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

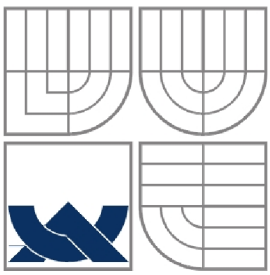
IMPLEMENTACE ALGORITMU PRO ZOBRAZOVÁNÍ TERÉNU S POMOCÍ WEBGL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

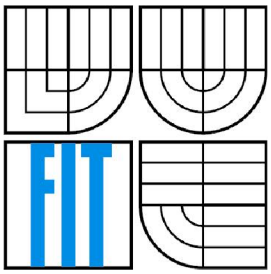
AUTOR PRÁCE
AUTHOR

BC. JAN KALÁB

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 PRO ZOBRAZOVÁNÍ TERÉNU S POMOCÍ WEBGL

IMPLEMENTATION OF WEBGL TERRAIN VISUALIZATION ALGORITHM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. JAN KALÁB

VEDOUCÍ PRÁCE
SUPERVISOR

ING. RADEK BARTOŇ

BRNO 2012

Abstrakt

Tato diplomová práce se zabývá zobrazováním rozsáhlých terénu v reálném čase pomocí WebGL. Práce se také zabývá způsoby měření a reprezentace výškových dat. V práci je také srovnáno několik frameworků pro WebGL a také popisuje praktické využití HTML5 technologií jako například WebWorkers. Kromě toho také srovnává výkon a kompatibilitu současných webových prohlížečů s technogiemi HTML5.

Abstract

This master thesis deals with the large terrain rendering in real time using WebGL. The thesis also deals with ways of measuring and representation of terrain height data. The paper also compares several frameworks for WebGL and also describes the practical use of HTML5 technologies such as WebWorkers. Furthermore, it also compares the performance and compatibility of current web browsers with HTML5 technologies.

Klíčová slova

WebGL, JavaScript, OpenGL, terén, úroveň detailu.

Keywords

WebGL, JavaScript, OpenGL, terrain, level of detail.

Citace

Jan Kaláb: Implementace algoritmu pro zobrazování terénu s pomocí WebGL, diplomová práce, Brno, FIT VUT v Brně, 2012

Implementace algoritmu pro zobrazování terénu s pomocí WebGL

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Radka Bartoně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jan Kaláb
21. května 2012

Poděkování

Ing. Radku Bartoňovi za vedení diplomové práce. Rodině za zázemí. Bc. Petře Kadlecové za korekturu a trpělivost. Bc. Tomáši Radějovi za zapůjčení anaglyfických 3D brýlí. Osazenstvu IRC kanálů #three.js, #geocaching.cz a #fit07 za podporu a pomoc.

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

Obsah

1 Úvod a motivace

Ilustrace 1: Terén z počítačové hry The Elder Scrolls V: Skyrim (Bethesda Game Studios)

Na obrázku výše můžeme vidět snímek obrazovky z počítačové hry Skyrim. Co je na něm zajímavé je, že hra se odehrává v otevřeném prostředí a hráč má možnost přijít k hoře v pozadí a vyšplhat se na ni. Bez jediné nahrávací obrazovky nebo zásadní změny frekvence snímků.

Tento „efekt“ není důležitý jen pro počítačové hry. Stejná funkcionalita je potřeba i pro geografické informační systémy a prakticky všude tam, kde je potřeba interaktivně vizualizovat rozsáhlé terény (například známý Google Earth).

V posledních letech se také začíná mluvit o WebGL – možnosti zobrazení 3D grafiky ve webovém prohlížeči bez nutnosti pluginů. Bude možné pomocí WebGL tuto funkcionalitu implementovat?

V následující kapitole bude vysvětlena teorie týkající se digitálního výškového modelu, včetně sběru dat a jeho tvorby. Ve třetí kapitole najdete informace o WebGL, včetně srovnání několika frameworků pro usnadnění práce. Ve čtvrté kapitole jsou popsány principy level of detail terénu, včetně zvolených algoritmů. V paté kapitole je popsána tvorba a fungování jednoduchého serveru pro generování dlaždic. V šesté kapitole je podrobně popsána implementace zvoleného algoritmu ve WebGL s pomocí frameworku Three.js a také jsou zde představeny další pomocné knihovny. V sedmé kapitole je shrnuta kompatibilita vytvořeného řešení se současnými (i budoucími) verzemi internetových

prohlížečů. Osmá kapitola se věnuje výkonu implementovaného řešení a shrnuje výsledky veřejných testů. V předposlední, deváté, kapitole jsou navrhnuty možná rozšíření vytvořené aplikace. V poslední, desáté, kapitole je pak shrnutí a zhodnocení dosažených výsledků.

2 Digitální výškový model

Abychom mohli zobrazovat nějaký terén, musíme o něm mít dostatek informací. Tou základní je výšková mapa, neboli digitální výškový model (digital elevation model, DEM). Kromě digitálního výškového modelu se také můžeme setkat s digitálním povrchovým modelem (digital surface model, DSM). Rozdíl je v tom, že povrchový model obsahuje i „umělé“ nerovnosti (domy, mosty silnice, lesy, …), zatímco výškový model reprezentuje pouze terén.[1]

Digitální výškový model nemusí představovat jen skutečný terén nějaké planety. Je také možné jej uměle vygenerovat pomocí různých šumových funkcí či jiných postupů. Jejich popis je ale nad rámec této práce, a tak se dále budeme zabývat reálnými daty.

2.1 Způsob uložení digitálního výškového modelu

Stejně jako jinde v počítačové grafice existují dva základní způsoby uložení digitálního výškového modelu: rastrový a vektorový.

Nelze říct, který z přístupů je obecně lepší, záleží na tom, k čemu chceme digitální výškový model použít. Rastrový se více hodí na analýzu terénu a simulace, například sklon, tok vody při záplavách a podobně. Vektorový je zase vhodnější pro vizualizace. V diplomové práci je použit rastrový formát.

2.1.1 Rastrový

Jednodušší ze způsobů uložení je rastrový. Někdy se můžeme setkat s označením výšková mapa. Jedná se o rastr, ve kterém je u každého bodu (pixelu, nemusí být pouze čtyřúhelníkový) uložena jeho nadmořská výška. Tento formát je často výstupem dálkového průzkumu dané planety či jiného tělesa (viz následující kapitola). Důležitou informací zde je rozlišení, neboli kolik skutečných metrů na povrchu představuje jeden pixel. Běžně se setkáváme s rozlišením v řádu desítek metrů.

Pro ukládání rastrových geografických dat je de facto standardem formát GeoTIFF[2]. Jedná se o rozšíření grafického formátu TIFF (Tagged Image File Format), který se používá v profesionální počítačové grafice a DTP pro ukládání 2D obrázků. GeoTIFF tedy umožňuje k obrázku připojit informace potřebné pro jeho georeferencování, například typ projekce, souřadný systém, elipsoid, datum (geodetické) a jiné. Formát TIFF už v základu umožňuje uložit v jednom souboru více obrázků, a tak může být typickým použitím mít v něm uloženou jak výškovou mapu, tak fotomapu a navíc třeba typ půdy či horniny.

2.1.2 Vektorový

Je zřejmé, že v rastrové reprezentaci je spousta bodů „zbytečně“, obzvlášť pokud je terén rovinný. Je tedy možné použít obecné level of detail algoritmy (Delaunayova traingulace[3]), a převést terén na nepravidelnou síť trojúhelníků (Triangulated Irregular Network, TIN[4]). Dat bude významně méně, ale jejich analýza a strojové zpracování se stanou náročnějšími.

2.2 Získávání dat, datasety

Abychom mohli vytvořit digitální výškový model, musíme nejdříve mít výšková data. Existuje několik způsobů jejich získávání: povrchový průzkum nebo dálkové snímání.

2.2.1 Povrchový průzkum

Jedná se o nejstarší známý přístup. Nemusí sloužit jen k měření nadmořské výšky, ale také například polohy. Princip spočívá v tom, že lidé s potřebným vybavením provádějí různá měření v terénu. Potřebným vybavením mohou být teodolity, kompasy, laserová měřidla vzdálenosti, přijímač GPS, a další.

Je jasné, že tento přístup je náročný jak časově, tak na zdroje. Je nutné mít dostatečné množství lidí jak na samotným průzkum, tak na následné zpracování. Na druhou stranu může být toto měření poměrně přesné a také jej lze provádět operativně.

Tento přístup je vhodný pro měření v malé oblasti (město, pohoří, CHKO, …), ale rozhodně s ním nelze zmapovat celou planetu.

2.2.2 Dálkový průzkum

Pro získání digitálního výškového modelu větších oblastí (i planety) je vhodnější použít dálkový průzkum. Ten lze provádět buď letecky nebo z vesmíru. Právě průzkumem z vesmíru vznikly dva v současnosti nejpoužívanější datasety: SRTM (Shuttle Radar Topography Mission) a ASTER GDEM (Advanced Spaceborne Thermal Emission and Reflection Radiometer Global Digital Elevation Model).

Dataset SRTM[5] vznikl v roce 2000 během mise STS-99 raketoplánu Endeavour. Pokryl téměř celou planetu, od 60. rovnoběžky na severu po 56. rovnoběžku na jihu. Měření bylo prováděno technikou Interferometric Synthetic Aperture Radar. Dataset je rozdělený na dlaždice a každá z nich pokrývá vždy jeden stupeň zeměpisné šířky i délky. Rozlišení je přibližně 1 úhlová sekunda (asi 30 metrů), ovšem takto detailně byly zveřejněny pouze pro USA. Pro zbytek světa je rozlišení přibližně 3 úhlové sekundy (tedy 90 metrů). Data však nejsou úplně perfektní, v oblastech s členitým

terénem (pohoří) jsou místa, kde nebyla získána žádná data. Vznikly proto projekty, které se pokusily tyto chyby napravit, buď použitím interpolace, anebo spojením s jinými daty. Jedním z těchto projektů je i CGIAR-CSI[6], který poskytuje opravená data pro celý dataset SRTM, a právě tato data jsou použita při vývoji diplomové práce.

ASTER[7] je název japonského snímače na družici Terra, který snímá povrch planety od roku 2000. Povrch planety je snímán v patnácti pásmech elektromagnetického spektra, od viditelné oblasti až po infračervenou a teplotní. Oproti datasetu SRTM má řadu výhod: větší pokrytí (± 83 °), větší rozlišení (30 metrů všude) a méně „děr“. I když se tento dataset nepodařilo získat, nemělo by to při implemenatci level of detail algoritmů vadit.

3 WebGL

WebGL (Web-based Graphics Library) je softwarová knihovna, která rozšiřuje schopnosti programovacího jazyka JavaScript a umožňuje mu tvorbu interaktivní 3D grafiky v libovolném kompatibilním webovém prohlížeči. Kód WebGL je prováděn na grafické kartě počítače, která musí podporovat shadery. WebGL je kontext HTML elementu `<canvas>` který poskytuje API pro 3D grafiku bez nutnosti použití zásuvných modulů. WebGL je spravováno neziskovou organizací Khronos Group, stejně jako OpenGL.[8], [9]

WebGL je implementací OpenGL ES 2.0, k dispozici je tedy programovatelná grafická pipeline, ovšem omezená o některé funkce. Možnosti WebGL jsou tak zcela identické například s vývojem pro mobilní platformy (Android, iPhone, …). I přepis aplikací z PC do WebGL není příliš náročný, pokud nepoužíváte nějaké pokročilé funkce a hacky.

WebGL je dnes podporován všemi moderními prohlížeči s výjimkou Internet Exploreru (a Opera má pro WebGL speciální testovací verzi). Výhodou také je, že není třeba řešit správu paměti, to má na starost přímo interpret JavaScriptu. Nevýhodou jsou ale problémy s bezpečností. WebGL poskytuje přístup přímo k hardwaru a spoléhá na správnou funkci ovladačů grafické karty. Pokud je v nich nějaká bezpečnostní chyba, je možné ji pomocí WebGL zneužít. Tak je možné například způsobit pád prohlížeče (nebo i zamrznutí operačního systému) pomocí velkých a špatně napsaných shaderů, nebo přistupovat k paměti grafické karty a získávat tak informace o obsahu obrazovky napadeného počítače.[10]

Jak již bylo zmíněno výše, WebGL využívá HTML element `<canvas>` a JavaScript. Nyní si tedy ukážeme, jak s WebGL pracovat. Do těla HTML dokumentu vložíme `<canvas id="glcanvas">Váš prohlížeč nepodporuje canvas!</canvas>`. Pomocí CSS ho můžeme ostylovat jak potřebujeme (pro účely ladění je vhodné přidat rámeček). Nyní v JavaScriptu pomocí následujícího ukázkového kódu 1 nejdříve získáme WebGL kontext do proměnné `gl` a poté pomocí běžných OpenGL příkazů vybarvíme plochu modrou barvou.[11]

```
var canvas = document.getElementById("glcanvas");
var gl = canvas.getContext("experimental-webgl");
gl.clearColor(0, 0, 1, 1);
gl.clear(gl.COLOR_BUFFER_BIT);
```

Kód 1: Inicializace WebGL

Jak jsme mohli vidět, základ je velmi jednoduchý. Nyní již můžeme s WebGL pomocí kontextu `gl` pracovat tak, jak jsme zvyklí. To s sebou přináší i základní nevýhodu: OpenGL je velice nízkoúrovňový přístup k 3D grafice. Pokud chcete např. vykreslit kostku, musíte ze všeho nejdříve vytvořit transformační matici pro perspektivu, poté do vertex bufferu umístit 8 vertexů s popisem jejich spo-

jení a přitom si dávat pozor na orientaci normálových vektorů atd. Přístup je to jistě správný, umožňuje zajímavé triky a optimalizace, ale na hony vzdálený tomu, na co jsou grafici zvyklí z 3D editorů jako Maya nebo Blender. Proto začaly vznikat nejrůznější frameworky, které umožňují právě vysokoúrovňový přístup. Existuje jich spousta, představeny zde budou tři: Three.js, SceneJS a GLGE.

3.1 Three.js

Three.js[12] patří mezi nejlepší a nejrozšířenější frameworky, dobrou referencí pro tento framework budiž spousta profesionálních projektů, videoklipů a grafických dem, které jej využívají (najdete je na stránkách frameworku).

Three.js má velice pěkně a intuitivně vyřešený objektový návrh. Řekněme, že chceme vytvořit červenou krychli:

```
var cube = new THREE.CubeGeometry(10, 10, 10);
var red = new THREE.MeshBasicMaterial({color: 0xff0000});
var redcube = new THREE.Mesh(cube, red);
```

Kód 2: Červená krychle pomocí Three.js

Třída `THREE.Mesh` je potomkem třídy `THREE.Object3D`, a tak poskytuje všechny jeho metody (změna pozice, rotace, velikosti, …). Třída `THREE.CubeGeometry` je zase potomkem třídy `THREE.Geometry`, a tak je možné s ní provádět maticové transformace. Maticovou algebru implementuje třída `THREE.Matrix4` (nebo `THREE.Matrix3`).

S materiály je to podobné. Jsou založeny na tzv. „ShaderChunk“, což je blok kódu, který se stará např. o světlo, stín, texturu nebo mlhu. Jednotlivé materiály (basic, phong, lambert, …) jsou pak z těchto chunků poskládány (chunky jsou běžné JavaScriptové řetězce) do výsledného shaderu.

Three.js poskytuje i nástroje pro graf scény – `THREE.CubeGeometry` je instancí `THREE.Object3D` a ten obsahuje metody `add()` a `remove()`, není tak problém do sebe objekty nořit a tím graf scény vytvářet.

