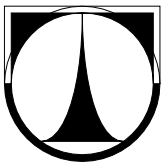
TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií



DIPLOMOVÁ PRÁCE

Liberec 2012 **Bc. Jiří Pánek**

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Vykreslování grafického formátu Collada pomocí WebGL

Rendering Collada graphics format using WebGL

Diplomová práce

Autor: **Bc. Jiří Pánek**
Vedoucí práce: Ing. Jiří Jeníček, Ph.D.

V Liberci 10. 1. 2012

Originální zadání

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé diplomové práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum: 10. 1. 2012

Podpis:

Poděkování

Děkuji zejména Ing. Jiřímu Jeníčkovi, Ph.D. za ochotu, trpělivost a poskytnuté rady během konzultací při tvorbě této práce, dále děkuji Ing. Janu Holubovi za rozšíření znalostí v oblasti programování webových aplikací. Děkuji také rodině za podporu při studiu.

Abstrakt

Cílem práce je prozkoumat možnosti technologie WebGL ve spojení s grafickým formátem Collada a na základě zjištěných informací vyvinout aplikaci pracující s oběma zmíněnými technologiemi - zobrazovač 3D modelů Collada ve webové stránce za pomoci rozhraní WebGL. Formát Collada poskytuje velmi obsáhlý datový kontejner pro ukládání nejen geometrických dat, ale i dalších grafických, fyzikálních informací, týkajících se zobrazení scény. Technologie WebGL vychází z OpenGL ES 2.0, poskytuje rozhraní pro vykreslování 3D grafiky v kompatibilním webovém prohlížeči s využitím grafického hardware a bez nutnosti instalace jakýchkoli doplňků (pluginů). Práce je rozdělena na teoretickou a praktickou část. V teoretické části jsou rozebrány technologie WebGL, Collada a související prostředky, potřebné pro následnou tvorbu aplikace, jmenovitě jazyk HTML 5, poskytující zobrazovací element canvas, JavaScript, jako programovací jazyk WebGL, programovatelné shadery pro zpracování scény na grafickém hardware. Důležitým prvkem je také rozbor prohlížečů v souvislosti s podporou WebGL. Teoretickou část uzavírá test již existujících řešení pro zobrazení 3D modelů ve WebGL. Praktická část se zabývá tvorbou JavaScriptového parseru pro získání potřebných dat z Collada modelu a jejich následným zobrazením v HTML 5 elementu canvas s využitím WebGL. Výsledkem práce je aplikace zobrazující vybrané modely formátu Collada ve webové stránce pomocí rozhraní WebGL s možností jednoduchého ovládání pomocí myši.

Klíčová slova: WebGL, Collada, Canvas, Shader, model, 3D grafika

Abstract

The thesis is focused on research of WebGL technology in conjuction with Collada graphic format and developing an aplication, based on research, with use of both technologies – 3D Collada model viewer in a web page using WebGL. Collada format provides comprehensive data storage for geometric and other graphic even physical information data, related with scene. WebGL technology is based on OpenGL ES 2.0 and provides plugin free interface for rendering 3D graphics in compatible web browser, with use of graphic hardware. Thesis consists of theoretical and practical part. In theoretical part are described WebGL, Collada in connection with other related technologies, namely HTML 5 language providing rendering canvas element, WebGL programming language JavaScript, programmable shaders used for cooperation with graphic hardware for rendering the scene. Important part is analysis of web browsers and their compatibility with WebGL. Teoretical part is concluded with testing already existing solutions for rendering 3D models in WebGL. Practical part deals with development of JavaScript parser for extracting required data from Collada model and using them within WebGL interface for rendering on the canvas element. The result is an aplication, rendering specific Collada models in a web page using WebGL interface with basic mouse control.

Keywords: WebGL, Collada, Canvas, Shader, model, 3D graphics

Obsah

Seznam symbolů, zkratek a termínů

HTML	Hyper Text Markup Language – značkovací jazyk pro webové stránky
XML	Extensible Markup Language – rozšiřitelný značkovací jazyk
XHTML	Extetensible Hyper Text Markup Language – rozšiřitelný hypertextový značkovací jazyk
PHP	Hypertext Preprocessor – hypertextový preprocesor
DOM	Document Object Model – objektový model dokumentu
Plugin	Zásuvný modul
GLSL	OpenGL Shading Language – programovací jazyk pro psaní shaderů
CPU	Central Processing Unit – centrální procesorová jednotka
GPU	Graphic Processing Unit – procesor na grafické kartě
CSS	Cascading Style Sheets – kaskádové styly
W3C	World Wide Web Consortium – konsorcium vyvíjející webové standardy
API	Application Programming Interface – rozhraní pro programování aplikací
URL	Uniform Resource Locator – jednoznačné určení zdroje
HTTP	Hyper Text Transfer Protocol – internetový protokol pro výměnu hypertextových dokumentů
UI	User Interface – uživatelské rozhraní
WHATWG	Web Hypertext Aplication Technology Working Group – komunita zabývající se vývojem webových standardů

1 Úvod

V současné době jsou webové stránky, kromě poskytování informativního obsahu, často využívány také pro prezentaci interaktivního grafického obsahu. Zvyšují se nároky na technologie související s vývojem webových aplikací, což vede k vylepšování stávajících a vzniku nových standardů a technologií. Jednou z těchto technologií je WebGL.

WebGL je JavaScriptové rozhraní umožňující vývoj interaktivních 3D aplikací s využitím grafického hardware, optimalizovaného pro zpracování náročných grafických výpočtů. WebGL je zatím jediná možnost pro tvorbu webové 3D aplikace bez nutnosti využití pluginů a instalace dalšího software. WebGL, vyvíjené společností Khronos group, úzce souvisí s novým prvkem standardu HTML 5 označeným značkou <canvas>. Canvas je v podstatě vykreslovací plátno umístěné do webové stránky, které umožňuje přístup k metodám a vlastnostem WebGL. Protože se jedná o relativně novou, stále vyvíjenou technologii, není k dispozici mnoho informací o jejích možnostech a stavu.

S problematikou 3D grafiky souvisí také pojem Collada. Collada je otevřený standard definující XML schéma pro souborový formát určený k ukládání a snadný přenos informací o 3D scéně. Jedná se o komplexní kontejner, který je schopen přenášet nejen geometrická data, ale také shadery, fyziku, animaci a další vlastnosti související se scénou.

Úkolem první části této práce je seznámení s problematikou vykreslování 3D scén v prostředí WebGL a popis možností zobrazení 3D modelů Collada s využitím tohoto prostředí. Znalosti z teoretické části budou následně využity k návrhu a realizaci vlastní knihovny pro vykreslení modelů formátu Collada pomocí WebGL v různých prohlížečích.

2 Teoretická část

Teoretická část se zabývá popisem základních komponent celého projektu. Kromě dvou hlavních technologií – rozhraní WebGL a modelů Collada s výsledným projektem souvisí řada dalších pojmů a nástrojů. WebGL, tedy webová grafická knihovna, je JavaScriptové rozhraní pro zobrazování interaktivní 3D grafiky v prohlížeči. Propojovacím prvkem mezi JavaScriptovým WebGL a samotnou webovou stránkou je nový HTML5 element canvas. Díky WebGL je možné pro vykreslování využívat grafického hardware. Spolupráce s grafickým hardware probíhá pomocí programovatelných shaderů, psaných přímo v HMTL kódu aplikace v shaderovacím jazyce GLSL. Jejich kód je vykonáván právě na GPU. Některé ze zmíněných pojmů budou v teoretické části probrány podrobněji.

2.1 HTML

HTML je jazyk pro strukturování a prezentaci obsahu World Wide Web.
HTML (Hyper Text Markup Language) je spolu s CSS (Cascading Style Sheets) základní technologií pro tvorbu webových stránek. Je vyvíjena skupinou WHATWG od roku 2004 a od roku 2007 na vývoji podílí i W3C HTML Working Group.

2.1.1 WHATWG

HTML je technologie využívaná pro tvorbu webových stránek vyvíjená společností WHATWG (Web Hypertext Aplication Technology Working Group) – stále se rozvíjející komunitou zabývající se vývojem webu, jejíž primární projekt je HTML (Hyper Text Markup Language). WHATWG byla založena několika zaměstnanci ze společností Apple, Mozilla Foundation a Opera Software v roce 2004 po W3C konferenci. Právě tyto společnosti byly znepokojeny nedostatečným zájmem a směrem, kterým se ubírala W3C, co se týče XHTML, HTML a potřeb autorů, což vyústilo ve vznik WHATWG. Hlavním aspektem vývoje WHATWG je právě HTML.

Poslední verzí specifikací HTML je tzv. žijící standard (living standard) HTML 5. Žijící standard, tedy standard, který je neustále updatován a rozvíjen, podle zpětné vazby web designérů, vývojařů prohlížečů a jiných nástrojů atd. Stále jsou přidávány nové funkce, vlastnosti a nástroje, přičemž podléhají neustálému testování a jsou přizpůsobovány potřebám prohlížečů. Ve specifikaci jsou odlišeny stabilní a testované části.

2.1.2 HTML 5

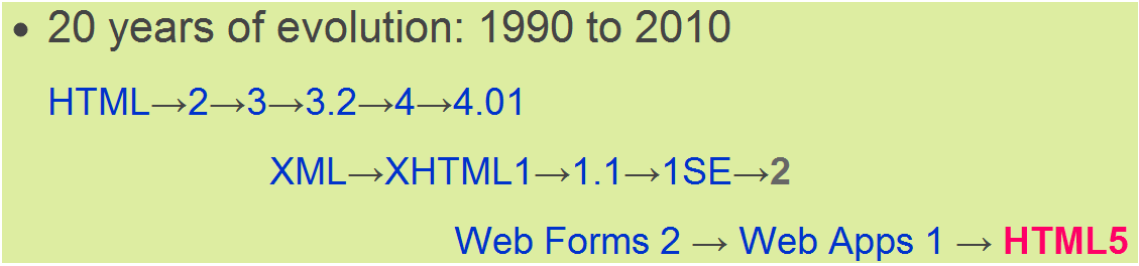
HTML 5 je novou verzí HTML 4, XHTML 1 a DOM (Document Object Model) Level 2. HTML 5 definuje jazyk, spojující HTML a XML (který může být psaný v HTML i XML), snaží se vylepšit předešlé iterace HTML a pokrýt potřeby nového aspektu Web Applications (Webových Aplikací).

