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

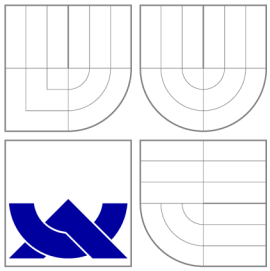
IMPLEMENTACE ALGORITMU LOD TERÉNU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

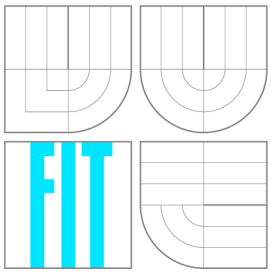
AUTOR PRÁCE
AUTHOR

Bc. PŘEMEK RADIL

BRNO 2012



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

IMPLEMENTACE ALGORITMU LOD TERÉNU

TERRAIN LOD ALGORITHM IMPLEMENTATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PŘEMEK RADIL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK BARTOŇ

BRNO 2012

Abstrakt

Tato práce pojednává o implementaci algoritmu pro LoD vizualizaci terénu Seamless Patches for GPU-Based Terrain Rendering jako rozšíření knihovny Coin3D. Prezentuje postupy, za pomoci kterých tento algoritmus zobrazuje rozsáhlé terénní datasety. Celý terén je složen z plátů, které jsou uloženy v hierarchické struktuře. Hierarchie plátů je pak za běhu programu procházena jsou z ní generovány aktivní pláty na základě pozice pozorovatele. Každý plát se skládá z předem definovaných dlaždic a spojovacích pruhů, takže nemusí udžovat žádnou konkrétní geometrii. Během vykreslování dlaždic a pruhů je aplikován displacement shader. Práce zhodnocuje výsledky dosažené implementací a navrhuje další úpravy, kterými by se dal běh algoritmu dále vylepšit.

Abstract

This thesis discusses implementation of LoD terrain visualization algorithm Seamless Patches for GPU-Based Terrain Rendering as extension for Coin3D library. It presents procedures which this algorithm uses for displaying large terrain datasets. Entire terrain is composed of patches that are stored in patch hierarchy. Patch hierarchy is traversed during runtime to generate active patches based on observer's position. Each patch consists of predefined tiles and connection strips so it doesn't need to store any geometry. During render of tiles and strips, displacement shader is applied. This thesis also evaluates results achieved in sample application and suggests some modifications to further increase algorithm performance.

Klíčová slova

LoD, Úroveň detailu, Terén, Bezešvé pláty, Coin3D, OpenGL, GLSL Displacement

Keywords

LoD, Level of Detail, Terrain, Seamless patches, Coin3D, OpenGL, GLSL, Displacement

Citace

Přemek Radil: Implementace algoritmu LoD terénu, diplomová práce, Brno, FIT VUT v Brně, 2012

Implementace algoritmu LoD terénu

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Radka Bartoňe. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Přemek Radil
22. května 2012

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce, panu Ing. Radku Bartoňovi za všechny rady, trpělivost při konzultacích a za veškerou pomoc, kterou mi byl ochoten poskytnout a rodině za podporu a pomoc, kterou mi v průběhu psaní práce poskytovala.

Obsah

Kapitola 1

Úvod

Satelitní a letecká snímací technika se stále zlepšuje a s ní se zvyšuje i rozlišení snímaných terénních datasetů. K zobrazení takových velkých objemů dat v reálném čase za přijatelné rychlosti a kvality je potřeba využít postupy, které ovlivňují kvalitu zobrazovaných dat v závislosti na pozici pozorovatele.

Tato práce se zabývá právě takovými postupy, které dokáži zobrazovat terén tak, aby byl pozorovatel schopen vnímat detaily v okolí a přitom nebylo vykreslování příliš pomalé. Programovým výstupem této práce je ukázková apliakce v níž je implementován jeden z vybraných algoritmů.

V první kapitole si ukážeme některé algoritmy pro zobrazování rozsáhlých terénů, počínaje algoritmem Seamless Patches for GPU-Based Terrain Rendering, který jsem si vybral pro implementaci jako programový výstup práce. Následuje přehled některých knihoven vhodných pro implementaci těchto algoritmů. V další kapitole pak nalezneme návrh implementace obsahující přehled tříd, diagram tříd a diagram použití takové, jak jsem si je představoval ve výsledném programu. Výsledná ukázková aplikace je napsána v prostředí knihovny Coin3D a provázána s uživatelským rozhraním v Qt přes knihovnu SoQt. V kapitole věnující se přímo implementaci jsou detailně popsány jednotlivé třídy, změny oproti původnímu návrhu, problémy, na které jsem během implementace narazil a optimalizace, které jsem provedl. Závěrečná kapitola zhodnocuje výsledky implementace ukázkové aplikace formou srovnávacích tabulek s počty snímků za sekundu, ukazuje snímky obrazovky a navrhuje některé kroky, kterými by se daly dosažené výsledky zlepšit.

Kapitola 2

Vybrané algoritmy zpracování LoD terénu

V této kapitole si předvedeme některé postupy pro vykreslování rozsáhlých terénů s použitím úrovní detailů v závislosti na pozici pozorovatele.

2.1 Seamless Patches for GPU-Based Terrain Rendering

Základem pro text této sekce je [9] a pro více detailů o algoritmu Seamless Patches for GPU-Based Terrain Rendering doporučuji nahlédnout do zmíněné publikace.

2.1.1 Přehled

Tento algoritmus představuje zobrazování velkých terénů za pomoci rozdělení terénu na čtyřúhelníkové pláty (patches) s různým rozlišením. Každý plát je reprezentován čtyřmi trojúhelníkovými dlaždicemi, z nichž každá může mít také různé rozlišení, a čtyřmi pruhy, které souvisle spojují tyto dlaždice. Za běhu programu jsou pak pláty použity ke zkonstruování úrovně detailu na základě parametrů pohledu. Zvolená úroveň detailu pak obsahuje rozmístění plátů a rozlišení na hranách těchto plátů. Protože sousedící pláty mají stejná hraniční rozlišení, nevznikají tak žádné mezery nebo vadné trojúhelníky. Na grafické kartě se pak generuje síť plátů použitím instancí předdefinovaných dlaždic uložených v paměti v různém měřítku a přiřazením výškového stupně jednotlivým vertexům z výškových textur.

2.1.2 Schéma plátů

Některé algoritmy používají renderování terénu pomocí čtyřúhelníkových nebo trojúhelníkových plátů. Algoritmy využívající čtyřúhelníkové pláty přidělují celému plátu pevné rozlišení a tak omezují lokální přizpůsobivost a způsobují mnoho obtíží při spojování přilehlých plátů. Na druhou stranu algoritmy, které využívají trojúhelníkové sítě, umožňují lehčí spojování a lepší přizpůsobivost, trpí ale nekompatibilitou s texturovacím rozhraním, které je obdélníkové. Tento algoritmus kombinuje výhody obou přístupů. Rozděluje terén na čtyřúhelníkové pláty, které sestávají z trojúhelníkových dlaždic (dále jen dlaždic) a umožňují tak více rozdílných rozlišení uvnitř jednoho plátu. To umožňuje jistou lokální adaptabilitu a bezešvé spoje přilehlých plátů (viz 2.1). Počet rozlišení jednotlivých dlaždic je předem definován a bývá relativně malý (většinou se pohybuje mezi dvěma a čtyřmi úrovněmi) a

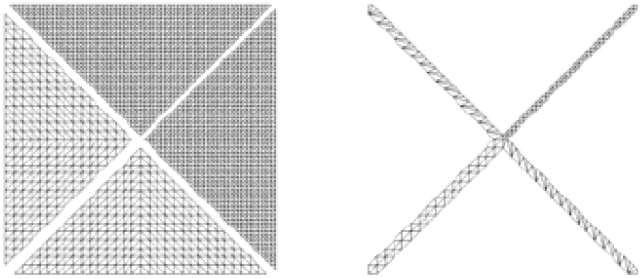
tak i počet jednotlivých spojovacích proužků bývá nízký (pro tři různá rozlišení dlaždic je třeba šest proužků).

2.1.3 Hierarchie plátů

Hierarchie plátů je konstruována odshora dolů dělením každého plátu na $R \times R$ potomků, kde R je větvící faktor hierarchie. Větvící faktor je určen počtem různých rozlišení pro dlaždice a je roven mocnině dvou o velikosti rozdílu mezi nejmenším a největším rozlišením.

$$R = 2^{R_{max} - R_{min}}$$

Například pro 2 různá rozlišení bude větvící faktor 2 a pro 3 rozlišení bude roven 4. To zajišťuje bezešvé spojení mezi přilehlými pláty s absencí děr. Hierarchie plátů neukládá geometrii. Místo toho ukládá pozici a dimenzi každého plátu vzhledem k terénu. Vejde se tedy snadno do lokální paměti i pro velmi rozlehlé terény.



Obrázek 2.1: Komponenty jednoho plátu: čtyři dlaždice (vlevo) a čtyři pruhy sloužící ke spojení dlaždic o různém rozlišení. Zdroj: [9]

2.1.4 Běh programu

Za běhu programu je hierarchie plátů použita k výběru různých úrovní detailu v závislosti na parametrech pohledu. Pro každý snímek je hierarchie plátů procházena odshora dolů pro výběr aktivních plátů, které vytvoří požadovanou úroveň detailu. Procházení začíná v kořenu a pro každý průchozí plát τ je spočtena míra chyby. Pokud je chyba příliš velká vzhledem k parametrům pohledu, potomci plátu τ jsou dále procházeni. Jinak jsou vypočítána rozlišení na okrajích plátu a plát je přidán mezi aktivní pláty. Polygonální sítě reprezentující jednotlivá rozlišení dlaždic jsou uložena v paměti textur. Pro každý frame jsou aktivní pláty odeslány na grafickou kartu pro rendering. Tato reprezentace aktivních plátů dramaticky redukuje komunikaci CPU-GPU. Rozlišení na okrajích plátů je disktretizováno aby souhlasilo s předdefinovanými dlaždicemi. To stačí k rozhodnutí, které dlaždice a spojovací pruhy se použijí k reprezentaci dlaždice τ. Nacachované instance vybraných dlaždic a pruhů jsou transformovány, aby vyplňovaly daný plát bez nutnosti znalosti okolních plátů. Protože rozlišení je společné pro okraje každých dvou sousedních plátů, nevznikají žádné trhliny ani vadné trojúhelníky. Vydlaždicování plátů pak vyprodukuje rovinnou plochu bez výškových rozdílů a barev. Tyto jsou přiřazeny pro každý vrchol ve vertex a pixel shaderech, které používají dvourozměrné koordináty daného vrcholu pro určení výškové úrovně a barvy z nacachovaných textur.

2.1.5 Úroveň detailu

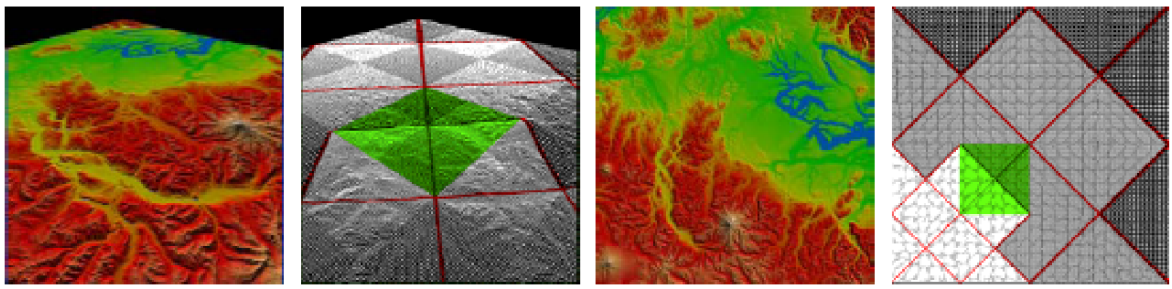
Úroveň detailu každého plátu je určena rozlišením jednotlivých dlaždic, které je dáno rozlišením okrajů plátu. Rozlišení okrajů plátu je spočteno na základě délky hrany l a vzdálenosti od kamery d s použitím rovnice

$$\varepsilon = \rho \frac{l}{d}$$

kde ρ je faktor přesnosti. Pokud je ε větší než 1, je plát rozdělen na potomky, jinak je rozlišení určeno jako εR_{max} a zaokrouhleno na nejbližší rozlišení, kde R_{max} je nejvyšší možné rozlišení.

2.1.6 Pyramida textur

Datasety terénu jsou většinou reprezentovány výškovou mapou a barevnou texturou, které uchovávají vlastnosti vrcholů v původním terénu. Tento algoritmus používá víceúrovňové pyramidy textur, kde každá další úroveň je mocninou dvou (podobně jako u mipmap), pro podporu zobrazení úrovní detailu. Tyto pyramidy jsou použity za běhu programu k získání věrohodných vzorků textur pro vrcholy v každé dlaždici. Protože jsou tyto pyramidy podobné mipmapám, můžeme je nechat zkonstruovat v hardware. Poté za běhu programu vertex shader rozhodne, kterou úroveň detailu použít pro výběr hodnot. Pro rozlehlé terény jsou víceúrovňové pyramidy zkonstruovány na CPU před tím, než jsou přesunuty do texturovací paměti. Pro každou úroveň je rozlišení na obou dimenzích sníženo na polovinu a hodnota pixelu je interpolována ze 4 odpovídajících pixelů předchozí úrovně.



Obrázek 2.2: Renderování terénu pomocí tohoto algoritmu. Zelený region označuje jeden plát. Zdroj: [9]

2.1.7 Zhodnocení

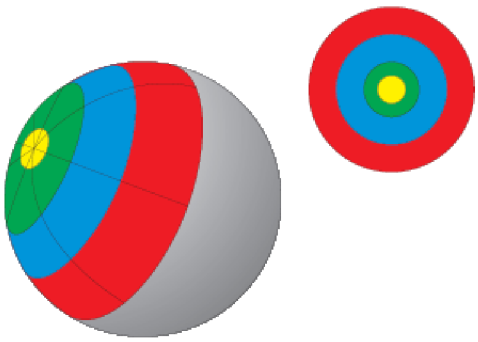
V porovnání se staršími algoritmy, kde se přenášely na grafickou kartu všechny tři souřadnice vrcholu, dochází u tohoto algoritmu ke značnému snížení objemu přenesených dat (přenáší se pouze výška, pozice se určuje přímo ze zpracovávaného vertexu). Vykreslování je proto velmi rychlé i pro rozlehlé terény.

2.2 Terrain Rendering Using Spherical Clipmaps

Základem pro text této sekce je [3] a pro více detailů o algoritmu Terrain Rendering Using Spherical Clipmaps doporučuji nahlédnout do zmíněné publikace.

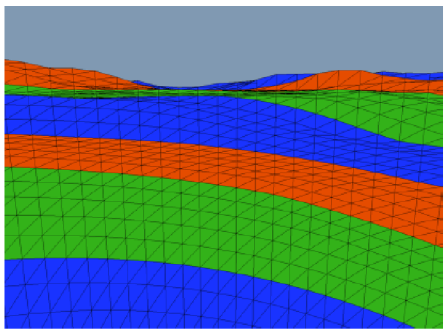
2.2.1 Přehled

Cílem tohoto algoritmu je zobrazování kulovitých terénů (například celých planet) v mnoha různých velikostech (ať už z pohledu z vesmíru, nebo při procházce po povrchu) na běžných výpočetních systémech. Většina populárních algoritmů pro vykreslování sférických těles má základ v algoritmech pro rovinné terény (adaptace algoritmů ROAM a BDAM), které rozdělují kouli na čtvercové regiony. Základ tohoto algoritmu leží v Geometrických Clipmapách, které byly původně určeny pro rovinné terény. Čtyřúhelníkové plochy původních Geometrických Clipmap však byly nahrazeny kruhovými, aby mohly rovnoměrně pokrývat celou hemisféru. Hustota ploch se zvyšuje směrem k pólu, který vždy směřuje k pozorovateli (2.3).



Obrázek 2.3: Rozdělení polokoule na kruhové plochy v závislosti na vzdálenosti od pólu. Zdroj: [3]

Kvůli použití kruhových ploch je však třeba používat jiný systém koordinátů, než pro rovinné algoritmy. Konkrétně se jedná o sférické koordináty určené úhlem náklonu od osy z a úhlem nad plochou x, y. Kruhové plochy jsou pak rozděleny na quady, jejichž velikost závisí na vzdálenosti od pozorovatele. Velikost quadů je nastavena tak, aby v prostoru obrazovky měli přibližně stejnou velikost (2.4). To umožňuje využít čtvercových textur, které jsou přirozené pro současné GPU, a lehce namapovat například výškovou mapu na hemisféru. Ve výsledné geometrii jsou quady reprezentovány dvěma trojúhelníky.



Obrázek 2.4: Velikost trojúhelníku v prostoru obrazovky. Zdroj: [3]

2.2.2 Rekapitulace zásadních informací

- Zobrazovaná polokoule je rozdělena na kruhové plochy
- Hustota ploch se zvyšuje směrem k pólu, který vždy směřuje k pozorovateli
- Plochy jsou rozděleny na quady, jejichž velikost závisí na vzdálenosti od pólu
- Quady jsou rozděleny na dvojice trojúhleníků

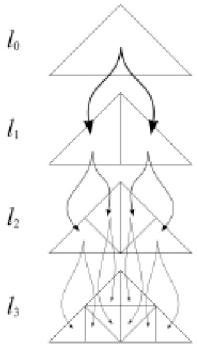
2.3 BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization

Základem pro text této sekce je [2] a pro více detailů o algoritmu Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization doporučuji nahlédnout do zmíněné publikace.

