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

NEKONEČNÁ JESKYNĚ

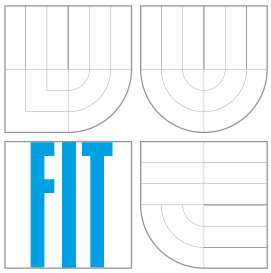
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR POSPÍŠIL

BRNO 2014

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

NEKONEČNÁ JESKYNĚ

ENDLESS CAVE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR POSPÍŠIL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ MILET

BRNO 2014

Abstrakt

Cílem této práce bylo implementovat aplikaci, které bude zobrazovat nekonečnou jeskyni. Základ této jeskyně tvoří šumová funkce Simplex noise. Šum získaný touto funkcí je následně prahován a na získanou mřížku bodů je poté aplikován algoritmus Marching tetrahedrons. Ten vstupní volumetrická data převádí do hraniční reprezentace. V aplikaci byl také použit Phongův osvětlovací model a dále Bump mapping pro zvýšení vizuální kvality. Aplikace je založena na knihovně OpenGL. V první polovině technické zprávy byly uvedené metody teoreticky popsány, druhá polovina pak obsahuje popis samotné implementace.

Abstract

The goal of this thesis is to implement an application showing an endless cave. The basis of this cave is simplex noise method. On the noise produced by this function is afterwards applied thresholding. Produced grid of points is used like input for marching tetrahedrons algorithm. This algorithm converts volumetric data to boundary representation. Phong reflection model and Bump mapping were used in the application, too, in order to improve the visual quality. The application is based on OpenGL library. The first part of the technical report contains theoretical description of mentioned methods, the second part contains description of implementation.

Klíčová slova

Nekonečná jeskyně, OpenGL, 3D grafika, procedurální modelování, volumetrická data, Perlinův šum, Marching tetrahedra.

Keywords

Endless cave, OpenGL, 3D graphic, procedural modeling, volumetric data, Perlin noise, Marching tetrahedra.

Citace

Petr Pospíšil: Nekonečná jeskyně, bakalářská práce, Brno, FIT VUT v Brně, 2014

Nekonečná jeskyně

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta

.....
Petr Pospíšil
20. května 2014

Poděkování

Tímto děkuji svému vedoucímu panu Ing. Tomáši Miletovi za motivaci k vytvoření této práce, vedení během práce na ní, cenné rady, které mi ušetřily mnoho času při programování, a přínosnou kritiku této technické zprávy.

Obsah

Kapitola 1

Úvod

Počítačová grafika je alternativa k textovému výstupu počítače. Lze tedy říci, že vše, co se nám zobrazí na monitoru, a nejde přitom o prostý text, je počítačová grafika. Patří sem ikony, symboly, obrázky, grafické uživatelské prostředí, 3D grafika a další grafické prvky. Tato práce spadá do kategorie 3D grafika. Jak již název napovídá, jedná se o druh grafiky uchovávající informace o zobrazovaném předmětu ve třech dimenzích, tak jak to známe v reálném světě. Nejedná se tedy o klasický obrázek, který je pouze projekcí 3D objektu na 2D plochu, ale spíše o model objektu rovnou ve třech dimenzích.

3D grafická data je možné získat mnoha způsoby. Od pokročilých postupů vytváření modelů na základě snímání reality, zahrnujících mnohočetné kamery a různé laserové měřiče vzdálenosti, přes „klasickou“ metodu modelování v k tomu určenému programu, až po poloautomatizované generování v reálném čase, založené na nějakém algoritmu. Právě touto technologií se zabývá tato práce a lze ji použít od samočinného generování stromů a listí, kde žádné dva listy nebo stromy nebudou stejné, přes např. mraky, nebo lidskou kůži až po nekonečné jeskyně. Tvorba jedné takové jeskyně byla předmětem této práce.

Kořeny procedurálního generování grafiky sahají zhruba do roku 1984. V tomto roce vyšla herní série *Elite*, která využívala procedurální generování herního světa. Vývojáře k použití této technologie vedly dva protichůdné požadavky. Prvním z nich byl požadavek na nízkou paměťovou náročnost, což bylo v té době nezbytné, protože počítače byly často osazeny operační pamětí s kapacitou pouze 32, 40, 48 nebo 64 kB. Druhým požadavkem byl otevřený herní svět s velkými prostory k prozkoumávání. Použití procedurálně generované grafiky tedy bylo velmi žádoucí.

Tato práce si klade za cíl prezentovat současné možnosti procedurálně generované grafiky. K tomu využívá příkladu jeskyně, která se s pohybem pozorovatele generuje dále a dále a prakticky tedy nikde nekončí, a přitom se téměř nikdy neopakuje. Programová část hojně využívá knihovnu OpenGL a pracuje také s programy pro grafickou kartu.

Téma nekonečná jeskyně mne zaujalo proto, že se o 3D grafiku zajímám již mnoho let a chtěl jsem si vyzkoušet procedurální generování, se kterým jsem do té doby neměl žádné zkušenosti.

Cílem práce je implementace aplikace, která bude zobrazovat nekonečnou jeskyni. Tuto jeskyni je tedy třeba nejprve procedurálně generovat. K tomu samozřejmě patří popis vybraných metod pro generování objemových dat, jejich transformaci do hraničního modelu a dále pro jejich následné otexturování a zobrazení. Dále je součástí cíle zhodnocení dosažených výsledků a navržení možností pokračování projektu. Poslední částí zadání je vytvoření plakátu, jenž bude prezentovat výsledky projektu.

Dále bude stručně popsána struktura práce.

Hned za úvodem následuje kapitola 2, věnující se teorii řešených problémů. Kapitola začíná stručnými základními informacemi o OpenGL (2.1). Následující kapitola se zabývá reprezentací 3D objektů v počítači (2.2). Dále je vysvětlen pojem procedurální modelování (2.3) a také jsou zde popsány různé druhy šumových funkcí (2.4). Následuje vysvětlení algoritmů pro práci s geometrií objektů (2.5). Za touto kapitolou se nachází kapitola o texturách a jejich mapování (2.6). Předposlední kapitola popisuje Phongův osvětlovací model (2.7) a stručně ho srovnává s některými alternativními modely. V pořadí třetí velkou kapitolou je Implementace (3), která se zabývá popisem vlastní práce. Tato kapitola se dělí na popis základní koncepce (3.1), ve které je nastíněno, o co v práci jde, popis fungování (3.2), kde je podrobně popsáno, jak aplikace pracuje, příklady použití (3.3) a kapitolu uzavírají informace o použitém vybavení (3.4). Poslední kapitolou je Závěr (4), ve kterém se nachází zhodnocení dosažených výsledků a také nástin možných pokračování v práci.

Kapitola 2

Generování grafického obsahu

Tato kapitola se věnuje teorii související s procedurálním modelováním (2.3). Ještě před tím se nachází úvod do OpenGL (2.1) a dále následuje kapitola o metodách reprezentace 3D objektů, které byly použity při tvorbě aplikace (2.2). Na závěr jsou pak popsány algoritmy pro generování šumu (2.4) a pro převod 3D modelu z volumetrické reprezentace do hraniční reprezentace (2.5), a také je zde popsán Phongův osvětlovací model (2.7), který je v aplikaci rovněž použit.

2.1 OpenGL

Pokud by někdo chtěl pracovat s 3D počítačovou grafikou bez OpenGL (nebo nějakého podobného vysokoúrovňového API), bylo by to samozřejmě možné. Bylo by však potřeba zapisovat přímo do *framebufferu*. Framebuffer je místo v paměti obsahující 2D mřížku, která představuje to, co se vykreslí na obrazovku. Bylo by tedy nutné „ručně“ zajistit zpracování všech potřebných grafických operací jako např. reprezentace obrazu, transformace barev, vykreslování různých typů křivek, vyplňování ploch, reprezentace těles, projekce, osvětlovací modely, řešení viditelnosti, řešení stínování, textury, animace, různé transformace (otáčení, změna měřítka, zkosení, kvaterniony), renderování atd. Tato práce by byla velmi časově náročná a navíc bychom nemohli využít HW akceleraci na grafické kartě. Proto je v této situaci velmi dobrý pomocníkem právě OpenGL.

OpenGL (Open Graphics Library) [15], [13] je programová knihovna poskytující API (aplikační programovací rozhraní) pro snadnou práci s počítačovou grafikou. Někdy je též označována jako standard, specifikující toto rozhraní. Stručně jej lze definovat jako softwarové rozhraní pro práci s grafickým hardwarem. Mezi její výhody patří snadná přenositelnost a vysoká výkonnost. Přenositelnost je na dobré úrovni nejen z platformy na platformu; OpenGL bylo zároveň navrženo tak, aby jej bylo možné použít v téměř libovolném programovacím jazyce. OpenGL je navrženo a chová se procedurálně, lze jej však použít i v objektově orientovaných jazycích. Další typickou vlastností je, že funguje jako stavový stroj, takže pokud je nastavena nějaká vlastnost, je tato vlastnost platná až do doby její další změny. Bylo vyvinuto a optimalizována firmou Silicon Graphic, Inc. (SGI). V současnosti je spravována konsorciem Architecture Review Board (ARB), jehož členy jsou např. Nvidia, AMD nebo Microsoft.

OpenGL bylo vytvořeno pro svou činnost na grafickém hardwaru — grafických kartách navržených pro práci s 3D grafikou, kde nabízí vysokou rychlost. Existuje však i čistě softwarová implementace, která není omezena na specifický hardware a lze ji tedy provozovat

na teoreticky libovolném stroji. Ta však disponuje značně nižším výkonem.

Použití OpenGL je skutečně široké. Uplatnění najde od CAD aplikací, přes letecké simulátory, multimediální aplikace, návrhářské a modelovací aplikace, až po počítačové hry. Stručně lze říci, že všude, kde se nějakým způsobem manipuluje s 3D grafikou, je možné (a často i velmi vhodné) použít OpenGL.

Velmi důležitou části OpenGL je GLSL (OpenGL Shading Language) [14]. Někdy je též označován jako samostatný, vysokoúrovňový programovací jazyk. Jedná se o specializovaný jazyk pro psaní shaderů — programovatelných jednotek grafické karty. Tímto způsobem lze vytvářet části aplikace, které budou zpracovávány přímo hardwarem a tedy velmi rychle, v reálném čase. To je možné díky použití masivní paralerizace. Syntaxe GLSL vychází z syntaxe jazyka C. Jsou zde podporovány stejné skalární datové typy, ale navíc také datové typy pro práci s vektory a maticemi, které se obecně v 3D grafice používají hojně. Existuje několik typů shaderů:

- **Vertex shader**. Tělo tohoto programu je vykonáno pro každý vrchol (anglicky *vertex*) renderované scény. V tomto typu shaderu nelze přidávat vertexy ani jinou geometrii — vždy je jeden vertex na vstupu a jeden na výstupu. Lze ho ale např. násobit maticemi, což se také často používá.

- **Fragment shader** . Někdy je též nazýván pixel shader. Tento shader se provádí nad každým pixelem výsledné scény, počítá se zde tedy jejich barva. V tomto shaderu lze aplikovat textury, ale také třeba počítat osvětlení, stíny, průhlednost a jiné efekty. Ve fragment shaderu je také možné vytvářet složitější efekty, jako např. *ambient occlusion* (česky zastínění okolím).

- **Geometry shader**. Jedná se o poměrně nový, tj. později přidaný shader. V OpenGL je teprve od verze 3.2. Jeho základní vlastností je, že může plně manipulovat s geometrií scény, tzn. může do scény přidávat nové prvky jako např. body, úsečky a trojúhelníky. Těmto prvkům se v OpenGL říká grafická primitiva. To dělá z geometry shaderu velmi mocný nástroj a umožňuje mu v reálném čase měnit povrch objektů přidáváním složitých efektů, jako např. trávy, různých zvrásnění povrchu, doplnění štěrku, efektů vody atd.

- **Tessellation shader**. Tento shader umožňuje také měnit geometrii scény, podobně jako geometry shader. Tento typ shaderu je podporován pouze u novějších grafických karet.

Alternativou k OpenGL může být DirectX [11]. Toto API je rozděleno na několik menších částí, např. Direct3D pro práci s 3D grafikou, DirectDraw pro práci s 2D grafikou, DirectWrite pro práci s fonty nebo DXGI pro správu monitorů a grafických adaptérů. DirectX bylo vytvořeno a je spravováno firmou Microsoft, z čehož plyne jeho velká nevýhoda a tou je možnost použití pouze na operačním systému Microsoft Windows (zatímco OpenGL může fungovat na nejrůznějších operačních systémech).

2.1.1 Vykreslovací řetězec

Vykreslovací řetězec [1] (anglicky rendering pipeline) znázorňuje, jakým způsobem OpenGL zpracovává data scény a převádí jejich obsah do framebufferu. Starší verze OpenGL pracovaly s fixním vykreslovacím řetězcem. To znamená, že chování každého prvku řetězce bylo

předem pevně stanovené. Od verze 2.0 však OpenGL obsahuje programovatelný vykreslovací řetězec. Programovat jeho jednotlivé prvky je možné pomocí shaderů. Díky tomu lze v reálném čase vytvářet i složité efekty přímo na grafické kartě.

2.1.2 GLEW

GLEW (OpenGL Extension Wrangler Library) je open-source knihovna pro OpenGL, která nabízí nástroj pro zjištění, která OpenGL rozšíření jsou dostupná na cílové platformě. To je možné provádět za chodu aplikace.

2.1.3 GLFW

GLFW je taktéž knihovna spolupracující s OpenGL. Tato knihovna usnadňuje práci s okny, vstupními zařízeními (myší, klávesnicí, joystickem), ale také třeba se schránkou nebo s časem.

2.2 Reprezentace 3D objektů

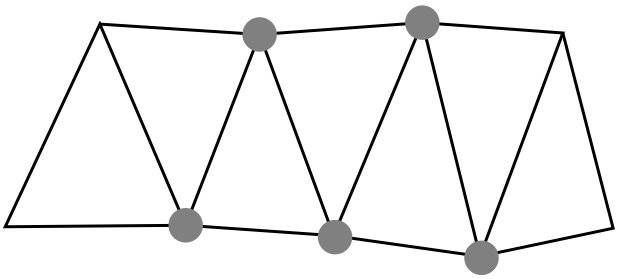
Reprezentace 3D objektů [10] určuje, jakým způsobem jsou tyto objekty v počítači uloženy. Zvláštní skupinu představují objekty typu úsečka, křivka nebo rovina. Ty mohou samozřejmě také existovat v 3D prostoru a existuje mnoho způsobů, jak je uložit. Tyto způsoby zde ale nebudou rozebírány. Způsobů popisu 3D dat existuje velké množství. Patří sem např. konstruktivní geometrie (CSG), dekompoziční modely, implicitní plochy, modelování pomocí deformací, ale také objemová reprezentace těles — popsaná v kapitole 2.2.2 a hraniční reprezentace těles, což je jedna z nejpoužívanějších. Tuto metodu lze ještě rozdělit např. na hranovou reprezentaci, jednoduchou plošnou reprezentaci, bodovou reprezentaci atd.; věnuje se jí kapitola 2.2.1. Podrobněji budou popsány pouze hraniční a objemová reprezentace, protože pouze ony jsou relevantní k vytvořené aplikaci.

2.2.1 Hraniční reprezentace

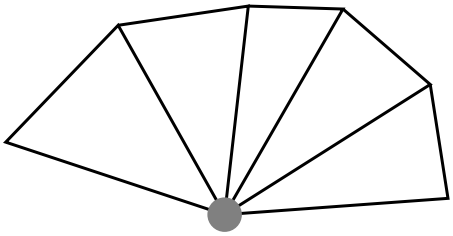
Hraniční reprezentace [10] je nejčastěji používanou reprezentací např. v počítačových hrách, nebo při tvorbě filmových efektů. Velmi se hodí také v případě, kdy je potřeba docílit realistické vizualizace. Pro její tvorbu se většinou používají specializované modelovací programy, jako jsou např. Autodesk 3ds Max, Autodesk Maya, Blender nebo Cinema 4D, které jsou často též vybaveny nástroji pro tvorbu animací. Objekty samotné jsou složeny ze sítě *polygonů*. Český překlad pro polygon je mnohoúhelník, v počítačové grafice se však téměř vždy používá anglická varianta a proto bude dodržována i v této práci. Oproti tomu slovo trojúhelník se v českých textech používá výhradně v české podobě. Polygony bývají často ještě rozloženy na jednotlivé trojúhelníky. Rozklad na trojúhelníky však někdy probíhá pouze interně v použitém programu a navenek se objekt jeví jako složen z polygonů. Tento rozklad má své opodstatnění ve výkonnosti. Zpracování jednotlivých trojúhelníků je velmi rychlé a téměř vždy hardwarově akcelerované. Další výhodou je, že vrcholy trojúhelníku vždy leží v rovině a existují rychlé algoritmy pro jeho vyplňování. Trojúhelníky se dále ukládají jako posloupnost tří bodů v prostoru.