Three.js již obsahuje i základní implementaci pro diskrétní level of detail. Ukázalo se ale, že tato implmentace není použitelná pro terén, neumožňuje totiž vytváření stromové struktury.

Součástí Three.js jsou také připravené třídy pro práci s kamerou, včetně procházení a rozhlížení se pomocí myši. Rozhlížení ale chce trochu cviku, protože JavaScript zatím neumožňuje zachytávání kurzoru myší, jak jsme tomu zvyklí z počítačových her. Řešení je ale blízko, v podobě draftu Mouse Lock API[13].

Pro animace Three.js používá metodu `requestAnimationFrame`[14], která se na rozdíl od běžné herní smyčky provádí maximálně šedesátkrát za sekundu a navíc se pozastaví, pokud panel s webovou stránkou není právě aktivní, čímž šetří zdroje.

Jak vidno, při návrhu Three.js autoři přemýšleli a je velice dobrý a připraven na použití. Třešničkou na dortu (nebo spíš pozůstatkem z historie) je možnost nepoužívat k vykreslování WebGL, ale např. SVG nebo 2D kontext canvasu. Samozřejmě pak vykreslování není tak rychlé a některé věci (shadery) nefungují vůbec. Jediné co by se dalo frameworku vytknout je dokumentace. Mnoho jí není. Na druhou stranu je kód dobře čitelný, existuje spoustu příkladů a i komunita na fóru nebo IRC ochotně poradí a pomůže.

3.2 SceneJS

Druhým zajímavým frameworkem je SceneJS[15]. Jak už je z názvu patrné, staví na grafu scény. K jeho popisu využívá schopností jazyka JavaScript a základní popis je tak spíše deklarativní, než imperativní – graf se vytváří zanořováním polí a objektů, lze využít i formát JSON.

Tento přístup je pěkný, intuitivní, ale občas připomíná „závorkové šílenství“ z jazyka Lisp – objekty jsou do sebe tak zanořené, že úprava stromu se stává velice náročnou, protože nevíte, které závorky můžete smazat. Příliš tomu nenapomáhá ani to, že transformace jako rotace nebo posun jsou také uzly grafu. To sice umožňuje chytře tyto transformace spojovat, ale přístup Three.js je přímočařejší.

Uzly grafu scény mohou být geometrie (součástí frameworku je i několik základních objektů), materiály, textury, shadery, transformace, světla a další v 3D grafice běžné objekty, přičemž platí, že potomci vždy dědí vlastnosti svých předchůdců.

Výhodou je, že se framework snaží právě díky znalosti grafu scény vykreslování urychlit. Důkazem budiž projekt BioDigital Human[16], ve kterém lze zobrazovat veškeré kosti a orgány lidského těla, a právě zde použití grafu scény dává smysl.

3.3 GLGE

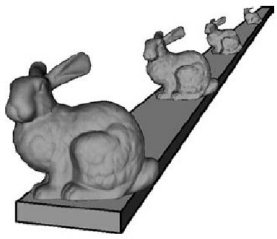
Poslední knihovnou je GLGE[17]. Také se snaží usnadnit práci s WebGL, ale má trochu jiný přístup. Pro popis scény používá XML, který si při startu pomocí AJAXu stáhne a vykreslí. Logika scény (kolize, reakce na kliknutí, interaktivní kamera, …) jsou pak psány v JavaScriptu. To s sebou přináší jeden drobný problém – k vývoji potřebujete mít webový server, protože většina prohlížečů z bezpečnostních důvodů nepovoluje AJAX pomocí pseudoprotokolu file: (toto omezení ale lze pomocí různých skrytých nastavení vypnout).

Tento framework trpí jednou zásadní nevýhodou a tou je dokumentace. Na webu sice naleznete přehled tříd a metod, ale nikde jsem nenašel popis XML formátu pro popis scény. Jediným řešením pak bylo hledání v příkladech, z nichž některé vůbec nefungují.

Z těchto důvodů, a také proto, že podle příkladů nepřináší nic nového oproti dříve zmíněným dvěma, nebyla tomuto frameworku dále věnována pozornost.

4 Level of detail

Level of detail[18] je technika počítačové grafiky sloužící ke snížení detailů geometrie objektu tam, kde jí není potřeba, typicky u vzdálených objektů. U těchto vzdálených objektů totiž dojde k tomu, že velikost některých polygonů se stane menší než bod na zobrazovacím zařízení, a přesto musí být tyto polygony zbytečně zpracovány a uchovávány v paměti. Viz obrázek vpravo: na první pohled se zdá, že se jedná stále o stejného zajíce, počet trojúhelníků však se vzdáleností výrazně klesá (nejbližší obsahuje 69 451 trojúhelníků, nejvzdálenější pouhých 76). Proto se snažíme najít způsoby, jak toto množství geometrie snížit, ideálně tak, aby si toho pozorovatel nevšiml, a v reálném čase.



Ilustrace 2: Ukázka použití level of detail

Obecně lze všechny algoritmy rozdělit na dvě skupiny:

1. Diskrétní
2. Spojité

Diskrétní fungují tak, že je někde řečeno, že v určité vzdálenosti se má zobrazovat tento model a od určité vzdálenosti zase jiný model. Těchto úrovní může být libovolný počet a modely bývají často předpočítány. Problémem tu může být právě přepínání modelů, které působí rušivě. Tento neduh se za určitých okolností dá omezit například alfa blendingem tak, že v okolí přechodových vzdáleností jsou zobrazeny oba modely zároveň a než dojde k přepnutí, modely se plynule prolnou.

U spojitého přístupu je model popsán nějakou funkcí, jejímž parametrem lze určit kvalitu výsledného modelu. Tento přístup je vhodný spíše pro modely popsané isoplochami nebo CSG.

Existuje mnoho přístupů, jak tento problém řešit a opět nelze říct, který je lepší. Některé jsou vhodné pro běžné modely (například z počítačových her), jiné zase pro velice detailní modely získané například laserovým snímáním soch nebo reliéfů. Navíc s příchodem teselace se objevují nové možnosti, jak problém řešit. Teselace ale není ve WebGL možná (nelze ji provádět ani v plnohodnotném OpenGL 2.0), a tak se mu dále nebudeme věnovat. Nás budou nejvíce zajímat algoritmy určené pro zobrazování terénu.

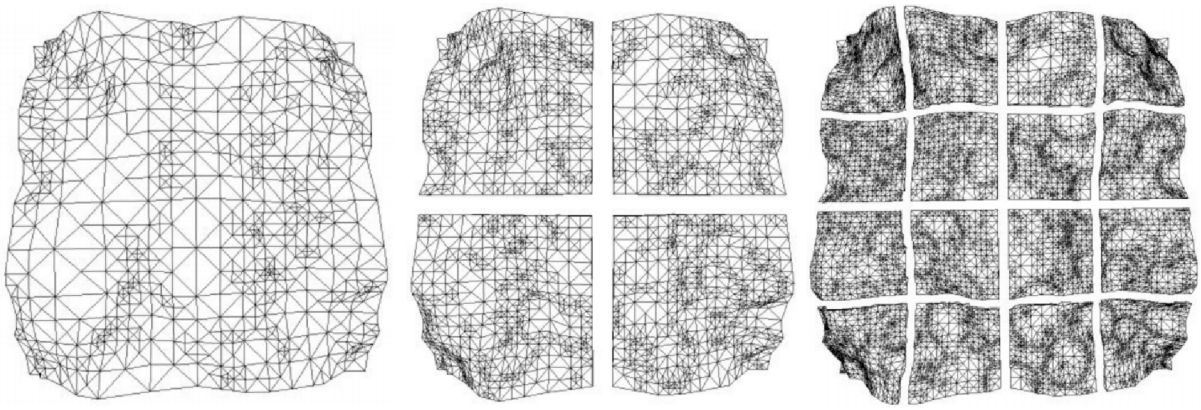
Co je na terénu tak specifického, že pro něj existují speciální level of detail algoritmy?[19] Především jde o to, že nelze brát celý terén jako jeden model, i když jsou vstupní data často jeden obrovský DEM rastrový dataset. To si můžeme bez problémů dovolit například u postaviček nebo předmětů v počítačových hrách, ale ne tady. Většina algoritmů tedy využívá nějakou stromovou

strukturu, ve které jsou uložené jednotlivé dlaždice s různými úrovněmi detailu, a strom se rekurzivně prochází, aby vždy byly viditelné jen ty dlaždice, které jsou třeba.

Ideální vykreslování terénu by mělo umožňovat toto: co možná nejdetailnější terén v blízkosti pozorovatele, ale zároveň by mělo být vidět i například pohoří v dálce. Navíc musí být možné, abychom mohli přejít (nebo přeletět) právě na vrchol onoho pohoří, a opět vidět do údolí, kde jsme byli před chvílí, a to bez nějakého načítání nebo snížení frekvence snímků.

4.1 Chunked LOD

Algoritmus Chunked LOD[20] patří mezi základní a poměrně jednoduché algoritmy. Jeho autorem je Thatcher Ulrich ze společnosti Oddworld Inhabitants (autoři herní série Oddworld) a pochází z roku 2002. Algoritmus je poměrně intuitivní a nejlépe bude k jeho vysvětlení použít obrázek.



Ilustrace 3: První tři úrovně Chunked LOD

Na obrázku výše můžeme vidět základní princip Chunked LOD: terén je postupně dělen na dlaždice s konstantním rozlišením (často 256 bodů), pokrývající sice stále menší plochu terénu, ale čím dál detailněji. Tyto dlaždice jsou uspořádány do stromové struktury (quadtree). Každá dlaždice obsahuje navíc informaci o svém ohraničujícím kvádru (bounding box) a geometrické chybě δ. Pro zjednodušení lze použít rovnici (1), kde L je úroveň dlaždice ve stromu (kořen stromu je 0).

$$\delta(L) = 2\,\delta(L-1) \tag{1}$$

Při vykreslování pak pro každou dlaždici spočítáme geometrickou chybu ρ v prostoru obrazovky pomocí vzorce (2), kde D je nejkratší vzdálenost dlaždice od pozorovatele.

$$\rho = \frac{\delta K}{D} \tag{2}$$

K je perspektivní škálovací faktor spočítaný pomocí rovnice (3).

$$K = \frac{\text{šířka okna}}{2\tan\left(\frac{fov}{2}\right)} \tag{3}$$

Dále si určíme maximální přípustnou geometrickou chybu τ. Nakonec budeme rekurzivně procházet strom, a pokud $\rho <= \tau$, dlaždice se vykreslí, jinak provedeme rekurzivní zanoření.

Dlaždice lze i texturovat, stačí ze zvolené textury vybrat jen tu oblast, která odpovídá dané dlaždici. Také se doporučuje adekvátně zmenšit velikost této textury, aby kořenová dlaždice, která bude zobrazena jen pokud bude pozorovatel velmi daleko od terénu, nebyla zbytečně v plném rozlišení.

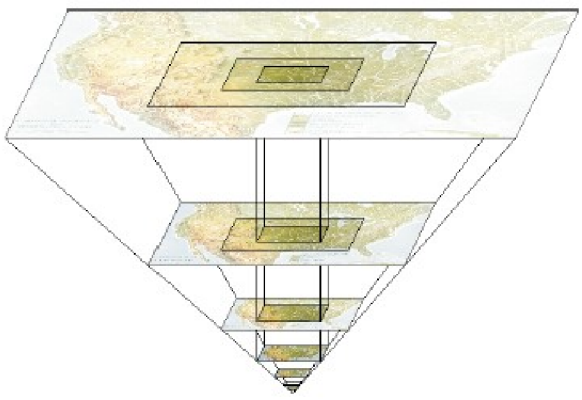
Tento postup má několik nevýhod. Jednou z nich je, že dlaždice s různou úrovní detailu vedle sebe mohou vytvářet díry. Navržené řešení je triviální: přidáme ke každé dlaždici „sukýnku“ (skirt) – vertikální pás kolem kraje dlaždice. Další nevýhodou je, že je potřeba provést předzpracování terénu kvůli vytvoření dlaždic. Z toho plyne i nemožnost změn terénu, dlaždice by se musely znovu vytvářet.

Tento přístup byl zvolen i jako výchozí algoritmus pro diplomovou práci. Důvodem je snadné rozšíření o progresivní načítání dat, jako je tomu například u JPEG nebo PNG obrázků, což je vhodné pro online prostředí. Uživateli se nejdříve zobrazí hrubý terén a detaily se dostahují podle potřeby. Právě toto chování vycházející z online prostředí budou pravděpodobně největší výzvou – asynchronní stahování dat, správa stromu a paměti a v neposlední řadě také rychlost JavaScriptu.

4.2 Geometry clipmaps

Autory tohoto algoritmu jsou Frank Losasso ze Stanfordské univerzity a Hugues Hoppe z Microsoftu. Algoritmus vzniknul v roce 2004.[21], [22]

Principem tohoto algoritmu je pyramida se zmenšujícími se a stále detailnějšími vrstvami geometrie terénu. Nejdetailnější a nejmenší vrstva je samozřejmě nejblíže pozorovateli. Vykreslování probíhá od nejhrubší vrstvy k nejjemnější a to tak, že, pokud chceme vykreslit novou vrstvu, vyřízneme „díru“ do předchozí vrstvy a do ní vložíme vrstvu detailnější. Na vykreslení celé pyramidy bývá často omezené množství času a při správné implementaci



Ilustrace 4: Pyramida s clipmapami

a zajištění atomicity operací (abychom neskončili s „dírou“ bez terénu) je vedlejším efektem to, že při rychlém pohybu pozorovatele, kdy je potřeba aktualizovat velké části všech vrstev, se v nejhorším případě nevykreslí terén s plnými detaily.

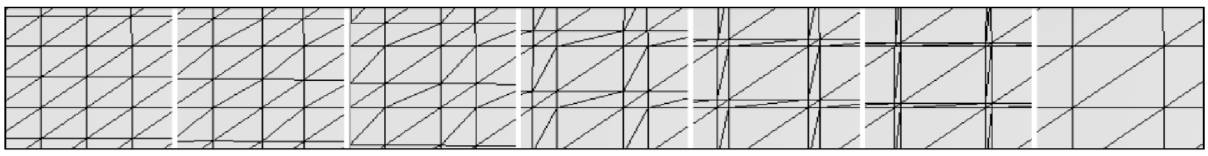
Tento algoritmus lze provádět z velké části na GPU, pokud si výškové mapy uložíme do textur a dále je použijeme jako displacement mapy.

Úzkým místem tohoto přístupu je doplňování dat při posunu pozorovatele. Jak už bylo zmíněno výše, může při něm dojít k tomu, že se nedokončí vykreslování celé pyramidy. Tento problém by byl ještě více umocněn, pokud by se data měla posílat přes internet. Proto tento algoritmus bohužel nebude v práci použit.

4.3 Continuous Distance-Dependent Level of Detail

Continuous Distance-Dependent Level of Detail (CDLOD)[23] funguje podobně jako Chunked LOD – také využívá quadtree pro uložení dlaždic, i matematika pro výběr vhodných dlaždic je prakticky stejná a využívá známou geometrickou chybu a vzdálenost od pozorovatele.