2.3.1 Přehled

Podobně jako v algoritmu ROAM, BDAM využívá hierarchii pravoúhlých trojúhelníků uložených jako binární strom (viz. 2.5), avšak jednotlivé trojúhelníky jsou nahrazeny předpočítanou sadou trojúhelníků (batch, odtud název) sestávající ze stovek trojúhelníků (v rozsahu 256 až 8000). Každá sada obsahuje geometrickou chybu, která se mění mezi úrovněmi stromu. V hierarchii pravoúhlých trojúhelníků (HTR) každý trojúhelník může být navázán na:

- trojúhelník stejné úrovně
- trojúhelník hrubší úrovně skrze svoji nejdelší hranu
- trojúhleník jemnější úrovně skrze dvě kratší hrany



Obrázek 2.5: Binární strom pravoúhlých trojúhelníků. Zdroj: [2]

Textury jsou na rozdíl od geometrie nejlépe spravovány jako čtyřúhelníkové dalždice, což vede k zavedení kvadrantového stromu. Pokud originální textura pokrývá stejnou rozlohu jako výšková data, každý element v kvadrantovém stromu pak odpovídá dvěma přilehlým elementům z binárního stromu geometrie. Dále pak postoupení o jednu úroveň v kvadrantovém stromu textur odpovídá sestupu o dvě úrovně v binárním stromu.

2.3.2 Rekapitulace zásadních informací

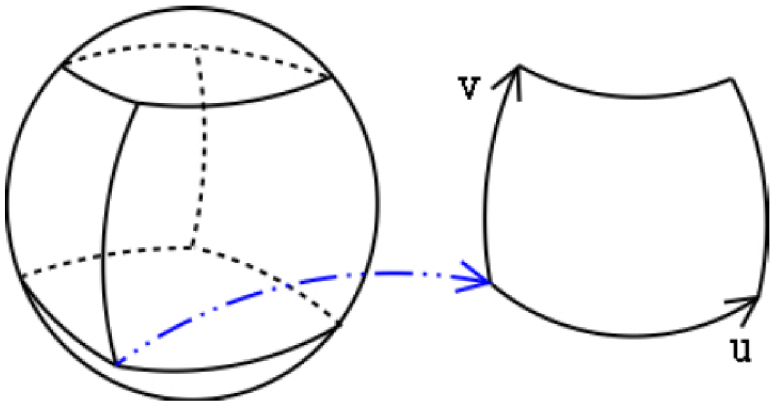
- Pro reprezentaci geometrických dat používá hierarchii pravoúhlých trojúhelníků, jejíž listy tvoří předpočítané pláty stovek trojúhelníků.
- Každý trojúhelník v hierarchii může sousedit s:
  - trojúhelníkem stejné úrovně
  - trojúhelníkem hrubší úrovně skrze svoji nejdelší hranu
  - trojúhleníkem jemnější úrovně skrze dvě kratší hrany
- Každý plát obsahuje geometrickou chybu měnící se mezi úrovněmi stromu.
- Textury jsou reprezentovány pomocí kvadrantového stromu.

2.4 Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM)

Základem pro text této sekce je [1] a pro více detailů o algoritmu Planet-Sized Batched Dynamic Adaptive Meshes doporučuji nahlédnout do zmíněné publikace.

2.4.1 Přehled

Tak jako algoritmus BDAM i P-BDAM využívá jako nejmenší zobrazovací primitivum sadu složenou ze stovek trojúhelníků. Tento algoritmus však na rozdíl od BDAM umožňuje zobrazování terénů, které nejsou na rovinné ploše (například celé planety). Povrch planety je nejprve rozdělen na čtvercové dlaždice (viz. 2.6). Narozdíl od BDAM se tedy zpracovává celý hierarchický les místo pouze jednoho stromu. Dlaždice mají přiřazeny parametry (u, v), které jsou využity k sestavení hierarchie a také pro texturové koordináty. Počet a velikost těchto dlaždic závisí na velikosti původního datasetu.



Obrázek 2.6: Rozdělení sférického tělesa na dlaždice. Zdroj: [1]

Jsou zde však jistá omezení: čísla s plovoucí řádovou čárkou s přesností na jedno desetinné místo musí být dostatečně přesné na reprezentaci lokálních koordinátů (tedy nesmí být více než 2^{23} pozic na každé ose) a velikost gerované struktury se musí vejít do limitů určených operačním systémem (většinou méně než 3GB dat).

2.4.2 Rekapitulace zásadních informací

- Umožňuje zobrazování neplošných terénů jako jsou například planety.
- Jako u BDAM je nejmenší zobrazovací primitivum sada stovek trojúhelníků.
- Povrch je rozdělen na čtvercové dlaždice.
- Dlaždice mají parametry (u, v), které jsou využity k sestavení hierarchie a pro texturové koordináty.

2.5 C-BDAM - Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering

Základem pro text této sekce je [5] a pro více detailů o rozšíření Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering doporučuji nahlédnout do zmíněné publikace.

2.5.1 Přehled

Jedná se o rozšíření k algoritmům BDAM a P-BDAM, které přináší zrychlení vykreslování zjednodušením datové struktury na základě reprezentace atributů vrcholů pomocí rozdílu od hodnoty predikované z nižší úrovně detailu. Dále poskytuje mimo jiné celkovou spojitost geometrie pro rovinné i sférické terény, efektivní kompresi pomocí waveletové transformace, rychlou konstrukci a real-time dekompresi. Vyžaduje poměrně dlouhý preprocessing přináší však významné snížení objemu dat a zvýšení rychlosti.

	Sample #	XY Step	Tolerance	Time	Output Size	bps	Rate
Puget Sound	265 M	10 m	1 m RMS	11 m	9.1 MB	0.28	57:1
Puget Sound	265 M	10 m	1 m AMAX	10 m	19.2 MB	0.61	26:1
Earth SRTM	29 G	90 m	16 m AMAX	30 h 38 m	869.9 MB	0.25	64:1
Paris	67 M	1 m	0.1 m AMAX	6 m	14.6 MB	1.80	9:1
Paris color	872 M	0.25 m	10/255 AMAX	1 h 17 m	254.2 MB	2.45	10:1

Obrázek 2.7: Tabulka výsledků preprocessing fáze CBDAM pro různé datasety. Zdroj: [5]

2.5.2 Rekapitulace zásadních informací

- Rozšíření BDAM a P-BDAM.
- Reprezentace vrcholů pomocí predikce z nižší úrovně detailu.
- Významná komprese objemu dat.
- Poměrně dlouhý preprocessing.

2.6 GPU-Friendly High-Quality Terrain Rendering

Základem pro text této sekce je [10] a pro více detailů o algoritmu GPU-Friendly High-Quality Terrain Rendering doporučuji nahlédnout do zmíněné publikace.

2.6.1 Přehled

Tato metoda vykreslování terénu kombinuje přístupy souvislého LoD a diskrétní LoD hierarchie a vyhýbá se tak opakované trinagulaci za běhu programu. Aliasu předchází používáním optimální filtrace geometrie na co nejlepším možném rozlišení. Za běhu algoritmu jsou diskrétní sady decimovaných polygonálních sítí přesouvány postupně, čímž dochází k velké efektivitě využití šířky pásma.

Oblast je nejprve rozdělena na sady stejně velkých dlaždic. Pro každou dlaždici je pak spočítána diskrétní sada úrovní detailu prostřednictvím hierarchie polygonálních sítí. Pro každou úroveň je terén decimován podle dané world-space chyby, což snižuje celkový počet trojúhelníků. Každá úroveň je reprezentována vertexy hrubší úrovně a vertexy dané úrovně, což umožňuje jejich postupný přenos na grafickou kartu. Pro texturování se pak využívají mipmapy s vysokým rozlišením.

2.6.2 Rekapitulace zásadních informací

- Oblast je rozdělena na dlaždice.
- Pro každou dlaždici je spočítána sada úrovní detailu.
- Každá úroveň detailu je reprezentována vertexy dané úrovně a vertexy s nižší úrovní detailu.
- Texturování pomocí mipmap s vysokým rozlišením.

2.7 Real-Time Optimal Adaptation for Planetary Geometry and Texture: 4-8 Tile Hierarchies

Základem pro text této sekce je [6] a pro více detailů o algoritmu Real-Time Optimal Adaptation for Planetary Geometry and Texture: 4-8 Tile Hierarchies doporučuji nahlédnout do zmíněné publikace.

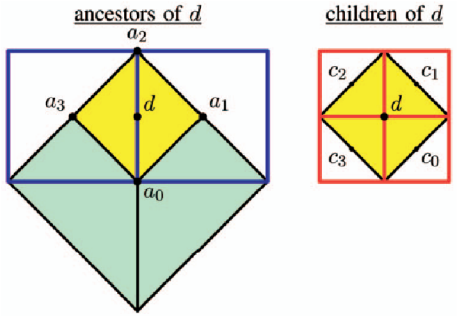
2.7.1 Přehled

Tento algoritmus používá pro geometrii strukturu diamantů, kde každý diamant má přiřazen jeden vertex (středový), jednu hranu a jednu čtyřúhelníkovou plochu. Každý diamant má čtyři předky a čtyři potomky. Předkem je myšlen diamant s nižší úrovní detailu, jehož plocha překrývá plochu původního diamantu. Potomek je pak diamant s vyšší úrovní detailu, který je překrýván původním diamantem.

Podobně jako u algoritmu ROAM se používají dvě fronty, jedna pro slučování a jedna pro rozdělování diamantů. Pro aplikaci textur je původní dataset rozřezán na dlaždice o pevné velikosti, které reprezentují texturu v nejlepším rozlišení. Poté je aplikován filtr typu dolní propusť. Pro každý zobrazovaný trojúhelníkový plát je vypočítáno optimální rozlišení textury.

2.7.2 Rekapitulace zásadních informací

- Struktura diamantů.
- Každý diamant má přiřazen vertex, hranu a čtyřúhelníkovou plochu.



Obrázek 2.8: Diamant d a jeho předci a potomci. Zdroj: [6]

- Každý diamant má 4 předky a 4 potomky.
- Dvě prioritní fronty pro rozdělování a spojování diamantů.

Kapitola 3

Knihovny vhodné ke zpracování

V této kapitole si předvedeme některé knihovny, pomocí kterých by se daly výše popsané algoritmy implementovat.

3.1 OpenGL

OpenGL (Open Graphics Library) bylo vytvořeno jako open-source alternativa k IrisGL, grafickému API společnosti Silicon Graphics, se kterým si bylo v mnohém podobné. První verze OpenGL byla vydána roku 1992. Roku 1998 bylo OpenGL revidováno konsorciem firem OpenGL Architecture Review Board (ARB). Nyní je OpenGL průmyslovým standardem specifikujícím multiplatformní rozhraní pro tvorbu aplikací počítačové grafiky. Více k historii OpenGL lze najít na [7].

Knihovna OpenGL je multiplatformní a nezávislá na správcích oken. Proto neobsahuje žádné funkce pro vytváření grafického rozhraní. Grafické rozhraní je pro OpenGL třeba vytvořit samostatné přímo přes správce oken popřípadě přes některou z nadstaveb jako je například GLUT (OpenGL Unitility Toolkit). OpenGL se dělí na klientskou a serverovou část a lze jednotlivé vykreslovací příkazy přenášet přes síťové rozhraní. Hlavičkové soubory OpenGL jsou k dispozici pro mnoho programovacích jazyků a jsou v nich definovány vlastní datové typy (GLdouble, GLint a další) kvůli platformové nezávislosti. OpenGL samo o sobě poskytuje pouze základní přístup ke grafickým akcelerátorům, ale existují knihovny které jeho funkcionalitu rozšiřují. Jednou z nejpoužívanějších knihoven je knihovna GLU (OpenGL Utilities), která umožňuje například generování mipmap pro textury nebo automatické transformování scény, aby byla viděna z určité pozice.

OpenGL se z programátorského hlediska chová jako stavový automat. Pokud se během příkazů pro vykreslování změní vlastnosti vykreslované scény nebo jednotlivých primitiv, zůstane toto nastavení zachováno až do jeho další změny, popřípadě do konce běhu programu. Výsledkem vykreslovacích příkazů OpenGL je rastrový obrázek, který je uložen do framebufferu a zobrazen na obrazovce, popřípadě do framebuffer objektu, kde s ním lze dále pracovat.

Existuje množství knihoven které fungují jako (převážně objektová) nadstavba nad OpenGL. Knihovna OpenGL s rozšířeními by šla pro zpracování ukázkové aplikace použít bez větších potíží. Níže si uvedeme některé z knihoven, které na vyreslování používají právě OpenGL.

3.2 Coin3D

Coin3D je objektově orientovaná knihovna napsaná v C++ tvořící nadstavbu nad vykreslováním pomocí OpenGL. Byla vyvinuta norskou společností SiM (Systems in Motion). Je plně kompatibilní s Open Inventorem, se kterým nesdílí žádný kód, ale z důvodů kompatibility má stejné API. Coin3D je distribuován jako open source (s možností nákupu komerční vývojářské licence). Často se využívá malými i velkými společnostmi pro vizualizaci širokého spektra aplikací, jako například CAD, zobrazení medicínských dat či prezentací. Více o knihovně Coin3D lze nalézt na [8].

Pomocí knihovny Coin3D lze vytvářet celé scény pomocí hierarchicky navázaných objektu a kontruovat tak graf scény. Jednotlivé uzly grafu scény pak ovlivňují stav vykreslování OpenGL, který má vliv na další uzly, nebo za aktuálního stavu vykreslují celé objekty. Během vykreslování je pak procházen celý graf scény postupně od kořene k listům a způsob vykreslování ovlivňují všechny předchozí uzly grafu. Scény v Coin3D lze pak vytvářet buď programově uvnitř aplikace nebo lze použít textového souboru, který lze načíst v programu jako scénu a z něj je vytvořen graf scény.

Pro vytvoření grafického rozhraní pro vykreslování pomocí knihovny Coin3D lze využít knihovnu SoQt, která pro jednotlivé prvky rozhraní a okno aplikace využívá knihovnu QT, nebo SoWin, která využívá správce oken ve Windows (takové rozhraní samozřejmě narozdíl od QT není multiplatformní).

3.3 OpenSceneGraph

OpenSceneGraph (OSG) je open-source 3D grafický toolkit určený pro vývoj aplikací jako jsou simulace, hry, virtuální realita, vědecké simulace a modelování. Je napsán v C++ a OpenGL a běží pod Windows, OSX, GNU/Linux, a dalšími operačními systémy. Projekt Scene Graph (SG) byl založen zaměstnancem Silicon Graphics, Inc Donem Burnsem jako systém pro zpracování grafu scény pro Linux. Jeho ideou bylo vytvořit SG jako open-source společně s kolegou Robertem Osfieldem. Don však později projekt opustil a byl přejmenován na OpenSceneGraph. Tým pracující na OSG se postupně rozrůstal a také samotný toolkit se rozšiřoval a nyní je OSG používáno mnoha aplikacemi pro vykreslování 2D a 2D scén hlavně na polích počítačových her, grafických informačních systémů a modelování. Více o OpenSceneGraphu lze nalézt na [4].

OpenSceneGraph stejně jako Coin3D udržuje graf scény který je během vykreslování scény procházen a jeho uzly ovlivňují stav OpenGL nebo vykreslují jednotlivé objekty. Narozdíl od Coin3D lze však aplikaci spustit samostatně jako konzolovou s oknem pro vykreslování nebo navázat třeba na QT pokud si přejeme čistě okenní aplikaci.

3.4 Ogre

OGRE (Object-Oriented Graphics Rendering Engine) je na scénu orientovaný, flexibilní 3D engine napsaný v C++ navržený pro zjednodušení a intuitivní práci pro vývojáře 3D grafických aplikací akcelerovaných hardwarem. Tuto knihovnu lze navázat jak na OpenGL tak na Direct3D. S vývojem OGRE začala společnost Sinbad rozšířením projektu DIMClass (založeným na Direct3D) tak, aby byl nezávislý na použité platformě a API. Později přináší podporu shaderů, HDR a dalších technik a připojuje další platformy jako Linux nebo iPhone. Více o knihovně lze nalézt na [11].

Kapitola 4

Návrh implementace

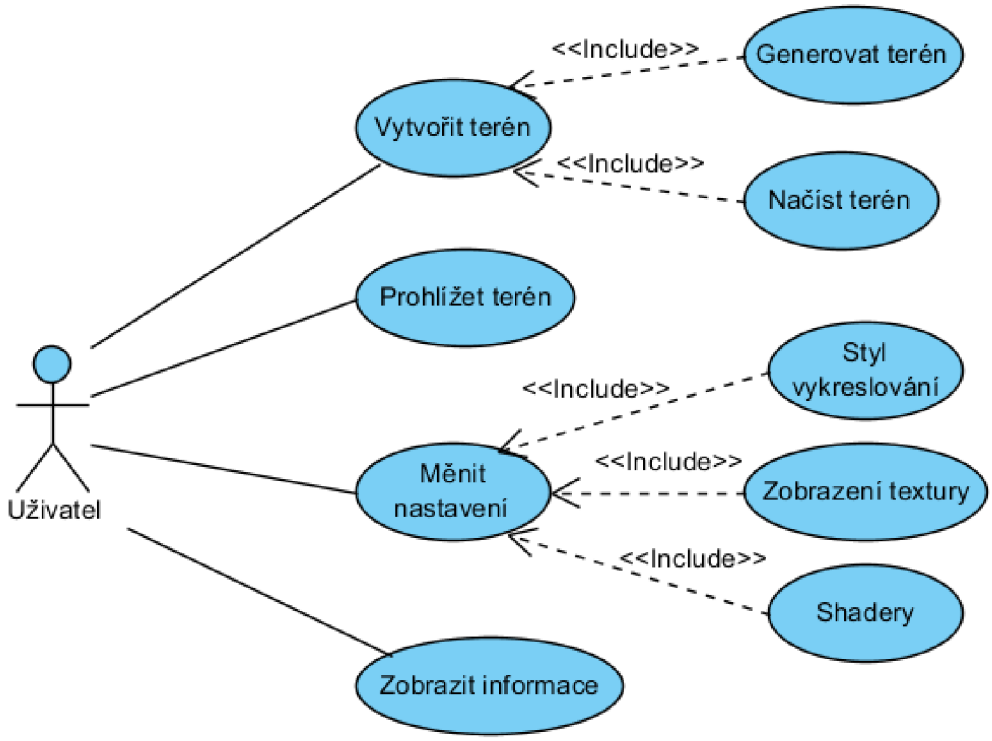
Tato kapitola pojímá informace o zvoleném algoritmu a knihovně, ve které jsem se rozhodl tento algoritmus zpracovat. Následuje návrh aplikace ve formě diagramu použití a diagramu tříd.

