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

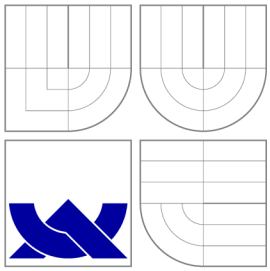
DISTRIBUOVANÝ RENDERING NA PLATFORMĚ WEBGL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

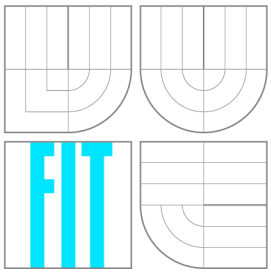
AUTOR PRÁCE
AUTHOR

LUKÁŠ SVAČINA

BRNO 2013



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

DISTRIBUOVANÝ RENDERING NA PLATFORMĚ WEBGL

DISTRIBUTED RENDERING ON WEBGL PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ SVAČINA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ŠOLONY MAREK,

BRNO 2013

Abstrakt

Bakalářská práce se zabývá studiem a aplikací možností moderních webových prohlížečů z pohledu 3D grafiky a distribuce dat. Jsou využity mnohé technologie nového standardu `HTML5`, jako je například technologie `WebGL` pro nízkoúrovňové vykreslování 3D grafiky s využitím přímého programování shaderů grafické karty nebo technologie `XHR2` pro asynchronní přenos binárních dat. Cílem práce je návrh a implementace jednoduchého distribuovaného vykreslování 3D grafiky za využití těchto technologií.

Abstract

Bachelor thesis deals with the research and the usage possibilities of modern web browsers from the view of 3D graphics and data distribution. In the research were used the new HTML5 standard technologies, such as the WebGL technology for low level graphics together with direct programming of the card shaders or XHR2 technology for asynchronous binary data transfers. The objective of the thesis is to implement a simple distributed 3D graphics renderer by application of these technologies.

Klíčová slova

HTML5, WebGL, OpenGL ES, shader, 3D grafika, distribuce výpočtů

Keywords

HTML5, WebGL, OpenGL ES, shader, 3D graphics, distributed computing

Citace

Lukáš Svačina: Distribuovaný rendering
na platformě WebGL, bakalářská práce, Brno, FIT VUT v Brně, 2013

Distribuovaný rendering na platformě WebGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Lukáše Poloka. Uvedl jsem veškeré literární prameny a publikace, ze kterých jsem čerpal informace.

.....
Lukáš Svačina
15. května 2013

Poděkování

Rád bych poděkoval panu Ing. Lukáši Polokovi za nespočet rad při tvorbě této práce a za dozor, který mi poskytl. Také bych rád poděkoval panu Bc. Petru Izraelovi za správné nasměrování při vývoji této práce.

Obsah

Kapitola 1

Úvod

Ještě donedávna byly aplikace a hry akcelerované grafickou kartou výsadou nativních aplikací operačních systémů. V současnosti nelze ignorovat migraci velké části nativních aplikací do prostředí webu, kde je jediným prostředníkem mezi uživatelem a aplikací moderní webový prohlížeč. Především s příchodem nového standardu `HTML5` [2.1] je prohlížečům coby prostředkům prezentujících jednoduchou dvourozměrnou grafiku a disponujících základními možnostmi interakce konec.

Tento standard, který je již široce podporován moderními prohlížeči, přináší mnoho nových možností pro vývojáře takových, aby bylo možné nejen obohatit současné webové prezentace o další dávku interakce, ale troufám si říct, posouvá možnosti webu jako takového do nové éry, kde je možné realizovat i zdánlivě neřešitelné úlohy.

Jednu takovou úlohu si klade za cíl právě tato bakalářská práce, kde bude vyzkoušeno otestovat maximum z možností moderních webových prohlížečů a zjištěno, zdali jsou již v takové fázi, aby šel využít jejich potenciál nejen pro uživatelsky co nejpříjemnější webové prezentace či hraní 3D her, ale taky, dle autorova názoru, pro přínosnější aplikace ve formě řešení náročných či dlouhotrvajících grafických a vědeckých výpočtů.

Pro toto maximální otestování je zadáním vytvořit prostředí pro vykreslování 3D animací na klientské straně a prostředí pro distribuované řízení mnoha takových klientů na straně serverové.

Klientská strana musí umět využít svoji grafickou kartu tak, aby mohla vykreslit a odeslat snímky, které po ní vyžaduje strana serverová.

Serverová strana má za úkol rozdělovat zvolenou 3D animaci tak, aby ji bylo možné vykreslovat využitím několika připojených klientů současně.

Tito klienti pro výpočet disponují pouze svými webovými prohlížeči bez využití jakýchkoliv externích doplňků. Jejich zapojení do distribuovaného vykreslování je tedy maximálně zjednodušeno na pouhé otevření vygenerované webové adresy v podporovaném webovém prohlížeči.

Výsledkem takového distribuovaného počinu je vždy kompletní sada vykreslených snímků uložená na straně serveru bez ohledu na to, který klient jaký snímek ve skutečnosti vykresloval a kolik klientů se na společném vykreslování podílelo.

Zdali je tato úloha distribuovaného vykreslování 3D animací s využitím moderních webových prohlížečů řešitelná, případně jestli nastávají některá omezení původní myšlenky, je předmětem obsahu dalších kapitol.

Kapitola 2

Teorie

V této kapitole je uveden výčet a stručný popis souvisejících technologií a principů. Pro výklad dalších kapitol je nezbytné především pochopení způsobu vykreslování 3D grafiky ve webovém prohlížeči a povědomí o formátu COLLADA [2.3] využívaném pro univerzální popis modelovaných scén. V kapitole [3.2] jsou shrnuty již existující řešení, které mohou usnadnit samotný vývoj a odstínit aplikaci teorie.

2.1 HTML5

HTML5 je standard pro tvorbu webových aplikací, který je ještě stále ve stádiu návrhu organizací W3C[1]. Podle zveřejněných odhadů by měl být kompletně dokončen do konce roku 2014. Přibližně do roku 2010 byl tento standard ve velmi raném stádiu, podpora v tehdejších prohlížečích byla mizivá a na prosazování nebyl vyvíjen dostatečný tlak. Většinu moderních funkcí zastalo rozšíření Adobe Flash Player, které bylo (a v současné době stále je) podporováno na více než 90 % klientských prohlížečů [11].

Ovšem právě roku 2010, tedy v roce masivního rozšiřování chytrých mobilních zařízení, vydal Steve Jobs (zakladatel Apple Inc.) prohlášení [12] o rozhodnutí nepodporovat ve svých produktech platformu Adobe Flash s úmyslem začít využívat pro přehrávání multimediálního obsahu nové standardy, jako je například HTML5. Rok poté firma Adobe oznámila [9] ukončení vývoje platformy Flash pro mobilní zařízení, uvolnila pozici multimediálního obsahu na mobilních platformách a tím umožnila další rozvoj HTML5.

Jak již bylo zmíněno, standard stále není dokončen a probíhají jeho úpravy současně s experimentální podporou v moderních webových prohlížečích. Toto experimentování dává pracovní skupině spravující tuto specifikaci cennou zpětnou vazbu, díky které lze pozorovat postupné vylepšování standardu.

Důležitou vlastností HTML5 je zpětná kompatibilita se starším standardem HTML4.01 a tedy kompatibilní zobrazení webů napsaných v jazyce HTML5 ve starších prohlížečích.

Standard HTML5 přináší nejen rozšíření stejnojmenného značkovacího jazyka, ale především specifikace nových JavaScript API[2]. Některé z nich jsou popsány v následujících podkapitolách.

[1] http://www.w3c.org

[2] V textu bude často použita zkratka API, která značí rozhraní pro programování aplikací, tedy sbírku funkcí či tříd, které může programátor při tvorbě aplikací využívat.

2.1.1 JavaScript

`JavaScript` je multiplatformní interpretovaný objektově orientovaný jazyk. Objektově orientované programování v tomto jazyce je založeno na prototypové dědičnosti. Pracuje s objektovým modelem dokumentu (DOM) a jeho modifikací dovoluje překonat původně statickou povahu webových stránek. Tento jazyk tedy zajistil rozšíření dynamických webových stránek a postupem času se zasloužil o standardizaci a vzájemně kompatibilní implementace napříč různými webovými prohlížeči.

Kompatibilita ovšem není úplná a proto nad tímto jazykem postupně vznikají různé frameworky[3], například `jQuery`.

Interpretované skripty v tomto jazyce jsou vkládány přímo do webových stránek (případně jako připojitelný externí soubor) a jsou vykonávány na straně klienta ve webovém prohlížeči. Tyto skripty mohou s využitím nových API rozhraní ze standardu `HTML5` například:

- přistupovat k souborovému systému uživatele,
- **odesílat a přijímat binární soubory pomocí technologie `AJAX`** [2.1.2],
- vytvářet vlákna a vykonávat skripty paralelně (tzv. `WebWorkers`),
- **přistupovat ke grafické kartě pomocí `WebGL` kontextu**,
- zjišťovat uživatelovu geolokaci (například pomocí GPS[4]),
- zjišťovat stav baterie zařízení (zde je vidět orientace na mobilní zařízení),
- zachytávat drag & drop dat (např. souborů) na okno webového prohlížeče,
- a mnoho dalšího ...

2.1.2 AJAX

Asynchronous **J**avaScript **a**nd **X**ML je obecné označení pro změnu obsahu webové stránky bez nutnosti jejího opakovaného načítání. Dovoluje rychlejší a uživatelsky příjemnější způsob aktualizace informací na jinak statické webové stránce.

Nejedná se o jednu technologii, nýbrž o kombinaci technologií `HTML` pro zobrazení dat, `JavaScript` pro úpravu `DOM` a především API `XMLHttpRequest` umožňujícího asynchronní odesílání a přijímání HTTP požadavků. `XML` v názvu objektu je z historických důvodů, kdy jeho použití sloužilo především ke stahování právě `XML` dokumentů.

V dnešní době se častěji využívá přenosu přímo `JSON`[5] objektů bez režie `XML` formátu. Využití `JSON` objektů má také nezanedbatelnou výhodu v tom, že se jedná o nativní datový formát jazyka `JavaScript` a není tedy nutné provádět další parsování (neboli zpracování) těchto dat.

V nejnovějších verzích prohlížečů je k dispozici API `XMLHttpRequest2` (XHR2), které vylepšuje původní verzi verzi zejména o:

- Cross Origin Requests[6],

[3] Framework zajišťuje jistou formu abstrakce nad řešeným problémem, v tomto případě především nad kompatibilitou jednotlivých prohlížečů.

[4] Global Positioning System - družicový polohový systém

[5] JavaScript Object Notation je otevřený standard určený pro výměnu dat. Vychází z jazyka JavaScript.

[6] AJAX požadavky mimo doménu na které je provozován daný skript

- ukazatele průběhu odesílání a stahování dat,
- odesílání a přijímání binárních dat, tedy i souborů,
- AJAXové odesílání celých formulářů (podpora tzv. `Blob` objektů).

Ve spojení s `File API`[7] se jedná o velice mocnou kombinaci pro kompletní správu souborů a jejich síťovou výměnu. Nutno podotknout, že podpora těchto funkcí je zatím velice omezená a je nutné vyvíjet taky zpětně kompatibilní řešení s využitím `XMLHttpRequest` první verze bez těchto vylepšení. Je také nutné uvažovat bezpečnostní rizika s těmito novými API spojená.

2.1.3 WebGL

Web **G**raphics **L**ibrary je nový standard a stejnojmenná multiplatformní 3D knihovna založená na `OpenGL ES 2.0` (viz 2.1.3). Zpřístupňuje JavaScriptové API pro renderování[8] 2D a také 3D grafiky pouze s využitím webového prohlížeče bez nutnosti instalovat jakékoliv doplňky (typicky `Flash player plugin`). Webové prohlížeče pro toto renderování dovolují využít akceleraci pomocí GPU[9] pro dostatečný výkon při zpracování nejen grafických efektů ale například i fyzikálních výpočtů.

Technologii `WebGL` využívá například **Google** a jeho aplikace *Google Maps*, **UC Berkeley** pro vizualizaci MRI mozku [10], připravované 3D videa na **YouTube** nebo nespočet webových konfigurátorů automobilek.

Khronos Group[11] je název organizace, která vyvíjí a spravuje toto API a standardy s ním spojené. Za zvláště naučné lze považovat možnost zapojení kohokoliv do diskuzí a samotného vývoje projektů, na kterých tato organizace pracuje, přes jejich veřejný `GitHub` repozitář [12].

`WebGL` program se skládá z řídící logiky napsané v jazyce JavaScript (jiná možnost není k dispozici) a především z programů připravených pro vykonání v shaderech grafické karty. Více o shaderech bude zmíněno v následující sekci. Za výhodu využití jazyka JavaScript lze považovat automatickou správu paměti.

Pro použití a zobrazení `WebGL` na webové stránce je využit element <`canvas`>, který lze zavoláním metody `getContext()` požádat o přidělení `WebGL` kontextu, na který již lze aplikovat jednotlivé metody, které API poskytuje. Lze žádat o 2D nebo 3D kontext. V následujícím textu je za kontext považován 3D kontext, protože ten reflektuje možnosti `WebGL` a provází celou tuto práci.

OpenGL ES 2.0

OpenGL for **E**mbedded **S**ystems je podmnožinou `OpenGL 2.0 API` a je určeno, na rozdíl od klasického `OpenGL`, pro různá mobilní zařízení, PDA, vestavěné systémy a je taky využito ve webovém prostředí prostřednictvím technologie `WebGL`. Toto API nebo například také `OpenGL` spravuje a vyvíjí stejná organizace a to opět Khronos Group.

[7] API pro přístup k souborovému systému uživatele.
[8] Neboli vykreslování. Dále v textu bude používáno především slovo renderování.
[9] Graphics Processing Unit - grafická karta
[10] http://www.webgl.com/2013/01/webgl-app-fmri-semantic-space/
[11] Tato organizace spravuje i mnoho dalších zajímavých a známých projektů jako například OpenGL. http://www.khronos.org/#slider_webgl
[12] https://github.com/KhronosGroup

Ve srovnání s `OpenGL 2.0` je odstraněna prakticky veškerá pevně naprogramovaná funkcionalita `OpenGL` a veškeré funkce musí programátor implementovat ručně za využití shaderů (viz 2.1.3). Tento krok lze také zpozorovat při přechodu z `OpenGL 3.0` na `OpenGL 3.1`.

Dále zde například chybí podpora pro vykreslování **jiných polygonů nežli trojúhelníků**, je omezena podpora pro antialiasing[13], omezena práce s velkými texturami a odstraněna podpora pro využití více bufferů (například akumulační buffer).

Zpětná kompatibilita tedy není dodržena a aplikace musejí být pro využití `OpenGL ES 2.0` patřičně přeprogramovány a musí být upuštěno od využívání pevně naprogramovaných shaderů a konstrukcí s nimi souvisejících. Stejně jako u `OpenGL` se využívá jazyka `GLSL`, který je popsán dále.

GLSL

Open**GL** **S**hading **L**anguage je jedním z vyšších jazyků, který slouží, stejně jako v `OpenGL`, pro programování shaderů grafické karty. Shader je počítačový program, který slouží k řízení jednotlivých částí programovatelného grafického řetězce. Pro programování shaderů je nutná grafická karta, jež jejich programování umožňuje.

`GLSL` vychází ze syntaxe jazyka `C`. Z pochopitelných důvodů nejsou k dispozici veškeré možnosti tohoto jazyka, jako například práce s ukazateli. Jsou však přidány užitečné vektorové typy `vecX` se zjednodušeným přístupem ke složkám vektoru pomocí tečkové notace, maticové typy, operace mezi vektory a maticemi a další.

Řízení toku v shaderu je limitováno tokem dopředným a cykly takovými, kde lze podmínku jejich iterace (a tedy maximální počet iterací) deterministicky zjistit při kompilaci shaderů.

Důležité jsou vyhrazené proměnné se speciálním významem jako například proměnná `gl_FragColor` určující barvu výsledného pixelu nebo `gl_Position` určující pozici vrcholu a podobně.

Vertex shader

Prvním ze dvou programovatelných shaderů ve `WebGL` je `Vertex shader`. Tento program se provede pro každý vrchol vstupní geometrie vykreslované scény. Na rozdíl od `Geometry shaderu`, kterým `WebGL` nedisponuje, má na vstupu vždy jediný vrchol stejně jako na výstupu. Přidávání nebo odebírání vrcholů tedy není možné.