Co je ale na CDLOD zajímavé, je jak odstraňuje problém s „vyskakováním“ jednotlivých dlaždic. K tomu je využit vertex shader, který během posledních 15 až 30 % vzdálenosti dlaždice plynule morfuje její detailnější čtvercovou síť na jemnější. Přesněji, nemorfuje se celá dlaždice, ale pouze ty osmice vrcholů, které jsou v onom 15 až 30% prahu. Teprve až je morfing všech čtyř dlaždic dokončen, zobrazí se dlaždice nová, která je prakticky identická s výsledkem morfování, ale již má čtvrtinu vrcholů. Tím se také vyřeší problém s dírami mezi dlaždicemi s různou úrovní detailu a nejsou tak potřeba sukýnky. Morfing ilustruje obrázek níže.



Ilustrace 5: Morfing sítě algoritmu CDLOD

5 Server pro generování dlaždic

Pokud máme být schopni zobrazovat výšková data pomocí algoritmu Chunked LOD, musíme být schopni vytvářet dlaždice s konstantním rozlišením, ale pokrývající stále menší a menší oblast terénu. Za tímto účelem byl vytvořen server v jazyce Python verze 2.x.

Jazyk Python byl zvolen především kvůli jeho vysoké abstrakci, „nucení“ k psaní přehledného kódu a dostupnosti potřebných knihoven. Důvodem k použití verze 2.x byla nedostupnost modulu pro práci s formátem GeoTIFF pro Python 3 (viz dále).

5.1 HTTP server v Pythonu

Protože pracujeme v prostředí internetu, bude server pracovat s protokolem HTTP, pro který Python obsahuje modul `BaseHTTPServer`[25] ve kterém jsou třídy `HTTPServer` a `BaseHTTPRequestHandler`, které obstarávají jak samotnou síťovou komunikaci přes HTTP protokol, tak zpracování jednotlivých požadavků.

Jednoduchý HTTP server bude tedy vypadat nějak takto:

```
from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler

class MyHandler(BaseHTTPRequestHandler):
      def do_GET(self):
            self.send_response(200)
            self.send_header("Content-type", "text/plain")
            self.end_headers()
            self.wfile.write("Hello world")

HTTPServer(("", 8000), MyHandler).serve_forever()
```

Kód 3: Jednoduchý HTTP server v Pythonu

Na začátku si naimportujeme potřebné třídy a moduly. Následně si nadefinujeme třídu `MyHandler`, které rozšiřuje třídu `BaseHTTPRequestHandler`. Tato třída obsahuje metody `do_GET`, `do_POST` a podobné pro jednotlivé HTTP požadavky. Pro každý z těchto požadavků tak můžeme definovat různé chování. Tělo metody `do_GET` musí být zřejmé každému, kdo alespoň trochu zná protokol HTTP. Na konci spustíme samotný `HTTPServer` na portu 8000 a nastavíme třídu `MyHandler` jako obslužnou třídu pro příchozí požadavky.

Pokud tento kód spustíme a otevřeme si ve webovém prohlížeči adresu http://localhost:8000, objeví se nám známý řetězec Hello world.

Třída `BaseHTTPRequestHandler` obsahuje proměnnou `path`,ve které je uložena celá URL požadavku. Podle ní se tedy můžeme rozhodovat, zda máme načíst HTML stránku, CSS styl,

nebo JavaScriptový kód. Pro další zpracování proměnné `path` je vhodné použít modul `urlparse`, který URL rozdělí na protokol, doménu, cestu, soubor, GET parametry a další.

5.2 Získání výškové mapy z GeoTIFFu

GeoTIFF[2] je formát metadat, který umožňuje georeferencovat rastrová data ve formátu TIFF. Pro jeho zpracování je nejpoužívanější knihovna GDAL[24]. Tato knihovna obsahuje podporu pro různé programovací jazyky, včetně Pythonu (bohužel pouze pro verzi 2.x, alespoň v Ubuntu 11.04).

Načtení výškové mapy z testovacích je poměrně triviální:

```
import gdal

dataset = gdal.Open("map.tiff", GA_ReadOnly)
dem = dataset.ReadAsArray()
```

Kód 4: Načtení výškové mapy z GeoTIFFu

Nejdřívě opět naimportujeme potřebný modul. Poté otevřeme GeoTIFF soubor pro čtení. Nakonec jej načteme jako pole. V proměnné `dataset` jsou kromě samotných dat uloženy také metadata, jako například název souboru, rozměry, počet vrstev a další.

Tento postup je sice nejjednodušší, ale není zcela optimální vzhledem k paměťové náročnosti. Metoda `ReadAsArray` totiž načte celý dataset do paměti. A takové datasety nebývají nejmenší. Náš server tedy po spuštění a načtení 500 MB datasetu ČR a okolí v rozlišení 60 metrů zabírá v paměti přibližně 600 MB a trvá několik sekund než se celý načte. Na druhou stranu, velikost operační paměti se dnes pohybuje v řádech GB, takže by to nemusel být nijak závažný nedostatek.

5.3 Zpracování výškové mapy

Nyní tedy máme načtenou výškovou mapu ve dvojrozměrném poli – každý záznam obsahuje nadmořskou výšku. Dále z něj potřebujeme dělat výřezy a provádět interpolaci. K tomu se výborně hodí moduly `NumPy` a `SciPy`[26] sloužící pro numerické a vědecké výpočty v Pythonu.

Nejdříve ale výřez. K tomu můžeme snadno použít indexování polí v Pythonu. Pokud při získávání dat z pole použijeme dvojtečku, dostaneme všechny prvky v daném rozsahu. Tedy například `array[2:5]` získá čtyři prvky, a to druhý až pátý. Pokud tedy známe počátek výřezu (pravý horní roh, proměnné `x` a `y`) a výšku a šířku výřezu (proměnné `w` a `h`), provedeme výřez následujícím způsobem:

```
tile = dem[x:x+w, y:y+h]
```

Kód 5: Získání výřezu z výškové mapy

Tento výřez ale rozhodně nemusí mít námi požadované rozměry! Proto musíme provést podvzorkování. K tomu vyžijeme metodu `scipy.ndimage.interpolation.zoom`:

```
scipy.ndimage.interpolation.zoom(tile, float(r) / tile.shape[0],
order = 1)
```

Kód 6: Zmenšení a interpolace výškových dat

Použití je poměrně zřejmé, jediná zajímavá část je druhý parametr: `float(r)/tile.shape[0]`, který dokumentace označuje jako „zoom factor“. Tento parametr říká, jako moc se má dlaždice zmenšit (nebo zvětšit). Proměnná `r` zde určuje cílovou velikost dlaždice (v našem případě 256 bodů) a `tile.shape[0]` pak velikost vstupní dlaždice. Dostaneme tedy takové číslo, kterým když vynásobíme vstupní rozměr dlaždice, bude výsledná dlaždice mít hranu 256 bodů. Poslední parametr `order` pak určuje řád interpolačního polynomu. Vzhledem k rychlosti je zvolen 1, čili lineární interpolace.

5.4 Výstupní formát

Výstupním formátem měl být původně JSON, ale okolnosti a technické obtíže nakonec vedly k tomu, že byl použit grafický formát PNG.

5.4.1 JSON

JSON (JavaScript Object Notation) je textový formát, jehož gramatika je ekvivalentní (jak už zkratka napovídá) zápisu objektů v JavaScriptu. Základem jsou jednoduchá pole, ohraničená hranatými závorkami, a asociativní pole, ohraničená složenými závorkami. Oba tyto typy pak obsahují čárkami oddělený seznam hodnot, asociativní pole pak navíc před každou hodnotou obsahuje dvojtečkou oddělený její název jako řetězec v uvozovkách. Hodnotou pak může být číslo, pole, asociativní pole nebo řetězec.

Tento formát je posledních několik let velmi rozšířený mezi vývojáři webových služeb a prakticky nahradil dříve používaný formát XML. Jeho výhodou oproti XML je především snadnější strojové zpracování, lepší lidská čitelnost a menší velikost. Překvapivě, právě velikost a strojové zpracování se zde stalo kamenem úrazu a vedlo k použití formátu PNG.

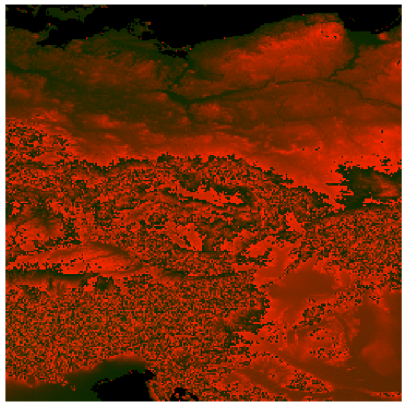
Co se velikosti týče, víme, že budeme používat dlaždice o hraně 256 bodů. Předpokládejme, že nadmořská výška se pohybuje v řádu stovek metrů. Každá hodnota tedy obsahuje tři číslice, plus většina z nich ještě čárku a mezeru jako oddělovač, tedy dohromady pět znaků. Každý znak je reprezentován jedním bajtem. Každá dlaždice tedy bude mít objem 256 × 256 ×5 = 327 680, tedy přibližně 330 kB. To se může na první pohled zdát jako poměrně malá velikost. Pokud ale vezmeme v potaz, že

Chunked LOD je založen na čtyřstromu (quadtree) těchto dlaždic, dostáváme poměrně velký objem dat (přes 1 MB pro každé čtyři poddlaždice), který musíme přenést po síti k cíli.

A co se strojového zpracování týče, byl velký problém s rychlostí vytvoření geometrie dlaždice z těchto dat. Tomuto problému se bude více věnovat kapitola 6.2.

5.4.2 PNG

PNG (Portable Network Graphic) je bezeztrátový formát pro ukládání obrázků. Tento formát umožňuje ukládat obrázky v různých barevných hloubkách, včetně alfa kanálu (průhlednosti). Formát PNG umožňuje i uložení v 16 bitech odstínů šedi, což by pro aplikaci bylo ideální. Bohužel se ukázalo že WebGL (OpenGL ES) neumožňuje použití textur v takovém formátu. Proto musíme zůstat u formátu RGB s 8 bity na každý kanál a 16 bitů do něj zakódovat.



Ilustrace 6: Základní dlaždice v RGB formátu

Důvody k použití PNG byly dva. V OpenGL lze použít textury jako displacement mapy pro geometrie (viz kapitola 6.2.2), což je podstatně rychlejší než tvorba nové geometrie podle JSON dat. Velikost jednotlivých dlaždic se pohybuje okolo 100 kB, protože formát PNG umožňuje (a doporučuje) použití komprese. Navíc, použití obrázků umožňuje snadnější kontrolu výsledku než textový soubor se spoustou čísel.

Python přímo neobsahuje podporu pro formát PNG. Existuje ale modul `PIL` (Python Imaging Library)[27], který umožňuje práci s tímto (a dalšími) formáty.

Před zápisem dlaždice do souboru je ale nutné ji ještě upravit. V oblastech s vodní hladinou nedokáží družice nadmořskou výšku správně změřit, a tak je často vyplněna nějakou velmi zápornou hodnou (nejčastěji −9 999). Do obrázku ale nelze záporné číslo zapsat, a tak je nejdříve nutné výškovou mapu posunout o onu zápornou nadmořskou výšku nahoru. Velké vodní plochy pak mají nadmořskou výšku 0 (černá barva). Protože je ale pole obsahující výšková data ve 32 bitech, pracuje s nimi PIL jako s formátem RGBA. Ve skutečnosti PIL pracuje s kanály v opačném pořadí, tedy alfa kanál nejvyšších 8 bitů, červený kanál nejspodnějších 8 bitů. Proto ještě ke výšce připočteme hodnotu `0xff000000`, jinak byl obrázek zcela průhledný.

Celý zápis výškových dat do PNG souboru tedy vypadá následovně::

```
imagedata -= dem.min()
imagedata += 0xff000000
pilImage = Image.frombuffer('RGBA', (r, r), imagedata,
'raw', 'RGBA', 0, 1)
pilImage.save(path)
```

Kód 7: Zápis obrázku do PNG

Při použití takto získané výškové mapy pak výslednou výšku získáme jednoduchým výpočtem: 255G + R. Modrý kanál můžeme zanedbat, protože takto velkou nadmořskou výšku na planetě Zemi nikde nenajdeme. Pokud by ale byla potřeba, stačí pouze modrý kanál vynásobit 255^2.

Výsledek (po drobných úpravách) můžete vidět na obrázku v úvodu kapitoly.

5.5 Cache

Všechny výše zmíněné kroky pro vytvoření dlaždice nějakou dobu trvají. Navíc by bylo zbytečné dělat je stále dokola pro již existující dlaždice (např. při opětovném spuštění serveru nebo připojení více klientů). Proto bylo vhodné použít nějakou formu jednoduchého kešování dlaždic.

Kešování je poměrně jednoduché a přímočaré. Nejdříve zjistíme, zda v keši již požadovaná dlaždice neexistuje. Pokud ne, vytvoříme jí postupy popsanými v předchozích kapitolách. Nyní si již můžeme být jistí že dlaždice existuje, takže ji můžeme z keše otevřít a poslat klientovi, který ji požaduje.

Pro práci s dočasnými soubory Python poskytuje modul `tempfile`, který navíc umožňuje i různé zabezpečení takto uložených dat, ale to nás nemusí zajímat. Pro kešování je důležitější metoda `gettempdir`, která vrátí na platformě závislou cestu k dočasnému adresáři (v Linuxu `/tmp`), a do něj se kešované dlaždice ukládají.

Jakmile jsou dlaždice jednou připravené v keši, je jejich načítání prakticky okamžité a jediné co zdržuje jejich načítání je následné zpracování v prohlížeči. Dlaždice jsou navíc po serveru požadovány jako běžné soubory (např. http://localhost:8000/chunk/0_6000_6000_300_300_256.png), takže další výhodou je, že stačí cache zkopírovat a terén lze prezentovat i bez použití serveru. Formát názvu souboru je následující:

1. Úroveň ve čtyřstromu (0 je nejvyšší)
2. X souřadnice počátku
3. Y souřadnice počátku
4. Šířka výřezu
5. Výška výřezu
6. Rozlišení

Nyní již známe vše potřebné pro vytvoření jednoduchého serveru pro tvorbu dlaždic. K takovému serveru není ani potřeba příliš mnoho kódu, ani ne 100 řádků. V tom je velká síla Pythonu jako vysokoúrovňového jazyka – programátor nemusí přemýšlet *jak* něco napsat, jako spíš *co* chce napsat.

6 Chunked LoD v Three.js

V úvodních kapitolách už byly zmíněny základní vlastnosti knihovny Three.js. Nyní se pojďme blíže podívat na praktické použití.

6.1 Vytvoření jednoduché scény

Aby bylo možné pomocí WebGL něco vykreslovat, je nejprve nutné vytvořit `<canvas>` a získat z něj `experimental-webgl` kontext. I od tohoto nás Three.js odstiňuje díky principu rendererů. Jak již bylo zmíněno výše, Three.js původně umožňoval vykreslování jen pomocí 2D canvasu, a až postupem času přibyla podpora pro WebGL, která je nyní prioritou. I když stále vznikají různé experimentální renderery, například renderer do ASCII artu, nebo vlastní implementace softwarového vykreslování pomocí Image API. Pro vytvoření WebGL rendereru si tedy vytvoříme novou instanci objektu `THREE.WebGLRenderer`. Jako parametr mu lze předat některé základní parametry, například zapnout vyhlazování. Také mu lze jako parametr předat `<canvas>` element. To ale není nutné, protože si v opačném případě renderer vytvoří `<canvas>` vlastní. Posledním krokem je nastavit velikost plátna metodou `setSize(w, h)` a `<canvas>` element vložit do DOM stromu dokumentu (například pomocí jQuery jako `$("body").append(renderer.domElement)`).