4.1 Zvolený způsob řešení

Pro řešení implementace LoD algoritmu pro zobrazování terénu jsem si vybral algoritmus Seamless Patches for GPU-Based Terrain Rendering. Rozhodl jsem se ho implementovat v programovacím jazyce C++ s využitím knihovny Coin3D navázanou přes SoQT na knihovnu Qt kvůli možnosti spuštění programu na různých platformách. S žádnou z výše uvedených knihoven jsem neměl osobně příležitost pracovat, avšak práci s knihovnou OpenSceneGraph jsem viděl u spolužáka. Knihovna Coin3D mi byla doporučena jako vhodná ke zpracování tohoto úkolu a protože to bylo něco nového, rozhodl jsem se ukázkovou aplikaci implementovat za použití právě této knihovny.

4.2 Diagram použití

Hlavní funkcí celé ukázkové aplikace by mělo být zobrazení terénu v úrovni detailu závislé na vzdálenosti dané části terénu od pozice kamery, tedy prohlížení terénu. Dále by bylo vhodné moci například měnit styl vykreslování, vypínat zobrazování textury čí vypínat a zapínat shadery. Jelikož jsou shadery důležitou součástí algoritmu, nejsem si jist, zda by to bylo žádoucí, protože aplikace by pak zobrazovala pouze rovnou plochu. Aplikace by také měla poskytovat nějaké informace o vykreslované scéně jako počet snímků za sekundu či počet zobrazovaných vrcholů, aby bylo vidět jakého výkonu při zobrazování dosahuje. Zde by se mohl vystkytnout problém s vypnutím double bufferingu, který omezuje maximální frekvenci vykreslování. Nejsem si jist zda a jak toho lze v prostředí Coin3D dosáhnout. Uživatel by mohl také načíst terén z výškové mapy či vygenerovat náhodně. Generování náhodného terénu by však vyžadovalo další netriviální postupy, které nejsou součástí této diplomové práce. Načtení výškové mapy bude pravděpodobně zpracováno formou načtení obrázku do textury.

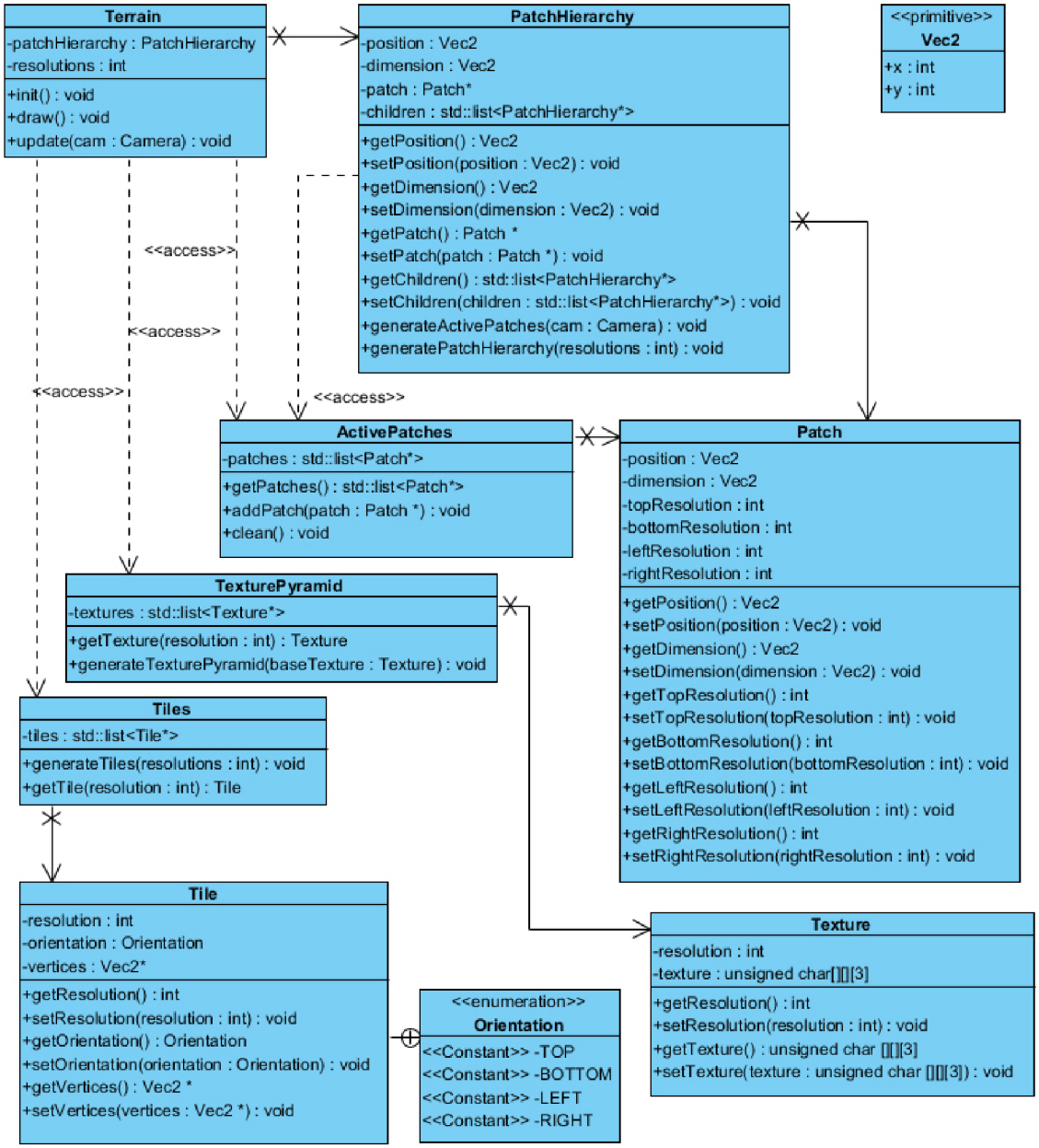


Obrázek 4.1: Use case diagram

4.3 Přehled navržených tříd

- Terrain
  - Hlavní třída pro zpracování algoritmu Seamless Patches for GPU-Based Terrain Rendering.
  - **update(Camera cam)** - tato metoda by na základě pozice kamery měla vytvořit z hierarchie plátů nový seznam aktivních plátů pro vykreslování
  - **draw()** - tato metoda by měla sloužit k vykreslení seznamu aktivních plátů
- PatchHierarchy
  - Třída sloužící k uchovávání a správě hierarchie polí. Generuje danou hierarchii na základě počtu různých rozlišení a generuje seznam aktivních polí pro vykreslení.
  - **generatePatchHierarchy()** - tato metoda by měla během inicializace vytvořit kompletní hierarchii plátů na základě počtu různých rozlišení dlaždi.c
  - **generateActivePacthes()** - metoda pro generování aktivních plátů na základě pozice kamery procházením vnořených hierarchií v poli **children**.
- Patch
  - Třída pro jednotlivé pláty. Uchovává v sobě informace o pozici, rozměrech a rozlišení jednotlivých dlaždic uvnitř plátu.

- Tile
  - Třída pro jednotlivé dlaždice. Uchovává rozlišení, orientaci a pole vrcholů pro různé varianty dlaždic.
- Texture
  - Třída pro jednotlivé úrovně textur. Uchovává rozlišení dané úrovně textury a její data.
  - **setTexture()** - metoda umožnující nastavení dat pro danou úroveň textury.
- TexturePyramid
  - Singleton třída pro správu pyramidy textur. Umožňuje vygenerovat textury s nižším rozlišením podobné mipmapám na základě originální textury a počtu různých rozlišení.
  - **generateTexturePyramid** - metoda generující pyramidu textur na základně textury nejvyšší úrovně. Každá následující úroveň bude mít poloviční velikost na obou hranách.
  - **getTexture()** - metoda vracející texturu na základě požadovaného rozlišení.
- ActivePatches
  - Singleton třída spravující seznam aktivních polí. Ten je během každého framu vyčištěn a znovu vygenerován pomocí PatchHierarchy.
  - **addPatch()** - metoda přidávající plát do seznamu aktivních plátů.
  - **getPacthes()** - metoda vracející seznam aktivních plátů pro vykreslení
  - **clean()** - vymaže seznam aktivních plátů
- Tiles
  - Singleton třída spravující seznam předdefinovaných dlaždic. Umožňuje během inicializace vygenerovat dlaždice a vrací dlaždice v závislosti na rozlišení.
  - **generateTiles()** - metoda která během inicializace vytvoří na základě počtu možných rozlišení geometrii pro jednotlivé dlaždice a spojovací pruhy.
  - **getTile()** - vrátí dalždici požadovaného rozlišení.



Obrázek 4.2: Diagram tříd

Kapitola 5

Implementace

Tato kapitola obsahuje popis práce v prostředí knihovny Coin3D, kompletní výpis všech tříd a jejich provázání ve výsledné ukázkové aplikaci. Při srovnání s původním návrhem aplikace výše, je vidět, že došlo k mírným změnám v implementaci v průběhu vývoje. Tyto změny jsou popsány v sekci 5.5 Změny vůči návrhu. Dále jsou v kapitole probírány problémy, na které jsem v průběhu implementace narazil a jejich řešení a také jednotlivé optimalizační kroky, které jsem v průběhu vývoje aplikace provedl a jaký měly vliv na celkový výkon aplikace.

5.1 Zvolené řešení

Jak je uvedeno výše, vybral jsem si pro zpracování algoritmus Seamless Patches for GPU-based terrain rendering. Aby bylo možné spustit výslednou aplikaci na více platformách, rozhodl jsem se použít knihovnu Qt pro uživatelké rozhraní a knihovnu Coin3D pro vykreslování. Pro komunikaci mezi těmito dvěma knihovnami slouží knihovna SoQt.

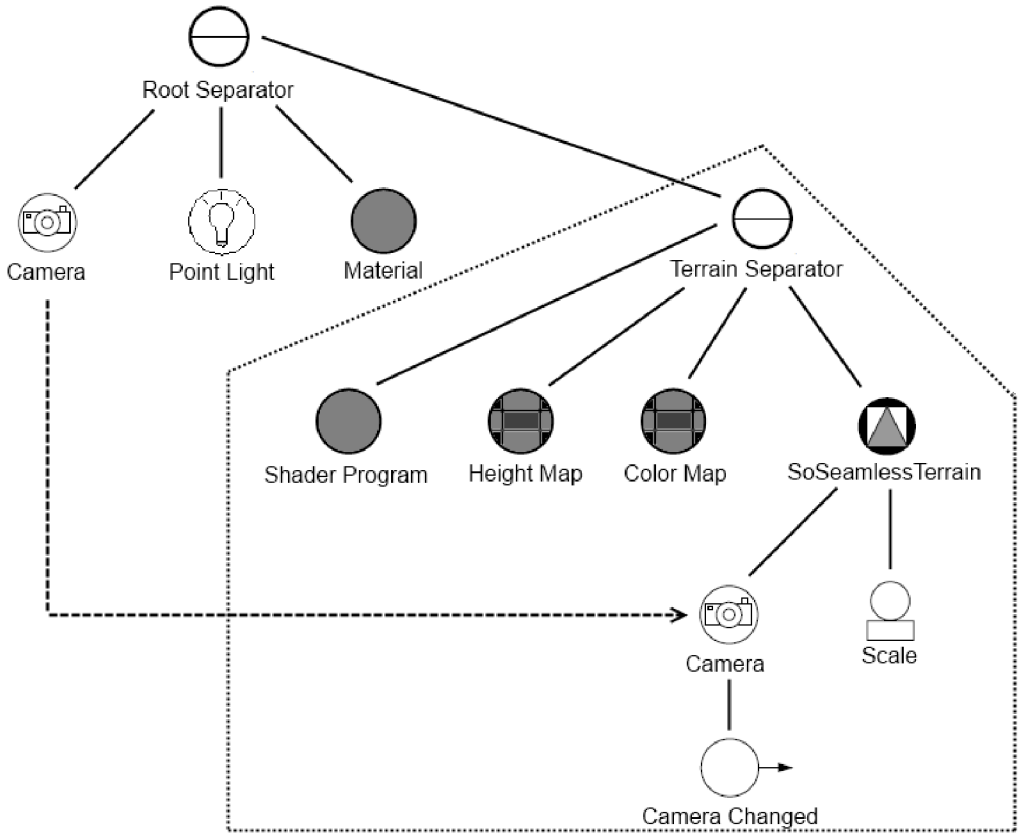
5.2 Prostředí Coin3D

Algoritmus je implementován ve třídě SoSeamlessTerrain, která je realizována jako rozšíření knihovny Coin3D a je tedy možné ji použít ve grafu scény. Bohužel vytváření nového uzlu pro prostředí Coin3D není na stránkách coinu [8] nijak dokumentováno. Řešení této problematiky lze však najít v publikaci The Inventor Toolmaker [13], ze které vychází tato sekce. Teoreticky by se dal algoritmus implementovat za pomocí některých již dostupných uzlů, například několika **SoIndexedFaceSet**y, které by se za běhu povolovali a zakazovali na základě parametrů kamery. Toto řešení by však přinášelo poměrně velké množství nadbytečné režie a nešlo by použít jako samostatný uzel. Uzly potřebné pro běh algoritmu jsou v grafu (5.1) označené tečkovanou čarou.

5.2.1 Tvorba nového shape uzlu

Tvorba nového tvarového (shape) uzlu vyžaduje implementaci několika reakcí na určité akce. Uzel potřebuje například v průběhu renderu vykreslovat svou geometrii (v tomto případě konkrétní sadu aktivních plátů), vykreslovat primitiva za účelem zjišťování průsečíku s paprskem a nastavovat správně rozměry při zjišťování obalového tělesa pro daný tvar.

Všechny nové tvary musejí definovat alespoň metody **generatePrimitives()** a **getBoundingBox()**. Po nadefinování metody **generatePrimitives()** lze podědit metody



Obrázek 5.1: Graf scény pro Seamless Patches.

GLRender() a **rayPick()** přímo ze třídy **SoShape**, protože ty standardně používají vygenerovaná primitiva. To šetří čas během prototypování, avšak je vhodné tyto metody reimplementovat za účelem zlepšení výkonu.

Generování Primitiv

Tvar z pohledu Coin3D může sestávat z trojúhelníků, linek nebo bodů. Informace o každém vertexu jsou uchovávány v instanci třídy **SoPrimitiveVertex** a pro každe vygenerované primitivum (trojúhelník, linka nebo bod) je zavolána příslušná metoda třídy SoShape. Například pokud tvar generuje trojúhelníky (například kužel **SoCone**), je zavolána pro každý vygenerovaný trojůhelník callback funkce pro trojůhelníky.

SoPrimitiveVertex

Každý **SoPrimitiveVertex** obsahuje všechny informace o vertexu:

- Koordináty
- Normálový vektor
- Texturové koordináty
- Index materiálu

- Ukazatel na **SoDetail** obsahující další informace (může být NULL)

Samotné vkládání vertexů je pak podobné OpenGL. Rodičovská třída SoShape poskytuje metodu **beginShape(action, shapeType)**, kde **shapeType** může nabývat hodnot **TRIANGLE_FAN**, **TRIANGLE_STRIP**, **TRIANGLES** nebo **POLYGON**. Následuje výčet **SoPrimitiveVertex**ů vložených přes metodu **shapeVertex()** a nakonec metoda **endShape()**.

Renderování

Pro renderování lze využít již nadefinované metody **generatePrimitives()**, ale v tomto případě bude každé primitivum generováno samostatně. Lze ale podědit metodu **GLRender()** z rodičovské třídy **SoShape** a naimplementovat vlastní efektivnější formu posílání vertexů do openGL. Před samotným renderem by měla metoda **GLRender** obsahovat kontrolu zda se má vůbec daný tvar renderovat. To lze zjistit pomocí metody **SoShape::shouldGLRender()**, která kontroluje zda není daný tvar **INVISIBLE**, zda je jeho obalové těleso uvnitř pohledového tělesa nebo zda nedošlo k přerušení renderovací sekvence. Coin sám si určí (lze změnit pomocí GUI nebo i programově), jaký styl vykreslování se bude používat (**glPolygonMode()**).

Protínání paprsků

Pro testování protínání paprsky lze podědit metodu **rayPick()** z rodičkovské třídy **SoShape**. V tomto případě pak metoda **rayPick()** testuje paprsek s každým primitivem vygenerovaným pomocí **generatePrimitives()**. Pokud paprsek protne primitivum, vytvoří **SoPickedPoint**. Pokud chceme implementovat vlastní **rayPick()** (například pokud nechceme testovat protínání s generovanými primitivy ale s vykreslovaným objektem), je třeba opět nejprve zkontrolovat, zda je třeba vůbec testovat protínání pomocí metody **SoShape::shouldRayPick()**.

Obalové těleso

Třída **SoShape** poskytuje metodu **getBoundingBox()**, která volá virtuální metodu **computeBBox()**, kterou je třeba definovat (tato metoda je také používána během renderingu na ořezávání). Pokud je tvořená třída děděna ze třídy **SoNonIndexedShape** lze použít metodu **computeCoordBBox()** která projde specifikované množství vertexů a z jejich minimálních a maximálních souřadnic vypočítá rozměry obalového tělesa a z průměrných hodnot jeho střed. Pokud nová třída dědí z **SoIndexedShape**, není třeba metodu definovat a lze použít zděděnou metodu, která se chová prakticky stejně jako **computeCoordBBox()** u **SoNonIndexedShape** s tím rozdílem, že počítá se všemi koordináty v seznamu koordiátů s nezáporným indexem.

5.2.2 Třída SoSeamlessTerrain a třídy s ní související

V následující části práce budou popsány třídy důležité pro běh implementovaného algoritmu a podstatné metody těchto tříd.

