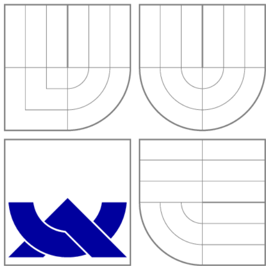
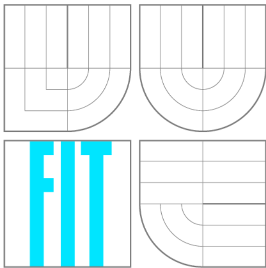


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GRAFICKÉ DEMO S VYUŽITÍM PROCEDURÁLNÍHO TEXTUROVÁNÍ

GRAPHICS DEMO EMPLOYING PROCEDURAL TEXTURES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — DUŠAN DREVICKÝ
AUTHOR

VEDOUCÍ PRÁCE — Ing. LUKÁŠ POLOK
SUPERVISOR

BRNO 2016

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Drevický Dušan**

Obor: Informační technologie

Téma: **Grafické demo s využitím procedurálního texturování**

Graphics Demo Employing Procedural Textures

Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se s fenoménem demoscény, zejména pak s disciplínou grafické intro s omezenou velikostí.
2. Prostudujte knihovnu OpenGL 4.0 a její rozšíření.
3. Nastudujte a popište techniky, vhodné pro generování animovaných procedurálních textur v 64kB demu.
4. Implementujte program, demonstrující použití zvolené metody.
5. Zhodnoťte dosažené výsledky a navrhněte možnosti dalšího vývoje.
6. Vytvořte video s prezentací projektu.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

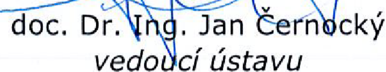
Vedoucí: **Polok Lukáš, Ing.**, UPGM FIT VUT

Datum zadání: 1. listopadu 2015

Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.



doc. Dr. Ing. Jan Černocký

vedoucí ústavu

Abstrakt

Práca sa zaoberá návrhom a implementáciou grafického dema veľkostne obmedzeného na 64 kB. Pre vytvorenie štruktúry a vzhľadu realistického prírodného terénu je využité procedurálne generovanie. Terén je definovaný pomocou výškovej mapy, ktorá vychádza zo šumových funkcií popísaných v práci. Zobrazovanie je implementované pomocou metódy ray marching a prebieha v reálnom čase. Všetky v práci popísané časti implementácie sú vytvorené výlučne vo fragment shaderi grafickej karty.

Abstract

This thesis deals with the design and implementation of a size-limited 64 kB graphics demo. Procedural generation is used for the definition of structure and appearance of realistic natural terrain. The terrain is described by a heightmap based on noise functions described in this work. Rendering utilizes the ray marching method and is done in real time. All parts of the implementation described in this work were created exclusively within the fragment shader of the GPU.

Kľúčové slová

grafické demo, veľkostne obmedzený spustiteľný súbor, zobrazovanie v reálnom čase, procedurálne textúry, procedurálne generovanie terénu, výšková mapa, šumové funkcie, ray marching, OpenGL, GLSL, fragment shader

Keywords

graphics demo, size-limited executable, real-time rendering, procedural textures, procedural terrain generation, heightmap, noise functions, ray marching, OpenGL, GLSL, fragment shader

Citácia

DREVICKÝ, Dušan. *Grafické demo s využitím procedurálního texturování.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Polok Lukáš.

Grafické demo s využitím procedurálního texturování

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostane pod vedením pána Ing. Lukáša Poloka. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

. . . .
Dušan Drevický
18. mája 2016

Poďakovanie

Rád by som sa poďakoval vedúcemu mojej bakalárskej práce pánovi Ing. Lukášovi Polokovi za jeho rady, pripomienky a ochotu pri konzultáciách.

© Dušan Drevický, 2016.
Táto práca vznikla ako školské dielo na FIT VUT v Brně. Práca je chránená autorským zákonom a jej využitie bez poskytnutia oprávnenia autorom je nezákonné, s výnimkou zákonne definovaných prípadov.

Obsah

1 Úvod 3

2 Grafické demo a demoscéna 5
2.1 Grafické demo . . . 5
2.2 Grafické intro s obmedzenou veľkosťou . . . 5

3 Procedurálne generovanie textúr 7
3.1 Perlinov šum . . . 7
3.2 Fractional Brownian Motion . . . 9
3.3 Kreslenie pomocou matematických predpisov . . . 9
3.4 Kombinácia funkcie a šumu . . . 11
3.5 Zobrazenie textúry na modeli v 3D scéne . . . 11
3.6 Fraktály . . . 12

4 Procedurálne generovanie modelov 15
4.1 Signed distance functions . . . 15

5 Zobrazovacie metódy 18
5.1 Ray tracing . . . 18
5.2 Ray marching . . . 19

6 Návrh riešenia 20
6.1 Štruktúra aplikácie . . . 20
6.2 Návrh shaderov . . . 20

7 Implementácia 22
7.1 Algoritmus zobrazovania . . . 22
7.2 Osvetľovací model . . . 28
7.3 Popis procedurálneho terénu . . . 30

8 Zhodnotenie rýchlosti zobrazovania 35

9 Záver 37

Literatúra 39

Prílohy 41
Zoznam príloh . . . 42

A Ukážka výsledných scén **43**

B Použitie aplikácie **45**

C Obsah CD **46**

Kapitola 1

Úvod

Pri tvorbe virtuálnych svetov pre počítačové hry, filmy alebo iné účely sa často kladie veľký dôraz na to, aby boli použité modely a textúry realistické a aby čo najviac pripomínali imitovaný svet. Tento cieľ je možné dosiahnuť viacerými spôsobmi. Model alebo textúra môže byť ručne vytvorená umelcom spolu s požadovanými vlastnosťami. Ak by sa však požiadavky zmenili, môže byť ďalšia úprava výsledku zložitá. Tento postup je navyše časovo náročný a teda drahý. V prípade, že má aplikácia za cieľ simulovať obrovský svet, ktorý obsahuje desiatky tisíc objektov, nie je fyzicky možné, aby bol každý z nich manuálne vytvorený.

Vhodným riešením je využitie procedurálne generovaného obsahu. Aj keď sa mnohé javy reálneho sveta vyznačujú na prvý pohľad vysokou mierou zložitosti, ukázalo sa, že v niektorých prípadoch je túto zložitosť možné veľmi dobre popísať pomocou matematiky a procedúr, ktoré určujú ako daný fenomén simulovať. Matematickým podkladom procedúr, ktoré takýto obsah vytvárajú, je úroveň nášho popisu reality, ktorá sa časom zlepšuje. Kvalita procedurálne generovaného obsahu preto rastie a nie je dôvod, aby sa tento trend zastavil. Kľúčovou vlastnosťou procedurálneho generovania je, že popisuje danú entitu, či už ide o textúru, model alebo efekt, pomocou sekvencie generujúcich inštrukcií a nie ako statický blok dát. Tie je potom možné vykonať až vtedy, keď je generovaný objekt potrebný. Zároveň ich je možné parametrizovať a prideľovať tak vytváraným objektom rôzne vlastnosti [13]. Takýmto spôsobom získame obsah prakticky neobmedzenej veľkosti. Príkazy pre jeho vytvorenie majú navyše veľmi nízke pamäťové nároky. Napríklad pri vytváraní procedurálne generovaného vesmíru je možné zadefinovať funkciu parametrizovanú pozíciou hráča, pričom je nutné vytvoriť len tú časť sveta, ktorú užívateľ vidí. Keďže generujúca funkcia sa nemení a je deterministická, pri opakovanej návšteve toho istého miesta je vytvorené vždy rovnaké prostredie. Tvorcovia hry *No Man's Sky* tento koncept využili pre vytvorenie vesmíru, v ktorom sa nachádza 1.8×10^9 rôznych planét s procedurálne generovanou faunou a flórou [16].

Cieľom práce je prezentovať niektoré z týchto metód pri vytváraní a zobrazovaní procedurálne generovaného prírodného terénu. Pomocou knižnice OpenGL bude vytvorené grafické demo veľkosťou obmedzené na 64kB, čo bude demonštrovať nízke nároky na veľkosť spustiteľného súboru pri využití procedurálneho generovania. Aplikácia bude vykreslovať procedurálne generovaný terén metódou ray marching v reálnom čase. Výhodou ray marchingu je, že dokáže na rozdiel od rasterizácie zobraziť terén, ktorý je popísaný implicitne (napr. výškovou mapou). Vyhne sa tak prevodu procedurálne generovaného sveta na polygonálny model, ktorý by bol zdĺhavý a mal vplyv na rýchlosť aplikácie. Výšková mapa generujúca terén a ďalšie textúry použité na jeho dotvorenie sa budú vytvárať výlučne vo

fragment shaderi bežiacom na grafickej karte.

Práca začína v kapitole 2 stručným popisom konceptu a formy grafického dema spolu s demoscénou, v ktorej takéto diela vznikajú. Zároveň sú vysvetlené dôvody, ktoré viedli k veľkostným obmedzeniam pre niektoré typy grafického dema. Kapitola 3 sa venuje tvorbe procedurálnych textúr a obsahuje prehľad vybraných techník, ktoré je pri nej možné použiť. Kapitola 4 sa zaoberá procedurálnym generovaním modelov. V Kapitole 5 je predstavená metóda ray tracing a jej variant ray marching, ktorá sa ďalej v práci používa, spolu s popisom ich rozdielov, výhod a nevýhod. Kapitola 6 obsahuje návrh štruktúry aplikácie so zameraním na shadery, ktoré sú jej kľúčovou časťou. Kapitola 7 popisuje samotnú implementáciu aplikácie. Jej prvá časť je zameraná na implementáciu algoritmu ray marching vo fragment shaderi grafickej karty. Popisuje sa v nej základná verzia ray marchingu spolu s jej možnými vylepšeniami. Ďalej je popísaný osvetľovací model aplikácie a možnosti antialiasingu pri použití ray marchingu. V závere kapitole je vysvetelený spôsob procedurálneho generovania použitý v práci. V kapitole 8 sa nachádza nachádza zhodnotenie rýchlosti výpočtu výsledného programu vychádzajúce z vykonaných testov.

Kapitola 2

Grafické demo a demoscéna

Demoscéna [23] je celosvetová komunita jedincov, ktorí sa zaujímajú o tvorbu grafického dema. Demo je relatívne krátky, väčšinou neinteraktívny počítačový program, zobrazujúci obrazový a zvukový obsah bežiaci v reálnom čase. Demo komunita vznikla koncom 70. rokov ako dôsledok zvyšujúcej sa dostupnosti osobných počítačov a sprístupnenia výpočtovej technológie domácnostiam a bežným užívateľom.

Demoscéna má korene v pirátskych skupinách, ktoré rozširovali ilegálne kópie hier a iného softwaru. Sprievodným prvkom týchto kópií boli krátke audiovizuálne práce, ktorých účelom bolo propagovať príslušnú pirátsku skupinu. Sú známe ako *crack intros*. Postupom času viedla táto, pôvodne nie až tak podstatná časť pirátskej činnosti, k vzniku inej komunity, zameranej len na ich programovanie a tvorbu. Táto komunita začala byť známa pod názvom demoscéna a programy, ktoré v nej vznikali známe ako grafické demá.

2.1 Grafické demo

Grafické demo je tiež možné definovať ako multimediálnu prezentáciu v reálnom čase alebo na jeho priblíženie použiť prirovnanie k hudobnému videu. Veľkosť výsledného spustiteľného súboru je stále relevantným meradlom, podľa ktorého sa produkty demoscény rozlišujú. Reálne pamäťové obmedzenia počítačov v období vzniku demoscény znamenali, že aj samotné demá museli byť veľkostne obmedzené. S pokrokom technológie (väčšie rozlíšenie obrazovky, zväčšenie pamätí atď.) sa maximálna povolená veľkosť dema zvyšovala. Tieto pomerne benevolentné obmedzenia dovoľujú autorom použiť v deme kvalitné obrázky, hudbu a videoklipy. Optimalizácia veľkosti teda nie je pri ich tvorbe taká podstatná.

2.2 Grafické intro s obmedzenou veľkosťou

Na rozdiel od dema je pri tvorbe grafického intra veľkosť výsledného spustiteľného súboru veľmi podstatná aj v súčasnosti. Najtypickejšie veľkostné kategórie na súťažiach sú 64 kB a 4 kB, ale existujú aj nižšie kategórie (1 kB, 256 B či dokonca 64 B)[1]. Tieto kategórie určujú maximálnu veľkosť, akú môže mať vstup do súťaže v tejto triede. Dôvodom ponechania týchto obmedzení aj napriek technologickému pokroku je súťaživosť členov demoscény. Pre autorov, ktorí sa do súťaží zapájajú, sa javí vytvorenie audiovizuálnej prezentácie na vysokej úrovni v obmedzenom pamäťovom priestore ako hodnotnejšie, než vytvorenie podobného diela bez obmedzenia. Dá sa povedať, že zámerom umelého obmedzenia pri tvorbe je snaha

[1] Pre porovnanie: súbor Microsoft Word 2013 obsahujúci jeden znak má na disku veľkosť 12 kB.

o povzbudenie kreativity a prekonávanie hraníc média. Podobným príkladom takýchto obmedzení z oblasti umenia môže byť origami alebo haiku.

Dané obmedzenia viedli k špecifickým postupom, ktoré museli byť pri tvorbe intra s obmedzenou veľkosťou vyvinuté. Bežnou technikou je skomprimovanie spustiteľného súboru spolu s používanými dátami. Výsledný súbor sa po spustení najprv dekomprimuje a potom je aktivovaný samotný program. Obrázky a hudbu vo vyššom rozlíšení však nie je možné použiť ani pri aplikácii kompresie. Na druhej strane je je veľkosť programového kódu relatívne malá, pričom však jeho využitie pre procedurálne generovanie obsahu poskytuje veľa možností.

Bežné aplikácie pri zobrazovaní 3D scény pracujú s polygonálnymi modelmi, ktoré sú vyžadované rasterizáciou. Každý objekt v scéne je definovaný (zväčša v súbore) ako postupnosť vertexov[2]. Obmedzená veľkosť grafického intra však jeho tvorcom takéto modely neumožňuje použiť, resp. umožňuje týmto spôsobom použiť len tie najjednoduchšie. Na zobrazenie výsledného obrazu sú pri tejto technike navyše potrebné ďalšie dáta informujúce shader o tom, ako má byť model zafarbený a osvetlený a aké normály sa majú pri tomto výpočte použiť. Tieto a ďalšie informácie, ktoré sa nenachádazajú vo vertexoch, sú najčastejšie uložené v textúrach. Použitie dostatočne kvalitných textúr načítaných zo súborov nie je (rovnako kvôli veľkostnému limitu) vhodné. Grafické intrá sa preto snažia tento problém obchádzať rôznymi spôsobmi. V kombinácii s rýchlym procesorom, a najmä grafickou kartou, je možné procedurálnym generovaním docieliť pôsobivé výsledky aj bez kvalitných predpripravených dát.

