľadu a v zátvorke sú udané konkrétne súradnice. Funkcia *vec3* slúži na vyjadrenie, že výsledok je 3D vektor. Ten sa skladá pre súradnice X a Y z podielu súradníc

pohľadu a následným násobením premennou *scale* (kapitola 2.3.3) a hodnotou 1.0 pre súradnicu Z. Výsledok je uložený do vektora $\overrightarrow{PV}$, ktorý reprezentuje parallax vektor.

Počet krokov trasovania

Vypočítaný $\overrightarrow{PV}$ vektor udáva smer budúceho trasovania, pre ktoré bude potrebné ešte udať počet krokov. Pri počte krokov vzniká ale problém. Ak sa udá príliš veľa krokov, celkový výpočet POM bude pomalý. Naopak, ak sa udá príliš malý počet, POM bude nepresný a výsledné zobrazenie bude nekorektné. Výpočet POM s malým počtom krokov demonštruje následný obrázok:



2.21: Demonštrácia chyby pri malom počte krokov [cs.utah.edu]

Ako je na ľavom obrázku vidno, výsledný prienik mal byť v bode *X*. Avšak kvôli malému počtu krokov a tým pádom veľkej vzdialenosti medzi krokmi sa pri trasovaní podarilo minúť výčnelok, na ktorom sa nachádza bod *X*. Následne bol vypočítaný prienik v blízkosti konečného bodu, čo nie je korektný prienik a vedie k nereálnemu zobrazeniu, ktoré je pozorovateľné na obrázku vpravo. Neexistuje predpis na ideálny počtet krokov. Záleží to od konkrétneho materiálu, vzdialenosti objektu od pozorovateľa, od premennej *scale* a od mnoho ďalších faktorov. Počet krokov teda musí užívateľ určiť sám.

Následne po určení počtu krokov, je možné pre isté prípady poupraviť tento počet. Pri hlbšom rozbore je možné dôjsť k myšlienke, že čím je uhol pohľadu k povrchu menší, tým treba na výpočet viacej krokov, a naopak, čím sa pozorovateľ kolmejšie pozerá na povrch, tým na presnosť výpočtu stačí menší počet krokov. V konečnom dôsledku, užívateľ určí minimálny (m_{min}) a maximálny (m_{max}) počet krokov a výsledný počet m pre daný rasterizovaný bod sa dá dopočítať podľa vzorca:

$$m = m_{min} + \vec{N} \cdot \vec{V} * (m_{max} - m_{min})$$

Kde, $\vec{N}$ je normálový vektor interpolovaný z vertexov pre daný rasterizovaný bod a $\vec{V}$ je vektor smerujúci k pozorovateľovi.

Trasovanie parallax vektora

Keď už bol vypočítaný parallax vektor a je aj známy počet krokov, je ešte nutné vypočítať poslednú vec pred samotným hľadaním nových textúrovacích súradníc. Tým je vektor $\overrightarrow{SV}$ udávajúci práve jeden krok.

$$\overrightarrow{SV} = \frac{\overrightarrow{PV}}{m}$$

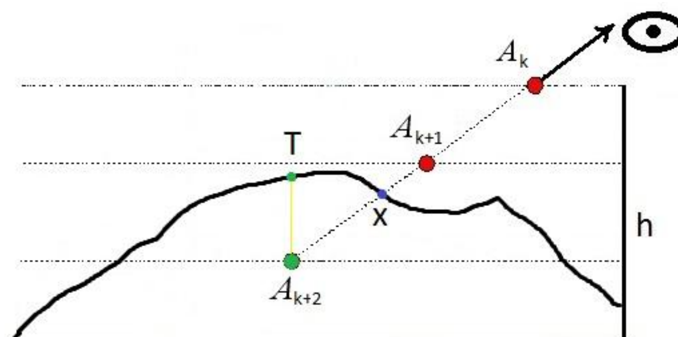
Nutnosťou je ešte definícia univerzálneho vzorca pre výpočet ľubovoľného trasovacieho bodu A_k ako na obrázku 2.20. Pozíciu bodu (X, Y, Z) A_k v kroku k je možné vypočítať podľa:

$$A_k = \overrightarrow{PV} - \overrightarrow{SV} * k$$

Samotné hľadanie prieniku prebieha tak, že postupne od kroku 0 pre všetky nastávajúce kroky sa vypočíta bod A_k, kde k je číslo aktuálneho kroku. Následne za pomoci rozdielu pôvodných textúrovacích súradníc (U a V) s jeho dvomi súradnicami (X a Y) sa prečíta z height mapy výška h v danom bode. Ak prečítaná výška h je menšia ako hodnota Z v bode A_k, bod sa nachádza mimo povrchu a pokračuje sa na výpočet ďalšieho kroku. Inak ak výška h je väčšia alebo rovná ako hodnota Z, tak sa bod A nachádza už v simulovanom povrchu a teda vyhľadávanie sa preruší a aktuálny bod sa prehlási za výsledný. Tento algoritmus sa nazýva lineárny a je postavený tak, že aspoň jeden kolízny bod sa vždy nájde a to minimálne posledný trasovací bod.

Presnejšie dopočítanie výsledného bodu

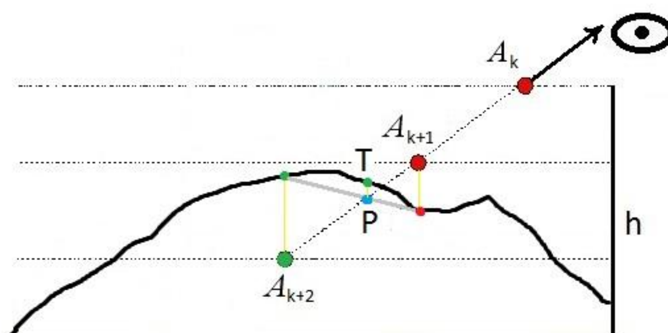
Pri hlbšej analýze sa dá zistiť, že ak sa pri hľadaní prieniku použije iba lineárne vyhľadávanie, výsledok môže byť stále pomerne nepresný. Tento jav je možné pozorovať na nasledujúcom obrázku:



2.22: Nepresnosť lineárneho vyhľadávania

Obrázok zobrazuje výsledok lineárneho vyhľadávania, a to zelený bod A_{k+2}. Avšak pravý prienik leží v bode *X*. Pri mnoho povrchoch a množstvu krokov je táto nepresnosť zanedbateľná, ale niekedy je značne nevyhovujúca.

Riešenia dopočítavania bodu bližšieho k *X* sú rôzne a aj v tomto sa práve líšia rozličné metódy uvedené v kapitole 2.4.1. Napríklad Steep parallax mapping tento problém ignoruje a za výsledný bod považuje A_{k+2} [McGuire 2005]. Real-time relief mapping na túto činnosť používa takzvané binárne vyhľadávanie [Policarpo 2005]. A nakoniec POM [Tatarchuk 2006] tento rozdiel dopočítava algoritmom, ktorý sa najlepšie vysvetlí za pomoci obrázka:

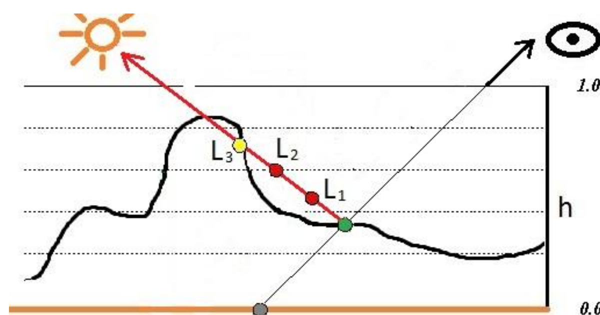


2.23: Dopočítanie presnejšieho výsledného bodu v POM

Po nájdení prvého bodu, ktorý sa už nachádza v simulovanom povrchu (na obrázku A_{k+2}), sa vypočíta prienik *P* medzi spojnicou dvoch posledných trasovacích bodov (na obrázku A_{k+2} a A_{k+1}) a spojnicou výšok v týchto dvoch bodoch. Tým sa dostane výsledný bod v metóde POM, ktorý je pomerne presný a samotný dopočet tohto prieniku nie je časovo náročný [Tatarchuk 2006].

2.4.4 Hard shadows a Soft shadows

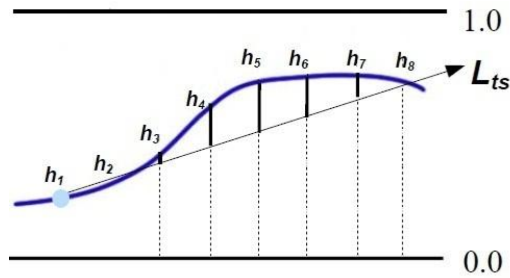
POM a podobné metódy poskytujú možnosť vykresľovania takzvaného self-shadowingu. Tento pojem znamená že objekty simulovaného povrchu medzi sebou môžu vrhať tiene. Prvou implementáciou boli takzvané Hard shadows [McGuire 2005], ktoré sa počítali taktiež trasovaním, ale v tomto prípade svetelného lúča. Idea bola jednoduchá, a to ak svetelný lúč smerom od novo vypočítaného bodu na povrchu (viď predchádzajúcu kapitolu) ku zdroju svetla niekde narazí, tak tento bod je v tieni. Tým pádom sa jeho výsledná farba stmaví. Túto myšlienku demonštruje nasledujúci obrázok:



2.24: Ilustrácia Hard shadows

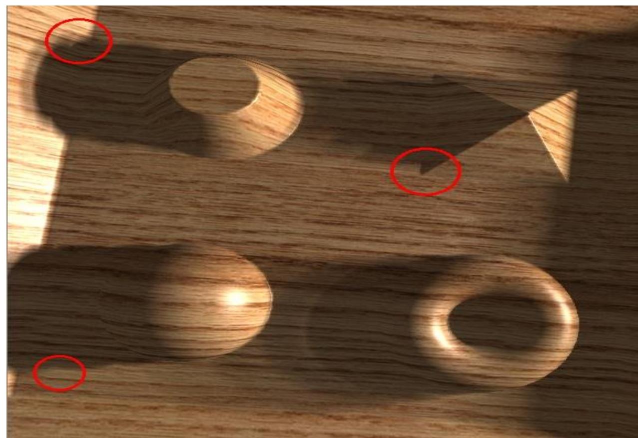
Takto definované tiene sa inak volajú aj „ostré“ tiene, lebo bod má len dve možnosti, a to že sa nachádza na svetle alebo v tieni. Nie je možné, aby bod ležal v polotieni a teda aby sa výsledný tieň postupne strácal tak ako je to v reálnom svete.