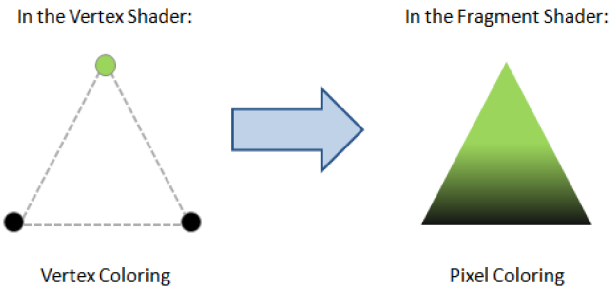
Účelem tohoto shaderu je transformace 3D pozice vrcholů do 2D souřadnic pro následné vykreslení na obrazovce. Typickými operacemi `Vertex shaderu` je transformace vrcholů tak, aby byl objekt správně umístěn ve scéně nebo například aby se simuloval pohled kamery. Lze jej však využít také pro různé efekty. Násobení pozice vrcholu s `model-view` maticí je popsáno v sekci 2.1.5.

Fragment shader

Druhým programovatelným shaderem ve `WebGL` je `Fragment` neboli `Pixel shader`. Je určen pro výpočet barvy každého pixelu vykreslované scény a aplikaci textur. Může upravovat hloubku pixelu, určovat který pixel se vykreslí před kterým, a tím docílit průhlednosti.

Při programování `Fragment shaderu` lze využít speciálního modifikátoru typu proměnné a to modifikátoru `varying`, který zajistí zpřístupnění zvolených proměnných z `Vertex`

[13] Vyhlazování neostrých hran objektů.



Obrázek 2.1: Interpolace ve fragment shaderu. Zdroj: [8], kapitola 2

`shaderu` ve `Fragment shaderu`. Tyto proměnné jsou automaticky interpolovány během provádění `grafického řetězce`. Typicky se využívá této vlastnosti při interpolaci normál vrcholů pro výpočet normál každého vykreslovaného bodu. Je ovšem důležité **znovu vektor normalizovat**, protože během lineární interpolace dochází ke změně délky vektoru. Ukázka interpolace barvy a vztah mezi shadery je znázorněn na obrázku 2.1.

Nutno podotknout, že počet těchto `varying` proměnných v shaderu je omezen, a že mohou být místem výkonnostních problémů. Není tedy vhodné využívat například polí `varying` proměnných (viz problém 4.1.7).

Grafický řetězec

Tento pojem značí sekvenci procesů, jejichž aplikací na data scény získáme dvourozměrný obraz takové scény. O provedení těchto procesů se stará `GPU`. V dnešní době je za grafický řetězec vetšinou považován **programovatelný** grafický řetězec. Na obrázku 2.2 je znázorněn zjednodušený diagram programovatelného grafického řetězce knihovny `WebGL`.

Vertex buffer objects představují objekty, které `WebGL` vyžaduje k popisu vykreslovaných objektů. Patří mezi ně především souřadnice vrcholů, jejich normály, barvy nebo souřadnice textur.

Vertex shader a **Fragment shader** jsou popsaný výše [2.1.3].

Frame buffer je dvourozměrné pole obsahující všechny pixely vykreslované scény a je tedy posledním krokem vykreslovacího řetězce. Tento `frame buffer` reflektuje konečný výstup na obrazovce.

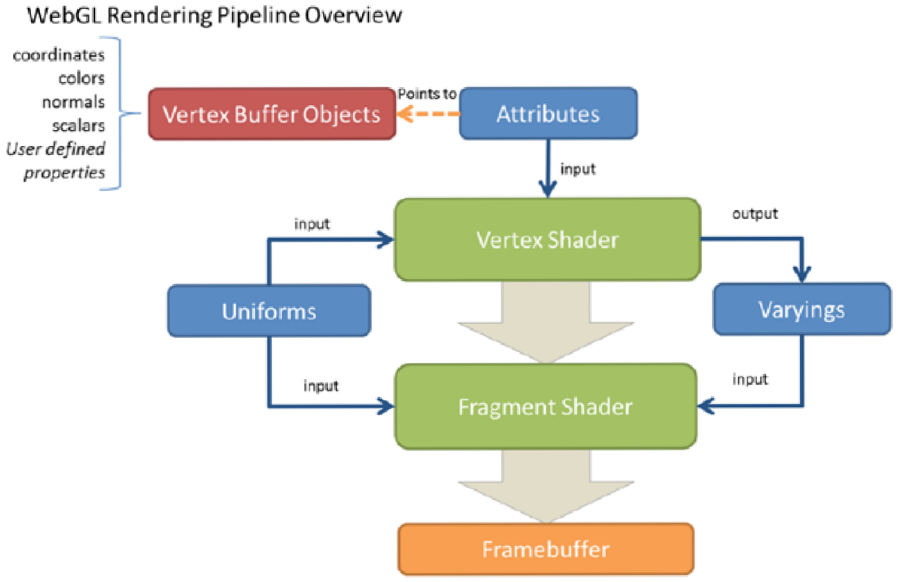
Attributes představují vstupní proměnné do `Vertex shaderu`. Typicky se využívají pro pozici či barvu vrcholu. Vzhledem k tomu, že je `Vertex shader` volán pro každý vrchol, budou tyto `attributes` nést pokaždé jiné hodnoty.

Uniforms představují vstupní proměnné přístupné z obou shaderů. Jsou konstantní, pokaždé nesou stejné hodnoty a proto se využívají například pro předávání pozic světel.

Varyings jsou používány pro přenos dat z `Vertex shaderu` do `Pixel shaderu` tak, jak bylo zmíněno výše v sekci 2.1.3.

2.1.4 Podpora v prohlížečích

Na obrázku 2.3 je uvedena podpora technologie `WebGL` v nejpoužívanějších webových prohlížečích během *května roku 2013*. Jednoznačně nejvíce a nejdéle podporuje tuto technologie prohlížeč `Google Chrome`, naopak `Internet Explorer` vydal prohlášení, že `WebGL` nepodporuje a podporovat nebude z důvodu bezpečnosti při tak nízkoúrovňovém přístupu, který tato technologie vyžaduje [4].



Obrázek 2.2: WebGL grafický řetězec. Zdroj: [8], kapitola 2.

	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	Opera Mobile	Chrome for Android	Firefox for Android
5 versions back	5.5	15.0	21.0	3.1	11.0			2.2		10.0		
4 versions back	6.0	16.0	22.0	3.2	11.1	3.2		2.3		11.0		
3 versions back	7.0	17.0	23.0	4.0	11.5	4.0-4.1		3.0		11.1		
2 versions back	8.0	18.0	24.0	5.0	11.6	4.2-4.3		4.0		11.5		
Previous version	9.0	19.0	25.0	5.1	12.0	5.0-5.1		4.1	7.0	12.0		
Current	10.0	20.0	26.0	6.0	12.1	6.0	5.0-7.0	4.2	10.0	12.1	25.0	19.0
Near future		21.0	27.0									
Farther future		22.0	28.0									

= Supported = Not supported = Partially supported = Support unknown

Obrázek 2.3: Podpora WebGL v jednotlivých prohlížečích. Zdroj: [5].

Za zmínku stojí vývoj podpory `WebGL` na mobilních platformách, kde již existuje mobilní verze prohlížeče `Firefox` nabízející zajímavou podporu této technologie.

Na operačních systémech založených na OS Linux je podpora `WebGL` nižší a nebyla k dispozici v žádném testovaném prohlížeči z důvodu nízké podpory hardwarové akcelerace testované grafické karty v systému `Fedora 16`.

2.1.5 Perspektivní projekce

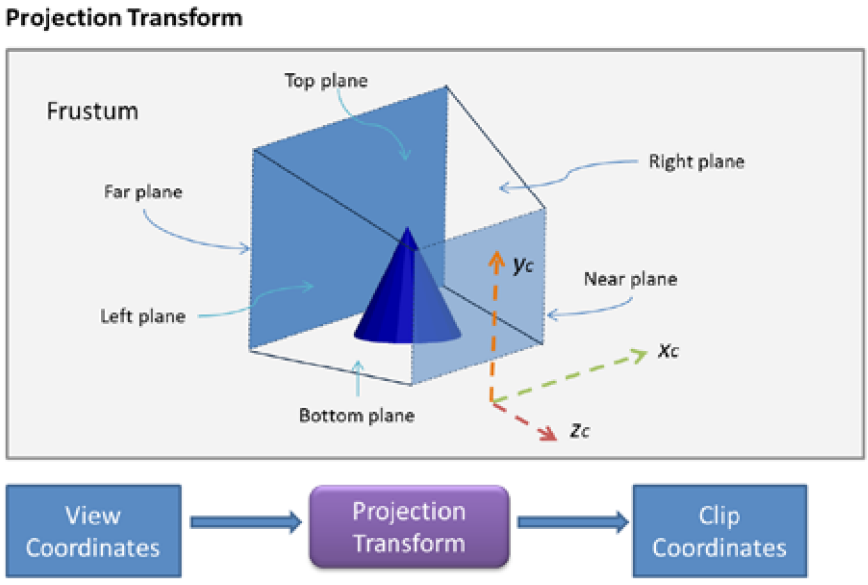
Tato projekce odpovídá našemu vidění reálného světa a proto se používá pro realistické zobrazení ve virtuální realitě nebo počítačových hrách.

Střed projekce představuje pozici pozorovatele a velikost objektů je závislá na jejich vzdálenosti od průmětny. Rovnoběžnost promítaných hran není zachována.

Pro simulaci kamery ve scéně (která ve skutečnosti neexistuje) využíváme vlastnosti perspektivní projekce a to možnost vyjádřit ji prostřednictvím transformační matice.

Tato perspektivní matice v sobě nese údaje potřebné k definici komolého jehlanu známého jako **frustum**. Objem tohoto útvaru určuje, které vrcholy budou vykresleny, a které budou zahozeny a vykreslovat se nebudou.

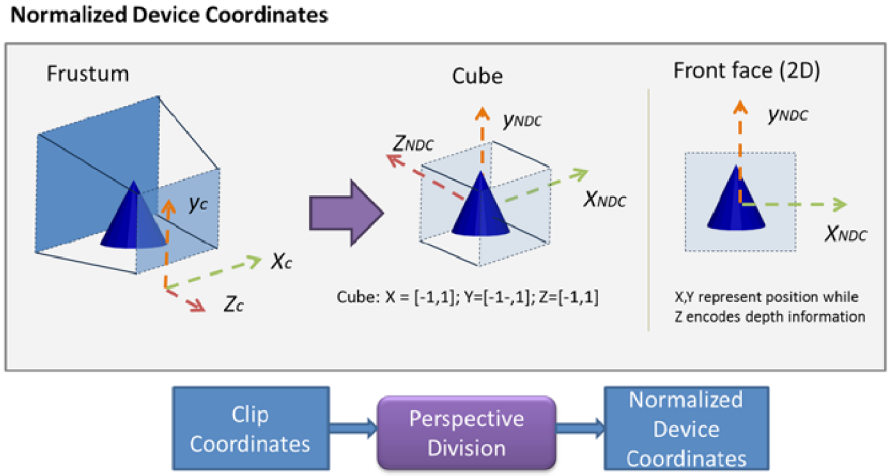
Podle zvolených parametrů perspektivní matice (near, far, top, bottom, right a left plane) lze řídit tvar projekčního jehlanu a tedy ovlivnit, o který typ projekce se jedná. Co jednotlivé parametry znamenají je znázorněno na obrázku 2.4. V případě shodně nastavených hodnot `far plane` a `near plane` se nejedná o projekci perspektivní, nýbrž o projekci paralelní, kterou se zde zabývat nebudeme.



Obrázek 2.4: Projekce. Zdroj: [8], kapitola 3.

Pro 2D výstup je nutné frustum namapovat do `near plane` (obrázek 2.4), což představuje výstup, který bude vykreslen na zobrazovacím zařízení.

Platforma `WebGL`, stejně jako platforma `OpenGL`, pracuje s tzv. **Normalized Device Coordinates** (NDC), což jsou normalizované souřadnice, které lze získat převodem z homogenních souřadnic do souřadnic Eukleidovského prostoru. Jedná se tedy o vydělení složek `x`,`y` a `z` složkou homogenních souřadnic `w`. Tento krok, známý jako `Perspective division` transformuje frustum do jednotkové krychle, která je vycentrovaná v počátku, jako je znázorněno na obrázku 2.5.



Obrázek 2.5: Převod do NDC. Zdroj: [8], kapitola 3.

***Homogenní souřadnice* bodu ve 3D s kartézskými souřadnicemi $[x, y, z]$ je uspořádaná čtveřice $[X, Y, Z, w]$ pro kterou platí $x = X/w, y = Y/w, z = Z/w$. Bod je svými homogenními souřadnicemi určen jednoznačně. Souřadnici w nazýváme váhou bodu.**

V dvojrozměrném prostoru NDC se pro určení pozice využívá složek `x` a `y` a třetí složka `z` udává informaci o hloubce, tedy který objekt bude vykreslen před nebo za objektem jiným.

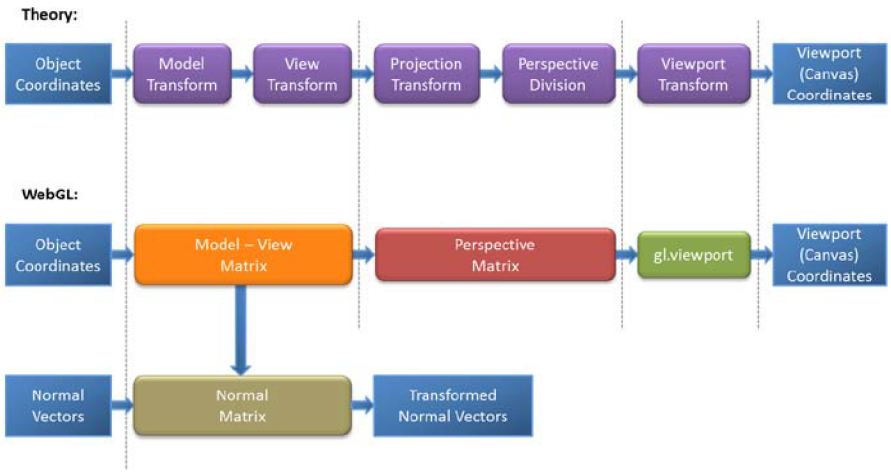
Z popisu objektů v normalizovaných souřadnicích NDC se generují souřadnice odpovídající konečnému zobrazení v elementu `canvas`. Tato transformace již probíhá v režii `WebGL` při zavolání `viewport` funkce.

Implementace ve WebGL

Ve `WebGL` se pro implementaci těchto kroků za účelem korektní projekce scény typicky využívá tří transformačních matic. Jedná se o:

- **Model-View matrix** sjednocuje matici pro transformaci lokálního prostoru souřadnic objektů do prostoru souřadnic kamery.
- **Normal matrix** se využívá pro transformaci normál tak, aby patřičně měnily svůj směr podle transformací aplikovaných na vrcholy pomocí předchozí `Model-View` matice. Lze získat invertováním a transponováním `Model-View` matice (odvození [8], kapitola 3).
- **Perspective matrix** sjednocuje projekční transformace s konečným mapováním do NDC.

Srovnání teorie a možné implementace této teorie ve `WebGL` je znázorněno na obrázku 2.6.



Obrázek 2.6: Srovnání teorie a implementace. Zdroj: [8], kapitola 3.

2.1.6 Osvětlovací model

Programovatelné shadery nám dovolují definovat matematický popis toho, jak bude scéna osvětlena. V této práci je použita kombinace `Phongova` osvětlovacího modelu s `Phongovým` stínováním.

Phongovo stínování

Při tomto stínování se, na rozdíl od stínování `Goraudova`, počítá výsledná barva ve `fragment shaderu`. Je tedy nutné znát normálový vektor pro každý vykreslovaný pixel. Pro výpočet těchto vektorů s výhodou využijeme `varying` proměnné shaderů (viz 2.1.3), které zajistí automaticky potřebnou interpolaci vrcholových normál.

Phongův osvětlovací model

Tento osvětlovací model počítá osvětlení scény pomocí součtu tří světelných složek:

- **Okolní** (`ambient`) světlo představuje všesměrové konstantní osvětlení objektů.
- **Difúzní** (`diffuse`) světlo určuje rovnoměrný odraz do všech směrů nezávisle na pozici pozorovatele.
- **Lesklé** (`specular`) světlo představuje odraz především v jednom směru závisle na pozici pozorovatele

Znázornění podílu těchto složek na výsledném osvětlení objektu spolu s odvozením potřebných rovnic je přehledně uvedeno v [8].