[2]Vertex je dátová štruktúra ukladajúca niektoré atribúty ako je napr. pozícia v priestore

Kapitola 3

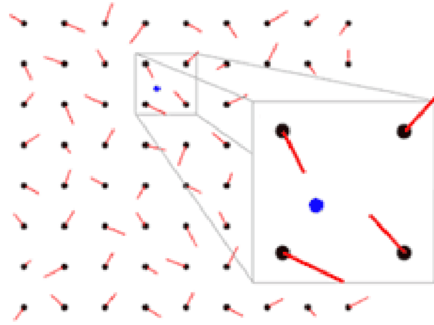
Procedurálne generovanie textúr

Podkladom pre procedurálne generovanie mnohých typov obsahu je koncept matematickej funkcie. Jej vstupnou hodnotou sú súradnice (ich počet závisí od dimenzionality funkcie) a výstupom je hodnota (alebo viac hodnôt), ktorá môže byť interpretovaná ako farba, výška terénu alebo iná veličina. Funkcie je navyše možné parametrizovať. Procedurálne generované textúry môžu byť vytvorené za behu programu, a to len v aktuálne potrebnej veľkosti. Z tohto dôvodu majú nízke pamäťové nároky a sú vhodné aj pre použitie v grafickom intre. Okrem demoscény sa často používajú v hrách alebo filmoch pri tvorbe realistických materiálov. V tejto kapitole bude predstavených niekoľko metód, ktoré je možné pri tvorbe procedurálnych textúr použiť.

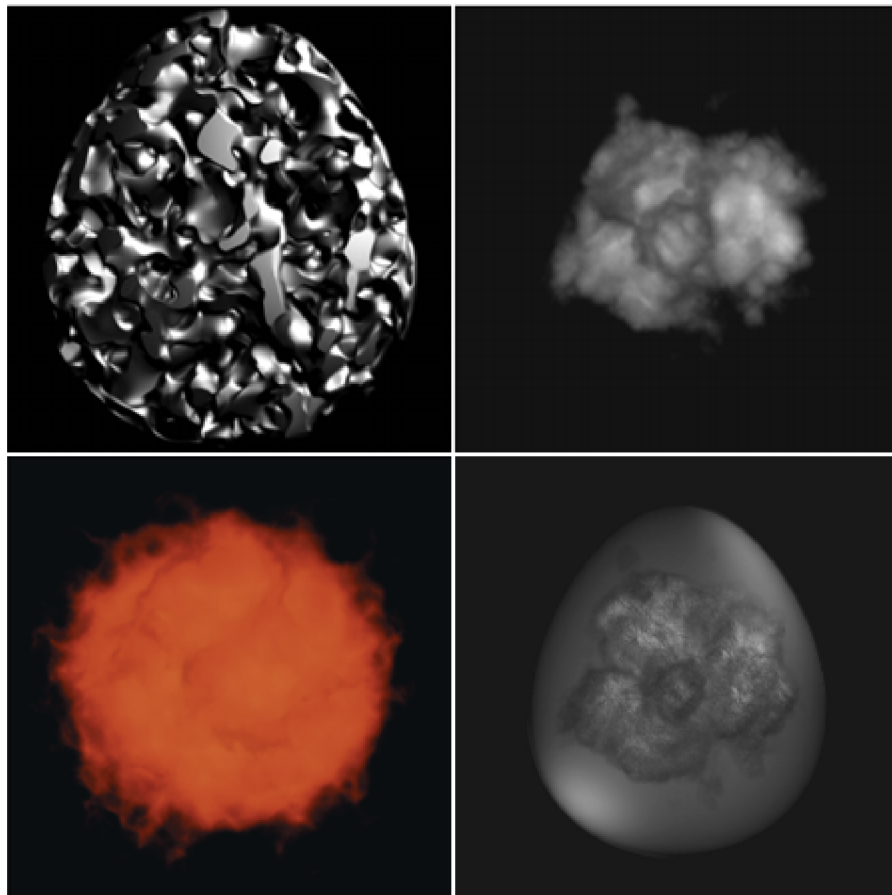
3.1 Perlinov šum

Perlinov šum [17] je funkcia generujúca pseudonáhodné hodnoty (pre rovnaké vstupy budú aj vrátené hodnoty vždy rovnaké) s plynulými prechodmi (neobsahuje ostré hrany). Kombinovaním s rôznymi matematickými výrazmi je možné jeho použitím vytvoriť veľké množstvo procedurálnych textúr. Textúry, ktoré Perlinov šum využívajú (napríklad pre povrch objektov, oheň, dym alebo oblaky), sú často používané s cieľom zvýšiť dojem prirodzenosti imitovaním kontrolovanej náhodnosti, ktorá sa objavuje v prírode. Pre tieto vlastnosti je tiež často využívaný v počítačových demách. Perlinov šum je zvyčajne implementovaný ako 2D, 3D alebo 4D funkcia, ale môže byť definovaný pre ľubovoľný počet dimenzií. Funkcia vracia reálne číslo a je volaná pre všetky body priestoru, pre ktorý má byť šum vygenerovaný.

Pre výpočet n-dimenzionálneho Perlinovho šumu je najprv vytvorená n-dimenzionálna mriežka. Každému priesečíku mriežky je priradený pseudonáhodný n-dimenzionálny jednotkový vektor určujúci gradient v danom bode. Pri volaní funkcie s n-dimenzionálnym argumentom (bodom v priestore) je ďalším krokom vyhodnotenie, do ktorej bunky mriežky daný bod spadá. Ďalej sú vypočítané vektory vzdialenosti medzi bodom a všetkými rohmi tejto bunky. Medzi týmito vektormi a vektormi gradientu v príslušnom rohu bunky je vypočítaný skalárny súčin. V 2 dimenziách ide o výpočet 4 vektorov vzdialenosti a 4 skalárnych súčinov, v 3 dimenziách 8 vektorov vzdialenosti a 8 skalárnych súčinov. Funkcia má teda pomerne vysokú zložitosť $\mathcal{O}(n \log n)$. Medzi skalárnymi súčinmi je ďalej vykonaná interpolácia. Každému rohu bunky je pridelená určitá váha podľa vzdialenosti bodu od daného rohu. Funkcia vracia vážený priemer vypočítaných skalárnych súčinov.



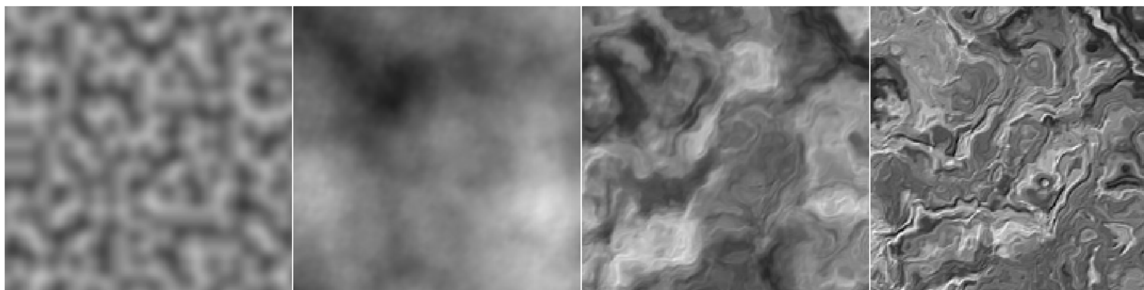
Obrázok 3.1: 2D mriežka pre Perlinov šum. Rohy buniek obsahujú vektory gradientu (prevzaté z [11])



Obrázok 3.2: Textúry vygenerované pomocou Perlinovho šumu (prevzaté z [8])

3.2 Fractional Brownian Motion

Fractional Brownian motion [8] vzniká kombináciou určitého počtu Perlinových šumov rôznej frekvencie a amplitúdy. Aj keď Perlinov šum pomerne dobre vystihuje kontrolovanú náhodnosť v prírode, nevie plne vyjadriť niektoré nepravidelnosti, ktoré sa v nej vyskytujú. Fractional Brownian motion je však pre popísanie fraktálovej povahy mnohých úkazov v prírode vhodný (napr. pri popise miery variácie v teréne: veľká miera – hory, stredná miera – skaly, nízka miera – kamene). Najvyšší podiel na výslednej hodnote má šum s nízkou frekvenciou a vysokou amplitúdou, ďalšie oktávy majú väčšinou dvojnásobne nižší prínos ako predchádzajúca oktáva [9]. Frekvencia ďalšej úrovne šumu sa väčšinou volí ako dvojnásobok predchádzajúcej (názov oktáva sa používa kvôli tomu, že podobnú vlastnosť majú aj hudobné oktávy).

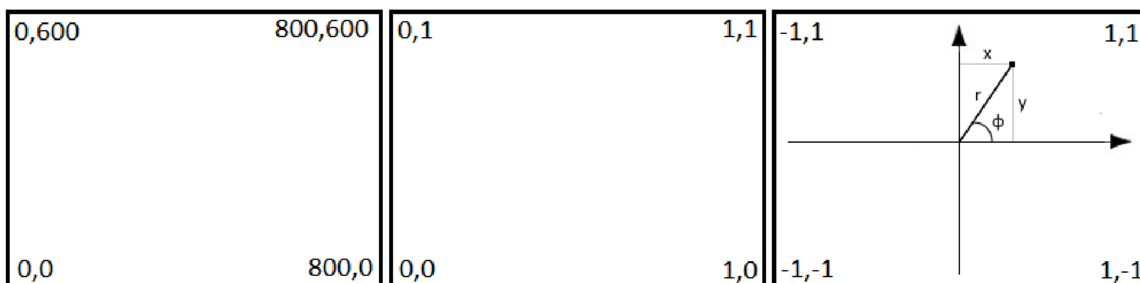


Obrázok 3.3: Textúry vygenerované rôznymi úpravami definičného oboru Perlinovho šumu. Zľava: 1. základný Perlinov šum (prevzaté z [17]), 2. $f(x) = p(x)$ - súčet niekoľkých oktáv Perlinových šumov, 3. $f(x) = p(x + p(x))$ - šum získaný použitím šumu z 2 ako vstupu pre ďalší Perlinov šum, 4. $f(x) = p(x + p(x + p(x)))$. (2, 3 a 4 prevzaté z [19])

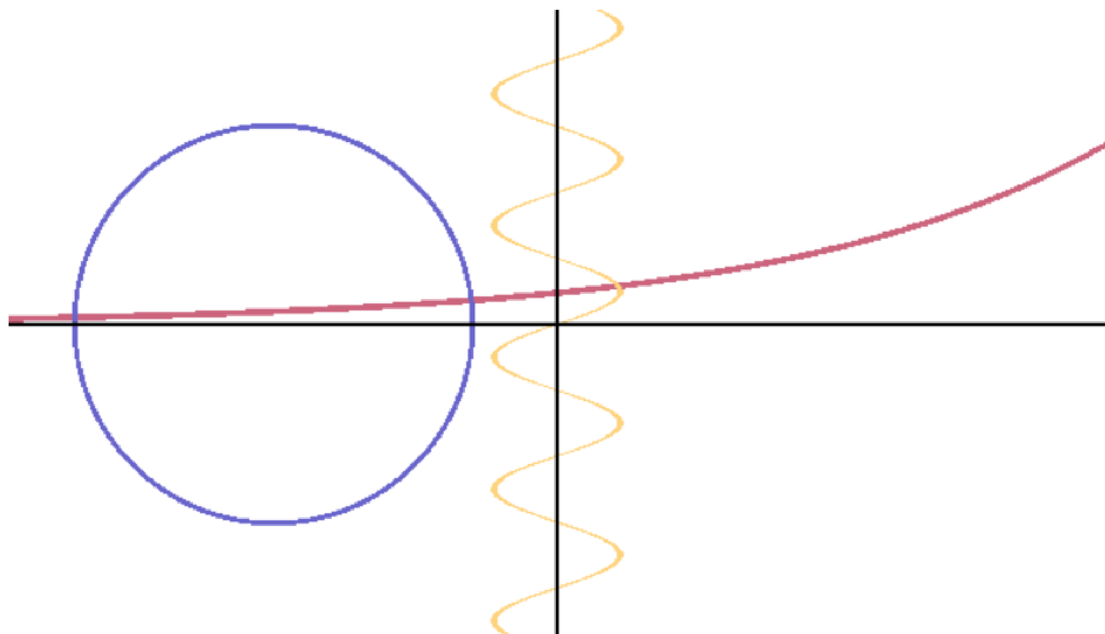
3.3 Kreslenie pomocou matematických predpisov

Táto metóda využíva pri tvorbe textúry matematické predpisy a vytvára ju priamo vo fragment shaderi grafickej karty. Vstupom predpisov je pozícia aktuálneho fragmentu, prípadne aktuálny čas, a výstupom je množina bodov, ktorým bude pridelená určitá farba. Pre každý fragment spracúvaný fragment shaderom je dostupná informácia o window-space pozícii pixelu, ktorý fragmentu zodpovedá v okne, do ktorého sa zobrazuje. Pre zjednodušenie práce a odstránenie závislosti na veľkosti okna je pozícia pixelu normalizovaná vydelením veľkosťou rozlíšenia okna (získame hodnoty v intervale (0, 1) v x-ovej aj y-ovej osi, pričom počiatok [0, 0] je vľavo dole). Ďalej je možné súradnicový systém transformovať, napríklad do systému, v ktorm sú hodnoty na oboch osách v intervale (-1, 1) a počiatok je v strede obrazovky v bode [0,0]. Z pozície pixelu je možné dopočítať korešpondujúce polárne súradnice. Tie sú pri tvorbe niektorých efektov vhodnejšie ako kartézske súradnice.

Aj keď na obrazovke sú súradnice pixelov označené hodnotami x a y, nie je nutné, aby vstupom pre funkciu bola len hodnota súradnice x. Vstupom môže byť aj hodnota súradnice y, prípadne iné odvodené hodnoty (napr. zmienené polárne súradnice). V prípade hodnoty súradnice y ako vstupu je následkom to, že funkčné hodnoty nie sú zobrazené na y-ovej osi (konvencia v dvojrozmernej karteziánskej sústave), ale na x-ovej osi. Na obrázku 3.5 je zobrazených niekoľko matematických predpisov spolu s kódom 3.1 vo fragment shaderi GLSL, ktorý im zodpovedá.



