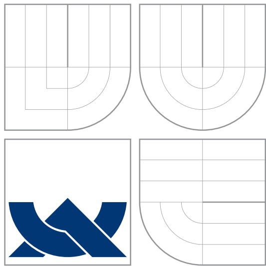
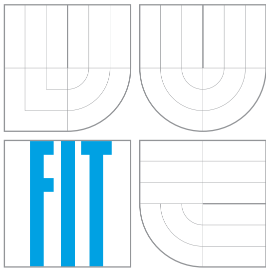


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

GENEROVÁNÍ VOXELOVÝCH SCÉN
GENERATING OF VOXEL SCENES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ VENHODA
AUTHOR

VEDOUCÍ PRÁCE — Ing. TOMÁŠ MILET
SUPERVISOR

BRNO 2016

Zadání bakalářské práce/18826/2015/xvenho02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií   Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Venhoda Lukáš**
Obor: Informační technologie
Téma: **Generování voxelových scén**
Generating of Voxel Scenes
Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se s knihovnou OpenGL, procedurálním generováním, vizualizačními technikami a voxelovou reprezentací scén.
2. Vybrané voxelové a vizualizační algoritmy popište.
3. Implementujte aplikaci, která dokáže generovat voxelové scény, umožní jejich editaci a export do souboru.
4. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu. Vytvořte video pro prezentování projektu.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a kostra aplikace

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: **Milet Tomáš, 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

Tato bakalářská práce se zabývá implementací aplikace, generující voxelové scény s možností jejich editace a uložení do souboru. Popisuje generování terénu pomocí Perlinova šumu, převod seznamu voxelů do struktury Sparse Voxel Octree, a metodu editace této struktury s výběrem voxelu pomocí ray picking algoritmu.

Abstract

This batcherlor's thesis deals with the implementation of application, which generates voxel scenes that can be edited, or saved to a file. The work describes generating of terrain by using Perlin noise, converting voxel list to Sparse Voxel Octree structure, and a method of editing of this structure with picking voxels by using ray picking algorithm.

Klíčová slova

OpenGL, Perlinův šum, Sparse Voxel Octree, procedurální generování, compute shadery, ray picking, voxely, C++, SDL, GPUEngine, GLSL.

Keywords

OpenGL, Perlin noise, Sparse Voxel Octree, procedurál generation, compute shaders, ray picking, voxels, C++, SDL, GPUEngine, GLSL.

Citace

VENHODA, Lukáš. *Generování voxelových scén.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Milet Tomáš.

Generování voxelových scén

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Lukáš Venhoda
18. května 2016

Poděkování

Děkuji vedoucímu práce Ing. Tomáši Miletovi za odborné vedení, pomoc při implementaci aplikace a ochotu při konzultacích.

© Lukáš Venhoda, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod 2

2 Procedurální generování 3
- 2.1 Generátor náhodných čísel . . . 4
- 2.2 Perlinův šum . . . 5

3 Voxely 11
- 3.1 Uložení voxelů v paměti . . . 12
- 3.2 Zobrazení voxelů . . . 14

4 Implementace 16
- 4.1 Tvorba terénu . . . 16
- 4.2 Převod na octree . . . 17
- 4.3 Renderování octree . . . 20
- 4.4 Raypicking . . . 21
- 4.5 Editace . . . 23

5 Závěr 25

Literatura 26

Přílohy 27
- Seznam příloh . . . 28

A Obsah CD 29

B Ovládání aplikace 30

Kapitola 1

Úvod

Cílem této práce je vytvoření aplikace, která procedurálně generuje, zobrazuje a umožňuje editovat grafickou scénu složenou z voxelů. Pod scénou si můžeme představit například prostředí obsahující terén a vodní hladinu.

Běžně se terén zobrazuje pomocí polygonální reprezentace, ta však přináší řadu nevýhod, v jejímž čele stojí složitost zobrazení a náročné způsoby editace v reálném čase. Tyto problémy lze z části vyřešit zobrazením terénu pomocí voxelové reprezentace, která narozdíl od reprezentace polygonální umožňuje jednoduchou modifikaci terénu. Tím lze například umožnit jeho destrukci, úpravu za běhu, nebo automatickou tvorbu jeskynních systémů a podobně. Voxely však kladou větší nároky na paměť, je proto nutné použít speciální struktury pro jejich skladování, jako například uniformně rozložené oddíly jménem chunky, nebo Sparse Voxel Octree.

Při zobrazování běžných polygonálních modelů je možné použít grafických karet pro vykreslení trojúhelníků. U voxelů je to však problém, protože jsou definovány jako body ve struktuře, která je obsahuje. Voxely je tedy nutné před zobrazením převést do polygonální reprezentace, například jako krychle. Jinou možností je použít vlastní renderovací mechanismus, například ray casting.

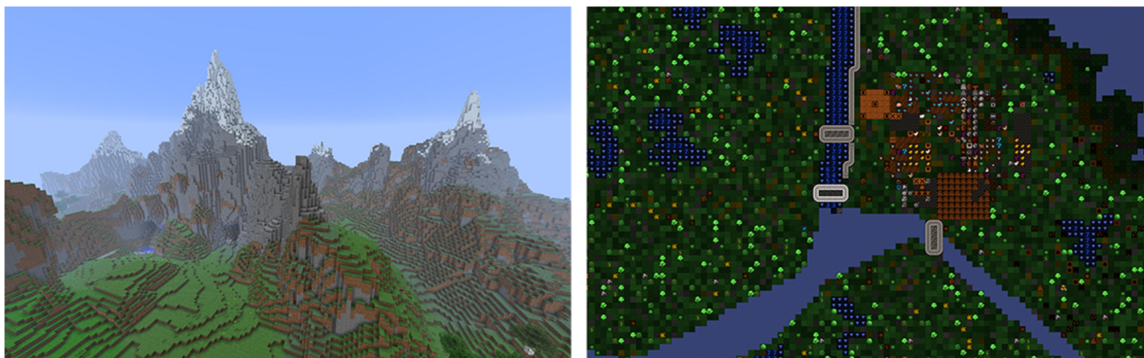
V neposlední řadě je pak nutné tento terén nějakým způsobem vyrobit. Nejčastěji se setkáváme s ručně vymodelovaným prostředím, které je následně uloženo jako síť trojúhelníků. Takový terén je však možné jednoduše vytvořit automaticky. K tomu se využívá metoda procedurálního generování, která umožňuje vytvářet většinu herního obsahu až po spuštění aplikace.

Kapitola 2

Procedurální generování

Procedurálního generování je metoda pro vytváření dat algoritmicky, aniž by byly ručně vytvářeny člověkem. Nejčastěji se využívá pro vytváření obsahu v počítačové grafice, jako jsou textury a objekty, v počítačových hrách se pak používá pro tvorbu různorodého obsahu, včetně samotného prostředí, rostlin, zvukových stop, nebo dokonce i samotného příběhu.

Mezi hlavní výhody procedurálního generování patří nižší paměťová náročnost, větší různorodost obsahu a možnost vytvářet unikátní obsah při každém spuštění aplikace. Problém však nastává při samotném generování, které může být výpočetně náročné a v rámci zmenšení paměťové náročnosti se může stát, že se výsledná aplikace dlouho načítá, pokud se obsah generuje předem, nebo navíc běží pomalu, pokud se obsah generuje za běhu.



Obrázek 2.1: Procedurálně vygenerované prostředí ve hře Minecraft a Dwarf Fortress.

Povětšinu se setkáváme s generováním obsahu ve hrách. Například u technologie SpeedTree [7], která byla použita pro vytváření velkého množství unikátních stromů ve hře The Elder Scrolls IV: Oblivion. Ve hrách jako Minecraft, je pak použita pro vytváření terénu, jako na obrázku 2.1, nebo No Man's Sky, kde generuje celé planety v momentě, když se k nim hráč přiblíží. V extrémních případech lze dokonce použít pro generování celého světa včetně jeho historie, obyvatel, národů, válek a podobně. Mezi takto generované hry například patří Dwarf Fortress.

Lze ji však nalézt také při tvorbě filmů, především pak při vytváření velkých skupin lidí, kde je třeba dojmu, že každá postava opravdu patří do scény, a nevypadá uměle. Takové scény by bylo velmi náročné vytvářet ručně, proto se často generují procedurálně a to včetně umělé inteligence postav a jejich animací. Tato technologie byla použita například pro vytváření armád skřetů ve filmové trilogii Pán Prstenů [6].

2.1 Generátor náhodných čísel

V jádru této techniky stojí generátor náhodných čísel. Takový generátor při každém zavolání vrací náhodnou hodnotu. Na rozdíl od implementace tohoto generátoru v hardware má však softwarový zásadní omezení. Nevrací totiž skutečně náhodné hodnoty, nýbrž hodnoty takzvaně pseudonáhodné. Jedná se o sérii deterministicky spočítaných čísel, které se jako náhodné jeví, ale po nějaké době se začnou opakovat.

Tento problém se často řeší tak, že generátor při inicializaci nastaví počáteční hodnotu, neboli seed, ze které se pak výsledná série počítá, na hodnotu aktuálního času, nejčastěji pak celočíselnou hodnotu počtu uplynulých sekund od počátku Epochy. Výsledná série je pak dostatečně náhodná pro účely procedurálního generování.

Někdy je však žádoucí, aby se při každém spuštění programu, případně při zadání stejných parametrů generátoru vracela identická posloupnost čísel. Tohoto se dosáhne tak, že se zvolí pevný seed, který se mezi spuštěními programu nemění. Výsledný obsah je pak statický, neboli při každém spuštění zůstává stejný. Tímto způsobem se například vytvářejí tzv. dema, což jsou animace vytvářené čistě pomocí procedurálního generování, jejichž binární soubory mají omezenou velikost, zpravidla 64kB. Tento postup lze však využít i ve hrách, kde se ušetří místo na disku, a obsah se vytvoří při spuštění hry, například složitější geometrie, nebo organicky vypadající textury. U takto vytvářených objektů je žádoucí, aby pokaždé vypadaly stejně.

2.1.1 Kongruentní generátor

Jeden z nejpoužívanějších generátorů pseudonáhodných čísel je lineární kongruentní generátor. Jedná se o algoritmus definovaný jako diferenční rovnice

$$X_{n+1} = (aX_n + c) \bmod m \tag{2.1}$$

kde X značí sérii pseudonáhodných čísel, m je modulo, a je násobitel v rozmezí $(0, m)$, c je přírůstek v rozmezí $\langle 0, m)$ a X_0 je počáteční hodnota, neboli seed, v rozmezí $\langle 0, m)$.