Mezi hlavní cíle patří vývoj nových komponent založených na technologiích HTML, CSS, DOM a JavaScript, lepší řešení chyb, snaha o nezávislost HTML5 na zobrazovacím zařízení a velký prostor je věnován spolupráci, zpětné vazbě a přístupnosti výrobního procesu veřejnosti. Obecně se snaží odstranit potřebu externích pluginů, například pro implementaci multimédií (Flash, Java). Přináší nové možnosti formulářů, jádro Open Web App platformy, nové značky sémanticky definující strukturu stránky, úložiště formou asociativního pole, podporu relačních databází a v neposlední řadě nativní podporu multimédií. K nejzajímavějším aspektům HTML 5 patří implementace podpory audia a videa, větší podpora pro lokální ukládání a offline aplikace, nové elementy pro zpřehlednění strukturování stránky (header, article, atp.). Důraz je kladen na zkrácené a rychlejší zápisy, jednoduchost a účinnost. Specifikace HTML 5 je stále upravována, nelze tedy stanovit finální konkrétní verzi. Některé části jsou stabilnější a podporované většinou prohlížečů, jiné jsou více experimentální a na jejich vývoji se stále pracuje. WHATWG HTML standard se skládá z podmnožiny obsahující pouze specifický HTML materiál a W3C HTML 5 specifikace je taktéž podmnožinou WHATWG HTML obsahující pouze stabilnější prvky. Více detailů o specifikacích a jejich prvcích na v tabulce Tab. 1.

	WHATWG Specifications (and sections therein)	Section links for Web Applications 1.0	W3C/IETF Specifications
HTML5 only (excluding newer features)	n/a	n/a	Single-page, multi-page (HTML WG)
HTML (including newer features)	WHATWG HTML	Everything not listed below!	
Microdata	In WHATWG HTML	Microdata	Microdata (HTML WG)
Canvas 2D Context	In WHATWG HTML	2D Context	2D Context (HTML WG)
Communications - Cross-document messaging	In WHATWG HTML	Cross-document messaging	HTML5 Web Messaging (HTML WG)
Communications - Channel messaging	In WHATWG HTML	Channel messaging	
Web Workers	only in WA1	Web Workers	Web Workers (WebApps WG)
Web Storage	only in WA1	Web Storage	Web Storage (WebApps WG)
Web Sockets API	only in WA1	Web Sockets API	Web Sockets API (WebApps WG)
Server-Sent Events	only in WA1	Server-sent Events	Server-sent Events (WebApps WG)
WebVTT	In WHATWG HTML and informally as WebVTT	WebVTT	
WebRTC	Informally as WebRTC	WebRTC	

Tab. 1: Specifikace HTML 5 (převzato z [23])

HTML reprezentuje strukturu stránky, CSS visuální rozvržení. Spolu grafikou a skriptováním je HTML a CSS základem pro tvorbu webových stránek a aplikací. HTML je jazyk pro popis struktury webových stránek. XHTML je odvozená varianta HTML, využívající syntax XML (Extensible Markup Language). Obsahuje stejné elementy jako HTML, ale syntaxe se lehce odlišuje. Protože XHTML je XML aplikace, je možné využívat další XML nástroje jako XSLT – jazyk zpracovávající XML obsah. XHTML využívá XML parser a HTML má svůj vlastní parser, který se více soustředí na zpracování HTML prohlížeči.



Graf 1: Vývoj HTML (převzato z [24])

Kompletní specifikaci je možné nalézt na webových stránkách [7], [8].

2.1.3 Canvas

Jedním z nových prvků HTML 5 je značka (tag) Canvas, podporovaná v některých novějších verzích prohlížečů a specializovaných vývojářských verzích (Mozilla Firefox, Google Chrome, Opera, Safari). Canvas je velmi podobný značce <img>, s tím rozdílem, že neobsahuje atributy src a alt. Je to plátno zasazené do webové stránky a umožňuje vykreslování grafiky do jeho těla pomocí JavaScriptu. Je možné jej využít například pro vykreslování grafů, herní grafiky a jiné zobrazování přímo za běhu. Element Canvas se zařadí do webové stránky, konkrétně do části jejího těla <body>, pomocí tagu <canvas>. Tělo Canvasu je definováno šířkou (width) a délkou (height) v pixelech. Canvas je relativně nový element a v některých prohlížečích není implementován, nebo není kompletně funkční. Dovnitř značky je tedy možné napsat text – náhradní obsah, který se zobrazí pouze, když prohlížeč Canvas nepodporuje, přičemž ignoruje grafický obsah kontejneru. Naopak prohlížeče podporující canvas ignorují náhradní obsah a canvas vyrenderují. Většinou se náhradní obsah používá pro

upozornění v podobě textu, že daná stránka nepodporuje WebGL, popřípadě jinou reprezentaci zobrazovaných dat (např. číselným vyjádřením namísto grafu).

Implementace značky canvas s použitím náhradního obsahu:

```
1 <canvas id="tutorial" width="150" height="150">
2   Textovy nahradni obsah
3 </canvas>
4
5 <canvas id="clock" width="150" height="150">
6   <img src="images/clock.png" width="150" height="150" alt=""/>
7 </canvas>
```

Poslední důležitý atribut je identifikátor ID, s jehož pomocí se lze v JavaScriptu snadno na Canvas odkazovat.

Značka <canvas> vytvoří plátno pro kreslení se specifikovanými rozměry a zpřístupní jeden nebo více tzv. contextů s odlišnými API (Application Programming Interface) - objektů používaných pro vytváření a manipulaci zobrazovaného obsahu. Objekt context již obsahuje vlastní funkce, vlastnosti pro kreslení a poskytuje přístup k vykreslovacímu API. Z typů contextů lze zmínit základní 2D context (značený „2d“), další typy poskytují rozlišné druhy renderování. K nejdůležitějším patří 3D context označený „experimental-webgl“ založený na OpenGL ES.

Canvas je po inicializaci prázdný. Pro zobrazení je třeba pomocí skriptu získat přístup k renderovacímu contexu a něco vykreslit. Element Canvas má DOM (Document Object Model) metodu zvanou getContext využívanou k získání zmíněného contextu a jeho vykreslovacím funkcím. Metoda getContext má pouze jeden parametr – typ contextu.

Získání contextu:

```
1 var canvas = document.getElementById('tutorial');
2 var context = canvas.getContext('2d');
```

V prvním řádku získáme DOM uzel Canvasu pomocí metody getElementById, následně je možné zpřístupnit context pomocí metody getContext.

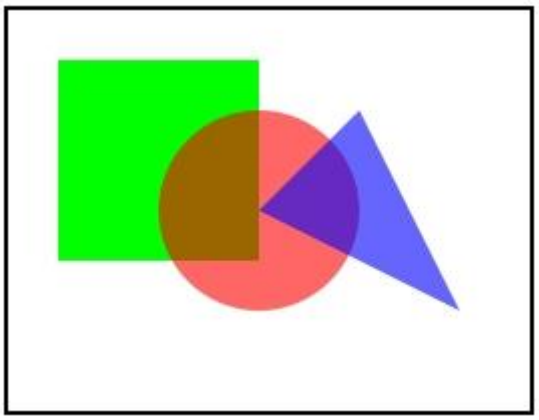
Podporu prohlížeče lze zobrazit pomocí náhradního obsahu v tagu, ale také při vykonávání skriptu pomocí metody getContext jak je znázorněno v následujícím příkladě.

```
1 var canvas = document.getElementById('tutorial');
2 if (canvas.getContext){
3   var context = canvas.getContext('2d');
4   // kód použitý pro vykreslení
5 } else {
6   // kód použitý pokud canvas není podporován
7 }
```

Po inicializaci contextu je již možné začít kreslit pomocí JavaSriptu na canvas. Vykreslování se liší podle typu použitého contextu. Mezi základní metody pro tvorbu jednoduchých obrazců na 2D contextu patří například kreslení obdélníků (metody context.fillRect, context.strokeRect, context.clearRect), kreslení libovolných tvarů neboli cest označovaných jako paths (metody context.beginPath, context.moveTo, context.lineTo, context.fill, context.stroke, context.closePath). Vykreslování těchto tvarů předchází nastavení barvy výplně, obrysové čáry a tloušťky obrysové čáry (context.fillStyle, context.strokeStyle, context.lineWidth). Je možné vkládat hotové obrázky pomocí metody context.drawImage. Dále lze zmínit operace s pixely a poli pixelů, vykreslování textu, využívání stínů, gradientů, transformace (posun, rotace, škálování), animace atd. Nejdůležitější funkce canvasu pro tuto diplomovou práci je však možnost spolupráce s WebGL pro tvorbu hardwarově akcelerované 3D grafiky na webových stránkách.

Příklad použití 2D canvasu ve webové stránce – kód a zobrazený výsledek:

```
01 <html>
02   <head>
03     <title>Canvas tutorial</title>
04     <script type="text/javascript">
05       function draw(){
06         var canvas = document.getElementById('tutorial');
07         if (canvas.getContext){
08           var context = canvas.getContext('2d');
09         }
10       }
11     </script>
12     <style type="text/css">
13       canvas { border: 1px solid black; }
14     </style>
15   </head>
16   <body onload="draw();">
17     <canvas id="tutorial" width="150" height="150"></canvas>
18   </body>
19 </html>
```



Obr. 1: Zobrazení jednoduchých tvarů na 2D canvas

2D context reprezentuje kartézský povrch s počátkem (0,0) umístěném v levém horním rohu s hodnotami souřadnic zvyšujícími se směrem dolů a doprava. Context obsahuje metody save(); a restore(); pro uložení a opětovné vyvolání aktuálního obsahu canvasu. 3D context je již přímo spojován s WebGL a má svou vlastní specifikaci, které se podrobně věnuje kapitola WebGL.

2.2 Prohlížeče

HTML 5 zatím není oficiální standard a žádný prohlížeč nedosahuje jeho plné podpory. Všechny funkce HTML 5 nejsou podporovány ve všech prohlížečích. Aplikace, které bez problémů fungují v jednom prohlížeči, v jiném fungovat nemusí. Většina majoritních prohlížečů (Chrome, Firefox, Opera, Internet Explorer, Safari) však stále pokračuje v začleňování nových vlastností HTML 5. Společnosti Apple, Google, Mozilla a Opera jsou také součástí pracovní skupiny Khronos Group, zpravující WebGL. Jednodušší 2D canvas podporuje většina prohlížečů.