Pokud by v souboru bylo uloženo více trojúhelníků za sebou, program, který bude tento soubor číst, bude vědět, že po každých třech bodech končí jeden trojúhelník a začíná další. Trojúhelníky lze také skládat do pruhu trojúhelníků (*triangle strip*) — pak vždy 3 trojúhelníky sdílí jeden vrchol. Tato situace je zobrazena na obrázku 2.1. Další možností skládání

trojúhelníků je skládat je do vějíře trojúhelníků (*triangle fan*). Pak může jeden vrchol sdílet ještě větší množství trojúhelníků, jak ukazuje obrázek 2.2. Při práci s trojúhelníky je občas třeba pracovat s algoritmy pro zjednodušování trojúhelníkové sítě nebo pro převod obecného mnohoúhelníku na trojúhelníky. Jak může polygonální model vypadat ukazuje obrázek 2.3.



Obrázek 2.1: Skupina trojúhelníků sdružených do pruhu trojúhelníků. Šedě vyznačené body označují vrcholy sdílené třemi trojúhelníky.



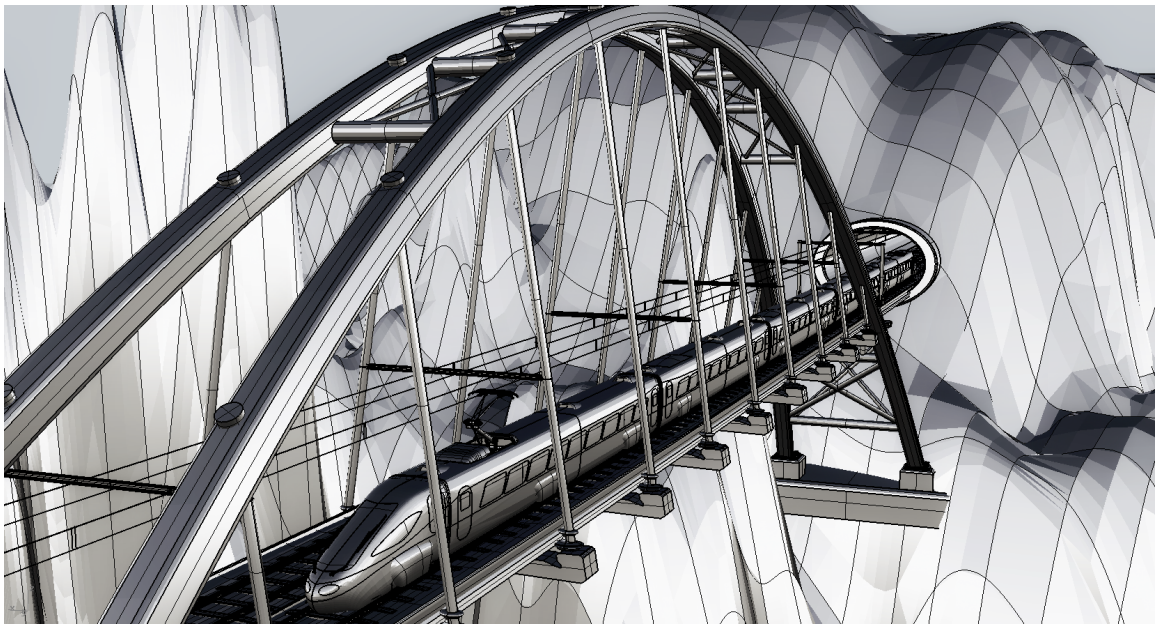
Obrázek 2.2: Skupina trojúhelníků sdružených do vějíře trojúhelníků. Je vidět, že jeden vrchol je v tomto případě sdílen hned pěti trojúhelníky.

Do hraniční reprezentace spadá i tzv. hranová reprezentace. V tomto případě se ukládají pouze informace o vrcholech a hranách tělesa, ale nikoli už o plochách. Tento model je velmi starý a jednoduchý. V souboru je pak uloženy seznam hran a seznam vrcholů. Zásadní nevýhodou této reprezentace je její nejednoznačnost.

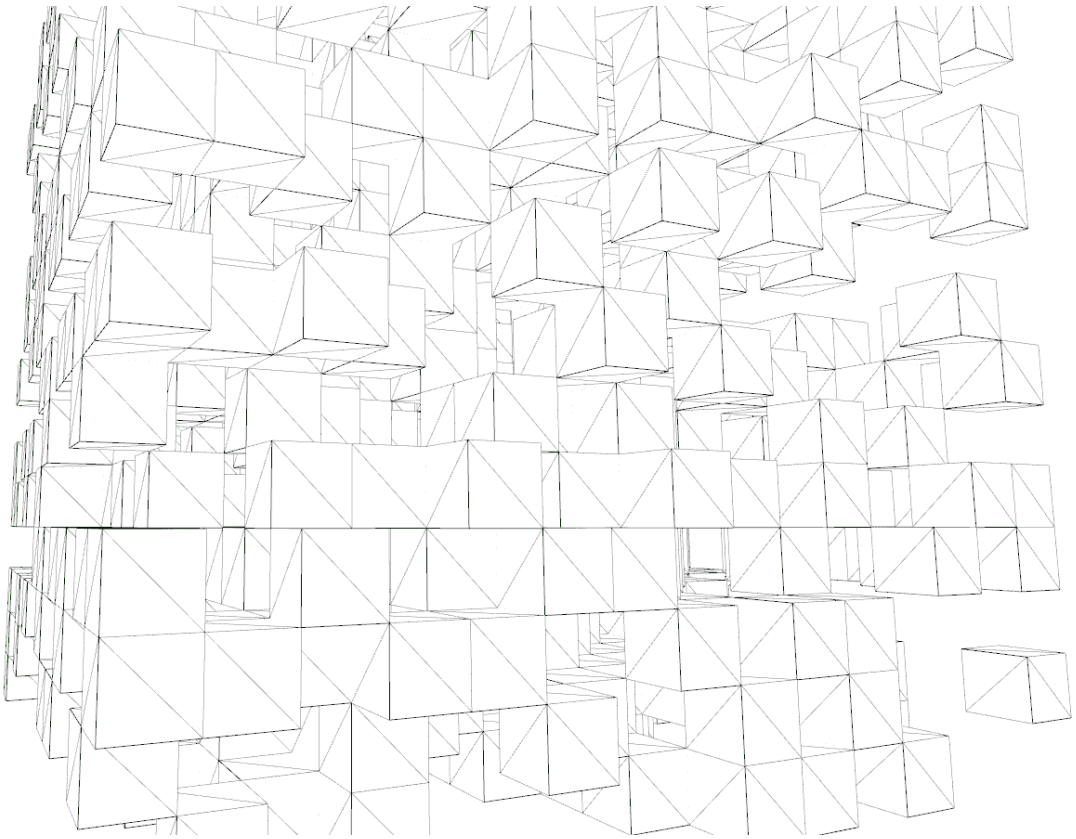
2.2.2 Volumetrické modely

Volumerticky reprezentovaný model [10] si lze představit jako klasický rastrový obrázek, ale se třemi rozměry. Tento obrázek je tedy tvořen 3D mřížkou hodnot. To, čemu se u klasického obrázku říká pixel, se zde jmenuje *voxel* (volumetrický element). Výhodou této reprezentace oproti hraničnímu modelu je to, že obsahuje informace o vnitřní struktuře modelu. To samozřejmě platí pouze do určité úrovně; k dispozici jsou informace o tom, kde jsou a nejsou voxely, už ale nejsou informace o tom, co je uvnitř jakého voxelu. Je však možné tuto metodu zkombinovat s tzv. *oktanovým stromem* (anglicky *octree*) a tím získat v místě potřeby další subvoxely a hierarchicky tak zvýšit „rozlišení“ modelu tam, kde je to třeba. Příklad volumetrického modelu zobrazuje obrázek 2.4. Jeden voxel je zde reprezentován jednou krychlí.

Volumetrické modely se typicky používají v počítačové tomografii nebo simulaci proudění kapalin. Také jsou vhodné pro popis objemových a diskrétních objektů, jako jsou např.



Obrázek 2.3: Ukázka polygonálního modelu ve stínovém zobrazení (hraniční reprezentace).



Obrázek 2.4: Ukázka zobrazení volumetrických dat.

mraky nebo mlha. Dále se používají v geologii nebo ve strojírenství. Výhodou těchto modelů je naprostá nezávislost na složitosti modelovaného objektu. Naopak nevýhodou můžou být (především u rozměrnějších objektů) vysoké nároky na paměť a při zpracování i na procesorový čas.

2.3 Procedurální modelování

V současné době známe tři způsoby získávání 3D grafiky. První a pravděpodobně nejčastěji používaný se jmenuje modelování. V tomto případě je zapotřebí člověka, který pracuje s modelovací aplikací a pomocí vstupních zařízení počítače vytváří model. Tato metoda je velmi pracná a pomalá, ale přináší nejvěrohodnější výsledky. Druhou možností je využít nějakou snímací techniku. Tou může být např. speciální skener, který je schopný oskenovat menší objekty, nebo i obyčejný digitální fotoaparát, s jehož pomocí je vytvořna série snímků zkoumaného objektu z více směrů a speciální aplikací jsou pak tyto snímky zpracovány a převedeny do 3D modelu. Tato metoda není příliš přesná a lze ji použít pouze na objekty s omezenou členitostí, ale používá se např. při tvorbě modelů měst, u kterých se snímky získavají z letadla. Poslední metodou je metoda procedurálního modelování [10], kterou se zabývá tato kapitola.

Při této metodě je model vytvářen nějakým algoritmem. Do této kategorie patří i např. generování ploch rotací, tažením nebo z křivek, ale tyto případy v této kapitole nejsou řešeny. Objekty, které tímto způsobem vznikají, jsou nejčastěji podobné objektům, se kterými se můžeme setkat v přírodě. Klasicky sem patří např. travnaté porosty, rostliny, sněhové vločky, jeskyně, krajiny, stromy atd. Pokud se programátorovi podaří přesně popsat strukturu a vzhled těchto objektů pomocí algoritmu, získá možnost generovat neomezený počet těchto objektů a přitom se žádný z nich nebude opakovat. To je veliká výhoda procedurálního modelování a motivace pro jeho výzkum. Tato vlastnost je velice výhodná v počítačových hrách, protože může hráči dodat dojem neomezeně velkého herního světa, který přitom na disku počítače zabírá velmi málo místa.

Procedurální modelování lze rozdělit do tří skupin. První z nich jsou *L-systémy*. Tyto systémy se používají především pro generování rostlin a vycházejí z gramatik. Mohou být stochastické a nedeterministické. Druhou skupinou je *fraktální geometrie*, která se používá pro generování objektů typu korály, stromy, krajiny a hory. Zde se velmi často používají rekurzivní algoritmy, které pracují se soběpodobností — objekt není náchylný na změnu měřítka, lze jej do nekonečna přibližovat nebo oddalovat. Poslední a trochu vybočující skupinou jsou *systémy částic*. Tato skupina naráží na problematiku modelování simulací a používá se např. pro modelování sněžení, proudění atd.

Mezi nejčastěji řešené problémy v této oblastí patří generování krajiny. Pro uložení krajiny se často používá dvourozměrná mřížka, u které je pro každý bod definována jeho nadmořská výška. Tento koncept je velmi jednoduchý a nedovoluje modelovat např. převisy. Někdy je označován jako 2,5 dimenzionální. Při tvorbě krajiny se často používají fraktální metody jako např. metoda přesouvání středového bodu, nebo metoda náhodných poruch. Model také často vzniká ve dvou fázích. První fází je získání přibližného tvaru krajiny různými způsoby, druhou fází je pak aplikace tzv. erozních algoritmů, které zajistí, že výsledek bude vypadat více realisticky. Zapracovat do modelu lze např. termoerozi (eroze způsobená vlivem teploty), hydroerozi (eroze způsobená vlivem působení vody), nebo větrnou erozi. Díky těmto postupům, které simulují, jak tvar objektu skutečně v přírodě vzniká, lze dosáhnout velmi realistických výsledků. Za zmínku stojí také to, že procedurálně lze generovat i textury, které budou podrobněji přiblíženy v kapitole 2.6.

2.4 Šumové funkce

Šum [10], [9] je obecně nějaké znečištění, které může být např. obrazové, zvukové, hlukové atd. Takové šumy jsou většinou nechtěným vedlejším produktem nějakých procesů. Například i do metalických kabelů přenášejicích data se může indukovat šum. Na analogových televizorech se vizuální šum projevoval typickým „zrněním“. U fotografií se šum projevuje barevným roztřesením a vzniká v důsledku nedostatku světla. Ve všech těchto případech je šum nežádoucí. V počítačové grafice však může být šum považován za informaci a existují dokonce algoritmy pro generování šumu. Takovýmto algoritmem je např. *Perlinův šum* popsaný v kapitole 2.4.1, nebo jeho vylepšení *Simplex noise* popsaný v kapitole 2.4.2. Tyto šumy lze použít např. při procedurálním generování textur.

2.4.1 Perlinův šum

Perlinův šum (anglicky Perlin noise, ale používá se i česká varianta) [10], [9] je rychlá šumová funkce, která se často používá při procedurálním generování grafiky. Funkci navrhl v roce 1985 Ken Perlin. Perlinova šumová funkce má tyto vlastnosti [10]:

- Metoda je statisticky invariantní vzhledem k otáčení i posunutí.
- Je spojitá.
- Funkce má omezené frekvenční spektrum.
- Stejné vstupní hodnoty produkují vždy stejné výstupní hodnoty. To znamená, že funkce je tzv. opakovatelná.

První bod říká, že nezáleží na tom, na jakých souřadnicích se začne funkce generovat, ani s jakým natočením se na ni bude pohlížet. Hodnoty funkce budou v každém jejím místě soběpodobné. Díky tomu, že je funkce spojitá, má svou maximální frekvenci a je teoreticky náchylná ke generování grafických artefaktů, těm však lze použitím vhodné funkce zabránit. Opakovatelnost této funkce je velmi důležitá, protože zajišťuje, že např. při použití funkce pro generování textury, vzdálení se od objektu a opětovné přiblížení (které vyvolá nové generování textury) se textura vygeneruje stejná jako při první návštěvě a objekt bude tedy konzistentní. Důležitou vlastností Perlinova šumu je, že je možné jej vytvořit pomocí generující funkce pracující s libovolným počtem dimenzí. V praxi se však nejčastěji používají dvě až čtyři dimenze.

Perlinova funkce pracuje s prostorem rozděleným do pravidelné mřížky. V každém vrcholu této mřížky je definována funkce zvaná *vlnka*. Vlnka je pseudonáhodná funkce procházející počátkem. Proces výpočtu pro jeden bod odpovídá následujícím krokům:

1. Rozhodne se, ve které buňce mřížky je dotazovaný bod umístěn.
2. Pro všech osm vrcholů této buňky je vypočtena vlnka.
3. Tyto hodnoty vlnek se sečtou.

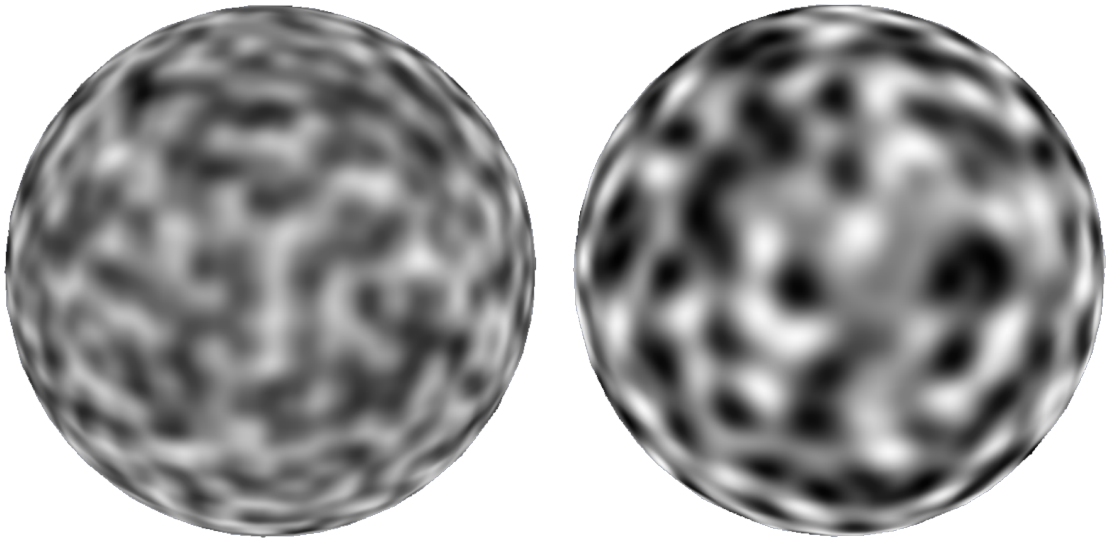
Perlinův šum se používá pro generování procedurálních textur, ale také je často využíván k vytvoření např. oblaků, nebo k docílení efektů vodní hladiny nebo kouře.

2.4.2 Simplex noise

Za vylepšení Perlinova šumu lze pokládat *Simplex noise* [9], [6] (tento název zatím nemá český ekvivalent). Tento algoritmus byl navržen opět Kenem Perlinem a to v roce 2001. Jeho výhodami oproti Perlinovu šumu jsou:

- Simplex noise je méně výpočetně náročný a vyžaduje menší počet násobení.
- Jeho nižší výpočetní náročnost se ještě více projevuje s narůstajícím počtem dimenzí. Jeho časová složitost je $O(N^2)$ [6], na rozdíl od $O(2^N)$ u klasického Perlinova šumu. N je v tomto označení složitosti počet dimenzí.
- Tento šum neprodukuje viditelné artefakty.
- Simplex noise má snadno definovaný spojitý gradient, který je proto jednoduše spočitatelný.
- Simplex noise je snadno implementovatelný v hardware.