Avšak za pomoci modernej metódy Soft shadows [Tatarchuk 2006] sú takéto tiene možné. Táto metóda nehľadá, či existuje medzi svetlom a bodom na povrchu nejaká ľubovoľná kolízia, ale hľadá zo všetkých trasovacích bodov tie, ktoré sa nachádzajú v simulovanom povrchu (čiže kolidujú). Pre každý takýto bod sa následne vypočíta rozdiel medzi výškou z height mapy na týchto súradniciach (X a Y trasovacieho bodu) a výškou tohto bodu (súradnica Z). Z týchto rozdielov sa potom vyberie ten, ktorý je najväčší a podľa neho sa následne vypočíta veľkosť stmavnutia bodu na povrchu. Takto sa docieli reálny prechod tieňa medzi viac a menej zatieneným bodom na povrchu. Túto metódu ilustruje nastávajúci obrázok [Tatarchuk 2006]:



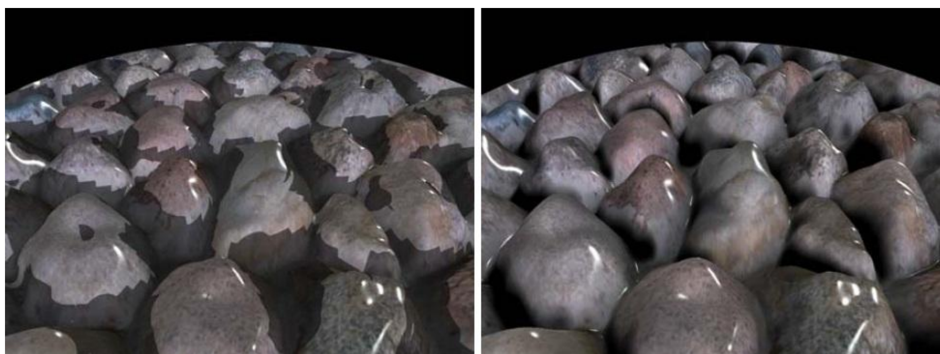
2.25: Princíp Soft shadows

Pri tomto príklade sa výsledný tieň vypočíta podľa výšky h_5. Avšak tak isto ako pri trasovaní pohľadu je nutné určiť správny počet krokov. Môže sa na to použiť vzorec ako pri výpočte množstva krokov trasovania pohľadu ale pri tieňovaní sú chyby, ktoré nastávajú menej zreteľné ako pri výpočte parallaxu. Preto sa počet krokov často udáva konštantným číslom. Ale nevypláca sa na druhú stranu podceniť tento faktor a určiť počet krokov na malé číslo. Vznikne totižto nepríjemný jav, ktorý demonštruje ďalší obrázok:



2.26: Chyba s malým počtom krokov v Soft shadows [cs.utah.edu]

Takto vykreslené tiene nepôsobia veľmi reálne i keď pre mnoho aplikácií by boli dostačujúce. Na záver si ukážeme prípad s extrémnou pozíciou svetla a porovnáme ako sa tieto dve technológie tieňovania s tým vysporiadajú. Zároveň obrázky demonštrujú možnosti Parallax occlusion mappingu pri zobrazovaní bežného typu povrchu. Obrázok je prevzatý z [Tatarchuk 2006] a vľavo je demonštrovaná metóda Hard Shadows a na pravo Soft shadows:



2.27: Porovnanie POM s Hard shadows a Soft shadows

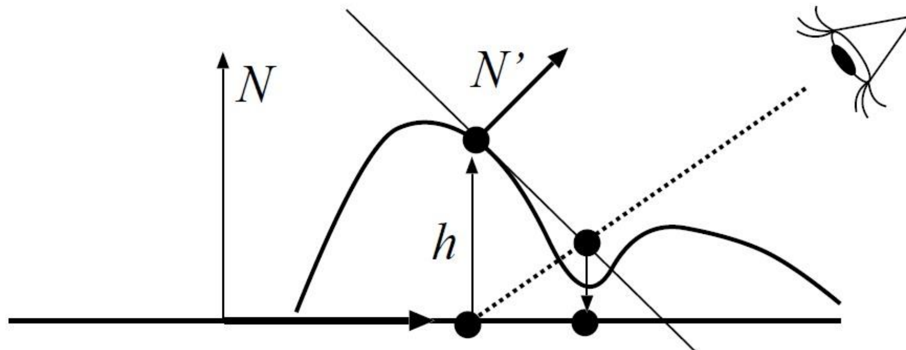
2.5 Iterative parallax mapping

2.5.1 História

Po úspechu Parallax occlusion mappingu a podobných technológií prichádza na scénu nová metóda. Táto poňala výpočet výsledných súradníc trochu inak ako to bolo pri predchádzajúcej technológii. Dá sa povedať, že sa ideami viac približuje obyčajnému Kanekovmu parallaxu [Kaneko 2001], než metódam trasovania pohľadu (kapitola 2.4). Metóda nedosahuje také presnosti ako POM [ShaderX3 2004] [Tatarchuk 2005] [Tatarchuk 2006], ale je od neho rýchlejšia a zároveň presnejšia ako obyčajný Kanekov parallax. Presne takáto optimalizácia chýbala medzi uvedenými technológiami, kde obyčajný parallax nestačil a trasovacie techniky bolo zbytočné použiť kvôli náročnosti výpočtov. Takýto kompromis dokázal spraviť v roku 2006 pán Mátyás Premecz [Premecz 2006] a túto metódu nazval Iterative parallax mapping with slope information.

2.5.2 Princíp metódy

Ako už bolo naznačené, táto metóda sa trochu vracia k princípom originálnej koncepcii Kaneka [Kaneko 2001] a teda neprebieha trasovanie vektora pohľadu. Metóda, ako doposiaľ jediná, kombinuje pre výpočet výsledných textúrovacích súradníc ako height mapu, tak aj informácie z normál mapy. Algoritmus je založený na istom počte iteračných krokov cyklu, pričom výpočet jedného kroku je znázornený na obrázku [Premecz 2006]:



2.28: Princíp Iterative parallax mappingu

Ako je na obrázku vidno, novo vypočítaný bod je blízko skutočného prieniku pohľadu so simulovaným povrchom. V porovnaní s obyčajným parallaxom je ďaleko presnejší výpočet docielený práve započítaním normálového vektoru v danom bode. Avšak sa nedá zaručiť, že bod vypočítaný takýmto algoritmom bude zároveň prvým kolíznym bodom medzi pozíciou pozorovateľa a bodom na povrchu polygónu [Premecz 2006]. Preto táto metóda je menej presná ako príbuzné metódy, ktoré trasujú vektor pohľadu. Čo sa týka náročnosti výpočtu iteratívneho parallax mappingu je táto metóda rýchlejšia ako technológie uvedené v predchádzajúcej kapitole. Pri každej iterácii sa musia vykonať dva náhľady do textúr, konkrétne do height mapy a na tej istej pozícii aj do normal mapy. Ale je vhodné opomenúť, že počet iterácií je radikálne menší, než počet krokov v trasovacích metódach. Pre ilustráciu v Iterative parallax mappingu sa počet iterácií obyčajne pohybuje okolo čísla 4, teda dokopy okolo 8 náhľadov do textúr. Naopak, pri trasovacích technikách sa v niektorých prípadoch množstvo krokov určí dokonca až na 100.

2.5.3 Výpočet textúrovacích súradníc

Ako už bolo vyššie uvedené, Iterative parallax mapping [Premecz 2006] používa na výpočet textúrovacích súradníc množstvo krokov s rovnakým algoritmom. Pred začatím prepočtu je za potreby tak ako v ostatných metódach simulujúcich nerovnosť povrchu previesť potrebné vektory z modelového priestoru do tečného priestoru za pomoci TBN matice (kapitola 2.2.4). Výpočet pre každý krok tejto metódy sa veľmi podobá originálnemu výpočtu v parallax mappingu od pána Kaneka [Kaneko 2001]. V jednotlivých krokoch je nutné vyjadriť výšku v danom bode a následne ju aplikovať pri posunu na iné textúrovacie súradnice. Zmenou je zakomponovanie normálového vektora do tohto prepočtu. Pri výpočte výšky z height mapy je vzorec pre obe metódy rovnaký ako v originálnej implementácii (kapitola 2.3.3):