Podpora 2D canvasu:

- Internet Explorer 9
- Mozilla Firefox 3.0+
- Google Chrome 3.0+
- Opera 10.0+
- Safari 3.0+
- iPhone 1.0+
- Andriod 1.0+

Oproti tomu u experimentálního 3D canvasu, tedy WebGL, stále ještě není spolehlivá funkčnost a podpora zajištěna. Některé typy prohlížečů nepodporují 3D canvas vůbec a ty, které ho podporují, mohou mít odlišné chování při jeho zpracování. Ze začátku se 3D canvasu s parametrem „experimental-webgl“ věnovaly pouze specializované vývojářské a experimentální verze nejznámějších prohlížečů. Tyto verze často potřebovaly spoustu manuálních úkonů pro umožnění funkce. Průkopníkem na tomto poli byl Google Chromium Canary build, který jako první zpřístupnil vývojářskou verzi s podporou 3D contextu. Záhy se přidal i Mozilla Firefox (nightly build). Po vydání WebGL 1.0 specifikace se poslední verze několika prohlížečů blížily plné shodě s touto technologií.

Google Chrome, Chromium

Prohlížeč je postavený na jádře WebKit, které první podporovalo WebGL. Podle posledního testování na většině platforem prochází 97 % testů a více. Pro získání contextu se stále používá metoda getContext("experimental-webgl"), stabilní getContext("webgl") ještě není povolena. První verze s podporou WebGL byla experimentální, označena 8.0. Experimentální verze jsou označené jako beta, dev a canary. Canary je automaticky vytvářená z posledního vydání otcovského Chromium projektu a není testována pro přímé vydání. V současné době je asi nespolehlivější, svědčí o tom i podpora ve stabilních verzích, počínaje Chrome 9.0.

Mozilla Firefox

Stabilní podpora WebGL je zabudována do prohlížečů Firefox od verze 4.0 na všech platformách s kompatibilními grafickými kartami a aktualizovanými ovladači. Předtím byla zavedena ve vývojářských verzích Firefoxu označených jako tzv. „nightly builds“, popřípadě velmi často updatovaná vývojařská verze označená jako Minefield. Pro testování/debugování v těchto prvních verzích bylo možné využít softwarového renderování za pomoci knihovny OSMesa. V konfiguraci prohlížeče bylo nutné povolit proměnnou "webgl.osmesalib" a nastavit cestu ke knihovně OSMesa.

Opera

Ačkoli se první pokusy Opera Software s WebGL datují již od roku 2007, na vývoji verze s podporou WebGL se začalo pracovat až v roce 2009, kdy začal standardizační proces WebGL. První verze Opery podporující WebGL je až vývojářská verze Opera 11.50. Je určena pouze pro operační systémy Windows, oproti ostatním konkurentům není příliš stabilní a funguje správně, pouze pokud grafická karta na počítači podporuje OpenGL 2.0. Stabilní verze prohlížeče zatím WebGL nepodporují.

Safari

WebGL bylo též podporované na strojích s operačním systémem Mac OS X 10.6 díky prohlížeči založeném na jádře WebKit – Safari WebKit nightly build. Po stažení a nainstalování prohlížeče je třeba v terminálu napsat následující instrukci:

defaults write com.apple.Safari WebKitWebGLEnabled -bool YES

Po zadání toho příkazu se budou již všechny následující verze prohlížeče spouštět s podporou WebGL. Další verze Safari od 5.1 mají podporu WebGL, ale také není v základním nastavení povolena.

Internet Explorer

Nabízí plnou hardwarovou akceleraci na systémech Windows Vista a Windows 7 díky technologii Direct X avšak WebGL v současné době nepodporuje a do budoucna s ním zatím nepočítá. Jsou zde alternativy v podobě pluginů, které WebGL v prohlížečích Internet Explorer umožňují (Chrome Frame, IEWebGL).

When can I use WebGL? View in interactive mode

Compatibility table for support of WebGL in desktop and mobile browsers.

= Supported = Not supported = Partially supported = Support unknown

WebGL - 3D Canvas graphics - **Other**

Method of generating dynamic 3D graphics using JavaScript, accelerated through hardware

Resources: Instructions on enabling WebGL Tutorial Firefox blog post Webkit blog post Opera blog post (not WebGL)

Global user stats*:
Support: 0%
Partial support: 36.7%
Total: 36.7%

	IE	Firefox	Safari	Chrome	Opera	iOS Safari	Opera Mini	Opera Mobile	Android Browser	
9 versions back				4.0						
8 versions back				5.0						
7 versions back				6.0	9.0					
6 versions back				7.0	9.5-9.6					
5 versions back		2.0		8.0	10.0-10.1					
4 versions back	5.5	3.0	3.1	9.0	10.5					
3 versions back	6.0	3.5	3.2	10.0	10.6					
2 versions back	7.0	3.6	4.0	11.0	11.0	3.2		10.0	2.1	
Previous version	8.0	4.0	5.0	12.0	11.1	4.0-4.1		11.0	2.2	
Current	9.0	5.0	5.1	13.0	11.5	4.2-4.3	5.0-6.0	11.1	2.3	3.0
Near future	9.0	6.0	5.1	14.0	12.0					
Farther future	10.0	7.0	6.0	15.0	12.1					

Note: All support is currently listed as "partial" because not all users with these browsers have WebGL access. This is due to the additional requirement for users to have up to date video drivers. Note that WebGL is part of the Khronos Group, not the W3C. Support in Opera 12 is based on support in experimental builds and is not certain to be included.

Tab. 2: Tabulka s označením podpory WebGL v prohlížečích (převzato z [16])

Pro jednoduché ověření podpory technologie WebGL v prohlížeči je možné využít webových stránek, zobrazujících podporu WebGL případně i dalších parametrů, týkajících se této problematiky. Příkladem takových stránek jsou například [17], [18]. Případně je možné funkci vyzkoušet přímo na nějaké WebGL aplikaci.

V současné době je tedy možné WebGL zprovoznit ve všech známějších prohlížečích v podobě stabilních nebo vývojářských verzí, a to na operačních systémech

Microsoft Windows, Linux i Mac OS. Důležitou roli ve funkci WebGL také hraje grafická karta a její ovladače. Grafická karta by měla být kompatibilní s technologií OpenGL 2.0, neměla by být uvedena na zveřejněném seznamu „blacklistu“ a ovladače by měly být nejnovější možné. Seznamy blokovaných karet lze najít na internetu podle výrobce prohlížeče, nebo na stránkách Khronos Group [3].

Parametr 3D contextu je stále „experimental-webgl“, v budoucnu při zajištění stoprocentní funkce se změní na „webgl“.

```
gl = canvas.getContext("experimental-webgl");
```

2.3 JavaScript

JavaScript je implementací standardu ECMAScript language a je primárně využíván ve webovém prostředí, což z něj zároveň dělá i ideální jazyk pro programování ve WebGL. Je to objektově orientovaný jazyk, nevyužívá třídy, ale klíčové slovo prototype, pomocí nějž jsou deklarovány objekty, které se mohou dále klonovat. Jedná se netypový jazyk, nemá definovány primitivní datové typy, všechno je považováno za objekt a deklarováno klíčovým slovem var. Vše je možné uložit. Je tedy samozřejmě možné využívat základní typy jako integer, float, char, string, boolean, array a objekt, ale není nutné definovat typ. Funkce jsou taktéž považovány za objekty, mohou být vytvářeny a modifikovány během vykonávání programu stejně jako objekt. JavaScript je interpretovaný jazyk, jeho kód je interpretován řádek po řádku, není kompilován.

Do webové stránky se JavaScript píše přímo, nebo může být skript vložen jako samostatný soubor např.:

```
<script type="text/javascript" src="Collada.js">
```

Debugování JavaScriptu je poměrně komplikované téma, neexistuje mnoho volně šiřitelných editorů s přímou podporou debugování. Pro práci s JavaScriptem byl využit editor Aptana Studio 2.0 založený na vývojovém prostředí Eclipse, který nabízí debugovací režim v podobně pluginu do prohlížečů Mozilla Firefox. Pro účely této práce je však z důvodu lepší podpory WebGL využíván prohlížeč Google Chromium, který nabízí podporu vestavěného debuggeru, javascriptové konzole, využívání breakpointů a dalších nástrojů. Debugger je možné vyvolat stiskem kláves

CTRL+SHIFT+I. Důležité proměnné, objekty lze vypisovat do přímo do konzole, chybný kód je při spuštění označen, případně popsán. Je důležité mít na paměti, že JavaScript je interpretovaný jazyk, debugovací režim tedy nalezne pouze chybný kód, který je vykonáván. Další neobjevené chyby se mohou vyskytovat v nevykonaném kódu.

2.4 Úvod do PC grafiky

V následujícím textu je vysvětleno několik základních pojmů z grafického prostředí.

Obraz – soubor pixelů (řady čtvercových teček), kde každý pixel má svou vlastní barvu, zobrazované v 2D dimenzi.

Model – je soubor dat uzpůsobených pro jednoduché vytvoření obrázku nebo sady obrázků. Zatím co obrázky obsahují jen barvu, modely mají většinou informace navíc – např. hloubka.

Kamera – bod pohledu, ze kterého je scéna renderována, využívá různé typy projekce a pole pohledu (field of view).

GPU – graphics processing unit (grafický procesor), počítačové zařízení zodpovědné za renderování v reálném čase, přístupná přes OpenGL a Direct3D, současné GPU jsou optimalizované pro resterizaci a neumí renderovat metodami ray-tracing a radiozitou (musí se zpracovat na CPU).

Většina obrázků je 2D a většina modelů je 3D.

Renderování

Renderování je proces převodu modelu na vykreslený obraz. Renderovacích technik je mnoho druhů, mají své výhody a nevýhody, které je brát třeba v potaz při použití v konkrétním případě. Tři základní techniky renderování jsou rasterizace (převod modelu na výstupní zařízení – rastrové, vekorové), ray-tracing (stopování paprsků odrážených modelem) a radiozita (metoda šíření světelné energie).

Animace

Animace je proces zobrazování série statických obrázků z modelu v rychlém sledu po sobě tak, že výsledek se jeví jako pohyblivý.

- Frame - 2D obraz obvykle generovaný z 3D scény (lze generovat i z 2D scény)
- Framerate - míra rychlosti zobrazovaných obrazů po sobě při zobrazování animace, ideálně by měla dosáhnout obnovovací frekvence monitoru (obvykle 60 framů/s).
- Double buffering - zpracování (renderování) jednoho framu v pozadí, zatímco jiný frame je právě zobrazován na monitoru, buffery obsahující framy jsou přehozeny vždy, když buffer pracující v pozadí dokončil renderování.