SoSeamlessTerrain

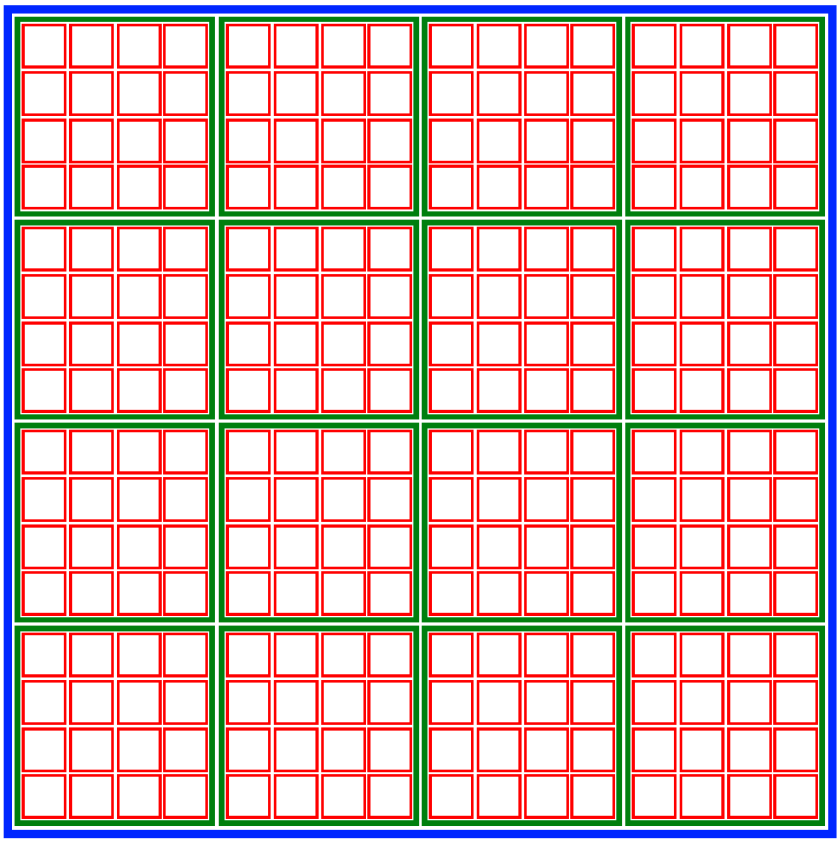
Samotná třída **SoSeamlessTerrain** během renderingu vytváří terén o délce hrany 1.0, která může být upravena přes field (parametr uzlu) **scale**, a maximální výšce 0.02, která

může být upravena přes parametr uzlu **maxHeight**. Jelikož se jedná o novou třídu implementovanou do prostředí Coin3D, je třeba ji na začátku aplikace inicializovat prostřednictvím metody **SoSeamlessTerrain::initClass()**. Ke správné funkčnosti algoritmu je třeba přiřadit třídě aktuální kameru zobrazované scény pro určení úrovně detailu jednotlivých dlaždic. Dále je třeba aby v separátoru, ve kterém se instace třídy nachází, byla před samotnou instancí umístěn **SoShaderProgram** s příslušnými shadery. Při prvním běhu zobrazovací sekvence se z OpenGL načte identifikátor aktuálního shader program pomocí **glGetIntegerv()**, který se později použije pro předávání uniform proměnných do GLSL. Pro vykreslování terénu samotného je implementována virtuální metoda **GLrender()** zděděná z mateřské třídy **SoShape**. Uvnitř této metody proběhne vymazání seznamu aktivních plátů a jeho novému vygenerování z hierarchie plátů. Seznam aktivních plátů je pak procházen a jednotlivé pláty jsou vykresleny. V pseudokódu by mohla vykreslovací sekvence třídy **SoSeamlassTerrain** vypadat například takto:

```
if (firstRun)
{
  getShaderId()
}
clearActivePatches()
generateActivePatches()
drawActivePatches()
```

PatchHierarchy

Tato třída uchovává hierarchii všech plátů. Neuchovává však žádné konkrétní vertexy, ale pouze infomace o umístění a velikosti plátů a seznam obsahující všechny podpláty daného plátu. Seznam podplátů je tvořen opět instancemi třídy **PatchHierarchy**, třída je tedy v tomto smyslu rekurzivní. Třída, kromě výše uvedených hodnot, obsahuje také informace o tom, zda je či není listem. Schéma hierarchie plátů je vidět na následujícím obrázku 5.2. Nejdůležitější metodou této třídy je metoda**generateActivePatches()**, která do seznamu aktivních plátů vloží v závislosti na pozici kamery pláty určené k vykreslení. Tato metoda je vždy volána na nejvyšší úroveň hierarchie plátů. Pokud by se měl v závislosti na vzdálenosti od kamery rozdělit plát na potomky, je tato metoda zavolána na každého z takto získaných potomků. K určení, zda se má plát rozdělit na potomky, je použita metoda **generatePatch()**, která je zavolána pro každý plát a vrací aktivní plát s informacemi o rozlišení jednotlivých dlaždic plátu. Pokud by bylo třeba větší rozlišení než je k dispozici, vrátí prázdný ukazatel, který je indikátorem potřeby rozdělení plátu na potomky. Ve své implementaci algoritmu Seamless Patches for GPU-Based Terrain Rendering jsem k určení rozlišení dlaždice použil nejbližší bod na dané hraně plátu. Vzdálenosti obou koncových bodů každé hrany jsou porovnány a pokud jsou stejné, je jako nejbližší bod určen střed mezi oběma body. Tak jsem docílil toho, že pro každé dvě vzájemě se dotýkající hrany je rozlišení stejné. Pro určení vzdálenosti nejbližšího bodu obsahuje hierarchie plátu metodu **distanceToEdge()**. Třída ještě obsahuje další tři parametry, které nejsou nutné pro běh algoritmu ale umožňují měnit výsledné zobrazování. Prvním takovým parametrem je parametr **culling**, přes který je možné vypínat a zapínat ořezávání zadních stran terénu a ořezávání pohledovým tělesem. Dalším parametrem je **drawMode**, který umožňuje přepínat vykreslování přes Vertex Buffer Objekty a přes přímé vykreslování pomocí **glVertex3f()**. Tyto dva parametry primárně slouží k porovnání výkonu vykreslování za použití různých postupů. Posledním z této sady parametrů je **colorMode**, který mění barvu vykreslova-



Obrázek 5.2: Schéma všech úrovní hierarchie plátů pro 3 rozlišení dlaždic a R = 4.

ného terénu. Defaultně je nastaven na vykreslování barvy podle barevné mapy terénu. Další nastavení rozlišuje vykreslené dlaždice od spojovacích pruhů, dlaždice jsou vykresleny zeleně, pruhy červeně. V posledním nastavení je barva terénu měněna v závislosti na úrovni detailu, podle které je určována výška vykreslovaných vrcholů.

ActivePatches

Účelem této třídy je udržovat seznam aktivních plátů určených pro renderování. Jedná se o singleton třídu, protože ke třídě je potřeba přistupovat z metod více různých tříd. V průběhu metody **SoSeamlessTerrain::GLRender()** je seznam nejprve vyprázdněn metodou **clear()**. Posléze je vygenerován nový seznam metodou **PatchHierarchy::generateActivePatches()**. Seznam aktivních plátů je poté procházen pomocí metody **next()**, která vrátí první plát a vloží ho na konec seznamu. Každý takto získaný plát je pak samostatně vykreslen. Aktivní pláty uvnitř seznamu jsou instance třídy **Patch**.

Patch

Třída **Patch** na rozdíl od třídy **PatchHierarchy** již uchovává další informace důležité k vykreslení plátů. Kromě pozice a velikosti plátu, které byly přítomny i ve zmíněné třídě nadále udržuje informace o základním rozlišení plátu a o rozlišení jednotlivých dlaždic uvnitř plátu. Toto rozlišení je reprezentováno (přes enumerátor) hodnotami určenými podle

rozlišení dlaždice (16, 32 a 64). Základní rozlišení plátu pro pláty generované z nejvyšší úrovně hierarchie plátů je šest a pro každou nižší úroveň toto rozlišení klesá o dvě (nejnižší základní rozlišení je tedy pro tři různá rozlišení rovno dvěma). Tato třída dále poskytuje dvě veřejné metody. První metodou je metoda **draw_primitive**, která slouží pro generování primitiv pro ray picking. V takto vygenerovaných primitivech jsou celé dlaždice a pruhy nahrazeny jedním trojúhelníkem. Pro celý plát se tedy generují pouze 4 trojúhelníky. Druhou metodou je metoda **draw()**, která pro každou dlaždici a každý spojující pruh zavolá metodu třídy **Tile** vykreslující daný prvek.

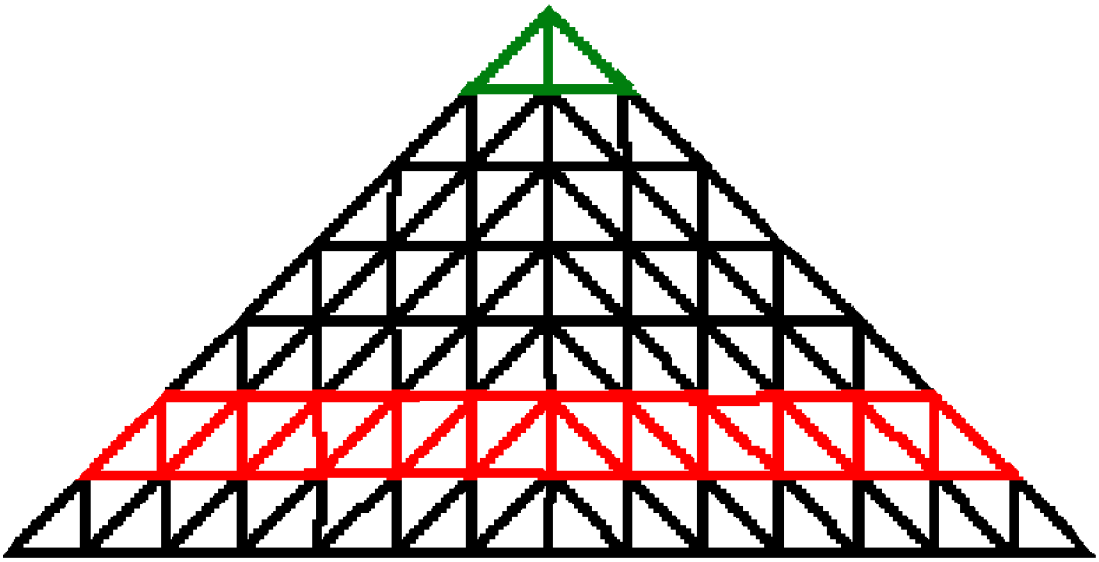
Tiles

Tato třída již uchovává konkrétní geometrii (seznam vertexů) jednotlivých dlaždic a pruhů. Během inicializace v konstruktoru jsou vytvořeny vertexy pro všechny varianty rozlišení dlaždic a pro všechny možné kombinace rozlišení pruhů. Vytvoření pozic vertexů pro dlaždice je poměrně jednoduché, jak je vidět následující ukázce:

```
foreach (resolution)
{
  midpoint = resolution / 2.0;
  for (displace = 0; displace < (midpoint - 1); displace++)
  {
    if (not lastLine)
    {
      generateLineOfTriangles();
    }
    else
    {
      generateTop();
    }
  }
}
```

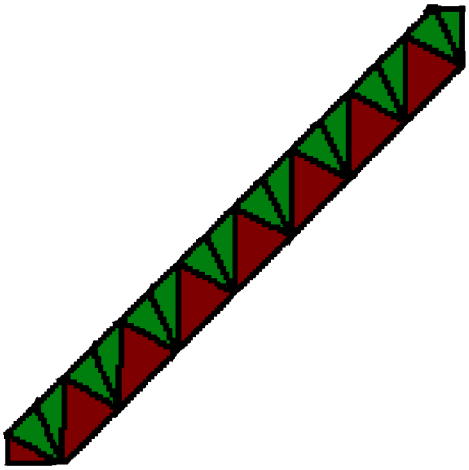
Kde **generateLineOfTriangles()** by vygenerovala jeden řádek dlaždice o daném rozlišení, jak ukazuje obrázek 5.3 červeně a **generateTop()** vrchol dlaždice (tentýž obrázek zeleně). Minimálně je nutné vygenerovat stejný počet dlaždic jako je rozlišení. Je však vhodné vytvořit všechny dlaždice ještě ve všech možných orientacích za účelem zrychlení vykreslování. Transfromovat dlaždici rotací třikrát pro každý plát je v případě velkého přiblížení poměrně náročné. Při tvorbě dlaždic si vystačíme s použitím vykreslovacích módů **GL_TRIANGLESTRIP** a **GL_TRIANGLES**. Důležité je při tvorbě vrcholů dlaždic generovat jednotlivé vrcholy po směru hodinových ručiček kvůli ořezávání zadních stěn, které je automaticky použito na celistvá tělesa, která jsou uvozena metodami **SoShape::beginSolidShape()** a **SoShape::endSolidShape()**.

Generování spojovacích pruhů je o něco náročnější. Pro tři různá rozlišení dlaždic je třeba udělat šest pruhů pro různé kombinace těchto dlaždic (prípadně pět, pokud se vyhýbáme extrémním hodnotám parametru ρ nemělo by dojít k propojování dlaždice s nejvyšším a nejnižším rozlišením). Opět jako u dlaždic i u spojovacích pruhů je výhodné vygenerovat další pruhy pro ostatní možné orientace pruhů. Narozdíl od dlaždic jsem při vykreslování spojovacích pruhů použil vykreslovací mód **GL_TRIANGLE_FAN** kombinovaný s **GL_TRIANGLES** pro spojení vyššího a nižšího rozlišení a **GL_TRIANGLESTRIP**



Obrázek 5.3: Schéma jedné dlaždice, jeden řádek je označen červeně

pro spojení stejných rozlišení. Na obrázku 5.4 je vidět rozdělení stripu spojujícího vyšší a nižší rozlišení na vějíře a trojúhelníky.



Obrázek 5.4: Schéma jednoho spojovacího pruhu spojujícího rozlišení 16 a 32 na delší hraně dlaždice. Červeně jsou označené trojúhelníky, zeleně vějíře.

Samotná třída **Tiles** kromě uchovávání samotné geometrie také umožnuje její vykreslování. K tomu jsou určeny metody **draw_tile()** a **draw_strip()**. Obě metody požadují v parametrech udání pozice, velikosti, orientace a rozlišení vykreslované dlaždice. Dalé také velikost celého terénu a zakládní úroveň detailu vykreslovaného plátu. V neposlední řadě je třeba také předat právě používaný shader program potřebný k předávání uniformů do GLSL. Metoda **draw_strip()** vyžaduje místo jednoho, dvě rozlišení pro levou a pravou spojovanou dlaždici. Základní úroveň detailu je pak podle rozlišení dlaždice snížena o hodnotu v závislosti na rozlišení. Nula pro nejnižší rozlišení, dva pro nejvyšší rozlišení (pokud používáme tří rozlišení). Všchny dlaždice a pruhy jsou vykreslovány s pozicí vrcholů v rozmezí

$< 0.0, 1.0 >$ kvůli generování dlaždic během inicializační fáze a ne až při vykreslování. Tato pozice je později použita pro zjištění texturovacích koordinátů pro dané vrcholy.

Vec2d a Vec2i

Tyto třídy jsou spíše podpůrného charakterou. Obě třídy pouze zapouzdřují dvě hodnoty, v jednom případě dvakrát **double** ve druhém případě dvakrát **integer**. Třídy jsou využity při předávání pozic a velikostí mezi ostatními třídamy.

FpsProfiler

Tato singleton třída stojí mimo implementaci algoritmu Seamless Patches for GPU-Based Terrain Rendering. Na základě metod **startMeasure()** a **stopMeasure()**, kterým se jako parametr předává identifikační číslo měřeného úseku kódu, měří počet průchodů tímto úsekem za sekundu. V případě vložení před a za vykreslovací řetězec lze jimi tedy měřit počet snímků za sekundu vykreslovaný aplikací. Třída má základ v **FpsProfiler**u poskytnutého panem ing. Radkem Bartoněm. Byla však předělána, aby fungovala pod operačním systémem windows a mírně zjednodušena. Pro uchovávání naměřených hodnot tato třída využívá strukturu **FpsResult**, do které se ukládají časová známka, informace zda se jedná o začátek čí konec měřeného úseku, přibližný počet vykreslovaných vrcholů a jeho identifikační číslo. Aktuální čas je získán ze systémové funkce clock() obsažené v **time.h**. Tato třída obsahuje ještě metody **clear()**, která vyčistí seznam výsledků, a **printResult()**, která projde seznam výsledků a do souboru získaného z parametru metody uloží informaci o tom, kolikrát proběhl daný úsek kódu v dané vteřině a kolik se v této vteřině průměrně vykreslilo vrcholů. Tisknutí výsledků je voláno při ukončení aplikace.

Shrnutí

Zjednodušené chování výše zmíněných tříd je shrnuto v diagramu 5.5. V tomto diagramu jsou vynechány pro přehlednost pomocné třídy **Vec2d**, **Vec2i** a **FpsProfiler**.

5.3 Shadery

Jak je zmíněno výše, shadery použité v ukázkové aplikaci jsou psány v jazyce GLSL. Tyto shadery jsou nutné ke správnému běhu algoritmu Seamless Patches for GPU-Based Terrain Rendering, neboť pomocí právě těchto shaderů je na vertexy dlaždic a spojovacích pruhů vygenerovaných třídou **SoSeamlessTerrain** aplikován displacement podle dané úrovně mipmapy výškové mapy a barva podle barevné mapy.