V případě 32 bitového počítače se nejčastěji používá $m = 2^{32}$, neboli implicitní maximální hodnota 32bit celočíselné proměnné bez znaménka. V ideálním případě pak takový generátor dokáže vygenerovat m hodnot, neboli po m násobném zavolání funkce se začne série čísel opakovat. Tento jev se nazývá perioda generátoru, a při špatném nastavení může být od této ideální hodnoty dramaticky odlišná. Perioda tohoto generátoru je maximálně m, a to jen za předpokladu, že platí [9]:

1. m a c jsou nesoudělná čísla
2. $a \equiv 1 \pmod{p}$ pro všechny prvky p prvočíselného rozkladu m
3. $a \equiv 1 \pmod{4}$ pokud m je násobek 4

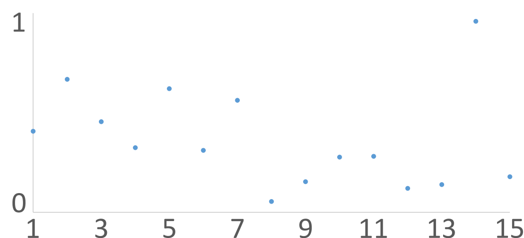
Za předpokladu, že m je mocnina dvou nám postačí, aby c bylo liché číslo a platilo $a \equiv 1 \pmod{4}$.

Takový generátor je příliš jednoduchý pro použití ve vědeckých výpočtech, ale pro účely generování obsahu je naprosto dostačující. Pro zlepšení náhodnosti výsledné série, lze používat jeho variantu, kdy se místo jednoduché rovnice použije polynom, jako například v rovnici 2.2.

$$X_{n+1} = (a \times (a \times (a + X_n) + X_n) + X_n) \bmod m \tag{2.2}$$

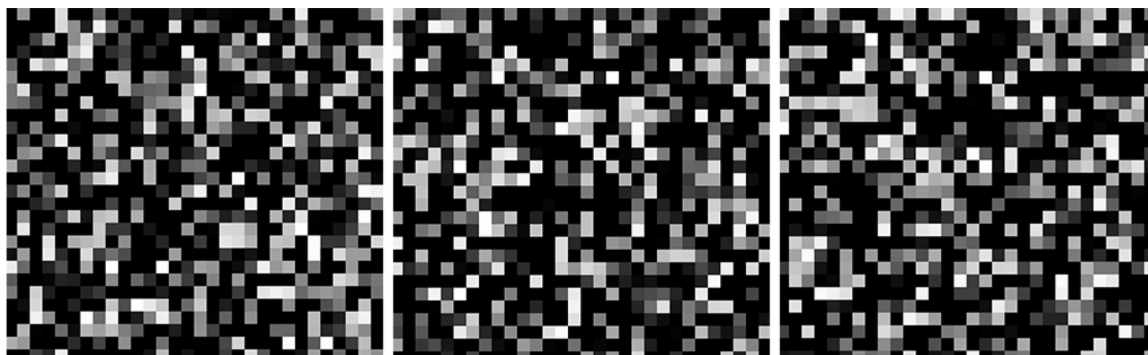
2.1.2 Generátor šumu

Pro vytvoření geometrie, nebo textury je nutné výsledek generátoru převést do rozmezí $\langle 0, 1 \rangle$, popřípadě do rozmezí $\langle -1, 1 \rangle$. Toho lze dosáhnout vydělením výsledku generátoru hodnotou m, respektive vydělením hodnotou $\frac{m}{2}$ a přičtením -1. Takový generátor se pak nazývá generátor šumu. Ten lze následně graficky zobrazit jako graf funkce od 0 do m s hodnotami v rozmezí například $\langle 0, 1 \rangle$. Výsledek lze vidět na obrázku 2.2.



Obrázek 2.2: Graf diskrétní jednodimensionální šumové funkce.

Pro zobrazení šumu ve vyšší dimenzi, například jako dvojrozměrnou texturu, je nutné začít přidávat parametry původnímu generátoru. Toho lze dosáhnout například opětovným voláním generátoru s jinou počáteční hodnotou pro každou souřadnici v textuře. Výsledek předešlého volání se předá jako vstup pro každé následující volání a konečný výsledek se pak převede do rozmezí $\langle 0, 1 \rangle$. Na tuto hodnotu se následně nastaví jeden, či více barevných kanálů v textuře, čímž získáme klasický digitální šum o frekvenci 1, neboli na každý jeden pixel patří jedna hodnota šumu. Výsledek tohoto postupu lze vidět na obrázku 2.3



Obrázek 2.3: Příklad různých dvourozměrných digitálních šumů.

Pro vygenerování přirozeně vypadajícího šumu tento postup nestačí, jelikož výsledek vypadá strojově vytvořen, a nehodí se prakticky k žádné technice procedurálního generování. Lze jej však použít jako základ pro další krok k vytvoření organicky vypadajícího šumu.

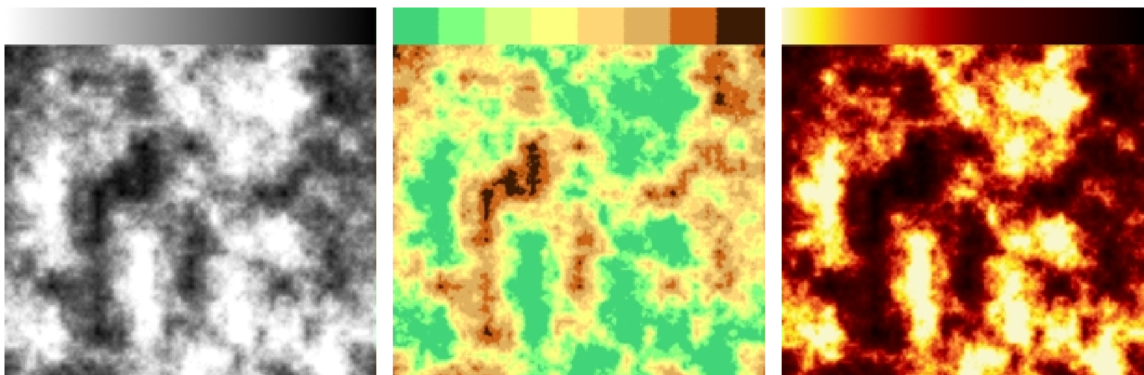
2.2 Perlinův šum

Klasický digitální šum má velmi ostré hrany a nehodí se tedy pro tvorbu organicky vypadajícího prostředí. Pro tyto účely byl vytvořen tzv. Perlinův šum [10]. Jedná se v základu o digitální interpolovaný šum, který je generován na základě speciálních gradient vektorů umístěných v n-dimenzionální mřížce. Jeho příklady lze nalézt na obrázku 2.4.

Obrázek 2.4: Několik příkladů dvourozměrného Perlinova šumu.

Nejprve se tyto jednotkové vektory musejí vygenerovat. Mohou být vytvořeny například pomocí kongruentního generátoru. V novějších implementacích Perlinova šumu se tyto vektory často generují předem, nebo se dokonce využívá čtyř, respektive dvanácti předem daných vektorů, které směřují postupně ke všem hranám čtverce, respektive krychle. Tento postup popsal Ken Perlin v článku Improving Noise [12].

Dalším krokem je pak výpočet skalárního součinu vektorů vzdálenosti bodu, který chceme vypočítat, od rohů buňky v mřížce s gradient vektorem dané buňky. Pro dvourozměrný šum se tedy počítají čtyři součiny a pro trojrozměrný pak osm součinů, náročnost výpočtu tedy exponenciálně roste. Výsledky těchto součinů se pak mezi sebou interpolují zvolenou metodou, kde se jako parametr použije pozice počítaného bodu v rámci mřížky gradient vektorů. Výsledkem této interpolace je pak hodnota šumu v tomto bodě.

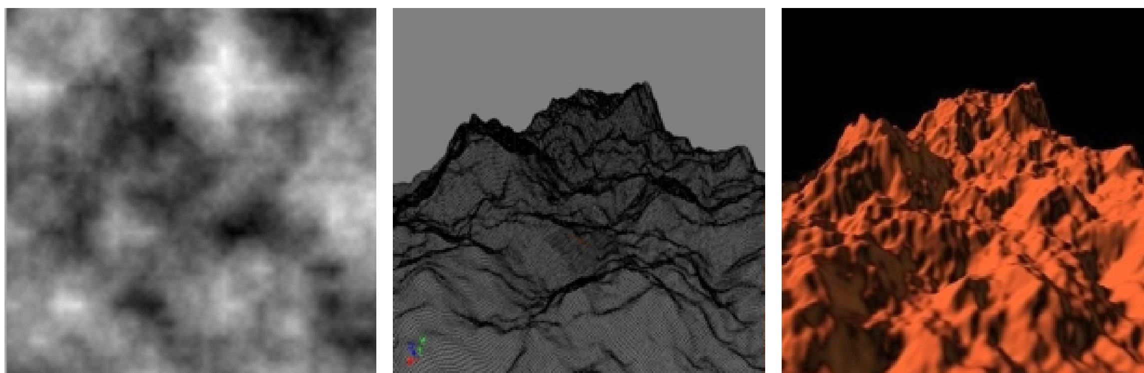


Obrázek 2.5: Příklad aplikování barevného přechodu na Perlinův šum.
Zleva výšková mapa, mapa prostředí, oheň
Převzato z [2].

Zvolený postup se několikrát opakuje s tím, že se při každém kroku mění měřítko mřížky, neboli frekvence a rozsah hodnot, neboli amplituda. Podle nastavení frekvence a amplitudy lze dosáhnout různých druhů výsledků. Některé se hodí například na generování ostrovů, jiné na pohoří, a tak podobně. Každému takovému kroku se říká oktáva šumu. Výsledky každého kroku se pak sečtou do finálního šumu, který se nazývá fraktální.

Perlinův šum lze v praxi využít k různým účelům. Mezi častá využití patří například generování výškových map, podle kterých se následně mění výšková pozice vrcholů terénu a dává tak dojem organicky vyhlížejícího terénu, který však nemusel být tvarován ručně. Příklad tohoto použití je pak na obrázku 2.6. Po aplikování barvy na výškovou mapu lze

vytvořit texturu, využitelnou například jako minimapu, nebo přímo pro obarvení terénu, podobně jako na obrázku 2.5. Takto se dá také aplikovat barevný přechod, což dá vzniknout například textuře ohně, lávy, oceánu a tak dále. Při použití trojrozměrného šumu se jako třetí dimenze využívá čas a vzniká tak textura animovaná.



Obrázek 2.6: Převod výškové mapy na model a následné otexturování.
Převzato z [2].

Může být využita pro tvorbu přechodů mezi texturami, například tráva a kamenná dlažba, nebo animovaný přechod z jednoho snímku na druhý. V neposlední řadě se dá využít pro vytváření různorodých modelů, kde je náhodnost vítaná, například kamení, zrnka písku, dlažba a tak dále. Zde je často využíván spolu s teselací.