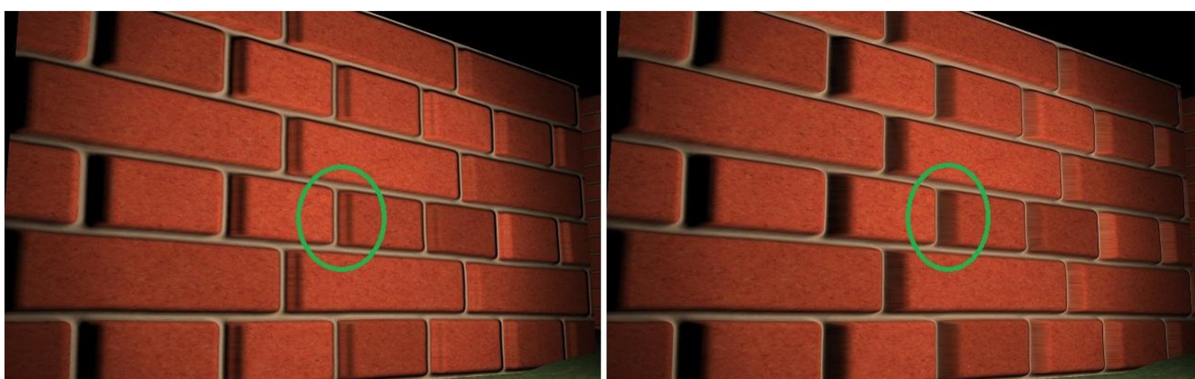
$$h' = h * scale + bias$$

Vo vzorci h značí načítanú hodnotu z height mapy a premenné *scale* a *bias* slúžia na dodatočné upravenie výšky podľa potreby. Aplikácia tejto vypočítanej výšky sa však voči pôvodnému vzorcu použitému v obyčajnom parallax mappingu trochu líši (vzorec je prevzatý a upravený z [Premecz 2006]) :

$$n'_{(u,v)} = V_{(x,y)} * h' * N_{(z)}$$

Pričom n' znamená novo vypočítané súradnice, $V_{(x,y)}$ je vektor pohľadu a $N_{(z)}$ je Z súradnica normálového vektoru v tom danom bode, ktorý bol prečítaný z normal mapy. Za povšimnutie stojí fakt, že algoritmus pracuje iba so Z súradnicou normálového vektoru. Tým pádom je podstatný len sklon k povrchu a nie aj smer - X a Y súradnice normály sú nepodstatné.

Na koniec kapitoly si ukážeme obrázok znázorňujúci grafické porovnanie medzi vykreslenými tehličkami z obrázka 2.9 za pomoci klasického parallax mappingu (na ľavo) a s aplikáciou Iterative parallax mapping (na pravo). Na ľavom obrázku je možné pozorovať defekt dopadu svetla, ktorý sa už na pravom obrázku nevyskytuje. A zároveň na pravom obrázku tehličky pôsobia viac vypuklejšie.



2.29: Porovnanie klasického parallax mappingu s iteračným parallax mappingom

3 Implementácia

Táto kapitola sa zaoberá implementačnou časťou zadania tejto práce. Avšak neobsahuje kompletné zdrojové kódy, ale sú tu popísane iba zaujímavejšie algoritmy a časti implementácie. Celé zdrojové kódy a súčasti implementácie je možné nájsť v prílohe.

Po teoretickej kapitole, kde sme si predstavili niektoré metódy na simuláciu nerovnosti povrchu, nastal čas si ukázať ich softvérové vyhotovenie (kapitola 3.1). Následne by som chcel predstaviť svoju vyvinutú metódu, ktorá sa snaží simulovať nerovnosť povrchu aj s istým napodobením siluet (kapitola 3.2). Tieto všetky implementované metódy sú použité a prezentované v graficko-hernej aplikácii, ktorá je súčasť tejto práce. Popisu implementácie demonštračnej aplikácie je venovaná kapitolu 3.3.

3.1 Implementácia metód

Metódy v tejto práci boli implementované na programovej úrovni grafických kariet. Novodobé Shadery, ktoré dávajú možnosť programátorovi meniť grafický cyklus v kartách, poskytujú ideálne prostriedky pre vytvorenie takýchto efektov. Programovanie shaderov prebiehalo za pomoci jazyka GLSL, ktorý je používaný v OpenGL. Na implementovanie metód boli využité konkrétne Vertex a Fragment (Pixel) shader. Prvý z nich sa používa pri spracovaní každého vertexu a umožňuje nad ním robiť radu operácií. Fragment shader je volaný pri každom rasterizovanom bode polygónu. Všetky metódy boli implementované za pomoci bodového svetla.

3.1.1 Phongovo osvetlenie

Vertex shader

Pri programovaní vertex shadera a Phongovho osvetlenia je okrem iných operácií nutné vyjadriť pozíciu vertexu, vektor pohľadu smerom k pozorovateľovi a zároveň vyjadriť normálu. To sa uskutoční pomocou kódu:

```
//výpočet pozície vertexu
  vVertex = vec3(gl_ModelViewMatrix * gl_Vertex);
//výpočet normalizovaného vektora pohľadu
  vViewDir = normalize(-vVertex);
//výpočet normály
  vNormal = normalize(gl_NormalMatrix * gl_Normal);
```

Fragment shader

Pri Fragment shaderu je potreba vykonať viacej operácií. Na začiatku je nutné vypočítať normalizovaný smer a vzdialenosť svetla od povrchu.

```
// smeru svetla id, vPosition - interpolovaná pozícia rasterizovaného bodu
  vec3 lightDir = gl_LightSource[id].position.xyz - vPosition;
//vypočítanie vzdialenosti svetla
  float distance = length(lightDir);
//následná normalizácia
  lightDir = normalize(lightDir);
```

Následne sa vypočíta uhol medzi normálou a vektorom ku zdroju svetla, aby bolo možné vypočítať diffuse zložku. Tak isto je potrebné vypočítať smer priameho odrazu svetla od povrchu, teda smer úplného lesku a aj jeho uhol s vektorom pohľadu. Tento výpočet je nutné spraviť pre správny určenie specular zložky.

```
//výpočet kosínusu uhla medzi normálov a svetlom. Výsledok bude prinajmenšom 0.
  float cosAngle = max(0.0, dot(vNormal, lightDir));
//premietnutie svetelného vektora cez normalový vektor pre vytvorenie reflexie
  vec3 reflectionDir = reflect(-lightDir, vNormal);
//následný výpočet kosínusu uhla medzi reflexiou a vektorom pohľadu
  float reflectionAngle = max(0.0, dot(vViewDir, reflectionDir));
```

Pri bodových svetlách nemožno zabudnúť na výpočet zoslabenia svetla závislého od vzdialenosti zdroja svetla, takzvané attenuation (viď kapitola 2.1.3).

```
//výpočet attenuation
  float att = 1.0 / (gl_LightSource[id].constantAttenuation +
       gl_LightSource[id].linearAttenuation * distance +
       gl_LightSource[id].quadraticAttenuation * distance * distance);
```

Posledným krokom sú výpočty všetkých potrebných zložiek pre Phongovo osvetlenie. Je nutné však zmieniť, že výpočet diffuse a specular zložky závisí od `cosAngle`, teda od uhla medzi svetlom a normálou. Ak tento uhol je väčší než kolmý, tieto dve zložky sa nevypočítajú a je vypočítaná iba ambient zložka.

```
//výpočet každej výslednej svetelnej zložky
  diffuse = att * cosAngle * gl_LightSource[id].diffuse *
        gl_FrontMaterial.diffuse;
  specular = att * gl_LightSource[id].specular * gl_FrontMaterial.specular *
        pow(reflectionAngle, gl_FrontMaterial.shininess);
  ambient = gl_LightSource[id].ambient * gl_FrontMaterial.ambient
```

3.1.2 Normal mapping

Vertex shader

Pri normal mappingu sa prvý krát budeme zaoberať takzvanou TBN maticou. Je ideálne ju vypočítať vo Vertex shadery a zároveň tu s ňou spraviť aj potrebné operácie, čo sú prevod niektorých vektorov do tečného priestoru za pomoci násobenia. Ako už bolo v teoretickej časti práce spomenuté, skladá sa z troch vektorov:

```
//výpočet premenných pre TBN maticu
  vec3 n = normalize(gl_NormalMatrix * gl_Normal);
  vec3 t = normalize(gl_NormalMatrix * tangentAttrib.xyz);
  vec3 b = cross(n, t);
```

V rovniciach nie je vysvetlený `tangentAttrib.` Táto premenná je atribútom pre vertex shader (premenná, ktorá sa posiela do vertex shaderu, ale je pre každý vertex zvlášť vyjadrená) a konkrétne

popisuje tangent hodnotu pre daný vrchol. Táto hodnota je vypočítana ešte v hlavnej aplikácii a teda pred poslaním do grafickej karty. Následne je možné vyjadriť TBN maticu:

```
//výpočet samotnej TBN matice
  vTBNMatrix = mat3(t.x, b.x, n.x,
                    t.y, b.y, n.y,
                    t.z, b.z, n.z);
```

V implementácii tejto práce je však TBN matica posielaná do fragment shaderu a vektory svetla sa počítajú až v ňom. Dôvodom je, že táto implementácia ráta s ôsmimi možnými svetlami, a teda je výpočtovo menej náročné interpolovať medzi vertexom a rasterizovaným bodom jednu 3x3 maticu ako 8 trojdimenzionálnych vektorov. Na druhú stranu je pravda, že potom pre každý rasterizovaný bod sa musí vykonať o 8 výpočtov viac, avšak testovanie ukázalo, že aplikácia bola rýchlejšia, keď bola interpolovaná TBN matica. Ale treba opomenúť, že pri jednom alebo dvoch svetlách by výpočet bol efektívnejší vo vertex shadery.