2.1.7 Světla

Základními zdroji osvětlení ve scéně jsou následující dva typy světel.

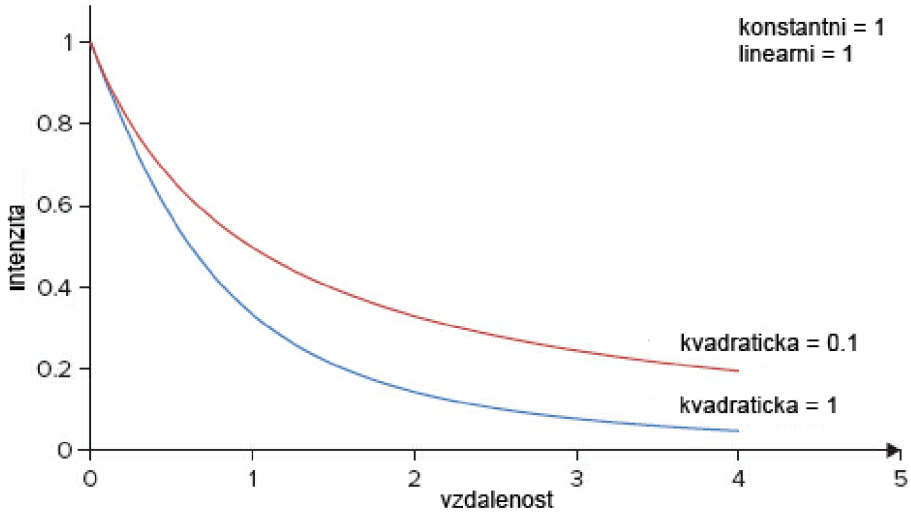
Bodové světlo

Bodové světlo neboli `positional` či `point light` je označení pro světlo, jehož pozice ve scéně ovlivňuje její výsledné osvětlení. Objekty od světla vzdálenější jsou ovlivňovány méně než objekty umístěné blíže. Toto světlo je tedy určeno bodem v prostoru, kde je umístěno.

Pro tento typ světla je typický jev atenuace, který napodobuje chování reálného světla tak, aby jeho intenzita se vzdáleností od objektu při průchodu médiem klesala. Pro výpočet se často užívá vztahu 2.1, kde je výsledná intenzita světla vypočtena pomocí proměnné `vzdalenost` značící vzdálenost světla od osvětlovaného objektu a konstant `konstantni`, `linearni` a `kvadraticka` značící jednotlivé složky atenuace.

Příklad vlivu volby konstant je znázorněn na obrázku 2.7.

$$intenzita = \frac{1}{konstantni + (vzdalenost * linearni) + (vzdalenost^2 * kvadraticka)} \tag{2.1}$$



Obrázek 2.7: Závislost vzdálenosti objektu a světla na jeho intenzitě.

Tento vztah používají mnohé modelovací software jako například `Blender` (3.1) a taky samotné `OpenGL` ve verzích s fixně programovanými shadery.

Směrové světlo

Směrové světlo neboli `directional` či `sun light` je označení pro světlo na jehož pozici ve scéně nezáleží, protože pro všechny objekty poskytuje stejné množství osvětlení (leží ve scéně nekonečně daleko). Záleží ovšem na jeho natočení a tedy na směru vektoru, který toto světlo určuje. Na rozdíl od bodového světla je tedy toto světlo určeno vektorem zohledňujícím jeho směr.

`Blender` pro `sun light` bez rotací předpokládá vektor $(0, 0, -1)$ [14]. Při rotaci takového světla je nutné aplikovat patřičné transformační matice na tento výchozí vektor. Výsledný vektor odpovídá vektoru tohoto světla nastavenému v softwaru `Blender`.

[14] Zjištěno z volně dostupných zdrojových kódů.

2.2 Animace

Základní možností animace grafických scén je využití klíčových snímků (`keyframes`) [15], které popisují stav (tzn. pozici, rotaci, barvu apod.) objektů na daném snímku. Mezi těmito klíčovými snímky je pomocí interpolací počítán stav objektů odpovídající každému ze snímků scény. Těchto interpolací běží mnoho současně, podle náročnosti scény. Zvolené interpolace mohou být těchto typů:

- **Konstantní** (`step`) interpolace vykoná skokovou změnu stavu objektu až když narazí na `keyframe`. Veškeré snímky ležící mezi dvěma `keyframes` nesou stav objektu odpovídající počátečnímu `keyframe` a žádnou animaci stavu nevykonávají.
- **Lineární** interpolace umožňuje lineární návaznost animací například při rotaci kamery ve scéně konstantní rychlostí. Nedochází ke zrychlování ani zpomalování animace mezi jednotlivými `keyframes`.
- **Bézierova** interpolace je interpolace mezi snímky taková, kde průběh interpolace odpovídá pohybu po Bézierově křivce.

2.2.1 Kubická Bézierova křivka

Parametrická křivka užívaná v grafických modelovacích softwarech pro snadný popis trajektorie pohybu objektů a jednoduchou manipulaci pomocí řídicích bodů. Je určena dvojicí bodů P_0 a P_1 určujících počáteční a koncový bod křivky a dvojicí řídicích bodů C_1 a C_2. V animaci počáteční a koncový bod odpovídají například počáteční a koncové hodnotě rotace objektu kolem osy X. Řídicí body poté určují tvar křivky pro regulaci průběhu rychlosti vývoje animace.

Ze vztahu 2.2 vyplývá, že pro pohyb na křivce je nutné znát parametr s ležící v intervalu `<0;1>`. Pro obecné použití Bézierovy křivky by bylo nutné použít některý z algoritmů pro rasterizaci, například `Algoritmus de Casteljau`. Pro použití v animacích, kde je pro modelovací software typické lineární mapování mezi časovým průběhem a parametrem s (tzn. během průběhu se přírůstek času nemění), lze pro odvození tohoto parametru využít vztah 2.3, kde $aktualniCas$ leží v intervalu `<pocatecniCas;koncovyCas>` a představuje časový údaj aktuálně vykreslovaného snímku scény.

$$B(s) = P_0(1-s)^3 + 3C_0 s\ (1-s)^2 + 3C_1 s^2(1-s) + P_1 s^3, s\ \in < 0,1 > \tag{2.2}$$

$$s\ = \frac{aktualniCas - pocatecniCas}{koncovyCas - pocatecniCas} \tag{2.3}$$

2.2.2 Závislé transformace

Pomocí výše zmíněných interpolací lze animovat veškeré transformace objektů jako je posunutí, rotace a změna měřítka, ale i transformace materiálu, barev, světel a kamer.

Pro rozmanité animování nabízí modelovací softwary možnost vzájemných animací objektů. Blender (viz exportéry 3.1) nabízí možnost nastavení rodičovského objektu pomocí `setParent` funkce. Pokud má objekt nastaveného rodiče, kopíruje jeho transformace a případně přidává transformace své. Každý objekt ve scéně tedy musí nejdříve aplikovat transformace svých rodičů a až nakonec transformace své. To je mimo jiné důvod, proč je, jak

[15] V textu se bude vyskytovat především anglická varianta tohoto termínu.

bylo později zjištěno, nutné respektovat u objektů ve scéně i zdánlivě nepodstatné transformace jako je například rotace bodového světla.

Další z možností modelovacích softwarů je umístění pomocných objektů (v Blenderu tzv. `Empty`) pro snazší tvorbu animací například při 360ti stupňové rotaci kamery ve scéně. Toho je typicky dosaženo umístěním `Empty` objektu do středu scény, kde časová osa tohoto pomocného objektu obsahuje klíčové snímky odpovídající rotaci objektu kolem své osy. Kamera, která má tento pomocný objekt nastavený jako svého rodiče kopíruje tuto rotační trajektorii a umožňuje tvorbu zmíněného efektu.

2.3 Formát COLLADA

Pro výměnu scén mezi modelovacími a jinými softwary existuje mnoho různých formátů. Většina z nich je ovšem uzpůsobena pro použití jen v konkrétní rodině produktů shodného výrobce nebo nenabízí možnost komplexního popisu scén včetně animací, kamer nebo fyziky. Takovým formátem však není formát `COLLADA` použitý napříč celé této práce.

Organizace Khronos Group stojí za vývojem i tohoto formátu `COLLADA`, kterému klade za cíl stát se univerzálním formátem, který by široce podporovaly různé aplikace založené na práci s 3D grafikou.

Jedná se o formát vycházející ze schématu `XML` umožňující přenos 3D scén, objektů, animací, fyziky a mnoha dalších informací mezi odlišnými programy a platformami. Specifikace syntaxe tohoto formátu je velice rozsáhlá z důvodu širokého a obecného záběru problematiky. Výtah z obsáhlé specifikace známý jako `Reference CARD` je k dispozici na oficiální webové stránce zmíněné organizace [3].

Tento široký záběr bohužel ústí ve velice roztříštěnou a specifickou podporu pro export a import scén v tomto formátu. Ve výsledku tedy prakticky neexistuje univerzální exportér a následně importér, který by nebyl uzpůsoben použitému software. Pro základní popis objektů a transformací se však přenositelnost dodržovat daří.

Typickou strukturu popisu scény v tomto formátu lze vidět na obrázku 2.8 [16]. Používaná přípona souborů `COLLADA` je **dae**.

2.3.1 Vlastnosti formátu

Mezi vlastnosti tohoto formátu pro tuto práci důležité, lze považovat především vlastnosti z následujícího výčtu.

- Animace jsou ukládány ve formě interpolací (viz 2.2) hodnot v závislosti na **čase**, ne na snímcích. Neukládá se údaj `FPS`, protože je nastavení `FPS` zohledňováno exportérem a časy animací jsou mu uzpůsobeny.
- Díky tomuto způsobu ukládání lze transformace a animace vypočítat pro jakékoliv požadované nastavení `FPS`. Například vteřinovou animaci lze rozdělit a vypočítat na 10 ale i 100 snímků.
- Různé matice jsou indexovány zvlášť (například pozice vrcholů a normály) a je tedy nutné před použitím ve `WebGL`, které požaduje sdružené pole indexů, vykonat přeskládání těchto matic a přečíslování indexů. Způsob realizace je popsán v 4.3.2.

[16] Pro podrobnou specifikaci jednotlivých elementů lze stáhnout kompletní specifikaci z adresy: http://www.khronos.org/files/collada_spec_1_4.pdf

```
<?xml version="1.0" encoding="utf-8"?>
<COLLADA xmlns="http://www.collada.org/2005/11/COLLADASchema" version="1.4.1">
  <asset>
    <contributor>
      <author>Blender User</author>
      <authoring_tool>Blender 2.65.0 r53189</authoring_tool>
    </contributor>
    <created>2013-05-10T16:44:37</created>
    <modified>2013-05-10T16:44:37</modified>
    <unit name="meter" meter="1"/>
    <up_axis>Z_UP</up_axis>
  </asset>
  <library_cameras>
    <camera id="Camera-camera" name="Camera">
      <optics>
        <technique_common>
          <perspective>
            <xfov sid="xfov">49.13434</xfov>
            <aspect_ratio>1.777778</aspect_ratio>
            <znear sid="znear">0.1</znear>
            <zfar sid="zfar">100</zfar>
          </perspective>
        </technique_common>
      </optics>
    </camera>
  </library_cameras>
  <library_lights>
    <light id="Lamp-light" name="Lamp">
      <technique_common>
        <point>
          <color sid="color">1 1 1</color>
          <constant_attenuation>1</constant_attenuation>
          <linear_attenuation>0</linear_attenuation>
          <quadratic_attenuation>0.00111109</quadratic_attenuation>
        </point>
      </technique_common>
...
```

Obrázek 2.8: Úryvek ukázkového COLLADA souboru vyexportovaného pomocí SW Blender.

- `COLLADA` nijak nerozlišuje techniku využití normál pro `Smooth` a `Flat` stínování. Převod je popsán v sekci 4.3.3.
- Používá kubické Bézierovy křivky (vysvětlení viz 2.2.1).
- Nutnost víceprůchodového `parsování` z důvodu komplikované syntaxe, odkazů a vazeb mezi prvky scény. S tím spojené vysoké nároky na `parser`.
- Redundance dat, zbytečné informace a žádná přenosová optimalizace dělají tento formát nevhodným pro síťové použití.

2.4 Distribuce práce a paralelizace

Distribuovaný výpočet je výpočet rozložený na více výpočetních uzlů tak, aby mohly pracovat paralelně a snížit tím čas potřebný k dokončení daného výpočtu.

Aby mohl být distribuovaný výpočet uskutečněn, je nutné rozdělit výpočet na více menších, méně náročných úloh, které lze paralelizovat tak, aby jejich výpočet mohl být vykonáván nezávisle na sobě. Nutnou podmínkou je tedy využívat při výpočtu takový algoritmus, který je možné paralelizovat.

Jako jednoduchý distribuovaný výpočet lze považovat rendering 3D animací, kde lze každý snímek renderovat nezávisle na ostatních a zmíněná podmínka je tedy splněna. Každému výpočetnímu uzlu lze přidělit práci (dávku snímků), kterou má vyrenderovat, zarezervovat mu jí a jakmile bude hotová, přidělit mu práci (dávku) další. Čím více uzlů bude v distribuční síti zapojeno, tím bude rendering výsledné animace rychlejší (při zanedbání omezených zdrojů serveru apod.).

Pro skládání renderovaných snímků do výsledného (často komprimovaného) videa je výhodné přidělovat práci postupně od nejnižšího snímku tak, aby byly snímky s nižším pořadovým číslem renderovány dříve než snímky s pořadovým číslem vyšším.

Kapitola 3

Návrh řešení

V této kapitole je zhodnocen průzkum existujících řešení, na kterých by bylo možné tuto práci založit, sepsány použité knihovny a uveden směr, kterým se práce dále ubírá. Je zde znázorněn návrh celého řešení spolu s komplikacemi, které jej doprovázely. V neposlední řadě je zde popsán navržený výměnný formát a způsob, jakým bude rendering distribuován.

3.1 Exportéry

Mezi softwary podporující formát `COLLADA` patří například `Maya`, `3DS Max`, `Blender` nebo `SketchUp`. Tato práce je uzpůsobena `COLLADA` exportéru softwaru `Blender`. Za jeho výhodu považuji především to, že je vydáván jako open-source a jeho zdrojové kódy lze volně studovat, jelikož jsem této možnosti nespočetněkrát využíval. Další výhodou je široké zázemí uživatelů a vývojářů, protože se jedná o software poskytovaný zdarma. Vzhledem k stále probíhajícímu vývoji exportéru, práce předpokládá užití softwaru `Blender` ve verzi 2.65, který respektuje schéma `COLLADA` ve verzi 1.4.1.

`Blender` nabízí export většiny základních objektů, transformací a nastavení 3D scény, zdaleka však ne všech. Především je důležité dbát na správné nastavení možností exportéru při samotném exportu scén. Většina modelů například využívá objektové modifikátory jako je zrcadlení nebo duplikace, avšak `Blender` má ve výchozím nastavení vypnuto respektování těchto modifikátorů a je nutné toto nastavení v dialogu `COLLADA` exportu změnit.

Jednotlivé omezení exportéru `Blenderu`, na které jsem během vývoje narazil, budou popsány v patřičných kapitolách.

3.2 Existující řešení

Počáteční fází této práce bylo prohledávání různých zdrojů a materiálů za účelem prozkoumání, které části studované problematiky již existují převedené do praxe a zdali některé vyhovují natolik, že by mohly být využity tak, že by na ně tato práce navazovala. Především byl kladen důraz na průzkum existujících frameworků nad `WebGL`, které by odstínily nízkoúrovňový přístup k vykreslování modelů a scén.

Bylo prozkoumáno mnoho existujících frameworků s porovnáním podpory pro funkcionalitu touto prací vyžadovanou. Prakticky veškeré testované frameworky nabízely velice uspokojivou úroveň odstínění `WebGL` přístupu a propracované možnosti vykreslování objektů včetně podpory pro textury, průhlednosti a technik stínování. V dalších zkoumaných, nutno podotknout důležitějších, hlediscích již však dopadly všechny velmi **neuspokojivě**.

A to především z hlediska podpory pro import scén nejen ve formátu COLLADA. Také co se týče podpory pro animace, stačilo vyzkoušet několik vyexportovaných základních animací, aby se ukázalo, že při načítání animací žádný současný framework uspokojivě **nepomůže**.