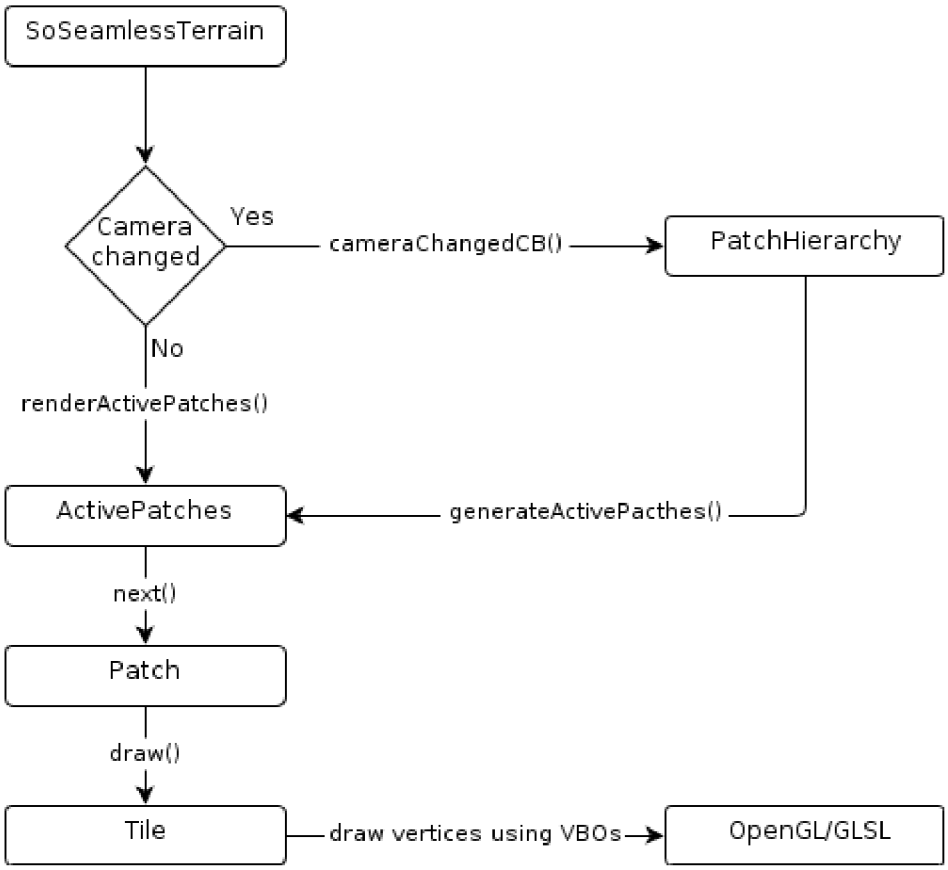
5.3.1 Vertex shader

Primární funkcí vertex shaderu je upravit pozici vertexů generovaných přes OpenGL na ose Z. K tomu je potřeba několik údajů předáváných z OpenGL jako uniform proměnné.

Dlaždice

Pro správnou funkčnost vertex shaderu aplikovaného na dlaždice jsou potřeba následující informace:

- Výšková mapa



Obrázek 5.5: Diagram zobrazující chování tříd během vykreslování jednoho snímku.

- Pozice vykreslovaného plátu
- Velikost vykreslovaného plátu
- Úroveň detailu dlaždice
- Informace zda je daná dlaždice v nejvyšším rozlišení
- Informace o tom zda se vykresluje dlaždice nebo pruh
- Maximální možná výška terénu

Nejprve jsou na základě pozice a velikosti dané dlaždice a pozice konkrétního vertexu vypočítány texturovací koordináty pro daný vertex. Koordináty jsou spočteny následovně:

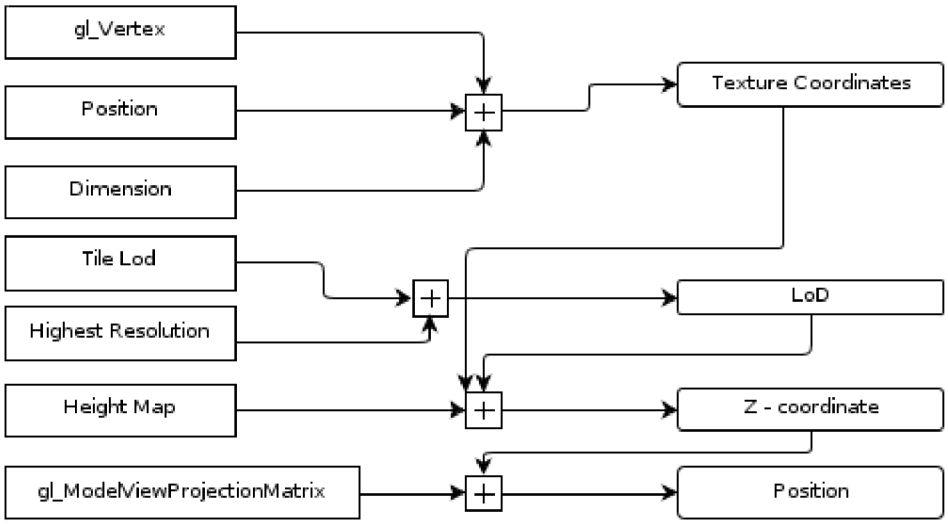
$$texCoord.x = position.x + (gl\_Vertex.x * dimension.x)$$

$$texCoord.y = position.y + (gl\_Vertex.y * dimension.y)$$

Následně se vertex shader podívá, zda se jedná o dlaždici. Pokud ano, nastaví hodnotu **lod** na hodnotu předávanou uniformem z OpenGL. Poté se pomocí funkce **texture2DLod()** načte barva z mipmapy výškové mapy úrovně uložené v **lod**. Z této barvy je poté vypočtena intenzita pomocí následujícího výpočtu:

$$I = (0.2989 * R) + (0.5870 * G) + (0.1141 * B)$$

Intenzita je poté vynásobena přiměřeně malou konstantou, aby vzniklé výšky nevypadaly nepřirozeně (v ukázkové aplikaci jsem použil defaultní hodnota 0.02, lze ji však za běhu měnit). Po této úpravě je vzniklá hodnota použita jako pozice na ose z. Pozice na ose x a y zůstávají z původního vertexu. Výstupem shaderu je pak projekční modelová matice (ModelViewProjectionMatrix) násobená novou pozicí vertexu, texturovací koordináty pro fragment shader a úroveň mipmapy z níž byla načítána hodnota pro elveaci vertexu. Pro přehled jsou vstupy a výstupy a jejich propojení znázorněny na diagramu 5.6



Obrázek 5.6: Diagram znázorňující vstupy a výstupy vertex shaderu při vykreslování dlaždice.

Spojovací pruhy

Narozdíl od dlaždic spojovací pruh potřebuje ve vertex shaderu některé informace navíc. Tedy celkem:

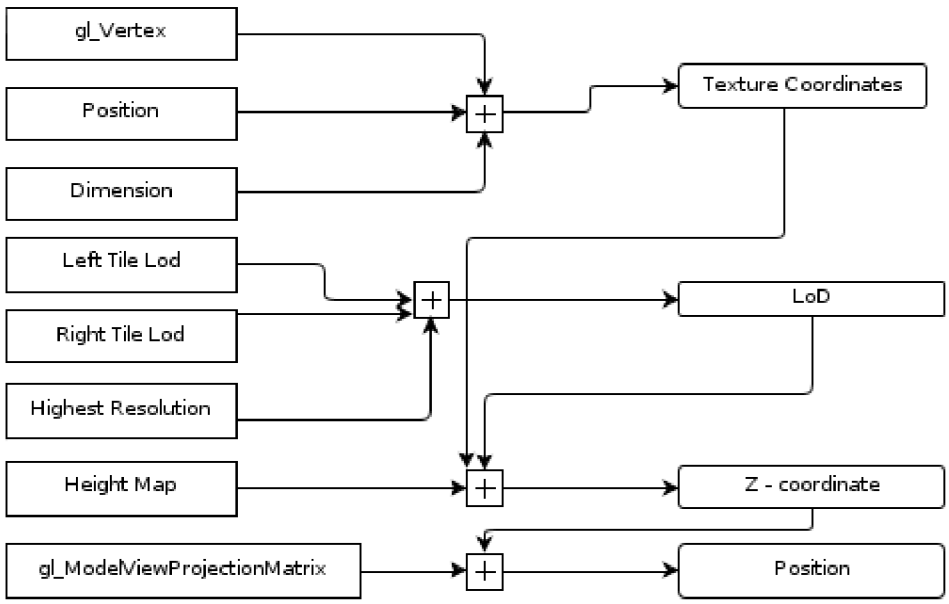
- Výšková mapa
- Pozice vykreslovaného plátu
- Velikost vykreslovaného plátu
- Úroveň detailu dlaždice nalevo od spojovacího pruhu
- Úroveň detailu dlaždice napravo od spojovacího pruhu
- Informace o tom zda se vykresluje dlaždice nebo pruh
- Orientace dlaždice

Stejně jako u dlaždice je nejprve spočtena hodnota texturovacích koordinátů pro daný vertex. Dále pokud se jedná o spojovací pruh je v závislosti na orientaci dlaždice (pruhu) zjištěno, zda se daný vertex nachází v levé nebo pravé části pruhu:

```
x = gl_Vertex.x
y = gl_Vertex.y
```

```
lod = lodRight
switch (orientation)
{
  case bottom:
    if (x > y)
      lod = lodLeft
    break
  case top:
    if (y > x)
      lod = lodLeft
    break
  case left:
    if (x > 1.0 - y)
      lod = lodLeft
    break;
  case right:
    if (y > 1.0 - x)
      lod = lodLeft
    break
}
```

Dále vertex shader pokračuje stejně jako při vykreslování dlaždice. Je spočtena intenzita barvy z mipmapy výškové mapy podle úrovně levé popřípadě pravé strany spojovacího pruhu, tato intenzita je po vynásobení konstantou použita jako hodnota na ose z a po vynásobení projekční modelovou maticí odeslána jako výstupní pozice vertexu. Jako u vykreslování dlaždice je na obrázku 5.7 schéma vstupů a výstupů pro vertex shader při zobrazování spojovacích pruhů.



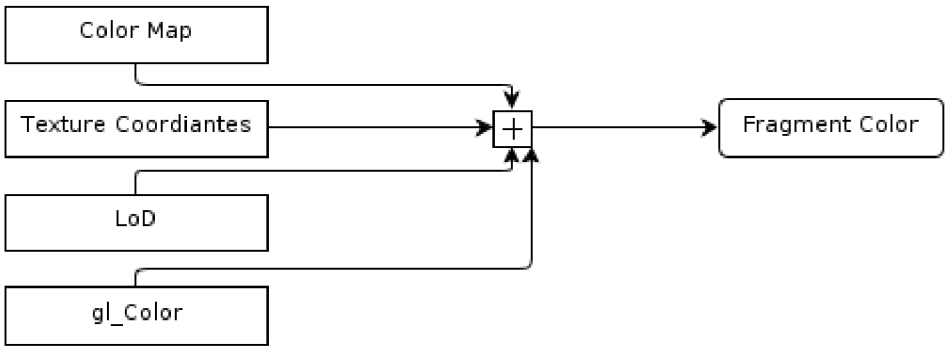
Obrázek 5.7: Diagram znázorňující vstupy a výstupy vertex shaderu při vykreslování spojovacího pruhu.

5.3.2 Fragment shader

Narozdíl od vertex shaderu funkce fragment shaderu je stejná pro dlaždice i pro spojovací pruhy. Jako vstupy fragment shaderu slouží:

- Barevná mapa
- Texturovací koordináty z vertex shaderu
- Úroveň detailu z vertex shaderu

Shader samotný je jednoduchý. Po načtení barvy z barevné mapy za pomoci funkce **texture2DLod()**, kde jako parametry slouží barevná mapa, texturovací koordináty a úroveň detailu, je takto načtená barva vynásobena barvou vertexů z OpenGL a poslána jako barva výstupního fragmentu z fragment shaderu. Přehled vstupů a výstupů shaderu je znázorněn na obrázku 5.8.



Obrázek 5.8: Diagram znázorňující vstupy a výstupy fragment shaderu.

5.4 Ukázková aplikace

Po přehledu všech postupů, tříd a shaderů potřebných pro běh aplikace samotné si v této sekci ukážeme, jak vypadá aplikace samotná. Na začátku funkce main je třeba inicializovat vytvořené třídy, které chceme zakomponovat do grafu scény prostředí Coin3D. V tomto případě se do grafu scény přidává jediný nový uzel a to je **SoSeamlessTerrain**. Všechny ostatní třídy tento uzel sice používá, ale není třeba aby byly součástí prostředí Coin3D. Dále je třeba inicializovat **FpsProfiler**, který si rezervuje místo v paměti pro ukládání výsledků měření a načte si počáteční čas pro měření uplynulého času. Následuje vytvoření QT okna pomocí příkazu:

```
QWidget* mainwin = SoQt::init(argc, argv, argv[0]);
```

Následuje vytvoření kořenového uzlu grafu scény, kterým musí být **SoSeparator** a prohlížeče knihovny SoQt, pomocí kterého se bude zobrazovat vykreslovaná scéna. Rozhodl jsem se pro **SoQtExaminerViewer**, který umožňuje rotovat a přibližovat celou scénu.

```
SoSeparator* root = new SoSeparator;
SoQtExaminerViewer* eviewer = new SoQtExaminerViewer(mainwin);
```

Dále je tomuto prohlížeci nastaveno vykreslování na single buffering, aby bylo vůbec možné zjistit rychlost vykreslování, protože se zapnutým double bufferingem je rychlost vykreslování limitována na 60 snímků za vteřinu. Kořenovému uzlu scény je třeba zvýšit počet referencí, aby nedošlo k jeho vymazání v průběhu generování scény. Všechny uzly, které nejsou ve scéně odkazovány, jsou později vymazány z paměti. Toho se využívá při vykreslování dynamicky se měnících scén s přibývajícími a ubývajícími objekty. Tyto změny jsou provedeny následujícími příkazy:

```
eviewer->setBufferingType(SoQtViewer::BUFFER\_SINGLE);
root->ref();
```

Jak je vidět ve grafu scény výsledné aplikace (5.1), je před samotným uzlem terénu přidána kamera scény. Tento krok pro samotný prohlížeč není nutný, prohlížeč si vytvoří vlastní kameru pokud žádná není k dispozici. K této kameře však není přístup a my ji potřebujeme k určování úrovní detailu jednotlivých plátů. Dále do grafu scény přidáme světlo a materiál. Světlu nastavíme pozici (**location**), barvu (**color**) a intenzitu (**intensity**) a materiálu ambientní (**ambientColor**), difuzní (**diffuseColor**), spekulární (**specularColor**) a emisní (**emmisiveColor**) barvevnou složku a lesklost (**shininess**). Všechny parametry jsou pole daných tříd a lze je nastavit přes metodu **setValue()**.

```
SoPerspectiveCamera *cam = new SoPerspectiveCamera();
SoPointLight *light = new SoPointLight();
SoMaterial *mat = new SoMaterial();
root->addChild(cam);
root->addChild(light);
root->addChild(mat);
```

Kvůli zobrazení informací o vykreslovaném terénu je vytvořen separátor pro textové položky. Pro tento separátor je třeba nastavit novou orthografickou kameru (pokud chceme zobrazovat text na stabilní pozici v okně), translaci pro přesun textu na požadované místo, barvu textu, obecnou callback funkci která bude zavolána při každém překreslení, samotný text a nakonec uzel resetující obalová tělesa pro určení počátečního pohledu. Poslední ze zmíněných je potřeba kvůlí správné funkčnosti metody **viewAll()**, protože uzel **SoText2** má poměrně velké obalové těleso a při jeho absenci dochází k extrémnímu oddálení obrazu. Nakonec je třeba vše vložit do textového separátoru a ten umístit do kořenového separátoru. Pro vložení dalšího řádku textu je třeba vytvořit nový text a translaci a vložit je do textového separátoru před reset obalových těles.

```
SoSeparator *textSeparator = new SoSeparator();
SoOrthographicCamera *orthCam = new SoOrthographicCamera();
SoCallback *textCallback = new SoCallback();
SoTranslation *textTranslation = new Translation();
SoBaseColor *textColor = new SoBaseColor();
SoText2 *text = new SoText2();
SoResetTransform *resetBBox = new SoResetTransform();

textCallback->setCallback(textCB, text);
textTranslation->translation = SbVec3f(-0.99f, 0.9f, 0.0f);
textColor->rgb.setValue(1.0, 1.0, 0.0);
text->string = "FPS: "
```

```
resetBBox->whatToReset = SoResetTransform::BBOX;

textSeparator->addChild(textCallback);
textSeparator->addChild(orthCam);
textSeparator->addChild(textColor);
textSeparator->addChild(textTranslation);
textSeparator->addChild(text);
textSeparator->addChild(resetBBox);
root->addChild(textSeparator);
```

Následuje vytvoření separátoru oddělujícího samotný terén od zbývajícího grafu scény, to je třeba kvůli potřebě algoritmu pracovat se shadery. Vytvořením nového separátoru zamezíme aplikaci použitých shaderů na ostatní objekty ve scéně. Tomuto separátoru již není třeba zvyšovat počet referencí, protože je odkazován v kořenovém separátoru do kterého je později přidán.

```
SoSeparator *terrainSeparator = new SoSeparator();
```

Po vytvoření separátoru můžeme vytvořit příslušný shader program a do něj umístit vertex shader a fragment shader. Obě části shader programu jsou samostatné uzly grafu scény, které se umisťují přímo do shader programu jako shader objekty. Shadery jsou automaticky načteny ze souboru poté, co je nastaven **sourceProgram** na instanci shader objektu.

```
SoVertexShader *vertexShader = new SoVertexShader();
vertexShader->sourceProgram.setValue("vertex.glsl");

SoFragmentShader *fragmentShader = new SoFragmentShader();
fragmentShader->sourceProgram.setValue("fragment.glsl");

SoShaderProgram *shaderProgram = new SoShaderProgram();
shaderProgram->shaderObject.set1Value(0, vertexShader);
shaderProgram->shaderObject.set1Value(1, fragmentShader);
```

Po vytvoření shader programu lze do GLSL posílat uniformní proměnně přes uzly grafu scény **SoShaderParameterNx**, kde "N"je počet proměnných a "x"je datový typ proměnných (tedy například **SoShaderParameter1i**). Následuje vytvoření **SoShaderParameter**ů pro předávání výškové mapy a barevné mapy do vertex a fragment shaderu. Jméno uniformní proměnné v GLSL je přiřazeno přes pole **name** a hodnota přes pole **value**. V tomto případě hodnota dané uniformní proměnné určuje, na které texturovací jednotce leží daná textura.

```
SoShaderParameter1i *uniformHeightMap = new SoShaderParameter1i();
uniformHeightMap->name = "heightMap";
uniformHeightMap->value = 0;

SoShaderParameter1i *uniformColorMap = new SoShaderParameter1i();
uniformColorMap->name = "colorMap";
uniformColorMap->value = 1;
```