Nyní máme připravený renderer, který ale zatím nic nekreslí. (Doslova nic, protože k vykreslení je potřeba zavolat metodu `render`, viz dále.) Dalším nutným krokem je připravit si scénu, tedy vytvořit instanci objektu `THREE.Scene`. Scéna vlastně není nic jiného, než objekt který udržuje strom objektů (graf scény), světla a kamery. Scéně lze také nastavit mlhu, aby objekty v dálce plynule mizely.

Dále vytvoříme nějaké objekty, které chceme vykreslit. Například pomocí kódu 2 z kapitoly 3.1, a tyto objekty pomoci metody `add` do scény přidáme. Stejným způsobem můžeme do scény přidat i světla.

Stejným postupem do scény přidáme i kameru. Nejdříve kameru vytvoříme z objektu `THREE.PerspectiveCamera` (z názvu je patrné, že je možné použít i jiné, například ortogonální, kamery), kterému jako parametry zadáme zorný úhel, poměr stran a začátek a konec ořezové roviny. Protože jsou objekty v počátku, bylo by vhodné kameru posunout, aby byly objekty hezky vidět. K tomu mají všechny objekty parametr `position`, což je třísložkový (`x`, `y` a `z`) vektor určující polohu. Poloha se určuje relativně podle rodiče v grafu scény. Dále by bylo vhodné, aby se kamera dívala přímo na objekt. Je tedy nutné kamerou otočit. Samozřejmě si můžeme vypočítat transformační matici, nebo využít metodu `lookAt` a klidně jí jako parametr předat parametr `position`

objektu, na který chceme aby, se kamera dívala. Na konci opět pomocí metody `add` kameru přidáme do scény.

Nyní je již vše připravené k vykreslení scény – objekty, světla a kamera jsou vloženy do scény. Samotné vykreslení provedeme zavoláním metody `render` rendereru. Jako parametry mu musíme předat scénu a kameru, kterou chceme scénu vykreslit.

6.1.1 Animace a pohyb ve scéně

Vykreslili jsme tedy jeden snímek. To ale zdaleka nestačí na pěkné vykreslování terénu. Bylo by dobré, kdyby nad terénem šlo létat. K tomu poskytuje Three.js takzvané controls. Jedná se o obecný postup, jak na základě vstupů z klávesnice a myši pohybovat s libovolným objektem, typicky s kamerou.

To ale trošku předbíháme, nejdříve musíme zajistit, aby se scéna stále dokola vykreslovala. K tomu slouží již dříve zmiňovaná funkce `requestAnimationFrame`[14], která jako parametr obsahuje funkci, jež se má co nejdříve (ideálně za šedesátinu sekundy) zavolat. Tato funkce typicky obsahuje další volání `requestAnimationFrame`, čímž dosáhneme plynule animace při šedesáti snímcích za sekundu. To, že se funkce volá právě po šedesátinách sekundy, již zajišťuje prohlížeč. Připomeňme taky, že tento postup má oproti běžné herní smyčce ještě jednu výhodu: při ztrátě fokusu (například při přepnutí na jiné okno nebo záložku) se animace pozastaví, a nevytěžuje tak CPU. Three.js navíc obsahuje polyfill[14] pro tuto funkci, který zkontroluje, zda je funkce v prohlížeči dostupná (přeci jen se jedná o funkci z draftu HTML5), případně zda je dostupná nějaká její `-moz-` nebo `-webkit-` varianta, a pokud není, použije běžnou herní smyčku. Three.js dále obsahuje objekt `THREE.Clock`, který slouží k měření času, především pak k měření rozdílu času od posledního zavolání metody `getDelta`, čehož využívá control objekt pro zamezení velkých skoků právě v případech, kdy je provádění funkce `requestAnimationFrame` pozastaveno.

Konečně můžeme přistoupit k vytvoření controleru. Těch opět existuje několik typů, podle toho jak chceme kameru ovládat. Zda chceme kamerou pohybovat ve stylu počítačových her jako Quake, zda chceme kamerou jen otáčet okolo nějakého objektu, nebo zda chceme udělat průlet kamery po nějaké křivce. My budeme chtít první možnost, což implementuje `THREE.FirstPersonControls`, kterému předáme jako parametry objekt, se kterým chceme pohybovat (kameru) a DOM objekt, ve kterém chceme reagovat na myš (typicky stejný jako renderer). Controleru můžeme nastavovat parametry jako rychlost pohybu, tlumení a podobně. Dále do funkce, která vykresluje snímek musíme připsat volání `control.update(clock.getDelta())`, což provede zpracování pohybu kamery podle stisklé klávesy a polohy myši nad rendererem.

Ovládání se pak provádí klávesami `W`, `A`, `S`, `D` pro pohyb do stran a `R` a `F` pro svislý pohyb, nebo kurzorovými šipkami. Poslední možností je použít levé a pravé tlačítko myši. Poněkud nepřirozené se může z počátku zdát rozhlížení myší. Jak již bylo zmíněno dříve, JavaScript umožňuje zachytávání myši pouze pomocí experimentálního API[13], které zatím není v Three.js (a ani ve většině prohlížečů) implementováno.

6.2 Vykreslování terénu

Když už máme server na generování dlaždic a dokážeme vykreslit jednoduchou scénu, můžeme přistoupit k vykreslování terénu.

Cílem tedy je vykreslit terén jako mřížku, kde výška jednotlivých průsečíku bude odpovídat výšce z výškové mapy. Dále je nutné těchto mřížek vytvořit a vykreslit mnoho, pro každou dlaždici jednu. To se může zdát jako triviální problém, nakonec to ale byla nejnáročnější část celé práce, bylo vyzkoušeno několik způsobů a kód byl mnohokrát přepsán, než bylo nalezeno optimální a nejrychlejší řešení.

Vytvoření samotné mřížky je vzhledem k postupům popsaných v minulých kapitolách jednoduché: stačí vytvořit nový mesh, který bude mít jako geometrii instanci objektu `THREE.PlaneGeometry`. Tomuto objektu v konstruktoru předáme parametry rozměr a počet řezů. V počátečních verzích bylo dále nutné geometrii ještě otočit, protože původně byla generována v osách XY (svisle). Autoři Three.js si ale v březnu řekli, že vodorovná orientace je intuitivnější, takže potřeba transformace zmizela.

6.2.1 JavaScript a JSON

Jak již bylo zmíněno v kapitole 5.4 o tvorbě dlaždic, prvním nápadem bylo posílat výšková data ve formátu JSON a podle nich vytvářet nové dlaždice. Postup byl jednoduchý:

1. Poslat s pomocí jQuery AJAX dotaz serveru
2. Až server vrátí JSON data, předat je jako parametr upravenému objektu `THREE.PlaneGeometry`.

Pojďme se nyní podívat, jak Three.js pracuje s geometriemi. V jádru Three.js je objekt `THREE.Geometry`, který všechny další konkrétní geometrické objekty (krychle, koule, toroid, …) používají jako svůj prototyp. V něm jsou uvedeny základní parametry každé geometrie: pole vertexů a z nich vzniklých tří/čtyřúhelníků. K uložení každého vertexu a n-úhelníku používá Three.js vlasní objekty `THREE.Vector3` (třísložkový vektor) a `THREE.Face4` (případně `Face3`), které kromě samotných hodnot obsahují ještě různé pomocné metody (například vektorovou algebru) nebo dodatečné informace, které se k nim vážou (normála tří/čtyřúhelníku, UV texturovací souřadnice, …).

Kromě toho je také možné do geometrie uložit i cíle animace (morphingu), materiály či barvy pro jednotlivé vrcholy a další. Při tvorbě geometrického objektu jsou pak tyto hodnoty vždy vypočítány a naplněny do polí.

Konkrétně u `PlaneGeometry` vypadá tvorba geometrie tak, že se nejdříve předpočítají různé pomocné proměnné (například šířka jednoho segmentu mřížky a podobně). Pak se ve dvou zanořených for cyklech vypočítají pozice vrcholů. Dále se v dalších dvou zanořených for cyklech vypočtou čtyřúhelníky včetně normál (používá se stále stejná) a UV texturovacích souřadnic. Na konci se ještě kvůli kompatibilitě s 2D vykreslováním vypočtou „centroidy" (jeden for cyklus). Při použití výškových dat se pak pouze upraví vytváření vrcholů tak, aby braly v potaz data z výškové mapy, a následně se místo výpočtu „centroidů" zavolá pomocná funkce na výpočet normál, protože nyní již nebudou všechny stejné. Stále ale dostáváme kvadratickou časovou složitost $O(n^2)$.

Testy ukázaly, že vytvoření jedné jediné dlaždice o hraně 256 segmentů trvá 500 milisekund a více (není započítána doba pro získání výškové mapy) v prohlížeči Google Chrome používajícím interpret V8[37], který je v současnosti nejrychlejším interpretem JavaScriptu. V prohlížeči Mozilla Firefox byly časy ještě horší. To rozhodně není přijatelné pro naši aplikaci, která by měla běžet při šedesáti snímcích za sekundu!

Protože je ale řešení v čistém JavaScriptu nejjednodušší, zkusme jej ještě trošku zrychlit použitím vláken.

6.2.1.1 WebWorkers

Jednou z novinek HTML5 je i API jménem WebWorkers, které umožňuje pracovat s vlákny ve webových stránkách. To dovoluje vývojářům provádět výpočetně náročné operace bez „zaseknutí" samotné webové stránky. To se výborně hodí i do naší aplikace, protože potřebujeme zajistit co možná nejplynulejší ovládání, i když bude aplikace vytvářet dlaždice.

Pojďme si nyní přiblížit použití a fungování WebWorkers.[28] Prvně je třeba zmínit, co WebWorkers neumí. Tím nejdůležitějším je práce s DOM. Není tedy možné přímo z WebWorkeru upravovat webovou stránku. Dále WebWorker neobsahuje objekt `window` (pouze některé jeho metody, například `setTimeout`) stejně tak má velice omezený přístup k objektu `navigator` (obsahuje jen řetězce popisující prohlížeč a jeho verzi). WebWorker ovšem smí zasílat AJAX požadavky.

WebWorkers fungují na principu zasílání zpráv – vytvořenému WebWorkeru můžeme zprávy zasílat, a on nám nějaké zprávy vrací. Zprávami může být cokoliv, co lze použít ve formátu JSON (čísla, řetězce, pole, slovníky a jejich kombinace). Také lze použít typovaná pole pro čistě binární data. JavaScript jako takový typy v polích nehlídá a umožňuje vytvářet heterogenní pole. Kvůli WebGL ale do JavaScriptu přibyly pole typovaná, protože pomocí nich se do WebGL předává

například seznam vrcholů. Z WebWorkeru nelze vracet objekt s metodami! To se ukázalo jako kritický nedostatek.

Nový WebWorker vytvoříme takto:

```
var worker = new Worker("GreetWorker.js");
```

Kód 8: Vytvoření WebWorkeru

Tím načteme kód WebWorkeru ze souboru `GreetWorker.js`. Tento soubor není třeba nikde zmiňovat v HTML stránce. Pro načtení tohoto souboru platí stejná pravidla jako pro AJAX, třeba same-origin policy nebo standardně nelze provozovat WebWorkers přes `file` protokol.

Dále vytvořenému WebWorkeru nastavíme listener pro příchozí zprávy:

```
worker.onmessage = function(event) {
      console.log(event.data);
};
```

Kód 9: Listener pro příchozí zprávy

Teď už zbývá jen WebWorkeru nějakou zprávu poslat:

```
worker.postMessage("Karel");
```

Kód 10: Poslání zprávy WebWorkeru

Jak jsme již zmínili, zprávou by mohlo být klidně pole nějakých hodnot nebo JSON objekt.

Samotný Webworker pak bude vypadat podobně, také bude obsahovat implementaci metody `onmessage` a `postMessage`:

```
onmessage = function(event) {
      self.postMessage("Hello " + event.data + "!");
};
```

Kód 11: GreetWorker.js

GreetWorker tedy přijme jméno a odpoví pozdravem.

WebWorkery lze ukončit buď z hlavního vlákna zavoláním metody `terminate`, nebo se může WebWorker ukončit sám zavoláním metody `close`. Při použití varianty close se zavolá přiřazená metoda `onclose`. Při použití `terminate` je vlákno „na tvrdo“ ukončeno, bez jakékoliv možnosti reakce.

Hlavní myšlenkou tedy bylo z JSON dat vytvořit geometrii ve WebWorkeru, a tím odlehčit hlavnímu vláknu, které řeší vykreslování. Problém ale nastal právě s tím, že WebWorker nedokáže vracet objekty s metodami. Jenže v Three.js jsou vrcholy i n-úhelníky objekty s metodami! Pomocí WebWorkers tedy není možné vytvářet Three.js geometrie. Respektive, lze je sice ve WebWorkeru vytvořit, ale nelze je poslat jako zprávu zpět.

Byla snaha ve WebWorkeru předpočítat potřebné hodnoty, ty poté vrátit a objekt rekonstruovat, ale nebyla úspěšná. Stále bylo nutné procházet struktury zanořenými for cykly, a složitost stále byla $O(n^2)$. Na stejném principu fungují i loadery z různuch formátů v Three.js, včetně loaderu z vlastního JSON formátu, který používají exportní skripty z různých 3D editorů. Celkový čas se sice nepatrně snížil, v nejlepších případech na 300 ms, to ale stále není dostatečné.

Tímto tedy definitivně padla možnost vytvářet dlaždice pomocí čistého JavaScriptu a bylo nutné najít jiné a rychlejší řešení. Tímto rychlým řešením jsou OpenGL shadery.

6.2.2 Shadery

Shadery jsou obecně řečeno programy, které se provádí buď pro každý vrchol geometrie (vertex shadery), nebo pro každý pixel obrazu (fragment shadery). Existují i další shadery (například geometry shader, umožňující přidávat a odstraňovat vrcholy a tím provádět teselaci), ty ale nejsou ve WebGL (OpenGL ES) dostupné.

Shadery se provádějí na grafické kartě, a proto jsou extrémně rychlé. Pro programování shaderů se používají jazyky GLSL (pro OpenGL) a HLSL (pro DirectX). Oba ale mají syntaxi velice podobnou jazyku C, obohaceným o datové typy a funkce pro vektorové a maticové operace.

Ve WebGL se shadery vytvářejí zcela stejně jako v běžných OpenGL programech – použitím příkazů (ve skutečnosti jsou to metody získaného 3D kontextu):

- `createShader` pro vytvoření nového objektu shaderu, parametrem se určuje zda půjde o vertex nebo fragment shader.
- `shaderSource` pro načtení zdrojového kódu shaderu, parametry jsou shader objekt (vytvořený `createShader`) a řetězec obsahující zdrojový kód.
- `compileShader` pro přeložení zdrojového kódu, parametrem je opět shader objekt.
- `createProgram` pro vytvoření programu. Program je kombinací více shaderů.
- `attachShader` pro přidání shaderu do programu.
- `linkProgram` pro slinkování programu.
- `useProgram` pro použití daného programu.

Shadery mohou s okolním světem komunikovat třemi způsoby:

- Uniforms – vlastnosti, které se nemění během vykreslování snímku (například poloha světel).
  - Jsou dostupné vertex i fragment shaderům.
  - Jsou pouze pro čtení.
- Attributes – vlastnosti vrcholů.

    - Jsou dostupné pouze vertex shaderům.
    - Jsou pouze pro čtení.
- Varyings – slouží pro komunikaci mezi vertex a fragment shaderem.
    - Jsou dostupné vertex i fragment shaderům.
    - Vertex shadery mohou číst i zapisovat, fragment shadery mohou pouze číst.