Fragment Shader

Do fragment shaderu sa pri týchto metódach (okrem Phongova osvetlenia) posiela z aplikácie normal mapa, za pomoci ktorej sa v tomto shaderi vyjadrí nová upravená normála pre povrch. Deje sa to tak, že shader prečíta z mapy na danom mieste (udané textúrovacími súradnicami) RGB hodnotu a z nej sa vypočíta normála:

```
//výpočet normály podľa normal mapy, gl_TexCoord[0].st – textúrovacie súradnice
  vec4 normalColor = texture2D(normalMap, gl_TexCoord[0].st);
  vec3 NormalDirT = normalize((normalColor.xyz * 2.0) - 1.0);
```

Ďalej je potrebné vyrátať pre každé jedno svetlo *i*, vektor v modelovom a následne aj v tečnom priestore. Na výpočet v tečnom priestore je použitá interpolovaná TBN matica z Vertex shaderu:

```
// výpočet svetelného vektora pre jedno svetlo vo svetovom a aj tečnom priestore
// kde vPosition je interpolovaná pozícia rasterizovaného bodu
  vec4 LightDirW = gl_LightSource[i].position.xyz - vPosition;
  vec4 LightDirT = vTBNMatrix * LightDirW;
```

Na takto vypočítané vektory sa aplikuje Phongovo osvetlenie a Normal mapping je hotový. Avšak môže dôjsť k istým nežiadúcim efektom, a to zobrazovanie nerovností povrchu aj na polygónoch, ktoré sú odklonené od svetla. To je spôsobené tým, že novo vypočítaný normálový vektor môže byť voči polygónu tak naklonený, že so svetlom ešte zviera uhol menší než 90°, pričom na samotný polygón by už svetlo nemalo vôbec dopadať. Takýto problém je v tejto práci riešený s využitím ďalšej pomocnej normály, ktorá je klasicky interpolovaná ako vo Phongovom osvetlení a pred výpočtom osvetlenia sa skontroluje uhol medzi svetlom a touto normálou. Ak je uhol väčší, vykoná sa iba ambientná zložka, inak prebehne celé Phongové osvetlenie.

```
//ak svetlo je odklonené od normály polygónu, vypočíta sa iba ambient zložka
  if (dot(normalize(LightDirW), normalize(vNormalDirW)) > 0.0)
    Phong(diffuse, specular, ambient);
  else
    ambient = gl_LightSource[i].ambient * gl_FrontMaterial.ambient;
```

3.1.3 Parallax mapping a Iterative Parallax mapping

Kvôli malým implementačným rozdielom sú Parallax mapping s offset limiting a Iterative parallax mapping popísané v jednej kapitole. Následne je treba povedať, že Vertex shader týchto dvoch metód je úplne totožný s Normal mappingovým Vertex shaderom, a teda nie je v tejto kapitole rozoberaný.

Fragment shader

Ako bolo v teoretickej časti uvedené, princípom je vypočítať nové textúrovacie súradnice. V implementačných ukážkach sú tieto súradnice nazvané premennou *newTexCoord*. Následne si ukážeme výpočet súradníc v Parallax mapping with offset limiting.

```
//normalizovanie pohľadu
  vec3 nView = normalize(vViewDirT);
//získanie výšky z height mapy
  float height = texture2D(heightMap, TexCoord).r;
//prepočet posunu súradníc (pomocou scale a bias)
  height = height * scale + bias;
  vec2 newTexCoord = TexCoord + (height * nView.xy);
```

V kóde sa nachádza premenná `vViewDirT`, ktorá reprezentuje vektor pohľadu v tečnom priestore a následne sa normalizuje. Premenná `TexCoord` reprezentujúca pôvodné textúrovacie súradnice vypočítané interpoláciou.

Veľmi podobný kód prináša Iterative parallax mapping. Ten je však obohatený naviac o jeden prístup do textúry a o cyklus, v ktorom prebiehajú iterácie.

```
//inicializácia premennych pred cyklom
  vec2 newTexCoord = TexCoord;
  vec3 nView = normalize(vViewDirT);
//cyklus pre výpočet posunu súradníc
  for (int t=0; t<pocetIter; t++)
  {
    //prístup aj do height mapy a aj do normal mapy
      height = texture2D(heightMap, newTexCoord).r;
      normalZ = texture2D(normalMap, newTexCoord).z;
    //výpočet posunu súradníc obohatený o Z súradnicu normály
      height = height * scale + bias;
      newTexCoord += height * nView.xy * normalZ;
  }
```

Premenná `pocetIter` znázorňuje množstvo iterácií pre výpočet efektu. Ako je vidno, zdrojové kódy sú veľmi podobné. Novo vyjadrené súradnice sa následne aplikujú na výpočet normal mappingu a Phongova osvetlenia. Vizuálny rozdiel je možné pozorovať na obrázku 2.29.

3.1.4 Parallax Occlusion Mapping so soft shadows

Vertex Shader

Podobne, ako v predchádzajúcej kapitole sa Vertex shader veľmi nezmení. V tomto prípade však pribudne jeden príkaz slúžiaci k vypočítaniu časti parallax vektora (kapitola 2.4.3), ktorý vychádza z vektora pohľadu a neskôr vo Fragment shadery sa bude trasovať.

```
//výpočet časti Parallax vektora
  vec2 vParVec = vViewDirT.xy / vViewDirT.z;
```

Ako je v teórii uvedené, tento takzvaný vektor by mal byť 3D a mal by mať Z súradnicu presne 1.0 . Vo Vertex shaderi sme vypočítali len X a Y súradnicu tohto vektora, navyše ešte neupravenú za pomoci premennej *scale*. Ďalšie operácie s týmto vektorom budú prebiehať už vo Fragment shaderi.

Fragment shader

Pred samotným počítaním zmeny textúrovacích súradníc je nutné definovať a inicializovať niektoré nové premenné.

```
//defaultné hodnoty nových textúrovacíc súradníc sú pôvodné hodnoty
  vec2 newTexCoord = TexCoord;
//Aktuálne načítanie výšky (v pôvodných súradniciach)
  float AktualnaVyska = texture2D(heightMap, TexCoord).r;
//defaultná hodnota 0.99 kvôli možným numerickým nepresnostiam
  float VyskovaHranica = 0.99;
//Premenná do ktorej sa neskôr budú ukladať medzikroky posunu súradníc
  vec2 UVAktual = newTexCoord;
//Premenná do ktorej sa neskôr bude ukladať predchádzajúca výška z height mapy
  float MinulaVyska = 0.0;
```

Je potrebné z týchto premenných niektoré objasniť. `AktualnaVyska` je premenná, do ktorej sa pri počítaní bude ukladať aktuálne prečítaná výška z height mapy. Ďalej `VyskovaHranica` je premenná obsahujúca v danom trasovanom kroku Z súradnicu pohľadu a následne `UVAktual` bude obsahovať X a Y súradnicu trasovaného pohľadu.

Hneď za tým prebehne výpočet počtu krokov trasovania a vyjadrenie všetkých hodnôt pre posun v jednom kroku.

```
//výpočet uhla medzi pohľadom a normálou
  float VdotN = dot(normalize(vViewDirW), normalize(vNormalDirW));
//na základe tohto uhla sa vypočíta počet krokov a z neho v zapätí výška kroku
  float pocet = mix(MaxSteps, MinSteps, VdotN);
  float VyskaKroku = 1.0 / pocet;
//výpočet posunu UV pre jeden krok
  vec2 ParVecScale = vParVec * ParScale;
  vec2 UVKrok = VyskaKroku * ParVecScale;
```

`MaxSteps` a `MinSteps` sú premenné, ktoré užívateľ zadá pre konkrétny povrch cez *uniform* (posielanie dát do shaderu) a zodpovedajú maximálnemu a minimálnemu počtu krokov pre trasovanie. Následne si ukážeme cyklus, ktorý uskutočňuje samotné trasovanie a hľadá prvý kolízny bod.

```
//cyklus trasovania pohľadu - ak nie je kolízia, počítaj ďalší krok
  while (AktualnaVyska < VyskovaHranica) {
    //uloženie si aktuálnej výšky do minulej
      MinulaVyska = AktualnaVyska;
    //výpočet textúrovacích súradníc a aktuálnej výšky ďalšieho korku
      UVAktual -= UVKrok;
      AktualnaVyska = texture2D(heightMap, UVAktual).r;
    //výpočet výškovej hranice pre ďalší krok
      VyskovaHranica -= VyskaKroku;
}
```

Za povšimnutie stojí, že nie je implementovaný parallax vektor ako 3D a následné ďalšie vektory na prácu s ním tiež nie sú 3D ako tomu bolo v kapitole 2.4.3. Celý tento proces trasovania používa miesto 3D vektorov viaceré 2D vektory a klasické premenné. Je to z dôvodu väčšej prehľadnosti zdrojového kódu. Na koniec treba ešte takto vypočítaný kolízny bod doupravovať na presnejšie súradnice (kapitola 2.4.3) [Tatarchuk 2006].

```
//konečný dopočet presnejších súradníc
  float Delta1 = AktualnaVyska - VyskovaHranica;
  float Delta2 = VyskovaHranica + VyskaKroku - MinulaVyska;
  float Posun = VyskovaHranica * Delta2 + (VyskovaHranica + VyskaKroku) * Delta1;
  Posun /= (Delta2 + Delta1);
  newTexCoord -= ParVecScale * (1.0 - Posun );
```

Presné textúrovacie súradnice budú použité ďalej vo výpočtoch normal mappingu a Phongova osvetlenia. Po týchto operáciách na úplný záver sa musí vypočítať ešte Soft shadows, čo je pre každé svetlo zvlášť proces. Ako prvý krok výpočtu Soft shadows je nutné upraviť si vektor svetla a vypočítať potrebné výškové premenné.

```
//úprava svetelného vektora a výpočet jedného kroku
  vec2 LightDir = (vLightDirT.xy / vLightDirT.z) * ShadowScale;
  vec2 LightVecStep = LightDir / float(ShadowSteps);
//inicializácia výšky na 1.0, ktorá postupným trasovaním bude klesať
  float AktualVyska = 1.0;
//výpočet výškového kroku (nie od 0, ale od výšky v bode na povrchu)
  float InitVyska = texture2D(heightMap, TextCoord).r;
  float VyskaKrok = (1.0 - InitVyska) / float(ShadowSteps);
```