2.5 WebGL

WebGL je programovací rozhraní (API), které umožňuje zpracování a zobrazení grafiky v reálném čase přímo ve webových stránkách při využití jejího renderování na GPU. WebGL se skládá z následujících součástí:

- OpenGL – soubor funkcí poskytujících rozhraní pro spolupráci s GPU a tvorbu počítačové grafiky v reálném čase, alternativa je technologie Direct3D (výhoda OpenGL je multiplatformnost).
- JavaScript – programovací jazyk původně pro webové stránky, využitý následně i pro WebGL.
- HTML5 – nejnovější verze HTML, poskytující nové funkce a rozšíření (element canvas pro zobrazení WebGL), nové funkce umožňují nahradit funkce Flash bez nutnosti využití jakýchkoli doplňků (addonů).

WebGL je postaveno na OpenGL, konkrétně verzi OpenGL ES 2.0 (ES = embedded systems) původně navržené pro mobilní zařízení. Poskytuje podporu většiny (ne všech) funkcí poslední verze OpenGL. Programy napsané pro WebGL jsou tedy nejen pro web ale i pro standartní desktop aplikace a mobilní zařízení (chytré telefony). Jedná se o multiplatformní webový standard pro přístup ke grafickému hardware, zpracování grafiky vysoké kvality a 3D grafiky ve webovém prohlížeči. Je zpřístupněný přes tzv. webgl context náležící HTML 5 elementu canvas (plátno) a rozhraní DOM (Document Object Model). Jedná se o API založené na shaderech (GLSL) s velmi podobnými konstrukcemi jako u OpenGL ES 2.0. WebGL se celkově velmi blíží OpenGL ES 2.0 specifikaci, s několika rozdíly týkajících se přizpůsobení webové

technologii a požadavkům webových vývojářů (např. přizpůsobení pro práci s JavaScriptem). Hlavní přínos WebGL je možnost pracovat s 3D grafikou přímo ve webovém prohlížeči, bez jakékoliv nutnosti využívání pluginů. Součástí pracovní skupiny WebGL jsou vývojáři prohlížečů jako Mozilla (Firefox), Google (Chrome), Apple (Safari) a Opera.

WebGL je popsáno specifikací 1.0, kterou je možné najít na stránkách Khronos group. Základem je rozbor rozhraní WebGLRenderingContext, které představuje WebGL API. Velké množství funkcí popsaných v této specifikaci obsahuje odkazy na OpenGL ES 2.0 manuály. Je kladen důraz na shodný obsah obou specifikací, v případě nalezení rozporů a nejasností by měla být v úvahu brána spíše specifikace Open GL ES 2.0.

Pro využití WebGL API je nutné nejprve přiřadit objekt WebGL context (WebGLRenderingContext) danému elementu canvas. Pomocí contextu lze přistupovat k WebGL, využívat jeho funkcí a možností k zobrazení scény pomocí vykreslovacího bufferu, který je nutné vytvořit stejně jako context. Vykreslovací buffer je předán HTML stránce před jejím složením, ale jen pokud se při této poslední operaci změnil obsah bufferu.

Objekt WebGLRenderingContext se vytváří voláním metody HTML elementu canvas - getContext(“experimental-webgl”). Po standardizaci se vnitřní řetězec změní na “webgl”. V tomto čase by měl být vytvořen i vykreslovací buffer.

Přiřazení WebGL API elementu Canvas:

```
<!DOCTYPE html>
<html><body>
  <canvas id=”c”></canvas>
  <script type=”text/javascript”>
    var canvas = document.getElementById(“c”);
    var gl = canvas.getContext(“webgl”);
    gl.clearColor(1.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
  </script>
</body></html>
```

V současné době WebGL podporuje další prostředky v podobě DOM objektů, jako textury, buffery, framebuffery, renderbuffery, shadery a programy. Podporované shadery musí vycházet z tzv. OpenGL ES Shading Language (GLSL).

Ve specifikaci WebGL je možné nalézt základy jako vytvoření contextu , práci s vykreslovacím bufferem a nastavení „viewportu“, popis podporovaných prostředků (např. textury, buffery, shadery,…), rozhraní DOM - výčet podporovaných datových typů, atributů funkce getcontext, část dokumentu je věnována bezpečnosti a nejvíce prostoru je věnováno WebGL objektům.

Rozhraní nejdůležitějších objektů	Objekt (vychází z OpenGL)
WebGLBuffer	Objekt bufferu
WebGLProgram	Objekt programu
WebGLRenderbuffer	Objekt renderbufferu
WebGLShader	Objekt shaderu
WebGLTexture	Objekt textuty
WebGLUniformLocation	Lokace uniform proměnné v shader programu

Tab. 3: Popis vybraných objektů WebGL

Data se ve WebGL zprácovávají pomocí bufferů. Buffer reprezentuje nestrukturovaná binární data. S bufferem souvisí tři základní operace:

```
positionBuffer = gl.createBuffer(); //vytvoření bufferu
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer); //přiřazení bufferu
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(c.geomData.vertexes),
gl.STATIC_DRAW); //přiřazení dat bufferu
```

Nastavení pohledu se provádí pomocí funkce viewport(x, y, width, height). Viewport specifikuje transformaci x a y z normalizovaných souřadnic do souřadnic vytvořeného okna. Texturové objekty slouží pro ukládání dat a stavu textur. Pokud není textura přiřazena, WebGL automaticky přiřadí chyby k operacím s texturou asociovaným. Unifrom proměnné jsou hodnoty předávané shaderům jako vertex atributy.

Data k vykreslení se ukládají do vykreslovacího bufferu, který je předán HTML stránce na začátku dalšího procesu skládání stránky. Zapisovat do vykreslovacího bufferu lze dvěma způsoby (metodami):

- drawArrays – vykreslení elementů daného typu postupně z pole vertexů
- drawElements – vykreslení elementů daného typu podle indexů ukazujících do pole vertexů

Elementy jsou například trojúhelníky, body, soubor přímek atd.

Kompletní specifikace WebGL a další související materiály jsou k nalezení na oficiálních stránkách Khronos group [1].

2.6 Collada

COLLADA (Collaborative Design Activity) definuje schéma (formát) založené na XML pro ukládání a přenos interaktivních 3D objektů a animací. Poskytuje možnost zakódování obsáhlých, komplexních vizuálních scén zahrnujících všechny důležité prvky pro moderní 3D aplikace - geometrii, shadery, efekty, ale i fyziku, animaci a kinematiku. Usnadňuje přenos 3D objektů mezi aplikacemi bez ztráty informace. Výhodou je možnost snadného rozšíření a upravení podle potřeby koncového uživatele.

Collada je otevřený standard jehož poslední specifikace je označena jako 1.5 a ve formátu PDF publikována na stránkách vývojářů. Pochází taktéž z dílny Khronos Group, proto je velmi výhodné použití v souvislosti s WebGL a OpenGL 2.0 ES. Na jeho vývoji se podílí velká skupina tvůrců včetně designérů, vývojářů a dalších společností pohybujících se ve světě 3D problematiky (např. Sony Computer Entertainment, která také s vývojem Collady začala). Je využívána předními nástroji pro tvorbu 3D aplikací a modelů jako například 3D Studio Max, Maya atd.

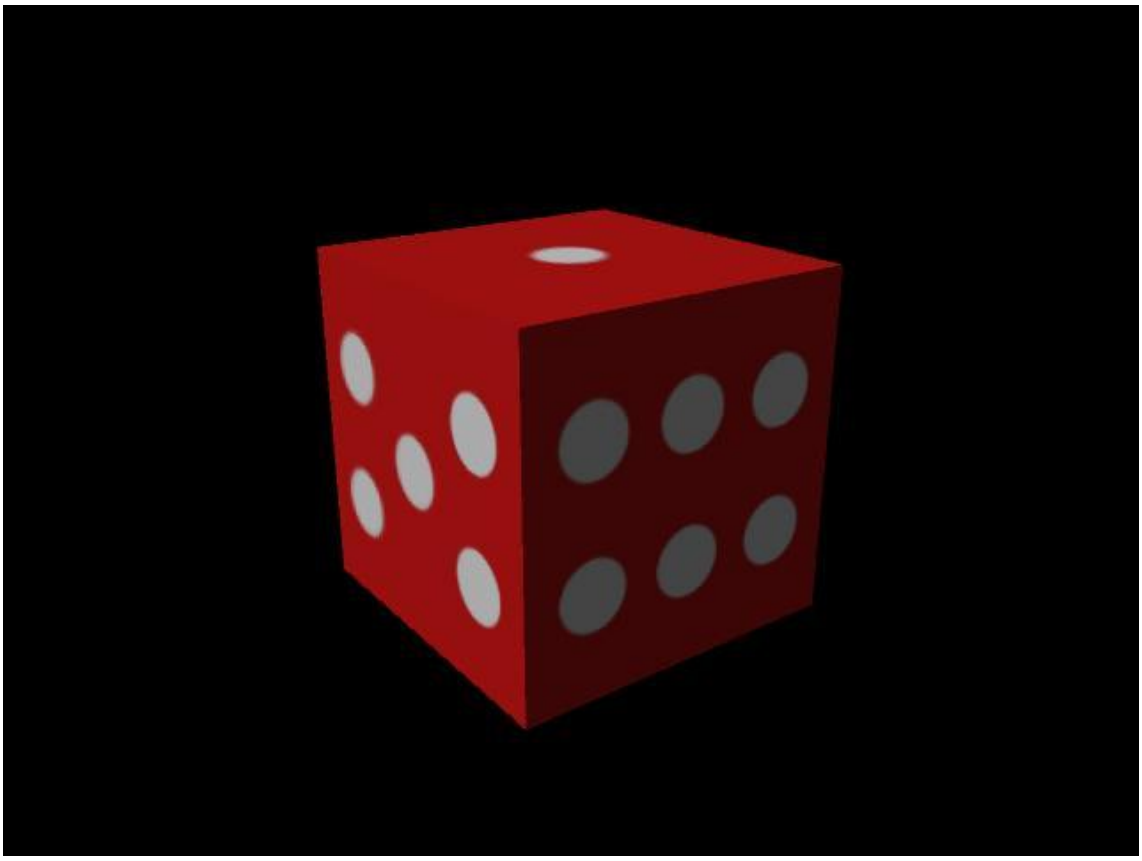
Na stránkách tvůrce – Khronos Group je možné najít důležité materiály jako poslední verzi specifikace, schéma a informace týkající se jednotlivých verzí, nových vlastností, nástrojů a další podpory.