Princip činnosti je podobný jako u Perlinova šumu, ale Simplex noise pracuje s vždy nejjednodušším a nejkompaktnějším tvarem, který může vyplnit N dimenzionální prostor. Např. pro dvourozměrný prostor je tímto tvarem rovnostranný trojúhelník. Obecně jde však vždy o útvar s $N + 1$ vrcholy. Z toho pramení hlavní rozdíl oproti Perlinovu šumu, který vždy používá hyperkrychli s 2^N vrcholy. Srovnání vizuálních výsledků Perlinova šumu a Simplex noise ukazuje obrázek 2.5.



Obrázek 2.5: Srovnání vizuálních výsledků Perlinova šumu (vlevo) a Simplex noise (vpravo). Oba obrázky ukazují byly získány použitím třídimenzionální varianty funkce. [1]

[1] Obrázek byl převzat z http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf a upraven.

2.5 Geometrie

V této kapitole budou popsány algoritmy využívané pro tvorbu polygonálního modelu z objemové reprezentace. V kapitole 2.5.1 je uveden princip a základní informace o metodě Marching cubes. Tato metoda sice nebyla použita ve výsledné aplikaci, ale byla implementována a testována a proto je i zde popsána. Následující kapitola 2.5.2 pak pojednává o algoritmu Marching tetrahedra.

2.5.1 Marching cubes

Marching cubes [16] je algoritmus sloužící ke konverzi 3D modelu uloženém ve formě volumetrických dat do hraniční reprezentace (polygonálního modelu). K tomu je zapotřebí nalézt *izoplochy*.

Pro vysvětlení pojmu izoplocha je vhodné nejprve popsat, co je to *izolinie*. Izolinie je typem značky v mapě nebo grafu. Jsou charakteristické tím, že spojují místa se stejnou hodnotou nějaké třetí veličiny (třetího rozměru). Jejich další vlastností je, že se nemohou křížit. Typickým příkladem izolinie je vrstevnice (*izohypsy*), u kterých je onou třetí veličinou nadmořská výška. Dá se říci, že čím jsou izolinie pro 2D diagramy, tím jsou izoplochy pro 3D objekty.

Algoritmus byl popsán v roce 1987 dvojicí W. E. Lorensen a Harvey E. Cline. Celý článek je dostupný zde [17].

Algoritmus

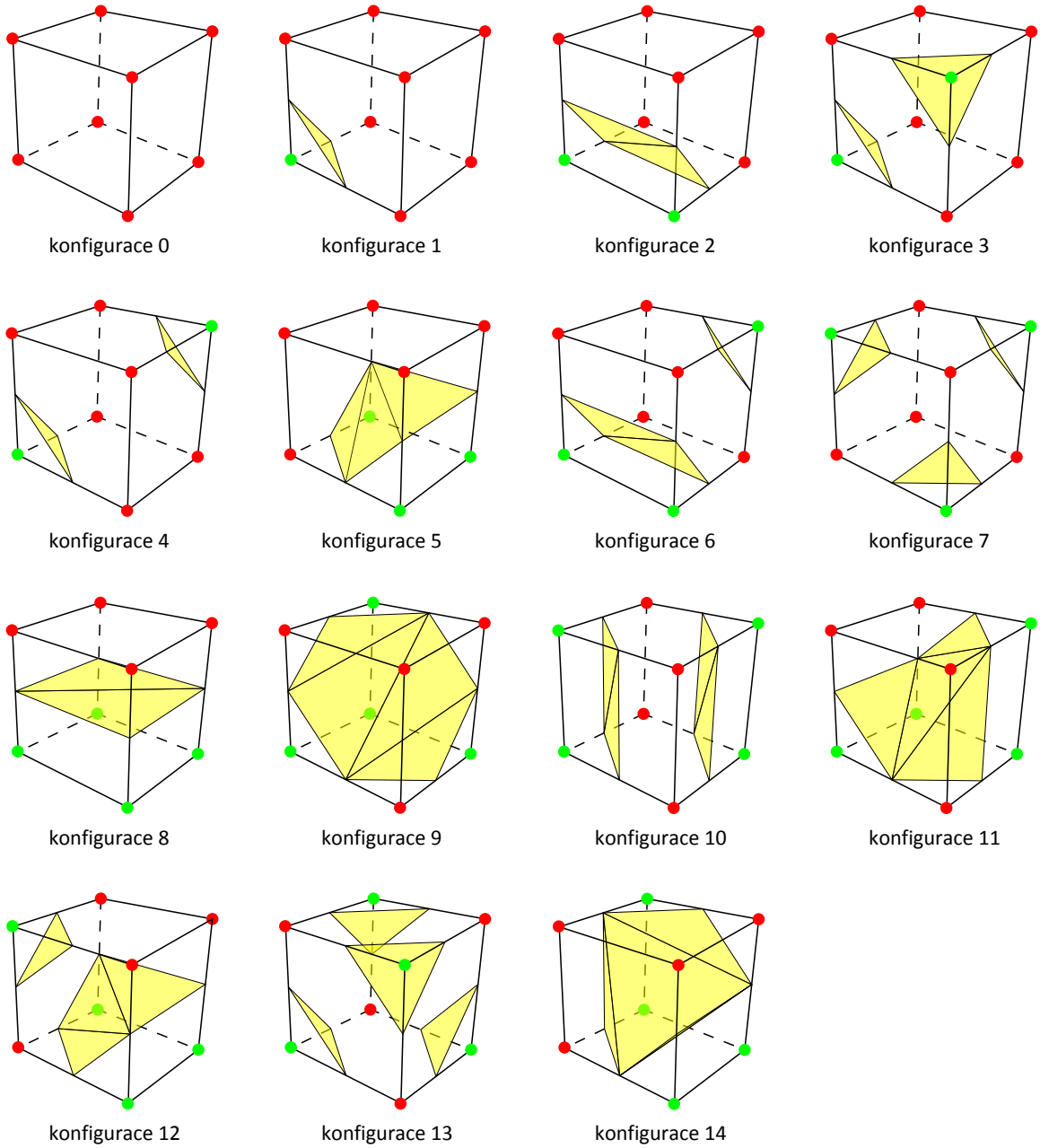
Na vstupu algoritmu je 3D mřížka bodů. U každého z těchto bodů je známo, zda je součástí modelovaného objektu nebo není, nebo je známa alespoň jeho intenzita v tomto bodě. (Intenzitu je možno získat např. z nějaké šumové funkce apod.) Pokud je známa pouze intenzita, je třeba tuto hodnotu nějak diskretizovat. To se provádí stanovením prahu, který rozhodne o tom, zda je bod součástí objektu či nikoliv.

Algoritmus vybere prvních 8 bodů mřížky tvořících imaginární krychli. Podle „orientace“ těchto bodů — tedy toho, zda se nacházejí uvnitř nebo vně modelovaného objektu je této krychli přiřazena jedna z možných konfigurací. Těchto konfigurací může být celkem 2^8 (pracuje se s 8 vrcholy a každý má 2 možné stavy), tedy 256. Přehled základních konfigurací zobrazuje obrázek 2.6.

Podle výsledné konfigurace jsou pak do výstupního modelu vloženy 1 až 4 trojúhelníky. Tato imaginární kostka je poté posouvána postupně po všech třech osách až se tímto způsobem projde celá mřížka a tím je vytvořen polygonální model. Přirozeně tam, kde má všech 8 vrcholů stejnou hodnotu (jsou tedy celé buď v modelu nebo mimo model), se negenerují žádné polygony. Příkladový model vytvořený algoritmem Marching cubes se nachází na obrázku 2.7.

Nekompatibilní stavy

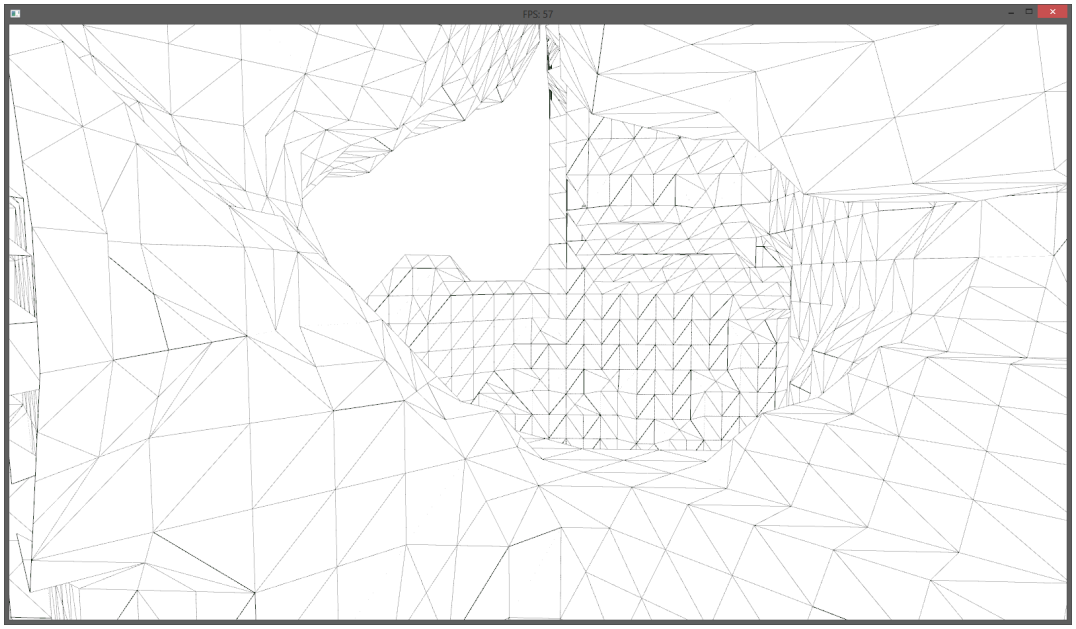
Algoritmus, tak jak byl do teď popsán, není dokonalý. V jistých situacích může generovat chyby — díry v modelu. Taková situace nastane, pokud budou vedle sebe 2 imaginární krychle v tzv. *nekompatibilních stavech*. To je způsobeno tím, že některé konfigurace imaginární krychle jsou dvojznačné. Řešením popsaného problému je zavést k těmto krychlím tzv. doplňkové konfigurace, které ukazuje obrázek 2.8. Např. pokud by byly vedle sebe umístěny krychle v konfiguracích č. 3 a 6 podle obrázku 2.6, vznikne v modelu díra. Přitom umístění



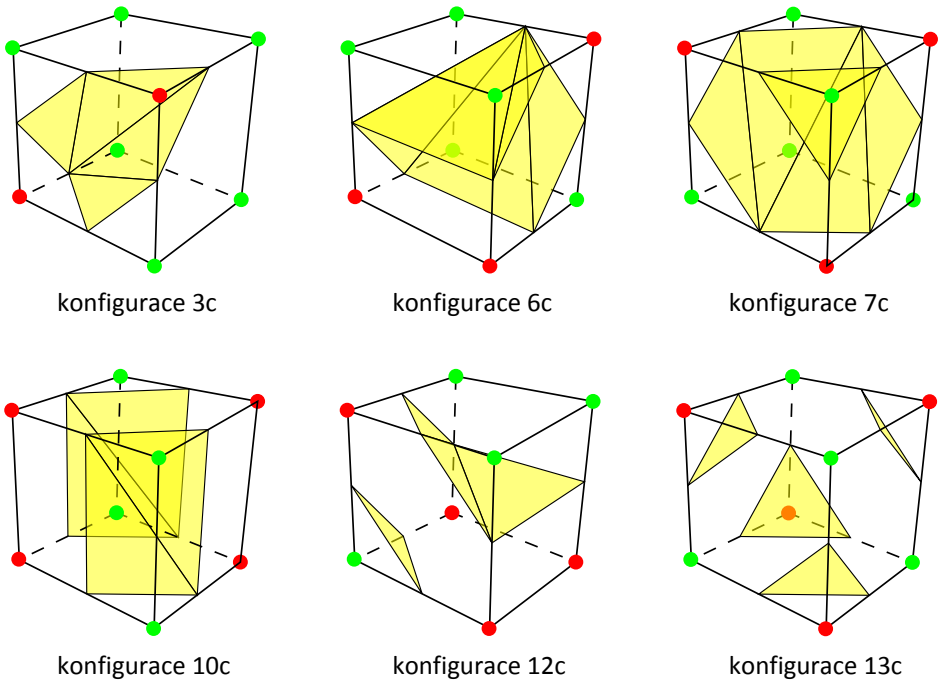
Obrázek 2.6: Základní konfigurace krychle v algoritmus Marching cubes. Červeně označené vrcholy jsou považovány za vrcholy umístěné ve vzduchu, zatímco zeleně označené vrcholy jsou součástí skály. Žlutě jsou zvýrazněné produkované trojúhelníky. Samozřejmě, při záměně klasifikace vrcholů ze skály na vzduch a naopak by se konfigurace nezměnila, pouze by byly obrácené normály vkládaných trojúhelníků.

těchto dvou konfigurací vedle sebe je možné, protože na jejích krychlích lze najít stěnu se stejným ohodnocením vrcholů. Řešením je tedy vedle krychle v konfiguraci 3 umístit krychli v konfiguraci 6c namísto 6.

Problém je, jak rozhodnout, zda použít primární nebo doplňkovou konfiguraci. K vyřešení tohoto problému je třeba „nahlédnout“ na sousední krychli a řešenou konfiguraci



Obrázek 2.7: Ukázka modelu vytvořeného pomocí Marching cubes.



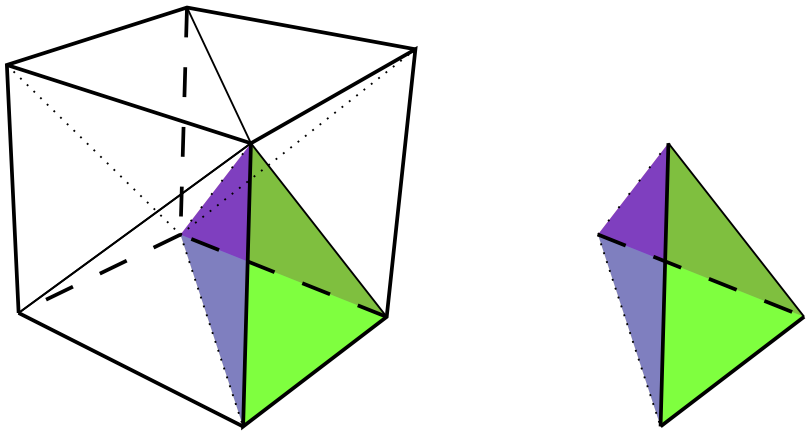
Obrázek 2.8: Doplňkové konfigurace algoritmu Marching cubes. Značení vrcholů je shodné s obrázkem 2.7. Krychle mají stejné vstupní ohodnocení vrcholů jako u primárních konfigurací, ale generují jiné množiny polygonů.

zvolit tak, aby model zůstal celistvý. Tento postup však významně zpomaluje algoritmus a činí jej tak značně méně výhodným.

2.5.2 Marching tetrahedra

Alternativou k Marching cubes (2.5.1) může být algoritmus Marching tetrahedra [4]. Ten z Marching cubes vychází, ale řeší jeho hlavní nedostatek — nekompatibilní stavy.

Na úvod je třeba vysvětlit, co je to *tetrahedron*. Tetrahedron česky znamená čtyřstěn, ale protože se algoritmus jmenuje Marching tetrahedra a jeho název se do češtiny nepřekládá, budu dále používat anglickou variantu. Tetrahedron je obecně čtyřstěn a může jím být i zcela pravidelný čtyřstěn. V případě marching tetrahedra jde ale o čtyřstěn, který vznikne rozdělením krychle na 6 shodných částí, pouze zrcadlově převrácených. Takový čtyřstěn je složen ze čtyř pravoúhlých trojúhelníků, z nichž 2 jsou navíc rovnoramenné.



Obrázek 2.9: Krychle rozdělená na 6 tetrahedronů a vedle ní jeden „vysunutý“ tetrahedron.