Takto vytvořené uzly jsou pak předány přímo shader objektu, pro který jsou určeny. Pro uniformní proměnné, které je třeba měnit uvnitř vykreslovacího řetězce v závislosti na vykreslované dlaždici, popřípadě spojovacím pruhu, jsem použil jiný způsob předávání hodnot do GLSL. Při prvním běhu programu je zjištěno id aktuálně používaného shader programu z OpenGL pomocí funkci **glGetIntegerv(GL_CURRENT_PROGRAM, &currProgram)**. Takto získaná hodnota je postupně předána do jednotlivých vykreslovacích funkcí a přes **glGetUniformLocation()** a **glUniformNx** ("N"a "x"mají stejný význam jako u **SoShaderParameterNx**) jsou předány potřebné hodnoty do GLSL. Jelikož hodnoty nijak nesouvisí s grafem scény či načítanými texturami (jako v případě prvních dvou uniformních proměnných), bylo by dle mého názoru použít zde **SoShaderParameter** kontraproduktivní.

```
vertexShader->parameter.set1Value(0, uniformHeightMap);
fragmentShader->parameter.set1Value(0, uniformColorMap);
```

Poté je třeba celý shader program s shader objekty a parametry předat do separátoru oddělujícího samotný terén a tento separátor je přidán do kořenového separátoru celé scény. Místo metody **addChild()** je použita metoda **insertChild()**, která umožňuje určit, na kterou pozici v daném separátoru je uzel umístěn. Aby bylo jisté, že se shader program aplikuje na všechny uzly uvnitř separátoru, je explicitně určeno umístění na nejlevějším (tedy prvním) místě v daném separátoru.

```
terrainSeparator->insertChild(shaderProgram, 0);
root->addChild(terrainSeparator);
```

Dále je třeba načíst výškovou a barevnou mapu ze souboru do textury a ty umístit do příslušných texturovacích jednotek tak, jak byly nastaveny uniformní parametry výše. Načtení textury ze souboru probíhá obdobně jako načtení jednotlivých shader objektů. Nastavím pole **filename** v uzlu **SoTexture2**. Pomocí uzlu **SoTextureUnit** lze určit, do které texturovací jednotky budou následující textury v grafu scény vloženy. Každou texturu tedy předchází uzel určující texturovací jednotku, do které se textura vloží. Pokud bychom nechali textury v grafu scény samotné, byly by vloženy do nejnižší volné texturovací jednotky, avšak kvůli správné funkčnosti shaderu je třeba, aby byly textury na pozici dané uniformní proměnou, kterou jsme vytvořili dříve. Dále je třeba zajistit, že z textur budou generovány mipmapy pro určování výšky jednotlivých vrcholů dlaždic. Toho lze dosáhnout použitím dalšího uzlu ve grafu scény, tentokráte uzlu **SoComplexity**. Tomu lze přes pole **textureQuality** předat požadovanou kvalitu textur dle tabulky 5.1.

```
SoComplexity *complex = new SoComplexity();
complex->textureQuality = 1.0;
terrainSeparator->addChild(complex);

SoTextureUnit *heightUnit = new SoTextureUnit();
heightUnit->unit.setValue(0);
SoTexture2 heightMap = new SoTexture2();
heightMap->filename.setValue("height.png");
terrainSeparator->addChild(heightUnit);
terrainSeparator->addChild(heighMap);

SoTextureUnit *colorUnit = new SoTextureUnit();
```

```
colorUnit->unit.setValue(1);
SoTexture2 colorMap = new SoTexture2();
colorMap->filename.setValue("color.png");
terrainSeparator->addChild(colorUnit);
terrainSeparator->addChild(colorMap);
```

Poslední zbývající uzel, který je nutno vložit do separátoru oddělujícího terén od zbylého grafu scény, je terén samotný. Terénu je třeba nastavit kameru, podle které se má počítat úroveň detailu pro jednotlivé pláty pomocí metody **setCamera()** a je možné mu nastavit i velikost pomocí pole **scale**.

```
SoSeamlessTerrain *t = new SoSeamlessTerrain();
t->scale.setValue(10.0);
t->setCamera(cam);
terrainSeparator->addChild(t);
```

Následuje vytvoření callback funkce pro ovládání aplikace. K tomu slouží uzel **SoEventCallback**, který volá danou funkci v reakci na různé události. V tomto případě se bude jednat o událost změny stavu klávesnice. Do callback funkce je, jako uživatelská data, odesílán ukazatel na objekt terénu a callback uzel je vložen na začátek grafu scény.

```
SoEventCallback *controlCB = new SoEventCallback();
controlCB->addEventCallback(SoKeyboardEvent::getClassTypeId(),
                            keyboardCB, t);
root->insertChild(controlCB, 0);
```

Dále je kamera nastavena, aby z její počáteční pozice byla vidět celá scéna. Parametry metody **viewAll()** jsou separátor grafu scény, jehož obsah chceme vidět a velikost viewportu, ve kterém chceme dané objekty vykreslit. K určení pohledu je pak využito porovnání pohledového tělesa s obalovými tělesy objektů uvnitř daného separátoru.

```
cam->viewAll(root, eviewer->getViewportRegion());
```

Nakonec je do prohlížeče vložen graf scény, je zobrazeno Qt okno aplikace a zahájena hlavní vykreslovací smyčka. Po ukončení smyčky je vymazán prohlížeč dereferencován kořenový separátor scény ukončena práce SoQt a v neposlední řadě vytisknuty výsledky nasbírané během běhu programu **FpsProfiler**em.

```
eveiwer->setSceneGraph(root);
eviewer->show();
SoQt::show(mainwin);
SoQt::mainLoop();
delete eviewer;
root->unref();
SoQt::done();
FpsProfiler::get()->printResult("result.txt");
return 0;
```

5.5 Změny vůči návrhu

S pokračující implementací jsem začal zjišťovat, že některé věci by šli udělat jinak, než přesně podle návrhu, a tak jsem se je rohodl mírně pozměnit.

5.5.1 Třídy

Nejvýznamější změny oproti návrhu jsou patrně z diagramu implementovaných tříd 4.2. Hlavní třída pro implementovaný algoritmus byla přejmenována na **SoSeamlessPatches**, aby lépe korespondovala s názvem algoritmu samotného. Třída **Tiles** byla implementována tak, aby neuchovávala žádná data, pouze indexy jednotlivých Vertex Buffer Objektů (VBO) potřebných pro vykreslování dlaždic a spojujících pruhů. Třída je implementována jako singleton a orientace a rozlišení dlaždic a pruhů je předáváno třídě přes parametry metod **draw_tile()** a **draw_strip()**. Kvůli předělání třídy **Tiles**, která nyní obsahuje i funkčnost navržené třídy **Tile**, začala být třída **Tile** přebytečnou a byla odstraněna. Třídy **Texture** a **TexturePyramid** byly vynechány úplně. Rozhodl jsem se totiž využít hardwarově akcelerovaného generování mipmap přímo v OpenGL, které je pro statické textury dostačující. Hodnota v nižším rozlišení mipmapy má být podle popisu algoritmu vypočítána lineárně ze čtyř hodnot ve vyšším rozlišení, čehož lze dosáhnout nastavením texturovacího paremetru **GL_TEXTURE_MIN_FILTER** na hodnotu **GL_LINEAR_MIPMAP_NEAREST**, popřípadě **GL_LINEAR_MIPMAP_LINEAR**, pokud chceme hodnotu interpolovat i mezi mipmapami. S použitím knihovny Coin3D lze toho dosánout i použitím uzlu **SoComplexity**, nastavením jeho pole **textureQuality** na hodnotu větší než 0.9 (tabulka 5.1) a přidáním uzlu před použité textury v grafu scény. Kompletní tabulku i s dalšími detaily o třídě **SoComplexity** lze nalézt na stránkách společnosti VSG [12].

textureQuality	minFilter	magFilter	Mipmaps
$\leq 0,5$	GL_NEAREST	GL_NEAREST	FALSE
$\leq 0,6$	GL_LINEAR	GL_NEAREST	FALSE
$\leq 0,7$	GL_NEAREST_MIPMAP_NEAREST	GL_NEAREST	TRUE
$\leq 0,8$	GL_NEAREST_MIPMAP_LINEAR	GL_LINEAR	TRUE
$\leq 0,9$	GL_LINEAR_MIPMAP_NEAREST	GL_LINEAR	TRUE
$\leq 1,0$	GL_LINEAR_MIPMAP_LINEAR	GL_LINEAR	TRUE

Tabulka 5.1: Tabulka hodnot pro filtry textur při různých nastaveních pole textureQuality v uzlu SoComplexity.

5.5.2 Use case diagram

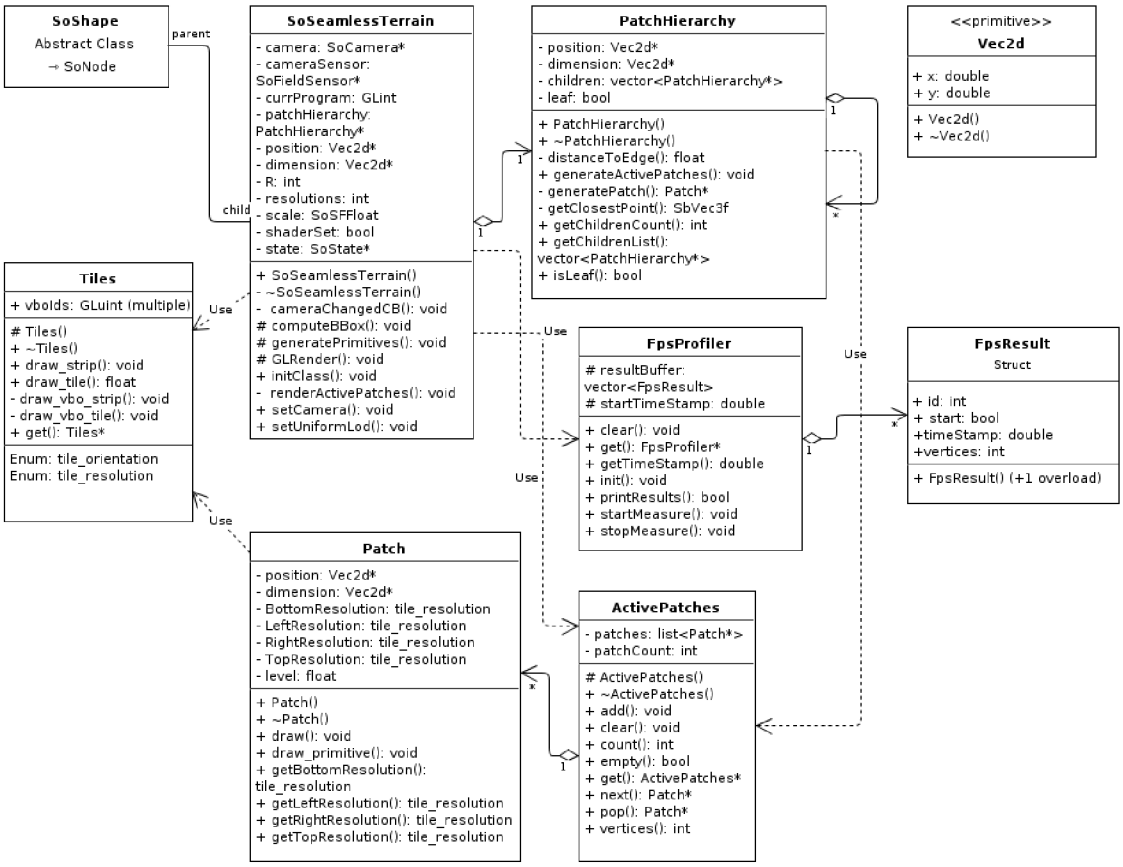
V průběhu implementace došlo také ke změně v diagramu použití. Není možno přímo v aplikaci generovat náhodný terén, ale je třeba jej načítat z textury. K načítání obrázků do textury slouží knihovna **simage1.dll** (popřípadě **simage1d.dll** pro debuging), která je součástí balíčku Coin3D. Samotné načtení textury je pak provedeno v uzlech textur **SoTexture2** nastavením hodnoty pole **filename** na požadovaný soubor.

5.6 Detaily řešení

Z původního popisu algoritmu Seamless Pathces for GPU-Based Terrain Rendering ([9]) nejsou patrné některé detaily nutné pro implementaci.

5.6.1 Návrh dlaždic a spojujících pruhů

Z [9] je zřejmé, že dlaždice, aby na sebe mohli přiléhat bezešvě, musejí mít na delší hraně korespondující rozlišení nebo, pokud je vedlejší plát rozdělen na podpláty, rozlišení rovno

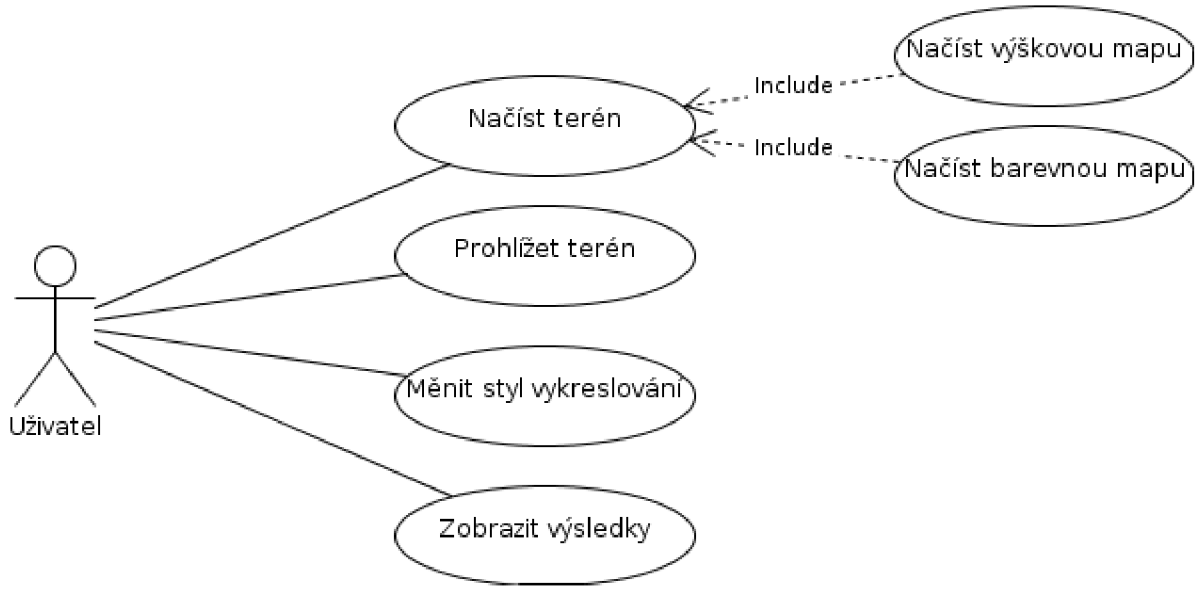


Obrázek 5.9: Diagram implementovaných tříd.

součtu rozlišení všech dlaždic a pruhů na všech podplátech dotýkajících se dané hrany. V ukázkové aplikaci jsou použita tři různá rozlišení dlaždic (16, 32 a 64) a tedy větvící faktor hierarchie je $R = 4$. Protože se rozlišení plátu určuje na základě nejbližšího bodu hrany, je zaručeno, že každé dva sousední pláty stejně úrovně na sebe budou bezešvě přiléhat. Pokud však bude jeden z plátů rozdělen na podpláty, je nutné, aby nejvyšší rozlišení bylo R-násobek nejnižšího. To lze zaručit tím, že každé další rozlišení bude dvojnásobkem předchozího. Dále je nutné místo pro spojovací pláty. Ty v ukázkové aplikaci zabírají místo jednoho trojúhelníku na delší hraně dlaždice. Dlaždice o rozlišení 16 nebude tedy mít na delší hraně 16 trojúhelníků, ale pouze 14, protože spojovací pruhy jsou na obou stranách dlaždice. Protože spojovací pruh takto rozměrově navazuje na dlaždici, je zaručeno bezešvé spojení i mezi pláty různých úrovní.

5.6.2 Parametr ρ

Faktor přesnosti ρ hraje poměrně významnou roli při určování rozlišení dlaždic a dělení na podpláty. V původním dokumentu popisujícím implementovaný algoritmus není uvedeno, jak tento parametr získat, či jakých hodnot by měl nabývat. Jelikož je hranice dělení na podpláty dána hodnotou 1, v implementaci tedy faktor přesnosti určuji na základě hloubky zanoření v hierarchii plátů. Pro pláty nejvyšší úrovně je ρ rovno jedné. Pro druhou úroveň je ρ rovno $0,9$ a ve třetí úrovni je ρ rovno $0,75$. Výpočet rozlišení dlaždic (popřípadě dělení) je pak během generování aktivních plátů proveden čtyřikrát pro každý plát v dané úrovni



Obrázek 5.10: Diagram použití výsledné aplikace.

hierarchie plátů.

5.6.3 Vzdálenost hrany plátu

Snad ještě vetší význam než parametr ρ má pro výpočet rozlišení dlaždic vzdálenost hrany plátu od kamery. V popisu algoritmu není uvedeno k jakému místu hrany se vzdálenost počítá. V ukázkové aplikaci počítám vzdálenost k nejbližšímu bodu od kamery na dané hraně. Tím je zaručeno, že rozlišení dlaždic bude stejné pro oba k sobě přiléhající pláty. K tomuto účelu je ve třídě **PatchHierarchy** implementována metoda **getClosestPoint**, která vrátí ze dvou zadaných bodů a pozice kamery bod nejbližší ke kameře. Pokud jsou oba body vzdáleny stejně, vypočte pozici uprostřed mezi oběma vstupními body a vrátí takto vypočítaný bod.

5.7 Problémy při implementaci

Při implementaci jsem narazil na několik problémů, které způsobovaly potíže ve vizualizaci robrazovaného terénu.