Three.js opět přináší mírné vylepšení a zjednodušení celého procesu, a tím jsou „shader chunks“. Shader chunks jsou znovupoužitelné kusy kódu, ze kterých lze složit celý materiál. Například shader pro phongův stínovací model nebo mlhu jsou stále stejné a bylo by zbytečné je psát stále dokola. Pokud se ovšem rozhodneme nějaký materiál upravit, můžeme použít připravené chunky a změnit pouze tu část, která nás zajímá. Kromě toho už Three.js obsahuje celou řadu základních shaderů / materiálů, především pro různé osvětlovací modely.

K čemu tedy shadery použijeme? K deformaci terénu podle výškové mapy. Tato výšková mapa ale nebude mít formát JSON, leč bude to PNG obrázek s výškou zakódovanou do RGB kanálů, který se použije jako textura. Podle výsledku výpočtu 255G + R se pak vrchol posune ve směru své normály, v našem případě tedy svisle. Takto použité textuře se říká displacement mapa. Pokud ovšem změníme pozici vrcholů, dojde tím i ke změně normálových vektorů. Proto bývá společně s displacement mapou ještě použita takzvaná normálová mapa, což je opět textura.

Opět se hojně využije Three.js. V Three.js už je připravený materiál (shader) nazvaný `normal`, který již má připraveny uniformní proměnné na předání displacement textury, normálové textury a patřičných parametrů (především „síla“ efektu). Mimo to umožňuje i použití textur pro ovlivnění ambientní a difuzní složky nasvícení a také texturu pro efekty ambient occlusion a cube-map zrcadlení. Jako osvětlovací model pak materiál používá Blinn-Phongův. Aby vše fungovalo správně, je ještě navíc nutné u geometrie předpočítat tangenty pomocí zavolání její metody `computeTangents`.

Pro samotné stažení a načtení displacement mapy použijeme pomocnou metodu `THREE.ImageUtils.loadTexture`, která texturu stáhne, připraví a nakonec zavolá zadanou callback funci. Díky ní můžeme odložit provádění některých operací až na dobu, kdy jsou nezbytně nutné. Například samotné vytvoření terénu. Pro uživatele by bylo matoucí, kdyby nejdříve uviděl plochý terén, který by se mu až po několika sekundách by se mu zdeformoval podle výškové mapy.

Pojďme si nyní prakticky ukázat, jak budeme postupovat:

```
var shader = THREE.ShaderUtils.lib["normal"];
var uniforms = THREE.UniformsUtils.clone(shader.uniforms);
uniforms["tDisplacement"].texture = THREE.ImageUtils.loadTexture(
      url,
      new THREE.UVMapping(),
      function() {
            uniforms["uDisplacementBias"].value = -9999/4;
            uniforms["uDisplacementScale"].value = 0xff/4;
            terrain = new THREE.Mesh(
                  grid,
                  new THREE.ShaderMaterial({
                        fragmentShader: shader.fragmentShader,
                        vertexShader: shader.vertexShader,
                        uniforms: uniforms,
                        lights: true,
                        fog: true
                  })
            );
      }
);
```

Kód 12: Použití displacement textury

Na prvním řádku si vytvoříme shader. Druhý řádek může být trošku překvapivý: proč dělat kopii uniforms? Je to proto, že v JavaScriptu se přes = vytvořil pouze odkaz na `normal` shader. Pokud bychom tedy měli dlaždic více (což mít budeme), sdílely by všechny stejné uniforms, a tedy i stejnou displacement texturu. Na následujícím řádku provedeme načtení textury pomocí `loadTexture` metody ze zadané URL, texturu budeme chtít mapovat pomocí předpočítaných UV souřadnic a nakonec nastavíme callback. V něm nastavíme uniform `uDisplacementBias` pro nastavení vertikálního posunu displacement textury (jinými slovy „hladiny moře“), protože jinak se pro nenulové hodnoty displacement textury posouvají vrcholy pouze do kladného směru normál. Pokud ale chceme vytvořit prohlubně, musíme `uDisplacementBias` nastavit na zápornou hodnotu, abychom kompenzovali že „nadmořská výška“ dna prohlubně není nula. Další uniform, který nastavíme je `uDisplacementScale`, který určuje míru (sílu) posunutí. Obě zde uvedené hodnoty byly získány experimentálně. Na konci vytvoříme mesh terénu. K tomu použijeme připravenou mřížku (`grid`) a `ShaderMaterial`, kterému nastavíme patřičný vertex a fragment shader a také uniforms. Dále mu zapneme osvětlení a mlhu.

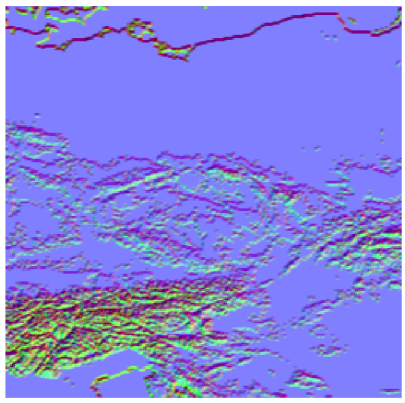
Tento postup má ještě jednu výhodu. Stačí mřížku vytvořit pouze jednou a pak ji lze opětovně používat a změnu velikosti jednotlivých dlaždic provádět na úrovni meshe.

6.2.2.1 Normály

Pokud bychom kód nechali tak ,jak je, zjistili bychom, že i když máme ve scéně světla, vidíme terén černý. Proč tomu tak je? Je to proto, že i když jsme terén zdeformovali, jeho normálové vektory

zůstávají stále stejné, jako by byl terén plochý. Musíme proto ještě vytvořit normálovou texturu, která určí směr normály.

Normálová textura je běžná textura v RGB formátu, kde jednotlivé barevné složky určují směr normály v jednotlivých osách. Normálové textury mají typicky nádech modré barvy (RGB 50 %, 50 %, 100 %), což značí rovinu (respektive nezměněnou původní normálu). Proč mají červená i zelená složka nenulové hodnoty, když je normála jednoduše svislá? Je to proto, že normálou musí být možné otáčet i do záporných hodnot a barevná složka záporná být nemůže. Proto je za nulovou hodnotu považováno 50 % (127). Na obrázku vpravo můžete vidět normálovou texturu vypočtenou z výškové mapy z ilustrace 6.



Ilustrace 7: Normálová textura

Jak tedy vypočítat normálu z výškové mapy? Nejprve získáme osm okolních bodů od aktuálního bodu postupně ve směru hodinových ručiček. Každý takový bod bude třísložkovým vektorem se souřadnicemi X a Y v rozsahu ±1 a Z právě nadmořská výška rekonstruovaná z červeného a zeleného barevného kanálu. Pokud se nacházíme na okraji, zopakujeme výšku okrajových bodů. Lze totiž předpokládat, že bod za hranicí výškové mapy bude mít velmi podobnou nadmořskou výšku jako bod na okraji. Dále body projdeme po dvojicích (vždy aktuální a následující), od každého z nich odečteme středový bod (čili prakticky jen jeho Z složku) a vypočteme jejich vektorový součin (cross product), čím získáme vektor na ně kolmý, který nakonec znormalizujeme. Tím dostaneme osm dalších vektorů. Tyto vektory po složkách sečteme a vydělíme osmi, čímž dostaneme aritmetický průměr a výsledný normálový vektor. Ten nyní pouze zakódujeme do RGB palety a máme vytvořenou normálovou texturu.

Protože `normal` materiál už je připravený na použití normálové textury, zbývá tuto texturu vytvořit. Three.js opět obsahuje implementaci výše uvedeného postupu (`THREE.ImageUtils.getNormalMap`), ten je ovšem opět vzhledem k použití dvou zanořených for cyklů velice pomalý a obzvláště při startu, kdy je potřeba načíst několik prvních dlaždic téměř zároveň, způsoboval dlouhé zamrzání aplikace a při používání pak další dlouhé zasekávání při výpočtu dalších normálových textur. Proto byly opět vyzkoušeny WebWorkers, tentokrát ovšem úspěšně.

Do callback funkce (protože normálovou texturu můžeme vytvářet, až když je načtena výšková mapa) metody `loadTexture` přidáme následující řádky:

```
uniforms["tDisplacement"].texture.image.width = 256;
uniforms["tDisplacement"].texture.image.height = 256;

var canvas = document.createElement("canvas");
canvas.width = uniforms["tDisplacement"].texture.image.width;
canvas.height = uniforms["tDisplacement"].texture.image.height;
var context = canvas.getContext("2d");
context.drawImage(uniforms["tDisplacement"].texture.image, 0, 0);

var worker = new Worker("NormalWorker.js");
worker.onmessage = function(e) {
      context.putImageData(e.data, 0, 0);
      uniforms["tNormal"].texture = new THREE.Texture(canvas);
      uniforms["tNormal"].texture.needsUpdate = true;
      //
      // Vytvoření meshe
      //
};

worker.postMessage({
      dem: context.getImageData(0, 0, canvas.width, canvas.height),
      normal: context.createImageData(canvas.width, canvas.height)
});
```

Kód 13: Příprava a spuštení WebWorkeru pro vytvoření normálové textury

Nejdříve nastavíme výškové textuře správnou velikost, a poté vytvoříme `<canvas>` element stejných rozměrů a získáme jeho 2D kontext. Díky tomu můžeme do canvasu vykreslit výškovou texturu. Dále známým způsobem připravíme samotný WebWorker. Po skončení WebWorkeru vykreslíme přijatá data do canvasu a nastavíme canvas jako normálovou texturu. Texturu nastavíme jako aktualizovanou (parametr `needsUpdate`) a můžeme vytvořit mesh. WebWorkeru nemůžeme poslat celý canvas element, protože WebWorkers neumí pracovat s DOMem. Můžeme mu ale poslat obrazová data, která získáme metodou `getImageData`. Abychom si je nepřepsali, vytvoříme ještě metodou `createImageData` prázdná obrazová data pro normálovou texturu. Tato obrazová data jsou ve skutečnosti obyčejným jednorozměrným polem obsahujícím za sebou RGBA hodnoty jednotlivých pixelů.

V samotném WebWorkeru pak pouze výše zmíněným způsobem vypočteme normály z výškových dat a výsledná obrazová data pošleme zpět.

Tímto řešením došlo ke znatelnému zrychlení aplikace, obzvlášť pokud aplikace běží na vícejádrových strojích, kdy výpočet pro každou dlaždici probíhá na vlastním jádře.

6.3 Chunked LoD

Základní informace o principu Chunked LoD byly již popsány v kapitole 4.1. Přesto si je pojďme ve stručnosti připomenout. Základní jednotkou Chunked LoD je čtyřstrom (quadtree), kde uzel představuje jednu dlaždici a jeho čtyři potomci jsou dlaždice se stejným rozlišením pokrývajícím jednotlivé kvadranty rodičovské dlaždice. Které dlaždice se budou zobrazovat se rozhodne podle vzorce (2) a proměnné τ.

Three.js již obsahuje implementaci základního LOD objektu. Ta je ale vhodná pouze pro objekty ve scéně (stromy, postavy, …), nikoliv pro objekty které scénu definují (terén). Objekt funguje tak, že se mu pomocí metody `addLevel` předá objekt a vzdálenost, při které se má objekt zobrazovat. Pomocí volání metody `update` při každém vykresleném snímku se poté zvolí nejvhodnější objekt k zobrazení. Princip použití metody update je jediné, co zůstane použito.

Byl tedy vytvořen nový objekt `ChunkedLOD`. Ten jako svůj prototyp používá `THREE.Object3D`, takže s ním lze provádět všechny běžné operace jako s každým jiným objektem. Jeho parametry jsou X a Y souřadnice výřezu, jeho posun relativní k rodičovské dlaždici, geometrie mřížky (používá se stále ta stejná) a úroveň ve stromu (0 je kořen). V jeho konstruktoru se provede vytvoření dlaždice dříve popsaným způsobem.. K tvorbě tedy dochází postupně. Nejdříve se stáhne výšková mapa, poté se z ní spočítá normálová textura a až na závěr se vytvoří samotný mesh. Tím docílíme toho, že uživatel uvidí terén, až když je připravený, a nedochází tak k rušivému „naskakování“ nehotových dlaždic. V konstruktoru také spočítáme geometrickou chybu δ jako šířku dlaždice děleno tisíci.

Povšimněme si toho, jak se do sebe noří callbacky jednotlivých metod. To je v JavaScriptu poměrně běžný jev a dá se velice dobře využít právě v případě, kdy je potřeba čekat na nějakou asynchronní akci (AJAX, stažení obrázku, …). Bohužel to ale často vede ke kódu označovanému jako „callback pyramid of doom“, protože přesně tak kód vypadá – nějaké jednoduché metody, callback, odsazení, další metody, callback, odsazení, …, spousta závorek a středníků při navracení zpět.

Mnohem zajímavější je metoda `update`. V ní se dynamicky vytváří a zobrazuje čtyřstrom dlaždic. Nejdříve si ověříme, zda vůbec existuje základní dlaždice. Kvůli čekání na výškovou mapu nebo normálovou texturu se totiž může stát že nikoliv. V takovém případě neděláme nic. Následně spočítáme pomocí vzorce (2) ρ. U výpočtu ρ se na chvíli zastavme. V dokumentu popisujícím fungování Chunked LoD algoritmu[20] se píše, že proměnná D by měla být „nejkratší vzdálenost obalového tělesa dlaždice k pozorovateli“. Pro zrychlení ale uvažujeme pouze vzdálenost ke středu dlaždice. K samotnému výpočtu vzdálenosti opět využijeme Three.js a jeho implementaci třírozměrného vektoru, který obsahuje metodu `distanceTo`, takže ji nemusíme počítat ručně rozšířením Pythagorovy věty. Také je vhodné zmínit, že nemůžeme použít vlastnost `position`, kterou používáme pro nasta-

vení polohy dlaždice! Vlastnost position je totiž určena relativně vzhledem k rodičovskému objektu. Musíme tedy použít `matrixWorld.getPosition()` čímž dostaneme pozici dlaždice ve scéně.

Také zjistíme, zda má současná dlaždice připravené všechny čtyři své poddlaždice. Opět se totiž může stát že nikoliv, a pak bychom určitě nechtěli zobrazovat jen několik z nich a místo chybějících mít díru nebo problikávající nadřazenou dlaždici.

Nyní máme všechny potřebné informace a můžeme se začít rozhodovat, co zobrazíme. Rozhodování u Chunked LoD je jednoduché:

```
If (ρ <= τ) {
      show_terrain;
} else {
      show_chunks;      //Rekurze
}
```

Kód 14: Rozhodování Chunked LoD

Ovšem tím, že se dlaždice vytváří dynamicky, se situace mírně komplikuje. Jednak musíme zobrazit terén, i když $\rho <= \tau$, a to v případě, kdy nemáme všechny čtyři poddlaždice připravené.

Stále jsme se také nedotkli vytváření samotného čtyřstromu. Čtyřstrom je rekurzivní struktura a objekt `ChunkedLOD` reprezentuje právě jeden uzel. Objekt má přidány dvě vlastnosti: `terrain`, reprezentující mesh terénu a `LODs`, reprezentující další (ideálně čtyři) poddlaždice (další `ChunkedLOD` objekty). Pokud tedy terén nebo dlaždice nejsou připraveny, je jejich hodnota buď `undefined` (nejsou připraveny vůbec) nebo v případě `LODs` je `LODs.children.length` menší než čtyři, pokud nejsou všechny poddlaždice připraveny.