2.2.1 Interpolace šumu

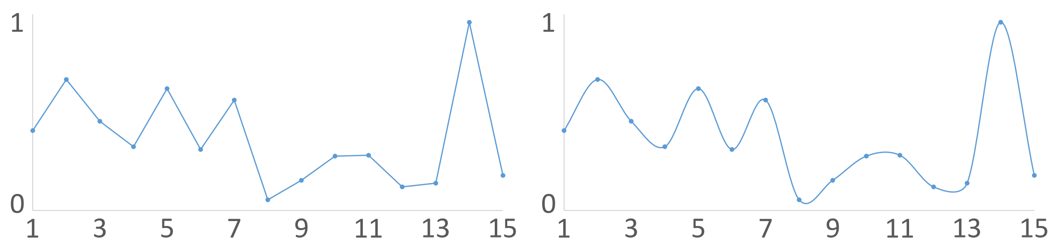
Aby šum vypadal organicky, musí se nejdříve zbavit ostrých digitálně vypadajících hran. Toho se nejčastěji dosahuje interpolací hodnot šumu mezi diskrétními body. Existuje mnoho různých způsobu jak tyto body interpolovat. Na obrázku 2.7 lze vidět jako příklad lineární interpolaci a interpolaci Beziérovým splajnem třetího řádu. Interpolace splajnem je velmi užitečná pro aproximaci křivek, ale pro potřeby interpolace šumu je zbytečně složitá na výpočet. Pro tyto účely se často používají interpolace lineární, kubická, nebo kosinová.

Lineární interpolace je nejjednodušší na výpočet, nemá však příliš kvalitní výsledky. Počítá se jednoduchou lineární funkcí:

$$f(x) = x_1 + t(x_2 - x_1) \tag{2.3}$$

kde x_1 a x_2 jsou body, které chceme interpolovat a t je parametr interpolace v rozsahu $\langle 0, 1 \rangle$.

Lineární interpolace je vhodná metoda pro vykreslování přímek, nebo pro dopočty chybějících údajů, kde si nemůžeme problémů spojených s touto metodou všimnout pouhým okem. Pro účely v grafice je však nedostačující, jelikož vytváří příliš ostré přechody, jako jde vidět na obrázku 2.7. Toto je ještě více viditelné ve vyšších rozměrech, kdy u dvourozměrného šumu tvoří lineární interpolace artefakty, jako na obrázku 2.8. Při použití v animacích pak tvoří nepřirozené přechody mezi jednotlivými vrstvami šumu. Tuto interpolaci lze tedy využít pouze v první dimenzi, v dalších je lepší využít pokročilejší metody interpolace.

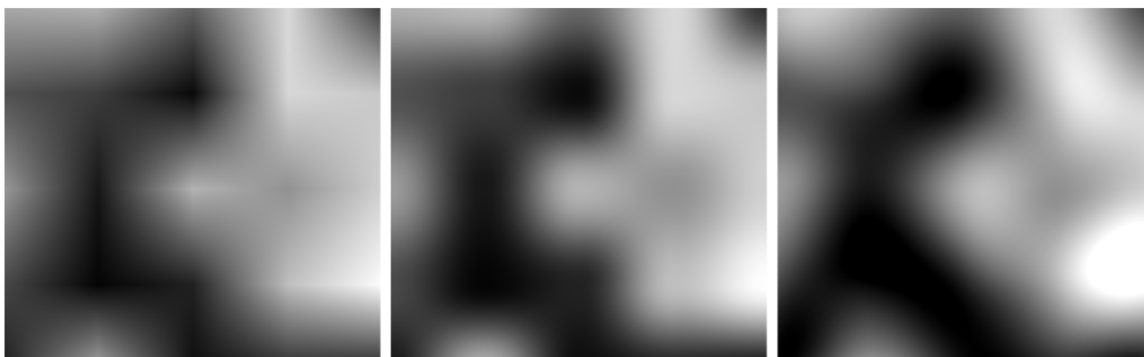


Obrázek 2.7: Porovnání lineární interpolace se splajnem.

Kubická interpolace využívá oproti lineární interpolaci čtyři okolní body pro výpočet bodu avšak do výpočtu je nezbytné zahrnout vyšší počet okolních bodů výsledného. Těmito body pak prochází kubický splajn. Tato metoda produkuje, na rozdíl od metody lineární, značně kvalitnější výsledky, což jde vidět na obrázku 2.8, avšak do výpočtu je nezbytné zahrnout vyšší počet okolních bodů. Tato skutečnost značně zpomaluje výkon této interpolace, zvláště pak ve vyšších dimenzích, kde se počet bodů zvětšuje s mocninou velikosti dimenze. Kubická interpolace je definována funkcí

$$\begin{aligned} a &= (x_4 - x_3) - (x_1 - x_2) \\ b &= (x_1 - x_2) \\ c &= (x_3 - x_1) \\ f(x) &= at^3 + (b - a)t^2 + ct + x_2 \end{aligned} \tag{2.4}$$

kde x_1, x_2, x_3 a x_4 jsou opět body, které chceme interpolovat a t je parametr interpolace.



Obrázek 2.8: Porovnání interpolací digitálního šumu.
Zleva lineární, kosinová a kubická interpolace.

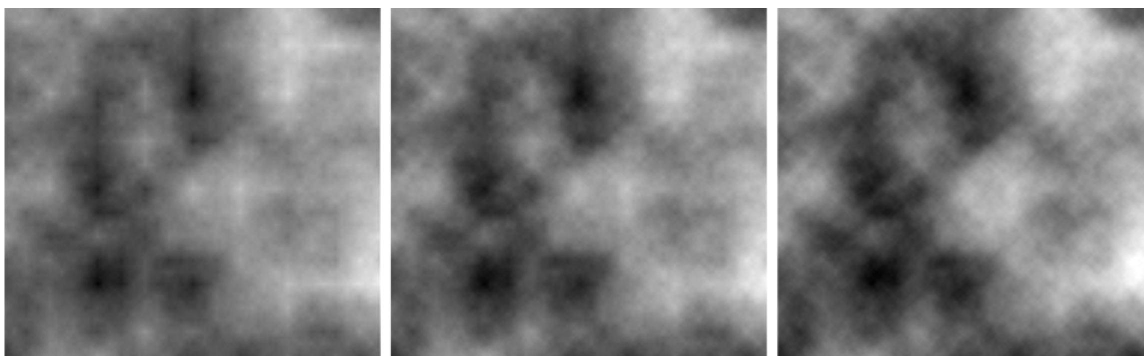
Dobrý kompromis mezi lineární a kubickou interpolací je pak interpolace kosinová, která se počítá pouze ze dvou bodů, jako v případě lineární interpolace, ale dosahuje podobných, ne-li identických výsledků jako interpolace kubická. Lze ji dobře využít jak v dvou, tak třech dimenzích, není však příliš vhodná pro animace, kde je znatelný efekt přechodu mezi dvěma krajními body. kosinová interpolace je definovaná funkcí

$$\begin{aligned} t' &= (1 - cos(t\pi) \cdot 0.5) \\ f(x) &= a + t'(x_2 - x_1) \end{aligned} \tag{2.5}$$

kde stejně jako u lineární interpolace jsou x_1 a x_2 body k interpolaci a t je poměr interpolace. Tato interpolace je taktéž vyobrazena na obrázku 2.8.

Při použití daných interpolací při výpočtu Perlinova šumu jsou některé artefakty méně znatelné, ale i přesto lze na obrázku 2.9 vidět, že kosinová a kubická interpolace produkují značně lépe vypadající výsledky, oproti interpolaci lineární. Mezi kubickou a kosinovou již není tak znatelný rozdíl, aby se jí vyplatilo věnovat drahocenný výkon. Toto platí i ve třech rozměrech za předpokladu, že tento rozměr není využíván pro animace. Pak je kubická metoda žádána.

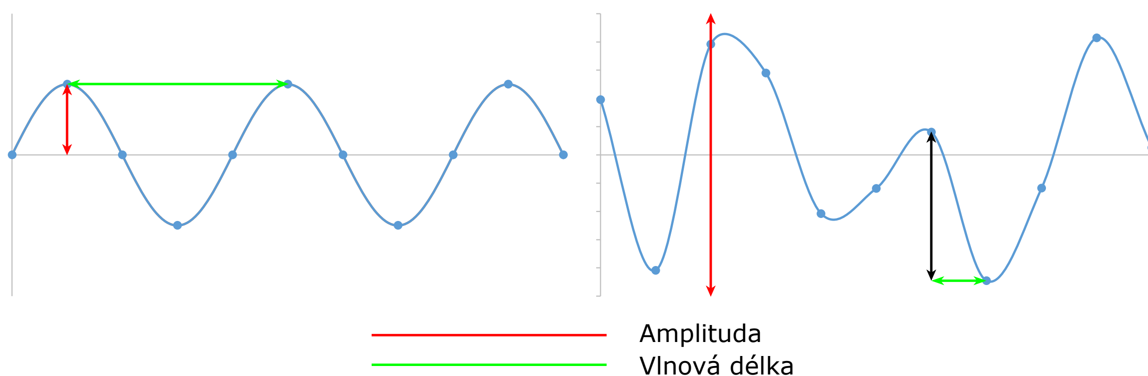
Metody lze samožřejmě libovolně kombinovat, lze tak například použít kosinovou interpolaci pro dvourozměrný šum a pro třetí rozměr použít interpolaci kubickou. Tohoto se dá využít právě v animacích, kde jednotlivé snímky budou vypadat dobře i s kosinovou interpolací, ale samotné přechody budou znatelně kvalitnější, než kdyby byla použita interpolace kosinová i na třetí rozměr.



Obrázek 2.9: Porovnání interpolací Perlinova šumu.
Zleva lineární, kosinová a kubická interpolace.

2.2.2 Persistence šumu

Při vytváření jednotlivých oktáv fraktálního šumu je nutné v každém kroku modifikovat parametry šumu tak, aby každá další oktáva obsahovala větší části o větším rozsahu hodnot. Tyto parametry jsou amplituda a frekvence šumu. Pro ilustraci je na obrázku 2.10 porovnání amplitudy a vlnové délky harmonické funkce s funkcí šumovou. Frekvence je definovaná jako $\frac{1}{\lambda}$, kde λ je vlnová délka.



Obrázek 2.10: Amplituda a vlnová délka harmonické a šumové funkce.