Obrázok 3.4: Príklad prevodu pozície pixelu z window-space (okno má veľkosť 800x600). Pozícia je najprv normalizovaná a následne je transformovaná do klasického kartézskeho súradnicového systému s počiatkom v strede obrazovky. Z pozície pixelu je možné dopočítať polárne súradnice



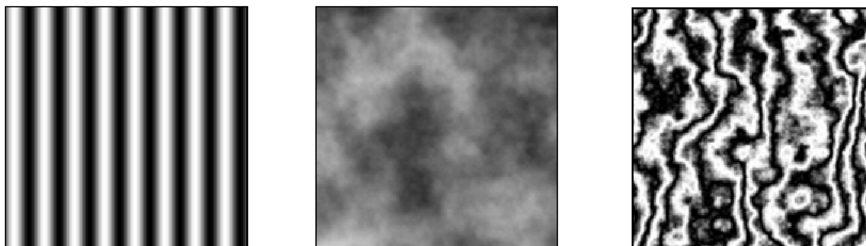
Obrázok 3.5: Zobrazenie predpisov $y = 0.1e^x$, $x = 0.2sin(15y)$ a $(x + 0.9)^2 + y^2 = 0.4$

```
vec2 pixelPosition = gl_fragCoord.xy / iResolution.xy;
pixelPosition = 2.0*p - 1.0;
vec2 p = pixelPosition;

float f1 = 0.1*exp(p.x);
if (abs(p.y - f1) < 0.01)
    color = red;

float f2 = (p.x + 0.9)*(p.x + 0.9) + p.y*p.y - 0.4;
if (abs(f2) < 0.01)
    color = blue;

float f3 = 0.2*sin(15.0*p.y);
if (abs(p.x - f3) < 0.01)
    color = yellow;
```

Kód 3.1: Fragment kódu v GLSL, ktorý zodpovedá obrázku 3.5

3.4 Kombinácia funkcie a šumu

Matematické funkcie je možné kombinovať so šumami a vytvárať tak textúry imitujúce prírodné materiály. Veľmi jednoduchým spôsobom je takto možné vytvoriť napríklad textúru pripomínajúcu mramor. Súradnice obrazovky sú posunuté o offsety dané funkciou generujúcou fractional Brownian motion. Výsledná hodnota je použitá ako vstup pre sínus so zvýšenou frekvenciou.



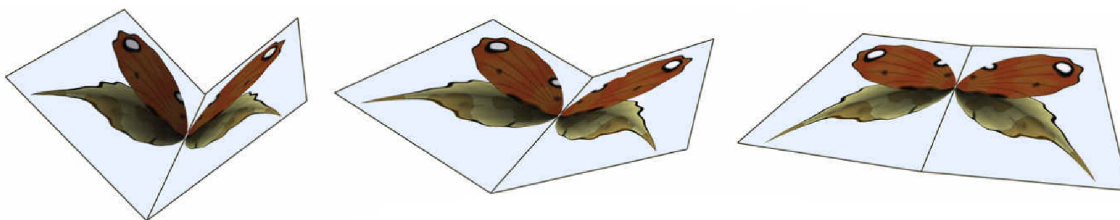
Obrázok 3.6: Textúry funkcie sínus, fractional Brownian motion a textúra vytvorená ich kombináciou (prevzaté z [15])

```
float fbm = fractionalBrownianMotion(5.0*p);
float sine = (1.0 + sin(50.0*p.x)) / 2.0;
float combination = (1.0 + sin((p.x + fbm / 2.0)*50.0)) / 2.0;
```

Kód 3.2: Fragmenty kódu generujúceho textúry na obrázku 3.6

3.5 Zobrazenie textúry na modeli v 3D scéne

Vytvorená procedurálna textúra môže byť nanesená na model v 3D scéne. Na obrázku 3.7 je textúra motýľa vytvorená metódou kreslenia matematickými predpismi. Textúra je síce statická, ale dvojica obdĺžnikov, na ktoré je nanesená, sa v čase scénou pohybuje. Zároveň sa v čase mení uhol zovretý obdĺžnikmi, čo vytvára dojem kmitajúcich krídel.



Obrázok 3.7: Procedurálne vytvorená textúra krídel je nanesená na dva obdĺžniky. Ich pozícia v scéne spolu s uhlom nimi zovretým sa v priebehu času mení. Takto je možné jednoducho simulovať pohyb motýľa (prevzaté z [18])

3.6 Fraktály

Fraktál [8] je prírodný úkaz alebo matematická množina, ktorá demonštruje opakujúci sa vzor zobraziteľný na rôznej úrovni (meradle priblíženia). Medzi rôznymi úrovňami priblíženia fraktálu môže byť veľká podobnosť alebo dokonca úplná zhoda. Fraktály sú ukážkou toho, ako je možné z jednoduchého princípu vytvoriť komplexný (v prípade Juliovej a Mandelbrotovej množiny dokonca nekonečne detailný) vzor.

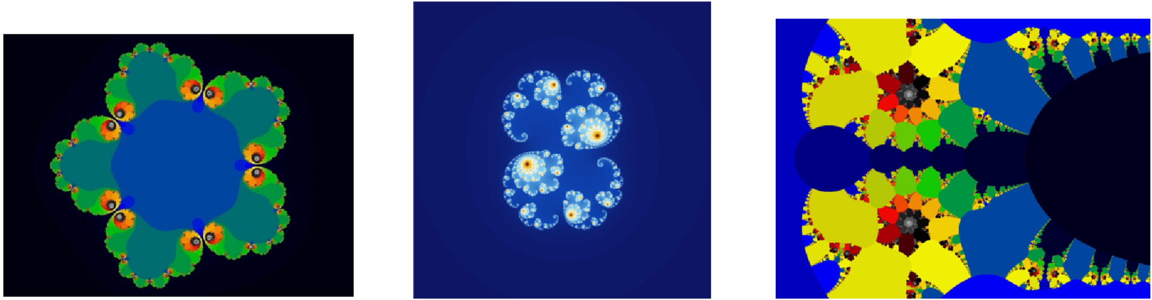
Mnohé prírodné úkazy, ktoré vykazujú na prvý pohľad vysokú úroveň komplexnosti, disponujú fraktálovou symetriou. Neznamená to však, že pri rôznom priblížení vyzerajú tieto úkazy rovnako, podobnosť môže byť len v hrubom tvare. Napríklad pre oblaky platí, že väčšia časť oblaku sa kvalitatívne podobá jeho menšej časti. Vetvy stromov, siete tokov riek alebo blesky sú ďalšími javmi, ktoré majú fraktálový charakter. Práve preto procedurálne generované textúry a modely, ktoré sú vytvorené pomocou fraktálového princípu, vierohodne pripomínajú objekty reálneho sveta. V nasledujúcom texte sú popísané a zobrazené matematické množiny, ktoré majú fraktálové vlastnosti a obrázky pomocou nich vytvorené je možné teoreticky približovať do nekonečna so zachovaním detailu zobrazenia (prakticky je úroveň priblíženia obmedzená výkonom počítača, ktorý množinu zobrazuje).

3.6.1 Juliove množiny

Juliove množiny [10] sú výsledkom štúdia iterácie racionálnych funkcií komplexných čísel. Ak $f(x)$ je funkcia, tak budeme túto funkciu iterovať pri jej počiatočnej aplikácii na hodnotu $x = a_0$. Nech $a_1 = f(a_0)$ je prvý iterát, $a_2 = f(a_1) = f(f(a_0))$ je druhý iterát atď. Zvažujeme teda nekonečnú postupnosť čísel

$$a_0,\ a1 = f(a_0),\ a_2 = f(a_1),\ a_3 = f(a_2),\ \ldots \tag{3.1}$$

V závislosti od počiatočnej hodnoty a_0 sa výsledná postupnosť môže správať dvoma spôsobmi: buď ostane absolútna hodnota iterátov ohraničená bez ohľadu na vykonaný počet iterácií alebo sa hodnoty postupnosti začnú blížiť k nekonečnu. Množina čísel a_0, pre ktoré sa postupnosť správa prvým spôsobom, sa nazýva Juliova množina. Ak absolútna hodnota niektorého člena postupnosti prekročí určitú medznú hranicu, je možné jednoznačne povedať, že počiatočný iterát a_0 nepatrí do Juliovej množiny. Pre hodnoty a_0, ktorých postupnosti v aktuálne vypočítanej iterácii nezačali divergovať, však nemôžme s určitosťou tvrdiť, že do Juliovej množiny patria. Pri ďalších vypočítaných iteráciách totiž môžu im odpovedajúce postupnosti začať divergovať. Jedná sa teda len o aproximáciu Juliovej množiny s určitou presnosťou, ktorá je daná počtom vypočítaných iterácií. Pri zobrazovaní konkrét-



Obrázok 3.8: Juliove množiny zobrazené v komplexnej rovine. Iterovanými funkciami sú zľava $f(x) = x^5 + 0.544$, $f(x) = x^2 + 0.285 + 0.01i$ a $f(x) = e^x - 0.65$ (prevzaté z [4], [3] a [2])

nej Juliovej množiny si tiež vyberieme len určitú podmnožinu komplexných čísel a_0, pre ktorú budeme výpočet realizovať.

Každé číslo a_0 má pridelenú súradnicu na obrazovke, ktorá zodpovedá jeho pozícii v komplexnej rovine. Všetky hodnoty a_0, ktoré vedú ku konvergujúcej postupnosti čísel, patria do Juliovej množiny a ich príslušným pixelom je pridelená určitá farba. Hodnoty a_0, ktoré vedú k divergujúcej postupnosti čísel, sú ďalej farebne rozdelené podľa rýchlosti, s akou začala z nich odvodená postupnosť divergovať. Počet iterácií teda zodpovedá úrovni detailu v obrázku.

3.6.2 Mandelbrotova množina

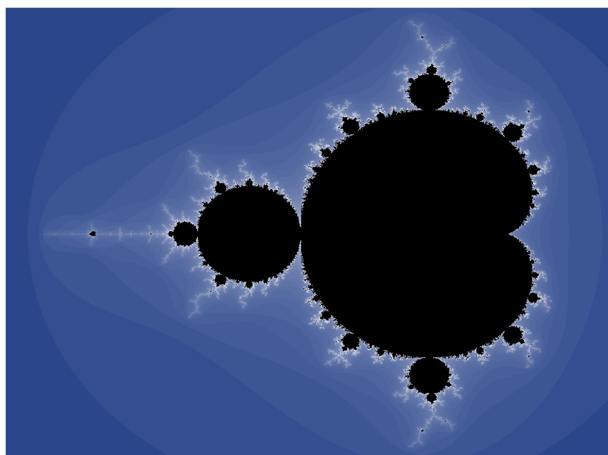
Mandelbrotova množina [25] je výsledkom štúdia celej rodiny kvadratických polynómov parametrizovaných komplexnou premennou μ:

$$f(x) = x^2 - \mu \tag{3.2}$$

Pri zmene μ sa mení aj zodpovedajúca Juliova množina funkcie $f(x)$ v komplexnej rovine. Niektoré z týchto Juliových množín budú súvislé a niektoré budú nesúvislé. Táto vlastnosť teda rozdelí parametrickú rovinu μ na dve časti. Hodnoty parametru μ, pre ktoré je zodpovedajúca Juliova množina súvislá, tvoria v parametrickej rovine Mandelbrotovu množinu. Pri zobrazovaní Mandelbrotovej množiny udáva hodnota μ pozíciu pixelu v komplexnej rovine. Sledujeme, či postupnosť

$$a_0 = 0,\ a1 = f(a_0),\ a_2 = f(a_1),\ a_3 = f(a_2),\ \ldots \tag{3.3}$$

začne pri vypočítanom stupni iterácie divergovať. Podmienkou konvergencie je, že pre všetky členy postupnosti a_n platí $|a_n| \leq 2$. Vykresľovanie prebieha podobne ako pri zobrazovaní Juliových množín. Body μ v komplexnej rovine a im zodpovedajúce pixely na obrazovke, farebne rozlišujeme podľa príslušnosti k Mandelbrotovej množine resp. podľa toho, v ktorom kroku pre ne začala postupnosť iterátov divergovať.



Obrázok 3.9: Vykreslenie Mandelbrotovej množiny v parametrickej rovine μ (prevzaté z [7])

Kapitola 4

Procedurálne generovanie modelov

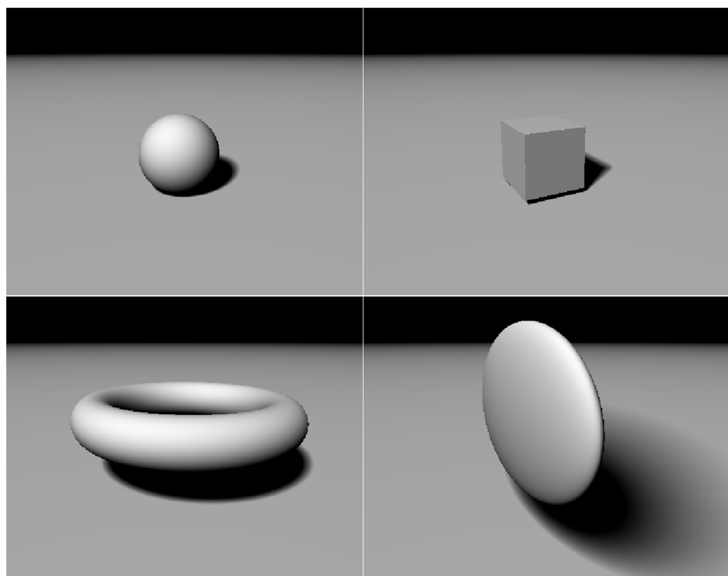
Procedurálne generované modely využívajú určitý predpis pre zostrojenie samotnej geometrie objektu. Pri ich tvorbe je často možné použiť aj metódy prezentované v kapitole 3, a to buď pre generovanie samotných modelov (tvorba procedurálneho sveta pomocou šumových funkcií) alebo ako doplnok inej techniky s cieľom zvýšiť realistický vzhľad objektu. Táto kapitola sa zameriava na rodinu objektov popísaných pomocou signed distance functions. Tie je možné zobraziť algoritmom ray marching popísaným v časti 5.2.