Jak je tedy čtyřstrom budován? Naivním řešením by bylo při startu aplikace začít strom procházet do šířky a stahovat dlaždice. Tím by ale docházelo ke zbytečné zátěži jak serveru tak klienta, protože s největší pravděpodobností stejně nikdy neprozkoumá terén do takové hloubky, aby byly potřeba všechny dlaždice. Navíc by stahování méně detailních dlaždic třeba z opačného konce terénu brzdilo stahování těch detailních, které jsou právě potřeba. Lepším řešením je proto dlaždice stahovat podle potřeby, tedy podle polohy pozorovatele, a to teprve v momentě, kdy jsou potřeba. Tedy v `else` větvi kódu 14. V ní nejprve ověříme, zda existuje `LODs` objekt. Pokud ne, vytvoříme jej a vložíme do něj čtyři poddlaždice, čímž se zahájí jejich stahování. Pokud už LODs objekt existuje, a obsahuje právě čtyři připravené potomky, zavoláme rekurzivně jejich metody `update`, čímž se celý proces zopakuje. Tím tedy vzniká řez stromu a podle hodnoty τ jsou zobrazovány v současné době nejlepší dostupné dlaždice.

S rekurzivním skrýváním objektů si opět pomůžeme Three.js a to metodou `THREE.SceneUtils.showHierarchy`, které jako parametry předáme rodičovský objekt, u kterého chceme začít,

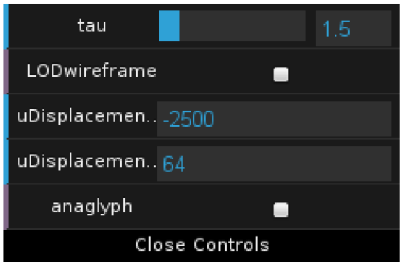
a hodnotu `true` nebo `false`, podle toho zda chceme všechny objekty v hierarchii skrýt nebo zobrazit.

6.4 Pomocné knihovny

Kromě velkých knihoven jQuery pro práci s DOMem a Three.js pro práci s WebGL byly použity ještě dvě další knihovny, které dále usnadnily vývoj a ladění aplikace. Pojďme si je nyní představit.

6.4.1 dat.GUI

Dat.GUI[29] je malá knihovna umožňující změnu hodnot zadaných vlastností objektů pomocí jednoduchého uživatelského rozhraní. Odpadám tím neustálé přepisování kódu a obnovování stránky při hledání té správné kombinace hodnot a parametrů. Knihovna dokáže sama rozpoznat datové typy a podle toho zobrazovat zaškrtávací políčka, textová pole, posuvníky (musí se zadat minimální a maximální hodnota), select boxy nebo výběr barev. Kromě možnosti měnit hodnoty lze také volat metody daných objektů. Položky je možné řadit do složek, případně mít připravené sady předdefinovaných hodnot.



Ilustrace 8: dat.GUI

Použití dat.GUI je velice jednoduché:

```
var gui = new dat.GUI();
gui.add(THREE, "tau", 0, 10).name("τ");
var displgui = gui.addFolder("Displacement");
var bias = displgui.add(THREE, "uDisplacementBias");
bias.name("Bias");
bias.onChange(function() {
      lod.displacement();
});
```

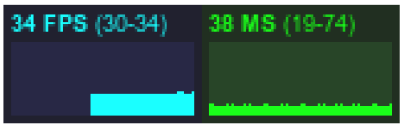
Kód 15: Použití dat.GUI

Nejdříve si vytvoříme instanci objektu. Tím dojde i k automatickému přidání uživatelského rozhraní do stránky. Poté voláním metody `add` přidáme ovládací prvky. Parametry metody jsou objekt, ve kterém se proměnná nachází, název proměnné (jako řetězec) a pokud se jedná o číslo, tak minimální a maximální hodnotu. Pokud mělo jít o výběr z několika řetězců, lze je zadat jako pole. Metodou `addFolder` přidáme složku, do které lze voláním její metody `add` vkládat vnořené ovládací prvky.

Občas nestačí pouze změnit hodnotu, ale je potřeba v případě její změny provádět nějaké další akce. K tomu slouží metody ovládacích prvků `onChange`, případně `onFinishChange`, pokud chceme reagovat až na výslednou hodnotu.

6.4.2 stats.js

Druhou malou (3 kB) pomocnou knihovnou je stats.js[30]. Tato knihovna má stejného autora jako Three.js a slouží k zobrazování grafů snímkové frekvence (nebo obecně jakýchkoliv opakujících se dějů). Kromě zobrazení počtu snímků za sekundu (včetně jeho minima a maxima) je také možné po kliknutí na graf zobrazit latence mezi snímky včetně grafu. Poslední možností je získávat data metodami `getFps` a `getMs`, případně jejich `Min` a `Max` variantami.



Ilustrace 9: stats.js

Použití stats.js je triviální. Nejdříve vytvoříme instanci objektu. Pak pomocí metody `getDomElement` získáme DOM element který nastylujeme podle potřeby (v příkladu bude vždy umístěn v pravém dolním rohu). Nakonec DOM element vložíme do DOM stromu (`container` je jQuery objekt).

```
var stats = new Stats();
stats.getDomElement().style.position = "absolute";
stats.getDomElement().style.bottom = 0;
stats.getDomElement().style.right = 0;
container.append(stats.getDomElement());
```

Kód 16: Vytvoření stats.js

Teď už jen do funkce vykreslující snímek přidáme `stats.update()`. Stats.js si hlídá časové prodlevy mezi voláním metody update a podle nich zobrazuje grafy.

7 Kompatibilita

Aplikace využívá ke svému běhu technologie ze standardu HTML5. Standard HTM5 ale zatím stále není dokončen (je ve stádiu working draft), a tudíž je jeho podpora mezi prohlížeči různá.

HTML5[31] má být normou, která má nahradit (a již pomalu nahrazuje) HTML 4 a XHTML. Specifikace HTML5 začala vznikat v roce 2004 (HTML 4.0 bylo publikováno v roce 1997) na výzvu Mozilla Foundation a Opera Software. HTML5 není je normou pro značkování, ale přináší i širší podporu pro multimédia bez nutnosti používat zásuvné moduly a snaží se prosadit HTML a Javascript jako nástroje pro vývoj plnohodnotných aplikací pro různé platformy od běžných počítačů přes televizory až po mobilní telefony.

Co se značkování týká, přináší nové značky jako `<header>`, `<footer>`, `<article>`, `<nav>` a další, které především dávají sémantický význam dříve používaným obecným značkám jako `<div>` a `<span>`, a tak napomáhají vyhledávačům při automatickém indexování obsahu. Došlo také ke zjednodušení definice typu dokumentu, která je nyní jednoduchá: `<!DOCTYPE html>`. Především bylo ale definováno chování parserů při nevalidním kódu (například křížení značek nebo jejich neuzavření), které si prohlížeče mohly v HTML 4 interpretovat jak chtěly. To ovšem může mít za následek úmyslné psaní nevalidního kódu, protože se nyní můžeme spolehnout na to, že se kód zpracuje vždy stejně. Tyto nové značky jsou již podporovány ve všech prohlížečích, včetně starších verzí. Jediným problémem byl Internet Explorer, který se do verze 9 choval tak, že elementy které neznal prostě ignoroval, nevkládal je do DOM stromu a neumožňoval je stylovat. Překvapivě ale umožňoval jejich vytvoření pomocí JavaScriptového příkazu `document.createElement("article")`, po kterém se elementy začaly chovat tak, jak by měly. Brzy tedy vnikla malá pomocná knihovna nazvaná HTML5 Shiv[32], kterou stačí vložit mezi skripty do hlavičky HTML dokumentu, a HTML5 značky lze použít i ve verzích Internet Exploreru 6 až 8.

Pro podporu multimédií slouží tagy `<audio>` a `<video>`. Pro použití tagu `<audio>` je nutné poskytnout zvuk ve formátu Ogg Vorbis. Problém je s tagem `<video>`, kde nepanuje shoda v tom, který kodek se má použít. Prohlížeče tak implementují některé ze tří navrhovaných kodeků: Ogg Theora, h.264 a WebM. Ogg Theora je otevřený kodek, ale nedosahuje příliš dobré kvality a není příliš rozšířená jeho hardwarová akcelerace. H.264 je oproti tomu velice kvalitní kodek s rozšířenou hardwarovou akcelerací na grafických kartách i v mobilních telefonech, ale jsou problémy s jeho patenty. WebM je snaha Googlu tyto problémy vyřešit, nabízí slušnou kvalitu obrazu, norma obsahuje referenční hardwarové implementace, ale k jeho rozšíření zatím nedochází, spíše naopak.

Samozřejmě nelze opomenout ani `<canvas>` a jeho WebGL kontext. Ten je podporován většinou moderních prohlížečů: Mozilla Firefox od verze 9, Google Chrome od verze 17 a Safari od verze 5.1. Opera bude v blízké budoucnosti WebGL podporovat také. V současné době lze stáhnout testovací verzi Opery s podporou WebGL. Za zmínku stojí, že Opera Mobile pro mobilní telefony již WebGL podporuje. Problémem je opět Internet Explorer. Microsoft nativní WebGL odmítá kvůli obavám o bezpečnost a stabilitu.[33] Je totiž fakt, že pomocí WebGL lze číst framebuffer grafické karty a získávat tak kusy obrazu na obrazovce. Další nevýhodou je, že WebGL může způsobit v případě chyby v ovladačích pád operačního systému. OpenGL příkazy posílané skrz 3D kontext se totiž nikterak dále nekontrolují a jdou tak „přímo" k ovladači a hardwaru. I přesto, že WebGL nelze v Internet Exploreru použít nativně, existuje několik zásuvných modulů které WebGL poskytují. Prvním z nich je IEWebGL[34] a druhým Chrome Frame[35]. První zmíněný obsahuje pouze WebGL. Chrome Frame ovšem nabízí celé jadro WebKit a JavaScriptový interpret V8[37], jako je to v prohlížeči Google Chrome. Zajímavostí je, že WebGL a <video> lze kombinovat a mít tak videoklipy jako textury.

Dalšími novinkami v HTML5 jsou i nová JavaScriptová API:

- Geolocation – zjištění polohy, například podle Wi-fi sítí
- Drag and Drop – například přetažení souboru ze správce souborů do prohlížeče a jeho následné nahrání
- WebStorage – náhrada cookies s širšími možnostmi
- WebSockets – přímá komunikace s protějškem na základě socketů
- …

Mezi tato nová API patří i WebWorkers[28] umožňující provádění výpočtů paralelně ve vláknech, viz kapitola 6.2.1.1. WebWorkers v současné době podporují všechny velké prohlížeče s výjimkou Internet Exploreru 9, ale ve verzi 10 již WebWorkers fungují. WebWorkers také nejsou podporovány na mobilních platformách, což se ale může změnit s nárůstem jejich výkonu a rozšíření vícejádrových telefonů.

Další HTML5 rozšíření které používáme je `requestAnimationFrame`[14] pro umožnění létání nad vykreslovanou krajinou. Toto API je opět dostupné v prohlížečích Mozilla Firefox a Google Chrome, ovšem pouze s prefixy `-moz-` respektive `-webkit-`. Internet Explorer má podporu připravenou v budoucí verzi 10, opět vyžadují vendor prefix `-ms-`. Opera také v současnosti `requestAnimationFrame` nepodporuje, a podpora v nové verzi 12 není známa. Také žádný z mobilních prohlížečů podporu `requestAnimationFrame` nedisponuje. Pomocí výše zmíněného polyfillu se však lze bez podpory v prohlížeči obejít.

Celkovou podporu vytvořené aplikace v různých prohlížečích shrnuje následující tabulka:[36]

Prohlížeč, funkce	Internet Explorer	Mozilla Firefox	Google Chrome	Opera	Mobilní
WebGL	Ne (pluginy)	Ano	Ano	Ne (12 ano)	Ne (pouze Opera Mobile)
WebWorkers	Ne (10 ano)	Ano	Ano	Ano	Ne
requestAnimationFrame	Ne (10 ano)	Ano	Ano	Ne	Ne
Celkem	Nyní ne, v budoucnu ano	Ano	Ano	Nyní ne, v budoucnu ano	Ne

Tabulka kompatibility webových prohlížečů s použitými HTML5 API

Vidíme tedy, že v současné době jsou jedinými plně podporovanými prohlížeči Mozilla Firefox a Google Chrome. V blízké budoucnosti se k nim připojí i Opera a Internet Explorer, u kterého bude nejspíš stále potřeba zásuvný modul pro běh WebGL.

8 Výkon

Na počátku panovaly obavy, zda vůbec bude možné Level of Detail terénu ve WebGL implementovat. Jednak kvůli omezení WebGL (OpenGL ES) a dále také rychlosti JavaScriptu. Ukázalo se, že oba problémy sice nastaly, ale podařilo se je vyřešit.

Práci s WebGL značně zjednodušilo použití Three.js frameworku. Pokud tedy nějaké problémy jsou, Three.js je vyřešilo a odstínilo. Jediný problém, který nastal byla neschopnost OpenGL ES použít jednokanálové šestnáctibitové textury. Tento problém vyřešilo použití běžných RGB textur s osmi bity na kanál a kódování hodnot do jednotlivých kanálů. Více se tomuto problému věnuje kapitola 6.2. Co se výkonu týče, tak by z principu fungování nemělo být WebGL o mnoho pomalejší než běžné OpenGL aplikace. Je to dáno tím, že mezi API WebGL a ovladačem grafické karty není žádná bariéra – všechna volání se tak přímo posílají ovladačům grafické karty, které je zpracovávají.

JavaScript je jazyk, o němž je známo, že výkon jeho interpretů není nijak závratný. Jeho největší využití je ve webových stránkách pro práci s DOM. V době jeho vzniku, kdy webové stránky byly převážně statické, nebyl příliš velký výkon ani potřeba. Stačilo, aby došlo například ke změně barvy nějakého prvku ve stránce tak, aby si uživatel nevšiml zpoždění. Postupem času se ale webové stránky staly stále více interaktivnějšími a obsahovaly stále více skriptů. Výsledkem je současný stav, kdy je trendem vytvářet běžně známé aplikace, jako emailové klienty nebo textové editory, přímo ve webových prohlížečích. Zda je tento trend správný ponechme stranou. Je zřejmé, že pro takovéto aplikace je kritické, aby JavaScript dokázal běžet dostatečně rychle. Průkopníkem se zde stal Google s jeho prohlížečem Chrome a především pak jeho JavaScriptové jádro V8[37]. To v době svého uvedení přineslo revoluci. Bylo daleko rychlejší než jádra všech ostatních prohlížečů a dodnes ho jádra ostatních prohlížečů dohánějí. Za zmínku také stojí na V8 založená platforma pro škálovatelné internetové aplikace node.js[38].