Algoritmus

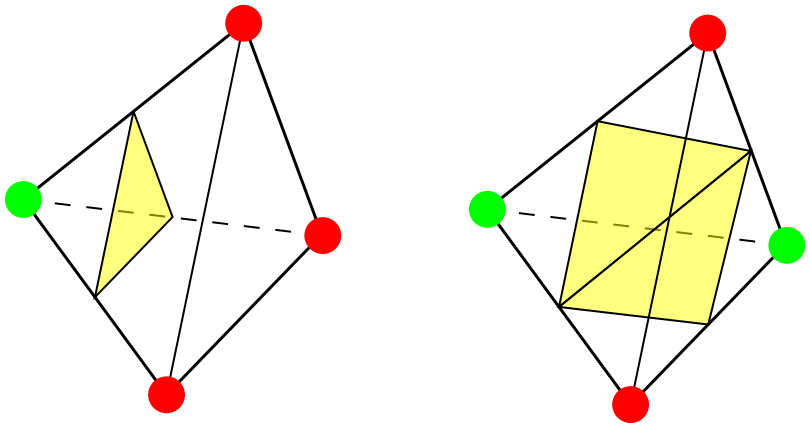
Jak již vyplývá z názvu, algoritmus nepracuje s celými imaginárními krychlemi (tak jako Marching cubes, dále jen MC), ale s imaginárními tetrahedrony. Díky tomu je algoritmus výrazně jednodušší a tudíž může být i rychlejší. Nejprve je tedy zpracovávaná krychle rozložena na šest tetrahedronů. Další postup je podobný jako u MC. Do výsledného modelu jsou vloženy trojúhelníky podle toho, které vrcholy jsou součástí modelu. Mohou nastat pouze dvě situace. Buď jsou rozdílné hodnoty v jednom vrcholu, nebo ve dvou. Pokud je součástí vstupního modelu jeden vrchol, vkládá se do výsledného modelu jeden trojúhelník. Pokud jsou součástí vstupního modelu dva vrcholy, vkládají se do výsledného modelu dva trojúhelníky. Pokud by byly tři, je situace stejná jako v případě, že je pouze jeden, jen výsledný trojúhelník bude mít opačnou normálu. Přehled těchto situací ukazuje obrázek 2.10.

V imaginární krychli je tímto způsobem zpracováno všech šest tetrahedronů a stejně tak jsou takto zpracovány všechny krychle ve vstupním modelu.

Výhodou oproti MC je vedle jednoduchosti a absenci nekompatibilních stavů také vyšší kvalita, což je způsobeno větším počtem generovaných trojúhelníků.

2.6 Textury

Textury [10] přináší výhodný způsob, jak zvýšit vizuální kvalitu modelů za nízkou cenu. Tím je myšleno, že práce s texturami je poměrně rychlá a výhodná je i z pohledu potřebného



Obrázek 2.10: Možné konfigurace tetrahedronu v algoritmu Marching tetrahedra s vyznačenými výslednými polygony (žlutě). Zeleně jsou zvýrazněny vrcholy nacházející se uvnitř skály, červeně vrcholy umístěné ve vzduchu. Levá polovina obrázku zobrazuje situaci s jedním bodem uvnitř skály, zatímco pravá polovina obrázku ukazuje situaci se dvěmi body uvnitř skály.

výpočetního výkonu. To platí minimálně ve srovnání se situací, kdy by bylo třeba stejného efektu dosáhnout bez použití textur, protože je výhodnější vytvořit jednoduchou geometrii a potřebné detaily přidat aplikací textury, než vytvářet složitou geometrii a jednoduché textury. Vizuální kvalita přitom může být na srovnatelné úrovni. Typicky se textury aplikují na povrchy budov, billboardů, silnic, střech atd.

Textura definuje vlastnosti povrchu v daném bodě. Těchto vlastností může být celá řada: od barvy, odrazivosti, přes hrbolatost až třeba po průhlednost. Kombinace těchto druhů textur (a dalších vlastností povrchu) tvoří materiál. Základní stavební prvek textury je *texel*.

Textury je možné rozdělit podle počtu jejich rozměrů. Používají se jedno, dvou, tří a čtyř rozměrné textury. Jednorozměrnou texturu si lze představit jako obarvenou čáru. Je možné ji použít např. pro tvorbu barevných přechodů nebo duhy a její výhodou je velmi malá paměťová náročnost. Kromě „klasických“ užití ji lze použít i např. pro obarvení mapy podle nadmořské výšky a vytvořit tak tzv. *fyzickou mapu*. V kombinaci s průhledností je s pomocí jednorozměrné textury možné snadno vytvořit efekt deště. Nejčastěji používaným typem textur jsou však textury dvourozměrné. Používány jsou v téměř každé 3D počítačové hře, simulátoru, návrhu nebo vizualizaci. Jsou charakteristické tím, že jsou vždy nanášeny na povrch objektu. I když to ve výsledné vizualizaci není viditelné, trojrozměrné textury nesou informaci o vnitřní struktuře materiálu. Proto se používají u objektů z takovýchto materiálů vyrobených. Těmi může být např. mramor nebo plísňový sýr. V těchto situacích by bylo možné použít i vhodnou 2D texturu, 3D textura však poskytne větší realističnost, nebude tak zřetelné, že použitá textura se opakuje a je možné ji snadno použít i u objektů, jejichž povrch nelze bez deformací rozvinout do 2D plochy. V těchto případech je totiž *mapování* 2D textur problematické a dochází zde k deformacím. Mapování bude podrobněji vysvětleno v následujících odstavcích. Nevýhodou trojrozměrných textur je velmi vysoká paměťová náročnost. Čtyřrozměrné textury se používají k animaci trojrozměrných. Dají se tedy využít např. k vytvoření efektu plápolajícího ohně.

Jak již bylo zmíněno v kapitole 2.3, textury lze generovat i procedurálně. Pak sa-

mozřejmě neplatí informace o paměťové náročnosti uvedené v předchozím odstavci. To se často používá u 3D textur. Textury lze aplikovat buď softwarově — pak je forma aplikace plně pod kontrolou programátora, nebo pomocí k tomu určených *texturovacích jednotek*.

2.6.1 Mapování textur

Mapování textur [10], [8] určuje, jakým způsobem se bude textura na objekt nanášet. Je zde třeba určit, kam se bude textura nanášet, ale také třeba její měřítko, natočení a jak se bude pokračovat v případě, že textura bude menší než objekt na který se nanáší.

Prvním krokem při mapování textur je nalezení funkce, přiřazující bodům na povrchu mapovaného objektu texely mapované textury. Tato funkce bývá nazývána *inverzní mapování* a musí zohledňovat tvar texturovaného objektu. Aplikaci 2D textury si lze představit jako polepení objektu papírem, aplikaci 3D textury pak jako vyřeznání objektu z materiálu s odpovídající vnitřní strukturou. Jak již z uvedeného příkladu u mapování 2D textur vyplývá, problém je u objektů, u kterých by bylo problematické takový papír vytvarovat. Jedná se o objekty, které nelze bez deformací rozvinout do 2D plochy, např. prohnutá váza, vejce nebo čajová konvice. V takových případech dochází ke zkreslení textury, s kterým je třeba při jejím návrhu počítat a navrhnout ji tak, aby při jejím zkreslení mapováním vypadala na povrchu objektu přirozeně. Problém zkreslení dobře ilustruje mapa světa ve *Mercatorově* zobrazení. U tohoto zobrazení je na mapě větší Grónsko než Austrálie, i když ve skutečnosti je tomu naopak. To je způsobeno tím, že Grónsko je blíže pólům, kde je i zkreslení větší. Směrem k pólům se totiž deformace zvyšuje a v pólech je dokonce textura smrsknuta do jednoho bodu. Tento příklad je analogií nanášení textury na kouli, kdy texturou je mapa světa a koulí je Zeměkoule.

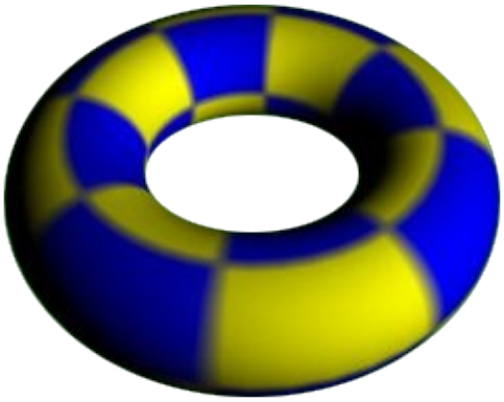
Problém nastává, pokud rozměry dvourozměrné textury po upravení jejího měřítka nekorespondují s rozměry objektu, na který se nanáší. Pokud je textura větší než objekt, je zpravidla nanášením oříznuta a na objektu se pak nezobrazí celá její plocha. Situace je o něco složitější v případě, že je textura menší než texturovaný objekt. V zásadě existují tři možnosti, jak tento problém řešit. První možností je jednoduše nanést texturu a místa, kam už textura „nevystačí“, ponechat bez textury. Druhou možností je opakování textury, které je možné nastavit v obou směrech. V tomto případě je ale třeba zajistit, aby na sebe textura navazovala. To znamená, aby její levý okraj plynule přecházel do pravého okraje a horní okraj přecházel do spodního okraje. Tím je zajištěno, že nevzniká žádný viditelný přechod při položení textury vedle sebe. Opakování textury může být výhodné z hlediska úspory potřebné paměti a hodí se např. pro cihlovou stěnu. Na druhou stranu, pokud se použije nevhodně (např. pro travnatý porost), může takovéto použití textury snižovat kvalitu výsledného dojmu ze scény. Poslední možností je, že se použije hodnota z okraje textury a tato hodnota se roztáhne do potřebných rozměrů.

2.6.2 Bump mapping

Bump mapping (do češtiny lze přeložit jako *hrbolaté textury*, nebo *mapování hrbolatých textur*, ale spíše se používá anglická varianta) [7], [10] je způsob, jak docílit efektu hrbolatého povrchu bez změny jeho geometrie. Tato metoda je velmi výhodná, protože skutečně vytvořit geometricky hrbolatý povrch by znamenalo vyvinout výrazně vyšší nároky jak na procesor, tak na paměť. Navíc bump mapping se často aplikuje v fragment shaderu (viz. kapitola 2.1) a proto může být použit v reálném čase. Na druhou stranu kvalita efektu není perfektní a zvláště na obrysech objektů je jasně patrné, že geometrie je ve skutečnosti

hladká. Navíc silně zvrásněný povrch by vrhal stín sám na sebe a to tato metoda také simulovat nedokáže.

Bump mapping je přímo závislý na použitém osvětlovacím modelu. Může fungovat pouze s těmi osvětlovacími modely, které pracují s normálovým vektorem povrchu (normálový vektor, nebo zkráceně jen normála je vektor kolmý k povrchu tělesa a lze s pomocí něj také zjistit, která strana plochy je „vnitřní“ a která „vnější“). Tato technika tedy upravuje normálové vektory aby odrážely světlo takovým způsobem, jako by povrch byl skutečně hrabolatý, čímž vzniká hrbolatý efekt. Metoda potřebuje odněkud čerpat informace o tom, jakým způsobem má normály upravit. Zdroji těchto informací se říká *normálová mapa* a lze si ji představit jako dvourozměrnou texturu ve stupních šedi. Normálovou mapu lze také generovat procedurálně. Bump mapping se hodí např. pro vytvoření efektu vlnek na vodní hladině, cihel na stěně, šupin na zvířeti, drsné oceli atd. Ukázka objektu s aplikovanou pouze klasickou texturou je na obrázku 2.11, stejný objekt po aplikaci bump mapping je na obrázku 2.12. Použitou normálovou mapu ukazuje obrázek 2.13.



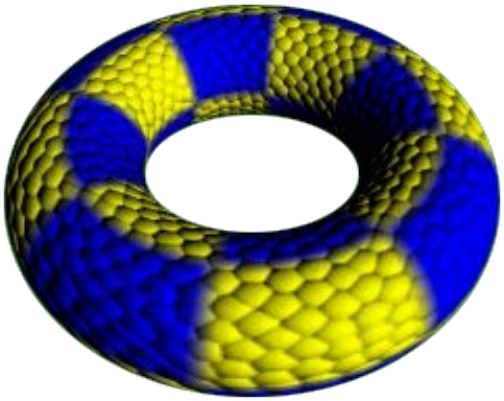
Obrázek 2.11: Objekt s aplikovanou pouze klasickou texturou. [2]

2.7 Phongovo stínování

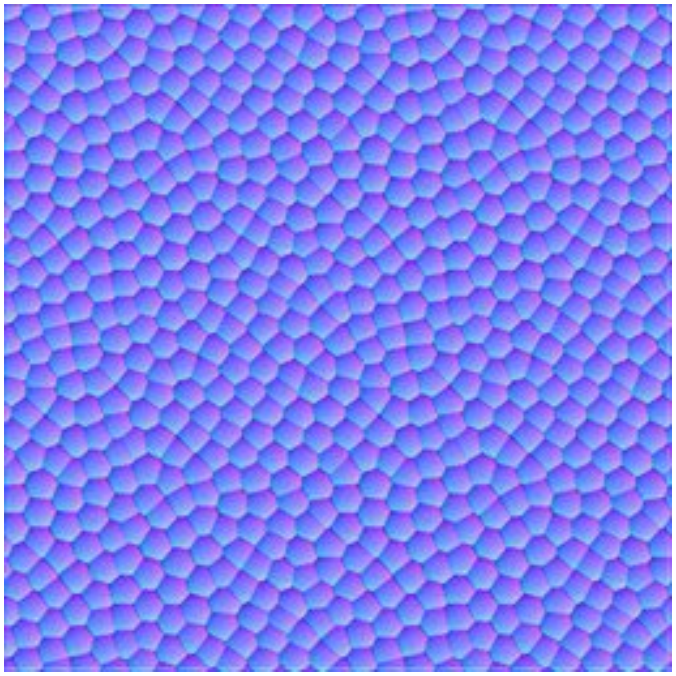
Phongovo stínování (často zaměňováno s *Phongovým osvětlovacím modelem*) [10], [12] je technika vyvinutá ke spojitému stínování objektů. Na rozdíl od *konstantního stínování* (anglicky *flat shading*), které je poměrně primitivní a plochu každého polygonu zobrazuje celou jednou barvou, má Phongovo stínování tu vlastnost, že objekty tvořené množinou rovinných plošek stínuje hladce. Tuto vlastnost splňuje i *Gouraudovo stínování*, které však stále nepodává příliš věrohodné výsledky, protože neumí zobrazit *zrcadlovou složku světla*. Tuto složku přidává právě Phongův osvětlovací model. Někdy je též nazývána *lesklé světlo* a je charakteristická právě pro Phongův osvětlovací model.

Phongovo stínování je tedy ze zmíněných nejdokonalejší, ale bohužel i nejpomalejší metodou. Jak již bylo zmíněno, plochy tvořené sítí navazujících polygonů zobrazuje tak, jako by byly dokonale hladké. Toho je docíleno použitím *bilineární interpolace*. Phongovo stínování pracuje s interpolací normál, ale samo o sobě ještě nemusí obsahovat zrcadlovou

[2] Obrázek převzat z http://www.paulsprojects.net/tutorials/simplebump/simplebump.html a upraven.



Obrázek 2.12: Stejný objekt jako na obrázku 2.11, ale nyní s přidaným bump mappingem. Je vidět, že povrch sice působí hrbolatě, ale okraje dokazují, že povrch je ve skutečnosti hladký. [3]



Obrázek 2.13: Normálová mapa použitá u obrázku 2.12. [4]

složku světla. Tato metoda je relativně pomalá, protože je zpracovávána pro každý fragment výsledného obrázku.

Phongův osvětlovací model dosahuje tak vysoké kvality, protože počítá se třemi složkami světla. Těmito složkami jsou: *okolní světlo* (anglicky *ambient light*), *difúzní světlo* (anglicky *diffuse light*) a *lesklé světlo* (anglicky *specular light*). Okolní světlo určuje množství světla,

[3] Obrázek převzat z http://www.paulsprojects.net/tutorials/simplebump/simplebump.html a upraven.
[4] Obrázek převzat z http://www.paulsprojects.net/tutorials/simplebump/simplebump.html.

které dopadá na objekty ve scéně ve všech směrech a všude se stejnou intenzitou. Napodobuje sekundární odražené a rozptýlené světlo a je v modelu proto, aby ty části scény, na které nedopadá žádné světlo přímo, nebyly zcela černé. Okolní světlo bývá někdy označováno jako *globální osvětlení*. Značí se E_a a jeho výpočet probíhá podle vztahu:

$$E_a = C_d \cdot K_a$$

kde C_d je barva difúzní složky, tedy vlastně barva povrchu a K_a je koeficient okolního světla (někdy též odrazový koeficient materiálu). Ten nabývá hodnot od 0 do 1. Hodnota 0 znamená, že okolní světlo se od povrchu vůbec neodráží a je beze zbytku pohlceno; naopak pro hodnotu 1 se všechno okolní světlo odráží. Okolní světlo samo o sobě nevytváří dojem prostorových objektů.

Druhou zastoupenou složkou je difúzní složka. Ta určuje sílu světla, které se od objektu rovnoměrně odráží do všech směrů. Označuje se E_d a spočítá se takto:

$$E_d = I_i \cdot C_d \cdot K_d \cdot (\bar{L} \cdot \bar{N})$$

I_i označuje intenzitu světelného zdroje, tedy jakousi obecnou sílu použitého svěla. Stejně jako u okolního světla, C_d označuje barvu difúzní složky. K_d je koeficient difúzního světla. Analogicky ke koeficientu okolního světla, koeficient difúzního světla značí odrazový koeficient materiálu. $\bar{L}$ a $\bar{N}$ jsou vektory. $\bar{L}$ značí vektor dopadu světla a $\bar{N}$ je normálový vektor povrchu. Symbol $\cdot$ zde značí skalární součin vektorů. Difúzní světlo odpovídá matnému povrchu tělesa, je neměnné vůči směru pohledu a jeho použití vytváří dojem trojrozměrných objektů, protože vytváří spojité stínování.