Nabízely se tedy dvě varianty možnosti vývoje a to buď vydat se směrem vylepšování existujícího řešení, které je ve sledovaných ohledech vývoje nejdále (framework THREE.js [10] s experimentálním, dále **nevyvíjeným** COLLADA loaderem [1]) , nebo začít vyvíjet aplikaci bez frameworku od nejnižší vrstvy přístupu, tedy od kreslení grafických primitiv do kontextu WebGL.

Ačkoliv byla již od začátku ukládána větší priorita samotnému zpracování animací (aby měla distribuce smysl) a načítání formátu COLLADA před kvalitou a rozmanitostí technik renderingu samotných snímků, bylo nejdříve vyzkoušeno zvolit směr vylepšování zmíněného COLLADA Loaderu. Po důkladném prozkoumání, poté co byl učiněn marný pokus se zorientovat v nedostatečně komentovaném kódu bez dokumentace a po diskuzi s vývojaři frameworku THREE.js, bylo rozhodnuto, že pro takovou kombinaci, kterou mi zadání práce ukládá, bude nejlepší volbou začít s vývojem od zmiňované nejnižší úrovně, tedy od WebGL.

Toto rozhodnutí potvrdila nevhodnost použití formátu COLLADA ve webovém prostředí (zmíněno výše 2.3.1). Řešení bude popsáno v kapitole 3.

Pro renderování snímků, řízení animací a distribuci práce tedy nakonec žádný framework použit nebyl, využívají se však tyto knihovny:

- **glMatrix** pro maticové operace 3D grafiky v jazyce JavaScript [2],
- **jQuery** pro snazší práci s DOM [3],
- **SimpleXML** pro serverové parsování XML souborů.

3.3 Způsob řešení

Základní myšlenkou návrhu je oddělení strany klientské a strany serverové. Klientská strana má za úkol renderování snímků, které jí přidělí server. Tento server má za úkol nejen rozdělovat práci připojeným klientům, ale především vystupuje jako parser nahrávaných COLLADA scén exportovaných z Blenderu do formátu JSON. Jako implementační jazyk pro aplikaci na straně serverové byl zvolen pro toto prostředí široce rozšířený jazyk PHP s využitím databázového systému SQLite nebo MySQL, podle náročnosti implementace serveru na práci s databázemi.

Důraz má být kladen na jednoduché použití ve webovém prostředí tak, aby server pro požadovanou scénu vytvořil unikátní odkaz, který lze jednoduše pomocí standardních komunikačních kanálů zaslat osobám, které se otevřením odkazu zapojí do distribuovaného výpočtu. Jako vzorová situace sloužila představa potřeby rychle vyrenderovat (dlouhou) animaci bez nutnosti fyzického kontaktu se stanicemi a nutností na každé stanici ručně spouštět Blender s danou animací nastavenou na odhadnutý podíl snímků, které má stanice vyrenderovat.

Parsování scén do aplikačního formátu JSON na straně serveru je výhodné zejména z důvodu možného předzpracování scény, odstranění nepodporovaných a nadbytečných informací, s tím spojeného snížení přenášeného objemu dat po síti a především proto, že

[1] https://github.com/mrdoob/three.js/blob/master/examples/js/loaders/ColladaLoader.js
[2] Oficiální GitHub: https://github.com/toji/gl-matrix
[3] Document Object Model

jednotlivé klientské stanice nebudou nuceny zdlouhavě redundantně provádět ono parsování, ale namísto toho budou přijímat scény v předpřipraveném formátu, který umí nativně zpracovat bez zvláštního parsování (JSON).

Konverze formátu `COLLADA` do nějaké formy interního formátu je společným rysem většiny studovaných frameworků, protože se nejedná o formát vhodný pro webové prostředí[4].

V prvních fázích návrhu řešení bylo navrhnuto počítání transformačních matic pro klienty na serveru a doručovat klientům namísto čísel snímků přímo tyto matice. Tento návrh byl později přepracován z důvodu enormní síťové zátěže podobného řešení. V konečném návrhu server zasílá klientům popis transformací objektů stejně jako popis objektů samotných a potřebné interpolace pro získání potřebných hodnot k vyrenderování požadovaných snímků vykonávají klienti samotní.

3.4 Navržený systém

Na diagramu 3.1 lze vidět základní případ užití a posloupnost akcí vykonaných v navrženém systému. Ukázková situace modeluje základní myšlenku systému - renderovat jednoduchou vymodelovanou scénu užitím webového prohlížeče za pomoci více výpočetních uzlů (klientů)[5].

Vstupní bod zpracování zahrnuje export scény z modelovacího software ve formátu `COLLADA` a jeho nahrání na server pomocí webového formuláře. Spolu s nahráním scény na server je potřeba nastavit požadované parametry renderování jako například číslo snímku od a po který se má renderování animace vykonat, požadované rozlišení renderovaných snímků, FPS apod. Tento krok je znázorněn v diagramu 3.1 jako **(bod 1)**.

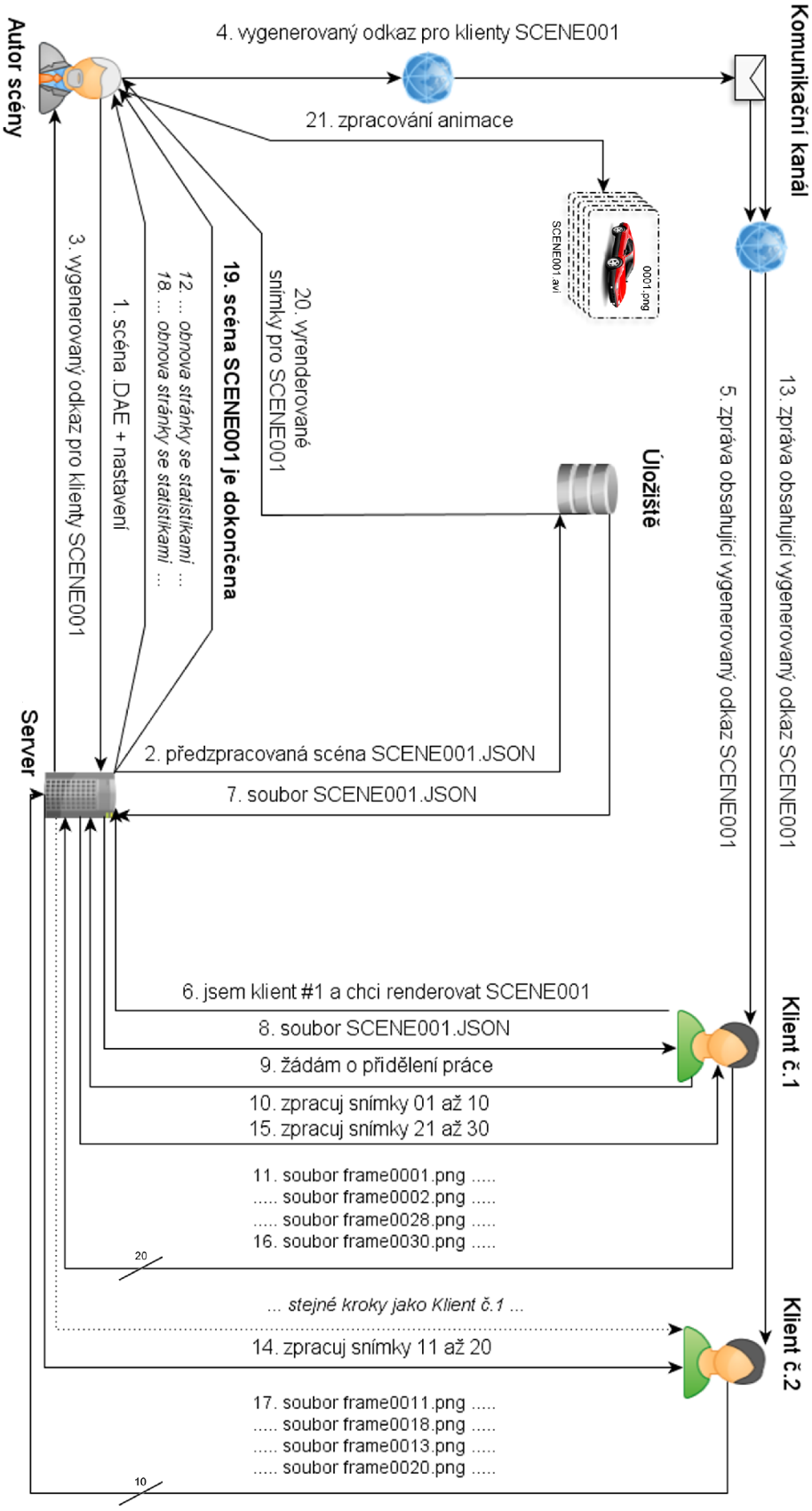
Pokračuje se parsováním nahrané scény do interního popisu (podrobnosti v sekci 3.5) scény ve formátu `JSON` **(bod 2)**. Tento interní popis je uložen na serveru a je pro něj vygenerován jedinečný odkaz, který se zpřístupní osobě, která scénu do systému nahrála **(bod 3)**. Ta musí nyní doručit tento odkaz na klientské počítače, které chce do renderingu zapojit **(bod 4)**. Pochopitelně může zapojit i sebe sama. Zároveň je pro tuto osobu zpřístupněn odkaz do přehledového modulu sloužící pro sledování průběhu distribuce práce a zobrazení statistik renderingu **(body 12 a 18)**.

Následující kroky již mohou probíhat paralelně u více klientů současně. Na diagramu jsou znázorněni takoví klienti dva. Jakmile spustí aplikaci ve svém podporovaném (seznam viz 2.1.4) webovém prohlížeči **(bod 5 a 13)**, vykoná se inicializační sekvence zahrnující jednoznačnou identifikaci daného klienta a stažení interního popisu scény z úložiště serveru **(bod 6, 7 a 8)**. Po dokončení stahování scény ze serveru se začne klient opakovaně dotazovat serveru o přidělení práce, kterou má jako uzel v distribuovaném výpočtu vykonat **(bod 9)**. Tato práce zahrnuje čísla snímků, které má za úkol klient vyrenderovat a zaslat zpět na server **(bod 10, 15 a 14)**. Tyto snímky jsou pro daného klienta dočasně zarezervovány a je tedy zaručeno, že nebudou přiděleny případným ostatním klientům.

Klient začne sekvenčně renderovat snímky z přiděleného rozsahu a postupně je odesílat na server. Vzhledem k různé velikosti snímků a paralelní povaze jejich AJAXového (viz 2.1.2) průběhu odesílání doráží na server v různém pořadí, vždy však z přiděleného rozsahu

[4] Khronos Group právě započala práci na vývoji nového formátu, který by měl být prostředníkem mezi webovými aplikacemi a právě formátem COLLADA. Pracovní název tohoto formátu ve velmi raném stádiu vývoje zní glTF a je k dispozici na oficiálním GitHubu organizace Khronos Group.

[5] V textu se bude vyskytovat obojí - slovo klient i slovo uzel. Uzel většinou v technickém kontextu, kdy lze odstínit myšlenku na uživatele se spuštěným webovým prohlížečem.



Obrázek 3.1: Diagram navrženého systému.

(body 11, 16 a 17). Během renderování má klient možnost zobrazit náhled aktuálně renderovaného snímku, vidět statistiky a má možnost regulovat množství prostředků aplikaci přidělených (ovlivňuje rychlost zpracování).

Jakmile je veškerá práce renderování této scény dokončená, klienti jsou zpraveni o ukončení renderingu a mohou aplikaci vypnout. Žádné další přenosy se již neuskutečňují. Zároveň může osoba zakládající požadavek na rendering spatřit informaci o kompletním dokončení scény ve zmiňovaném přehledovém modulu **(bod 19)**. Nyní lze jednotlivé snímky dále zpracovat **(bod 20 a 21)**, například je zkomprimovat do výsledného videa pomocí externího programu `ffmpeg` [6].

3.5 Výměnný formát

Jak již bylo zmíněno, vstupní popis scény ve formátu `COLLADA` bude konvertován s patřičnými úpravami do interního formátu vyvíjené aplikace. Tento interní formát `JSON` je, na rozdíl od formátu `COLLADA`, nativně zpracovatelný jazykem `JavaScript` v klientských prohlížečích.

Každý soubor v interním formátu bude popisovat právě jednu renderovanou scénu. Navržený formát tvoří především tyto části:

- **settings** - povinné nastavení scény, které lze zadat během nahrávání scény na server
  - **width** + **height** - rozměry renderovaných snímků
  - **fps** - požadovaný frame rate
  - **gammaCorrection** - nastavená gamma korekce scény
  - **clearColor** - barva pozadí renderovaných snímků
  - **sceneStartFrame** + **sceneStopFrame** - rozsah renderovaných snímků
- **mesh** - množina renderovaných objektů scény
  - **id** - interní název objektu
  - **vertices** + **normals** + **indices** - pole popisující geometrii objektu
  - **ambient** + **diffuse** + **specular** + **emission** - popis vlastností materiálu
  - **location** + **rotation** + **scale** - nastavená transformace objektu
- **lights** - množina přítomných světel ve scéně
  - **id** - interní název světla
  - **type** - bodové / směrové světlo
  - **color** - vyzařovaná barva
  - **location** + **rotation** + **scale** - nastavená transformace světla
  - **attenuation** - konstantní + lineární + kvadratická atenuace
- **cameras** - množina přítomných kamer ve scéně
  - **id** - interní název kamery
  - **fovy** + **near** + **far** - nastavení projekce

[6] Pro vytvoření videa ze série png souborů: `ffmpeg -r 24 -i %04d.png -qscale 0 output.avi`

  - **position** + **rotation** + **scale** - nastavená transformace kamery
  - **active** - kamera je / není použita

- **transforms** - množina transformací jakýchkoliv objektů scény
  - **id** - název **transformovaného objektu**
  - **location** + **rotation** + **scale** - popis geometrických transformací
    - **x** + **y** + **z** - souřadnice která odpovídá interpolovaným hodnotám **value**
      - **type** - typ interpolační křivky, která je následujícími vlastnostmi definována `(STEP/LINEAR/BEZIER)`.
      - **time** - množina časových bodů této křivky
      - **value** - množina interpolovaných hodnot této křivky
      - **in** - množina řídících `intangent`[7] bodů
      - **out** - množina řídících `outtangent`[8] bodů
  - **color** + **materialDiffuse** + **materialSpecular** - popis vizuálních transformací
    - **r** + **g** + **b** - barevná složka která odpovídá interpolovaným hodnotám **value**
      - **type** - typ interpolační křivky, která je následujícími vlastnostmi definována `(STEP/LINEAR/BEZIER)`.
      - **time** - množina časových bodů této křivky
      - **value** - množina interpolovaných hodnot této křivky
      - **in** - množina řídících `intangent` bodů
      - **out** - množina řídících `outtangent` bodů

- **extras** - prostor pro speciální objekty scény, jako například `Empty`. Shodné s elementem `mesh`.

Navržený formát dosahuje výrazně nižších objemových velikostí[9] než konvertovaný formát `COLLADA`. Za **největší úsporu** lze považovat přepracovaný způsob popisu transformací, kde je formát `COLLADA` pro účely této práce zbytečně obecný a neseskupuje transformace podle objektu, ke kterému se vážou. Proto je ukládání transformací (které představují stěžejní síťově přenášený prvek této práce) v původním formátu vysoce redundantní a **neefektivní**.

3.6 Způsob distribuce renderingu

Navržen je jednoduchý distribuovaný systém řízení renderingu animací po snímcích. Systém je řízen především těmito parametry, které jsou určeny k experimentování:

- JOB_FRAMES_MIN + JOB_FRAMES_MAX - hraniční velikosti přidělované dávky,
- RESERVATION_SECONDS - čas po který je přidělená dávka snímků klientovi ke zpracování rezervována a nebude přidělena nikomu jinému,

[7] Jedná se o řídící body C_1 Bézierovy interpolační křivky. Držíme se stejného názvosloví jako formát `COLLADA`, kde je pro tečny (HERMITE interpolace) i řídící body (BEZIERE interpolace) použito stejných klíčových slov).

[8] Platí to samé, akorát se jedná o řídící body C_0.

[9] U většiny testovaných scén. Především v případě scén s mnoha transformacemi.

- RANDOM_FRAMES - volí sekvenční nebo náhodné přidělování snímků.