Jak bylo řečeno, Collada je založená na XML schématu, informace se tedy ukládají do stromové struktury uzlů odlišených příslušným názvem (podle nichž už je většinou možné poznat o jakou informaci se jedná) a jejich vnořených potomků. Hlavní uzly věnující se podrobněji konkrétním součástem 3D modelu mají v názvu klíčové slovo library (např. <library_geometries>). K hlavním uzlům patří například geometrie, textury, světlo, materiály, nastavení scény, kamery atd. Kompletní schéma je velmi

komplexní a rozsáhlé, nebude zde tedy podrobně popisováno. Je možné jej najít na stránkách Khronos Group, jak bylo zmíněno výše a detailnější informace o modelu ve formátu Collada jsou v kapitole věnující se parseru, který má za úkol potřebná data z tohoto modelu získat. Parser se zaměřuje pouze na některé důležité uzly, nezpracovává celý model.

Modely Collada jsou využívány například v projektu Second Life – virtuálním 3D světě, do kterého mohou uživatelé přidávat právě Collada modely. Svět Second Life má v dnešní době rozlohu přibližně 2x tak velkou jako Hong Kong, což dokumentuje komplexnost modelů. Aplikace Google Earth využívá také 3D modely v zapouzdřeném Collada formátu .kmz. Na stránkách collada.org existuje banka testovacích modelů Collada (některé byly využity i v této práci), kde je možná sdílet a stahovat testovací modely. Standardní koncovka modelu je .dae.

Na přiložených obrázcích je možné porovnat jednoduchý a složitější Collada model.



Obr. 2: Příklad jednoduchého Collada modelu

Obr. 3: Příklad složitějšího Collada modelu

2.7 Shadery

Shadery jsou programy vykonávané na GPU, slouží primárně k výpočtu vykreslovaných efektů a obrazu pomocí grafického hardware. Jsou využívány pro naprogramování tzv. rendering pipeline, tedy nástroje na GPU, sloužícího pro zřetězené zpracování grafické informace. Naprogramování této zobrazovací pipeline umožňuje nahrazení pipeline s fixní funkcí (Fixed Function Pipeline) a jejích omezených možností (běžné geometrické transformace a pixel-shaderovací funkce) komplexnější funkcí a specifickými efekty. Podrobnější popis grafické pipeline a souvislosti programovatelné pipeline s shadery je uveden v samostatném odstavci níže v textu.

Ve WebGL jsou dva základní typy shaderů – vertex shader (vrcholy zpracovávány vertex procesorem) a fragment shader (fragment procesor). Výstupem vertex shaderu je typicky gl_Position (pozice vertexu, který je právě zpracováván), výstup fragment shaderu bývá gl_FragColor (barva fragmentu, který je zpracováván). Další zpracovávané vlastnosti shaderů jsou například texturové koordináty, průhlednost, odrazivost.

Pixel

- obrazovka je rozdělena na síť pixelů (picture element)
- obvykle mají čtvercový tvar a může jim být přiřazena barva
- 2D obrazy bývají ukládány jako soubor barev (po sobě jdoucích), kde každá reprezentuje právě pixel v pořadí, v jakém jsou pixely v obraze
- pixely mohou mít různý barevný formát, typický formát pro WebGL je RBG nebo RGBA

Fragment

- informace nutné pro vygenerování pixelu jako výstup renderování
- obsahuje barvu, pozici, průhlednost (pokud je dána), hloubku (fragmenty za sebou)
- fragment je na rozdíl od pixelu přístupný a může být ve WebGL měněn
- kromě vypočtené barevné informace pro pixel může fragment nést i další informace, které mohou, ale nemusí být využity pro zobrazení
- z fragmentů jsou tedy tvořeny pixely a typicky obsahují více informace než pixely

2.7.1 Život shaderu

Shader je program napsaný ve svém speciálním jazyce, po jehož kompilaci je možné změnit funkci grafického akcelerátoru, zpracování vrcholů a fragmentů na grafické kartě. Napsaný shader je tedy přeložen do binární podoby přímo v kódu aplikace pomocí použitých knihoven a jejich funkcí. Zjednodušeně se život shaderu dá popsat následovně: kód shaderu je napsaný přímo v aplikaci, kompiluje se pomocí knihoven, při překladu jsou nejprve vytvořeny prázdné objekty pro vertex a fragment

shader s parametry ID a typ objektu. Do těchto prázdných objektů se nahraje zdrojový kód vertex a fragment shaderu. Následně se shadery pomocí funkce compileShader kompilují. Pokud vše proběhne korektně, je vytvořen další objekt s hlavním programem, kterému se přiřadí oba shadery. Výsledný program a jeho objekty (shadery) jsou pomocí instrukce linkProgram slinkovány s grafickým hardware, na kterém bude provádět jejich kód. Nakonec je program nastaven jako aktivní (useProgram) a jeho modifikovaná funkce je použita pro zpracování vertexů a fragmentů. Další informace o postupném zpracování shaderu napsaném v GLSL jsou v kapitole označené GLSL, případně lze prostudovat přiložené přílohy s kódem aplikace.

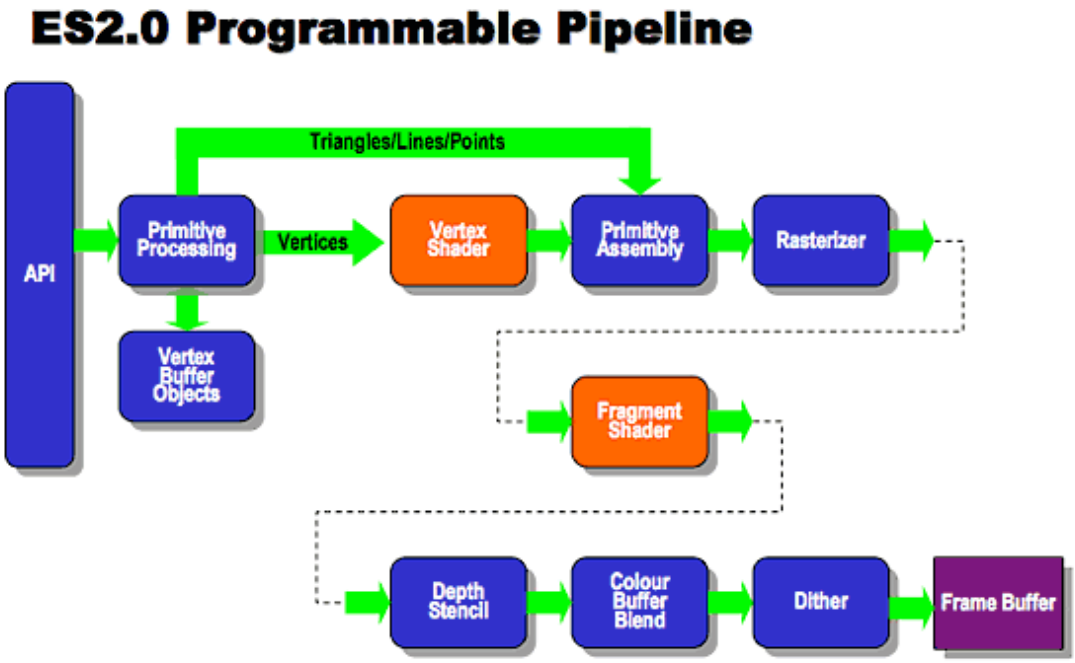
2.7.2 Zjednodušený popis grafické pipeline

Zobrazení 3D scény vyžaduje poměrně náročné a obsáhlé výpočty a operace v reálném čase. Tyto úkony velmi zatěžují CPU, proto se pro jejich zpracování využívá grafický akcelerátor (grafická karta), která je pro tyto výpočty optimalizována. Grafická pipeline je soustava prvků, provádějící postupně operace, jejichž výsledkem je scéna zobrazovaná na monitoru. V následujícím odstavci je zjednodušeně popsána funkce grafické pipeline.

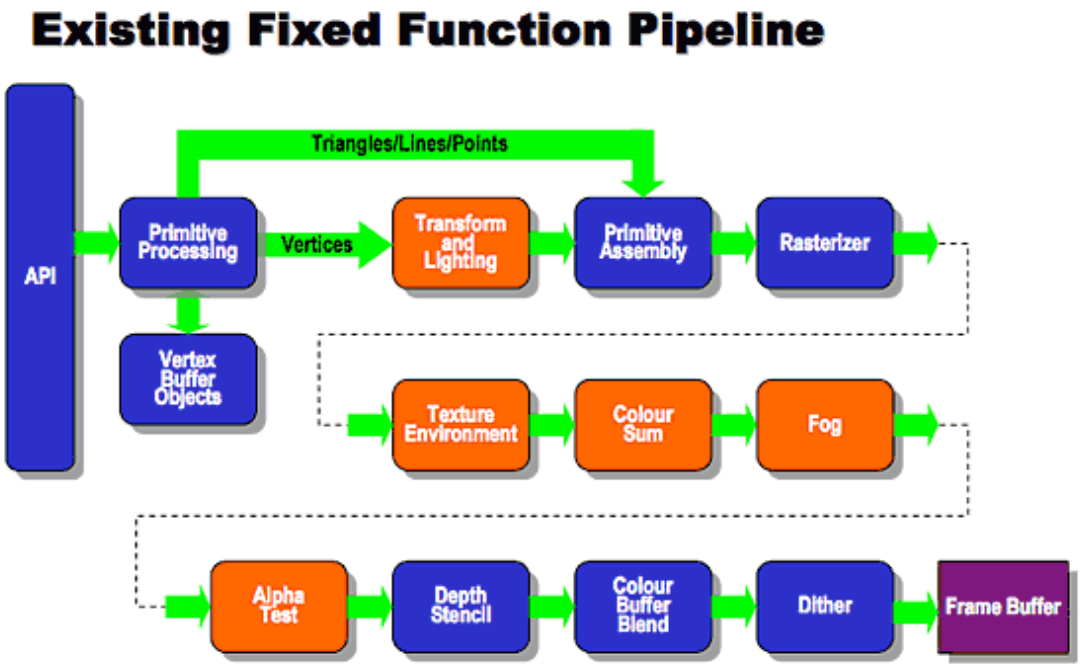
CPU pošle instrukce (zkompilované shadery – programy napsané v shaderovacím jazyce) a geometrická data do GPU na grafické kartě. Ve vertex shaderu je transformována geometrie a vypočteno osvětlení. Pokud je na grafické kartě geometrický shader, jsou provedeny také změny geometrie scény. Výsledná geometrie je triangulována (rozdělena na trojúhelníky) a konečně trojúhelníky jsou rozděleny na čtvercový pixel-fragment (pixel quad – 2x2 pixel). Z těchto fragmentů jsou následně vybrány ty, které budou zobrazeny na obrazovce jako pixely. Tento zjednodušený sled kroků využívá grafická pipeline pro transformaci 3D nebo 2D dat do výsledných 2D optimalizovaných dat pro zobrazení.