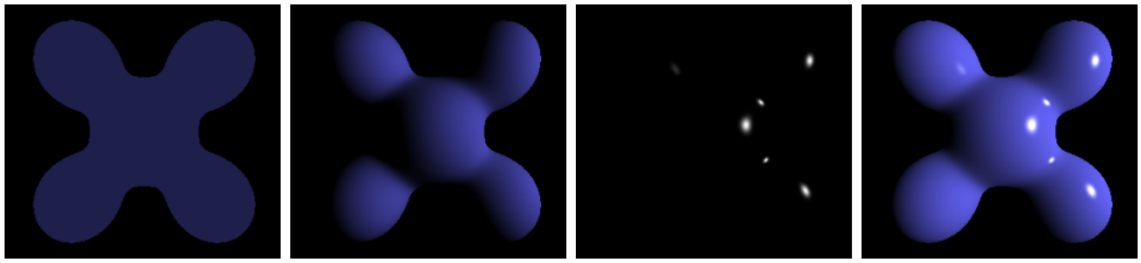
Poslední složkou světla je lesklé světlo. Tato složka udává množství světla odraženého převážně jedním směrem a simuluje tak odlesky. Označuje se E_s a spočítá se podle vztahu:

$$E_s = I_i \cdot C_s \cdot K_s \cdot (\bar{V} \cdot \bar{R})^n$$

C_s určuje barvu odraženého leského světla, K_s pak koeficient lesklého odrazu a podobně jako u předchozích složek nabývá hodnot od 0 do 1. $\bar{V}$ je vektor směřující k pozorovateli a $\bar{R}$ je vektor světla odraženého od povrchu; mezi nimi je opět skalární součin. Poslední proměnnou je n. Ta bývá někdy nazývána Phongův exponent a určuje míru lesklosti. Teoreticky může nabývat hodnot v rozsahu $(0; \infty)$, prakticky se ale nejčastěji používá v rozsahu $(5, 500)$.

Výsledná intenzita světla E se pak spočítá podle vztahu $E = E_a + E_d + E_s$, jak také ukazuje obrázek 2.14.

Phongův osvětlovací model není zcela fyzikálně věrný, porušuje např. symetrii, přesto však podává vysoce kvalitní výsledky. Phongův osvětlovací model není součástí OpenGL a pro jeho implementaci v OpenGL se používají shadery, konkrétně fragment shader. Tato technika stínování, stejně jako žádná jiná technika stínování, neřeší generování stínů, i když by se tomu tak mohlo podle názvu zdát. Tento model dokáže s jemnými úpravami uvedených vzorců pracovat s více světelnými zdroji. Jednotlivé objekty ve scéně se nijak neovlivňují z hlediska osvětlení a proto nedokáží zobrazovat např. odrazy. K docílení takových efektů je potřeba použití *globálních zobrazovacích metod*, např. *metodu sledování paprsku* (anglicky *ray tracing*). Tyto metody jsou oproti Phongovu stínování pomalejší, ale zato umí např. „implicitně“ zpracovávat stíny.



Obrázek 2.14: Ilustace skládání jednotlivých složek světla u Phongova osvětlovacího modelu. Těmito složkami jsou zleva okolní světlo, difúzní světlo a lesklé světlo. Poslední obrázek vpravo pak ukazuje kombinaci všech složek dohromady (jejich součet). [5]

2.8 Kruhový buffer

Kruhový buffer je jednou se základních datových struktur. Jeho hlavním rysem je pevně omezený počet prvků. Jakmile dojde k plnému obsazení bufferu a přijde požadavek na zápis dalšího prvku, začnou se přepisovat prvky na začátku bufferu. Tato struktura se tedy skutečně chová jako by byly prvky seřazeny v kruhu.

[5] Obrázek převzat z http://tomdalling.com/blog/modern-opengl/07-more-lighting-ambient-specular-attenuation-gamma/ a upraven.

Kapitola 3

Implementace

Tato kapitola se zabývá popisem vlastní práce. Nejprve je v kapitole 3.1 stručně popsána vytvořená aplikace z hlediska vnějších výstupů. V kapitole 3.2 je pak podrobně popsáno fungování aplikace jako celku, včetně detailnějšího vysvětlení způsobu dosažení výsledků. Poslední kapitola 3.3 pojednává o příkladech použití aplikace a možnostech nasazení v praxi.

3.1 Popis základní koncepce

Cílem práce bylo vytvořit aplikaci, která zobrazuje jeskyni. Tato jeskyně je generována pomocí speciálního algoritmu. V jeskyni je možné libovolně pohybovat kamerou do všech směrů. Jeskyně je generována procedurálně. Jsou použity takové algoritmy, že jeskyně působí jako nekonečná a přitom se žádná její část téměř nikdy neopakuje. Též textury jsou generovány procedurálně a navozují tedy dojem, že se nikde neopakují. Jeskyně je velmi členitá a obsahuje síť chodeb spojených komíny. Stěny chodeb vytváří dojem hrubého kamene. Jeskyně obsahuje i „smyčky“, ve kterých je možné se pohybovat stále dokola.

V okamžiku spuštění aplikace dojde k vygenerování jeskyně o určité omezené rozloze. Při pohybu s kamerou se pak jeskyně dopočítává právě tam, kde se kamera blíží okraji jeskyně. Algoritmus pracuje takovým způsobem, že při opuštění některé lokality a její opětovné návštěvě se tato lokalita vygeneruje přesně stejně jako při první návštěvě, jeskyně tedy působí spojitě.

3.2 Popis fungování

V této kapitole bude popsáno, jakým přesně způsobem probíhá tvorba jeskyně. Základní postup lze rozdělit do těchto kroků:

1. Vygenerování šumu.
2. Prahování tohoto šumu.
3. Vytvoření polygonálního modelu.
4. Otexturování a zobrazení.

3.2.1 Získání šumu

V této podkapitole bude podrobně popsán kód aplikace. Bude přesně vysvětleno, jak aplikace funguje, včetně toho, jaké funkce se volají a co se nastavuje kterým makrem.

Pro generování šumu je použita metoda Simplex noise, popsaná v kapitole 2.4.2. Metoda je implementována v souboru `simplexNoise.cpp`. Samotná funkce pro výpočet šumu se jmenuje `raw_noise_3d`. Jak již název napovídá, jedná se o třídimenzionální variantu této metody. Funkce očekává tři argumenty typu float, které jsou pojmenovány `x`, `y` a `z` a udávají souřadnice zkoumaného bodu. Funkce vrací hodnotu opět typu float. Tato hodnota je vždy v rozsahu $(-1; 1)$ a udává intenzitu šumu v daném bodě. Tuto funkci lze tedy použít pro zjištění intenzity šumu v jednom bodě v prostoru. Tato funkce byla převzata z [2]. Ve stejném souboru je též implementována funkce `fastfloor`, sloužící k rychlému zaokrouhlování směrem dolů na nejbližší celé číslo a funkce `dot` sloužící pro výpočet skalárního součinu. Funkce `dot` je zde přítomna ve třech variantách, pro dvou, tří a čtyřprvkové vektory. Obě tyto funkce byly také převzaty. Poslední funkcí v souboru je funkce `myraw_noise_3d`. Toto je obalovací funkce k `raw_noise_3d` a slouží ke změně frekvence výsledného použitého šumu. Její tělo ukazuje výpis 3.1.

```
return raw_noise_3d(x / SCALE_FACTOR, y / SCALE_FACTOR, z /
    SCALE_FACTOR);
```

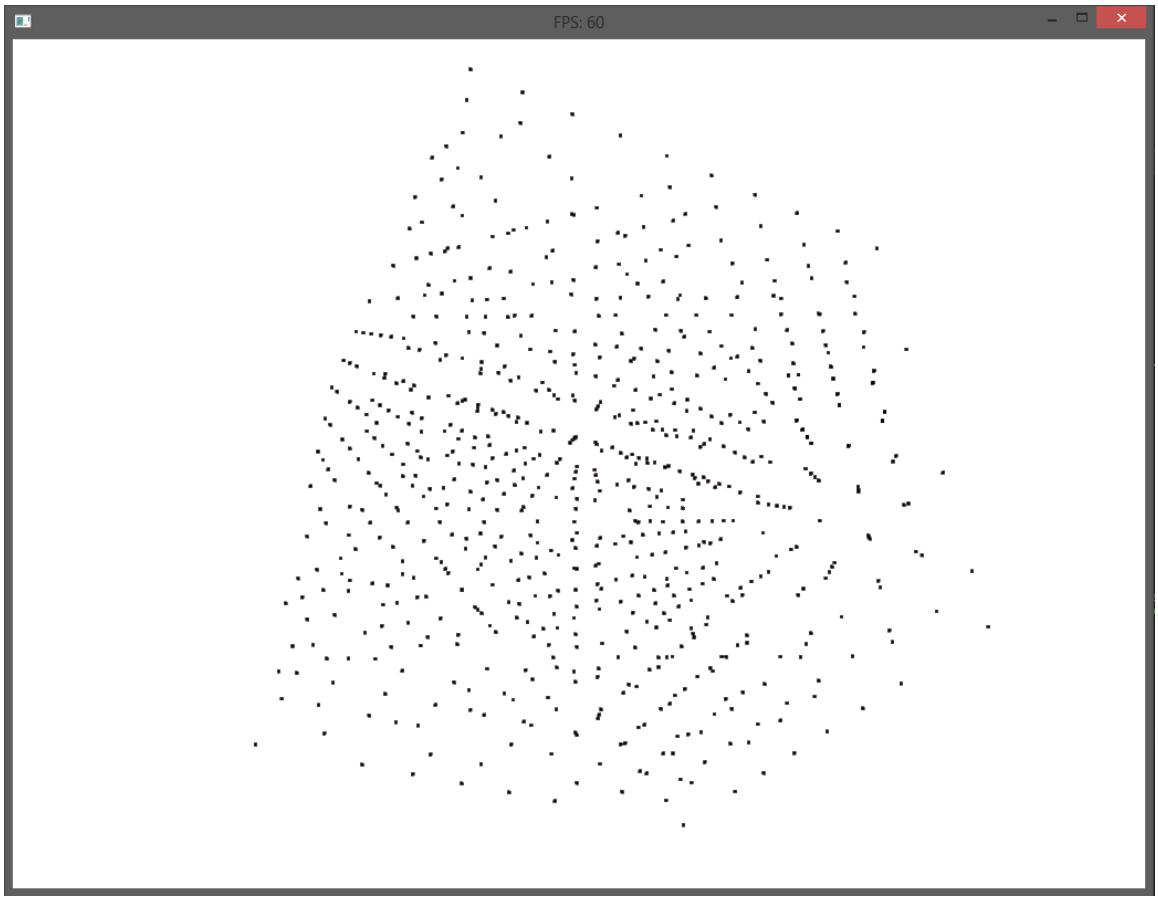
Výpis 3.1: Tělo funkce myraw_noise_3d.

`SCALE_FACTOR` je makro definované v souboru `simplexNoise.hpp` a udává „zvětšení“ šumu. Funkce `raw_noise_3d` se s výjimkou funkce `myraw_noise_3d` nikde nevolá přímo, vždy se tedy používá zvětšená verze. Makro `SCALE_FACTOR` je v základním nastavení aplikace definováno na hodnotu 10. To znamená, že všechny části jeskyně jsou 10x zvětšeny oproti nepoužití obalovací funkce. Tělo funkce `raw_noise_3d` není předmětem této práce a nebude tedy blíže popisováno.

Základní myšlenkou tvorby jeskyně je vygenerování třírozměrné mřížky šumu a následné vytvoření polygonálního modelu podle intenzity šumu. V aplikaci má tato mřížka tvar krychle a délka její strany je určena makrem `BIG_CUBE_SIZE` definovaným v souboru `marchingTetrahedrons.hpp`. V odevzdané aplikaci má toto makro hodnotu 30 a při spuštění aplikace tedy dojde k vygenerování šumu o rozměrech 30x30x30 jednotek. Tuto hodnotu však lze upravit v závislosti na použitém hardware. Na výkonnějších strojích je možno tuto hodnotu zvýšit a docílit tak delší viditelnosti, naopak na těch méně výkonných je vhodné tuto hodnotu snížit. Neadekvátně vysoká hodnota `BIG_CUBE_SIZE` může způsobit pokles FPS pod přijatelnou úroveň. Aby aplikace mohla správně fungovat, je třeba spolu s hodnotou `BIG_CUBE_SIZE` měnit i hodnotu `MAX_VERTICES` v souboru `endlessCave.hpp`. Význam hodnoty `MAX_VERTICES` bude podrobněji popsán v kapitole 3.2.6.

Než je možné začít podle tohoto šumu generovat síť polygonů, je třeba určit, jakým způsobem budou data šumu reprezentovat typ materiálu. Jak již bylo uvedeno výše, funkce pro Simplex noise vrací hodnoty v rozsahu $(-1; 1)$. Bylo zvoleno, že místa s touto intenzitou menší než 0 budou reprezentovat skálu a naopak body s intenzitou větší než 0 budou reprezentovat vzduch. Dochází zde tedy k prahování šumového signálu. Hranice jeskyně (místa přechodu skály a vzduchu, tedy to, co se bude vykreslovat), by tedy měly být umístěny přesně v těch bodech, kde je intenzita šumu rovna nule. Toto nastavení lze změnit úpravou hodnoty makra `TRESHOLD` v souboru `marchingTetrahedrons.hpp` a teoreticky tak lze změnit poměr vzduchu a skály ve výsledné scéně a vytvořit tak např. jeskyně s užšími chodbami. Autor práce však nedoporučuje tuto hodnotu měnit, protože by pak nemusel

správně fungovat výpočet normál a ani Marching tetrahedrons. Mřížku šumu po provedení prahování ukazuje obrázek 3.1. Následující obrázek 3.2 pak obsahuje výpis těchto bodů.

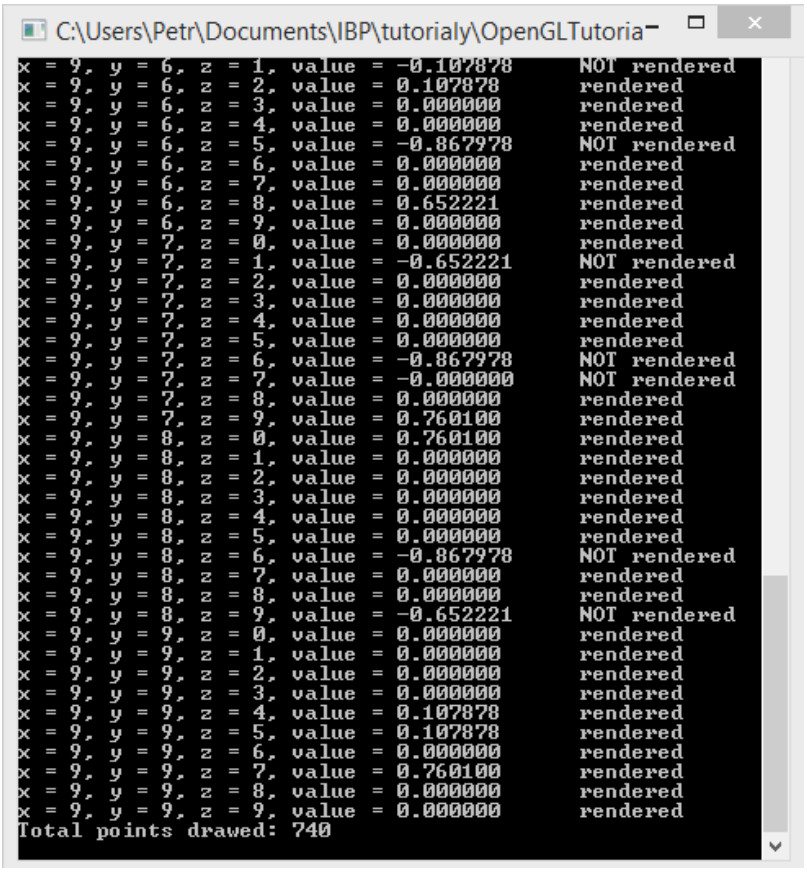


Obrázek 3.1: Šikmý pohled na mřížku bodů získanou z šumu. Vykresleny jsou pouze body s intenzitou šumu vyšší nebo rovnou nule, tedy body, které budou v budoucnu tvořit skálu. Na obrázku je mřížka o rozměrech 10x10x10. Na obrázku 3.2 je pak vidět výpis jednotlivých bodů, spolu s jejich intenzitou a informací o tom, zda byl tento bod vykreslen.

Pro provedení prahování je možné začít s vytvářením polygonálního modelu. K tomu byla nejprve zvolena metoda Marching cubes (popsána v 2.5.1). Během následného testování se však ukázalo, že metoda není příliš vhodná kvůli problémům s nejednoznačností a jejich obtížným řešením. Nakonec byla tedy ještě implementována metoda Marching tetrahedrons, která již byla použita ve výsledné aplikaci.