Podle počtu aktuálně pracujících uzlů je určena velikost přidělované dávky snímků. Velikost této dávky je určena vztahem $batch = \frac{length - rendered - reserved}{clients + 1}$ a je omezena intervalem `<JOB_FRAMES_MIN;JOB_FRAMES_MAX>`. Jinými slovy je velikost této dávky dána počtem snímků zbývajících k dokončení podělených počtem aktuálně pracujících uzlů (včetně mě) **navýšeným o jeden další potenciální uzel**. Pokud by toto navýšení ve vztahu nefigurovalo, byla by přidělena maximální velikost dávky již prvnímu připojenému uzlu[10].

S každým odeslaným vyrenderovaným snímkem jsou prodlouženy rezervace všech přidělených snímků danému klientovi (klient je považován za `alive`[11]). Jakmile klient přidělenou dávku zpracuje, dotazuje se serveru o dávku další. Pokud již nejsou veškeré snímky vyrenderovány nebo zarezervovány `alive` klienty, bude tomuto klientovi přidělena další dávka rezervovaných snímků po dobu `RESERVATION_SECONDS`.

Průběh přidělování dávek v sekvenčním a náhodném režimu je znázorněn v příloze A.

[10] Ve skutečnosti by však velikost této dávky byla omezena vhodně nastaveným parametrem JOB_FRAMES_MAX a problém by se týkal pouze rozdělování posledních dávek.

[11] Tedy za klienta, kterému nevypršel čas na oznámení, že stále pracuje.

Kapitola 4

Implementace

Implementace navrženého systému byla rozdělena do tří základních bloků. Nejdříve probíhala implementace klientského renderingu bez jakéhokoliv importu scén. Bylo nutné pochopit základy platformy `WebGL`, způsob programování shaderů a způsob komunikace `WebGL` s klientským `JavaScriptem`. Poté započala postupná implementace základních prvků grafických scén, jako je vykreslování geometrií modelů, implementace vhodného osvětlovacího modelu, podpora pro různé typy světel a v neposlední řadě ovládání kamery. Každý z těchto kroků byl srovnáván s implementací v `Blenderu` a jeho možnostmi `COLLADA` exportu.

Jakmile bylo možné uspokojivě vykreslovat jednoduché scény, přišlo na řadu dávkové načítání těchto scén. Pro tento úkol bylo nutné implementovat načítání interního formátu (3.5) uzpůsobeného možnostem naprogramovaného rendereru. Průběžně se doplňovala další funkcionalita jako `Gamma korekce` nebo podpora pro materiály. Opět byly veškeré implementace funkcionality založeny na testovacím exportování scén z `Blenderu`.

Teprve poté přišlo na řadu statické zpracování transformací objektů brzy následované zpracováním dynamickým, tedy podporou pro animování. Zpočátku byla implementována podpora pro jednoduché transformace objektů lineární interpolací (viz 2.2), později byla však přidána podpora pro animace pomocí dalších typů interpolací a především pro animaci dalších typů objektů ve scénách, jako jsou světla, kamery nebo pomocné `Empty` objekty. Již nebylo možné vykonávat pouze jednoduché translace objektů, nýbrž celý repertoár transformací včetně interpolací barev materiálů apod.

Ruční testování animací bylo nemyslitelné a proto byl současně vyvíjen serverový konvertor exportovaných `COLLADA` souborů do zmíněného interního formátu. Vzhledem ke složitému parsování těchto scén byl parser vyvíjen od této doby až do zakončení celé práce. Export libovolné animace z `Blenderu` ve formátu `COLLADA` již bylo možné jednoduše načíst a vyrenderovat ve webovém prohlížeči.

S podporou pro klientské renderování animací takovou, že bylo možné animace ve scéně libovolně krokovat, zrychlovat, zpomalovat a především renderovat přímo požadované snímky (v nástroji B.1), mohla započít poslední fáze této práce - distribuované řízení dávkového renderingu za pomoci serveru. Tato fáze vyžadovala volbu vhodného databázového systému, logické rozdělení celé aplikace na dílčí části a postupnou implementaci finálního rozhraní pro uživatelské nahrávání scén, sledování průběhu renderování a rozdělování práce.

Poté mohlo započít testování jednotlivých vlastností implementovaného distribuovaného renderingu.

4.1 Renderování

Pro klientské renderování byl vyvíjen a postupně vylepšován specializovaný framework nad `WebGL`, který umožnil v pozdějších fázích implementace odstínit potřebu získávání renderovacího kontextu [1], kompilaci shaderů, svazování proměnných jazyka `JavaScript` s proměnnými jazyka `GLSL` apod. Třídy [2] popisované v této kapitole tedy značí třídy implementované v jazyce `JavaScript` pro klientskou stranu práce.

4.1.1 Testovací nástroj

Současně je vyvíjen také testovací (debugovací) nástroj určený k prohlížení `COLLADA` souborů, který umožňuje ověřit správnost serverového parsování scén a především korektnost klientského renderingu ve srovnání s exportním softwarem (podrobněji v příloze B.1). Tento debugovací nástroj je přístupný i v konečné verzi pro prohlížení scén s možností otáčení kamery, krokování animací apod.

4.1.2 Shadery

Kódy shaderů jsou umístěny v externích souborech a jsou asynchronně načítány při spuštění aplikace. Po načtení jsou zkompilovány a připraveny na grafické kartě k použití. Pro přípravu a kompilaci shaderů kontext `WebGL` disponuje trojicí metod - `createShader()`, `shaderSource()` a `compileShader()`. Pro nahrání shaderů do grafické karty je možné použít metody `attachShader()`, `linkProgram()` a `useProgram()`.

Proměnné

Dalším krokem je mapování proměnných jazyka `GLSL`, tedy proměnných v shaderech, na proměnné jazyka `JavaScript` pro jejich pozdější inicializaci. Pro každou z `attribute` nebo `uniform` proměnných je nutné vykonat `WebGL` metodu `getAttribLocation()`, resp. `getUniformLocation()` [3]. Přístupem na vrácené reference jsou tyto `GLSL` proměnné zpřístupněny pro jejich pozdější inicializaci.

Je nutné podotknout, že `WebGL` funguje na principu stavového automatu, obdobně jako `OpenGL`. Volání zmíněných metod tedy mění vnitřní stav kontextu a záleží tedy na pořadí jejich volání.

Souhrn zmiňovaných metod kontextu `WebGL` s popisem je uveden v tabulce 4.1.

Buffery

Pro mapování bufferů vrcholů, normál a indexů je nutné využít dalších metod uvedených v tabulce 4.2. Klasická pole jazyka `JavaScript` uchovávající pole vrcholů (hodnoty typu `float`), případně indexovací pole (hodnoty typu `int`) musí být před mapováním do odpovídajících bufferů `WebGL` převedena na tzv. typovaná pole [4], které jsou do jazyka `JavaScript` nově zavedeny. Největší podporovaný rozsah těchto polí je momentálně `Float32Array` pro

[1] Základní objekt WebGL umožňující vykonávat veškeré WebGL operace. Dále v textu je slovem kontext myšlen tento objekt.

[2] Ačkoliv je jazyk JavaScript jazykem s prototypovou dědičností a prakticky vše je objektem a třídy v něm prakticky neexistují, budu se z důvodu přehlednosti držet zaběhnutého rozdělení na třídy a instance tříd, tedy objekty.

[3] Detailní popis funkcí: http://www.opengl.org/sdk/docs/man/xhtml/glGetAttribLocation.xml

[4] Specifikace: http://www.khronos.org/registry/typedarray/specs/latest/

Metoda	Popis
createProgram	Vytvoří nový program pro GPU.
linkProgram	Vytvoří spustitelnou verzi vertex / fragment shaderů.
getProgramParameter	Zjišťování stavu programu, např. stav linkování.
useProgram	Instaluje úspěšně linkovaný program na GPU.
getAttributeLocation	Vrací referenci na požadovaný attribute.
getUniformLocation	Vrací referenci na požadovaný uniform.
uniform[1234][fi]	Inicializace vektoru výčtem 1-4 složek FLOAT / INT.
uniform[1234][fi]v	Inicializace vektoru polem 1-4 složek FLOAT / INT.

Tabulka 4.1: Přehled metod `WebGL`.

Metoda	Popis
createBuffer	Vytvoří nový prázdný buffer.
bindBuffer(ARRAY_BUFFER)	Mapuje na GPU buffer pro popis vrcholů.
bindBuffer(ELEMENT_ARRAY_BUFFER)	Mapuje na GPU buffer pro popis indexů.
bufferData	Naplní buffer daty. `NULL` pro uvolnění bufferu.

Tabulka 4.2: Přehled dalších metod `WebGL`.

vrcholy a `UInt16Array` pro indexy. **Z toho vyplývá omezení na 16b rozsah indexů, tedy na maximálně 65535 elementů.** Pro popis rozsáhlejších geometrií je nutné jejich vykreslení rozdělit do více kroků (resp. rozdělit danou geometrii na více menších). Tento proces je popsán v sekci 4.3.5.

Pro samotné vykreslení snímku na elementu <**canvas**> je zavolána renderovací metoda `drawElements()` respektující nastavený indexový buffer. Díky tomu mohou být jednou definované vrcholy a normály využívány opakovaně a není nutné, aby se ve svých bufferech opakovaly. Je použit mód *TRIANGLES*, který vykresluje trojúhelníky vždy tak, že z indexového bufferu načte posloupnost tří vrcholů a ty vykreslí. Ostatní módy mohou být při vhodné úpravě bufferů zajímavou optimalizací (více o rozšířeních viz 6). Platforma `WebGL` **nepodporuje vykreslování větších mnohoúhelníků** nežli trojúhelníků.

4.1.3 Osvětlovací model

V shaderech je implementován výpočet Phongova osvětlovacího model (viz 2.1.6) pro každý fragment vykreslované scény. Jedná se tedy o Phongovo stínování. Mnoho inspirací bylo čerpáno z knihy *OpenGL 4.0 Shading Language Cookbook* [13].

Pro korektní osvětlení odvrácených polygonů scény byla nutná inverze normál při vykreslování těchto odvrácených polygonů. Kód ve `fragment` shaderu zajišťující toto chování je uveden v 4.1.

Ukázka kódu 4.1: Inverze normál

```
if (gl_FrontFacing == false) normal *= -1.0;
```

4.1.4 Geometrie

Pro načtení modelu do scény je naprogramována metoda `addObject()` třídy *Scene*, která vykoná vytvoření a namapování potřebných bufferů (viz 4.1.2) a uloží instanci nového

Vlastnost	Výchozí hodnota
`ambient`	[r:0, g:0, b:0, a:1]
`diffuse`	[r:0.64, g:0.64, b:0.64, a:1]
`specular`	[r:0.5, g:0.5, b:0.5, a:1]
`emission`	[r:0, g:0, b:0, a:1]
`shininess`	50

Tabulka 4.3: Přehled dalších metod `WebGL`.

objektu představujícího model. Později je při renderování scény iterováno nad tímto polem a pro každý uložený model se volá renderovací metoda kontextu *drawElements*().

Blender při exportování scén nijak nesdružuje popisy vrcholů modelů, tedy pokud je do scény přidán stejný objekt vícekrát, je tolikrát redundantně vyexportován jeho popis, takže není nutné implementovat vícenásobné načítání shodných modelů. Nicméně se jedná o možnou optimalizaci a v případě rozšíření parsovacího algoritmu na straně serveru by mohla být tato funkce doprogramována (více o rozšířeních viz 6).

Je podporován pouze popis geometrií ve formě úplných trojúhelníků s využitím indexovacího bufferu. Každou načítanou geometrii v programu reprezentuje instance třídy *Mesh*.

4.1.5 Materiály

V implementované aplikaci je podporována definice materiálů pro jednotlivé modely. Barvu materiálu určují `ambient`, `diffuse`, `specular`, `emission` a `shininess` složky. Texturování a další pokročilé techniky implementovány nejsou, ale mohou být implementovány v rámci rozšíření práce (více o rozšířeních viz 6). Nutno ovšem podotknout, že podpora textur v `COLLADA` exportéru Blender se nenachází na příliš vysoké úrovni.

Pokud není v Blenderu explicitně nastaven materiál objektu, není žádná informace o materiálech exportována a je tedy nutné použít výchozí materiál, který Blender používá. Jeho vlastnosti uvedené v tabulce 4.3 byly zjištěny exportováním modelu se stejným materiálem, ovšem explicitně nastaveným. Hodnota `shininess` v tabulce uvedená je vždy konstantní, protože Blender zohledňuje míru odlesků (`shininess`) již při exportu přepočtem složky `specular`. Údaj `shininess` je tedy při exportu nadbytečný.

Později bylo zjištěno, že při exportu z Blenderu může nastat situace, kdy je k jedné dílčí geometrii přiřazeno více materiálů. Aplikovaným řešením je duplikace těchto geometrií se zachováním shodného identifikátoru (názvu), ale přiřazeným jiným materiálem. Podobný princip využívá dělení rozsáhlých geometrií (viz 4.3.5). Vzhledem k návrhu implementace s možností výskytu duplicitních identifikátorů (názvů) geometrií nebylo nutné nic dalšího tomuto řešení uzpůsobovat.

Materiály jsou implementovány jako vlastnosti odpovídajících geometrií, tedy instancí třídy *Mesh*.

4.1.6 Kamery

Třída *Camera* představuje kameru, kterou lze přiřadit do scény. Je implementována tak, aby reflektovala exportovanou perspektivní projekci scény. Zároveň je pro testovací účely (použito v 4.1.1) obohacena o možnosti uživatelského ovládání s využitím objektu *CameraInteractor*.

Vzhledem k tomu, že Blender neexportuje informaci o aktivně nastavené kameře, je při parsování scény považována za aktivní první zpracovaná kamera a ostatní jsou při parsování ignorovány. Instance třídy *Camera* je tak v renderované scéně vždy pouze jedna,

protože nelze vyexportovat scénu například s animací takovou, která by zahrnovala změnu aktuální kamery. Možným rozšířením k implementaci by mohla být volba použité kamery při parsování scény (více o rozšířeních viz 6).

Pro správnou aplikaci natočení scény vůči kameře je nutné vykonat nejdříve posunutí kamery a poté postupnou rotaci kolem os Z, Y a nakonec X. Pořadí vychází z pravidla o skládání transformací zprava doleva (tedy odzadu). S využitím knihovny `glMatrix` je transformace kamery realizována pomocí posloupností příkazu uvedených v ukázce kódu 4.2, kde proměnná *transform* obsahuje modifikovanou transformační matici, vektor *position* souřadnice, na které má být kamera umístěna a vektor *rotation* jednotlivé úhly rotací. V ukázce je zanedbán převod vyjádření úhlů ve stupních na vyjádření v radiánech.

Ukázka kódu 4.2: Transformace kamery

```
mat4.translate(this.transform, this.transform, this.position);
mat4.rotateZ(this.transform, this.transform, this.rotation[2]);
mat4.rotateY(this.transform, this.transform, this.rotation[1]);
mat4.rotateX(this.transform, this.transform, this.rotation[0]);
```

4.1.7 Světla

Pro přidání světel do scény je implementována třída *Lights* disponující metodami pro správu objektů třídy *Light*. Třída *Lights* obsahuje stěžejní metodu `setMatrixUniforms()` zajišťující inicializaci proměnných shaderů využívaných pro definici vlastností světel. Jedná se o definici barev, umístění (resp. směrů) světel a definici jednotlivých složek atenuace (popsáno v 2.1.7).

Pro výpočet atenuace je použita zmíněná rovnice 2.1. Blender při exportu implicitně zohledňuje nastavení `falloff distance`, tedy vzdálenosti po které se intenzita světla zmenší na polovinu, do parametru kvadratické atenuace.

Vzhledem k faktu, že světel může být ve scéně definováno libovolné množství, je nutné v kódech shaderů pro zmíněné proměnné využít, obdobně jako v ostatních jazycích, polí hodnot. Tyto pole musí mít fixní velikost [5]. Libovolné množství světel ve scéně je tedy shora omezeno velikostí těchto polí.

Přidané světlo může být dvojího typu - implementovány jsou světla bodové a směrové (v souladu s 2.1.7). Pro směrové světlo je nutné vykonat převod popsaný v sekci 2.1.7.

Chyba implementace

V prvotních implementacích byly vektory světel počítány ve `vertex` shaderu a pomocí `varying` proměnných předávány do `fragment` shaderu k dalšímu zpracování. Bylo ovšem zjištěno, že **počet interpolovaných `varying` proměnných shaderů je striktně omezen** (na testovacím stroji maximálně 12 takových proměnných, tudíž zhruba 3 světla, protože se do tohoto omezení počítá každý prvek pole [6]).