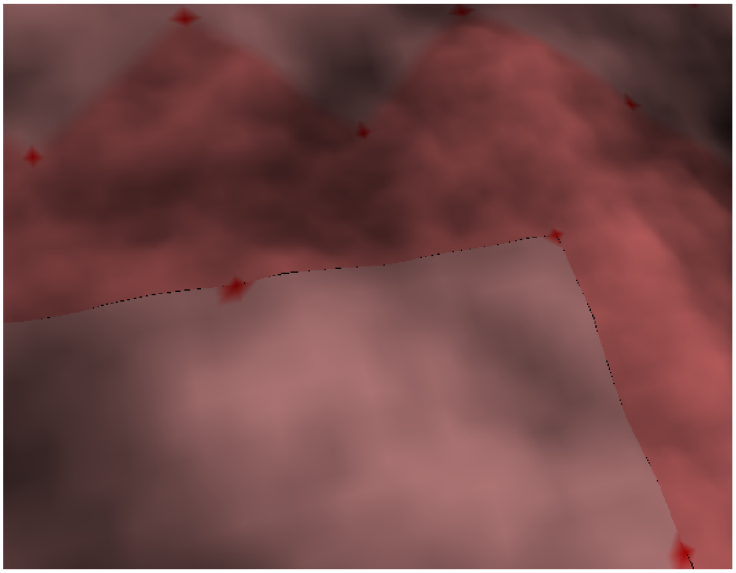
5.7.1 Mezery v terénu

V různých stádiích implementace vznikaly mezery na různých místech terénu. Nejprve mezery způsobené nestejným rozlišením přiléhajících dlaždic dvou plátů a později mezery způsobené změnou rozlišení mipmapy na různých dlaždicích ve stejném i v sousedních plátech.

Mezery na hranách

Nestejné rozlišení dvou dlaždic na sousedních plátech bylo nejprve způsobeno chybným výpočtem vzdálenosti hrany plátu od kamery. Zkoušel jsem počítat vzdálenost od středu

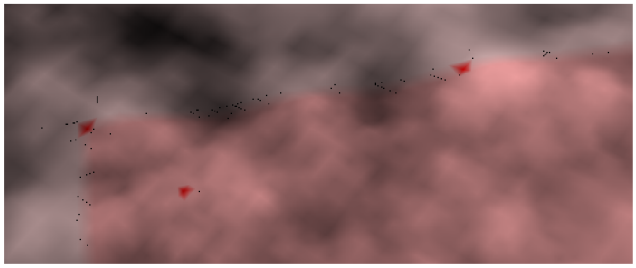
hrany i od krajního bodu ale vždy při jistých pozicích rozlišení na sousedních dlaždicích nesedělo. Problém jsem nakonec vyřešil počítáním vzdálenosti od kamery k nejbližšímu bodu na dané hraně, jak je popsáno výše. Další mezery na hranách vznikaly při přiléhání různých úrovní plátů. Těsně poblíž polohy kamery, kde docházelo k dělení plátu na podpláty, bylo možno najít pozici, kde se na hranách plátů nižší úrovně blíže ke kameře změnilo rozlišení na vyšší než nejnižší a tím došlo k narušení bezešvosti s plátem vyšší úrovně. Řešením byla změna hodnoty parametru ρ při výpočtu rozlišení dlaždice v závislosti na hloubce zanoření v hierarchii plátů. K hodnotám uvedeným výše jsem se dopracoval experimentálně.



Obrázek 5.11: Mezery vznikající v terénu nepřiléháním dlaždic.

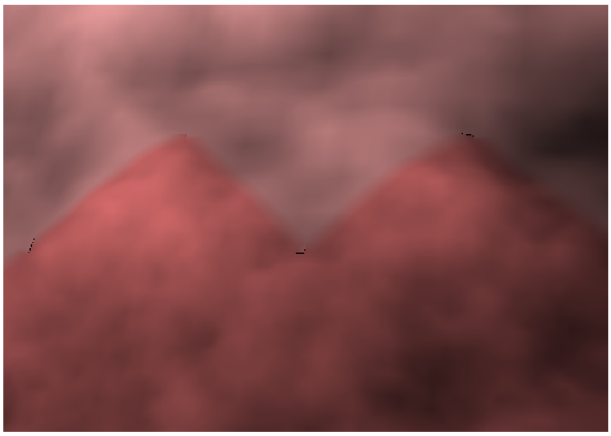
Mezery ve spojovacích pruzích

Do této kategorie spadají dva problémy. Prvním jsou mezery vzniklé hardwarově. Vznikají pouze na některých zařízeních (například na grafické kartě nVidia 8600M GT), kde při zjišťování hodnoty z mipmapy celočíselné úrovně (i při použití datových typů s plovoucí řádovou čárkou) k vrácení jiných hodnot pro vertex o stejné pozici. K tomuto dochází pouze pokud renderovaný polygon načítá pro jednotlivé vertexy z různích úrovní mipmap, tedy pouze u spojovacích pruhů. Řešením bylo zvýšit hodnotu úrovně detailu odesílanou parametrem v GLSL do funkce **texture2DLod()** přičíst malou hodnotu. Použil jsem hodnotu $0,001$ která má na výslednou hodnotu minimální vliv.



Obrázek 5.12: Mezery vznikající v terénu hardwarovou chybou.

Druhým problémem spadajícím do této kategorie byly mezery vzniklé na spojovacích pruzích ve středu a na rozích plátů. Úroveň mipmapy, ze které se načítá výška jednotlivých vertexů, je pro dlaždice jasná, koresponduje s rozlišením dlaždic. Tedy s jedním vrcholem na dlaždici koresponduje jeden pixel ve výškové mapě. U spojovacích pruhů se však rozlišení prolíná. Na každé straně pruhu je rozlišení podle dlaždice, se kterou daná strana sousedí. Problém však nastal právě na středu, kde se setkávala různá rozlišení dlaždic a tedy středový vrchol pro každý spojovací pruh mohl mít hodnotu načtenou z jiné úrovně mipmapy a tedy potenciálně různou. Ke stejnému problému docházelo na rozích plátů, kde se mohly setkávat různá rozlišení dlaždic sousedících se spojovacími pruhy na různých plátech. Tento problém jsem vyřešil použitím nejvyšší úrovně mipmapy práve pro vrcholy ve středu plátů a na jeho rozích (vždy pro jeden daný vrchol). Tím ale vznikl další problém. Na rozích plátu, pokud sousedil s plátem nižší úrovně hierarchie plátů, vznikaly nové mezery, způsobené kontaktem vrcholů spočtených z mipmapy úrovně nula a vrcholů spočtených z mipmapy dané dlaždicí plátu vyšší úrovně. Řešením tohoto problému bylo předání dalšího uniformního parametru do GLSL, který určuje, zda se vykresluje dlaždice s nejvyšším rozlišením (tedy potenciální soused plátu nižší úrovně). Pokud ano, je na vrcholech na každé čtvrtině dlaždice použita úroveň mipmapy nula. Pokud ne, je dlaždice vykreslena běžným způsobem. V případě, že dlaždice opravdu sousedí s plátem nižší úrovně, přilehají tyto vrcholy přímo na rohy plátů nižší úrovně. V opačné případě, kdy není vedleší plát rozdělen na podpláty, má sousední dlaždice také nejvyšší možné rozlišení a tedy vrcholy z mipmapy úrovně nula přiléhají na vrcholy stejné úrovně na sousední dlaždici.



Obrázek 5.13: Mezery vznikající v terénu způsobených rozdílným rozlišením na rozích a ve středu plátů.

5.8 Optimalizační kroky

Vývoj téměř jakéhokoliv softwarového produktu se nebejde bez prototypování a následné reimplementace některých částí. Proto i v ukázkové aplikaci nešlo vše zprvu zcela ideálně a některé části se mi podařilo, co se výkonu týče, postupem času o něco vylepšit. V této kapitole uvedu některé optimalizační kroky, které jsem v průběhu vývoje ukázkové aplikace provedl a jaký měly vliv na výsledný výkon celé aplikace.

5.8.1 Nahrazení přímého vykreslování vykreslováním za pomoci VBO

Bezpochyby největší vliv na zvýšení výkonu vykreslování terénu mělo přesunutí pozic vrcholů pro dlaždice a spojovací pruhy z operační paměti počítače na grafickou kartu pomocí Vertex Buffer Objektů. Protože není možné texturové koordináty jednotlivých vrcholů předpočítat, leda že by si každý plát udržoval informace o texturových souřadnicích, čímž by došlo k opětovnému zpomalení celého vykreslování, zároveň s touto úpravou jsem přenesl výpočet texturových koordinátů z OpenGL části aplikace do GLSL vertex shaderu.

5.8.2 Ořezávání

Během generování seznamu aktivních plátů ve výsledné implementaci ukázkové aplikace je před vložením plátu do tohoto seznamu zkontrolováno, zda plát neleží mimo pohledové těleso. Pokud plát leží mimo, je přeskočen a dále již se nepočítá s ním ani jeho potomky. Pokud je plát uvnitř pohledového tělesa a je rozdělen na podpláty nižší úrovně. Pro každý takový podplát je opět proveden test, zda leží uvnitř pohledového tělesa. Ořezáváním pohledovým tělesem v kombinaci s vykreslováním pouze horní strany terénu (back-face culling) došlo k navýšení výkonu o přibližně 10%. Bylo však třeba předělat vrcholy dlaždic a spojovacích pruhů do správného pořadí, aby bylo možné určit, která strana je horní a která spodní.

5.8.3 Zrušení výpočtu úrovně mipmapy na grafické kartě

Během původního řešení problému s mezerami mezi pláty jsem místo předávání úrovně mipmapy z OpenGL podle rozlišení vykreslované dlaždice použil výpočet úrovně mipmapy pro každý vertex zvlášť ve vertex shaderu. To sice vyřešilo problém s mezerami mezi pláty, avšak nebylo to přesně podle popisu algoritmu uvedeného v [9]. Navíc počítat znovu poměrně náročný výpočet pro každý vertex místo pro každou dlaždici bylo výkonově náročné. Popis řešení toho problému je popsán výše.

5.8.4 Generování aktivních plátů pouze při změně kamery

Místo aby se generovaly aktivní pláty při každém snímku znovu, zkusil jsem je generovat pouze při přesunu kamery. To mělo velmi výrazný vliv na počet snímků za sekundu při statické kameře (okolo 200%), avšak téměr nulový vliv při pohyblivé kameře. Bohužel se mi nepodařilo do callback funkce, kterou jsem za tímto účelem implementoval, předat **SoGLRenderAction**, která je nutná k ořezávání pohledovým tělesem. Instance této akce totiž není za běhu callback funkce přístupná, a po ukončení vykreslování se obsha této akce vyprázdní. Kvůli tomu jsem se rozhodl raději ztratit významnou část výkonu při zachování statické kamery na úkor výkonu při kameře pohyblivé.

Kapitola 6

Výsledky

Algoritmů pro vykreslování terénu v různých úrovních kvality podle vzdálenosti od kamery existuje celá řada. Mnou vybraný algoritmus kombinuje přístupy některých z těchto algoritmů, čímž poskytuje slušnou lokální adaptabilitu a přitom si uchovává jistou časovou nenáročnost. Základní principy algoritmu jsou poměrně jednoduché a intuitivní. Některé detaily sice nejsou v algoritmu přímo popsány (například jak získat parametr ρ nebo informace k jaké části hrany se má počítat vzdálenost od kamery), ale z popisu algoritmu je poměrně dobře patrné jaký vliv mají tyto detaily na výsledek běhu algoritmu a proto lze odvodit jak by je šlo asi vyřešit. S výběrem knihovny Coin3D jsem sice ze začátku nebyl příliš spokojen, protože se mi nepodařilo najít dostatek informačních pramenů pokrývajících problematiku tvorby nového uzlu do grafu scény, avšak s tím, že Coin3D sdílí stejné API s knihovnou Open Inventor, jsem nakonec z různých střípků zjistil potřebné informace. To, v kombinaci s trochou experimentování, vedlo k úspěšne implementaci nového uzlu a zdárnému dokončení ukázkové aplikace která tento uzel zobrazuje. Práce v prostředí Coin3D mimo tvorbu nového uzlu, to znamená konstrukce grafu scény a nastavování jednotlivých uzlů, byla přijemná a poměrně intuitivní. Pokud by byla aplikace napsána pouze v OpenGL, byl by počet snímků za sekundu jistě o něco vyšší. Avšak použitelnost výsledné implementace algoritmu by se omezila pouze na ukázkovou aplikaci. Takto může být tato implementace algoritmu Seamless Patches for GPU-Based Terrain Rendering využita v grafech scény v prostředí Coin3D.

6.1 Ukázková aplikace

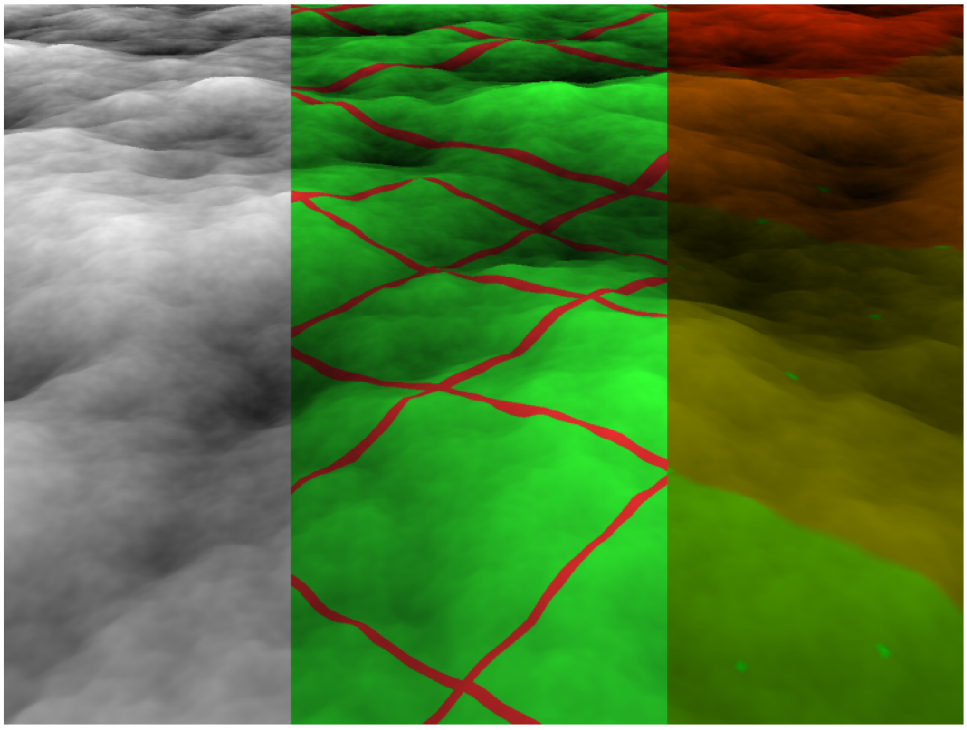
Výsledná aplikace, jak je uvedeno výše, zobrazuje vyrenderovaný terén pomocí vybraného algoritmu. Během incializace jsou vygenerovány jednotlivé vrcholy pro dlaždice a spojovací pruhy, tato část může na pomalejších strojích trvat přibližně vteřinu, nejedná se tedy o výrazné prodlení. Po zobrazení okna aplikace se rovnou zobrazí terén, jehož výšková a barevná mapa jsou načteny z obrázků **height.png** a **color.png**, které jsou umístěny ve stejné složce jako spouštěcí soubor aplikace. Změnou těchto obrázků lze zobrazit jiný terén. Jako referenční obrázek pro výškovou i barevnou mapu je použit perlinův šum, který se často používá pro náhodně generované terény. Za běhu aplikace jsou v levém horním rohu zobrazeny údaje o vykreslování. Tyto údaje jsou:

- Počet snímků za sekundu
- Přibližný počet zobrazovaných trojúhelníků

- Informace zda je zapnuté či vypnuté ořezávání
- Informace zda je zapnuté vykreslování pomocí VBO či přímé vykreslování
- Aktuální barevný mód
- Maximální výška terénu

Naneštěstí zobrazování textů v prostředí Coin3D je značně náročné na výkon a k zobrazení počtu snímků za sekundu a zobrazovaných trojůhelníků je třeba nahlížet do třídy **FpsProfiler**. Lze tedy za běhu programu vykreslování textů vypnout čímč dojde ke významnému zrychlení vykreslování. Prohlížeč knihovny SoQt však veškeré vstupy z klávesnice přesměrovává na ovládání základních funkcí prohlížeče, je tedy nutné před stiskem kláves které ovlivňují vykreslování terénu a zobrazování textu stisknout klávesu escape, pomocí které se prohlížeč přepne ovládání z módu tažení do ukazovacího módu, ve kterém lze již volat callback funkce na událost stisku klávesy.

Aplikace umožňuje pro testovací účely vypnou ořezávání pohledovým tělesem a ořezávání zadních stěn. Dále umožňuje přepínat mezi vykreslováním pomocí Vertex Buffer Objektů na přímým vykreslováním. Lze také přepínat barevné módy mezi normálním, kde je použita čistá barevná mapa pro jednotlivé fragmenty, obarvením dlaždic na zeleno a spojovacích pruhů na červeno a zobrazením úrovně mipmapy ze které se načítá výška pro daný vrchol (obrázek 6.1).