3.2.2 Implementace Marching cubes

Ještě před začátkem řešení samotného Marching cubes bylo potřeba zjistit si data v jednotlivých vrcholech krychle. V aplikaci bylo vytvořeno trojrozměrné pole `noiseData` o velikosti `BIG_CUBE_SIZE` v každém rozměru, které bylo před zavoláním funkce pro Marching cubes naplněno. To se provedlo velmi jednoduše pomocí tří zanořených cyklů, kdy v těle nejvnitřnějšího cyklu se volala funkce `myraw_noise_3d` pro zjištění hodnoty šumu. Data pro Marching cubes byla tedy připravena předem. Toto řešení má velkou výhodu ve výpočetní



Obrázek 3.2: Částečný výpis bodů k obrázku 3.1. Jednotlivé sloupce výpisu (oddělené čárku) obsahují souřadnice v osách x, y, z a dále intenzitu šumu v tomto bodě (hodnota value). Poslední sloupec vpravo pak nese informaci o tom, zda byl daný bod vykreslen.

náročnosti — hodnota šumu v každém bodě mřížky je vždy počítána pouze jednou (teď není uvažováno počítání hodnoty šumu pro určení normály). Nevýhodou tohoto řešení je ale vyšší paměťová náročnost.

Imaginární krychle algoritmu Marching cubes může podle hodnot ve vrcholech nabývat až 256 stavů. Jak již bylo popsáno v teoretické části, u každého vrcholu se rozlišují pouze dva stavy, v tomto případě je to buď skála nebo vzduch. Jedním z řešených problémů bylo, jak mezi stavy rychle rozlišit — tedy jak rychle zjistit, v jakém stavu krychle je. Nejtriviálnější možností jak to udělat by bylo, vytvořit sadu celkem 256 podmínek, tedy jednu podmínku pro každý stav. Přitom každá z těchto podmínek by ještě musela být složena z osmi dílčích podmínek, které by musely platit současně. Takový kód by tedy byl velice dlouhý, nepřehledný, náchylný k chybám a také poměrně pomalý.

Proto byla použita metoda ohodnocování vrcholů. Vrcholy byly očíslovány od 0 do 7 a byly jim přiřazeny váhy mocnin dvou, tedy váhy od 1 do 128. Při vyhodnocování krychle pak byly sečteny váhy těch vrcholů, které byly součástí skály. Výsledkem součtu pak byla hodnota od 0 do 255, která značila celkovou váhu krychle a přesně určovala jednu konfiguraci Marching cubes. Tímto způsobem se tedy podařilo vyhnout složité sérii podmínek, které bylo možno nahradit jedním příkazem switch. Tento switch však měl stále tu negativní vlastnost, že musel obsahovat všech 256 případů. Unikátních případů sice je

256, ale také si je lze představit jako pouze 14 konfigurací, které jsou pouze různě převrácené nebo otočené. Aby se vyhnulo duplikaci dat (tedy situaci, kde by byly některé konfigurace uloženy vícekrát) a také kvůli předejití chybám, bylo formou vektorů typu *int* uloženo pouze těchto 14 konfigurací. Jednotlivé konfigurace byly uloženy jako sekvence hran, které je třeba spojit. Pro pojmenování hran a os byly předem vytvořeny makra. Aby bylo možné dopočítat ostatní konfigurace, byly implementovány funkce `std::vector<int> flipByAxe(int axe, std::vector<int> input)` a `std::vector<int> rotateByAxe(int axe, std::vector<int> input)`, které mají na vstupu jednu konfiguraci (parametr `input`) a osu (parametr `axe`). Funkce poté vracely transformované konfigurace. Z důvodu jednoduchosti byla pro rotace implementována pouze jedna funkce. Tato funkce otáčela vždy o 90°ve směru doprava při pohledu zepředu. Při potřebě otočení doleva bylo tedy třeba třikrát po sobě použít funkci pro rotaci doprava. Toto řešení bylo sice výpočetně pomalejší, ale kód byl jednodušší a pro testování toto řešení postačovalo. Díky tomuto řešení byl kód algoritmu sice dlouhý, ale i přesto poměrně přehledný a hlavně v něm šly snadno odhalit chyby. Výpis 3.2 ukazuje kód jedné konfigurace.

```
case 38:
    figure = figure5;
    figure = rotateByAxe(Z, figure);
    figure = flipByAxe(X, figure);
    figure = rotateByAxe(X, figure);
    pushFigure(figure, vertices, x, y, z, nubmerOfVertices);
    break;
```

Výpis 3.2: Kód zpracovávající konfiguraci č. 38.

Uvedený kód pracuje takto: Nejprve se načte jedna z 14 konfigurací, v tomto případě je to konfigurace číslo 5. Tato konfigurace je rotována podle osy Z, následně převrácena podél osy X a poté ještě rotována podél osy X. Nakonec už je konfigurace pouze uložena do vektoru vrcholů. Přirozeně, tímto způsobem byly zpracovány všechny krychle z celého objemu.

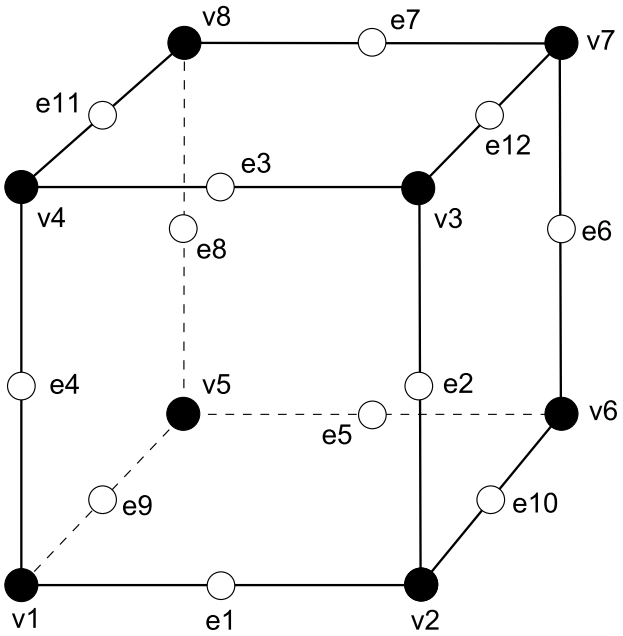
Obrázek 3.3 ukazuje schéma imaginární krychle se zvoleným označením vrcholů a hran. Celý kód implementovaného Marching cubes je uložen na přiloženém DVD v souborech `marchingCubes.cpp` a `marchingCubes.hpp`.

Obrázek 3.4 poté ukazuje testovací scénu zpracovanou algoritmem Marching cubes. Vstupními daty této scény byla skutečná šumová data z Simplex noise.

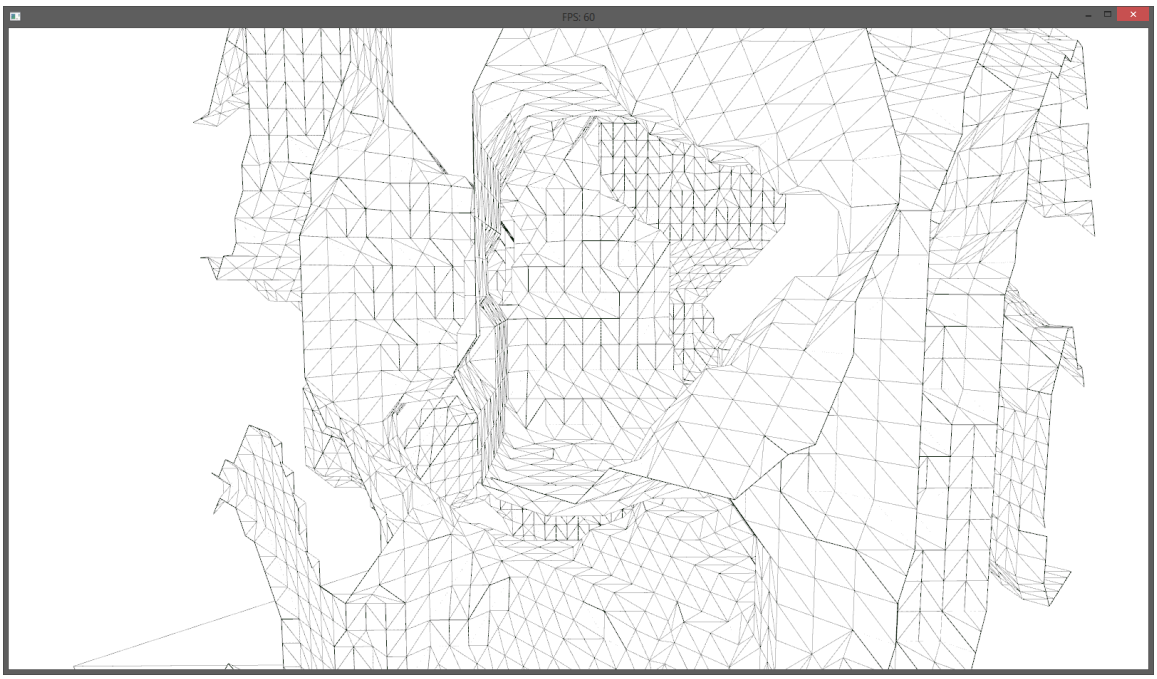
3.2.3 Implementace Marching tetrahedrons

Na rozdíl od Marching cubes, kde bylo zvoleno nejprve si data pro celou zpracovávanou mřížku vypočítat předem a uložit do paměti, u Marching tetrahedrons byl zvolen způsob počítat hodnoty šumu vždy, až když je to potřeba u právě zpracovávané krychle. Toto řešení je výpočetně náročnější, protože intenzity šumu se v některých vrcholech mohou počítat vícekrát. V nejhorším případě je to osmkrát ve zcela vnitřních bodech, v nejlepším případě pak pouze jednou v rohových bodech. Nutno podotknout, že řešení tohoto problému nijak nesouvisí s algoritmy pro tvorbu polygonálního modelu a bylo zvoleno čistě na základě rozhodnutí autora práce.

Algoritmus zpracovává trojrozměrnou mřížku dat o nějakých rozměrech. Obsahuje tedy tři vnořené cykly pro každý rozměr. Dalším krokem je výše popsané zjištění intenzit šumu ve vrcholech krychle a uložení do pole. Poté následuje v pořadí již čtvrtý vnořený cyklus. Tento cyklus však již iteruje pouze šestkrát a to přes tetrahedrony v krychli. Další postup



Obrázek 3.3: Schéma imaginární krychle algoritmu Marching cubes s vyznačeným označením vrcholů a hran. Značení odpovídá značení v kódu.

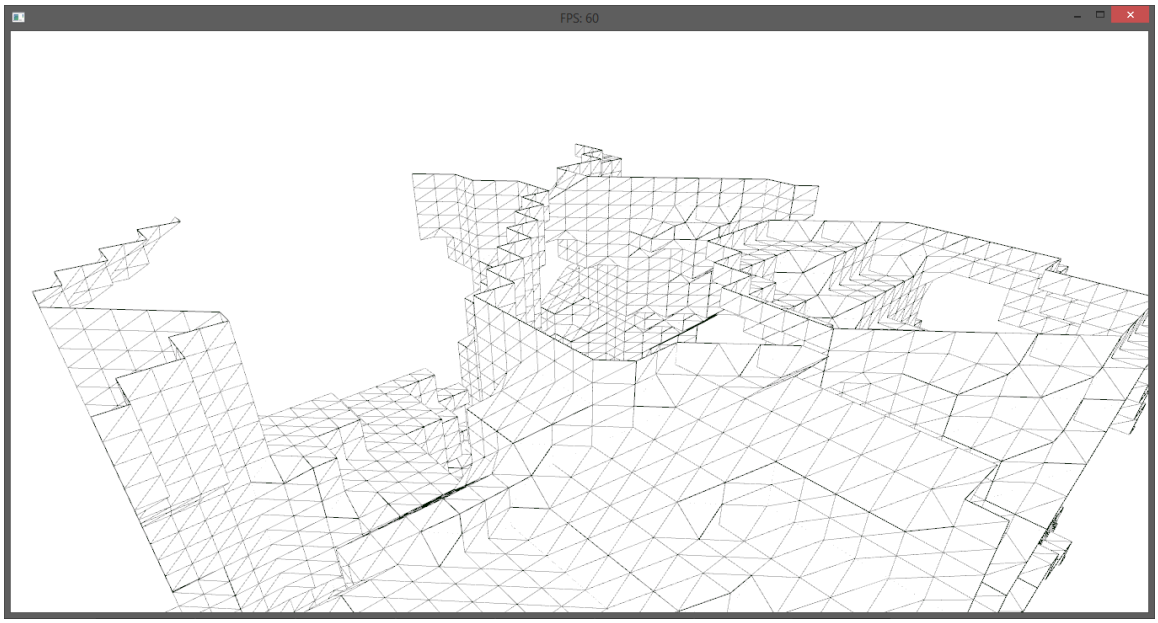


Obrázek 3.4: Šumová data konvertovaná do hraniční reprezentace algoritmem Marching cubes.

připomíná jakési „marching tetrahedron uvnitř cube“. Je zde pole o délce čtyři, které obsahuje hodnoty ve vrcholech tetrahedronu. Toto pole je třeba naplnit. K tomu je využit

další vnořený cyklus a speciální tabulka `cube2tetra` reprezentovaná dvourozměrným polem. Souřadnicemi do této tabulky jsou 1. číslo tetrahedronu v krychli a 2. číslo vrcholu tetrahedronu. Tímto způsobem se pole `vt` obsahující intenzity ve vrcholech tetrahedronu naplní správnými daty z intenzit šumu ve vrcholech krychle v závislosti na tom, který tetrahedron se právě zpracovává.

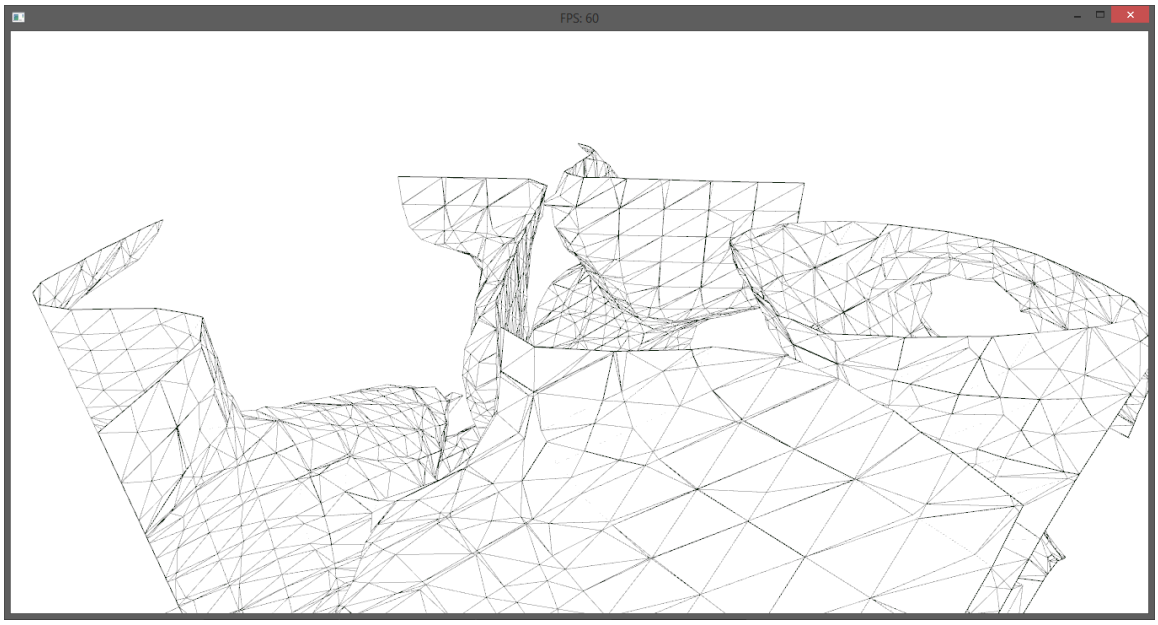
Vrcholům tetrahedronu jsou, podobně jako u Marching cubes přiřazeny váhy. Ty teď ale nabývají pouze hodnot 1, 2, 4 a 8. Je spočítán součet vah v kladných vrcholech a tím váha tetrahedronu. Pokud je zjištěno, že tetrahedron je celý ve skále nebo celý ve vzduchu — bude se vykreslovat nula trojúhelníků, je možno přejít na novou iteraci a začít tak řešit další tetrahedron. Dále je načtena odpovídající konfigurace tetrahedronu z vektoru `figures`. Tento vektor by sice měl teoreticky obsahovat 16 prvků (2^4, 4 vrcholy tetrahedronu a 2 možné stavy) ale stavy „plná skála“ a „plný vzduch“ je možno vynechat. Navíc, pokud je např. ve skále jeden vrchol, je situace zcela stejná, jako když jsou ve skále tři vrcholy — vždy se vykreslí jeden trojúhelník na stejnou pozici, proto lze počet prvků pole `figures` ještě snížit na polovinu — hodnotu sedm. Předposledním krokem je převod vypočtené konfigurace ze souřadnic tetrahedronu do souřadnic kostky. K tomu slouží převodní tabulka `tetra2cube`. Nakonec už jsou jen výsledné vrcholy vypočtených trojúhelníků vloženy do vektoru vrcholů. Obrázek 3.5 ukazuje scénu s daty z šumu zpracovanou metodou Marching tetrahedrons.



Obrázek 3.5: Testovací scéna vytvořená algoritmem Marching tetrahedrons.