Výpočet byl přesunut do `fragment` shaderu, kde žádné omezení nenastává. Původní způsob implementace je zachován v komentářích zdrojových kódů shaderů.

[5] Konstanta MAX_POINT_LIGHTS, resp. MAX_DIR_LIGHTS ve shaderech s výchozími hodnotami rovnými desíti.

[6] Pro zjištění maximálního počtu varying proměnných konkrétní `WebGL` implementace kontext poskytuje možnost volání metody getParameter s konstantou gl.MAX_VERTEX_UNIFORM_VECTORS, resp. gl.MAX_FRAGMENT_UNIFORM_VECTORS.

Gamma korekce

Během testování podobnosti snímků renderovaných Blenderem se snímky renderovanými vyvíjenou aplikací byly zjištěny odchylky v barvách způsobené tím, že Blender zohledňuje nastavení gamma korekce uživatele[7]. Stejně jako v Blenderu byla korekce implementována a lze jí regulovat jedním z nastavení při nahrávání scény na server. Kontext `WebGL` **nemá k dispozici žádnou možnost automatické gamma korekce** (což by navíc bylo nežádoucí kvůli povaze řešeného problému).

Hodnota nastavené gamma korekce je předána do `fragment` shaderu, kde je zohledněna při konečném nastavení barvy vykreslovaného fragmentu podle vztahu 4.1.

$$R = R^{\frac{1}{korekce}}, G = G^{\frac{1}{korekce}}, B = B^{\frac{1}{korekce}} \tag{4.1}$$

4.1.8 Export snímků

Jakmile je snímek korektně vyrenderovaný, je nutné získat jeho podobu ve formě obrázku, který může být dále zaslán serveru pro jeho uložení. WebGL kontext disponuje dvojicí standardních metod `readPixels()` a `toDataURL()` a nově i metodou `toBlob()` k tomuto účelu určených. První zmíněná metoda dokáže vrátit pole pixelů zadaného výřezu, které lze dále libovolně zpracovat. Druhá, v implementaci využitá, metoda vrací URL reprezentaci vyrenderovaného obrázku ve formátu `Base64`[8]. Použitý exportovaný formát je nastaven na **image/png**.

Poslední, taktéž v implementaci využitá, metoda exportu snímků vrací `Blob` objekt představující binární reprezentaci **image/png** obrázku, který lze odeslat využitím `XHR2` (popsáno v 2.1.2). Tato metoda je momentálně podporována pouze prohlížečem `Mozilla Firefox` a jeho využití oproti ostatním prohlížečům **přináší výhodu ve výrazně nižším objemu přenášených dat.**

4.2 Animace

Každý objekt scény představující model, světlo či kameru může nést jednotlivé složky transformace - translaci, rotaci a změnu měřítka. Dále je každý takový objekt opatřen ukazatelem na případného rodiče, ke kterému může své transformace vztahovat. Pro implementaci animací scén shodnou se softwarem Blender bylo zapotřebí shodně implementovat tři metody interpolací popsané v kapitole 2.2.

Pro centrální řízení výpočtu animací je určena třída $Transforms$, která disponuje metodou `loadFrameTransforms(frameNumber)`, která dokáže zajistit vykonání všech transformací veškerých objektů přítomných ve scéně. Jediný její parametr `frameNumber` udávající číslo snímku je uzpůsoben pro jednoduché použití v pozdější fázi distribuce práce.

4.2.1 Zpracování transformací

Jednotlivé transformace jsou načítány z výměnného formátu `JSON` a jsou předparsovány na serveru (viz 4.3). Uložením načtených transformací do instance třídy $Transforms$ umožníme jejich pozdější vykonání v rámci načtení aktuálního stavu scény pro libovolný snímek.

[7] Vysvětlení problematiky gamma korekce lze najít na http://devmaster.net/posts/3022/shader-effects-gamma-correction

[8] Převádí binární data do tisknutelných ASCII znaků, což má za důsledek typicky 33% nárůst objemu dat.

Vykonání metody `loadFrameTransforms(frameNumber)` zahrnuje iteraci nad asociativním polem uložených transformací a jejich postupnou aplikaci voláním implementovaných interpolačních metod s parametrem udávajícím číslo renderovaného snímku. Pokud daný snímek leží vně snímkového rozsahu[9] dílčí animace, musí být aplikována interpolace podle toho, zdali snímek leží před zahajovacím snímkem dílčí animace - parametr interpolace nabývá nejnižší hodnoty $\mathbf{s = 0}$ nebo za konečným snímkem dílčí animace - parametr interpolace nabývá hodnoty nejvyšší $\mathbf{s = 1}$. Pomocí této implementované techniky je zajištěno, že při požadavku na snímek ležící za ukončenou nebo před započatou dílčí animací scéna zůstává v konzistentním stavu.

4.2.2 Problém složené transformační matice

Bylo zmíněno, že každý objekt nese jednotlivé dílčí složky transformace. Nabízí se otázka, proč neukládat pouze, například serverem dopředně vypočtenou, složenou transformační matici namísto transformací dílčích. Původně bylo implementováno takovéto řešení, které však narazilo na nutnost kombinace výchozího stavu objektů (pozice + rotace + měřítko) s **pouze některou exportovanou transformací**, například rotací kolem osy Y. Toto exportování stejných informací na dvou místech `COLLADA` schématu způsobilo mnohé komplikace a vyžádalo zobecnění implementace transformací s ukládáním jednotlivých transformací zvlášť.

Před konečným renderováním, po aplikaci a složení všech transformací, je nakonec stejně vypočtena transformační matice daného objektu, ale vzhledem k tomu, že je tato matice předána přímo do shaderu, žádný další problém se nevyskytuje. Ve shaderech jsou tedy pro aktuálně vykreslovaný objekt přítomny transformační matice jeho vrcholů a normál.

4.2.3 Závislé transformace

Blender v současnosti nedokáže exportovat do formátu `COLLADA` volně se vyskytující křivky, pomocí kterých by šlo objekty animovat (tzv. `path following`). Dokáže však bez potíží exportovat transformace vztahu rodič-potomek, které lze modelovat použitím funkce `setParent` softwaru Blender (popsáno v 2.2.2).

Je implementována obecná podpora pro **libovolné zanořování závislostí transformací** užitím této funkce. Realizace této funkce spočívá v rekurzivním průchodu všech rodičů v řadě za účelem postupného násobení jejich transformačních matic s transformační maticí vykreslovaného objektu. Byla implementována metoda `composeParentalMatrices()` třídy *Scene* k tomuto účelu naprogramovaná. Vzhledem k otočené povaze skládání transformací lze bez problémů postupovat se skládáním transformací směrem od potomka k nejvyššímu rodiči.

Situaci ještě v některých případech komplikuje export inverzní transformační matice `parentInverse` aplikované na rodiče, kterou je nutné v `COLLADA` schématu také parsovat, transponovat (protože `COLLADA` používá opačný způsob zápisu matic nežli `OpenGL`) a doručit správnému objektu.

Byla také přidána podpora pro výskyt `Empty` objektů určených pro vytváření závislých animací na ve skutečnosti neexistujících objektech.

[9] Dříve bylo zmíněno, že animace je v COLLADA formátu reprezentována časovým rozsahem. Nyní již však pracujeme s přepočtenými časy na konkrétní snímky dle nastaveného FPS.

4.2.4 Ostatní transformace

Kromě zmíněných geometrických transformací objektů je dále implementovaná podpora pro:

- **transformace materiálů** - pouze `specular` a `diffuse` složky, protože ostatní Blender neexportuje,
- **transformace barev světel** - také barvy světel mají v Blenderu svoji časovou osu a klíčové snímky.

Zde se naskytuje prostor pro další vylepšování, například přidání podpory pro textury a jejich transformace (více o rozšířeních viz 6).

4.3 Výměnný formát

Navržený výměnný formát (popis viz 3.5) byl téměř beze změn použit při implementaci parseru formátu `COLLADA`. Tím se potvrdilo, že navržený formát je pro podobně zaměřené aplikace dostatečně obecný a lze doprogramovat i jakýkoliv jiný parser dalších formátu, jako například `Wavefront` nebo `X3D`. Vzniká tedy prostor pro možné rozšíření implementace, případně vylepšování implementace stávající (další rozšíření viz 6).

Vzhledem k tomu, že samotné parsování scén trvá zpracovat serveru desítky vteřin, bylo rozhodnutí serverového vykonávání převodu formátu `COLLADA` do nativního `JSON` formátu rozhodnutím výhodným (každý klient by musel minimálně po takovou dobu jen parsovat scénu a to ještě z řádově většího souboru než nyní).

Parser je implementován jako součást serverové strany v jazyce `PHP` s využitím rozšíření `SimpleXML`. V průběhu implementace parseru bylo nutné vyřešit různé komplikace způsobené **vzájemnou nekompatibilitou** formátu `COLLADA` a platformy `WebGL`. Nejdůležitější z nich jsou popsány dále v této kapitole.

4.3.1 Triangulace polygonů

Při exportu scény není standardně zvolena možnost triangulace objektů. Většina exportovaných scén tak obsahuje kromě trojúhelníku (`triangle face`) taky čtyřúhelníky (`quad face`). N-úhelníky s větším počtem vrcholů nežli čtyři Blender při exportu převádí na trojúhelníky automaticky. Pro renderování na platformě `WebGL` je ovšem nutnou podmínkou vykreslovat pouze trojúhelníky, tudíž bylo třeba implementovat jednoduchou triangulaci znázorněnou úryvkem kódu 4.3.

Ukázka kódu 4.3: Jednoduchá triangulace

```
$quadFace = [$a, $b, $c, $d];
$triangle1Face = [$a, $b, $d];
$triangle2Face = [$b, $c, $d];
```

Vzhledem k obecné povaze schématu `COLLADA` je nejkomplikovanější částí triangulace samotná extrakce smíchaných trojúhelníků a čtyřúhelníků z jednoho obecného pole. Při programování této části parseru byl kladen zvláštní důraz na její dostatečné okomentování a pro podrobnosti použitých principů je čtenář odkázán na samotný zdrojový kód[10].

[10] server/libs/Mesh.php, metoda `setGeometry()`, krok 3

4.3.2 Sloučení indexových bufferů

Schéma `COLLADA` ukládá odděleně jednotlivé buffery pro popis vrcholů, normál aj. Nepoužívá však jednotný indexový bufferu (`IBO`) jako platforma `WebGL`, ale používá tyto buffery odděleně zvlášť pro každou ukládanou informaci. Je tedy nutné buffery vrcholů a normál přeorganizovat tak, aby bylo možno každý z nich indexovat pomocí jediného `IBO`.

Jako řídící buffer, tedy ten, podle kterého budou ostatní přeorganizovány považujeme buffer první a tedy buffer vrcholový. Pro druhý, v naší práci využívaný, buffer normálový vygenerujeme pomocnou převodní tabulku indexů. Pomocí té následně vygenerujeme nový, přeorganizovaný buffer normál, který již lze indexovat jednotně pomocí společného `IBO`. Výpočet normál namísto jejich parsování z exportovaného souboru bylo odzkoušeno a zavrženo z důvodu **snížené kvality** výsledného osvětlení ve srovnání s užitím normál exportovaných.

Opět vzhledem k povaze schématu `COLLADA`, kdy je nutné veškeré tyto buffery parsovat z různě provázaných polí, je čtenář v případě zájmu o detaily procesu parsování odkázán na bohatě okomentovaný zdrojový kód[11].

4.3.3 Flat shading

Pro objekty, které nemají vyexportován stejný počet vrcholů jako normál a nelze je tedy standardně implementovaným způsobem vykreslit (rozměry polí `WebGL` musí **vždy** souhlasit), je implementován jednoduchý převod tak, aby tento počet souhlasil. Tato situace nastává když mají exportované objekty nastaveno tzv. `Flat shading`, což značí, že je exportováno méně normál nežli vrcholů (typicky normály vycházející ze středu polygonů).

V této situaci implementovaný parsovací algoritmus automaticky pro každý vrchol zprůměruje příslušící normály, takže vykoná převod `Flat shading` na tzv. `Smooth shading`. Poté již počet vrcholů souhlasí s počtem normál a objekt může být vyrenderován. Podrobnosti převodu opět ve zdrojovém kódu[12].

4.3.4 Přepočet projekce

Většina knihoven, včetně použité `glMatrix`, využívá pro popis projekční matice parametry `fovy`, `aspect`, `near` a `far` (znázorněno na 2.4). Blender však interně využívá a exportuje do popisu projekce namísto parametru `fovy` parametr **fovx**, ačkoliv formát `COLLADA` umožňuje specifikaci parametru obou. Bylo tedy nutné přepočítat tento parametr tak, aby bylo zachováno shodné perspektivní zobrazení.

Specifikace formátu `COLLADA` uvádí [2, 5-93] **chybný vztah** pro přepočet $aspect\_ratio = \frac{xfov}{yfov}$, který je sice možné použít při počítání s lineárními `fov` hodnotami, ne však při práci s hodnotami úhlovými, kterých `COLLADA` využívá [13]. Ze zdrojových kódů exportéru Blenderu lze vyčíst správný vztah (4.2), ovšem v opačném tvaru. Použitím základních goniometrických úprav byl odvozen vztah pro hledaný parametr jako 4.5.

[11] server/libs/Mesh.php, metoda `setGeometry()`, krok 1 a 2
[12] server/libs/Mesh.php, metoda `setGeometry()`, krok 2
[13] Zdroj: ohlášený bug http://sourceforge.net/p/collada-dom/bugs/144/

$$fovx = 2 * atan\left(aspect\_ratio * tan\left(\frac{fovy}{2}\right)\right) \tag{4.2}$$

$$tan\left(\frac{fovx}{2}\right) = aspect\_ratio * tan\left(\frac{fovy}{2}\right) \tag{4.3}$$

$$atan\left(\frac{tan\left(\frac{fovx}{2}\right)}{aspect\_ratio}\right) = \frac{fovy}{2} \tag{4.4}$$

$$fovy = 2 * atan\left(\frac{1}{aspect\_ratio} * tan\left(\frac{fovx}{2}\right)\right) \tag{4.5}$$

Jakmile byl odvozený převod implementován[14], renderované snímky opět odpovídaly snímkům renderovaným pomocí nástroje Blender.

4.3.5 Dělení složitých geometrií

Vzhledem k omezení platformy `WebGL` co do maximální velikosti mapovaných bufferů (viz 4.1.2), bylo nutné, především pro řádné otestování složitých modelů, implementovat dělení vykreslovaných objektů pokud popis jejich geometrie vyžaduje použití větších bufferů nežli platforma povoluje.

Pokud je tedy popis geometrie objektu příliš rozsáhlý, je tento objekt rozdělen na adekvátní počet jednodušších objektů s rozsahem geometrie maximálně podporovaným[15]. Toto dělení má na starost fáze exportu parsovaných dat. Implementačně bylo nutné vykonat vhodné rozdělení vrcholových a normálových bufferů podle bufferu indexového. Typicky vznikají tyto složité geometrie při použití `ARRAY` modifikátoru Blenderu.

Opět pro detaily algoritmu, kde je nutné vykonat přemapování indexových bufferů, je čtenář odkázán na zdrojový kód [16].

4.4 Distribuce práce

Při implementaci systému pro distribuci implementovaného renderování bylo nutné řešit několik zajímavých komplikací. Většina jich pramenila z tohoto nestandardního využívání klientských webových prohlížečů.

Pro identifikaci jednotlivých klientů bylo zvoleno užití náhodně vygenerovaného identifikátoru serverem. Každý renderovací proces tak má unikátní identifikaci dovolující vykonávat více paralelních procesů v jednom prohlížeči současně (výhodné pro testování).

Po obdržení dávky snímků ke zpracování (pouze čísla požadovaných snímků) započne jejich sekvenční renderování a průběžné asynchronní odesílání. Nastává tedy situace, kdy je dávka vyrenderována, ale ještě není odeslána na server. Tento jev se objevuje především na pomalých sítích, které nedisponují dostatečnou propustností pro okamžité odeslání snímků. Proto byl v klientském renderingu zaveden poměr `RENDER_RATIO`, který je udržován tak, aby klient zbytečně nežádal o další dávku snímků (a zbytečně ji nerezervoval), když ještě nemá odeslanou vyrenderovanou dávku předchozí.