4.1 Signed distance functions

Vstupom pre signed distance functions [21] je bod (najčastejšie v trojrozmernom priestore) a vrátenou hodnotou je znamienková vzdialenosť tohto bodu k povrchu objektu, ktorý daná funkcia vzdialenosti popisuje. Môžu nastať tri prípady: ak sa bod nachádza mimo telesa, vracia funkcia kladnú vzdialenosť. Ak sa nachádza vo vnútri telesa, je vrátená záporná vzdialenosť, a ak sa bod nachádza presne na hranici oddeľujúcej vnútornú a vonkajšiu časť telesa, je vrátená nulová vzdialenosť. Implementácie funkcií vzdialenosti pre vybrané geometrické objekty sú v kóde 4.1.

```
float sphere(vec3 p, float s)
{
    return length(p) - s;
}
float box(vec3 p, vec3 b)
{
    vec3 d = abs(p) - b;
    return min(max(d.x, max(d.y, d.z)), 0.0) + length(max(d, 0.0));
}
float torus(vec3 p, vec2 t)
{
    vec2 d = vec2(length(p.xz) - t.x, p.y);
    return length(d) - t.y;
}
float ellipsoid(vec3 p, in vec3 r)
{
    return (length(p / r) - 1.0) * min(min(r.x, r.y), r.z);
}
```

Kód 4.1: Signed distance functions niektorých jednoduchých objektov (prevzaté z [21])

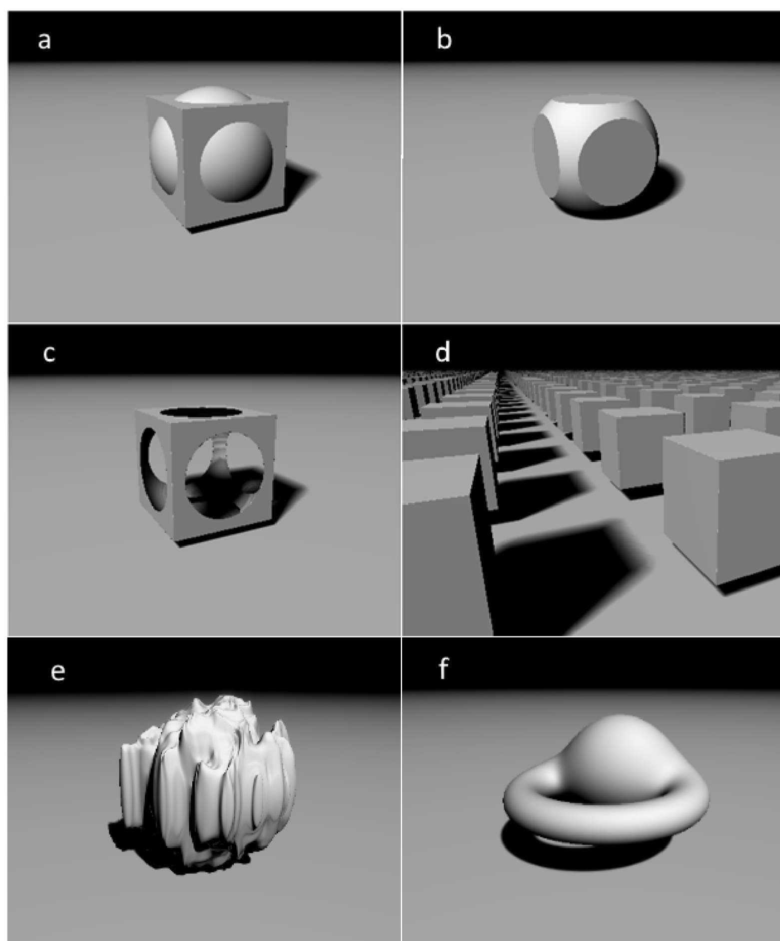


Obrázok 4.1: Jednoduché objekty popísané pomocou signed distance functions v kóde 4.1 a vykreslené algoritmom ray marching

Funkcie vzdialenosti je možné vzájomne kombinovať pomocou rôznych operácií (napr. množinové zjednotenie, prienik alebo rozdiel) a vytvárať tak z jednoduchých objektov komplikovanejšie scény. Obzvlášť zaujímavé je aplikovanie operácie modulo na body scény pred tým, než sú pre ne vyhodnotené funkcie vzdialenosti. Tento postup umožňuje nekonečné opakovanie objektov s určitými rozostupmi, pričom však náročnosť výpočtu pri zobrazovaní ostáva prakticky nezmenená. Aplikácia týchto operácií nad guľou, kvádrom a tórusom je zobrazená na obrázku 4.2. Operácia plynulého zjednotenia objektov je implementovaná pomocou polynomiálnej interpolácie.

```
float union(float d1, float d2)
{
    return min(d1, d2);
}
float intersect(float d1, float d2)
{
    return max(d1, d2);
}
float subtract(float d1, float d2)
{
    return max(-d2, d1);
}
float smoothUnion(float d1, float d2)
{
    float h = clamp(0.5 + 0.5*(d1 - d2), 0.0, 1.0 );
    return mix(d1, d2, h) - h*(1.0 - h);
}
vec3 repeat(vec3 p, vec3 c)
{
    return mod(p, c) - 0.5*c;
}
```

Kód 4.2: Operácie aplikovateľné nad signed distance functions (prevzaté z [21])



Obrázok 4.2: Ukážka operácií nad signed distance functions z kódu 4.2: (a) zjednotenie gule a kvádru, (b) prienik gule a kvádru, (c) rozdiel kvádru a gule, (d) aplikácia operácie modulo na súradnicove v scéne, (e) signed distance function gule modifikovaná použitím šumovej funkcie, (d) plynulé zjednotenie elipsoidu a gule

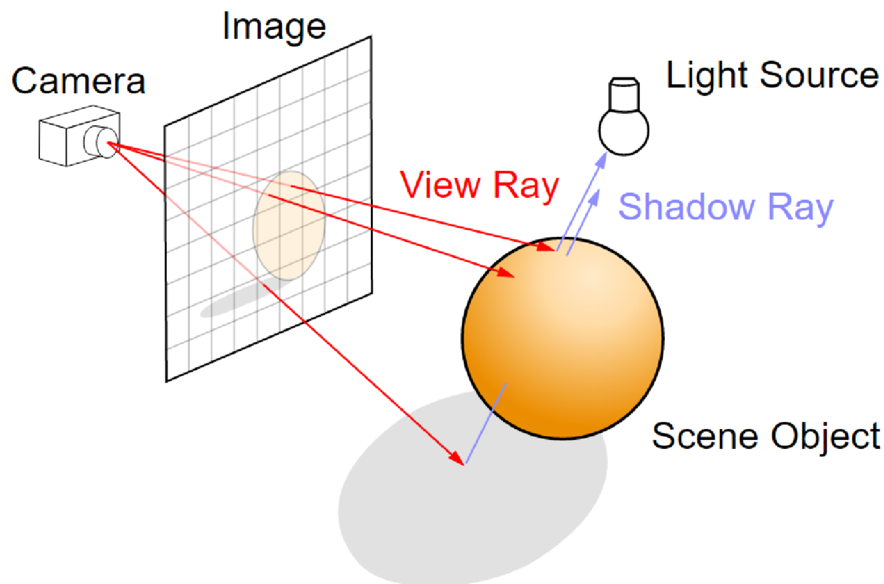
Kapitola 5

Zobrazovacie metódy

Hlavnou použitou zobrazovacou metódou bude pri tvorbe práce variant techniky ray tracing zvaný ray marching, pričom výpočet zobrazenia bude prebiehať na grafickej karte. V súčasnosti používané grafické karty sú prispôsobené pre rasterizáciu, tá však nie je pre túto prácu vhodná, keďže využitie polygonálnych modelov by viedlo k zvýšeniu pamäťovej náročnosti aplikácie.

5.1 Ray tracing

Ray tracing [24] je zobrazovacia technika, ktorá vypočítava farbu pre každý pixel osobitne. Pre každý pixel je nutné nájsť objekt, ktorý sa na jeho pozícii zobrazí. Cez všetky pixely smeruje do scény v inom smere lúč a každý objekt, ktorý je cez daný pixel videný, musí tento lúč pretnúť. V pixeli sa zobrazí objekt, ktorý preťal tento lúč a zároveň je zo všetkých objektov, ktoré zdieľajú túto vlastnosť, najbližšie ku kamere.



Obrázok 5.1: Algoritmus ray tracing [1]

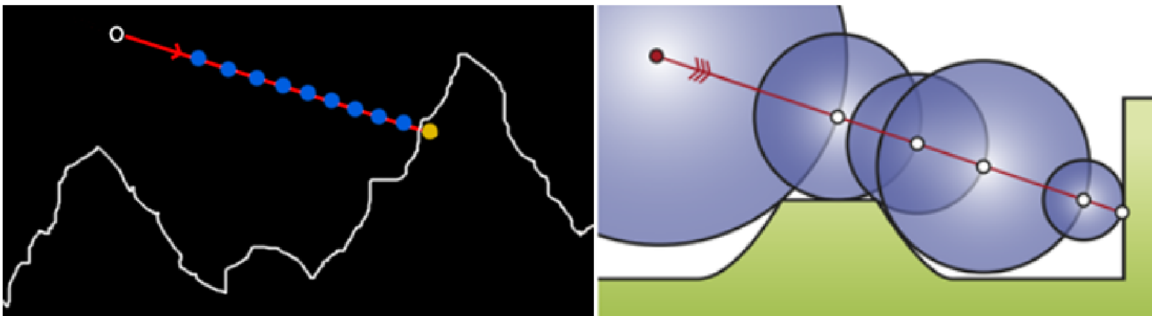
Po nájdení zobrazovaného objektu nasleduje pre daný pixel výpočet farby, ktorý využíva polohu bodu v ktorom došlo k prieniku s objektom, normálu objektu, polohy svetiel v scéne

a v závislosti na požadovaných efektoch dodatočné informácie. Zo zisteného miesta stretu s objektom je možné vyslať sekundárne lúče smerujúce do zdrojov svetla. Ak sekundárny lúč nedorazí do cieľa ale pretne nejaký objekt, tak je bod, z ktorého bol lúč vyslaný, v tieni.

Pomocou ray tracingu je možné dosiahnuť vysokú kvalitu zobrazenia a v porovnaní s rasterizáciou pomerne jednoducho a elegantne implementovať aj zložitejšie efekty. Algoritmus je však pri súčasných grafických kartách príliš pomalý a nie je v takejto podobe vhodný pre zobrazovanie v reálnom čase. Jeho najpomalším článkom je analytické hľadanie priesečíkov vyslaných lúčov a objektov v scéne. Všetky objekty v scéne je teda nutné analyticky popísať, čo môže byť u zložitejších objektov komplikované.

5.2 Ray marching

Pri ray marchingu [22] sa z pozície kamery do každého pixela, podobne ako pri ray tracingu, vysielajú lúče. Prienik s objektami scény sa ale nezisťuje analyticky. Z pozície kamery sa v smere lúča vykonávajú kroky vhodnej dĺžky a po každom kroku sa zisťuje, či došlo k pretnutiu s povrchom objektu. Objekty scény môžu byť popísané buď pomocou výškovej mapy, pomocou signed distance functions alebo inak.



Obrázok 5.2: Algoritmus ray marching v základnej verzii (prevzaté z [22]) a vo verzii s premenlivou dĺžkou kroku pri použití signed distance functions (prevzaté z [12])

Metóda ray marching je citlivá na nastavanie veľkosti kroku. Pri príliš veľkom kroku sa môže stať, že lúč preskočí povrch objektu. Pri malom kroku je síce metóda presnejšia, ak však chceme, aby aplikácia pracovala v reálnom čase s prijateľným počtom snímok za sekundu, nemusí byť dostatočne rýchla. Vyváženie požiadaviek na presnosť a rýchlosť výpočtu je vhodné pre každú zobrazovanú scénu vyladiť osobitne, keďže správne nastavenie veľkosti kroku bude nutne závisieť od zobrazovanej funkcie.

Ak sú objekty v scéne popísané pomocou signed distance functions, tak vieme v každom bode scény určiť vzdialenosť k najbližšiemu z nich. Je teda možné rýchlosť ray marchingu optimalizovať nastavením dĺžky kroku na túto vzdialenosť. Takto vypočítanú dĺžku kroku musíme po každom jeho vykonaní znova prepočítať.

Kapitola 6

Návrh riešenia

Cieľom tejto práce je definovanie a vykreslenie prírodného terénu, ktoré bude prebiehať v reálnom čase na grafickej karte. Zvolenou metódou pre generovanie procedurálneho terénu je šumová funkcia fractional Brownian motion popísaná v časti 3.2. Tento typ funkcie umožňuje simulovať prirodzene vyzerajúci prírodný terén s vysokou úrovňou detailu pri rôznom priblížení.

6.1 Štruktúra aplikácie

Na úrovni procesoru zabezpečí aplikácia vytvorenie zobrazovacieho okna. Okno bude vytvorené len pomocou volania funkcií rozhrania operačného systému (z dôvodu veľkostného obmedzenia spustiteľného súboru nebudú využité žiadne externé knižnice). Vytvorené okno bude potom napojené na kontext OpenGL, a toto rozhranie bude využité pre prácu s grafickou kartou. Procesor ďalej zadá príkazy pre vykreslenie obdĺžnika, na ktorý sa bude vo fragment shaderi zobrazovať vytvorený terén. Procesor teda zabezpečuje len najnutnejšie minimum, zvyšná časť aplikácie (definovanie terénu a jeho zobrazenie) sa bude vykonávať na grafickej karte.

6.2 Návrh shaderov

Vertex shader nemá pri tomto druhu zobrazovania takmer žiadnu zodpovednosť. Jeho jedinou úlohou je spracovanie vrcholov obdĺžnika pokrývajúceho celé okno. Pixely obrazovky budú potom na tento obdĺžnik natiahnuté a jeden beh fragment shadera bude vo výsledku vytvárať farbu pre práve jeden pixel. Vertexový popis ďalších 3D modelov sa vôbec nepoužíva. Obdĺžnik je spolu textúrou zobrazený priamo ako výsledný snímok.

Fragment shader vytvára modelovaný svet v reálnom čase. Z procesoru prijme informáciu o pozícii a natočení kamery. Následne z tohto miesta a daným smerom začne vysielať lúče smerujúce do jednotlivých pixelov obrazovky, ktorá bola presunutá na pozíciu kamery. Program potom vyslané lúče krokuje a po každom kroku testuje, či došlo k prieniku s procedurálnym terénom, ktorý je definovaný pomocou funkcie fractional Brownian motion (v každom bode sveta táto funkcia vráti výšku terénu). Miesto stretu lúča s povrchom terénu je potom spolu s ďalšími informáciami o scéne (normála povrchu v bode stretu, pozícia svetla, tieniace objekty a pod.) využité na určenie farby daného pixelu.

Prvou požiadavkou na aplikáciu je, aby bola schopná plynule bežať v reálnom čase. Algoritmy vykresľovania a shadingu bude preto nutné s týmto cieľom optimalizovať. Druhou

požiadavkou je, aby bol terén zobrazený v istej kvalite. Keďže sa však jedná o demo s obmedzenou veľkosťou, nie je možné očakávať kvalitu porovnateľnú s rasterizáciou. V kapitole venovanej implementácii budú prezentované metódy, ktoré umožňujú vyvážiť obe požiadavky. Nakoniec by veľkosť spustiteľného súboru nemala prekročiť stanovený limit 64kB. V aplikácii sa preto budú využívať len zdieľané knižnice sprístupnené operačným systémom, ktorých využitie nemá vplyv na veľkosť spustiteľného súboru.

Kapitola 7

Implementácia

V prvej časti tejto kapitoly je popísaná implementácia zobrazovacích algoritmov, ktoré sú v práci použité. Popis začína základnou verziou algoritmu ray marching, ktorá je ďalej optimalizovaná s cieľom dosiahnuť interaktívnu rýchlosť zobrazovania. V rámci podkapitoly venovanej zobrazovaniu sú ďalej popísané techniky pre výpočet osvetlenia scény, výpočet tieňov a metódy antialiasingu. Podkapitola 7.2 sa zaoberá použitým osvetľovacím modelom. Podkapitola 7.3 sa venuje definovaniu vzhľadu vykresľovaného procedurálneho terénu.

7.1 Algoritmus zobrazovania

Keďže pre zobrazovanie scény je najpodstatnejší obsah fragment shadera, je mu v nasledujúcej časti vyhradená najväčšia pozornosť. Zároveň sú všetky prezentované časti kódu implementované vo fragment shaderi GLSL.