Scéna na obrázku 3.5 je i přes poměrně vysokou hustotu trojúhelníků hodně „hranatá“ a obsahuje nežádoucí artefakty způsobené pevně daným rozložením tetrahedronů v krychli. Tyto problémy lze úspěšně řešit pomocí metody posouvání bodů po hraně. Až do tohoto bodu se všechny vrcholy vkládaly do středu hrany, nehledě na úrovně intenzit ve vrcholech. Posouvání bodů po hraně zohledňuje i intenzity ve vrcholech a ne pouze, že jeden je uvnitř a druhý vně. Pokud se má vrchol vložit např. mezi dva body s intenzitami 0,9 a -0,1, bod se vloží v příslušném poměru blíže bodu -0,1. Díky tomu se vrcholy vkládají velmi blízko skutečnému rozhraní skála/vzduch, tedy tam, kde je intenzita rovna nule. To přináší

významně hladší a kvalitnější model. Míra posunutí bodu je počítána jednoduchou funkcí `getShift`, umístěnou v souboru `marchingTetrahedrons.cpp`, kde se nachází i zbytek algoritmu Marching tetrahedrons. Toto vylepšení by bylo možné použít i u Marching cubes, kde však nebylo implementováno, protože by to bylo zbytečné. Scénu zpracovanou s tímto vylepšením zobrazuje obrázek 3.6.



Obrázek 3.6: Stejná scéna jako na obrázku 3.5, nyní vytvořená metodou Marching tetrahedrons s posouváním bodu.

3.2.4 Výpočet normál

V prvních fázích vývoje byla nejprve vygenerována veškerá geometrie, a poté byly pouze z této geometrie počítány normály. Tento postup však nebyl příliš přesný a navíc s sebou nesl problém s orientací normál. Byl problém zajistit, aby normály na jednom povrchu byly orientovány jednotně a stávalo se tak, že výsledný objekt byl „flekatý“ díky náhodné orientaci normál.

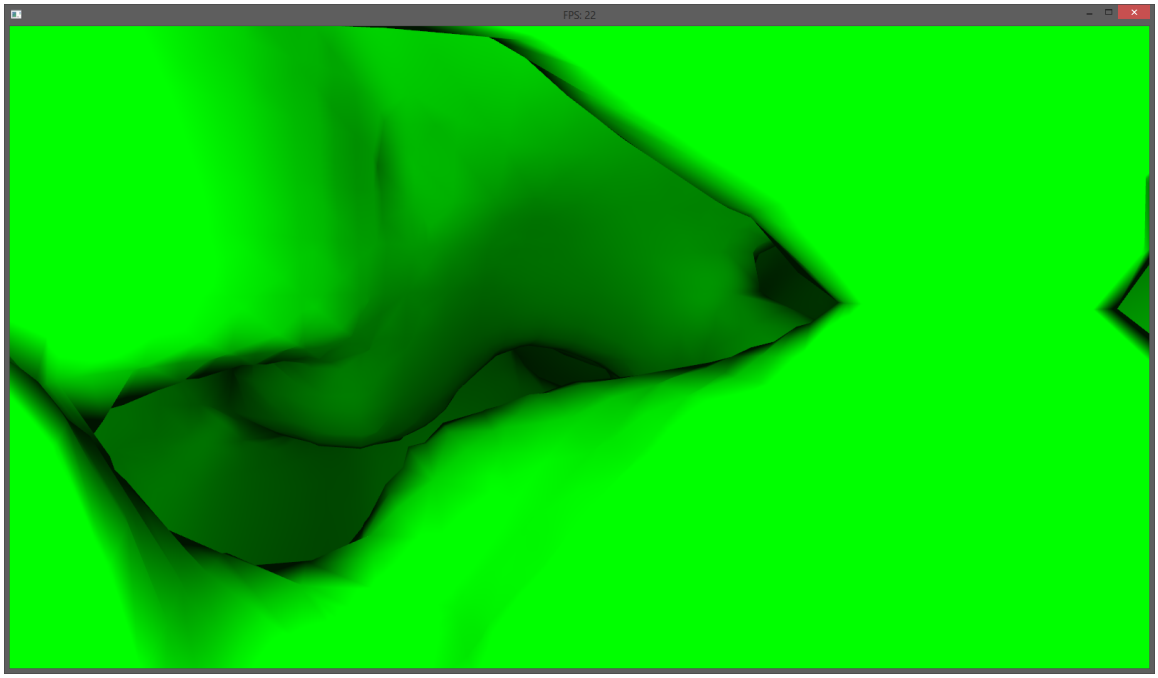
Řešením bylo počítat normály přímo z dat šumu. Pro výpočet normály bylo použito šestiokolí vrcholu. Normála byla počítána pro každý vrchol scény. Normála v prostoru je tříprvkový vektor. Pro výpočet prvního prvku se vzala intenzita šumu v bodě vzdáleném o předem danou hodnotu v kladném směru po ose X, a od této hodnoty se odečetla intenzita šumu v bodě posunutém o stejnou hodnotu po této ose v opačném směru. Tato vzdálenost „styčného bodu“ pro výpočet normály je definována makrem `NORMAL_DISTANCE` v souboru `endlessCave.hpp`. Ve výchozím nastavení se počítá s hodnotou 1.0. Obdobně pro druhý prvek se pracovalo s osou Y a pro třetí prvek s osou Z. Výpis 3.3 demonstruje výpočet jedné normály.

```
normal = glm::vec3(
    myraw_noise_3d(vertices[i].x + NORMAL_DISTANCE, vertices[i].y,
        vertices[i].z) - myraw_noise_3d(vertices[i].x - NORMAL_DISTANCE,
        vertices[i].y, vertices[i].z),
```

```
    myraw_noise_3d(vertices[i].x, vertices[i].y + NORMAL_DISTANCE,
        vertices[i].z) - myraw_noise_3d(vertices[i].x, vertices[i].y -
        NORMAL_DISTANCE, vertices[i].z),
    myraw_noise_3d(vertices[i].x, vertices[i].y, vertices[i].z +
        NORMAL_DISTANCE) - myraw_noise_3d(vertices[i].x, vertices[i].y,
        vertices[i].z - NORMAL_DISTANCE)
);
```

Výpis 3.3: Kód pro výpočet normály. Používá se šestiokolí vrcholu a data jsou získávána přímo z šumu.

Jak je z kódu patrné, pro výpočet jedné normály se šestkrát počítá hodnota šumu. Toto řešení bylo zvoleno z důvodu maximální jednoduchosti, ale obsahuje prostor pro možné vylepšení a optimalizaci, protože intenzita šumu na stejných souřadnicích se zde jistě počítá více než jednou. Navíc, pokud by bylo třeba dosáhnout kvalitnějších výsledků, bylo by možné pracovat i s 26-okolím bodu. Toto řešení by však bylo neúměrně pomalejší, a proto bylo v aplikaci použito pouze šestiokolí. Scénu zobrazující jeskyni vybavenou normálami ukazuje obrázek 3.7. Tato scéna zatím neobsahuje žádné textury, pouze je zde nastavena barva povrchu. Pro zobrazení bylo použito Gouraudovo stínování, takže povrch působí hladce, i když je ve skutečnosti hranatý.



Obrázek 3.7: Ukázka jeskyně po dodělání výpočtu normál.

3.2.5 Shadery

Aplikace využívá vertex shader a fragment shader. První z jmenovaných je velmi jednoduchý a poměrně nezajímavý. Kromě pozice vertexů se zde počítají také normály, pohledový vektor a vektor světla, které jde od vertexu ke světlu. K výpočtu těchto vektorů se používá násobení různými maticemi, např. modelovou maticí nebo pohledovou maticí.

Zajímavější je druhý — fragment shader. Ten kromě „obvyklých“ výpočtů obsahuje také další implementaci Perlinova šumu a Simplex noise. Ty jsou zde použity např. pro tvorbu textur. Jeskyně je texturována 3D texturou vytvářenou pomocí Perlinova šumu. Pro tento účel byl testován i Simplex noise, ale ten nepřinášel kvalitní výsledky. Funkce pro výpočet šumu potřebuje mít na vstupu trojrozměrnou souřadnici. K tomu byla použita interpolovaná data o pozici z vertex shaderu. Aby bylo dosaženo věrohodnějších výsledků, jsou tyto souřadnice před použitím ještě násobeny proměnnou `scaleFactor`. Její hodnota se mění na základě „ypsilonové“ souřadnice, tedy na základě nadmořské výšky. Tím je docíleno efektu, že textura je jemnější (tzn. navozuje dojem složení z menších kamenů) v nižších polohách a naopak hrubší v vyšších polohách.

Z šumu je tedy pro každý fragment získána jedna hodnota v pohyblivé řádové čárce. Tuto hodnotu by bylo možné přímo použít jako difúzní barvu materiálu. V aplikaci však bylo implementováno ještě jedno vylepšení. Pro získání difúzní barvy materiálu se používá jednorozměrná textura a získaná intenzita šumu slouží pouze jako ukazatel do této textury; respektive intenzita šumu určuje, z jakého místa jednorozměrné textury se bude čerpat odstín. Jednorozměrná textura je klasický obrázek o rozměrech 256x1 bodů uložený v souboru `tex1.png`. Tato textura již není generovaná procedurálně ale byla předem vytvořena v grafickém editoru. Výpis 3.4 obsahuje kód právě popsaného řešení.

```
1        scaleFactor = 2 + (Position_worldspace.y / 100);
2        float shade = noise(Position_worldspace * scaleFactor);
3        shade = 0.5 + 0.5 * shade;
4        vec3 MaterialDiffuseColor = texture(tex1dSampler, shade).rgb;
```

Výpis 3.4: Kód získání difúzní barvy materiálu ve fragment shaderu.

Na řádku 1 probíhá výpočet hodnoty `scaleFactor`, na řádku 2 získání odstínu z šumu na základě polohy a `scaleFactoru`, na řádku 3 převod odstínu z rozsahu (-1, 1) do rozsahu (0, 1) a řádek 4 obsahuje použití odstínu jako ukazatele do 1D textury a získání barvy. Konstanty 2 a 100 byly určeny jako vhodné náhodným testováním. Pro načtení textury se používá knihovna `stblib` a její načtení probíhá v souboru `endlessCave.cpp`. Toto použití jednorozměrné textury přináší tyto výhody:

1. Výsledná scéna může být barevná. To by za použití pouze hodnoty z šumu šlo těžko.
2. Celkově se zvýšila kvalita a věrohodnost výsledku. Je možné lépe definovat, jak má jeskyně vypadat.

Ve fragment shaderu se také aplikuje Bump mapping. V jeho implementaci lze pozorovat podobnost s výpočtem difúzní barvy materiálu. Jeho výpočet je také založen na Perlinově šumu. Vstupní souřadnice se opět upravují proměnnou `scaleFactor`. Tentokrát však takovým způsobem, aby hrbolatost vytvořená Bump mappingem byla několikanásobně jemnější než obarvení texturou. Tímto způsobem vznikne hodnota nazvaná `bumpFactor`. Touto hodnotou je poté ovlivněna vypočtená normála, která je následně znovu normalizována. Ještě před úpravou normály je ale třeba vytvořit imitaci mechu. Ta se vytvoří tak, že tam, kde je „ypsilonová“ složka normály větší než 0 je tato složka drobně navýšena u difúzní barvy materiálu, a tím se materiál stane v tomto místě více zeleným. V kódu to vypadá takto: `if (n.y > 0) MaterialDiffuseColor.y += (n.y / 40);`. Pracuje se zde s předpokladem, že mech roste na vodorovných površích.

Poslední částí implementovanou ve fragment shaderu je Phongovo stínování. Implementace tohoto algoritmu není předmětem práce, a proto byla jeho implementace převzata z [3],

odkud byla převzata i většinová část vertex shaderu. Stejně tak implementace šumu v GLSL byla převzata z [5]. Proto nejsou tyto části ani nijak podrobně popisovány.

K fragment shaderu je ještě třeba doplnit, jak se v aplikaci pracuje se světly. Ve scéně je pouze jedno všesměrové světlo. Toto světlo se pohybuje spolu s kamerou a má s ním identickou pozici, takže vzdálenější místa se zdají tmavší a osvětlí se až při přiblížení se. Intenzitu tohoto světla lze měnit makrem `LIGHTPOWER` v souboru `FragmentShader.glsl`.

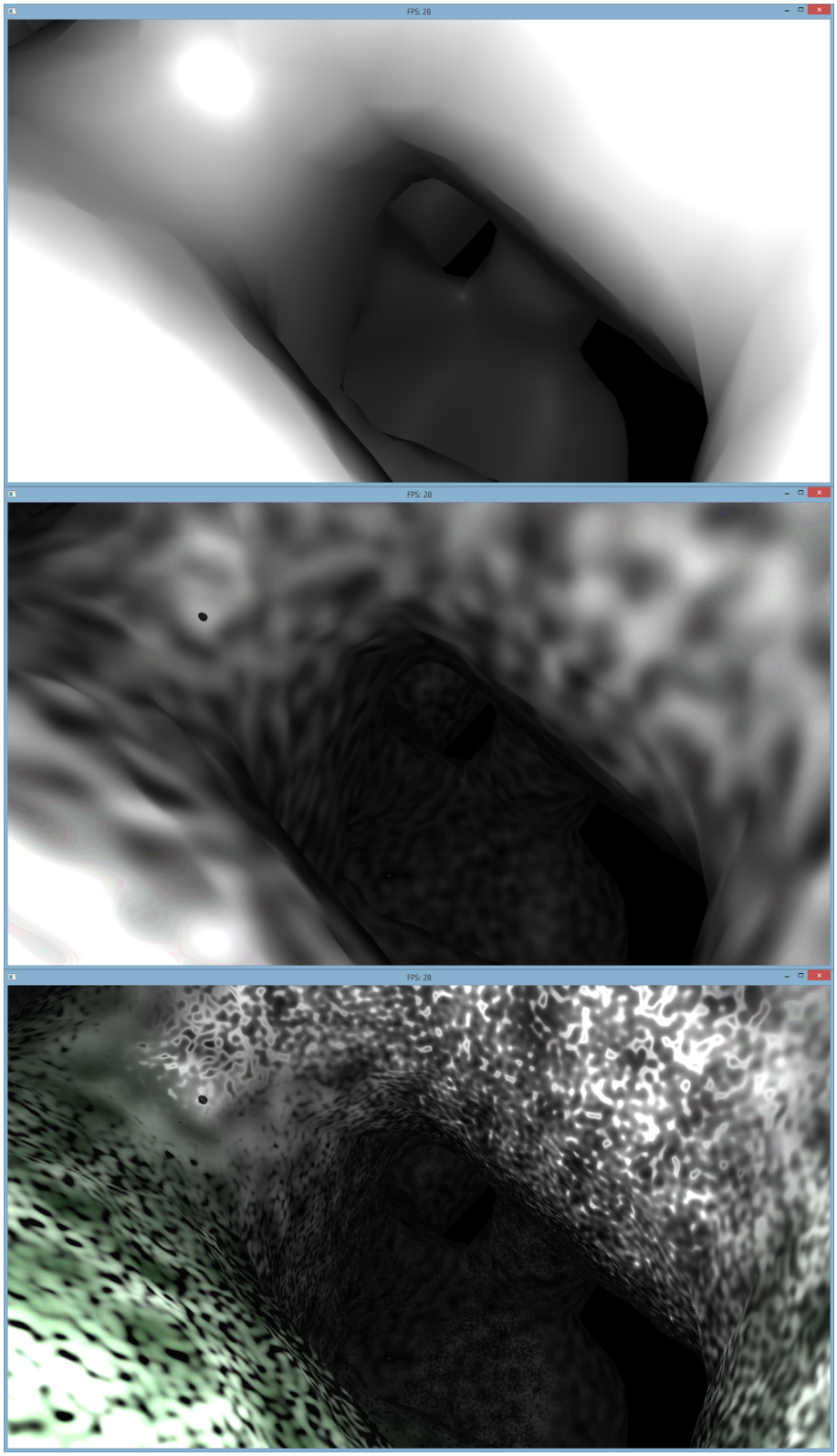
Obrázek 3.8 zobrazuje třikrát stejnou scénu, ale s různým nastavením fragment shaderu. V první třetině obrázku má scéna pouze nastavený odstín difúzní barvy. V prostřední části obrázku je již aplikována trojrozměrná textura. V poslední třetině je přidán ještě bump mapping a efekt mechu. Všechny tyto 3 obrázky byly získány již s Phongovým stínováním.

3.2.6 Zajištění nekonečnosti

Nekonečnost je jedním ze základních požadavků tohoto zadání. V aplikaci existuje vektor `vertices` typu `glm::vec3`, do kterého se ukládají všechny vrcholy, které se později ve vykreslovací smyčce vykreslí. GLM je pouze název knihovny pro pohodlnou práci s vektory a vec3 znamená, že vkládané vektory budou obsahovat 3 prvky — to je logické, protože aplikace pracuje v trojrozměrném prostoru. Podobně je v aplikaci vektor `normals`, do kterého se ukládají normály.