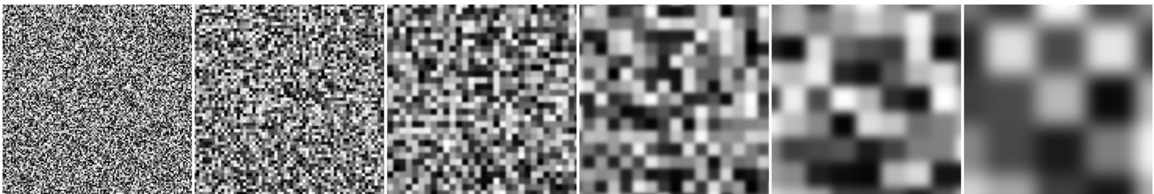
Amplitudou se myslí rozsah hodnot, kterých může daná oktáva šumu nabývat, a frekvencí velikost mřížky gradient vektorů. Z této skutečnosti plyne, že čím větší frekvence,

tím je výsledná oktáva více šumivá. Společně s menším rozsahem hodnot má však každá oktáva s vyšší frekvencí menší dopad na výsledek. To zajistí, že v šumu nebudou příliš ostré přechody mezi maximem a minimem, a naopak vyhladí výslednou křivku.

Pro vytvoření výsledného fraktálního šumu je tedy nutné vygenerovat několik oktáv s postupně se měnící frekvencí a amplitudou. Aby se v každém kroku obě měnily stejně, je zpravidla dobré si určit, v jakém poměru bude ke změně docházet. Tomuto poměru se pak říká persistence šumu, a používá se jako vstupní parametr do šumové funkce. Tento poměr není standardizovaný, proto se často liší mezi různými implementacemi. Jedna z možných variant poměru je například zde [3]. V této práci byl zvolen vzorec

$$\begin{aligned} f &= (2^k)^{-1} \\ a &= (p^k)^{-1} \end{aligned} \tag{2.6}$$

kde f je frekvence, a je amplituda, p je persistence a k je stupeň oktávy. Tedy čím větší persistence, tím větší mají nižší oktávy dopad na výsledný šum. Při takto zvoleném vzorci jsou pak nižší oktávy více šumivé, s menším rozsahem hodnot, než vyšší. Tento vzorec se často implementuje tak, že se amplituda zvětšuje nad rozsah hodnot výsledného šumu, a ten se po sečtení všech oktáv podělí součtem amplitud jednotlivých oktáv.



Obrázek 2.11: Porovnání poměrů frekvence a amplitudy.

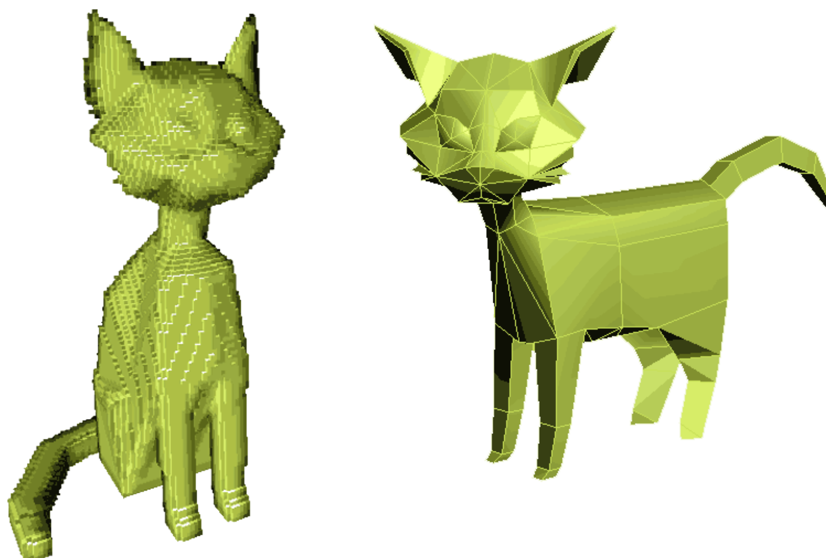
Jednotlivé oktávy pak mohou vypadat podobně jako na obrázku 2.11. Po jejich složení vznikne výsledný fraktální Perlinův šum na obrázku 2.12. Tyto šumy byly vygenerovány s persistencí 0.5, čili každá další oktáva má poloviční frekvenci a dvojnásobnou amplitudu. Při tomto způsobu generování jsou tedy často vyžadovány vyšší oktávy fraktálních šumu, aby vypadaly dobře. U tohoto šumu bylo zvoleno devět oktáv, ale na obrázku 2.11 jsou zobrazeny jen některé.

Obrázek 2.12: Výsledný fraktální šum po sečtení všech oktáv.

Kapitola 3

Voxely

Voxel je vizuální reprezentace dat v trojrozměrné uniformní mřížce. Název voxel pochází ze spojení anglických slov volumetric (prostorový, objemový) a element (prvek). Stejně jako jeho příbuzný prvek jménem pixel neobsahuje žádné údaje o své pozici, pouze data, která nese. Pozice je pak implicitně daná pozicí voxelu v mřížce.



Obrázek 3.1: Porovnání voxelové reprezentace s polygonální.
Převzato z [1].

Voxely slouží pro vyobrazení objemových dat, jako jsou například skeny reálných objektů. Důvod, proč jsou k tomuto účelu vhodnější je ten, že na rozdíl od polygonální reprezentace voxely nesou data i o vnitřku tělesa. Díky tomu je u těchto těles možno počítat například hustotu materiálu, fyzikálně přesnou deformaci, nebo také simulaci kapalin.

Nejčastější využití voxely nalézají v medicíně, kde se využívá pro uložení skenů z magnetické rezonance, a pak také v počítačových hrách, kde se užívá pro umožnění zničitelnosti terénu, jako například ve hře Minecraft. AutoMontage Engine [5] pak využívá voxelů pro kompletní zobrazení scény a fyzikální simulaci chování terénu, například protáčení kol vozidel v písku.

3.1 Uložení voxelů v paměti

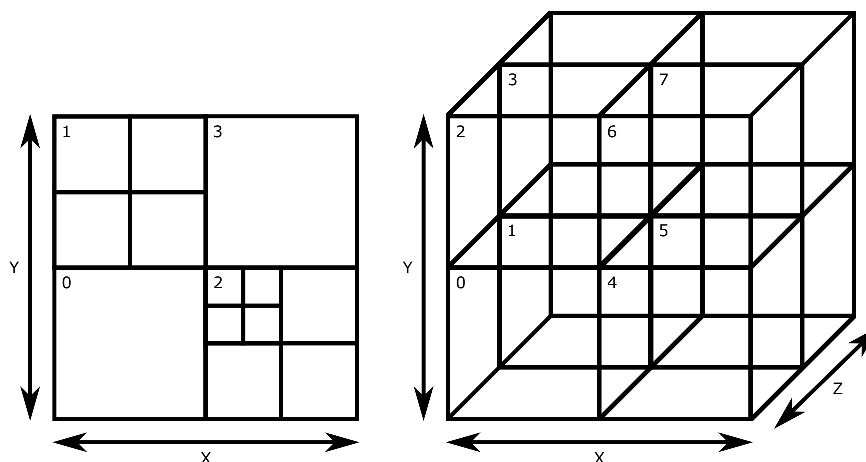
Na rozdíl od polygonálních modelů, které dokáží popsat i velmi obsáhlá tělesa relativně malým objemem dat, voxely jsou velmi paměťově náročné. Pokud by se uložily v paměti stejně, jako jsou vyobrazeny v uniformní mřížce, bylo by potřeba tolik paměti, jako je součin největších rozměrů v každé ose objektu. To samozřejmě není problém u terénu, který svým tvarem v podstatě odpovídá tvaru uniformní mřížky, pro uložení objektů, je tedy potřeba najít pokročilejší způsob reprezentace voxelů v paměti.

3.1.1 Seznam voxelů

První z těchto reprezentací je seznam voxelů. Jedná se o datovou strukturu podobnou polygonální reprezentaci. Každý voxel má k datové hodnotě přidanou také svou absolutní pozici ve scéně. To sice zvyšuje paměťovou náročnost samotných voxelů, ale u těles, která jsou relativně prázdná, nemusíme ukládat data o prázdných voxelech. Velkou nevýhodou tohoto způsobu uložení voxelů v paměti je pak náročnost na průchod v paměti. Jelikož není pozice voxelů uložená implicitně nelze jednoznačně určit, zdali aktuálně zkoumaný voxel sousedí s jiným, aniž bychom nejprve zkontrolovali všechny voxely ve scéně.

3.1.2 Sparse Voxel Octree

Další možností jak voxely uložit je pak takzvané octree. Princip tohoto uložení se dá lépe vysvětlit na zjednodušeném modelu quadtree, který je prakticky octree, ale pouze dvoudimenzionální. Quadtree se dá představit jako čtverec, který se neustále dělí na 4 další podčtverce tak dlouho, dokud není velikost podčtverce právě jeden pixel, respektive voxel u octree. Pokud čtverec obsahuje alespoň jeden pixel, pak je rozdělen na podčtverce, neboli pokud je čtverec prázdný, již se nedělí. Tímto se dá velmi výhodně popsat útvar, který má velké souvisle plné, či prázdné oblasti. Příklad quadtree i octree je vidět na obrázku 3.2.

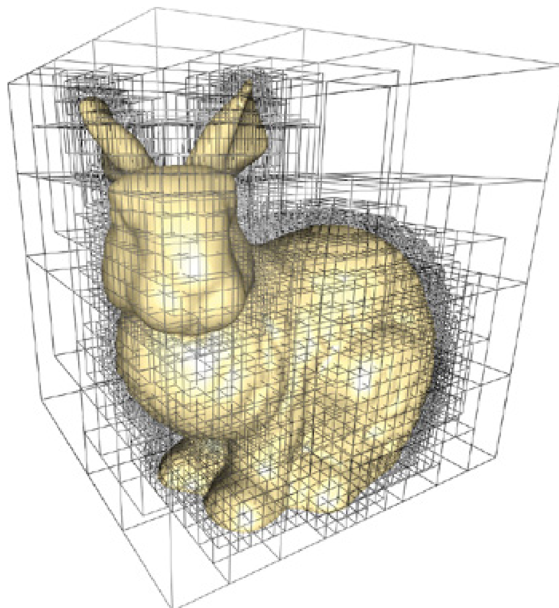


Obrázek 3.2: Příklad quadtree a octree struktury.

Octree je pak v principu identický s quadtree, akorát je zobrazen v trojrozměrném prostoru, čili místo čtverce a 4 podčtverců obsahuje krychli a 8 podkrychlí. Příklad uložení voxelově reprezentovaného objektu v octree lze vidět na obrázku 3.3.

Octree lze vytvořit relativně snadno ze seznamu voxelů buďto systémem bottom-up (sdola nahoru), nebo top-down (shora dolů). Bottom-up algoritmy nejprve generují nejnižší

vrstvy octree a následně navrstvují nadkrychle, dokud se nedostanou k největší krychli, které se přezdívá kořen (root). Top-down naopak generují od kořene směrem dolů až k jednotlivým voxelům, kterým se pak přezdívá listy (leaves). Každá krychle mezi kořenem a listy se pak nazývá uzel (node).