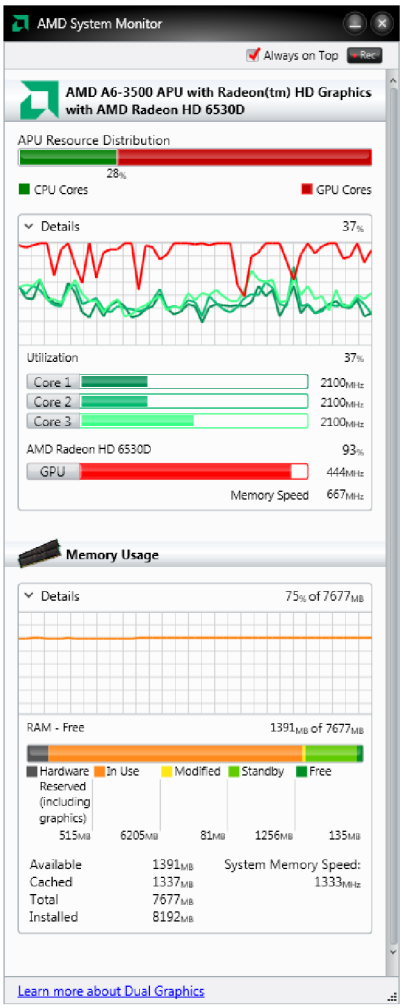
Vzhledem k výše uvedeným parametrům by se dalo očekávat, že hlavním problémem bude rychlost (nebo spíše pomalost) JavaScriptu. To se ukázalo jako částečně pravdivé. Pokud byla geometrie terénu počítána JavaScriptem, byla rychlost neúnosně malá. Když však byl tento úkol delegován na shadery grafické karty, bylo po problému. Respektive hlavním parametrem určující výkon se stala grafická karta, což potvrzují testy výkonu v následující kapitole. Překvapivě nebyl vůbec problém s procházením čtyřstromu a přepínáním viditelnosti dlaždic.

8.1 Testy výkonu

Pro otestování výkonu byla připravena demoverze aplikace, která obsahovala pouze čtyři první úrovně stromu dlaždic. Vše ostatní zůstalo nezměněno, dokonce byly některé nekritické části (GUI

pro nastavení parametrů, princip ovládání, …) dále aktualizovány. Spolu s tím byl vytvořen formulář pomocí služby Google Dokumenty (nyní Drive), kde byl popsán princip ovládání, a testující měli za úkol aplikaci chvíli testovat (prolétávat nad krajinou, měnit parametry, atd.) a poté do formuláře vyplnit především průměrnou snímkovou frekvenci, operační systém a prohlížeč. Volitelně také procesor, grafickou kartu, komentář a kontakt. Podařilo se získat přes 50 měření ze spousty kombinací operačních systémů, verzí prohlížečů a hardwaru.

Z průzkumu jasně vyplynulo několik věcí. Jednak se potvrdily předpoklady ohledně kompatibility internetových prohlížečů. V prohlížečích Google Chrome, Mozilla Firefox a odvozených aplikace fungovala (kromě několika případů velmi zastaralé verze Firefoxu). V Ostatních prohlížečích jako Opera a Internet Explorer nikoliv, překvapivě v několika případech ani v Opeře 12. Pravděpodobně ovšem šlo o nedostatečnou grafickou kartu. Vzhledem k tomu, že testery byly převážně studenti informačních technologií a jiných technických oborů, nebylo velkým překvapením, že téměř nikdo nepoužíval Internet Explorer. Trošku překvapivé ale bylo, že také skoro nikdo nepoužívá beta a testovací verze prohlížečů.



Ilustrace 10: Rozdělení zátěže na CPU AMD A6-3500 s integrovanou grafickou kartou Radeon HD 6530D

Z testů dále vyplynulo, že výkon aplikace je velmi závislý na výkonu grafické karty. Sestavy (především laptopy) se slabšími grafickými kartami, jako postarší integrované grafické karty Intel, v lepším případě mobilní verze akcelerátorů nVidie a Ati/AMD, nefungovaly buď vůbec nebo se snímkou frekvencí v řádu jednotek snímků za sekundu. Běžné kancelářské sestavy se pohybovaly na hodnotách okolo 20 a 30 snímků za sekundu. Vyšší střední třída již neměla problém udržet se na 60 snímcích za sekundu. Jeden z testerů poskytl i snímek panelu (obrázek napravo), který sleduje rozdělení zátěže, na kterém je jasně patrné, že grafická karta (červené části grafů) vyžaduje 72 % výkonu a procesor (zelené části grafů) jen 28 %.

Rozptyl testovaných sestav byl bohužel příliš veliký na to, aby šlo učinit jasné závěry co se srovnání výkonu mezi prohlížeči týče. Z toho důvodu jsem tento test provedl na svém vlastním počítači následující konfigurace:

- CPU Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz
- GPU NVIDIA G94 [GeForce 9600 GT] (1 GB VRAM, ovladače 295.49)

- 4 GB RAM
- OS Ubuntu 12.04 64bit (Unity 2D)
- Google Chrome 20.0.1132.8 (Oficiální sestavení 137054) dev
- Mozilla Firefox 12.0

Nejprve byly provedeny testy s prohlížečem Google Chrome, což byl prohlížeč, na kterém probíhal vývoj (již od verze 18-dev). Kamera byla při maximalizovaném okně (vykreslovací plocha přibližně 1 920 × 1 000 pixelů) vždy umístěna přibližně na stejné místo a po načtení dlaždic byl odečten počet snímků za sekundu. To samé bylo provedeno i při zmenšeném okně (přibližně 800 × 600 pixelů), aby byl vidět vliv rozlišení na výkon. Testy dopadly následovně:

	Maximalizované okno			**Malé okno**		
	$\tau = 3$	**$\tau = 2$**	**$\tau = 1$**	**$\tau = 1$**	**$T = 0{,}8$**	**$T = 0{,}2$**
Google Chrome 20-dev	60 FPS	30 FPS	20 FPS	60 FPS	60 FPS	60 FPS
Mozilla Firefox 12	25 FPS	10 FPS	5 FPS	65 FPS	36 FPS	16 FPS

Tabulka naměřených snímkových frekvencí

Chování prohlížeče Google Chrome bylo poněkud zvláštní – při maximalizovaném zobrazení a zobrazení jiné než nejhrubější dlaždice docházelo k „přepínání“ snímkové frekvence mezi 30 a 60 FPS (při $\tau < 3$). K tomu však během vývoje nedocházelo, a pravděpodobně se tak jedná o nějakou chybu v prohlížeči. Přeci jen se jedná o vývojovou verzi.

Z výsledků je patrný značný náskok prohlížeče Google Chrome nad Mozillou Firefox. Výsledky se však víceméně shodují s výsledkem veřejného testu, kde se většina výsledků Mozilly Firefox pohybuje okolo 40 snímků za vteřinu, zatímco Google Chrome nemá problém vykreslit 60.

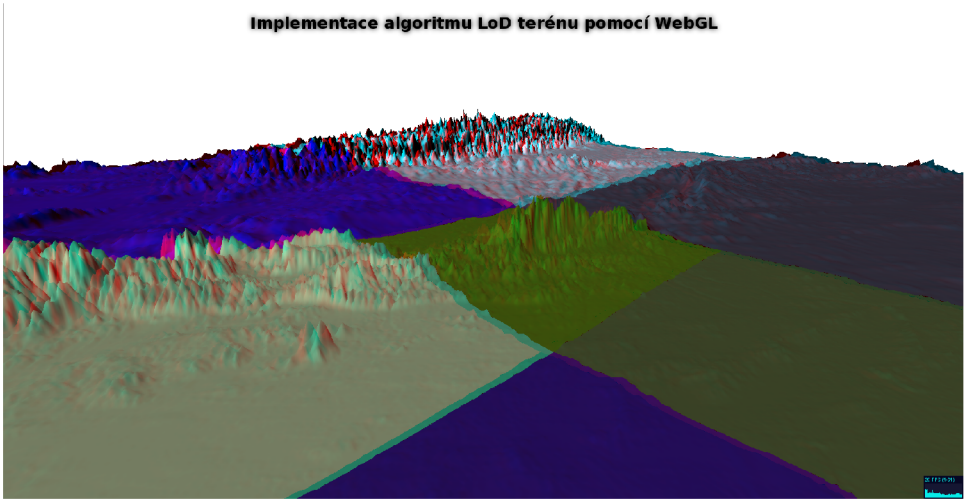
Spotřeba paměti byla v prohlížečích Google Chrome i Mozilla Firefox shodná, a pohybovala se okolo 300 MB. To není zrovna málo. Je ale třeba vzít v potaz, že i moderní webové aplikace jako například Gmail, Twitter nebo Facebook také spotřebují přibližně 150 MB, čili polovinu. Na „vině“ tedy bude pravděpodobně interpret JavaScriptu. JavaScript neumožňuje ruční správu paměti (nic jako `malloc` a `free`) a tak nezbývá, než se spoléhat na garbage collector interpretu.

Ukázali jsme tedy, že výsledná aplikace je použitelná, pokud neběží na úplně pomalých počítačích (notebooky) a pokud máme k dispozici alespoň průměrně výkonnou grafickou kartu. Také se ukázalo, že kromě grafické karty na výkon velký vliv také zvolený prohlížeč a jeho interpret JavaScriptu, což je ve shodě s očekáváním.

9 Rozšíření

Pokud máme implementované základní zobrazování terénu, můžeme začít uvažovat nad možnými rozšířeními a vylepšeními. V této kapitole se tedy budeme zabývat těmito možnými vylepšeními. Jedno z nich bylo implementováno, u zbylých si rozebereme alespoň návrh jejich implementace.

9.1 Anaglyf



Ilustrace 11: Anaglyfické 3D zobrazení

Anaglyfické zobrazení[39] je způsob stereoskopického zobrazení, umožňuje tedy lépe vnímat prostorovou hloubku obrazu. Základní princip je takový, že se různými barvami vykreslí scéna pro levé a pravé oko a výsledný obraz je pozorován přes barevné filtry (barevné brýle), které opět oddělí barevné složky pro každé oko. Příklad můžete vidět na obrázku výše.

Nejčastěji se dnes můžeme setkat s červenou barvou pro levé oko a azurovou pro pravé. Cílem je, aby tyto barvy byly co nejvíce inverzní. Pokud tedy má levé oko červenou a používáme RGB paletu, je inverzní barvou kombinace zelené a modré, tedy azurová. V tištěných médiích se pak můžeme setkat ještě s kombinací červená a zelená.

Pokud tedy chceme vykreslit anaglyfickou scénu, musíme ji nejprve vykreslit do textur dvěma kamerami, pro každé oko zvlášť. Kamery by navíc neměly být paralelně, ale měly by směřovat do stejného bodu, tam, kam chceme zaostřit. Někoho by mohlo napadnout i využít Z-buffer a ušetřit si tím nutnost scénu vykreslovat dvakrát. To ale není dobrý nápad, protože právě díky tomu, že scénu vykreslujeme dvakrát můžeme vidět „za objekty“. Nyní pomocí fragment shaderu sloučíme obě textury do jedné a upravíme barvy. Pro úpravu barev existuje několik metod[40]. V aplikaci je použita ta, která je v článku označena jako „Optimized Anaglyphs“. Výsledný obraz tedy vznikne tak, že čer-

vený kanál bude smíchaný ze zelené a modré složky levého snímku v poměru sedm ku třem a zelený a modrý kanál pak zůstane ponechán z pravého snímku. Alfa kanál je pak součtem alfa kanálů z obou snímků.

Opět nám usnadní práci Three.js, protože výše zmíněný postup je již implementován v jeho příkladech jako `THREE.AnaglyphEffect`. Jako parametr se předá existující renderer a s efektem se pak dále pracuje jako s běžným rendererem. Jedinými drobnými problémemy jsou viditelné hrany terénu v mlze a nefungující antialiasing.

9.2 Texturování terénu

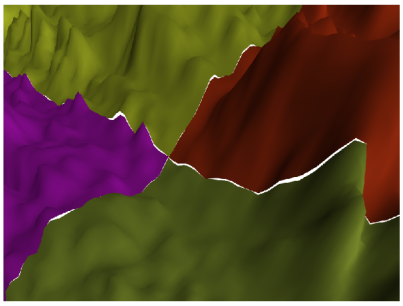
Momentálně je terén vykreslován plnou barvou (a aby byly dlaždice rozpoznatelné, je každá vykreslována jinou barvou). Určitě by bylo pěkné, kdyby byl terén otexturovaný například fotomapami nebo vektorovými mapami z projektu OpenStreetMap[41].

Problémem zde není samotné otexturování, k tomu stačí pouze texturu přiřadit do uniformní proměnné `tDiffuse`, stejně jako je tomu u výškové a normálové textury. Daleko větší problém je se samotnými mapami. Prvním z nich je, že by se textury musely georeferencovat proti výškové mapě a nejspíš i upravovat projekce a tak dále. Navíc dlaždice výškové mapy by se pravděpodobně neshodovaly s dlaždicemi textury a musely by se tak různě přepočítávat, spojovat a znovu rozřezávat. Problém to rozhodně není neřešitelný, ale řešení by nebylo triviální.

U rasterizovaných vektorových map z OpenStreetMap (a pravděpodobně i s fotomapami, pokud by obsahovaly popisky) by navíc nastal problém s měřítkem. Nestačilo by mít pouze jednu velkou základní dlaždici, byť vytvořenou spojením nejdetailnějších dostupných dlaždic. Pokud totiž mapa obsahuje popisky, je potřeba i na ně uplatnit úroveň detailu. Je totiž zbytečné při pohledu na celý stát vidět názvy ulic ve městech, nebo v opačném případě při přiblížení vidět města jen jako jejich obrys a jméno. Bylo by tedy nutné mít předpřipravené textury pro několik přiblížení a z nich pak vybírat tu nejvhodnější. Tím by ale vznikl další problém, a tím je návaznost dlaždic – vzdálené (velké) dlaždice by měly méně detailní textury zatímco ty blízké (malé) by měly textury s detaily a byl by viditelný jejich přechod.

Náhradní variantou by mohlo být místo textury na terén mapovat barevný přechod podle jeho nadmořské výšky. Samozřejmě by se musela kompenzovat nadmořská výška moře (−9 999 metrů).

9.3 Návaznost dlaždic



Ilustrace 12: Díry mezi nenavazujícími dlaždicemi

Kvůli chybám v interpolaci a při přechodech mezi úrovněmi detailu na sebe dlaždice nenavazují a vznikají mezi nimi úzké „díry“, které viditelně narušují jinak hladký terén (viz obrázek). Tento problém doporučuje paper popisující algoritmus Chunked LOD[20] řešit pomocí takzvaných sukýnek (skirts). Sukýnky byly už popsány v kapitole 4.1, ale pojďme si je připomenout. Jedná se o vertikální pás vedoucí po obvodu dlaždice kolmo dolů, a to vždy na stejnou výšku. Tento pruh má pak stejnou barvu (tedy i normálu) jako hrana ze které vychází. Tím dojde k vyplnění děr a dlaždice vypadá jako by normálně pokračovala a navazovala na dlaždici sousední.

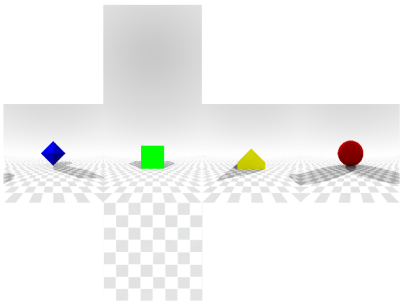
Při současné implementaci by bylo nutné změnit několik věcí. V první řadě by bylo nutné místo jednoduché mřížky generovat geometrii obsahující sukýnku. Při generování by navíc bylo nezbytné aby sukýnka měla stejné UV souřadnice pro textury jako hrana nad ní. Tím se vyřeší problém s barvou a normálou. Pokud by nevadilo, že spodek sukýnky nebude zarovnaný na stejnou výšku, tak bychom zde mohli skončit. Displacement shader by totiž posunul vrcholy ve směru normály, ten by byl shodný s normálou mřížky (tedy svislý) a vrcholy sukýnky by se tak posunuly po svislé ose stejně jako mřížka. Pokud bychom ale vyžadovali, aby sukýnka zůstala zarovnaná, museli bychom nejspíše přes atributy vrcholů označit vrcholy sukýnky a upravit shader tak, aby se nad takto označenými vrcholy neprováděl.