[14] A ve skutečnosti ještě rozšířen o korektní převod úhlů ze stupňů na radiány.
[15] Tedy 65535 vrcholů.
[16] server/libs/Parser.php, metoda `splitMesh()`

Kromě toho byl zpozorován jev, kdy toto omezení ani nestihne zasáhnout z důvodu zahlcení odesílací linky prohlížeče tak, že `HTTP` požadavek na další práci není zpracován, dokud není uvolněna fronta odesílaných snímků. Prohlížeče bohužel **nedisponují žádnou možností regulace priority jednotlivých `HTTP` požadavků**. V zásadě by nebylo potřeba tento problém řešit, ovšem dochází zde k časovým mezerám, kdy klient nemůže získat další dávku ke zpracování a jeho rendering je po tuto dobu **pozastaven**. Tento problém je námětem k možnému vylepšení (více o rozšířeních 6).

Pro „rozumné“ využívání systémových zdrojů webovou aplikací podporují prohlížeče funkci `requestAnimationFrame`, která zavolá překreslení elementu <`canvas`>, jakmile to bude „rozumné“. Toto chování ovšem není akceptovatelné v této práci, kde je naopak usilováno o využití maxima, co prohlížeč dokáže. Samozřejmě však není účelem dovolit, aby prohlížeč zamrzával a stával se neovladatelným. Namísto zmíněné funkce je využito plánování prohlížeče pomocí funkce `setTimeout` s nastaveným umělým zpožděním mezi jednotlivými renderovacími cykly. Již samotné použití této funkce bez nastaveného zpoždění (tedy s parametrem `timeout` rovným nule) způsobí naplánování volání funkcí tak, že nedochází k úplnému pozastavení reakcí prohlížeče. Pro lepší uživatelský zážitek je však při renderingu přítomen posuvník umožňující jednoduché ovládání tohoto **uměle vkládaného zpoždění**.

Vzhledem k možnému selhání síťové komunikace, kdy se některé snímky mohou ztratit během přenosu, jsou veškeré síťové požadavky pojištěny mechanismem automatického zotavení pokusem o opětovné odeslání sebe sama po uplynutí zotavovací doby[17].

Rezervace snímků pro jednotlivé klienty je implementována v souladu s návrhem v kapitole 3.6. Realizace je založena na ukládání časových razítek snímků do databáze, kde jsou rezervace prodlužovány nebo rušeny podle vypršení nastaveného `RESERVATION_SECONDS` limitu. Pokud klient rendering náhle přeruší, nejpozději po uplynutí tohoto limitu bude jeho práce přerozdělena mezi klienty ostatní.

Ukládání veškerého stavu distribuovaného renderingu, tedy aktuálně vyrenderovaných snímků, jejich rezervací, nahraných scén a připojených klientů je realizováno s využitím databázového systému. Pro jednoduchost a snadnou přenositelnost byl původně zvolen databázový systém `SQLite 3`. Později byl však nahrazen systémem `MySQL 5` z důvodu **nedostačující podpory pro vzájemně operující přístupy** do databáze[18].

[17] Tato doba je nastavena v konstantách v souboru `join.js`.

[18] Oficiální doporučení viz odstavec `High Concurrency`, web: http://www.sqlite.org/whentouse.html

Kapitola 5

Testování

Pro otestování reálného využití vyvíjeného programu bylo nutné vymodelovat jednoduché scény, které by měly využít potenciál aplikace. Vzhledem k výkonu současných grafických karet, kdy implementovaný kód ve shaderech (viz 4.1.2) je příliš jednoduchý pro vysoké zatížení jednotky GPU, bylo nutné zaměřit testování na vysoce polygonální modely a dlouhé animace.

Testovací scény, přiložené na CD, jsou velice jednoduché a jejich cílem je pouze demonstrovat implementovanou funkcionalitu. Jsou použity pouze modely volně dostupné ke stažení ze zdroje [6] a modely ukázkových příkladů knihy [8].

Pro renderování odpovídající výstupu modelovacího programu Blender, je nutné při exportu nastavit některé parametry. Především se jedná o vypnutí metody `Raytrace`, stínů a textur. Zároveň je možné vypnout také `Anti-Aliasing`.

Předmětem testování jsou především tři scény – s vysokým počtem polygonů (`gardenCar800.dae`), s vysokým počtem snímků (`earth7200.dae`) a scénu průměrnou (`carObserve700.dae`). Tyto názvy budou dále v textu používány.

5.0.1 Způsob testování

Pokud není řečeno jinak, je renderováno 300 snímků dané testované scény a je měřen čas dokud nejsou veškeré snímky úspěšně odeslány na server umístěný na síti LAN. Je použita **náhodná strategie přidělování** snímků (viz A). Standardně je ponecháno nastavení nahrávané scény na výchozích hodnotách[1]. Pro testování bylo **vynuceno zakázání** jakýchkoliv `cache` pamětí přidáním náhodného identifikátoru do adres přenášených souborů.

5.0.2 Srovnání prohlížečů

Prvním testem bylo srovnání webových prohlížečů za účelem zjištění, který je v této aplikaci nejrychlejší a bude použit v dalších testech. Srovnání shrnuje tabulka 5.1. Testovací scény jsou renderovány vždy pouze jedním uzlem (pouze testovaný prohlížeč) na serveru `localhost`.

Prohlížeč `Opera` musel být vyřazen, protože exportoval pouze prázdné černé snímky[2]. Ačkoliv je v čistém renderování rychlejší prohlížeč `Google Chrome` [3], jeho nevýhoda ex-

[1] Tedy rozlišení 960x600 pixelů, 24 fps a automatické hranice animace.

[2] Později bylo zjištěno, že problém způsobuje nedostatečná kompatibilita Opery se standardy. Nelze přistupovat do polí nestatickým identifikátorem. Problémem černých snímků tedy bylo osvětlení s proměnným počtem světel.

[3] Podle testovacího nástroje, kde dokázal využít téměř 100% CPU a GPU. Více o nástroji v sekci 4.1.1

Prohlížeč	carObserve700 [s]	gardenCar800 [s]	earth7200 [s]	ϕ [s]
Google Chrome 26	60,2	79,8	78,8	72,9
Mozilla Firefox 20	51,1	64,8	69,7	**61,8**
~~**Opera Browser 12**~~	0	0	0	0

Tabulka 5.1: Srovnání rychlosti webových prohlížečů. Nižší doba zpracování je lepší.

Síť (upload)	carObserve700 [s]	gardenCar800 [s]	earth7200 [s]	ϕ [s]
LAN ($\sim$ 40 Mbit/s)	57,2	76,8	75,8	69,9
WAN ($\sim$ 5 Mbit/s)	203,5	308,6	470,2	327,4

Tabulka 5.2: Srovnání přenosových sítí. Nižší doba zpracování je lepší.

portu snímků ve formátu `Base64` (viz 4.1.8) měřené výpočetní časy ovlivnila natolik, že se nejrychlejším prohlížečem pro renderování testovacích scén stal prohlížeč **Mozilla Firefox 20**. Další testování již bude probíhat pouze v něm.

5.0.3 Srovnání sítí

Ačkoliv jsou výsledky tohoto testu předvídatelné, bylo pro srovnání rozdílů otestováno distribuované renderování na síti s vysokou přenosovou kapacitou (LAN) a na síti s kapacitou omezenou (WAN). Vzhledem k výsledkům, které shrnuje tabulka 5.2, je pro další testování uvažována pouze síť **LAN**.

Při provozu na síti WAN byla zaplněna fronta snímků k odesílání na takovou úroveň, že nebylo možné získat další dávku snímků ke zpracování ihned, ale bylo nutné čekat, dokud se část fronty neuvolnila (viz 4.4). Využití výpočetní síly daného uzlu bylo tedy minimální (velké pauzy před stáhnutím další práce). **Tato síť je tedy z hlediska přenosové kapacity pro tuto aplikaci nevhodná.**

Pokud by renderování jednoho snímku trvalo výrazně déle (například naprogramováním složitých shaderů), použití této sítě by **začalo mít smysl** (více o rozšířeních viz 6).

5.0.4 Srovnání rozlišení

Zde byla testována vybraná scéna s různým nastavení rozlišení požadovaných snímků. Shrnutí lze vidět v tabulce 5.3.

Větší rozlišení renderovaných snímků vyžadovalo větší nároky na přenosové pásmo, což zapříčinilo postupné plnění fronty snímků k odeslání taky na síti LAN. Tato fronta se však průběžně uvolňovala a samotný přenos dat tedy téměř neovlivnil výsledné časy zpracování. Čas potřebný pro rendering větších snímků roste úměrně s jejich rozlišením.

Znatelné bylo **zpomalení UI prohlížeče** při extrémním rozlišení 3840x2400 px[4] .

[4]Vhodné například pro antialiasing, kdy bude výsledný snímek zmenšen.

Rozlišení [px]	carObserve700 [s]
960x600	51,1
1920x1200	130,2
3840x2400	438,0

Tabulka 5.3: Srovnání rozlišení renderovaných snímků. Nižší doba zpracování je lepší.

Počet klientů	carObserve700 [s]	gardenCar800 [s]	earth7200 [s]	ϕ [s]
1	145,4	195,7	1754,4	698,5
2	78,5	103,2	1454,5	545,4
3	59,6	82,3	691,8	**277,9**

Tabulka 5.4: Srovnání počtu uzlů distribuovaného renderingu. Nižší doba zpracování je lepší.

Počet klientů	earth7200 [s]
1	5806,0
5	1236,5
10	611,3

Tabulka 5.5: Výsledky testování distribuovaného renderingu v prostorách CVT. Nižší doba zpracování je lepší.

5.0.5 Srovnání distribuovaného renderingu

Patrně nejzajímavější výsledky testování nabízí testování vlivu zapojení různého počtu klientů do distribuovaného renderingu. Při těchto testech je pro každou scénu vykonáno renderování se standardním nastavením v **maximálním rozsahu** animací, tedy ne pouze 300 snímků jako v testech předchozích. Měřen je čas, dokud nebudou veškeré snímky dané scény úspěšně uloženy na serveru umístěném v síti LAN.

Ze srovnání naměřených hodnot v tabulce 5.4 je patrné, že mezi jedním a třemi současně renderujícími klienty vzniká zhruba **240%** zrychlení s minimálním rozdílem typu renderované scény. Pro ověření podezřelých výsledků dlouhé animace **earth7200** bylo uskutečněno další měření v jiném prostředí s více dostupnými výpočetními uzly. Jednalo se o prostory laboratoří **Centra Výpočetní Techniky, FIT VUT, Brno**. Zde byla měřena tato scéna za pomoci současně až 10ti renderujících uzlů. Výsledky jsou shrnuty v tabulce 5.5. Zrychlení je téměř úměrné počtu renderujících stanic, protože na zdejší gigabitové síti lze naprosto zanedbat síťové přenosy. Ve srovnání s předchozími testy v domácích podmínkách zde byly přítomny počítače s o několik řádů slabšími grafickými kartami, což bylo silně znát i při práci s testovacím nástrojem (jeho popis viz 4.1.1). Operační systém, verze prohlížečů i ostatní parametry byly nastaveny shodně s domácími podmínkami. Vyloučit lze přetížení odděleného řídícího serveru, protože s každým nově připojeným klientem **rychlost distribuovaného výpočtu rostla**.

Za nejvýkonnější testovanou kombinaci tak lze považovat kombinaci tří vysoce výkonných uzlů v domácím prostředí na dostatečně rychlé **100Mbit LAN** síti. Pokud by byly přenášeny výrazně větší snímky (např. kvůli velmi vysokému požadovanému rozlišení), testovací prostředí CVT s **1000Mbit LAN** sítí by jistě získalo určitou výhodu.

Nejlepší získaný výsledek, tedy rendering dlouhé animace `earth7200`, za čas **691,8 vteřin** byl postaven do kontrastu s časem potřebným pro vyrenderování stejné scény v SW Blender[5]. Jelikož se jedná o rozdílně zaměřené renderovací platformy (Blender využívá jiných, pokročilejších algoritmů pro celkově kvalitnější renderovaný výstup), jsou naměřené hodnoty uvedeny pouze pro zajímavost a to v tabulce 5.6.

Posledním testem bylo využití GPU. Pokud byla zapnuto exportování snímků (nesouvisí nutně s jejich odesíláním po sítí), tzn. pokud byly použity metody `toBlob()` a `toDataURL()`,

[5]Nastavení optimalizováno podle: http://www.blenderguru.com/13-ways-to-reduce-render-times/

Platforma	earth7200
Blender	více než 2 hod
WebGL 1x	téměř 30 min
WebGL 3x	téměř **12 min**

Tabulka 5.6: Srovnání nejlepších výsledku platformy WebGL s výsledkem SW Blender při nastavení na nejvyšší možný výkon (a tedy nejnižší možnou kvalitu).

GPU nebylo možné kontinuálně vytížit nad **30%**. Po vypnutí tohoto exportu bylo možné (v prohlížeči Google Chrome) využít **až 99% GPU** (s odpovídajícím dopadem na odezvu celého hostitelského systému).

Kapitola 6

Závěr

V úvodu bylo jako cíl práce stanoveno otestovat maximum možností, které nám všem moderní webové prohlížeče nabízejí. Jak bylo v rámci testování zjištěno, netradiční úkol, který klade zadání práce, je ideální ukázkou, čeho všeho jsou tyto prohlížeče schopny. Především pohled na místnost plnou desítek počítačů, které paralelně jen pomocí webových prohlížečů vykonávají vykreslování scény, která byla ještě před pár chvílemi určena pouze pro specializované modelovací programy, je toho důkazem.

Bylo zjištěno, že platforma `WebGL` je pro vykreslování 3D grafiky zajímavou alternativou ke standardním možnostem vykreslování pomocí nativních aplikací operačních systémů. Jazyk `JavaScript`, který byl pro tuto platformu patřičně uzpůsoben, disponuje dostatečným potenciálem pro výpočty potřebných interpolací během animování scény. Méně známé API tohoto jazyka, zvané `Web Workers`, které umožňuje používání vláken v prostředí webového prohlížeče, by při vhodném využití mohlo nabídnout zajímavé rozšíření práce v ohledu rychlejšího klientského zpracování a vylepšení odezvy prohlížeče.

Vzhledem k rozhodnutí implementovat vykreslování bez pomoci knihoven a frameworků bylo nutné postupovat od úplných základů vykreslování grafických primitiv pomocí shaderů grafických karet. Bylo nutné se také zaměřit na výpočty animací, konvertování popisu scén z formátu `COLLADA` a v neposlední řadě také na distribuované řízení renderingu. Implementace oblastí jako je podpora pro průhlednosti, textury a stínování je tedy přenechána pro případné rozšíření vyvíjeného frameworku.

Aby bylo načítání scén z formátu `COLLADA` efektivní, bylo nutné vytvořit konvertor do navrženého výměnného formátu `JSON`, který dosahuje nižších velikostí scénových souborů a disponuje lepší čitelností obsažených dat. Tento formát je podporován vyvíjeným klientským frameworkem a lze do něj v případě rozšíření konvertovat i libovolné vstupní formáty jiné.

Nároky na server, který řídí distribuci práce a především vykonává převod `COLLADA` souborů, jsou větší než bylo očekáváno. Pro náročnější scény je použité rozšíření `SimpleXML` nevhodné, pomalé a neefektivní. Za momentálně nejdůležitější budoucí rozšíření práce je tedy považováno přeprogramování této části serveru, aby mohly být testovány ještě rozsáhlejší scény.

Počet klientů zapojených do distribuovaného výpočtu má výrazný dopad na rychlost celého vykreslování, což je uspokojivý výsledek pro část zabývající se distribuovaným přídělem práce. V testech vyšlo najevo, že připojení dalšího výpočetního uzlu zlepšilo průměrný čas vykreslování o přibližně 65%. Zbytek zaujímá režie spojená s přenášením snímků a rozdělováním práce. Méně uspokojivé bylo zjištění, že prohlížeče nejsou optimalizovány pro další využití již jednou vykreslených dat a rychlost zpracování těchto dat pro následné odeslání na server výrazně omezilo využití maximálního potenciálu jednotlivých uzlů.

Původní myšlenka připouštěla využití této práce na sítích s omezenou šířkou pásma (typicky na Internetu). Již základní testování odhalilo extrémní náročnost na objem přenášených dat a tedy na šířku přenosového pásma. Pro rozšíření původní myšlenky by bylo nutné implementovat využití ztrátové komprese nebo jinou metodu redukce přenášeného objemu dat. Zvláště neefektivní je přenášení snímků ve formátu `Base64` při použití jiného prohlížeče než `Mozilla Firefox`, který jako jediný dokáže přenášet snímky binárním přenosem.