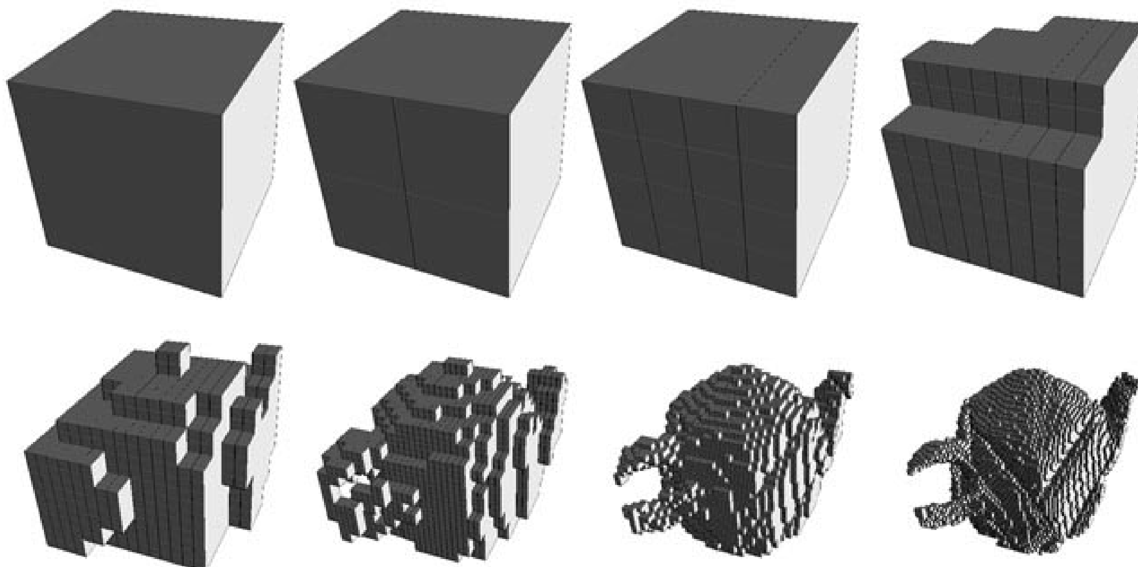
Obrázek 3.3: Příklad umístění objektu v octree.
Převzato z [13].

Po vytvoření stromu je nutné provést takzvaný mipmapping, neboli odstranění nepotřebných informací z octree. Ten se zpravidla provádí bottom-up, přičemž pokud je nadkrychle plná, či naopak prázdná, poznamená se do ní tato informace a všechny její děti se smažou z paměti. Tímto zabráníme zbytečnému plýtvání paměti v případě velkých zaplněných, či naopak prázdných oblastí. Nevýhoda tohoto postupu je, že pokud se bude octree někdy v budoucnu měnit, musí se tyto informace zpětně dogenerovat. Mipmapping se může dělat buď při vytváření jednotlivých úrovní, nebo až po kompletním vytvoření.

Díky víceúrovňové struktuře octree je možné objekty v nich uložené zobrazovat s různou úrovní detailů (Level of Detail, nebo také LoD), aniž by se tyto úrovně musely předem vytvářet ručně, jako je tomu u polygonální reprezentace. Při zobrazování stačí určit do které úrovně octree má vykreslovací algoritmus prostupovat a výsledný objekt se objeví s nižší úrovní detailů. Tohoto lze využít například pro vykreslování vzdálených objektů, které není třeba vykreslovat stejně detailně jako objekty blíž ke kameře. Objekty s nižší úrovní detailů se vykreslují znatelně rychleji, a při určité vzdálenosti od kamery, kdy by se jednotlivé voxely nepromítly do jednotlivých pixelů, by detailní objekty pouze zpomalovaly běh aplikace. Příklad objektu ve více úrovních detailů lze vidět na obrázku 3.4.

Výhody octree jsou zřejmé pro velké objekty s obsáhlými plnými, či naopak prázdnými oblastmi, kde zabírají daleko méně místa v paměti. Jako další výhodu pak lze brát průchod octree, který se dá využít při vykreslování pomocí vysílání paprsků, neboli ray casting, nebo při zjišťování, zda kurzor ukazuje na voxel, což se také nazývá ray picking. Průsečík paprsku s octree se totiž počítá vždy od nejvyšší úrovně do nižší, takže pokud paprsek úplně mine velkou část octree, není důvod počítat průsečík s danými voxely.

Nevýhodou octree je pak zvýšená paměťová náročnost pro objekty, které mají spoustu



Obrázek 3.4: Srovnání objektu v s různými úrovněmi detailů.
Převzato z [4].

přechodů mezi plnými a prázdnými oblastmi. U těchto objektů je často nutné generovat velkou část octree, což může ve výsledku zabírat více místa, než pouhá uniformní mřížka. Dalším problémem je pak editace octree, kdy se při přidání, či naopak odebrání voxelů musí přepočítat mipmapping celé struktury.

3.1.3 Chunk

Při uložení do uniformní mřížky u terénu lze také najít spoustu optimalizací. Jednou z nejčastěji používaných je umístění voxelu to takzvaných chunků, neboli menších bloků obsahujících pevně daný počet voxelů. Tato struktura má všechny výhody uniformní mřížky, jako rychlý přístup do paměti a jednoduché hledání sousedních voxelů. Přístup se o něco zpomalí pouze při průchodu mezi dvěma sousedními chunky. V praxi se pak v paměti vygeneruje určitý počet chunků, který zpravidla odpovídá dohledové vzdálenosti, a postupně se chunky z paměti odstraňují a generují nové podle potřeby. V paměti je tak stabilní množství voxelů, ale terén může být ve výsledku poskládán z mnohonásobně většího počtu voxelů.

Tento způsob uložení se často používá u zničitelného terénu, kde se předpokládá velký důraz na rychlost hledání nejbližších sousedů a náročnost na velikost terénu v horizontálních osách. Oproti octree je tedy vhodnější pro velká prostředí, protože umožňuje zvětšovat velikost světa, aniž by se zvětšovala výška, která by zůstala nevyužita.

3.2 Zobrazení voxelů

S voxely přichází zajímavý problém, a to jak je zobrazit na obrazovce. Dnešní grafické akcelerátory jsou navrženy především pro použití s polygonální reprezentací, a pro vykreslování voxelů nemají žádné vestavěné prostředky. Je tedy nutné buďto převést voxely do polygonální reprezentace, nebo použít vlastní vykreslovací metody.

3.2.1 Převod na polygonální model

Jednodušší z obou variant je převod každého z voxelů na polygonální reprezentaci krychle. Tento postup je velmi častý, díky jeho jednoduché implementaci a možnosti použít již známé metody vykreslování pro voxely, jako například stínování, fyzikálně přesné vykreslování, nebo stíny.

Nastávají zde však problémy spjaté s vykreslováním velkého počtu objektů ve scéně, především obtíž předem určit, zdali se daný objekt celý nenachází za jiným objektem. Pak je jeho vykreslení přebytečné a zbytečně tak zpomaluje vykreslovací proces. Toto se dá zčásti vyřešit stejně jako u klasických vykreslovacích systémů, a to před výpočtem překrývání trojúhelníků pomocí depth bufferu.

Dalším problémem je pak případ, kdy je velké množství voxelů na obrazovce tak či tak renderováno. Dochází k tomu především v situacích, kdy je velká plocha téměř zaplněná, ale stále se vykreslují voxely na nejnižší úrovni. Tomuto lze zabránit přidáním třetího stavu pro nadkrychle, tedy místo prázdná a neprázdná bude prázdná, neprázdná a plná. Pokud je krychle prázdná, nic se nevykresluje jako doposud, pokud je neprázdná vykreslí se podkrychle, a pokud je plná, vykreslí se pouze jedna krychle s rozměry této nadkrychle a vykreslování podkrychlí se již neřeší.

3.2.2 Raycasting

Druhou možností je pak úplně vynechat vykreslovací proces grafické karty a vykreslovat ručně pomocí vysílání paprsků do octree, nebo ray casting [11]. Tato metoda přináší značné zrychlení jak pro překrývající se voxely, tak pro plné oblasti, protože nijak neřeší pozice jednotlivých voxelů při vykreslování, ale pouze používá algoritmu průchodu octree pro kolizi paprsku s voxely.

Z každého pixelu na obrazovce se vyšle paprsek směrem určeným úhlem pohledu kamery. Pokud tento paprsek protne krychli v nejvyšší úrovni, tedy v kořenu, spustí se výpočet kolize paprsku s každým voxelem v nižší úrovni. Toto se děje tak dlouho, dokud paprsek nekoliduje s voxelem na nejnižší úrovni. Jeho barva se pak zapíše do daného pixelu na obrazovce. Lze samozřejmě dodatečně použít i simulaci stínování.

Pokud se paprsek následně vyšle z bodu dotyku směrem ke zdroji světla, vzniká pokročilý algoritmus trasování paprsku, neboli ray tracing, který umožňuje kontrolu, zdali na daný voxel svítí světlo, nebo je zakryt jiným voxelem, a je tím pádem ve stínu. Toto je při standardní polygonální rasterizaci docela náročné, ale u trasování paprsku nejde o příliš náročný výpočet.

Tento algoritmus je také možné použít k určení, na který voxel ve scéně ukazuje kurzor myši. Místo všech pixelů na obrazovce se vyšle paprsek z místa, kde se kurzor myši nachází, a vrátí se pozice voxelu, se kterým paprsek koliduje. Tento voxel můžeme například nabarvit odlišnou barvou, aby šlo vidět, že je vybrán kurzorem.

Kapitola 4

Implementace

Aplikace samotná je implementována v jazyce C++ ve verzi ISO C++11. Tento jazyk byl zvolen především pro jeho robustnost a také z důvodu obeznámení autora s daným jazykem. Pro vytvoření okna aplikace byla zvolena knihovna SDL (Simple DirecMedia Layer [1]) ve verzi 2.0.5. Tato knihovna nabízí jednoduchou implementaci základní okenní aplikace bez nutnosti vázat se na specifický operační systém, nebo architekturu. Nad touto knihovnou je následně implementována třída `class CWindow` v knihovně `Window`, která se stará o obsluhu hlavní herní smyčky, událostí systému a vstup z klávesnice a myši.

Na samotné vykreslování je použita knihovna OpenGL [2], respektive její nadstavba GPUEngine [3]. Pro zjednodušení volání funkcí knihovny OpenGL je pak použita pomocná knihovna GLEW (OpenGL Extension Wrangler [4]) a pro matematické výpočty byla zvolena knihovna GLM (OpenGL Mathematics [5]).