Jednotlivé vrcholy z bufferu jsou poslány do vertex procesoru, kde je každý zpracován vertex shaderem, výstupem jsou upravené, transformované vrcholy, ze kterých jsou po několika dalších operacích sestaveny fragmenty a jsou předány fragment procesoru. Fragment procesor určí barvu každého procházejícího fragmentu (ne pozici). Výstupní fragmenty jsou ve framebufferu převedeny na pixely.

Technologie WebGL vychází z OpenGL ES 2.0. Hlavním rozdílem mezi OpenGL ES 2.0 a starší Open GL 1.X je právě programovatelná grafická pipeline nahrazující pipeline s fixní funkcí. Umožňuje kombinovat programovatelné shadery adaptované na potřeby aplikace a API OpenGL, kde je fixní funkce pipeline snadno nahrazena shader programy (vertex a fragment shader), efektivněji zpracovávajícími pokročilou grafiku se sníženými nároky na systém. Blokové schéma grafické pipeline je zobrazeno na obrázku Obr. 4 a pro porovnání rozdílů je na dalším obrázku Obr. 5 také původní fixní grafická pipeline.



Obr. 4: Programovatelná grafická pipeline OpenGL ES 2.0 (převzato z [25])



Obr. 5: Fixní grafická pipeline OpenGL 1.X (převzato z [25])

GLSL

Nízkoúrovňový jazyk pro programování shaderu je assembler, ten však není přehledný, dobře čitelný a je náročný pro programování složitějších funkcí. Proto byly vyvinuty vyšší programovací jazyky jako GLSL, GC, HLSL.

GLSL (OpenGL Shading Language) je programovací jazyk pro zpracování shaderů na GPU. Jazyk je podobný OpenGL ES (vycházejí z jazyka C stejně jako JavaScript), obsahuje však specializované datové typy a funkce pro výpočty ve 3D grafických aplikacích. Shadery jsou napsány přímo do html kódu, odlišené tagem <script>, ale nejedná se o zmíněný JavaScript. Při zpracování stránky jsou shadery rozpoznány podle jejich id (shader-fs, shader-vs) a typu (x-shader/x-fragment, x-shader/x-vertex).

Vykreslování v OpenGL ES 2.0 vyžaduje použití shaderů. Shadery musí být načteny ze zdroje (shaderSource), zkompilovány (compileShader) a přiřazeny programu (attachShader), který musí být slinkován (linkProgram) před výsledným použitím (useProgram).

GLSL pracuje se základními datovými typy jako void, bool, int, float, různými typy n-násobných vektorů a matic a tzv. samplery, umožňujícími přístup k textuře. Mezi další prvky patří struktury a pole. Pomocí znaku # se zapisují direktivy preprocesoru

(#define, #if, #ifdef,…). Proměnné je možné deklarovat s několika možnými typy kvalifikátorů – const (konstanta, jen pro čtení), attribute (propojení vertex shaderu a OpenGL ES daty vertexů), uniform (hodnota se nemění během zpracování primitiva, uniform proměnná určuje propojení mezi shaderem, OpenGL ES a aplikací) a varying proměnná (propojení mezi vertex a fragment shaderem pro interpolovaná data). Existují i další méně používané typy kvalifikátorů. V shaderech se provádí velké množství matematických operací, tomu také odpovídá množství definovaných operátorů a výrazů využívaných pro skalární, vektorové a maticové operace. Dále jsou k dispozici různé typy funkcí – úhlové, trigonometrické, geometrické, maticové, vektorové funkce, funkce pro zpracování textur a další běžné funkce (asb, min,…). Podrobnější popis jazyka GLSL je možné nalézt na oficiálních stránkách skupiny Khronos.

2.7.3 Vertex shader

Vertex shader zpracovává každý vertex předaný grafickému procesoru. Účel tohoto zpracování je transformace 3D pozice (ve virtuálním prostoru) každého vertexu do 2D souřadnic s hodnotou hloubky (Z-buffer) tak, jak se zobrazí na obrazovce. Vertex shader může měnit vlastnosti jako pozice, barva, normály a texturové koordináty, avšak nemůže vytvářet vrcholy. Výstup z vertex shaderu pokračuje do další části zpracování v grafické pipeline (geometrický shader, pokud je k dispozici, nebo následuje rasterizace).

Vertex Shader – příklad (GLSL)

```
uniform mat4 mvp_matrix; // model-view-projection matrix
uniform mat3 normal_matrix; // normal matrix
uniform vec3 ec_light_dir; // light direction in eye coords
attribute vec4 a_vertex; // vertex position
attribute vec3 a_normal;  // vertex normal
attribute vec2 a_texcoord;  // texture coordinates
varying float v_diffuse;
varying vec2 v_texcoord;
void main(void)
{
   // put vertex normal into eye coords
```

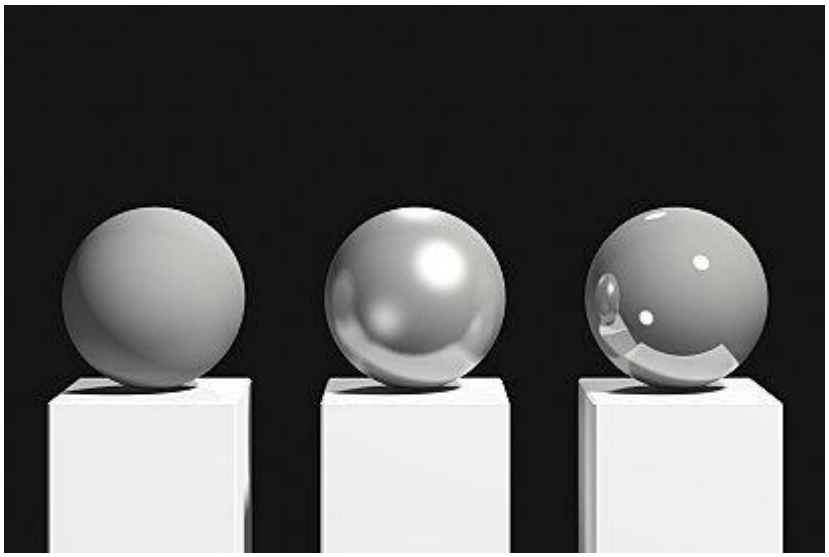
```
    vec3 ec_normal = normalize(normal_matrix * a_normal);
    // emit diffuse scale factor, texcoord, and position
    v_diffuse = max(dot(ec_light_dir, ec_normal), 0.0);
    v_texcoord = a_texcoord;
    gl_Position = mvp_matrix * a_vertex;
}
```

2.7.4 Fragment shader

Někdy bývá označován jako pixel shader, slouží primárně pro výpočet barvy, ale řeší i další vlastnosti, jako například osvětlení, stíny, odrazivost (zrcadlení), průsvitnost a další. Může měnit hloubku pixelu (Z-buffer), nebo postytovat více než jednu barvu na výstupu pro vícenásobné renderování. Pokud má objekt přiřazenu texturu, je zpracována také ve fragment shaderu. Pozice fragmentu je k dispozici avšak nelze ji v shaderu měnit. Samotný fragment shader nezpracuje větší, komplexnější efekty, protože zpracovává pouze jeden pixel bez znalosti geometrie celé scény. Fragmenty nejsou zpracovávány všechny, jsou z nich vybrány pouze ty, které budou zobrazeny.

Fragment Shader – příklad (GLSL)

```
precision mediump float;
uniform sampler2D t_reflectance;
uniform vec4 i_ambient;
varying float v_diffuse;
varying vec2 v_texcoord;
void main (void)
{
    vec4 color = texture2D(t_reflectance, v_texcoord);
    gl_FragColor = color * (vec4(v_diffuse) + i_ambient);
}
```



Obr. 6: Odlišné typy shaderů

2.8 Testování existujících knihoven

S příchodem WebGL se samozřejmě začaly vyvíjet nástroje, frameworky a knihovny usnadňující práci s WebGL, rozšiřující funkčnost a možnosti. Pro účely této diplomové práce je třeba se konkrétně zaměřit na nástroje podporující jak WebGL tak formát Collada, což výběr velmi zužuje. Mezi nejznámější patří:

- GLGE – JavaScriptová knihovna usnadňující použití WebGL, poskytuje API pro přístup k OpenGL ES 2.0, tedy hardwarově akcelerované 2D/3D aplikace bez nutnosti využití plug-inů, podporuje práci s formátem Collada, stále je ve stádiu vývoje
- Scene.js – open source JavaScriptový 3D engine, poskytující WebGL API založené na JSON formátu scény, podporuje formát Collada
- C3DL – Canvas 3D JavaScript library, JavaScriptová knihovna určená pro jednoduché psaní 3D aplikací využívajících WebGL, jedná se o soubor matematických, scénických a 3D objektových tříd ulehčujících vývoj prohlížečových aplikací s 3D obsahem, podporuje formát Collada (avšak ukázková aplikace nebyla funkční)
- CopperLicht – komerční JavaScriptový 3D engine pro vývoj her a 3D aplikací ve webovém prohlížeči, podporuje WebGL canvas a tedy vykreslování akcelerované 3D grafiky bez využití pluginů

- SpiderGL – JavaScriptová knihovna určená pro zobrazování 3D grafiky v reálném čase, poskytuje základní struktury a algoritmy pro vývoj webových 3D aplikací
- Další (PhiloGL, TDL, Cubic VR, WebGLU, three.js,…)

Uvedené knihovny však mohou posloužit pouze jako inspirace, jejich využití má v tomto případě několik nevýhod – jsou stále ve vývoji a mohou obsahovat chyby, jsou pomalé, některé nebylo možné spustit nebo vyžadovaly specifické nastavení, které nebylo možné aplikovat, další nepodporují vyžadovaný formát Collada. Funkční nástroje jsou naopak často netransparentní z hlediska kódu, určeny pro komerční využití a jejich komplexní funkce a velký rozsah je pro využití v problematice renderování formátu Collada ve WebGL zbytečné. Z těchto důvodů se dále práce bude zabývat tvorbou vlastního nástroje, optimalizovaného pro účel aplikace.