Testování na výkonné stanici odvádělo od myšlenky vymodelování dostatečně náročné scény na to, aby byť i jediný snímek trvalo vykreslovat několik vteřin. Při testování na o několik řádů slabších stanicích tato myšlenka opět přišla v úvahu, stejně jako naprogramování mnohem náročnějších shaderů tak, aby distribuovaný rendering nabyl dalšího rozměru využití. Tyto možnosti jsou ponechány jako prostor pro zajímavé vylepšení práce.

Nelze říct, který typ testované scény se pro distribuovaný výpočet hodil nejvíce, protože bylo zjištěno, že velmi záleží na počtu zapojených uzlů. Pro dlouhé animace s méně náročnými modely je rozdíl mezi jedním a dvěma uzly řádově menší než mezi třemi, čtyřmi a tak dále. Naopak pro krátké animace s náročnými modely je znát výrazný nárůst zrychlení již od dvou zapojených uzlů. Za nejobjektivnější test lze považovat test v prostorách CVT, kdy bylo využito 10ti současně pracujících uzlů a bylo dosaženo celkového zrychlení **o více než 950%** ve srovnání s uzlem jediným.

Dalším námětem k rozšíření práce je experimentování s parametry serveru zmíněnými v sekci 3.6. Bylo zpozorováno zbytečné čekání klientů při přidělování dávek snímků ke zpracování. Bylo by tedy vhodné upravit klientskou logiku tak, aby bylo o dávky žádáno s předstihem a ideálně také logiku serverovou, aby velikost přidělovaných dávek zohledňovala nejen vůči aktuálnímu počtu pracujících klientů ale i vůči počtu snímků, které již klient dodal. Byli by tím vhodně upřednostněni rychlejší klienti, kteří momentálně nijak zvýhodněni nejsou.

Kromě původního účelu zrychlení renderování dlouhých scén lze implementované řešení využít například, vzhledem k možnosti jednoduchého přeprogramování shaderů, pro distribuované renderování vysoce náročných scén, které nelze renderovat jedním uzlem v reálném čase. Zastoupení více uzlů v distribuovaném prostředí by mohlo toto renderování v reálném čase umožnit a simulovat ho. Zajímavým využitím distribuovaného přístupu by mohlo být řízené přesouvání `viewportu` ve scéně, čehož by šlo využít například v technice globálního osvětlování `Final Gathering`.

Práce byla úspěšně dokončena v souladu se zadáním. Prakticky v každé části vývoje se však objevovaly nápady na další a další možnosti vylepšování a rozšiřování implementované funkcionality. Nebylo však možné na bezmála osmi tisících řádcích naprogramovaného kódu stihnout zrealizovat všechny a proto zde bylo uvedeno několik těch, které by autor rád zrealizoval alespoň v budoucnu. Vzhledem k nedostatku podobných existujících řešení bude zváženo uvolnění některých částí jako Open Source projekt.

Tato bakalářská práce autorovi umožnila široké rozšíření obzorů v oborech počítačové grafiky a distribuce výpočtů. Mezi nově nabyté zkušenosti lze zařadit programování shaderů moderních grafických karet, modelování animací v softwaru Blender a prozkoumání hranic pojmu moderní webový prohlížeč. Tato práce ovlivnila směr, kterým bude autor směřovat dále své vzdělání.

Na závěr možná překvapující informace - v extrémních případech prohlížeče dovolí vyčerpání prakticky všech systémových prostředků libovolnou běžící webovou stránkou a to i v prostředí Internetu. Tím je částečně opodstatněno rozhodnutí firmy Microsoft ohledně implementace technologie `WebGL` do svého prohlížeče [4].

Literatura

[1] WebGL Inspector - An advanced WebGL debugging toolkit. http://benvanik.github.io/WebGL-Inspector/.

[2] COLLADA - Digital Asset Schema Release 1.4.1. http://www.khronos.org/files/collada_spec_1_4.pdf, březen 2008.

[3] COLLADA 1.4 Quick Reference Card. http://www.khronos.org/files/collada_reference_card_1_4.pdf, 2010.

[4] Microsoft: WebGL Considered Harmful. http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx, červen 2011.

[5] Compatibility tables for support of HTML5, CSS3, SVG and more in desktop and mobile browsers. http://caniuse.com/webgl, květen 2013.

[6] Google Sketchup 3D Warehouse: Galerie 3D objektů. http://sketchup.google.com/3dwarehouse/, květen 2013.

[7] Learning WebGL: 3D Programming for the Web. http://learningwebgl.com/blog/?page_id=1217, květen 2013.

[8] Cantor, D.; Jones, B.: *WebGL Beginner's Guide*. Packt Publishing, Limited, 2012, ISBN 9781849691734.
URL http://books.google.cz/books?id=uEmzeCKd8ZEC

[9] Danny Winokur, A.: Aggressively Contribute to HTML5. http://blogs.adobe.com/conversations/2011/11/flash-focus.html, listopad 2011.

[10] Parisi, T.: *WebGL: Up and Running*. Oreilly and Associate Series, O'Reilly Media, Incorporated, 2012, ISBN 9781449323578.
URL http://books.google.cz/books?id=uYnyaBClb3IC

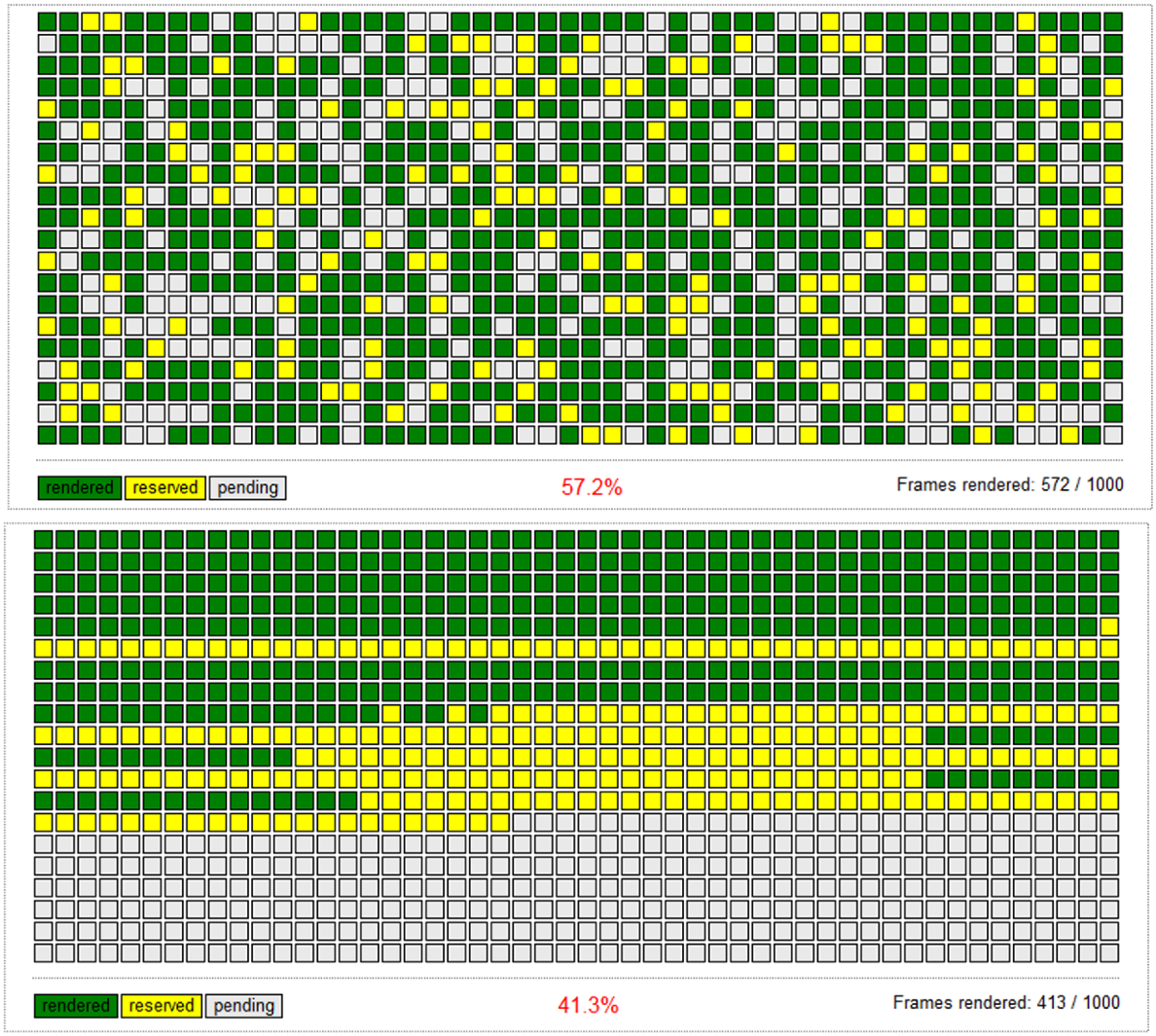
[11] StatOWL: Report analyzing Adobe Flash Player support and Flash usage statistics. http://www.statowl.com/flash.php, květen 2013.

[12] Steve Jobs, A. I.: Thoughts on Flash. http://www.apple.com/hotnews/thoughts-on-flash/, květen 2010.

[13] Wolff, D.: *OpenGL 4.0 Shading Language Cookbook: Over 60 Highly Focused, Practical Recipes to Maximize Your Use of the OpenGL Shading Language*. Packt Publishing, Limited, 2011, ISBN 9781849514767.
URL http://books.google.cz/books?id=Zd2fwnMDvP0C

Příloha A

Porovnání distribučních strategií



Obrázek A.1: Porovnání shodné renderované scény čtyřmi uzly. Výše je použita náhodná strategie, níže strategie sekvenční.

Příloha B

Ovládání aplikace

Pro úspěšné zprovoznění aplikace je nutné disponovat instalací serveru `Apache 2.4` a vyšším s přidanou podporou pro `PHP 5.4.4` a vyšší[1]. Dále je nutné disponovat přístupným serverem `MySQL 5.5` a vyšším s nutností importovat strukturu modelované databáze ze souboru `server/database.sql`[2]. Pravděpodobně bude nutné změnit konfigurační údaje pro připojení k takto vytvořené databázi v souboru `server/libs/Common.php`. Doporučenou a zároveň otestovanou platformou pro běh aplikace je **Windows 7 64b**.

Pro řádný běh této náročné[3] serverové aplikace je nutné v nastavení PHP (php.ini) **zvýšit** hodnoty konfiguračních direktiv `max_input_time`, `max_execution_time`, `post_max_size`, `upload_max_filesize` a **především konfigurační direktivy** `memory_limit` **na ideálně neomezenou hodnotu (-1)**.

Z podporovaných prohlížečů je, vzhledem k vykonaným testům v kapitole 5.0.1, doporučeno použití prohlížeče `Firefox` ve verzi **20** a vyšší.

Vstupní bod aplikace je soubor `index.htm` v kořenovém adresáři. Jedná se o rozhraní k celé implementované funkcionalitě. První krok, `Upload scene .DAE` je určen pro **nahrání a spuštění konverze** `COLLADA` scény. Jakmile je tento krok úspěšně dokončen, jsou vygenerovány odkazy pro **zapojení do distribuovaného renderingu** (znázorněno na B.1), sledování aktuálního **průběhu renderování** a pro spuštění **testovacího nástroje** (viz dále). Tyto odkazy jsou rovněž přístupné přes vstupní bod aplikace. Navíc jsou tam však k dispozici dva pomocné nástroje. První je odlehčený databázový správce `Adminer`[4] a druhý pomocný skript pro úklid všech nahraných scén, snímků apod.

Doporučuji **nespouštět** více instancí vykreslování v jednom prohlížeči současně. Objevuje se bug prohlížečů, kdy snímek z plátna jedné instance problikává v plátně instance jiné.

B.1 Testovací nástroj

Bylo zmíněno, že kromě zapojení uzlů do distribuovaného výpočtu, lze provádět testování klientského renderingu za pomoci vyvíjeného testovacího nástroje. Pomocí něj je možné krokovat animace ve scéně, zkoušet různé rychlosti přehrávání (a tím taky hranice konkrétního výpočetního stroje), ovládat pohled kamery pomocí myši nebo zobrazit scénu jako

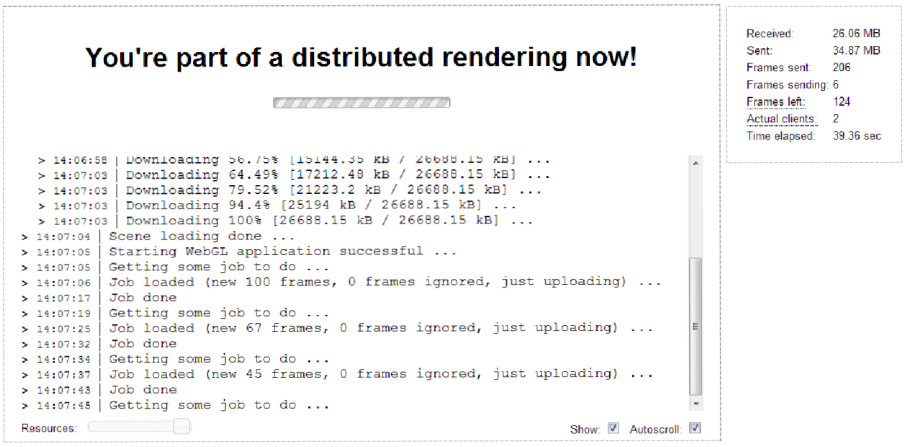
[1] Z důvodu používání mnohých vylepšení jazyka: http://php.net/manual/en/migration54.new-features.php

[2] Pro přehled struktury CD viz README.txt v kořenovém adresáři média.

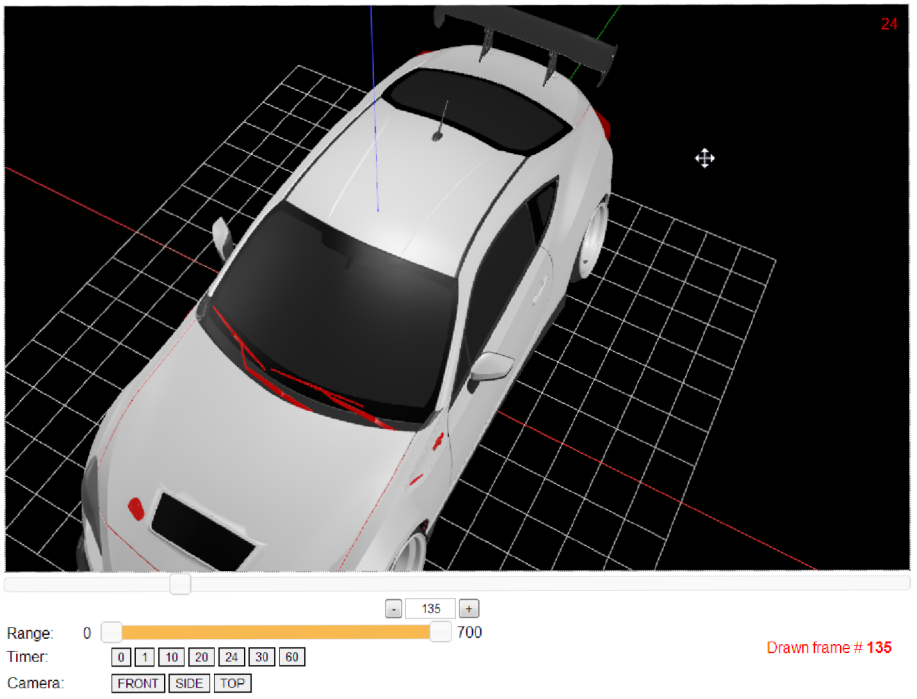
[3] Nutno podotknout příliš náročné kvůli využití nevhodného rozšíření SimpleXML.

[4] Autorem tohoto povedeného jednosouborového nástroje je Jakub Vrána.

půdorys, bokorys či nárys. Tyto pomocná zobrazení jsou určena pro ladění především statických scén. Zároveň je v tomto nástroji do scény přidána generovaná mřížková podlaha, barevně odlišené osy a barevné koule odpovídající polohou a barvou světlům scény. Znázornění práce v tomto nástroji je uvedeno na obrázku B.2.



Obrázek B.1: Ukázka připojeného klienta.



Obrázek B.2: Práce v testovacím prostředí.