7.1.1 Inicializácia

Každý spracúvaný pixel môže pristupovať k systémovej premennej *gl_FragCoord*, ktorá udáva jeho pozíciu na obrazovke. Počiatok tohto súradnicového systému začína v ľavom dolnom rohu okna na pixeli (0, 0). Prvým krokom je transformácia takto zadefinovaného systému do intervalu (-1, 1). V prípade, že je veľkosť strán obrazovky nerovnomerná, nasleduje ešte korekcia súradníc.

```
vec2 pixelPosition = (gl_FragCoord.xy / iResolution.xy)*2.0 - 1.0;
pixelPosition.x *= iResolution.x / iResolution.y;
```

Kód 7.1: Transformácia súradníc pixelu

S takto upravenými súradnicami budeme môcť jednoducho pracovať pri vysielaní lúčov do scény. Pri práci vo fragment shaderi je procesorom sprístupnená pozícia kamery a miesto, kam je kamera nasmerovaná. Aby sme mohli cez pixel do scény vyslať lúč, musíme pomocou týchto informácií zostrojiť lokálny súradnicový systém správne natočenej kamery.

```
vec3 forward = normalize(cameraTarget - cameraPosition);
vec3 right = normalize(cross(forward, vec3(0.0, 1.0, 0.0)));
vec3 up = normalize(cross(right, forward));
vec3 rd = normalize(pixelPosition.x*right+ pixelPosition.y*up + forward);
vec3 ro = cameraPosition;
```

Kód 7.2: Zostrojenie lokálneho súradnicového systému kamery. Premenná *ro* určuje počiatočný bod lúča a premenná *rd* určuje smer lúča

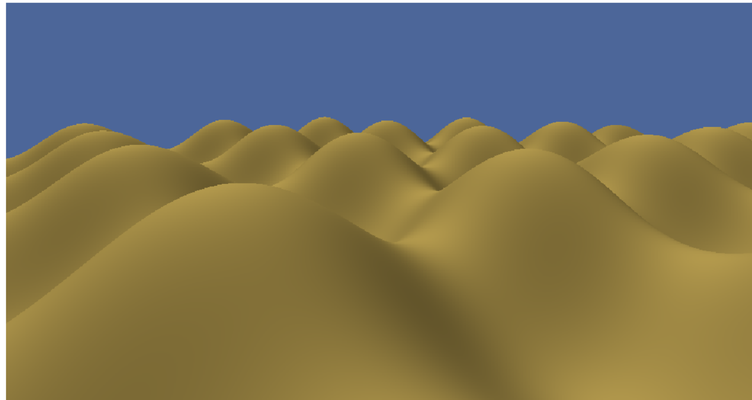
Zostrojený počiatok a smer lúča sú argumentami funkcie shade, ktorá najprv pomocou algoritmu ray marching vo funkcii castRay zistí, či lúč preťal terén scény, a podľa toho následne vyhodnotí farbu, ktorá bude pixelu pridelená. *FragmentColor* je výstupná premenná fragment shadera vracajúca farbu aktuálneho pixelu.

```
vec3 shade(vec3 ro, vec3 rd)
{
    bool terrainIntersected = castRay(ro, rd);
    if (terrainIntersected)
        return terrainColor;
    else
        return skyColor;
}
```

Kód 7.3: Funkcia shade, v ktorej sa na základe vykonaného testu na pretnutie lúča s povrchom vyhodnocuje farba pixelu. V kóde sa nachádza iba základná logika funkcie. V praxi je v nej implementovaný osvetľovací model, tieňovanie, prípadne ďalšie techniky modifikujúce farbu

7.1.2 Ray marching – naivná implementácia

Nasleduje samotný proces ray marchingu. Lúče vychádzajú z pozície kamery a postupujú v smere zadefinovanom v kóde 7.2 pohybujúc sa konštantným krokom. Po každom kroku sa porovnáva y-ová súradnica vrcholu lúču so súradnicou terénu vo výškovej mape. Ak je súradnica lúča menšia ako súradnica vo výškovej mape (resp. líšia sa len o hodnotu reprezentujúcu stanovenú presnosť zobrazenia), tak lúč preťal povrch a algoritmus vracia *true*. Ak je dosiahnutá maximálná povolená dĺžka lúča a lúč stále nepreťal povrch, tak algoritmus vracia *false*. Výsledok pri zobrazovaní výškovej mapy, ktorá je daná jednoduchou funkciou je na obrázku 7.1.



Obrázok 7.1: Vykreslenie výškovej mapy danej funkciou $10sin(0.1x)sin(0.1z)$. Obrázok je vykreslený naivnou verziou ray marchingu s pevnou dĺžkou kroku. Spôsob výpočtu osvetlenia ja vysvetlený v časti 7.2.

```
bool castRay(vec3 ro, vec3 rd, out float t)
{
    for (t = 0; t < MAX_T; t += STEP_SIZE)
    {
        vec3 point = ro + rd*t;
        if (point.y - PRECISION < map(point))
            return true;
    }
    return false;
}
float map (vec3 point)
{
    return 10.0*sin(0.1*point.x)*sin(0.1*point.z);
}
```

Kód 7.4: Prvá verzia algoritmu ray marching a mapovacia funkcia použitá pre vykreslenie obrázku 7.1

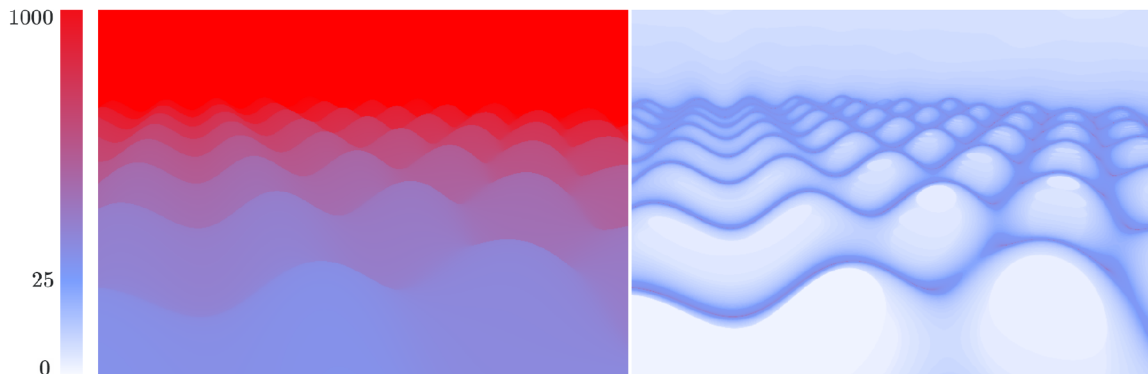
7.1.3 Ray marching s dynamickou veľkosťou kroku

Táto prvotná implementácia síce terén bez problémov vykreslila, ale je značne pomalá. Jej najväčším problémom je konštantná dĺžka kroku. Tú je síce možné zvýšiť, ale v tom prípade sa nevyhneme vo výslednom zobrazení strate kvality. Rýchlosť výpočtu môžme podstatne vylepšiť technikou navrhnutou v časti 5.2 pre vykresľovanie signed distance functions. Dĺžka kroku pozdĺž lúča nebude stanovená pevne, ale bude daná rozdielom aktuálnej výšky lúča v svete a hodnotou vo výškovej mape v tomto bode. Na rozdiel od signed distance functions táto hodnota nie je zaručene najnižšia veľkosť kroku o ktorú sa musíme pohnúť v scéne pozdĺž lúča tak, aby sme sa dostali k najbližšiemu objektu. Pri vykresľovaní výškovej mapy sa teda môže stať, že sa o takto vypočítanú hodnotu pohneme a hranicu povrchu preskočíme. V závislosti na vykresľovanej funkcii môže byť tento jav rôzne častý. V prípade, že kvalita zobrazenia začne neprijateľne klesať, je možné takto vypočítanú dĺžku kroku znížiť napríklad na polovicu.

Upravená verzia algoritmu ray marching je uvedená v kóde 7.5. Súčasne je nutné upraviť mapovaciu funkciu tak, aby vracala výškový rozdiel medzi skúmaným bodom a hodnotou vrátenou výškovou mapou. Ak zobrazíme touto verziou ray marchingu rovnakú výškovú mapu ako na obrázku 7.1, výsledok je voľným okom nerozoznateľný od verzie zobrazenej naivnou implementáciou. Rýchlosť výpočtu vykonaného týmto spôsobom je však podstatne vyššia.

Rozlíšenie okna	640x480	1366x768	1920x1080
Metóda 7.1.2	8.4 ms (119.6 FPS)	29.0 ms (34.5 FPS)	57.7 ms (17.3 FPS)
Metóda 7.1.3	0.6 ms (1689.8 FPS)	1.4 ms (729.7 FPS)	2.3 ms (426.6 FPS)

Tabuľka 7.1: Porovnanie priemernej rýchlosti výpočtu jednej snímky pre naivný ray marching a ray marching s dynamickou veľkosťou kroku. Optimalizovaná verzia je v priemere zhruba 20-krát rýchlejšia. Testovacia zostava je uvedená v kapitole 8



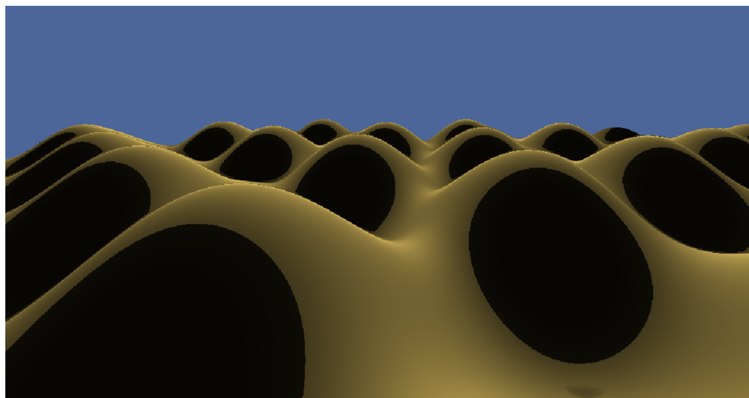
Obrázok 7.2: Porovnanie počtu vykonaných krokov algoritmu ray marching pri verzii 7.1.2 vľavo, a pri optimalizovaniej verzii 7.1.3 vpravo. Maximálny počet vykonaných krokov je obmedzený hodnotou 1000. Testovanie prebiehalo na zostave č.1 uvedenej v kapitole 8

```
bool castRay(vec3 ro, vec3 rd, out float t)
{
    t = 0.0;
    float stepSize = 0.0;
    for (int i = 0; i < MAX_STEPS; ++i)
    {
        vec3 point = ro + rd*t;
        stepSize = map(point);
        t += stepSize;
        if (stepSize < PRECISION)
            return true;
        if (t > MAX_T)
            return false;
    }
    return false;
}

float map (vec3 point)
{
    float heightMapValue = 10.0*sin(0.1*p.x)*sin(0.1*p.z);
    return point.y - heightMapValue;
}
```

Kód 7.5: Optimalizovaná verzia ray marchingu s dynamickou dĺžkou kroku. Bolo zároveň nutné upraviť aj mapovaciu funkciu tak, aby vracala rozdiel medzi skúmaným bodom (pozíciou lúča) a výškou terénu v danom mieste

7.1.4 Ostré tiene

Vzhľadom na to, že mapovacia funkcia nám poskytuje informácie o celej scéne, je možné pri použití ray marchingu veľmi jednoducho implementovať zobrazovanie tieňov. Pri zisťovaní zatienenia bodu sa použije podobný postup ako pri vrhaní lúčov z kamery. Zdrojom sekundárneho lúča bude daný bod a lúč bude smerovať do zdroja svetla. Po každom kroku je nutné kontrolovať, či lúč narazil na nejaký objekt. Ak lúč narazil, znamená to, že medzi bodom a svetlom je prekážka a objekt je v tieni. Proces je znázornený na obrázku 5.1. Ide o tzv. *hard shadow*, keďže môžu nastať len dve možnosti: buď vyslaný lúč narazí na prekážku



Obrázok 7.3: Scéna vykreslená s použitím hard shadows

a bod je zatienený alebo nenarazí a bod zatienený nie je. Použitý algoritmus je uvedený v kóde 7.6. Pri vysielaní lúča musí byť jeho počiatočná magnitúda nastavená na hodnotu blízku nule z toho dôvodu, že lúč vysielame z bodu, o ktorom vieme, že sa nachádza na povrchu objektu alebo terénu. Od tohto bodu sa teda potrebujeme hneď na začiatku vzdialiť, inak by funkcia map hneď v prvom cykle algoritmu vrátila nulovú vzdialenosť k povrchu a bod by bol nesprávne vyhodnotený ako zatienený.

```
float calculateShadow(vec3 ro, vec3 rd)
{
    float shadow = 1.0;
    float t = 0.2;
    for (int i = 0; i < MAX_STEPS; ++i)
    {
        float stepSize = map(ro + rd*t);
        if (stepSize < PRECISION)
        {
            shadow = 0.0;
            return shadow;
        }
        if (t > MAX_T)
            break;
        t += stepSize;
    }
    return shadow;
}
```

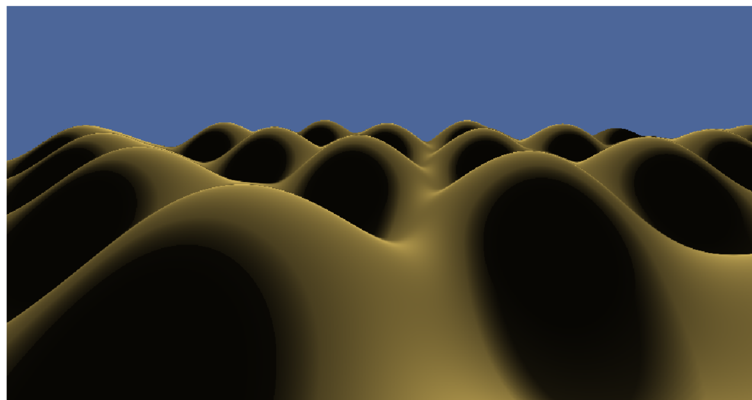
Kód 7.6: Algoritmus vysielania sekundárnych lúčov pre výpočet hard shadows

7.1.5 Mäkké tiene

Okrem lúčov, ktoré narazili na prekážku, sa berú pri výpočte *soft shadows* do úvahy aj lúče, ktoré *takmer* narazili na prekážku. Okrem ostrého tieňa tak vzniká aj spojité spektrum tieňov zvané penumbra. Čím bližšie bol lúč k pretnutiu nejakej prekážky, tým je zatienenie bodu väčšie. Zohľadňuje sa aj vzdialenosť medzi miestom eventuálnej zrážky a zatieneným bodom. Ak sa lúč takmer zrazil s objektom veľmi blízko zatieňovaného bodu, bude efekt penumbry väčší, než keby k tomu došlo ďalej pozdĺž lúča.

```
float calculateShadow(vec3 ro, vec3 rd)
{
    float k = 4.0;
    float t = 0.02;
    float shadow = 1.0;
    for (int i = 0; i < MAX_STEPS; ++i)
    {
        float stepSize = map(ro + rd *t);
        if (stepSize < PRECISION)
        {
            shadow = 0.0;
            return shadow;
        }
        if (t > MAX_T)
            break;
        shadow = min(shadow, k * stepSize / t);
        t += stepSize;
    }
    return shadow;
}
```

Kód 7.7: Algoritmus pre výpočet mäkkých tieňov. Zatienenie objektu je ovplyvnené aj lúčmi, ktoré takmer narazili na prekážku a aj tým , v akej vzdialenosti od pôvodného bodu k tomu došlo. Premenná *stepSize* zohľadňuje prvú požiadavku, premenná t druhú. Konštanta k je faktor mäkkosti tieňa (postup prevatý z [20])

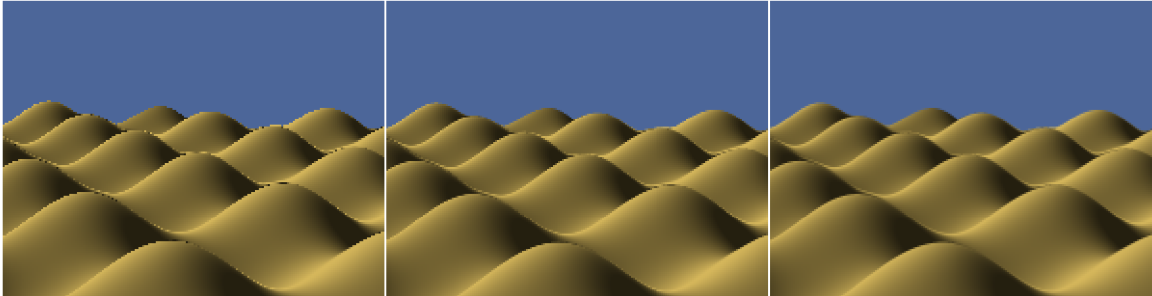


Obrázok 7.4: Scéna vykreslená s použitím soft shadows

7.1.6 Antialiasing

Vzhľadom na to, že každému pixelu okna je pridelená farba práve jedného vykresleného objektu, vzniká vo výslednom obraze aliasing. Tento jav je najviac viditeľný na obrysoch útvarov. Pri použití ray marchingu je dôvodom jeho vzniku fakt, že do scény pre každý pixel vysielame len jeden lúč, ktorý smeruje do stredu pixelu. Aby sme aliasing odstránili, proces prideľovania farby pixelu musí zohľadňovať informáciu z viacerých vyslaných lúčov.