Premenné `ShadowScale` a `ShadowSteps` sú opäť užívateľsky nastaviteľné cez *uniform*. `ShadowScale` slúži na nastavenie veľkosti tieňa a následne `ShadowSteps` značí počet krokov pre výpočet tieňa. Ďalej premenná `vLightDirT` je svetelný vektor v tečnom priestore. `VyskaKrok` vyjadruje výškový krok medzi výškou 1.0 a výškou bodu na povrchu. Na koniec ostáva vykonať samotné trasovanie svetla.

```
//inicializácia maximálneho rozdielu výšok
  maxH = 0.0;
//cyklus pre trasovanie svetla
  for (int t=0; t<ShadowSteps; t++){
    //výpočet nového kroku
      AktualVyska -= VyskaKrok;
      LightDir -= LightVecStep;
      H = texture2D(heightMap, inTextCoord + LightDir).r;
    //Zapíše doposiaľ najväčší kladný rozdiel výšok
      maxH = max(H - AktualVyska, maxH);
  }
  //výpočet výsledného koeficientu
  TienK = 1.0 - maxH;
```

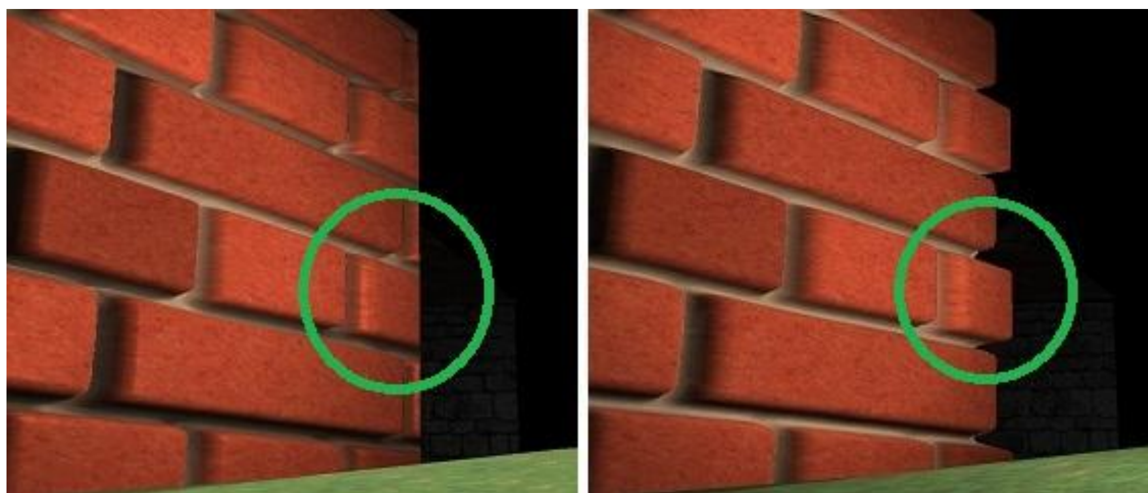
Premenná `maxH` je inicializovaná na 0.0 a tým sa zaručí, že `maxH` nikdy nebude menší než 0. Postupne sa počíta najväčší rozdiel výšok a ten na konci udáva aká je najväčšia prekážka medzi zobrazovaným bodom a svetlom. Na základe tohto výškového maxima sa potom určí koeficient (*TienK*), ktorým sa výsledná zobrazovaná farba (po všetkých operáciách, vrátane normal mappingu a Phongova osvetlenia) vynásobí a tým je aplikovaný tieň.

3.2 Vlastná metóda siluet

Táto metóda vznikla len kvôli môjmu veľkému záujmu o tento typ efektu a kvôli tomu, že oficiálne zdroje k obdobným metódam som našiel až krátky čas pred odovzdaním tejto práce a zároveň až po tom, čo táto metóda bola už implementovaná. Dodatočne nájdený oficiálny zdroj k metóde riešiacej podobný problém, avšak iným spôsobom, je [Oliveira 2005].

3.2.1 Účel metódy

Doposiaľ uvedené parallax metódy majú skvelé výsledky, ale neriešia jeden optický jav, ktorý po prepočte nastáva. Stred polygónu a jeho okolie vypadajú reálne a metódy docielia na týchto miestach dojem vypuklosti. Avšak, keď sa pozorovateľ pozrel na okraj polygónu, je pochopiteľne rovný a efekt vypuklosti sa stráca (obrázok 3.1 na ľavo). Ak užívateľovi tento jav prekáža, obvykle mu nezostáva nič iné, len prácne vymodelovať každú tehličku zvlášť, čím sa stráca zmysel použitia metód simulujúcich nerovnosť povrchu a pravdepodobne to vedie aj k redukcii výkonu aplikácie. Ale je tu ešte aj iná možnosť. Takýto jav sa snažia odstrániť metódy s výpočtom siluet [Oliveira 2005]. Nasledujúci obrázok demonštruje tento problém a vzápätí je možné pozorovať aj riešenie, ktoré siluety ponúkajú.



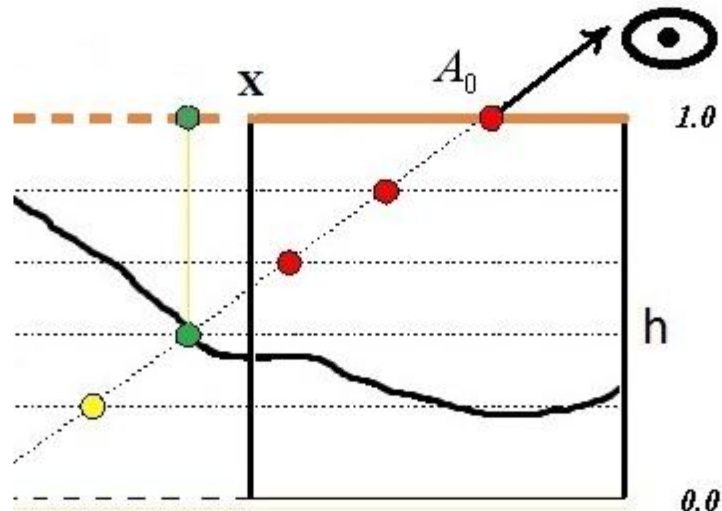
3.1: Porovnanie Parallax occlusion mapping bez siluet (na ľavo) a so siluetami (na pravo)

Povrch na okrajoch steny pôsobí na pravom obrázku ďaleko realistickejšie ako rovný okraj na ľavom obrázku. Základná idea metódy sa voči ostatným metódam vôbec nemení, a to že geometria telesa nebola modifikovaná. Na oboch obrázkoch je vidno iba jeden polygón s použitím patričnej metódy simulácie nerovnosti povrchu. Z toho vyplýva, že základný princíp aplikovania siluet je niektoré body na povrchu polygónu spraviť neviditeľné, aby objekty pôsobili vypuklo aj na okrajoch.

3.2.2 Návrh metódy

V tejto podkapitole je opísaný spôsob ako vypočítať, ktoré body na povrchu polygónu majú zmiznúť a vzápätí, ktoré sa majú zobraziť. Myšlienkový postup uvediem na príklade. Predstavte si ako sa zobrazuje okraj tehlovej steny v reálnom svete. Je možne si povšimnúť, že priehľadné časti pri okrajoch steny by mali byť práve v oblasti medzi tehličkami, a teda tam kde je malta. Čiže, čím hlbšie je malta vtlačená do steny, tým je možné pozorovať medzi tehličkami na okraji väčšiu medzeru. Tým

pádom sa dá povedať hypotéza, že body, ktoré majú menšiu výšku v height mape, budú skôr prehliadnuté ako vyššie položené body. Totižto, s nižšími výškami nedôjde ku kolízií počas trasovania vektora pohľadu a pri okrajoch polygónu teda pohľad môže smerovať až mimo polygón. V takýchto prípadoch je nutné umožniť priehľadnosť polygónu v danom bode. Myšlienku znázorňuje nasledujúci obrázok.



3.2: Ilustrácia priehľadnosti polygónu v danom bode

Na tomto obrázku je polygón znázornený oranžovou plnou čiarou. Prerušovaná oranžová čiara ilustruje, kde by sa polygón nachádzal, kebyže je väčší. Bod A_0 je práve rasterizovaný bod na povrchu polygónu. Vertikálna úsečka pod písmenom *X* predstavuje koniec polygónu. Ako je vidno, pohľadu nič nestojí v ceste, a teda by sa v bode A_0 zobrazil bod ležiaci už mimo polygónu. Na základe obrázku je možné povedať princíp mojej metódy, a to taký, že ak novo vypočítané textúrovacie súradnice sa nachádzajú za definovanou hranicou (na obrázku je to hranica polygónu), tak pre daný rasterizovaný bod na povrchu polygónu sa nevypočíta žiadna farba, a teda sa stane neviditeľným.

Definované hranice textúrovacích súradníc nie vždy musia byť práve na okraji každého polygónu. Príkladom je stena, ktorá je rovná ale skladá sa z viacerých polygónov. V takomto prípade by bol okraj aspoň jedného polygónu niekde v stene, čo spôsobí vymazanie bodov v tomto mieste a nepríjemne deravú stenu. Preto nie je ideálne rátať hranice na základe hrán polygónu, ale je lepšie určiť hranice ručne pre celú stenu. Čiže, pre model steny sa určia hranice maximálnou a minimálnou hodnotou UV súradníc, ktoré obsahujú krajné vertexy. Takto definované hranice sa dajú uložiť do jedného 4D vektoru (U_{min} , U_{max} , V_{min} , V_{max}).