9.4 Postprodukční efekty

Posledními navrhovanými změnami jsou různé postprodukční efekty, které nemají žádný výrazný vliv na funkcionalitu, pouze vylepšují vizuální kvalitu. Tyto efekty spadají do kategorie takzvaných postprodukčních efekt, protože není potřeba pro jejich aplikaci výrazně modifikovat scénu a stačí scénu pouze vykreslovat ve více průchodech, které se následně spojí do výsledného obrazu.

Tyto efekty se dnes běžně používají v moderních počítačových hrách kde by se bez nich po grafických orgiích bažící hráči jen těžko obešli.

9.4.1 Cube mapa oblohy

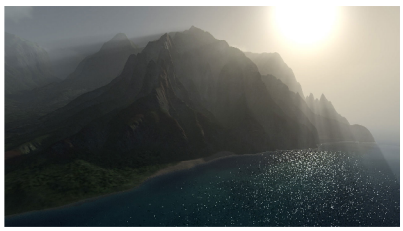


Ilustrace 13: Cube mapa, zdroj: Wikimedia Commons, Panorama cube map.png

Prvním z takových efektů by mohla být například cube-mapa oblohy. Tento efekt slouží k tomu, aby měl pozorovatel ponětí o tom, kterým směrem se dívá pokud zrovna nevidí terén. I když tento efekt není čistě postprodukční, přesto jej ponechejme v této kapitole. Pro tento efekt je nejdříve nutné získat šest obrázků pro jednotlivé stěny krychle. Příklad můžete vidět na obrázku vpravo. Těmito šesti obrázky následně otexturujeme krychli. Tato krychle může mít libovolné rozměry, musí mít normály orientované do svého středu a její pozice kopíruje pozici kamery. Při vykreslování scény se pak nejdříve vykreslí tato krychle bez zápisu do Z-bufferu. Poté se opět povolí zápis do Z-bufferu a vykreslí se zbytek scény. I když byla snaha mít tento efekt ve finální verzi aplikace, výsledek nevypadal dobře – i když krychle kopírovala pozici kamery, mapování textury se chovalo zvláštně, a když byl vypnutý zápis do Z-bufferu, terén krychlí podivně problikával. Pokus byl v kódu ponechán pouze zakomentován.

Krychlové mapy lze také použít pro napodobení zrcadlení. Stačí ze scény odstranit zrcadlící objekt, vyrenderovat z jeho pozice scénu jako textury krychle a poté takto vzniklé textury vhodně mapovat na původní objekt vzhledem k pozici kamery.

9.4.2 Sluneční paprsky

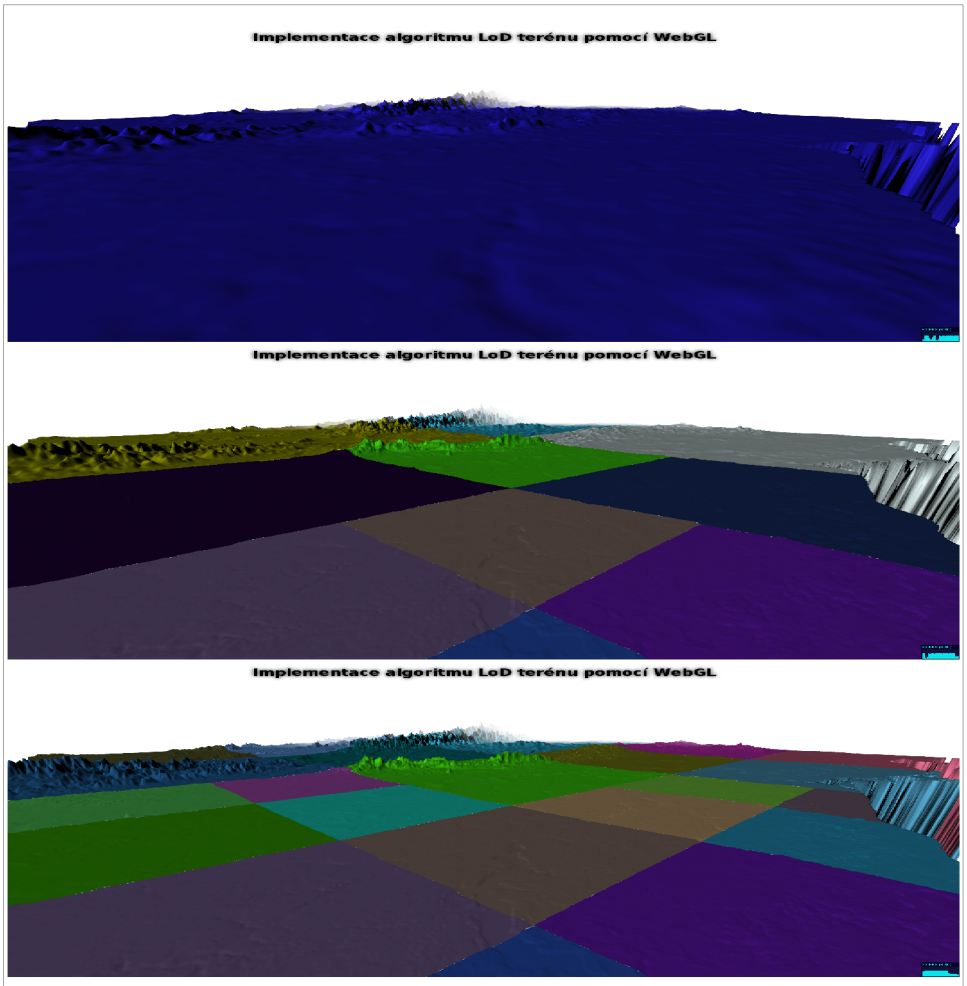


Ilustrace 14: Godrays ze hry Far Cry 2, CryEngine 2, Crytek, 2008

Další z pěkných efektů by mohly být takzvané godrays, neboli viditelné paprsky slunce prosvítající za horami. Tento efekt se dělá tak, že se vyrenderuje scéna jen jako maska (bílá obloha, černý terén nebo jiné objekty ve scéně). Na masku se aplikuje radial blur filtr ze směru kde je slunce a nakonec se rozmazané maska přičte k normálně vykreslené scéně.

Určitě by s trochou fantazie šlo vymyslet i řadu dalších podobných postprodukčních efektů, třeba hloubka ostrosti s využitím Z-bufferu, vodní hladina a další.

10 Závěr



Ilustrace 15: Terén vykreslovaný algoritmem Chunked LoD s různými hodnotami parametru τ

Po nastudování různých algoritmů pro zobrazování terénů v reálném čase a zvolení algoritmu Chunked LoD[20] kvůli jeho vhodnosti pro nasazení v online prostředí byl tento algoritmus implementován ve WebGL za použití knihovny Three.js[12]. Kromě WebGL byly využity i další technologie z HTML, například WebWorkers pro výpočet normálových map ve vláknech.

Během implementace nastalo několik problémů spojených především s výkonem JavaScriptu v prohlížečích. Tyto problémy však byly vyřešeny použitím shaderů grafické karty.

Kromě samotné webové aplikace pro zobrazování terénu v prohlížeči byl v jazyce Python vytvořen i server pro generování dlaždic z výškových dat ve formátu GeoTIFF[2].

Po dokončení bylo vytvořeno omezené demo (pouze čtyři úrovně dlaždic) společně s krátkým formulářem za účelem veřejného testu výkonu a kompatibility. Kompatibilita dopadla podle očekávání – aplikace fungovala bezvadně v prohlížečích Mozilla Firefox a Google Chrome. Testy dále odhalily, že výkon aplikace je více závislý na výkonu grafické karty než na procesoru.

Během vývoje také proběhly pokusy o implementaci některých rozšíření, které většinou selhaly. Výjimkou je anaglyfické stereoskopické zobrazení které v aplikaci zůstalo a lze jej zapnout.

Celkově testeři aplikaci chválili a aplikace se tedy jeví jako použitelná například pro rychlou vizualizaci dat z různých geodetických měření.

Literatura

[1] Wikipedia contributors. Digital elevation model. *Wikipedia, The Free Encyclopedia*. 462945198. 2011. [cit. 2011-12-26]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Digital_elevation_model&oldid=462945198

[2] *GeoTIFF Format Specification*. 1.8.2. GeoTIFF Working Group, 2000. Dostupné z: http://www.remotesensing.org/geotiff/spec/geotiffhome.html

[3] Wikipedia contributors. Delaunay triangulation. *Wikipedia, The Free Encyclopedia*. 463902359. 2011. [cit. 2011-12-26]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Delaunay_triangulation&oldid=463902359

[4] Wikipedia contributors. Triangulated irregular network. *Wikipedia, The Free Encyclopedia*. 462104859. 2011. [cit. 2011-12-26]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Triangulated_irregular_network&oldid=462104859

[5] *Shuttle Radar Topography Mission* [online]. NASA, 2000, 2009 [cit. 2011-12-26]. Dostupné z: http://www2.jpl.nasa.gov/srtm

[6] *CGIAR-CSI SRTM 90m DEM Digital Elevation Database* [online]. 2008 [cit. 2011-12-26]. Dostupné z: http://srtm.csi.cgiar.org

[7] *ASTER: Advanced Spaceborne Thermal Emission and Reflection Radiometer* [online]. NASA, 2011 [cit. 2011-12-26]. Dostupné z: http://asterweb.jpl.nasa.gov

[8] Wikipedia contributors. WebGL. *Wikipedia, The Free Encyclopedia*. 462104859. 2011. [cit. 2011-12-26]. Dostupné z: http://en.wikipedia.org/w/index.php?title=WebGL&oldid=465418833

[9] *WebGL – OpenGL ES 2.0 for the Web* [online]. Kronos Group, 2011 [cit. 2011-12-26]. Dostupné z: http://www.khronos.org/webgl

[10] FORSHAW, James, Paul STONE a Michael JORDON. WebGL: More WebGL Security Flaws. *Context Information Security*. 2011 [cit. 2011-12-26]. Dostupné z: http://www.contextis.com/research/blog/webgl2

[11] GILES, Thomas. WebGL Lesson 1: A triangle and a square. *Learning WebGL: lessons 'n' links*. 2009 [cit. 2011-12-26]. Dostupné z: http://learningwebgl.com/blog/?p=28

[12] *Three.js: javascript 3D engine* [online]. Mr.doob, 2011 [cit. 2011-12-26]. Dostupné z: https://github.com/mrdoob/three.js

[13] *Mouse Lock*. SCHEIB Vincent a Google, 2011. Dostupné z: http://dvcs.w3.org/hg/webevents/raw-file/default/mouse-lock.html

[14] *Timing control for script-based animations*. ROBINSON, James a MCCORMACK, Cameron, 2011. Dostupné z: http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/RequestAnimationFrame/Overview.html

[15] *SceneJS – WebGL Scene Graph Library* [online]. Xeolabs, 2011 [cit. 2011-12-26]. Dostupné z: http://scenejs.org

[16] *BioDigital Human Platfrom* [online]. BioDigital, 2011 [cit. 2011-12-26]. Dostupné z: http://www.biodigital.com/biodigital-human.html

[17] *GLGE: WebGL for the lazy* [online]. BRUNT, Paul, 2011 [cit. 2011-12-26]. Dostupné z: http://www.glge.org

[18] Wikipedia contributors. Level of detail. *Wikipedia, The Free Encyclopedia*. 459736077. 2011. [cit. 2011-12-26]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Level_of_detail&oldid=459736077

[19] *Terrain LOD Published Papers* [online]. [cit. 2011-12-26]. Dostupné z: http://vterrain.org/LOD/Papers

[20] THATCHER, Ulrich. Renderring Massive Terrains using Chunked Chunked Level of Detail Control. [online]. 2000, 2002 [cit. 2011-12-26]. Dostupné z: http://tulrich.com/geekstuff/sig-notes.pdf

[21] LOSASSO, Frank a HOPPE Huguess. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. [online]. 2004 [cit. 2011-12-26]. Dostupné z: http://research.microsoft.com/en-us/um/people/hoppe/geomclipmap.pdf

[22] ASIRVATHAM Arul a HOPPE Huguess. GPU Gems 2: Chapter 2. Terrain Rendering Using GPU-Based Geometry Clipmaps. [online]. 2009 [cit. 2011-12-26]. Dostupné z: http://developer.nvidia.com/node/19

[23] STRUGAR Filip. Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD). [online]. 2010 [cit. 2011-12-26]. Dostupné z: http://www.vertexasylum.com/downloads/cdlod/cdlod_latest.pdf

[24] *GDAL – Geospatial Data Abstraction Library* [online]. [cit. 2011-12-26]. Dostupné z: http://www.gdal.org

[25] *BaseHTTPServer — Basic HTTP server* [online]. [cit. 2012-04-22]. Dostupné z: http://docs.python.org/library/basehttpserver.html

[26] ENTHOUGHT, Inc. *SciPy* [online]. 2012-04-18 [cit. 2012-04-22]. Dostupné z: http://www.scipy.org

[27] *Python Imaging Library* [online]. PythonWare, 2011 [cit. 2011-05-01]. Dostupné z: http://www.pythonware.com/products/pil

[28] *Using web workers – MDN* [online]. [cit. 2012-04-27]. Dostupné z: https://developer.mozilla.org/en/Using_web_workers

[29] *dat.GUI – A lightweight controller library for JavaScript* [online]. [cit. 2012-05-01]. Dostupné z: http://code.google.com/p/dat-gui/

[30] *stats.js* [online]. [cit. 2012-05-01]. Dostupné z: https://github.com/mrdoob/stats.js

[31] HTML5. *A vocabulary and associated APIs for HTML and XHTML*. Working Draft. W3C, 2012. Dostupné z: http://www.w3.org/TR/html5

[32] SHARP, Remy. HTLM5 enabling script. [online]. 2009 [cit. 2012-05-03]. Dostupné z: http://remysharp.com/2009/01/07/html5-enabling-script

[33] WebGL Considered Harmful. In: *Security Research & Defense: Information from Microsoft about vulnerabilities, mitigations and workarounds, active attacks, security research, tools and guidance* [online]. Microsoft, 2011 [cit. 2012-05-03]. Dostupné z: https://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx

[34] *IEWebGL: WebGL for Internet Explorer* [online]. 2011, 2012 [cit. 2012-05-03]. Dostupné z: http://iewebgl.com

[35] GOOGLE. *Google Chrome Frame* [online]. 2012 [cit. 2012-05-03]. Dostupné z: http://www.google.com/chromeframe

[36] *When can I use…: Compatibility tables for support of HTML5, CSS3, SVG and more in desktop and mobile browsers*. [online]. 2012 [cit. 2012-05-04]. Dostupné z: http://caniuse.com

[37] GOOGLE. *V8 JavaScript Engine* [online]. 2012 [cit. 2012-05-13]. Dostupné z: http://code.google.com/p/v8

[38] JOYENT INC. *node.js* [online]. 2012 [cit. 2012-05-13]. Dostupné z: http://node.js

[39] Wikipedia contributors. Anaglyph 3D. *Wikipedia, The Free Encyclopedia*. 492408061. 2012. [cit. 2012-05-17]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Anaglyph_3D&oldid=492408061

[40] WIMMER, Peter. Anaglyph Methods Comparison. [online]. [cit. 2012-05-17]. Dostupné z: http://3dtv.at/Knowhow/AnaglyphComparison_en.aspx

[41] *OpenStreetMap*. [online]. 2012 [cit. 2012-05-18]. Dostupné z: http://www.openstreetmap.org

Seznam příloh

1. CD