Výsledek aplikace je zobrazení terénu vygenerovaného pomocí zjednodušené varianty Perlinova šumu, uloženého do struktury octree. Je umožněno tento terén editovat přidáváním, nebo odebíráním voxelů, pohyb kolem samotného terénu a změna úrovně detailů zobrazování octree.

Většina zde uváděných algoritmů je implementována na grafické kartě pro zvýšení paralelismu při výpočtu a tak dosáhnutí výkonu, který umožňuje vykreslovat scénu v reálném čase. Pro tvorbu terénu, převod na octree, vybírání voxelů byly použity compute shadery, které se na podobné obecné výpočty velmi dobře hodí. Na vykreslování jsou pak použity klasické vertex a fragment shadery pro vykreslování krychlí. Pro zrychlení vykreslování je použita varianta pro vykreslení mnoha instancí stejného objektu najednou.

4.1 Tvorba terénu

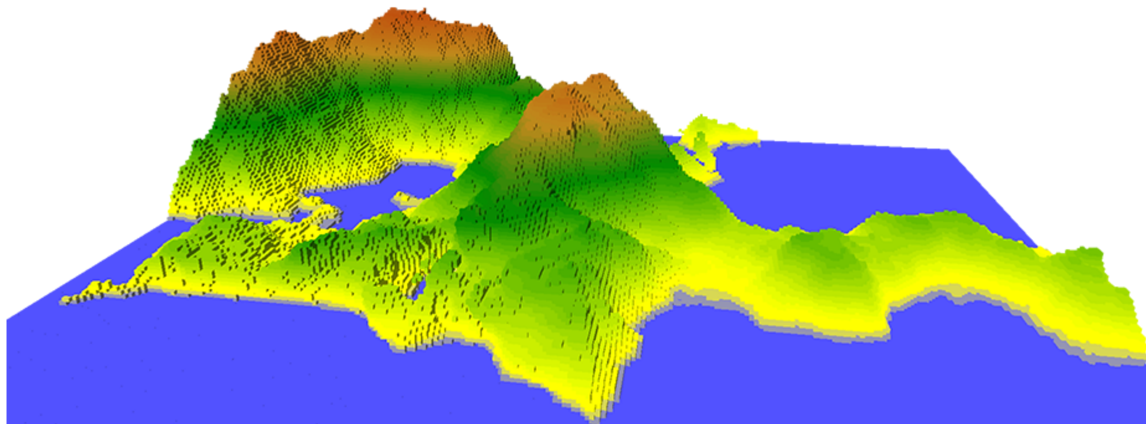
Pro vytvoření terénu byl původně zvolen algoritmus Perlinova šumu, avšak pro terén samotný nebylo potřeba tvořit gradient vektory, jelikož byl výpočet příliš náročný a nepřinášel znatelně lepší výsledky. Je tedy použit kongruentní generátor, který generuje digitální šum v několika úrovních, které jsou na každé z nich interpolovány kosinovou interpolací a následně sečteny do výsledného šumu.

[1]https://www.libsdl.org/
[2]https://www.opengl.org/
[3]https://sourceforge.net/projects/gpuengine/
[4]http://glew.sourceforge.net/
[5]http://glm.g-truc.net/

Tento postup je implementován na grafické kartě pomocí compute shaderu v souboru `voxels.comp`. Ten při překladu dovoluje nastavit libovolnou kombinaci interpolace, šumu (digitální, interpolovaný, nebo fraktální) a persistenci. Pro nejlepší výsledky byly zvoleny možnosti fraktální šum, kosinová interpolace a persistence 0.5 při devíti oktávách šumu.

Jako parametry šumu byly zvoleny souřadnice v prostoru, pro které se nejdříve spočítá dvojrozměrný šum jakožto výšková mapa. Pokud je daný voxel stejně vysoko, nebo níž, než maximální voxel v této mapě, je pro něj spočítán trojrozměrný šum, díky čemuž je možno vytvářet například jeskyně, krátery a plovoucí útvary. Práh v trojrozměrném šumu, po kterém se voxel vytvoří, byl experimentálně zvolen na hodnotu 0.02, čili pokud trojrozměrný šum v daném místě vrátí hodnotu menší, voxel se nebude generovat. To vytváří v terénu zajímavé krátery a jeskyně. Při vyšším prahu pak trojrozměrný šum maže velké množství voxelů, a terén již vypadá znatelně hůř.

Pro větší náhodnost je pro každou souřadnici šumu zvolen seed posunutý o jeden, respektive dva bity doleva pro y, respektive z souřadnici, pro x je pak seed zachován. Výsledek jedné souřadnice je pak dán na vstup další souřadnice v pořadí od x po z. Výsledná hodnota šumu je následně převedena do rozmezí $\langle 0, 1 \rangle$, kde pozice 0 je považována za vodu, a pozice 1 za nejvyšší možný vrchol hor.



Obrázek 4.1: Terén vygenerovaný pomocí trojrozměrného šumu do seznamu voxelů.

Výsledek tohoto programu je pak seznam voxelů, které ve svojí datové složce obsahují 32bitovou celočíselnou hodnotu, kde je na spodních třicet bitů zakódována souřadnice v trojrozměrném prostoru, neboli deset bitů na souřadnici v každé z os. Vrchní dva bity jsou pak nevyužité. Výsledný terén zobrazený pomocí tohoto seznamu lze vidět na obrázku 4.1.

4.2 Převod na octree

Dalším krokem je převod seznamu voxelů na octree. Tradičně se tento postup dělá ve třech krocích, kdy se v prvním vygeneruje aktuální úroveň octree, v druhém se vytvoří struktura pro další úroveň a v posledním kroku se inicializuje paměť. Tento postup je popsán zde [8]. Při práci na grafické kartě je však práce s dynamickým zvětšováním paměti velmi pomalá. Z tohoto důvodu se paměť alokuje celá. Výsledné octree bude zabírat v paměti $\sum_{i=1}^{n} 8^i$ 32bitových celých čísel, tedy čtyřikrát tolik bytů. Při osmi úrovních je to přibližně 66MB, při devíti pak kolem 600MB. Následně se tato paměť inicializuje na nuly a až pak se začne generovat první úroveň.

Prvním krokem implementovaného algoritmu je proto spustit vlákno na grafické kartě pro všechny voxely v seznamu. Každé vlákno si z daného voxelu dekóduje souřadnice, a pak na místě, kde by se měl voxel v aktuální úrovni nacházet nastaví nejvyšší bit na jedničku. Toto značí, že je daná nadkrychle neprázdná, a bude se dělit ne podkrychle. Tento krok je implementován v compute shaderu v souboru `octree1.comp`.

Dekódování souřadnic je poměrně jednoduché. Ze zásobníku voxelů každé vlákno na adresuje svým indexem jeden voxel, a následně jeho datovou složku rozdělí po deseti bitech spodních třicet bitů. Na nejvyšší úrovni je zakódována souřadnice v ose x, na druhé y a na nejnižších deseti bitech pak osa z.

Tyto souřadnice se pak musí zakódovat do indexu pro průchod octree, kde je v každé úrovni osm ukazatelů na první podkrychli dané nadkrychle. V první úrovni je tedy osm indexů, které každý ukazuje do paměti na dalších osm indexů, celkově tedy bude v druhé úrovni šedesát čtyři indexů a tak dále. Při průchodu octree se pak nejnižší bity z každé z tří souřadnic přičtou na nejnižší tři bity aktuálního indexu v pořadí xyz, což umožní indexaci podkrychlí jako na obrázku 3.2. Tento krok je implementován v compute shaderu v souboru `octree2.comp`.

Pro každou úroveň je potřeba tři bity na za adresování krychle v aktuální úrovni. Jelikož jsou však na grafických kartách pouze 32bit registry, dá se celé octree za adresovat pouze pomocí maximálně třiceti bitů, tedy maximální velikost octree je deset úrovní. Tento problém by se dal vyřešit použitím nadstavby, která dokáže počítat i s 64bit čísly v 32bit registrech, ale to by značně zpomalovalo výkon grafické karty. Dále pak při aktuálním způsobu renderování nelze reálně vykreslit octree větší, než 9 úrovní na průměrně silné grafické kartě.

Celý algoritmus včetně dekódování, převodu globálních souřadnic voxelů na index v octree a nastavení dané, nově vytvořené, podkrychle na plnou je pak znázorněn v kódu 4.1. V tom je `gl_GlobalInvocationID.x` identifikační číslo aktuálního vlákna grafické karty od 0 do počtu voxelů, `levels` je počet úrovní v octree a `currentLevel` je aktuální úroveň, pro kterou je algoritmus spuštěn.

Kód 4.1: Algoritmus převodu globální pozice voxelu na pozici v octree.

```
1  uint voxelID = voxelList[gl_GlobalInvocationID.x];
2
3  position.x = (voxelID >> 20) & 0x3FF;
4  position.y = (voxelID >> 10) & 0x3FF;
5  position.z = (voxelID      ) & 0x3FF;
6
7  for (int i = 1; i <= currentLevel; i++) {
8    octreeIndex += ((position.x >> (levels - i)) & 1) << 2;
9    octreeIndex += ((position.y >> (levels - i)) & 1) << 1;
10   octreeIndex += ((position.z >> (levels - i)) & 1);
11   if (i == currentLevel) { break; }
12
13   octreeIndex = octree[octreeIndex];
14 }
15
16 octree[octreeIndex] = (1 << 31);
```

Průchod octree je pak opakem tohoto postupu, neboli v každém kroku se globální pozice voxelu, brána většinou z čísla vlákna, postupně dělí mocninami osmi od nejvyšší po nejnižší,

což má za následek odfiltrování vyšších bitů pozice. Zbytek se pak použije pro adresaci v aktuální úrovni, tedy opět nejvyšší tři bity (pokud se tedy nepočítají nejvyšší dva, které pro pozici nejsou využity) pro nejvyšší úroveň a tak dále. Každý krok pak adresuje do octree a vrací počáteční index další podkrychle, pro kterou se bude pokračovat. Tento algoritmus je popsán v kódu 4.2. Opět je `gl_GlobalInvocationID.x` identifikační číslo aktuálního vlákna, `levels` počet úrovní v octree a `currentLevel` aktuální úroveň. Navíc je zde `index[0]`, což je zásobník o jednom prvku, který reprezentuje posun adresace o osm při každém nově vytvořeném uzlu. Funkce `atomicAdd()` pak zajišťuje, aby se index posunu zvětšoval atomicky, a nedocházelo ke kolizi vláken grafické karty.

Kód 4.2: Algoritmus převodu čísla vlákna na voxel v octree.