Implementácia tejto metódy nie je zložitá a je to iba doplnok ku akejkoľvek inej Parallax metóde. To znamená, že Vertex a aj Fragment shader nie sú veľmi zmenené, avšak vo Fragment Shadery sa jedna zmena udeje. Doňho treba poslať definované hranice (napríklad cez *uniform*) a treba implementovať jednu podmienku, ktorá po vypočítaní nových textúrovacích súradníc zistí, či sa nenachádzajú za definovanými hranicami. Ak sa nachádzajú, tak sa Fragment shader ukončí bez zápisu výslednej farby (príkaz *discard*). Takto je docielená priehľadnosť bodu na polygóne.

Táto metóda je veľmi jednoduchá, ale na obyčajné steny dosahuje kvalitné výsledky. Bohužiaľ pri aplikácii na geometricky nerovný a zložitejší model, sa nedajú hranice tak presne určiť ako pri obyčajnej stene a metóda spôsobuje značné deformácie.

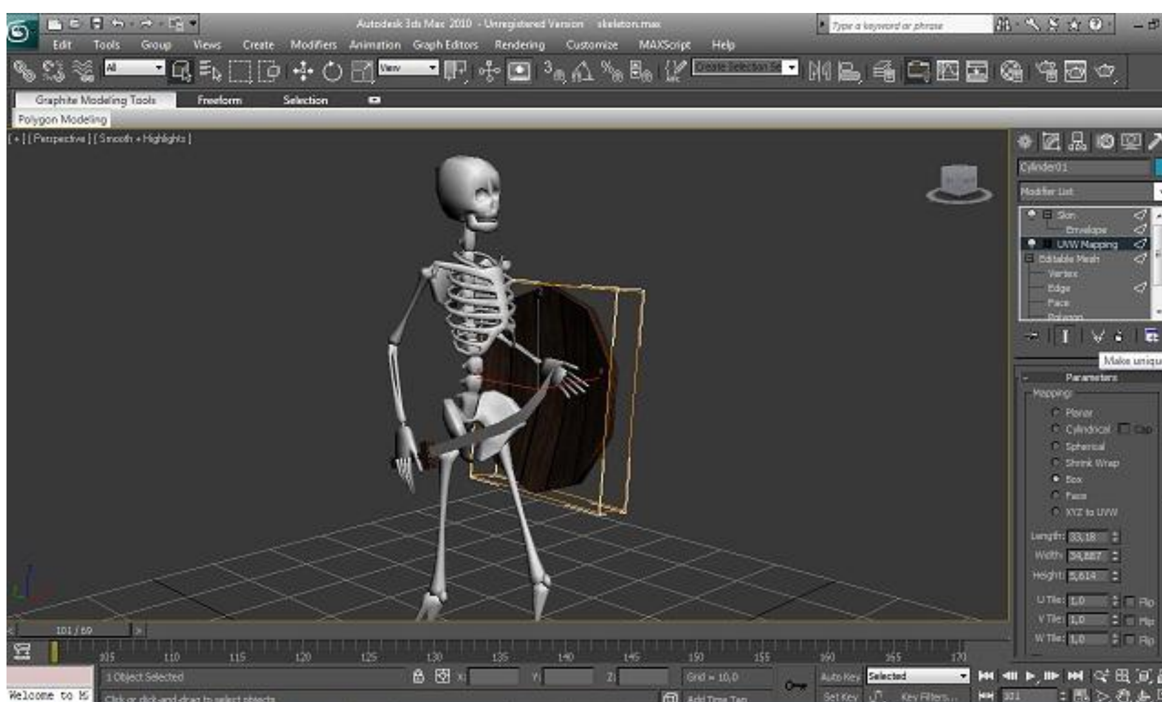
3.3 Graficko-herná aplikácia

Ako už bolo naznačené, táto kapitola sa zaoberá návrhom a implementáciou graficko-hernej aplikácie, v ktorej sú uvedené príklady všetkých metód z kapitoly 3.1. Na vytvorenie aplikácie bol použitý jazyk C++ a otvorená knižnica Delta3D, ktorá sa skladá z rozličných systémov. Napríklad pre grafiku používa knižnicu OSG implementovanú s OpenGL. Ďalej na animácie je integrovaná knižnica Cal3D a na fyziku je prednastavený systém ODE, ktorý sa dá však zameniť. Samotná knižnica Delta3D má zopár užitočných nástrojov pre uľahčenie práce programátorom hier. Z nich sa pri vývoji použili napríklad nástroje ako je STAGE – level editor, Object Viewer slúžiaci na prezeranie objektov a Animation Viewer, čo je šikovná utilita, keď si animátor chce pozrieť ako budú animácie v hre vyzerať. Niektoré použité modely a zvuky boli stiahnuté z [turbosquid.com], [fallingpixel.com], [freesound.org] a naopak niektoré boli ručne vyrobené.

Samotná hra je typu FPS (hráč sa pozerá pohľadom hrdinu) a je postavená do stredovekého obdobia, kde hráč ako hlavný hrdina so sekerou v ruke bojuje proti agresívnym kostlivcom. Boje sa odohrávajú vo vyľudnenej dedinke, ktorá bola jeho domovom pred obsadením kostlivcami. Hra začína na hrdinovej záhrade. Keďže je vyučený kamenár je tu možné si prehliadnuť jeho ponúkané materiály, ktoré aj zároveň predstavujú ukážky jednotlivých metód simulujúcich zakrivenie povrchu. Po celú dobu hráča sprevádza hororová hudba na privodenie správnej atmosféry.

3.3.1 Vytvorenie modelov a animácií

V mnohých grafických aplikáciách sa nachádzajú rôzne 3D modely. V niektorých sa vytvárajú prepracované a detailné modely a v iných stačia jednoduché objekty ako je kocka a guľa. Keď je nutné vytvoriť komplexnejší model ako štandardné primitíva, je veľmi prácne ho „písať“ v zdrojovom kóde po vertexoch a polygónoch. V takomto prípade je už potrebné použiť nejaký modelovací nástroj. V dnešnej dobe je mnoho editorov, v ktorých sa dá vytvoriť celá scéna aj s animáciami a aj so zvukmi. V tejto práci som na vytváranie modelov a animácií použil jeden z najznámejších nástrojov tohto typu, a to 3D Studio Max (konkrétne verziu 2010).



3.3: 3D Studio Max 2010

3D Studio Max je veľmi prepracovaný a pohodlný nástroj na vytváranie modelov a animácií. Bolo však nutné doňho pridať plugin OSG Exporter kvôli možnosti exportu modelov aj s materiálmi v špecifickom OSG formáte, ktorý Delta3D plne podporuje. Podobná situácia nastala aj pri animáciách a bol potrebný ďalší plugin, a to CAL3D exporter. Ten ponúka 4 typy exportovaných súborov, a to súbory zvlášť pre model, animácie, kostru a materiál. Zaujímavosťou je, že oba exportéri majú vlastný formát súboru pre uloženie modelu a materiálu. Statické objekty sa teda exportovali do OSG formátu, ktorý poskytuje podstatne väčšie možnosti pre popis modelu a naopak menšie možnosti pre animácie. Preto animované objekty boli exportované do modelového súboru CAL3D (so súborom OSG nevie pracovať). V hre boli použité iba takzvané kostné animácie, pre ktoré CAL3D ponúka široký sortiment funkcií.

OSG formát sám o sebe poskytuje možnosť zapísania shaderov do modelu, avšak exportér v 3D Studio Maxu túto funkciu nepodporuje. Preto importovanie metód pre simuláciu nerovností povrchu do modelu bolo vykonané až dodatočne. Tento proces prebiehal v jednoducho implementovanej aplikácii, ktorá po štarte načítala model, priložila doňho preddefinovaný shader a znova uložila do OSG súboru.

3.3.2 Implementácia hráča a kostlivca

Trieda TActPerson

Väčšina hier sa dnes nezaobíde bez dynamiky objektov a schopnosti simulovať na nich procesy z reálneho sveta. Preto som do projektu implementoval triedu *TActPerson*, ktorá poskytuje abstrakciu nad základnými procesmi a funkciami akéhokoľvek tvora. Samotný hráč aj kostlivec z tejto triedy dedia a využívajú jej funkcie. Trieda obsahuje implementáciu celého životného cyklu od vytvorenia objektu až po jeho zánik. Tento životný cyklus sa rozdeľuje na isté procesy. Napríklad proces vytvorenia, ktorý začína vyvolaním metódy *OnBornStart* u potomka. Následne sa v každý vykresľovaný snímok zavolá metóda *OnBornUpdate*, až kým potomok nepovie, že proces rodenia už skončil a vzápätí uňho trieda vyvolá metódu *OnBornEnd* a prepne sa do stavu žitia. Takýmto spôsobom sú robené aj ďalšie procesy ako sú napríklad smrť, útok, pohyb atď. Za pomoci takejto otcovskej triedy stačí len do potomka podopĺňať telá metód, ktoré špecifikujú ako má reagovať na dané udalosti. Tým sa značná časť logiky hry oddelí od konkrétnych postáv a tie sa týmto spôsobom elegantne špecifikujú do konkrétnej podoby akú majú reprezentovať.

Trieda TActPlayer

Táto trieda reprezentuje implementáciu hráča a ako už bolo zmienené, je potomkom triedy *TActPerson*. Text je zameraný hlavne na riešenie fyziky hráča. To je predovšetkým narážanie do stien a detekcia iných osôb v okolí.