Možným riešením je vyslanie väčšieho počtu lúčov, z ktorých každý smeruje do pixelu s inou odchýlkou. Ide v podstate o verziu supersamplingu [6] adaptovanú pre algoritmus ray marchingu. V GLSL je vo fragment shaderi súradnica stredu pixelu daná premennou



Obrázok 7.5: Porovnanie scény vykreslenej bez antialiasingu (vľavo), so 4 lúčmi na pixel (v strede) a so 16 lúčmi na pixel (vpravo). Aliasing je najviac viditeľný na vrcholkoch kopcov v najvzdialenejšej časti scény, ktorá je na obrázkoch zväčšená pomocou algoritmu nearest neighbor

gl_FragCoord. Pripočítaním vhodného offsetu k tejto premennej získame súradnice samplov s podpixelovým rozlíšením. Na každú z týchto súradníc potom aplikujeme rovnaký postup ako v kóde 7.2, čím získame lúče smerujúce do rôznych častí pixelu. Vytvorené lúče sú následne známym postupom vyslané do scény a výsledná farba pixelu je daná aritmetickým priemerom farieb vrátených pre všetky lúče. Výsledok pre rôzny počet samplov je znázornený na obrázku 7.5. Pri vykresľovaní boli vzorky v rámci pixelu distribuované do rovnomernej mriežky. V závislosti od kompozície objektov scény je tiež vhodné okrem takéhoto typu distribúcie vzoriek vyskúšať aj alternatívne spôsoby ako napr. ich náhodnú distribúciu alebo náhodnú distribúciu s Poissonovým rozložením.

Aj keď takto implementovaný postup aliasing odstráni, výpočet farby pre viac lúčov na pixel neúmerne zvyšuje náročnosť aplikácie a rýchlosť výpočtu znateľne klesá. Supersampling sa preto v interaktívnych demách väčšinou nepoužíva. Ak by však bolo naším cieľom vykresliť metódou ray marching čo najkvalitnejší obraz, a nezáležalo by na rýchlosti výpočtu, toto riešenie by našlo opodstatnenie.

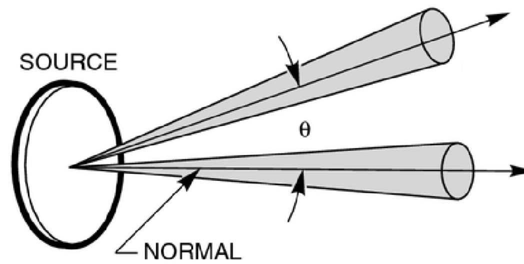
7.2 Osvetľovací model

Pri vykresľovaní scény je základom osvetľovacieho modelu kombinácia difúzneho a ambientného osvetlenia. Podľa Lambertovho modelu odrazu [14] sa farebná intenzita osvetlených objektov zvyšuje spolu s tým, ako klesá uhol zovretý vektorom normály ich povrchu a vektorom smerujúcim do zdroja svetla.

To by však znamenalo, že objekty ktoré nie sú lúčmi svetla priamo zasiahnuté, budú čierne. V prírode sa lúče svetla po dopade na prvý objekt v prostredí ďalej odrážajú a takýto stav preto nenastane. Toto odrazené svetlo v osvetľovacom modeli aplikácie simuluje pridanie ambientnej zložky osvetlenia, ktorá má pre všetky objekty určitú konštantnú hodnotu. Výsledná rovnica osvetlenia má teda nasledujúci tvar:

$$I = L_a C_a + L_d C_d \max\{0, \mathbf{n} \cdot \mathbf{l}\} \tag{7.1}$$

kde L_a, L_d sú intenzity ambientného a difúzneho zdroja svetla, C_a, C_d sú ambientné a difúzne farebné zložky osvetľovaného objektu, vektor $\mathbf{n}$ je normála povrchu objektu a $\mathbf{l}$ je vektor smerujúci do zdroja svetla.



Obrázok 7.6: Intenzita osvetlenia objektu je daná uhlom zovretým normálou povrchu a lúčom smerujúcim do zdroja svetla (prevzaté z [5])

7.2.1 Výpočet normály povrchu

Keďže povrch terénu je zadaný pomocou výškovej mapy, vieme pre každý bod terénu určiť gradient, ktorý udáva zmenu funkcie v smeroch x a z. Ten bude normálu dobre aproximovať. Gradient je definovaný ako

$$\nabla f = \frac{\partial f}{\partial x}\hat{\mathbf{x}} + \frac{\partial f}{\partial y}\hat{\mathbf{y}} + \frac{\partial f}{\partial z}\hat{\mathbf{z}} \tag{7.2}$$

Aj keď hodnoty výškovej mapy sa v smere y nemenia, funkcia map definovaná v kóde 7.5 vracia vzdialenosť k povrchu. Hodnota funkcie teda rastie so zvyšujúcou sa súradnicou y. Takto definovaný výpočet normály by fungoval aj pri zobrazovaní objektov definovaných pomocou signed distance functions miesto výškovej mapy.

```
vec3 calculateNormal(vec3 point)
{
    float h = 0.002;
    vec3 normal;
    normal.x = map(point + vec3(h, 0, 0)) - map(point - vec3(h, 0, 0));
    normal.y = map(point + vec3(0, h, 0)) - map(point - vec3(0, h, 0));
    normal.z = map(point + vec3(0, 0, h)) - map(point - vec3(0, 0, h));
    return normalize(normal);
}
```

Kód 7.8: Výpočet normály povrchu v danom bode pomocou mapovacej funkcie

7.3 Popis procedurálneho terénu

Ako bolo zmienené v časti 3.2, pre modelovanie prírodného terénu s variáciou na rôznych úrovniach je vhodné použiť funkciu fractional Brownian motion (ďalej len fBm). Táto funkcia je zložená zo súčtu určitého počtu oktáv šumu. Väčšinou má každá vytvorená oktáva oproti predchádzajúcej dvojnásobnú frekvenciu a polovičnú amplitúdu. Prvá oktáva s nízkou frekvenciou a vysokou amplitúdou zodpovedá v modelovanom teréne kopcom a údoliam. Ďalšie oktávy terén modifikujú nerovnomernosťami v stále nižšej mierke. Napríklad ôsma oktáva môže v prostredí simulovať rozmanitosť na veľkostnej úrovni kameňov.

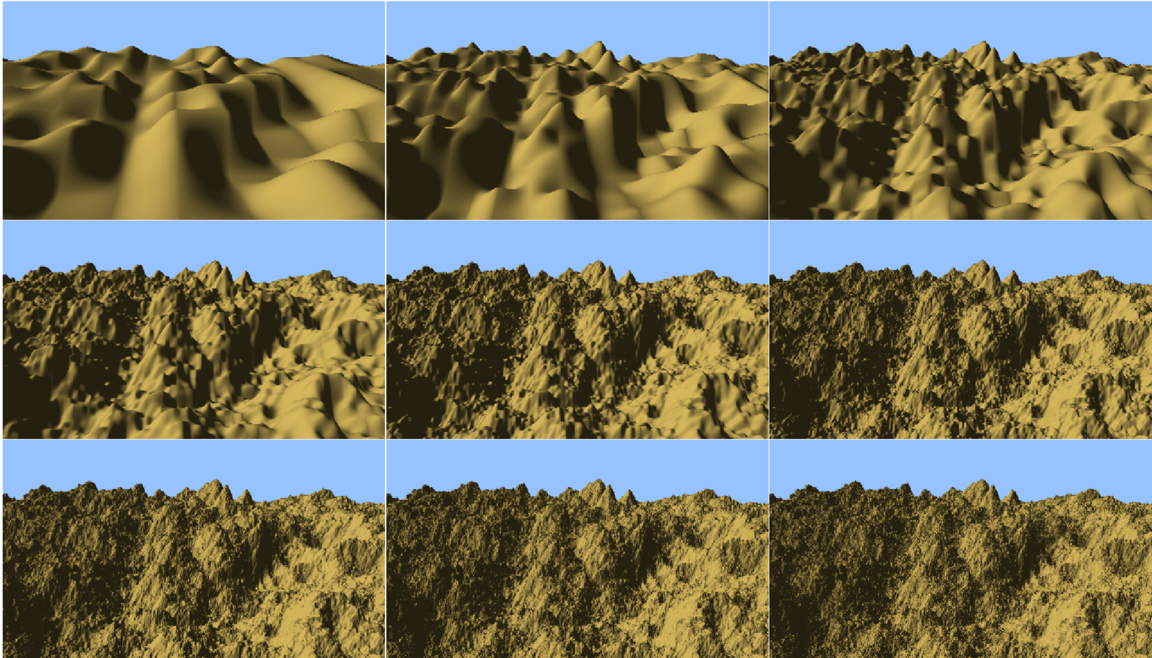
```
float fbm(vec2 position, int octaves)
{
    float value = 0.0;
    float amplitude = 1.0;
    float frequency = 0.5;

    for (int i = 0; i < octaves; i++)
    {
        value += amplitude * noise(frequency * position);
        amplitude *= 0.5;
        frequency *= 2.0;
    }
    return value;
}
```

Kód 7.9: Konštrukcia fractional Brownian motion. Pre vytvorenie zaujímavej výškovej mapy je vhodné vyskúšať rôzne koeficienty pre jednotlivé premenné

Počet oktáv šumu z ktorých sa fBm vytvorí nie je pevne daný, ale závisí od požadovanej úrovne detailu. Pri teste na prienik lúča s povrchom je vhodné použiť nízky počet oktáv a teda nízku úroveň detailu. Keďže ray marching je najnáročnejšia časť aplikácie, výpočet sa podstatne urýchli. Detail vo výslednom obraze sa potom dosiahne tým, že pri vyhodnocovaní normály povrchu bude použitý fBm s vysokým počtom oktáv[1]. V určitom bode prestane mať pripočítavanie ďalších oktáv zmysel, keďže výpočet je okrem iného obmedzený presnosťou zobrazenia čísel v počítači a kvalita obrazu je obmedzená rozlíšením okna.

Pri voľbe počtu šumových oktáv fBm, ktoré sú použité pre výpočet normál, je vhodné zohľadniť aktuálnu vzdialenosť kamery od povrchu terénu. So zvyšujúcou sa vzdialenosťou kamery od povrchu by sa mal znižovať počet oktáv fBm, ktorý je použitý pre výpočet normál. Takýmto spôsobom je možné zvýšiť rýchlosť výpočtu, ale hlavne minimalizovať výskyt aliasingu vznikajúceho pri zobrazovaní terénu vo väčšej vzdialenosti.

[1] Princíp je podobný technike *normal mapping*, ktorá sa používa pri rasterizácii. Pri tejto technike sa vykresľujú modely s nízkym počtom polygónov, ale pri osvetľovaní sa využíva textúra obsahujúca normály, ktoré boli vygenerované z modelu s vysokým počtom polygónov.



Obrázok 7.7: Vykreslenie funkcie fractional Brownian motion zloženej z rôzneho počtu oktáv šumu. V hornom riadku zľava 1 až 3 oktávy, v strednom riadku 4 až 6 oktáv a v spodnom riadku 7 až 8 oktáv. Kamera sa nachádza v pomerne veľkej vzdialenosti od povrchu a pri vyššom počte oktáv začína vznikať v obraze aliasing

7.3.1 Hmla

Keďže maximálna vzdialenosť v ktorej je ešte prostredie vykresľované je obmedzená, vzniká vo výslednom obraze neprirodzený ostrý prechod. Jeho efekt je možné zmierniť pridaním exponenciálnej hmly, ktorá interpoluje medzi farbou prostredia a farbou oblohy:

$$f = \min\{e^{\rho d} - 1, 1\} \tag{7.3}$$

kde f je váha hmly obmedzená do intervalu $[0, 1]$, ρ je hustota hmly a d je vzdialenosť bodu, pre ktorý je váha hmly počítaná, od kamery. So vzdialenosťou od kamery teda sila hmly rastie.

7.3.2 Obloha

Pre pixely, na ktorých by sa malo nachádzať slnko, bude hodnota skalárneho súčinu medzi pozíciou slnka v scéne, a lúčmi vyslanými do pixelov blízka číslu jedna.

```
vec3 sky(vec3 rd)
{
    float sunDot = max(dot(rd, sunPosition), 0.0);
    float power = 80.0;
    return skyColor + pow(sunDot, power) * sunColor;
}
```

Kód 7.10: Funkcia vracia farbu oblohy pre daný pixel. Premenná *power* určuje aký veľký bude polomer slnka. Pri nižších hodnotách v scéne vznikne tzv. *glare effect*

7.3.3 Oblaky

Fractional Brownian motion je tiež možné využiť na vytvorenie jednoduchých dvojrozmerných oblakov. V určitej výške cloudHeight budeme farbu oblohy kombinovať s farbou oblakov pomocou váhy danej funkciou fBm. Pre každý vyslaný lúč nájdeme analyticky bod prieniku s rovinou $y = cloudHeight$ a využijeme ho ako vstup pre fBm. Výslednú hodnotu vynásobíme y-vou zložkou lúča, čo zabezpečí, že oblaky budú v horizonte (kde majú vyslané lúče túto zložku nulovú) miznúť.