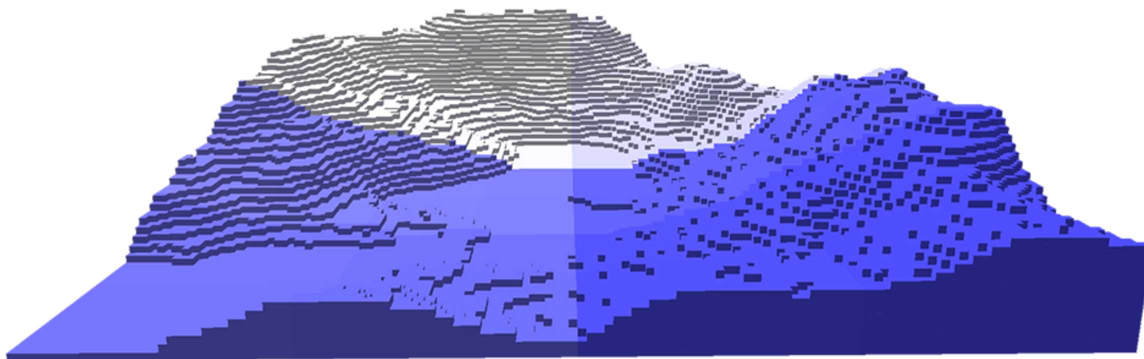
```
uint divisor = uint(pow(8, currentLevel - 1));

for (uint i = 1; i < currentLevel; i++) {
  octreeIndex += (gl_GlobalInvocationID.x/divisor) & 0x7;
  octreeIndex = octree[octreeIndex];
  divisor /= 8;
}

octreeIndex += gl_GlobalInvocationID.x & 0x7;

uint newNodeIndex = atomicAdd(index[0], 8);
octree[octreeIndex] += newNodeIndex;
```

V druhém kroku se pak spustí 8^i vláken, kde i je aktuální úroveň, neboli jedno vlákno na každou vygenerovanou podkrychli z prvního kroku. Každé vlákno pak prochází octree pomocí algoritmu popsaného výše, dokud nedojde na nově vygenerované krychle a připíše na její místo v paměti index, který bude ukazovat na pozici první z osmi podkrychlí dané nadkrychle v paměti. Tento index je počítán tak, že každé vlákno atomicky přičte osmičku k minulému indexu, čili index vždy poskočí o osm podkrychlí. Toto je opět popsáno v kódu 4.2.



Obrázek 4.2: Příklad umístění objektu v octree.

Tyto kroky se opakují pro všechny úrovně kromě první a poslední. První je dána implicitně, takže se předpokládá, že octree bude obsahovat alespoň jeden voxel. Pokud ne, promítne se to až v první úrovni. V poslední úrovni se pak spustí pouze první krok, protože již není nutné zapisovat indexy pro podkrychle, jelikož jsou na poslední úrovni čistě voxely.

Příklad terénu převedeného do octree je pak možné vidět na obrázku 4.2. Barva voxelů značí jejich pozici danou číslem v octree podle schématu 3.2, kde bílá barva je voxel číslo 0 a plně modrá barva pak voxel s maximální hodnotou. Pro zobrazení octree byla použita metoda polygonální reprezentace, jelikož byla jednoduchá na implementaci a pro tuto jednoduchou aplikaci je dostatečná.

4.3 Renderování octree

Pro vykreslení octree je možno zvolit z dvou postupů popsaných v teoretické části 3.2. Nakonec byl zvolen jednodušší postup polygonální reprezentace, protože pro tak malý projekt nepřinášel ray casting mnoho výhod, a zároveň byl velmi časově náročný.

Polygonální reprezentace je implementována tak, že se spustí instancované vykreslování krychlí, kde počet voxelů ve scéně je počet instancí. Za předpokladu, že je octree plný, se tak generuje přesně maximální zaplnění octree.

Kód 4.3: Algoritmus převodu čísla instance na pozici voxelu ve světě.

```
1  uint voxelID = gl_InstanceID;
2
3  while(voxelID > 0) {
4    x += ((voxelID & 4) >> 2)*i;
5    y += ((voxelID & 2) >> 1)*i;
6    z~+= ((voxelID & 1)     )*i;
7
8    voxelID = voxelID >> 3;
9    i *= 2;
10 }
```

Kód 4.4: Posun souřadnic voxelů při nižší úrovni detailů

```
1  uint lodLevelDIff = uint(pow(2, (levels - lod)));
2  vec3 pos = vec3(x, y, z);
3  pos *= lodLevelDIff;
4  pos += float(lodLevelDIff)/2.0;
```

Tento stav však nastává výjimečně a je tedy nutné některé voxely nezobrazovat. To se provádí v několika krocích. Nejprve se vezme identifikátor instance jako číslo voxelu, a převede se do globální pozice voxelu ve světě. Toto se provádí pomocí algoritmu 4.3, tedy z čísla voxelu se postupně odebírají nejnižší bity a přičítají se k jednotlivým souřadnicím. Při nižší úrovni detailů se pak výsledné souřadnice musí posunout tak, aby se při nižší úrovni detailů nabudil dojem, že se velikost octree nemění. Toto se provádí algoritmem 4.4, tedy pozice se zvětší o poměr úrovně detailů s úrovněmi octree a následně se posune o polovinu této vzdálenosti, čili se pozice změní od středu směrem ven.

Dalším krokem je pak průchod octree strukturou, a zjišťování, zdali je daný voxel v octree, nebo ne. To se provádí modifikovaným algortimem 4.2, který je označen jako kód 4.5. Tento algoritmus již prochází stromem od začátku až po zanoření na úrovni detailů. Při každém průchodu testuje, zdali má aktuální krychle nejvyšší bit nastaven na jedničku. Pokud ano, pokračuje, pokud ne, končí průchod. Dále se pak znovu tento bit testuje při samotném vykreslování. Pokud je bit nastaven na jedničku, krychle se vykreslí na pozici dané voxelem. Jestliže tomu tak není, všechny vrcholy krychle se přesunou do bodu $[0, 0, 0]$, tedy krychle se skryje do nekonečně malého bodu a nejde tím pádem vidět.

Nakonec se zjistí výška daného voxelu, podělí se výškou maximální a výsledek se pošle do fragment shaderu, zároveň s informací, jestli je daný voxel označen, či nikoliv. Ve fragment shaderu se pak daná hodnota vloží do barevného přechodu a výsledná barva se aplikuje na voxel zároveň s Phongovým osvětlovacím modelem. Pokud je však daný voxel vybrán kurzorem, nastaví se základní barva na žlutou.

Kód 4.5: Algoritmus pro průchod octree až na danou úroveň detailů. Smyčka se zastaví, pokud na některé úrovni nalezne prázdnou krychli.

```
1 uint divisor = uint(pow(8, levels-1));
2
3 for (uint i = 0; i < lod; i++) {
4   octreeIndex += (voxelID/divisor) & 0x7;
5   octreeIndex = octree[octreeIndex];
6   divisor /= 8u;
7   if ((octreeIndex & 0x80000000) == 0) { break; }
8   }
9 }
```

4.4 Raypicking

Pro výběr voxelu, který má být v octree editován je nutné vybrat bod na obrazovce, který danému voxelu odpovídá. Ten se vybere tak, že se z obrazového bodu, na kterém je kurzor myši, vyšle paprsek směrem do scény, a zjišťuje se, zdali tento paprsek protnul octree. Pokud ano, opakuje se testování pro jednotlivé podkrychle, dokud se algoritmus nedostane až na úroveň jednotlivých voxelů, nebo na nejnižší úroveň detailů. K tomuto slouží algoritmus pro průchod octree popsaný zde [14].

Kód 4.6: Testování průniku paprsku s krychlí.

```
1  vec3 t0 = (minPoint - ray_origin)*(1.0/ray_direction));
2  vec3 t1 = (maxPoint - ray_origin)*(1.0/ray_direction));
3
4  vec3 tminvec = min(t0, t1);
5  vec3 tmaxvec = max(t0, t1);
6
7  float tmin = max(max(tminvec.x, tminvec.y), tminvec.z);
8  float tmax = min(min(tmaxvec.x, tmaxvec.y), tmaxvec.z);
9
10 voxelPos = (minPoint + maxPoint)/2.0;
11 voxelID = voxelPos2ID(voxelPos, currLevel);
12
13 if ((tmax < 0) || (tmin > tmax)) {
14   return false;
15 }
16
17 return checkVoxelFull(voxelID, currLevel);
```

Pro samotný test průniku paprsku s krychlí se používá algoritmus popsaný kódem 4.6. Nejprve se vytvoří vektor, který udává vzdálenost zdroje paprsku od jednotlivých os krychle.

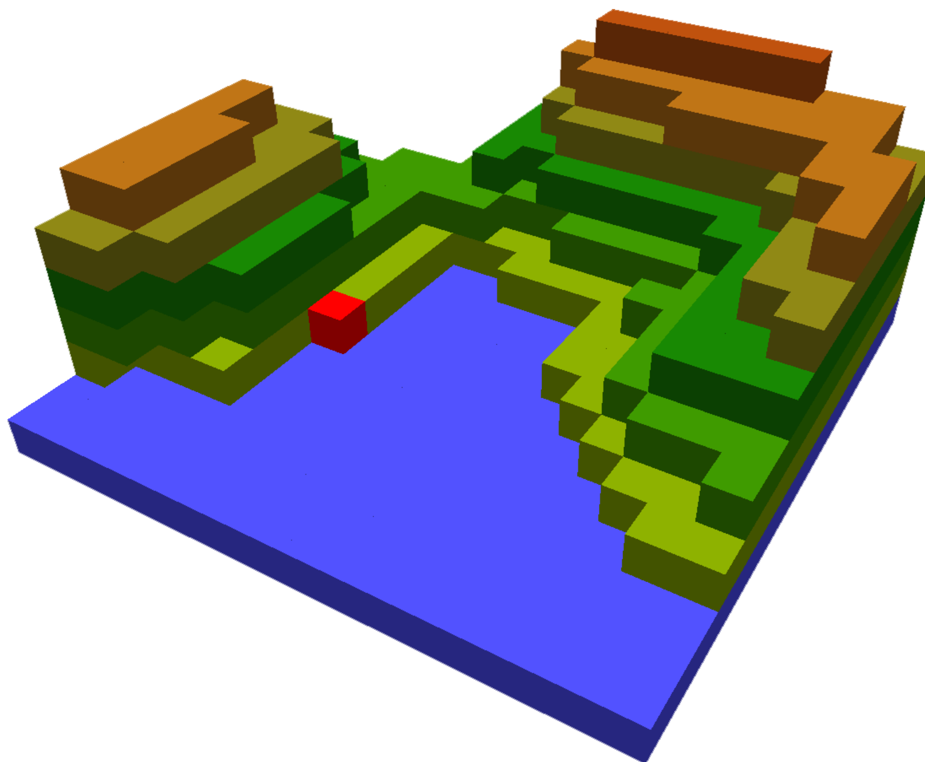
Tento vektor udává, která osa je zdroji paprsku nejbližší. Následně se provede test průniku paprsku v jednotlivých osách. Pokud je vzdálenost průniku v nejzazší ose větší, než v nejbližší ose, pak paprsek trefil krychli, jinak minul. Nakonec je ještě nutné zjistit, zda daná krychle obsahuje alespoň jeden voxel. Pokud ne, je považována za prázdnou a algoritmus pokračuje další krychlí. Tento test je nutné opakovat pro každou další podkrychli, problém však nastává, když paprsek danou podkrychli minul, a algoritmus se musí vrátit o úroveň dřív. Pro vyřešení tohoto problému je použita rekurze. Při průniku paprsku s krychlí se algoritmus spustí znova pro všechny podkrychle dané nadkrychle. Pokud narazí na voxel, algoritmus končí, jinak pokračuje s další sousední krychlí. Příklad vybrání voxelu myší je na obrázku 4.3.