Hráč z pohľadu prvej osoby sa voľne pohybuje po mape, pričom koliduje s ostatnými objektmi. Knižnica Delta3D obsahuje mnoho funkcií na uľahčenie takéhoto typu fyziky. Najbežnejšou metódou ako docieliť toto správanie je použiť takzvaný *CollisionMotionModel*. Je to trieda určená presne na pohyb kamery, ktorá je ovládateľná buď klávesmi WDAS alebo šípkami a rotácia prebieha pohybom myšky. I keď táto trieda má pár nedostatkov, bola použitá v projekte. Na to aby kamera kolidovala s objektmi, je nutné v nich nastaviť parametre kolízie. Najdôležitejšie nastavenie je určiť kolízny model. Ten je často jeden z preddefinovaných ako je napríklad guľa, kužeľ alebo kocka. Ak sa objekt nedá vložiť do takéhoto kolízneho modelu, alebo by spôsoboval veľmi nepresnú kolíziu, je možnosť určiť kolízny model vlastnou polygonálnou sieťou. Preddefinované kolízne modely však šetria výpočtový čas, lebo sú podstatne jednoduchšie ako polygonálna sieť a preto sa odporúčajú prednostne používať. Takto sa vytvorí základný pohyb hráča

po mape. Je nutné však zmieniť, že detekcia kolízie s kostlivcom, prebieha trochu inak. V zásade nie je potrebné počítať presnú kolíziu, ale stačí len zistiť jeho prítomnosť v okolí. To sa docieli jednoduchým výpočtom vzdialeností dvoch bodov, konkrétne pozície kamery a kostlivca. Teda miesto toho aby sa hľadala geometrická kolízia, stačí prejsť zoznam kostlivcov v scéne a zistiť vzdialenosť. Ak sa kamera nachádza príliš blízko kostlivca, odsunie sa vektorom smerujúcim od kostlivca na preddefinovanú minimálnu vzdialenosť. Podobným princípom sa dá vypočítať aj kolízia pri útoku hráča na kostlivca. Stačí skontrolovať vzdialenosť a uhol natočenia kamery smerom ku kostlivcovi a keď spĺňajú preddefinované hodnoty, je možné zahájiť útok. Pri útoku dochádza k spustení animácie zaťatia sekery a k prehratiu patričného zvuku.

Trieda TActKostlivec

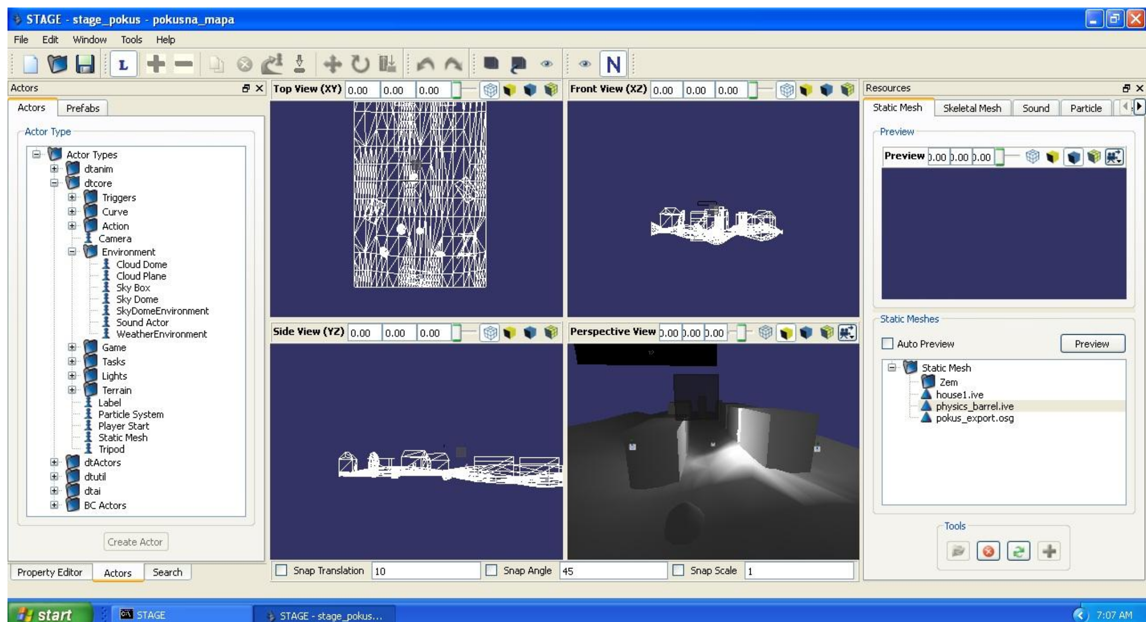
Kostlivec so všetkou logikou a fyzikou je implementovaný v triede *TActKostlivec*. Jeho účelom je zlikvidovať hráča. Pre pohyb kostlivca sa v tomto prípade nepoužíva žiaden *CollisionMotionModel*, ale posuny na nejakú pozíciu a prepočet kolízie so zemou sa počítajú ručne. Je to z dôvodu, že *CollisionMotionModel* je primárne určený pre kameru a zároveň vlastná implementácia prispieva k väčšej prispôsobivosti. Každý kostlivec má premennú, ktorá obsahuje pozíciu cieľa kam sa má pohybom dostať. Pri štarte pohybu sa zavolá metóda *OnMoveStart*, v ktorej je implementované pustenie animácie chôdze kostlivca. Pri pohybe sa v každom takte volá metóda *OnMoveUpdate*, kde sa aktualizuje a kontroluje pozícia. Ak je táto pozícia v tolerovanej blízkosti cieľa, vyvolá sa metóda *OnMoveEnd* kde sa vypne animácia a kostlivec prestane kráčať.

Kostlivec sa pohybuje len smerom k hráčovi. To, či vidí hráča, je implementované za pomoci detekcie kolízie medzi dvoma bodmi. Prvý bod sa nastaví na pozíciu kostlivca a druhý na pozíciu hráča a ak sa nenachádza medzi nimi žiadny statický objekt, kostlivec vidí hráča. V tomto prípade, si nastaví cieľ pohybu na pozíciu hráča a snaží sa k nemu dostať. Môže sa stať, že hráč sa schová za nejaký objekt a tým pádom dochádza k strate viditeľnosti. V takejto situácii, kostlivec dôjde na posledné miesto, kde bol hráč pozorovaný a zostáva tam kým hráča opäť neuvidí.

Ak však hráča dobehne a dostane sa k nemu na preddefinovanú vzdialenosť, vyvolá sa metóda *OnAttackStart*, pustí sa animácia a kostlivec začne útočiť. Samotný čin bodnutia (a následného ubratia života hráčovi) však počas celého procesu útočenia býva len jeden. Tento moment sa určí na základe ubehnutého času od spustenia animácie útoku a pre každú animáciu je určený zvlášť čas bodnutia, aby celý jav vypadal čo najreálnejšie. Pri samotnom bodnutí, sa u hráča zavolá funkcia *Hurt* s parametrom útoku, ktorá má na starosti ubratie života hráčovi podľa poslaného útoku a jeho aktuálnej obrany. Tento proces ublíženia je dedený od spoločného rodiča a teda aj pri útoku hráča je u kostlivca volaná funkcia *Hurt*. Ak nastane zánik kostlivca, pustí sa animácia pádu na zem a tým preňho životný cyklus končí.

3.3.3 Editor STAGE

Delta3D má prepracovanú myšlienku ako uľahčiť vývojárom tvorbu hier. Existuje tu trieda *GameManager*, ktorá slúži ako jadro pre akúkoľvek hru. Poskytuje možnosti pre spravovanie objektov v hre, pre zasielania správ medzi objektmi a uľahčuje prácu so vstupnými zariadeniami. *GameManager* obsahuje predovšetkým inštancie takzvaných aktorov, ktoré reprezentujú samotné objekty v scéne. V tomto prípade hráč, kostlivci a aj mapa majú spoločného predka, triedu *GameActor* a tým pádom sa stávajú aktormi pre *GameManager*. Doňho sa dajú pridávať takéto inštancie ručným zadávaním príkazov, alebo existuje pohodlnejšia cesta a to STAGE Editor.



3.4: STAGE Editor

Tento editor slúži na pospájanie všetkých aktorov a ostatných súčastí *GameManageru* do uceleného levelu. Tak isto je možné vytvoreným aktorom určovať vlastnosti ako sú napríklad pozícia, rotácia, fyzika a podobné. Pre import vlastných aktorov (pre túto hru sú to aktori hráča a kostlivca) STAGE ponúka jeden zaujímavý spôsob. Najprv sa napríklad v C++ naprogramuje celý aktor aj s chovaním, jeho konkrétnymi špecifikáciami, editovateľnými vlastnosťami atď. Následne sa z takto naprogramovaných aktorov vytvorí knižnica, ktorú je možné importovať do STAGEu. Týmto sa zaručí, že všetky dôležité algoritmy ostanú v binárnej forme a teda nie je nutné nič skriptovať, čo by spomalilo výkon aplikácie. Takto vytvorený level sa dá následne otvoriť vo vlastnej hre a *GameManager* a aktori sa budú chovať podľa preddefinovaného správania.

Delta3D je schopná moderná knižnica pre tvorbu simulácií a hier. Nachádza sa v nej zopár veľmi zaujímavých a originálnych myšlienok na riešenia niektorých bežných problémov. Knižnica je stále pod vývojom a býva pravidelne aktualizovaná. V prípade vytvorenia novej verzie niektorého zo subsystémov je táto verzia v blízkej dobe integrovaná aj do Delta3D knižnice.