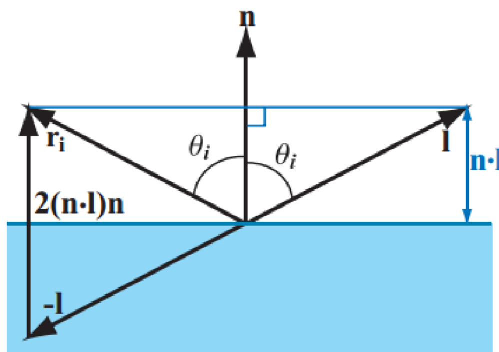
```
vec3 clouds(vec3 color, vec3 ro, vec3 rd)
{
    vec3 skyPoint = ro + rd * ((cloudHeight - ro.y)/rd.y);
    skyPoint += windSpeed*time;
    float cloudfBm = fBm(skyPoint.xz, 4);
    cloudWeight = clamp(cloudfBm*rd.y, 0.0, 1.0);
    return mix(color, cloudColor, cloudWeight);
}
```

Kód 7.11: Funkcia interpolujúca aktuálnu farbu oblohy s oblakmi. Pripočítanie času ku skyPoint spôsobí pohyb oblakov a simuluje v scéne vietor

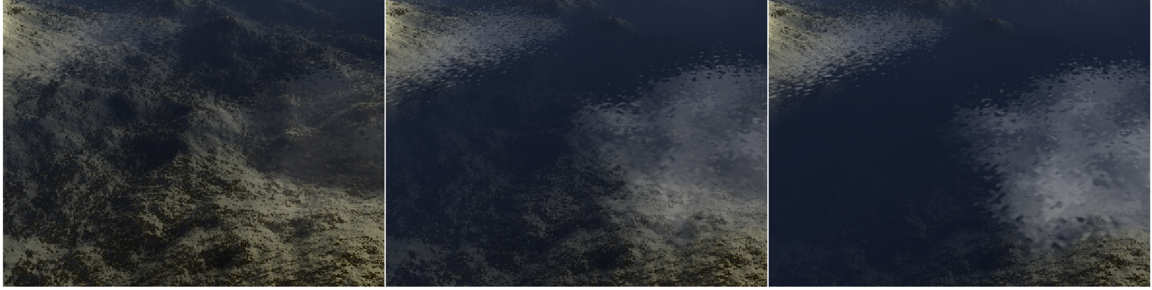
7.3.4 Voda

Pre vytvorenie realisticky vyzerajúcej vody, musí toto médium odrážať a zároveň prepúšťať časť slnečných lúčov. Vodná plocha by nemala byť dokonale rovná ale zvlnená. Najprv je vyhodnotená farba podliehajúceho terénu, pretože bude potrebná pri následnom simulovaní refrakcie. Ak je potom y-ová hodnota bodu, ktorý bol nájdený pri prieniku lúča s terénom nižšia, ako stanovená línia vody, vykoná sa tvorba vodnej hladiny. Podobne ako pri tvorbe oblakov je analyticky nájdený bod pozdĺž vyslaného lúča, v ktorom by došlo k pretnutiu hypotetickej hladiny vody. Pre tento bod je potom vytvorená pomocou šumu a trigonometrických funkcií upravená normála, ktorá simuluje na hladine vody vlnenie.

Na hladine vody je potrebné skombinovať farbu, dodanú do kamery svetlom odrazeným z prostredia, a farbu terénu pod vodou, ktorý môže byť pod určitým uhlom vidno. Z dôvodu urýchlenia výpočtu sa na vode odráža iba obloha a nezohľadňuje sa odraz okolitého terénu. Získaný vektor sa použije ako argument pre funkcie, ktoré vracajú farbu oblohy a oblakov.



Obrázok 7.8: Nájdenie vektora, ktorý prináša do kamery svetlo odrazené z prostredia. Vektor **l** smeruje z miesta dotyku lúča s hladinou do kamery. Vektor **n** je normála vodnej hladiny. Vektor $\mathbf{r}_i$ je hľadaný vektor (prevzaté z [6])

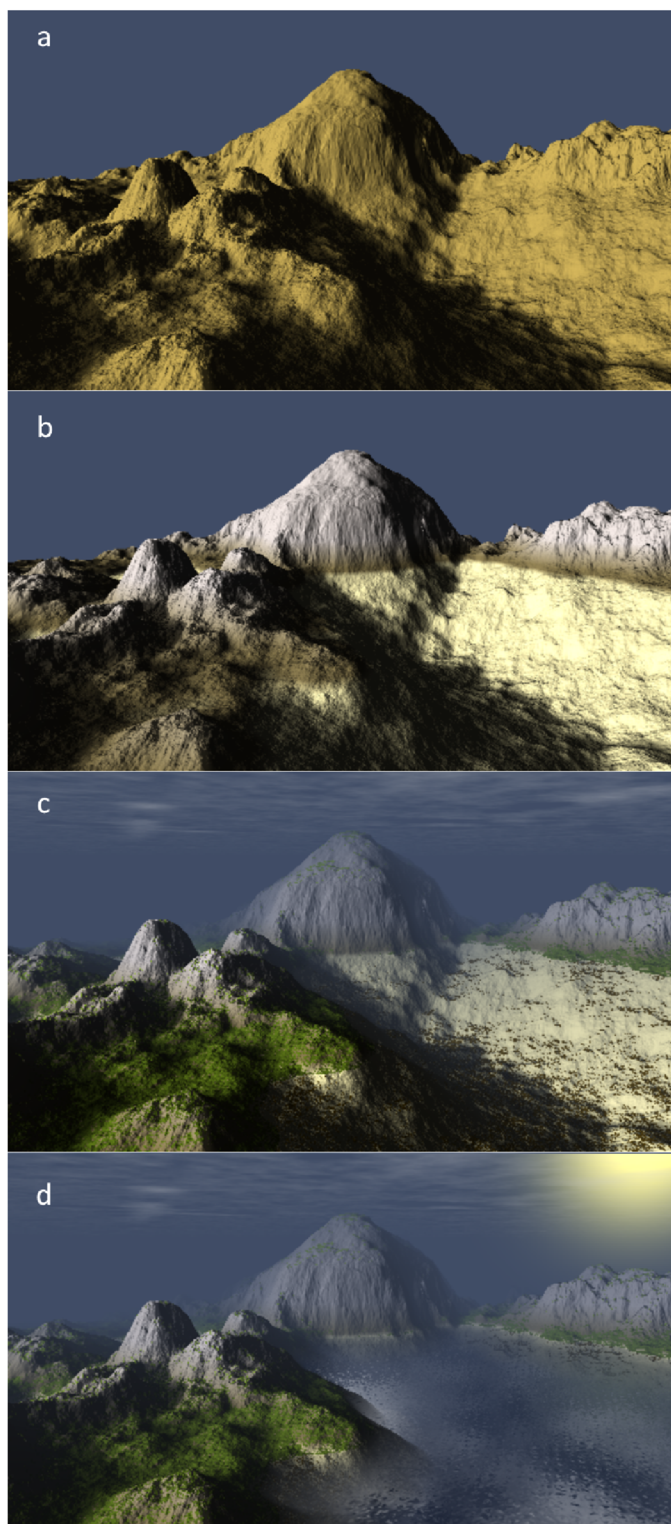


Obrázok 7.9: Porovnanie odrazivosti a refrakcie vodnej hladiny pri konštantom uhle kamery a rôznych hodnotách parametra κ rovnice 7.4. Zľava $\kappa = 1$, $\kappa = 3$ a $\kappa = 6$

Z Fresnelových rovníc [6] vyplýva, že odrazivosť materiálu má inverzný vzťah k veľkosti uhla, ktorý je zovretý povrchom (v tomto prípade hladinou vody) a okom pozorovateľa. Ak sa teda kamera díva smerom kolmo dolu, mal by byť viditeľný povrch pod vodnou hladinou. Ak je naopak kamera takmer rovnobežná s hladinou, bude vo vode zreteľné len svetlo odrazené z okolitého prostredia. Fresnelov efekt je pre potreby práce definovaný nasledujúcim vzťahom:

$$f = |\mathbf{v} \cdot \mathbf{n}|^{\kappa} \tag{7.4}$$

kde f je koeficient refrakcie, ktorý je nepriamo úmerný odrazivosti materiálu, $\mathbf{v}$ je smer lúča vyslaného z kamery, $\mathbf{n}$ je normála vodnej hladiny a κ popisuje mieru refrakcie v prípade, že sa kamera nedíva priamo nadol. Ak je hodnota parametra κ nízka, bude k refrakcii dochádzať aj pri uhle θ_i menšom ako 90 stupňov (pozri 7.8). Pri určovaní farby bodu na vodnej hladine je vypočítaný koeficient refrakcie f použitý ako váha pri interpolácii medzi farbou získanou z odrazu prostredia a farbou povrchu pod vodou.



Obrázok 7.10: Vykreslenie terénu: (a) terén definovaný len pomocou výškovej mapy, (b) rozlíšenie materiálu prostredia pomocou farebného gradientu, (c) pridanie oblakov, hmly a trávy na miesta, kde normála ukazuje najmä smerom hore, (d) finálne prostredie s vodnou plochou

Kapitola 8

Zhodnotenie rýchlosti zobrazovania

Cieľom práce bolo vytvoriť aplikáciu schopnú plynulého behu v reálnom čase. Pre overenie dosiahnutých výsledkov bolo spustených niekoľko testovacích behov programu, pre ktoré bola určená priemerná doba výpočtu jednej snímky ako priemer dôb výpočtu posledných sto zobrazených snímok.

Rýchlosť výpočtu nutne závisí od zobrazovanej scény, preto bola kamera pri každom testovacom behu natočená na inú časť vykresľovaného prostredia. Algoritmus ray marching vykoná najviac krokov, a je teda najpomalší, pri pohľade do vzdialenejšej časti scény (obzvlášť ak je kamera umiestnená v malej výške). Ak však kamera smeruje na oblohu, skončí výpočet farby pixelu veľmi rýchlo, keďže vyslané lúče po nízkom počte vykonaných krokov prekročia maximálnu povolenú dĺžku. Namerané hodnoty by mali odpovedať najčastejším hodnotám, ktoré je možné odsledovať pri behu aplikácie. Pri rýchlostnom testovaní bol preto pohľad kamery zameraný na tie časti scény, ktoré neviedli k vzniku žiadneho z týchto extrémov.

Testy boli vykonané na dvoch zostavách s nasledujúcou špecifikáciou:

Zostava č. 1:

- **Typ notebooku:** Lenovo IdeaPad Y50-70
- **Procesor:** Intel Core i5 4210H Haswell 2.90GHz
- **Pamäť:** 8GB Dual-Channel DDR3 798MHz
- **Grafická karta:** NVIDIA GeForce GTX 860M 2GB (verzia ovládačov 365.19)

Zostava č. 2:

- **Typ notebooku:** Dell XPS 15 9550
- **Procesor:** Intel Core i7 6700HQ 2.60GHz
- **Pamäť:** 16GB DDR4-2133 1066 MHz
- **Grafická karta:** NVIDIA GeForce GTX 960M 2GB (verzia ovládačov 365.19)

Hodnoty získané z jednotlivých testovacích behov boli následne spriemerované. Tento postup sa opakoval pre všetky testované rozlíšenia a pre obe testovacie zostavy. Veľmi podobné výsledky na porovnateľne kvalitných testovacích zostavách svedčia o stabilnom

správnaní aplikácie. Vyplýva z nich, že na oboch testovacích zostavách môže aplikácia pri nižších a stredných rozlíšeniach bežať plynule. Pri rozlíšení 1920x1080 je však rýchlosť zobrazovania v oboch prípadoch na hranici prijateľnosti.

Rozlíšenie okna	640x480	1366x768	1920x1080
Zostava 1	6.6 ms (152.1 FPS)	20.2 ms (49.5 FPS)	40.2 ms (24.9 FPS)
Zostava 2	5.1 ms (196.1 FPS)	17.5 ms (57.1 FPS)	35.4 ms (28.2 FPS)

Tabuľka 8.1: Priemerná doba výpočtu jednej snímky vo výslednej aplikácii.

Kapitola 9

Záver

Cieľom tejto práce bolo vytvoriť grafické demo využívajúce procedurálne generovaný obsah, ktorého spustiteľný súbor neprekročí 64kB a bude bežať na grafickej karte v reálnom čase. V implementácii sa tieto ciele podarilo úspešne splniť. Veľkosť výsledného súboru je 29kB a rýchlosť výpočtu bola pri použitých testovacích zostavách dostatočná pre plynulý chod programu.

Práca je rozdelená do niekoľkých logických celkov. V prvej časti bola stručne predstavená demoscéna a grafické demo. Ďalej boli naštudované metódy generovania procedurálnych textúr, ktoré boli v práci využité pre tvorbu prírodného terénu a pre úpravu jeho vzhľadu. Terén bol potom vykreslený metódou ray marching, popísanou teoreticky v časti venujúcej sa zobrazovaniu. V kapitole venujúcej sa implementácii bol vysvetlený spôsob, akým je možné vytvárať scénu vo fragment shaderi pri minimálnej spolupráci s procesorom. V tejto kapitole boli tiež porovnané verzie ray marchingu s pevným krokom a s dynamickou veľkosťou kroku. Z výsledkov testovania vyplýva, že v závislosti od vykresľovanej scény, môže byť verzia algoritmu s dynamickou veľkosťou kroku rádovo rýchlejšia ako verzia s pevným krokom. Bola tiež vyskúšaná možnosť využívať pri ray marchingu supersampling s cieľom znížiť v obraze aliasing. Namiesto jedného lúča bolo do každého pixelu vyslaných viac lúčov a pixelu bola priradená farba, získaná ako priemer farieb vypočítaných pre jednotlivé lúče. Táto technika síce priniesla požadovaný efekt, ale zároveň viedla k neprijateľnému poklesu rýchlosti, takže v práci ďalej nebola použitá.

V poslednej časti kapitoly venovanej implementácii bolo prezentované využitie funkcie fractional Brownian motion pre generovanie prakticky neobmedzene veľkého prírodného terénu. Terén bol procedurálne zafarbený a do prostredia bola pridaná hmla a oblaky. Zrejme najzaujímavejšou časťou vytvoreného prostredia je vodná plocha, na ktorej sa jednak odráža obloha, ale tiež cez ňu čiastočne presvitajú skaly pod vodnou hladinou. Posledná kapitola sa venuje testovaniu rýchlosti aplikácie.

Z hľadiska zobrazovacej metódy by bolo možné pokračovať v započatej optimalizácii algoritmu ray marching. Z hľadiska rýchlosti a presnosti je dôležitá veľkosť kroku (krok je daný aktuálnou vzdialenosťou lúča k povrchu), o ktorý sa počas jednej iterácie algoritmus posunie. Najkritickejšou časťou krokovania je jeho záver, kedy sa lúč môže nachádzať veľmi blízko povrchu terénu a preto sú vykonávané kroky malé. Najzložitejší prípad nastáva, ak je lúč blízko povrchu s ktorým je rovnobežný, a teda sa s ním nemôže pretnúť. Vtedy sa bude lúč posúvať po zbytočne malých krokoch. Riešením by mohlo byť krokovanie známym spôsobom až po dosiahnutie určitej medznej veľkosti kroku. V tomto bode by krokovanie prestalo a algoritmus by sa pokúsil nájsť prienik s povrchom pomocou binárneho vyhľadávania. Vykonával by sa len pevne stanovený počet krokov, ktorý by bol nastavený na

minimálnu hodnotu postačujúcu pre požadovanú kvalitu obrazu. V najdôležitejšej časti by mal teda algoritmus konštantnú zložitosť.