Při spuštění aplikace je zavolána funkce `getCaveData`. Tato funkce vypočítá výseč prvotní jeskyně. Tato výseč bude mít tvar krychle o délce strany definované makrem `BIG_CUBE_SIZE`. Pro snadnější popis bude tato krychle nazývána *velká kostka*. Jak bylo zmíněno výše, hodnotu tohoto makra je možné měnit v souboru `marchingTetrahedrons.hpp`. Funkce ve svém těle volá funkci pro použití Marching tetrahedrons, která zase volá funkci pro výpočet hodnot šumu. Výsledná data z této funkce jsou uložena právě do vektorů `vertices` a `normals`. Funkci `getCaveData` je možné parametrem specifikovat, na jaké souřadnice se má krychle s prvotní jeskyní umístit. Při spuštění aplikace je kamera umístěna do středu této krychle. Poslední věc, kterou tato funkce udělá je, že zaznamená, kde jsou hranice aktuálně vygenerované jeskyně, tedy kde začíná a končí ona zpracovávaná kostka. Tyto hranice byly nazvány *konec vesmíru*, v aplikaci je to proměnná `endOfUniverse`.

V hlavní smyčce aplikace jsou při každém průchodu porovnávány souřadnice kamery s *koncem vesmíru*. Jakmile rozdíl poloviny délky strany *velké kostky* a vzdálenosti kamery od kteréhokoli okraje vesmíru klesne pod danou toleranci, je na patřičnou stranu kostky přidán „plát“ jeskyně. Tato tolerance je definována makrem `TOLERANCE` v souboru `endlessCave.hpp`. Bez této tolerance by mohlo docházet k přidávání plátů náhodně. Plátem se rozumí pravidelný čtyřboký hranol o rozměrech `BIG_CUBE_SIZE`x`BIG_CUBE_SIZE`x1. Právě k tomuto slouží funkce `addCaveSlice`. Ta opět přidává data do vektorů `vertices` a `normals`. Její jediný vstupní parametr určuje, na kterou stranu velké kostky se má plát přidat, tedy např. doleva nebo dolů. Při každém tomto „přidání plátu“ je třeba aktualizovat hranice *konce vesmíru*. Oproti funkci `getCaveData` je zde ještě jeden rozdíl. Protože paměť každého počítače je omezená, nelze data do vektorů přidávat do nekonečna. Z toho důvodu byla přidána opatření zajišťující, že se vektory `vertices` a `normals` chovají jako kruhové buffery. Pokud počet prvků v nich překročí určitý limit, začnou se přepisovat data na začátku těchto vektorů. Tento limit je definován makrem `MAX_VERTICES` v souboru `endlessCave.hpp`. Hodnoty maker `BIG_CUBE_SIZE` a `MAX_VERTICES` spolu musí být v jistém poměru. Pokud by byla hodnota makra `MAX_VERTICES` vzhledem k `BIG_CUBE_SIZE` příliš nízká, budou ve scéně vznikat místa s viditelně chybějícími daty. Pokud by byla naopak příliš vysoká, aplikace by byla zbytečně moc náročná na paměť.



Obrázek 3.8: Ukázka jednotlivých efektů aplikovaných v fragment shaderu.

Mohlo by se zdát, že toto řešení zbytečně generuje velké množství dat, která nejsou vidět, protože jsou za kamerou. Testováno bylo i řešení, kdy byla kamera vždy na kraji *velké kostky* a orientována směrem do kostky. Tato metoda přináší vyšší viditelnost. Její velkou nevýhodou je ale potřeba nového počítání všech viditelných dat, tedy celé *velké kostky* při prudkém otočení kamerou, např. o 180°. Navíc je nutné počítat i data viditelná během samotného otáčení. To dělalo z otáčení kamerou velmi pomalý úkon, kdy náhlý pokles FPS degradoval dojem z aplikace. Použitá metoda má tedy sice poloviční viditelnost, ale otáčení kamerou je naprosto plynulé, protože se během něho nepočítají žádná data a také celková plynulost aplikace je větší, protože maximálně je vždy třeba dopočítat jeden „plát“ a nikdy ne celou *velkou kostku* znova.

3.2.7 Ovládání

Tato kapitola obsahuje popis implementace ovládání aplikace. Samotný manuál, jak aplikaci ovládat se nachází v příloze B.

Ovládání kamery je implementováno v souboru `controls.cpp`. Jedinou složitější funkcí je zde funkce `computeViewMatrix`. Jak již název vypovídá, v těle funkce se počítá pohledová matice. Pohledová matice nese všechny informace o pozici a natočení kamery, které jsou do ní zaneseny na základě změny pozice kurzoru myši a stisknutých kláves. Práci s maticemi zde velmi usnadňuje použitá knihovna GLM a práci se vstupními zařízeními knihovna GLFW. Zajímavé na této funkci je, že vrací aktuální pozici kamery, čehož je později využito k dříve popsanému zajištění nekonečnosti.

Ve funkci jsou nejprve na základě změny pozice myši vypočteny nové úhly natočení kamery — vertikální a horizontální. Z nich jsou poté pomocí goniometrických funkcí vypočteny vektory reprezentující směr pohledu a kolmé vektory „doprava“ a „nahoru“. Poté jsou zaznamenávány stisknuté klávesy a s pomocí nich a těchto vektorů je počítána nová pozice kamery. Základ této funkce byl převzat z [3].

3.3 Příklady použití

Aplikace má využití v širokém spektru případů. Lze ji použít pouze jako prezentaci možností procedurálního generování grafiky, ale také zakomponovat do počítačové hry nebo např. simulátoru. Aplikace může sloužit také jako demonstrace procedurálně generovaných objektů, nebo jako intro do hry. V neposlední řadě by bylo možné ji použít také jako efektivní spořič obrazovky.

3.4 Použité vybavení

Aplikace byla vytvořena a testována na notebooku Asus UL80Jt. Tento stroj byl vybaven procesorem Intel® Core™ i5 430UM, pamětí RAM o kapacitě 4 GB a freknveci 1066 MHz DDR3 a grafickou kartou NVIDIA® GeForce® 310M. Dále byl osazen SSD disk Kingston SSDNow V300.

Aplikace je napsána v jazyce C++, a její shadery v GSLS. Vývoj probíhal pod operačním systémem Windows 8.1 Pro. Dále bylo použito vývojové prostředí Microsoft Visual Studio Professional 2013 a jeho vestavěný překladač.

Kapitola 4

Závěr

Cílem této práce bylo vytvořit aplikaci simulující a zobrazující nekonečnou jeskyni. Tento záměr byl splněn implementací aplikace nacházející se na přiloženém DVD.

Součástí zadání bylo seznámit se s generováním grafiky a knihovnou OpenGL — to bylo splněno prostudováním citované literatury. Dále bylo úkolem popsat metody a algoritmy pro generování volumetrických dat — to bylo uděláno v první polovině této práce. Dalším bodem zadání bylo implementovat aplikaci, která bude zobrazovat nekonečnou jeskyni. Tento bod byl také splněn jak je napsáno v předchozím odstavci. Posledním úkolem bylo zhodnotit dosažené výsledky a vytvořit plakát pro prezentování projektu. Výsledky jsou zhodnoceny v tomto závěru a plakát se nachází na přiloženém DVD.

Při tvorbě aplikace jsem použil metodu Simplex noise pro generování šumu. Tento šum je následně prahován pro rozhodnutí o tom, kde bude skála a kde vzduch. Tímto způsobem jsem získal volumetrická data, tvořící základ jeskyně. Tato data jsou následně konvertována do polygonové sítě metodou Marching tetrahedrons. Na tuto geometrii poté nanáším 3D textury generované taktéž procedurálně, za pomoci fragment shaderu. Pro zlepšení vizuální kvality jsem ještě aplikoval Bump mapping. Pro zobrazení scény využívám Phongova stínování. Efektu nekonečnosti jsem docílil tak, že jeskyně se dynamicky dopočítává v tom směru, kam se pohybuje kamera a naopak tam, odkud se kamera vzdaluje se data jeskyně uvolňují z paměti.

Aplikace uspokojivě simuluje pohyb jeskyní. Jeskyně opravdu působí nekonečně a výsledný obraz vytváří poměrně realistický dojem. Povrch jeskyně imituje hrbolatý kamen, přičemž charakteristiku tohoto povrchu lze nastavovat pomocí popsaných maker.

Tato práce mi dala zkušenosti s manipulací s volumetrickými daty a s implementací grafických algoritmů. Také jsem se naučil lépe pracovat s OpenGL. Technická zpráva mne zase naučila vytvářet vektorové obrázky a prohloubila mou znalost LaTeXu.

V práci by bylo možné pokračovat například přidáním vody do jeskyně. Dále by bylo možné vytvořit generování dalších objektů, jako např. různých krápníků nebo kamenů.

Literatura

[1] Dave Shreiner, B. T. K. O. A. W. G.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Pearson Education, Inc., 1994, iSBN 0-201-63274-8.

[2] Eshelman, E.: battlestar-tux [online]. `https://code.google.com/p/battlestar-tux/source/browse/procedural/simplexnoise.cpp`, 2012-02-11 [cit. 2014-04-23].

[3] Eshelman, E.: Tutorials for modern OpenGL (3.3+). `http://www.opengl-tutorial.org/beginners-tutorials/tutorial-8-basic-shading/`, [cit. 2014-02-20].

[4] G.M. Treece, A. G., R.W. Prager: Regularised marching tetrahedra: improved iso-surface extraction [online]. `ftp://svr-ftp.eng.cam.ac.uk/reports/treece_tr333.pdf`, 1998 [cit. 2014-04-06].

[5] Gustavson, S.: GLSL noise. 2004-12-05 [cit. 2014-03-10].

[6] Gustavson, S.: Simplex noise demystified [online]. `http://staffwww.itn.liu.se/štegu/simplexnoise/simplexnoise.pdf`, 2005-03-22 [cit. 2014-04-13].

[7] Hast, A.: Bump Mapping [online]. `http://www.it.uu.se/edu/course/homepage/grafik1/ht04/Lectures/L09CG2004.pdf`, [cit. 2014-04-12].

[8] Heckbert, P. S.: Survey of texture mapping [online]. `http://www.cs.cmu.edu/p̃h/texsurv.pdf`, [cit. 2014-04-11].

[9] Jakubíková, R.: Perlinovy šumové funkce pro generování texturr: bakalářská práce [online]. `https://dspace.vutbr.cz/xmlui/bitstream/handle/11012/9471/xjakub09.pdf`, 2012 [cit. 2014-04-13].

[10] Jiří Žára, J. S. P. F., Bedřich Beneš: *Moderní počítačová grafika*. Computer Press, 2004, iSBN 80-251-0454-0.

[11] Jones, W.: *Beginning DirectX® 10 Game Programming*. Thomson Course Technology PTR, a division of Thomson Learning Inc, 2008, iSBN 10: 1-59863-361-9, ISBN 13: 978-1-59863-361-0.

[12] Pelikán, J.: Phongův světelný model [online]. `http://cgg.mff.cuni.cz/ pepca/lectures/pdf/pg1-31-phong.pdf`, 1996–2004 [cit. 2014-04-12].

[13] Procházka, D.: Základy OpenGL, Počítačová grafika 2 [online]. https://akela.mendelu.cz/x̌kral01/Statnice/Statnice/_st%E1tnice/grafika/PG2%20-%20primitiva,%20zpracov%E1n%ED/PG2_02_zaklady_opengl.pdf, 2010-09-16 [cit. 2014-04-07].

[14] Randi J. Rost, B. L.-K.: *OpenGL Shading Language Third Edition*. Pearson Education, Inc., 2009, iSBN 10: 0-321-63763-1, ISBN 13:0-321-63763-1.

[15] Richard S. Wright, M. S., Jr.: *OpenGL SuperBible Second Edition*. Waite Group Press, 2000, iSBN 1-57169-164-2.

[16] Timothy S. Newman, H. Y.: A survey of the marching cubes algorithm [online]. http://www.proxyarch.com/util/techpapers/papers/survey%20of%20marching%20cubes.pdf, 2006 [cit. 2014-04-06].

[17] William E. Lorensen, H. E. C.: Marching cubes: A high resolution 3D surface construction algorithm [online]. http://www.cs.virginia.edu/johntran/GLunch/marchingcubes.pdf, 1987 [cit. 2014-04-06].

Příloha A

Obsah DVD

Přiložené DVD obsahuje v kořenovém adresáři 3 složky. Jejich názvy jsou: aplikace, plakat a text. Ve složce aplikace se nachází veškeré zdrojové kódy aplikace, včetně textury a souboru projektu pro Visual Studio. V podsložce marchingCubes je pak kód algoritmu Marching cubes, který byl implementován, ale nebyl nasazen ve výsledné aplikaci. Složka plakat obsahuje plakát, který byl vytvořen pro prezentaci výsledků práce. Poslední adresář text obsahuje technickou zprávu ve formátu pdf a také veškeré zdrojové texty této zprávy.

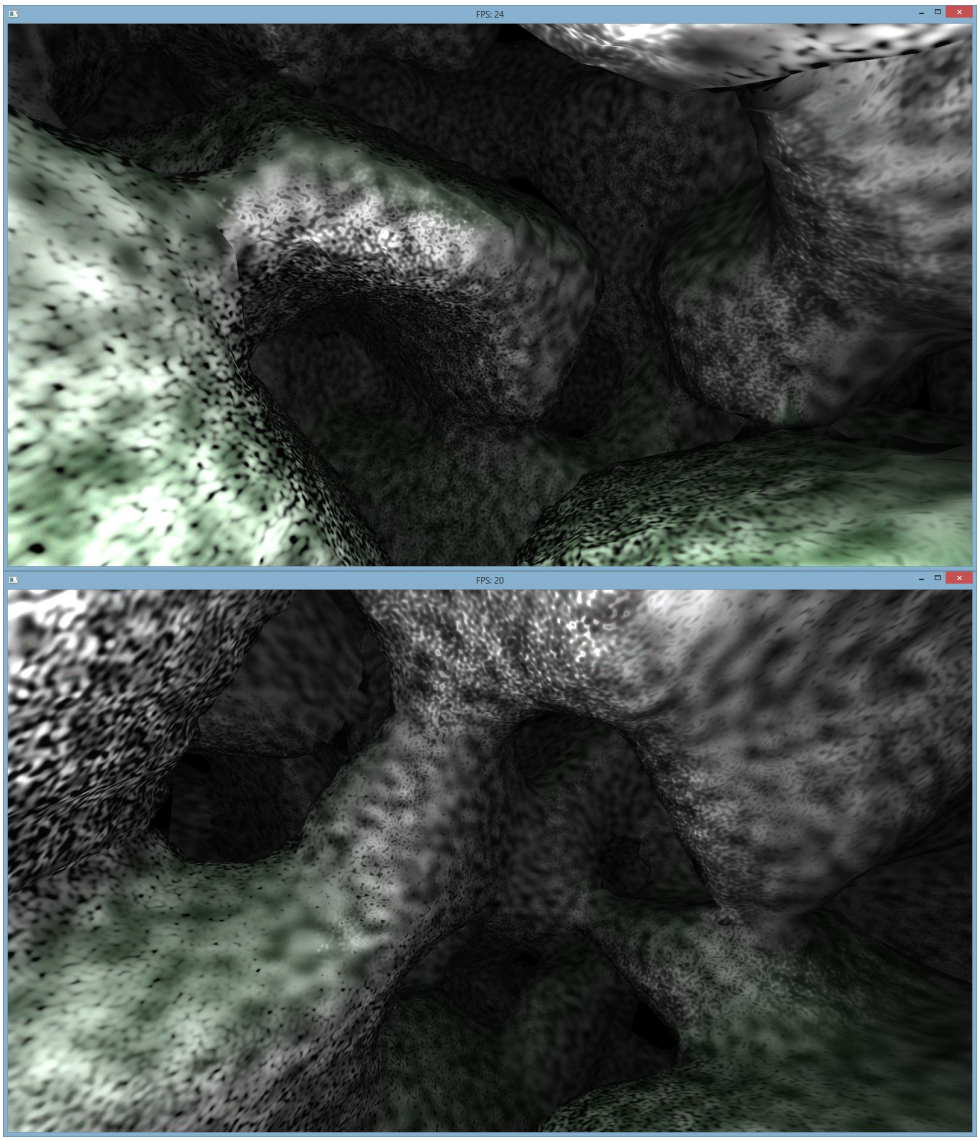
Příloha B

Manuál pro ovládání aplikace

Aplikace obsahuje dva režimy ovládání. Zjednodušený režim je koncipován hlavně pro použití na klávesnicích bez numerického bloku. Pohyb kamery se ovládá kurzorovými šipkami. Šipka ↑ vyvolá posun kamery "vpřed", šipka ↓ pak posun kamery vzad. Klávesy ← a → pak způsobují pohyb kamery do stran. Druhý režim funguje parelelně s prvním a nabízí komfortní ovládání na klávesnicích s numerickým blokem. Klávesy 4 , 8 , 6 a 2 vyvolají posun kamery v tomto pořadí doleva, nahoru, doprava a dolů. Fungují i diagonální směry, které jsou ovládány klávesami 1 , 7 , 9 a 3 . Pro posun kamery vpřed slouží klávesa 5 a pro posun vzad klávesa 0 . Je zde tedy možný posun kamery do všech šesti směrů. Posun kamery nezávisí na natočení souřadného systému, ale na aktuálním natočení kamery. Kamerou je možné otáčet přirozeně v obou směrech pohybem myši.

Příloha C

Screenshoty aplikace



Obrázek C.1: Screenshoty aplikace