4 Porovnanie metód

Táto kapitola ukáže rýchlostné porovnanie metód pre simuláciu zakrivenia povrchu. Vizuálne porovnanie prebiehalo postupne v celej práci počas rozoberania metód. Pri rýchlosti boli merané dva faktory. Prvým bola rýchlosť vykresľovania udaná v počte vykreslených obrázkov za sekundu (FPS) a druhým bol počet prístupov do textúry pre vykreslenie jedného pixelu. Názov druhého faktora naznačuje, že ide skorej o pamäťovú záležitosť, ale keďže prístup do textúry je drahá záležitosť na výpočtový čas, značne ovplyvňuje rýchlosť. Je nutné dodať, že pri rade metód je druhý faktor ovplyvnený vektorom pohľadu a definíciou konkrétneho materiálu. Preto v týchto prípadoch sú hodnoty udané v intervale od minima po maximum prístupov. Ďalej je dôležité uviesť parametre počítača, na ktorom prebehli testy. Z nich uvádzam len najvšeobecnejšie, a to:

Rozlíšenie obrazovky: 1920 x 1080 pixelov
CPU a RAM: Intel Core 2 Duo (3.0 GHz), 2048 MB
Grafická karta: NVIDIA Quadro FX 4600 (úplná grafická RAM: 1534 MB)

Následne je uvedená tabuľka porovnávania FPS pre všetky metódy s rôznym počtom svetiel (pre trasovacie metódy je v zátvorke uvedený počet krokov trasovania pohľadu a počet krokov trasovania tieňa):

Názov metódy a jej nastavenia	FPS (1 svetlo)	FPS (2 svetlá)	FPS (8 svetiel)
Phongovo osvetlenie	404	231	64
Normal mapping	287	167	47
Parallax mapping with offset limiting	263	159	46
Iterative parallax mapping – 2 iterácie	241	153	46
Iterative parallax mapping – 5 iterácií	200	139	45
POM (20, 5)	134	93	31
POM (20, 10)	122	82	26
POM (20, 20)	45	27	7
POM (50, 5)	101	76	29
POM (50, 10)	94	69	25
POM (50, 20)	39	25	7
POM (100, 5)	72	59	27
POM (100, 10)	68	55	23
POM (100, 20)	31	21	6
POM so siluetami (20, 5)	137	95	33
POM so siluetami (20, 10)	125	84	27
POM so siluetami (20, 20)	46	28	7
POM so siluetami (50, 5)	105	79	30
POM so siluetami (50, 10)	97	71	26
POM so siluetami (50, 20)	41	26	7
POM so siluetami (100, 5)	75	63	28
POM so siluetami (100, 10)	70	57	24
POM so siluetami (100, 20)	32	22	7

Tabuľka 1: Porovnanie FPS jednotlivých metód

Ako je z tabuľky vidno, najlepšie výsledky dosahuje Phongovo osvetlenie. Potom je možné pozorovať radikálny prepad FPS pri zmene z 10 na 20 tieňových krokov, konkrétne pri metódach POM a POM so siluetami. Ďalšou zaujímavosťou je, že pri 8 svetlách je prakticky zanedbateľný rozdiel medzi rýchlosťou Normal mappingu a zložitejšieho Iterative Parallax mappingu. Následne je uvedená tabuľka pre počet prístupov do textúry pri rovnakých podmienkach ako v Tabuľke 1:

Názov metódy a jej nastavenia	**Počet prístupov (1 svetlo)**	**Počet prístupov (2 svetlá)**	**Počet prístupov (8 svetiel)**
Phongovo osvetlenie	1	1	1
Normal mapping	2	2	2
Parallax mapping with offset limiting	3	3	3
Iterative parallax mapping – 2 iterácie	6	6	6
Iterative parallax mapping – 5 iterácií	12	12	12
POM (20, 5)	9 - 29	15 - 35	51 - 71
POM (20, 10)	14 - 34	25 - 45	91 - 111
POM (20, 20)	24 - 44	45 - 65	171 - 191
POM (50, 5)	9 - 59	15 - 65	51 - 101
POM (50, 10)	14 - 64	25 - 75	91 - 141
POM (50, 20)	24 - 74	45 - 95	171 - 221
POM (100, 5)	9 - 109	15 - 115	51 - 151
POM (100, 10)	14 - 114	25 - 125	91 - 191
POM (100, 20)	24 - 124	45 - 145	171 - 271
POM so siluetami (20, 5)	2 - 29	2 - 35	2 - 71
POM so siluetami (20, 10)	2 - 34	2 - 45	2 - 111
POM so siluetami (20, 20)	2 - 44	2 - 65	2 - 191
POM so siluetami (50, 5)	2 - 59	2 - 65	2 - 101
POM so siluetami (50, 10)	2 - 64	2 - 75	2 - 141
POM so siluetami (50, 20)	2 - 74	2 - 95	2 - 221
POM so siluetami (100, 5)	2 - 109	2 - 115	2 - 151
POM so siluetami (100, 10)	2 - 114	2 - 125	2 - 191
POM so siluetami (100, 20)	2 - 124	2 - 145	2 - 271

Tabuľka 2: Porovnanie počtu prístupov do textúry jednotlivých metód

Ako je z tabuľky vidno, POM so siluetami v istom prípade už po dvoch prístupoch do textúry môže poznať, že je za hranicami, a teda ukončiť výpočet.

Každá z metód má svoje zápory a klady. Niektoré sú rýchlejšie, ale zato nie sú moc presné. Iné sú pomalšie, ale dokážu simulovať prakticky akýkoľvek povrch. Preto neexistuje jedna univerzálne najlepšia metóda, ktorá by riešila každý problém a voľba metódy záleží na mnoho faktoroch ako sú napríklad materiál povrchu, simulovaná výška, ako ďaleko sa bude objekt nachádzať alebo či je detail dôležitý.

5 Záver

Zadaním tejto práce bolo predovšetkým si naštudovať algoritmy parallax mappingu a príbuzných technológií. V práci boli podrobne popísané viaceré metódy simulujúce nerovný povrch polygónu. Od základného Phongova osvetlenia, ktoré bolo nutné vysvetliť pre pochopenie ostatných metód, sa prešlo cez množstvo technológií až k najnovšej, a to iterative parallax mapping. Metódy pre simuláciu nerovností povrchu majú výborné výsledky a určite je v nich aj potenciál do budúcnosti.

Následne bol navrhnutý herný projekt pre ilustráciu týchto metód. Projekt bol vyvinutý za pomoci knižnice Delta3D a obsahoval okrem implementovaných metód aj charakterové animácie a fyziku hráča i fyziku protivníka. Delta3D poskytla výborné zázemie pre vývoj projektu. Je to moderná knižnica s množstvom zaujímavých koncepcií.

Dodatočne ako rozšírenie bola vyvinutá jednoduchá metóda, ktorá sa snažila parallax efektu dodať dojem siluet. Metóda je len základná a v budúcnosti by sa dala značne rozšíriť predovšetkým už o existujúce riešenie od [Oliveira 2005].

Pokračovať v projektu sa dá viacerými smermi. Je možné vyvinúť iné technológie pre svetelné efekty alebo pokračovať v skúmaní ďalších metód pre simuláciu nerovností povrchu. Tak isto na hernom projekte je možné implementovať väčšie množstvo levelov, viac nepriateľov alebo prepracovanejšiu umelú inteligenciu. Zapracovaním týchto vecí by bolo možné v budúcnosti vyvinúť plnohodnotnú hru.

Literatúra

BLINN, J. F. 1977: *Models of light reflection for computer synthesized pictures*. SIGGRAPH Comput. Graph. 11, 2 (Aug, 1977). DOI=http://doi.acm.org/10.1145/965141.563893

BLINN, J. F. 1978: Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (1978), pp. 286–292.

COOK, R. L. 1984: *Shade trees*. In *SIGGRAPH '84*: Proceedings of the 11th annual conference on Computer graphics and interactive techniques (New York, NY, USA,), ACM Press

GATH, J. 2006: *Derivation of the Tangent Space Matrix*. Blacksmith Studios, www.blacksmith-studios.dk/projects/downloads/tangent_matrix_derivation.php.

GOURAUD, H. 1971: *Continuous shading of curved surfaces*. IEEE Transactions on Computers , C-20(6):623–629, 1971.

KANEKO, T. a kol. 2001: Detailed shape representation with parallax mapping. In Proceedings of *ICAT 2001*.

MCGUIRE, M., MCGUIRE, M. 2005: Steep parallax mapping. In *I3D 2005 Poster* (2005). http://www.cs.brown.edu/research/graphics/games/SteepParallax/index.htm.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000: *Relief texture mapping*. In Siggraph 2000, Computer Graphics Proceedings, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., pp. 359–368.

OLIVEIRA, M. M, POLICARPO, F. 2005: *An Efficient Representation for Surface Details*. UFRGS Technical Report RP-351.

PHONG, B. T. 1973: *Illumination for Computer-Generated Images*. Ph.D. Thesis. UMI Order Number: AAI7402100., The University of Utah.

POLICARPO, F., OLIVEIRA, M. M., COMBA, J. 2005: *Real-Time Relief Mapping on Arbitrary Polygonal Surfaces*. In ACM SIGGRAPH Symposium on Interactive 3D Graphics Proceedings, ACM Press

PREMECZ, M. 2006: *Iterative Parallax Mapping with Slope Information.* Budapest University of Technology

STEHLÍK L. 2007: *Zobrazování povrchových detailů pomocí mapování textur.* Master Thesis. Charles University in Prague

TATARCHUK, N. 2005: *Practical dynamic parallax occlusion mapping*. In *ACM SIGGRAPH 2005 Sketches* (SIGGRAPH '05), Juan Buhler (Ed.). ACM, New York, NY, USA, , Article 106 . DOI=10.1145/1187112.1187240

TATARCHUK, N. 2006: *Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering*. In *ACM SIGGRAPH 2006 Courses* (SIGGRAPH '06). ACM, New York, NY, USA, 81-112. DOI=10.1145/1185657.1185830

WELSH, T. 2004: *Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces*. Tech. rep., Infiscape Corporation,.

WOLFGANG, E. 2004. *Shaderx3: Advanced Rendering with DirectX and OpenGL (Shaderx Series)*. Charles River Media, Inc., Rockland, MA, USA., pp. 135–154.

Zoznam príloh

Príloha 1. DVD obsahujúce zdrojové súbory projektu, spustiteľnú verziu hernej aplikácie, pdf s touto správou, zdrojový text správy vo formáte docx so všetkými obrázkami a pdf s plagátom.

Príloha 2. Plagát k projektu