Bohužel tento postup nelze jednoduše použít na grafické kartě, protože ta v základu nepodporuje rekurzivní volání funkcí. Tento fakt se dá však obejít implementací vlastního zásobníku, a přepracováním rekurze na smyčku, viz [11]. Tento nový algoritmus je pak popsán zjednodušeným kódem 4.7. Pokud je při testování nalezen průnik, a aktuální úroveň zanoření voxely je rovná úrovni detailů, test končí a daný voxel je označen jako vybraný. Jinak se do zásobníku vloží všechny podkrychle a test pokračuje, dokud není nalezen nějaký voxel, nebo není vyprázdněn zásobník.

Kód 4.7: Převod rekurze na smyčku, pro testování průniku paprsku s octree.

```
1  for (i = 7; i >= 0; i--) {
2    stackPush((1 << 3) + i);
3  }
4
5  while (!stackEmpty()) {
6    stackData = stackPop();
7    voxelIndex = stackData.voxelIndex;
8    currLevel = stackData.level;
9
10   if (rayAABBIntersection(currLevel, voxelID)) {
11     if (currLevel == lod) {
12       //voxel nalezen
13       return true;
14     } else {
15       for (int i = 7; i >= 0; i--) {
16         stackPush((1 << 3) + i);
17       }
18     }
19   }
20 }
21 //voxel nenalezen
22 return false;
```

Pokud je mód editace nastavený na přidávání, provádí se navíc ještě jeden krok, který zjistí, na kterou stranu krychle ukazuje kurzor myši, a vrací identifikační číslo voxelu, který s vybraným voxelem sousedí. Ten je pak zvolen jako označený, pro visuální reprezentaci přidávaného voxelu.



Obrázek 4.3: Příklad zvolení voxelu pro úpravu kurzorem myši.

4.5 Editace

Voxel vybraný postupem popsaným v kapitole 4.4 je pak možno buďto smazat, nebo přidat nový na jednu z jeho stran. Taková editace může probíhat na libovolné úrovni detailů, jelikož úprava voxelu se promítne i do nižších úrovní. Takto lze například rychle smazat velkou část scény. Průchod octree a přepnutí vybraného voxelu na plný, či naopak prázdný je popsáno algoritmem 4.8.

Kód 4.8: Průchod octree s nastavením voxelu na plný/prázdný.

```
1  uint divisor = uint(pow(8, lod - 1));
2  for (uint i = 0; i < lod; i++) {
3    octreeIndex += (voxelID/divisor) & 0x7;
4    if (i == lod-1) { break; }
5    octreeIndex = octree[octreeIndex];
6    divisor /= 8;
7  }
8
9  if (adding) {
10   octree[octreeIndex] = octree[octreeIndex] | 0x80000000;
11 } else {
12   octree[octreeIndex] = octree[octreeIndex] & 0x3FFFFFFF;
13 }
```

Při smazání se musí zkontrolovat, zdali nebyla smazána poslední podkrychle, tedy jestli se má daná nadkrychle zobrazovat při nižší úrovni detailů. Naopak při přidávání se musí

nadkrychle také přidat. Tento krok je nutné opakovat pro každou úroveň, až po kořen. Tento postup je pak popsán v kódu 4.9.

Kód 4.9: Zpětný průchod octree a nastavení nadkrychlí na plnou/prázdnou.

```
1  while (lod > 0) {
2    voxelID = voxelID >> 3;
3    octreeIndex = traverseOctree(voxelID, lod);
4
5    if (adding) {
6      if (countNonEmptyNodes(octreeIndex) != 0) {
7        octree[octreeIndex] = octree[octreeIndex] | 0x80000000;
8        continue;
9      }
10   } else {
11     if (countNonEmptyNodes(octreeIndex) == 0) {
12       octree[octreeIndex] = octree[octreeIndex] & 0x3FFFFFFF;
13       continue;
14     }
15   }
16   break;
17 }
```

Jako poslední krok je pak opětovný průchod jako při tvorbě octree algoritmem 4.2, tentokrát však počínaje upravenou krychlí a níž. Všechny krychle v nižších úrovních musí být nastaveny stejně jako v té aktuální, čili pokud se smaže nadkrychle v první úrovni, musí se smazat příslušné podkrychle i ve všech následujících úrovních. Případně u přidávání se musí naopak také přidat. Tento průchod je rozdělen na dvě části. V první se nalezne pozice aktuální krychle v octree a následně se pokračuje pro všechny podkrychle na dané úrovni. Tento algoritmus se pak spouští několikrát, pro každou úroveň, kterou je třeba změnit.

Kapitola 5

Závěr

Tato práce se zaobírala tématy procedurálního generování scény, především pak terénu tvořeného voxely. V první části byl tento problém rozebrán teoreticky, především pak možnosti generování textur pomocí šumů a jejich využití, teoretické využití voxelů, způsoby jejich uložení do paměti a následné vykreslování.

Součástí práce je také aplikace, jejíž implementace je popsána v kapitole 4. Tato aplikace byla vytvořena pro ověření platnosti popsaných algoritmů v praxi, především pak jejich implementace na moderní grafické kartě. Po spuštění generuje pomocí trojrozměrného šumu výškovou mapu terénu, ze kterého vytváří seznam voxelů. Ten pak převádí na voxel octree, který umožňuje modifikovat přidáváním a odebíráním voxelů v různých úrovních detailů. Terén dokáže exportovat do souboru a také z něj importovat. Lze ovládat myší a klávesnicí a částečně i standardním herním ovladačem.

Při vytváření aplikace bylo nalezeno několik problémů používání Sparse Voxel Octree, především pak nutnost přegenerování celé struktury při její modifikaci. Tato skutečnost vedla k použití plného octree, čili bez použití mipmappingu. Výsledná velikost v paměti nebyla výrazně větší, a rychlost modifikace se razantně zvýšila. Dalším problémem bylo zobrazení octree pomocí polygonální reprezentace, která sic jednoduchá na implementaci byla na slabších grafických kartách poměrně pomalá. Pro rychlejší zobrazování by tedy bylo vhodnější implementovat metodu ray castingu, nebo metodu s polygonální reprezentací příslušně optimalizovat, například vykreslováním pouze viditelných voxelů.

V budoucnu by se tedy práce dala rozvést implementací robustního vykreslovacího algoritmu používajícího metodu ray casting. Ten by razantně zvýšil rychlost a umožnil by jednoduchou implementaci dynamických stínů. Dále by bylo možné přidat více možností pro editování, například větší rozměry krychlí i při větší úrovni detailů, možnost změny barvy jednotlivých voxelů, nebo pokročilé možnosti editace i pomocí herního ovladače. Ten se aktuálně používá primárně pro ovládání kamery. V neposlední řadě by pak bylo možné generovat různé druhy biomů, které by se lišili v barvě a výšce terénu, případně by mohli obsahovat i vegetaci.

Literatura

[1] *Computer Graphics: Voxels vs. Polygons*. [Online; navštíveno 16.5.2016].
URL http://matthewprongecs100w.blogspot.cz/2013/12/normal-0-false-false-false-en-us-x-none.html

[2] *How to Use Perlin Noise in Your Games*. [Online; navštíveno 16.5.2016].
URL http://devmag.org.za/2009/04/25/perlin-noise/

[3] *Perlin Noise*. [Online; navštíveno 16.5.2016].
URL http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

[4] *Voxel Based Visual Hulls*. Duben 2012, [Online; navštíveno 16.5.2016].
URL http://willgallia.com/#!opencv

[5] *Atomontage Engine*. 2016, [Online; navštíveno 16.5.2016].
URL http://www.atomontage.com/

[6] *Massive Software - Simulating Life*. 2016, [Online; navštíveno 16.5.2016].
URL http://www.massivesoftware.com/

[7] *SpeedTree Animated Trees & Plants Modeling & Render Software*. 2016, [Online; navštíveno 16.5.2016].
URL http://www.speedtree.com/

[8] Cozzi, P.; Riccio, C.: *OpenGL Insights*. CRC Press, Červenec 2012, ISBN 14-398-9376-4.

[9] Hull, T. E.; Dobell, A. R.: *Random Number Generators*. *SIAM Review*, ročník 4, č. 3, Červenec 1962.

[10] Ken Perlin: *An Image Synthesizer*. *SIGGRAPH Comput. Graph.*, ročník 19, č. 3, Červenec 1985.

[11] Laine, S.; Karras, T.: *Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation*. Technická zpráva, NVIDIA Corporation, Únor 2010.

[12] Perlin, K.: *Improving Noise*. *ACM Trans. Graph.*, ročník 21, č. 3, Červenec 2002.

[13] Pharr, M.; Fernando, R.: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (GPU Gems)*. Addison-Wesley Professional, 2005, ISBN 0321335597.

[14] Revelles, J.; Ureña, C.; Lastra, M.: *An Efficient Parametric Algorithm for Octree Traversal*. In *Journal of WSCG*, 2000.

Přílohy

Seznam příloh

A Obsah CD **29**

B Ovládání aplikace **30**

Příloha A

Obsah CD

Přiložené CD obsahuje:

- Technická dokumentace – Zdrojový text v jazyce LaTeXa výsledná dokumentace ve formátu PDF.
- Aplikace – Zdrojové kódy napsané v jazyce C++ a GLSL, použité knihovny v příslušných verzích, soubory pro překlad systémem CMake a spustitelný soubor s příklady spuštění.
- Readme – Popis spuštění a ovládání aplikace
- Video demonstrující chod aplikace

Příloha B

Ovládání aplikace

Akce	Klávesnice + Myš	Ovladač
Pohyb	WASD	Levá páčka
Nahoru a dolů	Mezerník a Control	LB a RB
Kamera	Myš + Levé tlačítko myši	Pravá páčka
Změna LoD	Q a E	D-Pad Nahoru a dolů
Editace	Myš + Pravé tlačítko myši	A
Změna režimu	Tabulátor	Y
Rychlá editace	Shift	
Ukončení	Escape	B