3 Praktická část

Na základě vědomostí získaných v teoretické části diplomové práce se praktická část věnuje návrhu a realizaci vlastní knihovny pro vykreslení formátu Collada pomocí WebGL. Je rozdělena na dvě hlavní části. První část se věnuje JavaScriptovému parseru, sloužícímu pro vyseparování dat potřebných pro zobrazení modelu a jejich uložení do výsledného datového objektu. Druhá část sdružuje parser, jednoduchý HTML kód webové stránky s několika základními prvky a samotným WebGL kódem realizujícím vykreslení na plátno canvas.

3.1 Pomocné knihovny

Knihovny WebGL jsou poskytovány každým prohlížečem podporujícím WebGL avšak existuje více knihoven, nástrojů rozšiřujících a zjednodušujících práci s WebGL jako funkce a výpočty s vektory, maticemi, které jsou poměrně složité na kódování, také poskytují předem připravené základní funkce WebGL jako je inicializace contextu, práce s buffery, texturami, nastavení pohledu, perspektivy, práce s kamerou atd.

- sylvester.js – zjednodušuje matematické operace s vektory, maticemi, linkami a rovinami, 3D prostor
- glUtils.js – poskytuje základní maticové a projekční funkce
- glMatrix-0.9.5.min.js – vektorové a maticové operace
- webgl-utils.js – inicializace WebGL, práce s buffery, texturami, uložení a vyvolání matic
- Collada.js – vytvořený parser pro extrakci dat z modelu do připravené podoby pro WebGL (buffery)

JavaScriptové soubory vložené do hlavní webové stránky stejným způsobem jak bylo popsáno v kapitole JavaScript, tedy:

```
<script type="text/javascript" src="glMatrix-0.9.5.min.js">
```

Kromě uvedených knihoven existuje i řada dalších velmi podobných, které lze využít, popřípadě lze potřebné funkce implementovat vlastním způsobem. Důležitým parametrem kromě samotných funkcí je rychlost jejich provádění.

3.2 Parser – Collada.js

Parser má za úkol extrahovat vybraná data potřebná pro zobrazení z Collada modelu, tedy souboru s příponou .dae. Celý je napsaný v JavaScriptu v souboru Collada.js. V zobrazovači Viewer.php je zakomponován pomocí instrukce v hlavičce:

```
<script type="text/javascript" src="Collada.js">
</script>
```

Základní metody parseru jsou load() pro načtení modelu a parse() pro vyseparování potřebných dat. Ve Viewer.php se vytvoří nová instance – tedy objekt Collada, následně se zavolá metoda objektu load(); s cestou k souboru modelu jako parametrem.

```
c = new Collada();
c.load("modely/duck.dae");
```

Po načtení modelu a extrakci potřebných dat již jsou informace obsažené přímo v novém objektu a připravené k využití.

3.2.1 Funkce load();

Funkce slouží pro načtení zvoleného modelu, což je i její parametr. Udává se celá cesta z kořenového adresáře, kde je soubor Collada.js umístěn, až k adresáři s Collada modelem s koncovkou .dae.

- Definována: Collada.prototype.load = function(src){}

Klíčové slovo prototype znamená, že objektu můžeme přidávat další vlastnosti a metody, v tomto případě právě funkci load(). Pro snadné odkazování na objekt je možné použít klíčové slovo this nebo pro lepší přehlednost přímo definovat proměnnou:

```
var collada = this;
```

Pro samotné načtení souboru ve webovém prostředí poslouží XMLHttpRequest. Je to objekt, určený pro snadnou výměnu dat na serveru pomocí http požadavků, s jehož pomocí lze požadovat data po načtení stránky, data přijímat a posílat a tím i upravovat stránku bez nutnosti ji znovu načítat. Vyžádaná data se načtou přímo do JavaScriptu jako XML text nebo klasický text. Uživatel vytvoří instanci tohoto objektu, otevře URL adresu a vyšle požadavek. Request objekt obsahuje status výsledku transakce a po jejím dokončení i samotný obsah výsledku. Uvnitř funkce load(); je tedy vytvořen nový

XMLHttpRequest, dále je implementována funkce pro sledování statusu vyřizovaného požadavku. Název funkce je req.onreadystatechange. Po zavolání metody req.open, která vytvoří nový požadavek typu GET na zdroj scr (URL adresa modelu uvedená v parametru load();) se právě funkcí req.onreadystatechange pravidelně kontroluje stav vyřízení požadavku. Požadavek se následně uvede do chodu pomocí req.send(null) a kdykoli se status požadavku změní, je volána callback funkce onreadystatechange. Pokud jsou splněny podmínky pro dokončené načtení obsahu a zároveň je nastaven příznak (status) pro úspěšné vyřízení http požadavku, funkce uloží výsledek přenosu do proměnné xml a zavolá druhou důležitou funkci následující po load(); - funkci parse(); s výsledkem http přenosu xml = req.responseXML; jako parametrem. Kompletní programový kód funkce load(); s obsluhou http požadovku je následující:

```
Collada.prototype.load = function(src){
        var collada = this;
        var req = new XMLHttpRequest();
        req.onreadystatechange = function(){ //callback
                // Status of 0 handles files coming off the local disk
                if (req.readyState == 4 && (req.status == 200 || req.status == 0)) {
                        var xml = req.responseXML; //vysledek typu XML
                        collada.parse(xml); // 2)
                        if (self._loadHandler) {
                        runSoon(function () { self._loadHandler.apply(window); });
                        }
                }
        }
        req.open("GET", src, true);
        req.overrideMimeType("text/xml");
        req.setRequestHeader("Content-Type", "text/xml");
        req.send(null);
        console.log("dokument byl nacten");
}
```

3.2.2 Funkce parse();

Po načtení XML obsahu modelu ze souboru se výsledný obsah předá funkci parse();. Jedná se o nejrozsáhlejší funkci parseru, jejíž účel je získat potřebná data z načteného XML stromu Collada modelu. Je složena z několika téměř identických částí zajišťujících vstup do určitého místa hierarchické struktury dokumentu a vyseparování potřebných dat. Další součástí metody parse(); jsou funkce pro zpracování získaných dat, které se liší podle typu extrahovaných dat. Funkce samotné nejsou příliš komplikované, bohužel velmi obsáhlý formát Collada, mnoho různých variací a způsobů pro uložení dat činí i z těchto základních funkcí náročné a pracné operace. Proto byly vybrány zatím jen nejdůležitější části modelu Collada, potřebné pro správné vykreslení modelu s několika pravidly pro skladbu modelu. Postup ukládání informací do Collada souboru totiž není nijak standardizován a v podstatě každý software, který model Collada vytváří, ho ukládá jiným způsobem. V budoucnosti je možné pracovat na rozšíření a větší univerzálnosti parseru.

Collada je složená ze struktury uzlů obsahujících konkrétní informace a data. Těchto uzlů může být velký počet, mohou se opakovat, pokud je model složen z více částí a jsou různě složité.

Parser je v současné době schopen správně zpracovat modely bez textury, složené i z více geometrických částí, modely s texturou složené z více geometrických částí, ale je zde kladen požadavek, aby geometrie modelu byla uložena v triangulárním tvaru, tedy jednotlivé plošky, ze kterých se model skládá, tvoří trojúhelníky. V budoucnu je možné rozšířit funkci na obecnější druhy těchto ploch (faců).

Parser zpracovává následující uzly modelu:

```
<asset>
      <contributor>...</contributor>
      <created>2006-06-21T21:23:22Z</created>
      <modified>2006-06-21T21:23:22Z</modified>
      <unit meter="0.01" name="centimeter"/>
      <up_axis>Y_UP</up_axis>
</asset>
```

- Zde je obsažena informace o osové orientaci modelu.

```
<library_images>
      <image id="file2" name="file2" depth="1">
            <init_from>duck.png</init_from>
      </image>
</library_images>
```

- Zde se nachází odkaz na soubor s texturou modelu.

Obr. 7: Zobrazení uzlu geometrie modelu v XML editoru

Získání osového systému modelu

Vzhledem k tomu, že daný uzel se vyskytuje v celé struktuře modelu pouze jednou, je jeho získání triviální záležitostí. Nachází se v několika vnořených uzlech dokumentu, avšak tyto uzly můžeme přeskočit a díky unikátnímu názvu uzlu, obsahujícím informaci o orientaci osového systému stačí vyseparovat uzel s názvem <up_axis>. I když uzel se vyskytuje v dokumentu pouze jednou, je nutné

zadat parametr [0] v metodě getElementsByTagName(„up_axis“)[0], protože funkce getElementsByTagName() vrací pouze pole. Pokud element <up_axis> existuje, načteme jeho obsah do výsledného datového objektu pomocí .nodeValue. Hodnota obsahu může být X_UP, Y_UP. Z_UP. Funkce může být implementována několika rozličnými způsoby, některé jsou v parseru uvedeny.

Získání textury

Model může obsahovat texturu namapovanou na jeho geometrii. Textura se zpracovává v grafickém formátu PNG, pokud je v jiném formátu, měla by se manuálně, nebo programově převést na PNG. Tato část programového kódu se soustředí právě na vyseparování absolutní cesty (adresy) souboru textury a její uložení do výsledného datového objektu. Texturové koordináty důležité pro správné namapování textury na objekt jsou zpracovávány až v části programu zaměřené na geometrická data.

Na počátku parser vybere z dokumentu všechny uzly <image>, obsahující informaci o textuře. Model obecně nemusí mít žádnou texturu, jednu texturu nebo více textur, proto kód pomocí for cyklu prochází všechny nalezené uzly <image>, uloží jejich id a z potomků <init_from> vybere hodnotu – název souboru s texturou, uloží ji do proměnné name. Následuje test koncovky name. Pokud je koncovka „.tga“ program ji změní na „.png“. Pro výslednou adresu textury je potřeba určit nejen název souboru, ale i jeho umístění. K tomu pomůže funkce funkce xml.baseUri.toString();, která vrátí absolutní cestu do složky s modelem a na konec tohoto řetězce se ještě přidá dříve uložená proměnná name s názvem textury. Vše se uloží do proměnné uri. Tím jsou požadovaná data kompletní, stačí jen vytvořit nový objekt image a do vlastnosti img.src uložit získaná data, tedy proměnnou uri.