Iným prístupom by bolo opustenie ray marchingu a využitie použitej výškovej mapy na generovanie polygonálneho modelu, ktorý by bol vykresľovaný rasterizáciou. To by pravdepodobne viedlo k zvýšeniu kvality obrazu aj rýchlosti vykresľovania. Niektoré prezentované techniky by však potom nebolo možné použiť, keďže sú závislé od lúčov, ktoré sú pri ray marchingu do scény vysielané.

Terén by bolo možné vylepšiť vytvorením väčších výškových rozdielov medzi rôznymi miestami vo svete. Mohol by napríklad obsahovať zasnežené hory bez vegetácie. Ďalším krokom by mohlo byť nahradenie dvojrozmerných oblakov ich volumetrickou verziou. Všetky tieto úpravy by samozrejme museli zohľadniť zachovanie rýchlosti výpočtu.

Literatúra

[1] *Ray Tracing Diagram*. Wikimedia Commons, 2008 [cit. 2016-04-20], [Online].
URL https://upload.wikimedia.org/wikipedia/commons/8/83/Ray_trace_diagram.svg

[2] *Julia set* $e^x - 0.65$. Wikimedia Commons, 2011 [cit. 2016-04-20], [Online].
URL https://upload.wikimedia.org/wikipedia/commons/2/2c/JULIA_Exp%28z%29%2Bc_CX%3D_0.65.jmb.jpg

[3] *Julia set* $x^2 + 0.285 + 0.01i$. Wikimedia Commons, 2011 [cit. 2016-04-20], [Online].
URL https://upload.wikimedia.org/wikipedia/en/d/d4/Julia_0.285_0.01.png

[4] *Julia set* $x^5 + 0.544$. Wikimedia Commons, 2011 [cit. 2016-04-20], [Online].
URL https://upload.wikimedia.org/wikipedia/commons/9/93/JULIA5.jmb.jpg

[5] *Laws of Radiation*. [cit. 2016-04-20], [Online].
URL http://www.newport.com/Laws-of-Radiation/381843/1033/content.aspx

[6] Akenine-Möller, T.; Haines, E.; Hoffman, N.: *Real-Time Rendering*. A K Peters, třetí vydání, 2008.

[7] Alfeld, P.: *The Mandelbrot Set*. The University of Utah, 1998 [cit. 2016-04-20], [Online].
URL http://www.math.utah.edu/~alfeld/math/mandelbrot/mandelbrot.html

[8] Ebert, D. S.; Musgrave, F. K.; Peachey, D.; aj.: *Texturing & Modeling*. Morgan Kaufmann Publishers, třetí vydání, 2003.

[9] Elias, H.: *Perlin Noise*. [cit. 2016-04-20], [Online].
URL http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

[10] Joyce, D. E.: *Julia and Mandelbrot Sets*. 2003 [cit. 2016-04-20], [Online].
URL http://aleph0.clarku.edu/~djoyce/julia/julia.html

[11] Jönsson, A.: *Generating Perlin Noise*. 2002 [cit. 2016-04-20], [Online].
URL http://www.angelcode.com/dev/perlin/perlin.html

[12] Keinert, B.; Schäfer, H.; Korndörfer, J.; aj.: *Enhanced Sphere Tracing*. STAG: Smart Tools & Apps for Graphics, 2014 [cit. 2016-04-20], [Online].
URL lgdv.cs.fau.de/get/2234

[13] Kelly, G.; McCabe, H.: *A Survey of Procedural Techniques for City Generation. ITB Journal*, ročník 14, 2006 [cit. 2016-04-20]: s. 87–130, [Online].
URL http://www.citygen.net/files/Procedural_City_Generation_Survey.pdf

[14] Lengyel, E.: *Mathematics for 3D game programming and computer graphics*. Course Technology PTR, třetí vydání, 2012.

[15] McCombs, S.: *Procedural Textures*. [cit. 2016-04-20], [Online].
URL http://www.upvector.com/?section=Tutorials&subsection=Intro%20to%20Procedural%20Textures

[16] Parkin, S.: *No Man's Sky: the game where you can explore 18 quintillion planets*. The Guardian, 2015-07-12 [cit. 2016-04-20], [Online].
URL https://www.theguardian.com/technology/2015/jul/12/no-mans-sky-18-quintillion-planets-hello-games

[17] Perlin, K.: *Improving Noise. ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2002*, ročník 21, 2002 [cit. 2016-04-20]: s. 681–682, [Online].
URL http://mrl.nyu.edu/~perlin/paper445.pdf

[18] Íñigo Quílez: *Drawing, Sculpting and Animating Nature with Maths*. ICM 2006 Madrid, 2006 [cit. 2016-04-20], [Online].
URL http://www.iquilezles.org/www/material/icm2006/inigo_slides.pdf

[19] Íñigo Quílez: *Domain Warping*. [cit. 2016-04-20], [Online].
URL http://www.iquilezles.org/www/articles/warp/warp.htm

[20] Íñigo Quílez: *Free penumbra shadows for raymarching distance fields*. [cit. 2016-04-20], [Online].
URL http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

[21] Íñigo Quílez: *Modelling with Distance Functions*. [cit. 2016-04-20], [Online].
URL http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

[22] Íñigo Quílez: *Terrain Raymarching*. [cit. 2016-04-20], [Online].
URL http://www.iquilezles.org/www/articles/terrainmarching/terrainmarching.htm

[23] Reunanen, M.: *Computer Demos—What Makes Them Tick?* Dizertační práce, Aalto University, 2010, [Online].
URL https://www.kameli.net/demoresearch2/reunanen-licthesis.pdf

[24] Shirley, P.; Marschner, S.: *Fundamentals of Computer Graphics*. CRC Press, třetí vydání, 2009.

[25] Weisstein, E.: *Mandelbrot Set*. MathWorld–A Wolfram Web Resource, 2016 [cit. 2016-04-20], [Online].
URL http://mathworld.wolfram.com/MandelbrotSet.html

Prílohy

Zoznam príloh

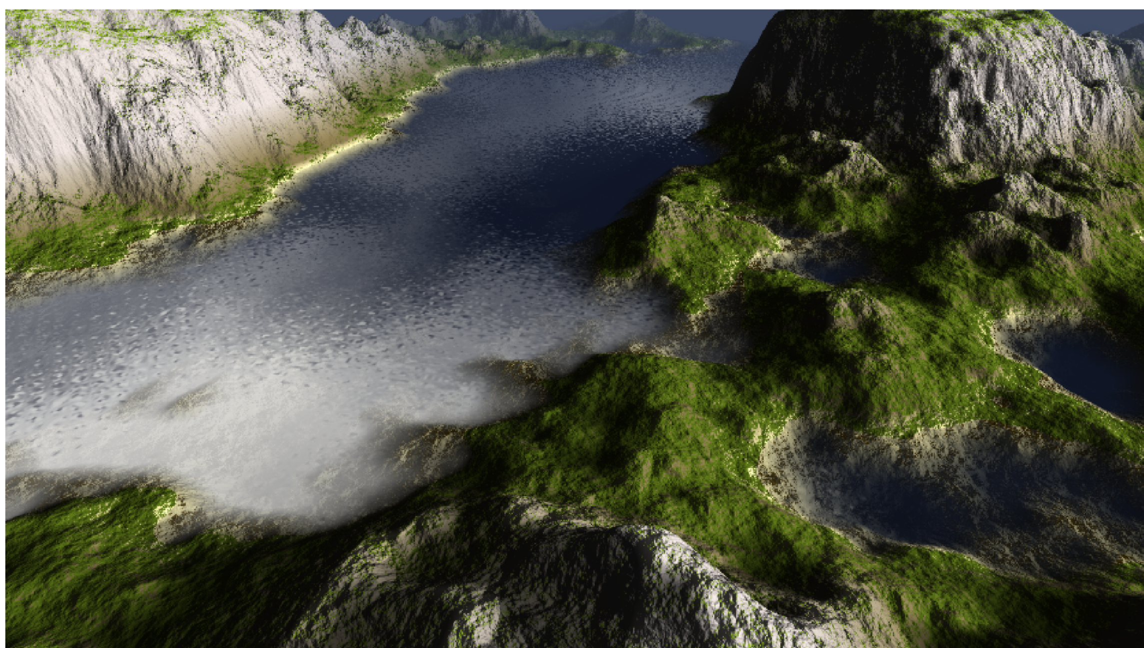
A Ukážka výsledných scén 43

B Použitie aplikácie 45

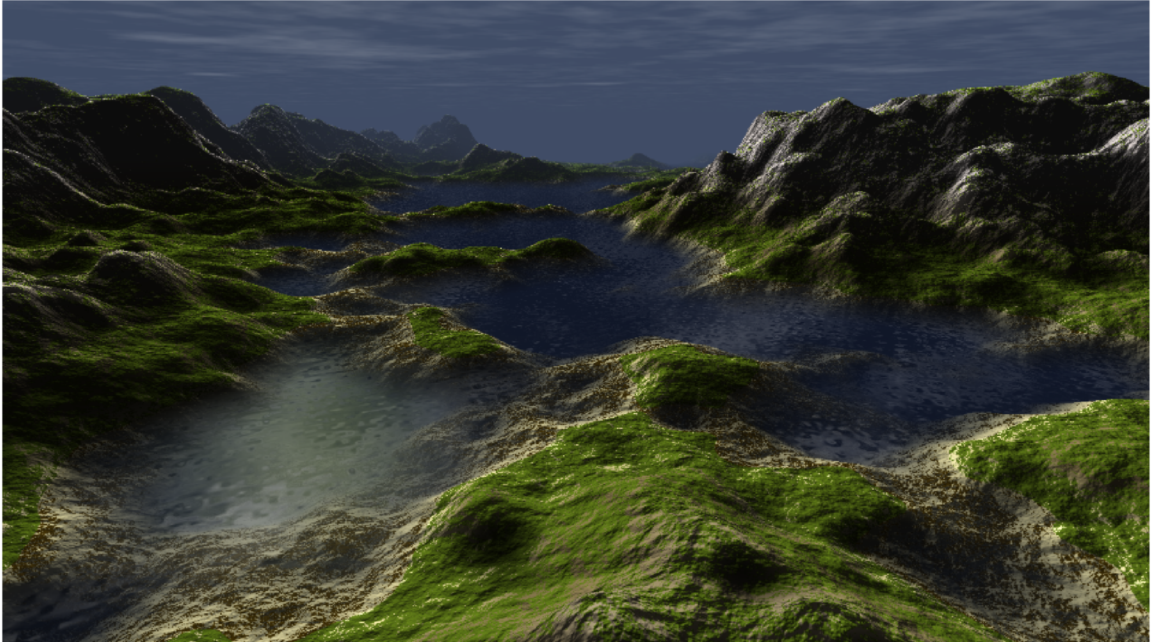
C Obsah CD 46

Príloha A

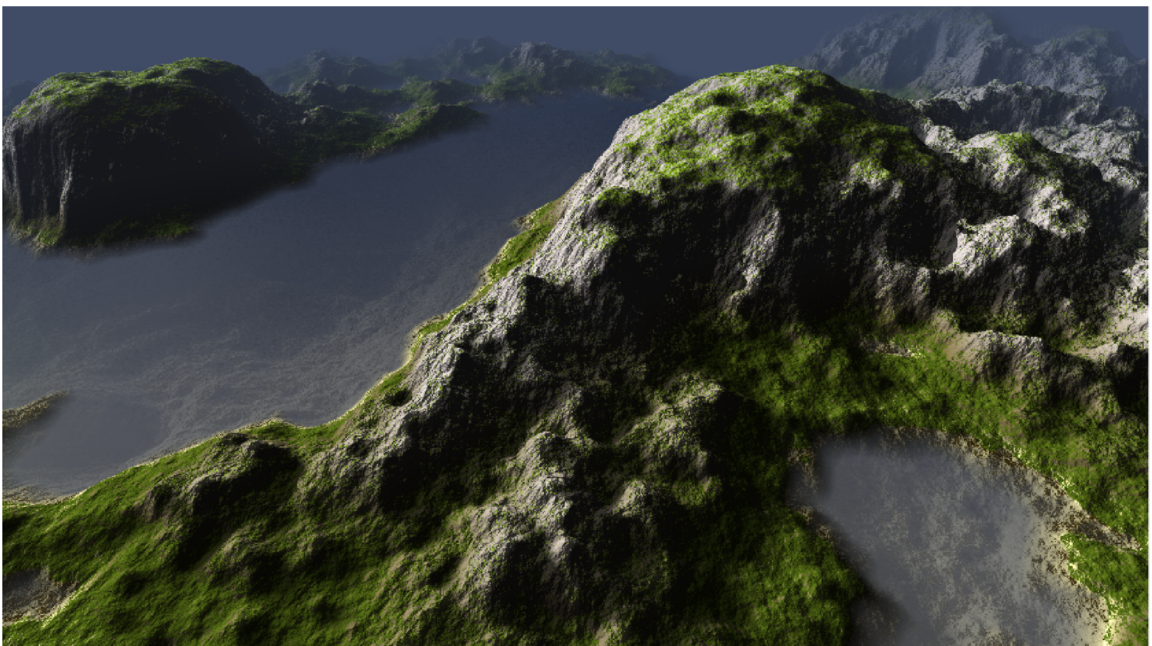
Ukážka výsledných scén



Obrázok A.1



Obrázok A.2



Obrázok A.3

Príloha B

Použitie aplikácie

Aplikácia je z dôvodu minimalizovania veľkosti spustiteľného súboru schopná bežať iba na platforme Windows. Použitým vývojovým prostredím je Visual Studio 2015. Po vykonaní prekladu je v priečinku **Solution/Release** resp. **Solution/Debug** vytvorený spustiteľný súbor Demo64.exe. V zdrojovom súbore Def.h je možné manuálne nastaviť rozlíšenie okna aplikácie. Okrem toho sú v zložke **Executables** umiestnené predpripravené spustiteľné súbory s rôznymi rozlíšeniami okna.

Po spustení sa dá v scéne pohybovať pomocou myši a klávesy **W**. Aplikácia sa ukončuje stlačením klávesy **Esc**.

Príloha C

Obsah CD

Nástroj Visual Studio môže po spustení projektu vygenerovať ďalšie pomocné priečinky. Zložka so zdrojovými súbormi shaderov je na CD umiestnená len pre úplnosť. Aplikácia využíva pre ich vytvorenie kód definovaný v reťazci priamo v zdrojových súboroch C++.

Video Obsahuje krátke video demonštrujúce výsledok práce.

Executables Obsahuje spustiteľné súbory v rôznom rozlíšení.

Solution Zložka pre solution.

- **Demo64** Zložka pre projekt.
  - **src** Zdrojové súbory v C++.
  - **shaders** Zdrojové súbory shaderov v GLSL.
- **Release** Zložka v ktorej sa vytvorí release verzia projektu.
- **Debug** Zložka v ktorej sa vytvorí debug verzia projektu.

Text Text práce.

Text Source Zdrojový kód tejto práce vytvorený pomocou programu LaTeX.