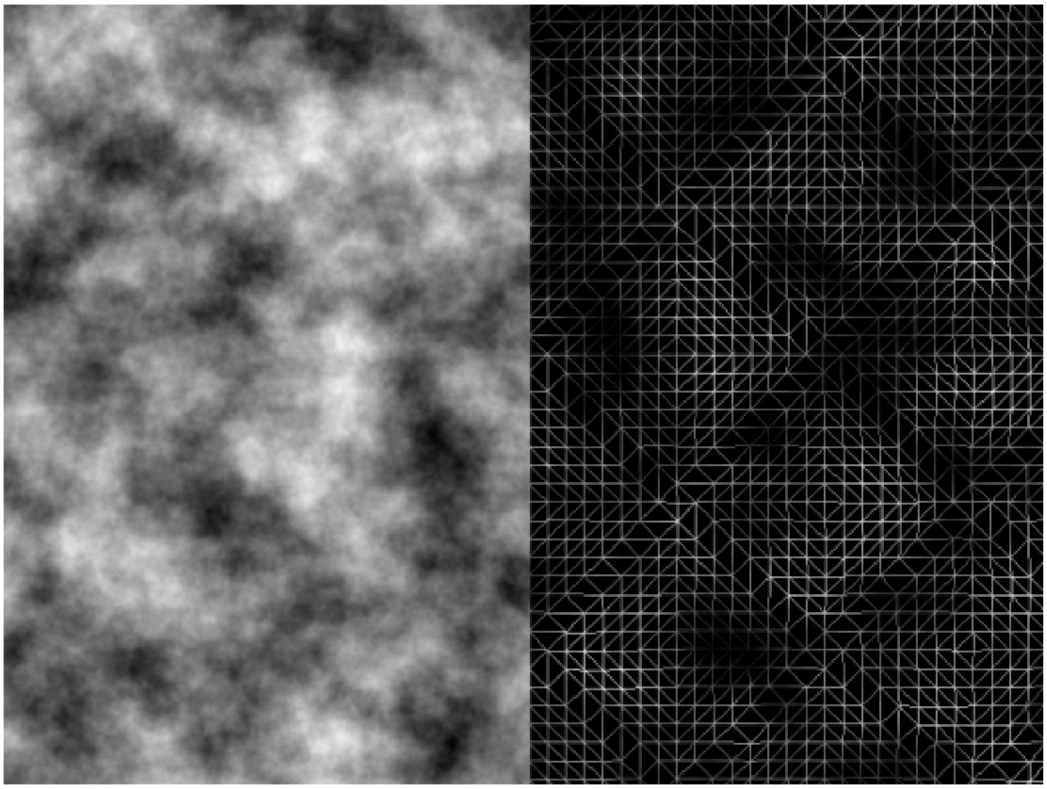


Obrázek 6.1: Jednotlivé barevné módy pro zobrazení terénu. Zleva: Normální, Dlaždice/Pruhy a Úroveň mipmapy.

6.2 Výkon vykreslování

Výkon výsledné implementace ukázkové aplikace jsem testoval na dvou počítačích. Prvním byl Intel Core2Duo T7300 2,0GHz se 4GB paměti a grafickou kartou NVIDIA GeForce 8600M GT (tabulka 6.1), druhým pak Intel Core i7 2600K se 16GM paměti a NVIDIA GeForce GTX 560 Ti (tabulka 6.2). Jako zdroj pro výškovou mapu jsem použil obrázky ve formátu PNG o rozlišení $4096x4096$ pixelů.

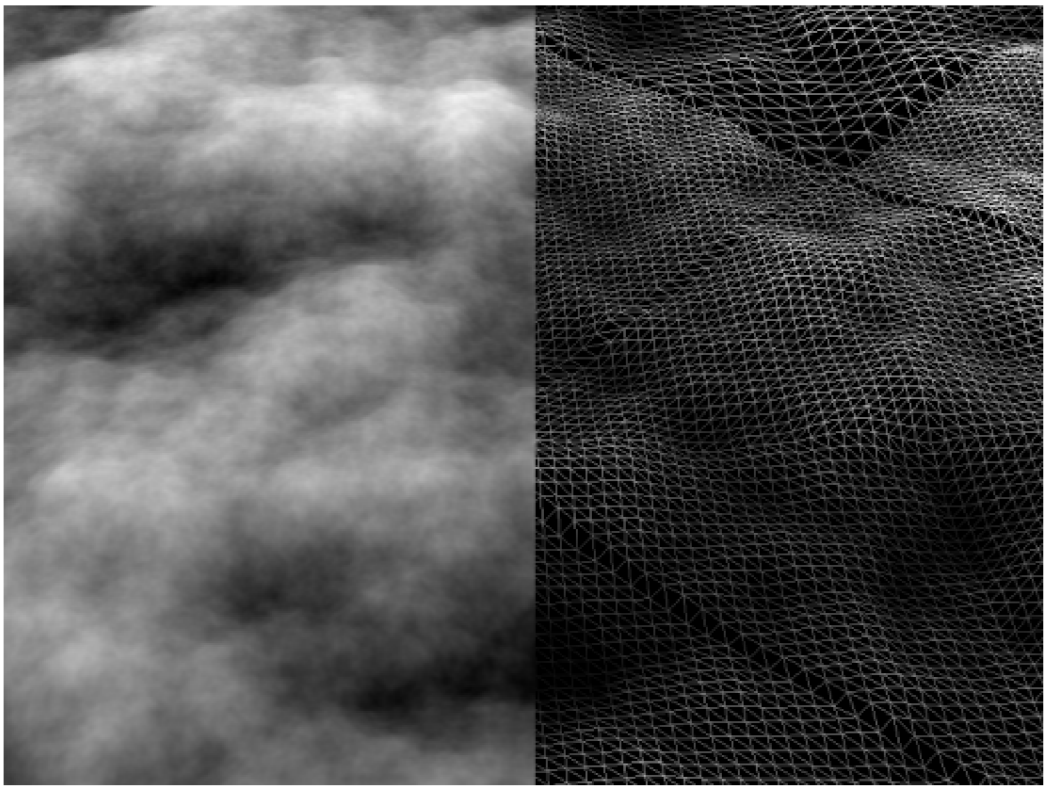
Srovnání výkonu na obou počítačích lze nalézt v následujících tabulkách. Počet snímků za sekundu byl měřen při různých pohledech na terén, aby bylo možné srovnat při různých průchodech hieararchií plátů. Jako varianty pohledů jsem použil pohled **shora** (obrázek 6.2) na celý terén, aby byl celý terén vidět a rozlišení všech plátů bylo minimální. Pohled **z boku** (obrázek 6.4) s kamerou na rohu terénu, aby z terénu byla vidět co největší část, přičemž nejbližší část terénu je v nejlepší úrovni detailu a pohled přibližně **uprostřed** (obrázek 6.3) terénu, směřující k jeho okraji tak, aby terén zabíral celý obraz. Tyto pohledy znázorňují následující obrázky v normálním zobrazení a zobrazení pomocí drátěného modelu, aby bylo vidět rozložení trojúhelníků. V tabulce je pak ukázán hardware na kterém se měřilo,



Obrázek 6.2: Pohled na terén shora.

přibližný počet zobrazovaných trojúhelníků, a počet snímků za vteřinu (Fps) se zapnutým i vypnutým ořezáváním a vykreslováním přes VBO i na přímo. Pravděpodobně kvůli vnitřním procesům knihovny Coin3D dojde po přibližně třiceti sekundám k významnému zvýšení snímků za sekundu. Hodnoty v tabulce jsou naměřeny až po tomto zvýšení.

Jak je vidět z tabulek 6.1 a 6.2 a jak se dalo předpokládat, mají oba měřené optimalizační kroky největší vliv při zobrazení nejvíce trojúhelníků. Ořezávání pohledovým tělesem při generování plátů a potažmo i ořezávání zadní strany objektů, má při pohledu shora



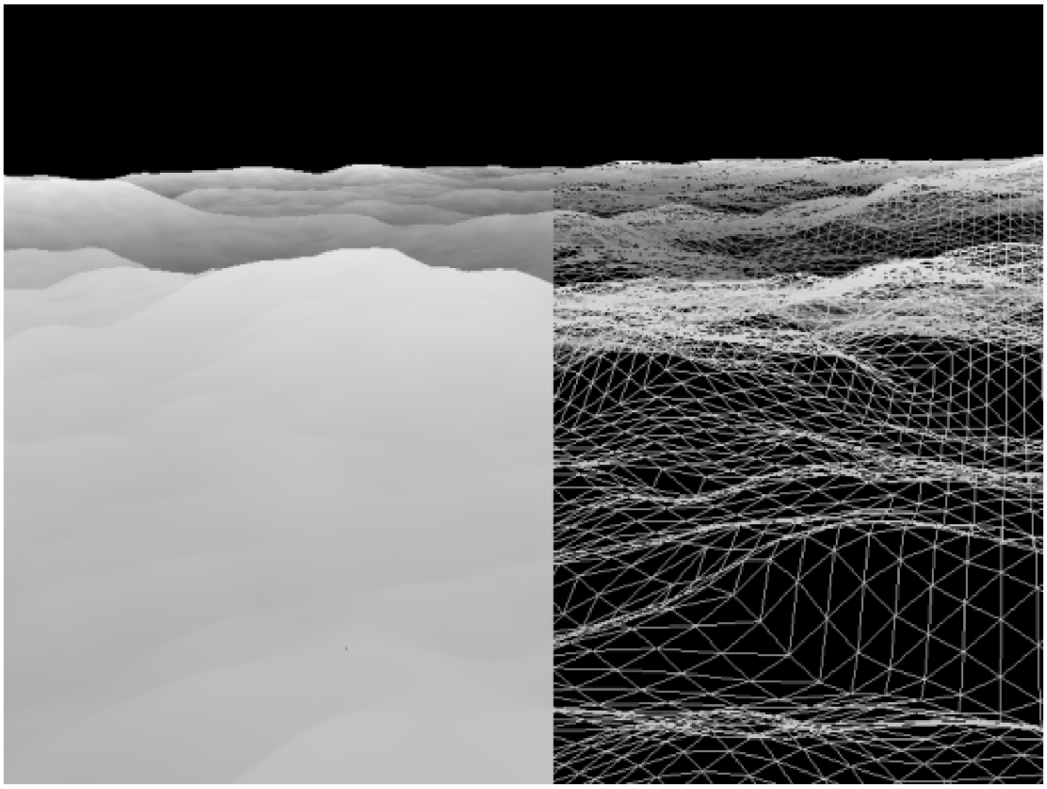
Obrázek 6.3: Pohled na střed terénu.

v průměru vliv přibližně 1% výkonu, při pohledu na střed 112% aa při pohledu z boku 92%. Změna vykreslování z přímého na vykreslování pomocí Vertex Buffer Objektů má při pohledu shora průměrný vliv přibližně 9%, při pohledu na střed 90% a při pohledu z boku 135%.

6.3 Pokračování práce

Jistě by se na této implementaci tohoto algoritmu dalo najít mnoho chyb a nedostatků, které by se daly odstranit. Dle mých předpokladů zejména v částech sloužících k propojení s knihovnou Coin3D. Například by jistě šlo využít některých dalších elementů (potomků třídy **SoElement**) k detekci aktuálních vykreslovacích detailů a přizpůsobit jim vykreslování terénu. Mezi ty poměrně zřejmé bych zařadil například detekci snahy o vykreslování drátěného modelu překrývajícího pevné modely a její realizaci. Dále by šlo také jistě najít několik cest jak dále vykreslování terénu optimalizovat, například generováním seznamu aktivních plátů v jiném vlákně, či úspěšnou implementací generování aktivních plátů pouze při změně kamery.

Kromě pokračování ve zlepšování výkonu vykreslování by šel také doladit vzhled zobrazeného terénu. Dalo by se přidat například stínování a různé materiály na různé části terénu podle barevné mapy nebo přidat novou mapu pro tento účel. Stejně tak by bylo možné přidat například bump mapping pro realističtější vzhle terénu. Tyto změny by však měli negativní dopad na výkon zobrazování a bylo by třeba najít správný poměr mezi kvlaitou zobrazení a výkonem.



Obrázek 6.4: Pohled ze strany terénu.

Pohled	Ořezávání	Počet vrcholů	VBO	Fps
Shora	Ano	8192	Ano	341
Shora	Ne	8192	Ano	338
Shora	Ano	8192	Ne	335
Shora	Ne	8192	Ne	333
Z boku	Ano	121472	Ano	223
Z boku	Ne	336896	Ano	114
Z boku	Ano	121472	Ne	98
Z boku	Ne	336896	Ne	41
Uprostřed	Ano	37632	Ano	382
Uprostřed	Ne	157952	Ano	202
Uprostřed	Ano	37632	Ne	223
Uprostřed	Ne	157952	Ne	82

Tabulka 6.1: Tabulka udávající počet snímků za sekundu na Intel Core2Duo T7300 2,0GHz, 4GB RAM, NVIDIA GeForce 8600M GT.

Pohled	Ořezávání	Počet vrcholů	VBO	Fps
Shora	Ano	8192	Ano	1346
Shora	Ne	8192	Ano	1342
Shora	Ano	8192	Ne	1163
Shora	Ne	8192	Ne	1159
Z boku	Ano	129024	Ano	522
Z boku	Ne	255872	Ano	321
Z boku	Ano	129024	Ne	265
Z boku	Ne	255872	Ne	146
Uprostřed	Ano	37888	Ano	1097
Uprostřed	Ne	100352	Ano	624
Uprostřed	Ano	37888	Ne	703
Uprostřed	Ne	100352	Ne	329

Tabulka 6.2: Tabulka udávající počet snímků za sekundu na Intel Core i7 2600K 3,4GHz, 16GB RAM, NVIDIA GeForce GTX 560 Ti.

Kapitola 7

Závěr

V úvodních kapitolách této práce jsme si uvedli několik algoritmů pro LoD zobrazování rozlehlých terénů. Největší prostor je věnován algoritmu Seamless Patches for GPU-Based Terrain Rendering, který jsem si vybral pro implementaci. Základním principem tohoto algoritmu je rozdělení zobrazovaného terénu na čtyřúhelníkové pláty (patches), z nichž každý sestává ze čtyř předdefinovaných trojúhelníkových dlaždic o různém rozlišení a čtyř pruhů, které tyto dlaždice spojují. Návaznost vedlejších plátů je zaručena tím, že každě dva sousední pláty mají na společné hraně stejné rozlišení. Aplikace textur se provádí pomocí texturových pyramid, které se svoji strukturou podobají mipmapám. Výšková mapa se pak aplikuje přes vertex shader jako displacement.

V další kapitole je obsažen přehled některých knihoven s jejich použitím by se dal daný algoritmus implementovat. Následuje návrh implementace pomocí diagramu užití (use case diagram) a přehledem tříd potřebných pro implementaci tohoto algoritmu a některých významných metod příslušejících k těmto třídám. Pro implementaci ukázkové aplikace jsem si vybral knihovnu Coin3D. Grafické uživatelské rozhraní aplikace je vytvořeno pomocí knihovny Qt, kvůli přenositelnosti výsledné aplikace. Tato knihovna je s knihovnou Coin3D svázána přes knihovnu SoQt.

Implemetace algoritmu Seamless Patches for GPU-Based Terrain Rendering obsahuje jednu hlavní třídu pro komunikaci s knihovnou Coin3D. Dále pak třídu pro správu hierarchié plátů, třídu pro jednotlivé pláty s určením rozlišení jednotlivých dlaždic a singleton třídy pro správu aktivních plátů a předdefinovaných dlaždic a spojovacích pruhů. Během vykreslování třída pro správu hierarchie plátů na základě pozice kamery generuje pláty do seznamu aktivních plátů a určuje jaké rozlišení dlaždic tyto pláty mají mít. Seznam aktivních plátů je poté procházen a jednotlivé dlaždice a spojovací pruhy plátů jsou vykreslovány. Na základě úrovně detailu dlaždic a pruhů je pak vybrána úroveň mipmapy, podle které je proveden ve vertex shaderu displacement jednotlivých vrcholů.

Na konci práce můžeme nalézt výsledky kterých bylo dosaženo v ukázkové aplikaci a některé možné postupy, kterými by se dalo v práci pokračovat.

Literatura

[1] Cignoni, P.; Ganovelli, F.; Gobbetti, E.; aj.: Planet–Sized Batched Dynamic Adaptive Meshes (P-BDAM). In *Proceedings IEEE Visualization*, Conference held in Seattle, WA, USA: IEEE Computer Society Press, Říjen 2003, s. 147–155.

[2] Cigonz, P.; Ganovelli, F.; Gobbetti, E.; aj.: BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization. *Computer Graphics Forum*, ročník 22, 2003: s. 505–514.

[3] Clasen, M.; Hege, H.-C.: Terrain Rendering using Spherical Clipmaps. In *EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization*, editace B. S. Santos; T. Ertl; K. Joy, Lisbon, Portugal: Eurographics Association, 2006, ISBN 3-905673-31-2, ISSN 1727-5296, s. 91–98.

[4] Fakulta Informačních Technologií - Vysoké Učení Technické v Brně: MerlinWiki: OpenSceneGraph [online]. 2011 [cit. 2011-01-05]. Dostupné z: merlin.fit.vutbr.cz/wiki/index.php/OpenSceneGraph.

[5] Gobbetti, E.; Marton, F.; Cignoni, P.; aj.: C-BDAM - Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering. *Comput. Graph. Forum*, ročník 25, č. 3, 2006: s. 333–342.

[6] Hwa, L. M.; Duchaineau, M. A.; Joy, K. I.: Real-Time Optimal Adaptation for Planetary Geometry and Texture: 4-8 Tile Hierarchies. *IEEE Trans. Vis. Comput. Graph*, ročník 11, č. 4, 2005: s. 355–368.

[7] Khronos Group: OpenGL [online]. 1997 [cit. 2012-01-09]. Dostupné z: www.opengl.org.

[8] Kongsberg Oil & Gas Technologies: Coin3D [online]. [cit. 2011-01-05]. Dostupné z: www.coin3d.org.

[9] Livny, Y.; Kogan, Z.; Sana, J. E.: Seamless patches for GPU-based terrain rendering. *The Visual Computer*, ročník 25, č. 3, Březen 2009.

[10] Schneider, J.; Westermann, R.: GPU-Friendly High-Quality Terrain Rendering. *Journal of WSCG*, ročník 14, č. 1-3, 2006: s. 49–56, ISSN 1727-5296.

[11] Torus Knot Software Ltd: OGRE [online]. 2000 [cit. 2011-01-05]. Dostupné z: www.ogre3d.org.

[12] Visualizaton Sciences Group S.A.S.: Open Inventor by VSG: SoComplexity Class Reference [online]. 2012 [cit. 2012-05-18]. Dostupné z: oivdoc90.vsg3d.com/APIS/RefManCpp/class_so_complexity.html.

[13] Wernecke, J.: *The Inventor Toolmaker*. Reading, Massachusetts: Addison Wesley, páté vydání, 1995, ISBN 0-201-62493-1.