Geometrická data modelu

Geometrická data představují nejdůležitější aspekt, na který se parser soustředí. Tato obsáhlá číselná data reprezentují vzhled a tvar - body, přímky, jejich vztahy a složení, pomocí nichž lze sestavit model ve 3D prostoru. Geometrická data mohou být popsána mnoha způsoby, nejběžnější jsou mesh (lze si představit jako síť), na který se zaměří parser, dále různé druhy křívek (Bézier, B-Spline, NURBS) a patch mesh. Model může být složen z linek, trojúhelníků, polygonů, polylistu a křivek. Polygon je

obecný n-úhelník, obvykle se využívá 3, 4, 5-úhelník, větší počet bodů již není efektivní. Parser se prozatím zaměří na objekty složené z trojúhelníků (nejčastěji požívané). Trojúhelníky jsou uloženy v poli <triangles> ve formě indexů, odkazujících do polí s konkrétními hodnotami – vertexy, normály a texturové koordináty (vertexes, normals, texcoords).

Jednotlivé geometrie <geometry> jsou uloženy v knihovnách <library_geometries>. Knihoven i geometrií může být více podle počtu objektů, z kterých se model skládá. Na počátku kódu pro získávání geometrických dat jsou načteny metodou getElementsByTagName(); všechny uzly typu <library_geometries> do pole libsGeo. Pro načtení geometrie slouží dva za sebou vnořené for cykly. První prochází všechny uzly knihoven (pole libsGeo) a načítá všechny uzly geometrií (pole geometries) a druhý prochází každou získanou geometrii a vykonává sekvenci pro výpočet samotných dat (vertexů, normál a texturových koordinátů). Sekvence pokračuje uložením atributu id (getAttribute(„id“);) podle kterého lze geometrii identifikovat ve výsledném poli geometrií a výběrem prvního potomka geometrie – uzlu <mesh>. Mesh, jak je zmíněno výše, je způsob popisu geometrických dat. Obsahuje tři základní typy uzlů pro popis geometrie a to uzly typu <source>, <vertices> a uzel určující typ objektu, ze kterých se bude model skládat (lines, linestrips, polygons, polylist, spline, triangles) – v tomto případě byly zvoleny modely složené z trojúhelníků <triangles>. Uzly typu source mohou být zastoupeny v počtu dva a více. Pro jednoduchý jednobarevný model bez textury (kostka) jsou pole source dvě – pozice vertexů a normály, častěji se vyskytují složitější vícebarevné, případně texturované modely, popsané třemi uzly source – pozice vertexů, normály a texturové koordináty. Důležité je správné propojení těchto masivních dat s trojúhelníky <triangles>, které teprve vyberou pomocí odkazů (index) do polí source správné hodnoty, tvořící strukturu modelu. Uzel <triangles> udává tři podstatné parametry použité pro výběr správných hodnot z datových polí source a sice typ vybíraných hodnot (vertex, normal, texcoord), ke každému typu přiřadí offset, s kterým se vybírají jednotlivé indexy a také unikátní atribut id, odkazující na pole ze kterého se vybírají konkrétní hodnoty. Uzel <triangles> může vypadat následovně:

```
<triangles count="4212" material="blinn3SG">

      <input offset="0" semantic="VERTEX" source="#LOD3spShape-lib-
      vertices"/>
      <input offset="1" semantic="NORMAL" source="#LOD3spShape-lib-
      normals"/>
      <input offset="2" semantic="TEXCOORD" source="#LOD3spShape-lib-
      map1" set="0"/>
      <p> „indexy vybíraných hodnot“ </p>
</triangles>
```

Uzel <source> je rozdělen na pole typu float s hodnotami ve <float_array> a specifické informace pro tento typ uzlu vnořené v potomcích <technique_common>, <accessor>. Uzel <accessor> poskytuje informaci o počtu hodnot tvořících celkovou jednotku. Informace je uložena v atributu stride (krok). Pro 3D model je stride = „3“ pro data týkající se vertexů (pozice) a normál, pro texturové koordináty je pak stride = „2“. Uvnitř uzlu jsou ještě vnořeny potomci typu <param> definující jména jednotlivých hodnot podle počtu stride a jejich typ. Uzel source může vypadat následovně:

```
<source id="box-lib-positions" name="position">
      <float_array id="box-lib-positions-array" count="24">
      -50 50 50 50 50 50 -50 -50 50 50 -50 50 -50 50 -50 50 50 -50 -50
      -50 -50 50 -50 -50
      </float_array>
      <technique_common>
            <accessor count="8" offset="0" source="#box-lib-positions-
            array" stride="3">
                  <param name="X" type="float"/>
                  <param name="Y" type="float"/>
                  <param name="Z" type="float"/>
            </accessor>
      </technique_common>
</source>
```

Prvním krokem pro získání hodnot vertexů, normál a texturových koordinátů je načtení všech uzlů <source> a ve for cyklu každý projde sekvencí pro získání dat. Sekvence se skládá z načtení aktuálně procházeného uzlu source do proměnné aktsource, uložením atributu id do proměnné sid (podle id bude jednotlivý zdroj uložen do pole) a načtení pole hodnot typu float. Hodnoty jsou uloženy v uzlu <float_array> a jsou odděleny mezerou. Program vybere uzel <float_array>, vezme jeho obsah (hodnoty s mezerami) a pomocí funkce rozdelFloat(); je rozdělí a uloží do potřebného pole:

```
fdata2 = rozdelFloat(aktsource.getElementsByTagName("float_array")[0].textContent);
```

Data jsou připravena, na konci sekvence program z uzlu <accessor> zjistí hodnotu kroku stride:

```
var accessor = aktsource.getElementsByTagName("accessor")[0];
var stride = parseInt(accessor.getAttribute("stride"));
```

Hodnoty a krok stride každého uzlu <source> jsou uloženy do pole source[sid] podle klíčového identifikátoru sid. Později se z nich vyberou podle indexů v <triangles> konkrétní trojice (vertexy, normály) a dvojice (texturové koordináty) hodnot, tvořících strukturu modelu. Tím je ukončen for cyklus pro uzly <source>.

```
source[sid] = {
        stride: stride,
        data: fdata2
}
```

Funkce parse pokračuje ve for cyklu pro geometrie. Po uzlech <source> přichází na řadu element <vertices>. Element deklaruje atributy a popisuje pozici vertexů meshe. Je spojovacím článkem mezi hodnotami ze <source> a polem tvořícím z těchto hodnot pomocí indexů polygony modelu (v případě této práce se jedná o pole <triangles>).

Element <input> propojuje surová data s nějakým sémantickým vyjádřením těchto dat, aby se s nimi mohlo dále pracovat. Propojení představuje datový tok od zdroje <source> k příjemci – rodičovskému elementu uzlu <input>. Například element uzlu <triangles> - <input semantic="VERTEX" source ="ukazatel_na_vertices"> ukazuje pomocí parametru source na element <vertices> a element obsluhující pozice vertexů <input semantic="POSITION" source="ukazatel_na_source_position"> uzlu <vertices> ukazuje dále pomocí atributu source na id zdrojového pole <source> se specifickým id ze kterého se čerpají hodnoty jednotlivých vertexů skládaných do útvarů.

Uzel <vertices> může mít více potomků <input> specifikovaných hodnotou atributu semantic, na které hodnoty source se zaměřují, podmínkou je <input semantic="POSITION">, tedy pro popis pozice vertexů v meshi, musí být deklarován vždy.

```
<mesh>
  <source id="LOD3spShape-lib-positions" name="position">...</source>
  <source id="LOD3spShape-lib-normals" name="normal">...</source>
  <source id="LOD3spShape-lib-map1" name="map1">...</source>
  <vertices id="LOD3spShape-lib-vertices">
    <input semantic="POSITION" source="#LOD3spShape-lib-positions"/>
  </vertices>
  <triangles count="4212" material="blinn3SG">
    <input offset="0" semantic="VERTEX" source="#LOD3spShape-lib-
    vertices"/>
    <input offset="1" semantic="NORMAL" source="#LOD3spShape-lib-
     normals"/>
    <input offset="2" semantic="TEXCOORD" source="#LOD3spShape-lib-
    map1" set="0"/>
    <p>...</p>
  </triangles>
</mesh>
```

- Znázornění propojení indexů trojúhelníků, tvořících model s konkrétními hodnotami jejich vertexů.

Parser tedy načte element <vertices>, uloží jeho id pro pozdější porovnání s ukazatelem source v <input> elementu uzlu <triangles>, zároveň získá povinnou vnitřní hodnotu atributu source svého elementu <input>, ukazující směrem ke zdroji dat pro pozice vertexů. Za pomoci podmínek if porovnávajících atribut elementu <input> semantic s všemi jeho možnými typy (POSITION, NORMAL, TEXCOORD) uloží pro každý typ (pokud je uveden) ukazatel na jeho datové pole hodnot. Jak je následně znázorněno na příkladě pro povinný typ POSITION:

```
if (inpVert.getAttribute("semantic") === "POSITION") {
        ukazatelVertices = inpVert.getAttribute("source").substr(1);
        console.log("ukazatelVertices ve VERTICES: " + ukazatelVertices);
        }
```

Následuje konečná fáze získání hodnot vertexů, normál a texturových koordinátů z indexů v <triangles> odkazujících se na datová pole <source>. Jsou vybrána všechna pole <triangles> v dané geometrii a pokud existují, načtou se jejich elementy <input> a hodnoty indexů tvořících trojúhelníky z pole <p>. Hodnoty indexů jsou převedeny do požadovaného pole pomocí funkce rozdelInt();. Ve for cyklu se prochází všechny načtené elementy <input> a zpracují se atributy offset, semantic a source potřebné pro získání správných hodnot meshe. Ve třech podmínkách je postupně atribut semantic porovnáván s jeho jednotlivými typy (VERTEX, NORMAL, TEXCOORD) a jsou uloženy parametry offset a source (proměnné začínající na „ukazatel…“), které budou potřeba ve vzorci pro výpočet finálních hodnot vertexů, normál a texturových koordinátů. Podmínka zpracovávající <input> s parametrem semanic=“VERTEX“ se od ostatních odlišuje, protože ukazatel nemusí ukazovat přímo do pole dat <source>, ale většinou se odkazuje na element <vertices>, který specifikuje poziční hodnoty vertexů. Je zde navíc proveden test, zda ukazatel odpovídá dříve uloženému ukazateli vertId, který již odkazuje přímo na pole dat. Pokud ano, je tento ukazatel přímo přiřazen (ukazatelVertex = ukazatelVertices;) a bude využit ve vzorci. Tím jsou zajištěny hodnoty ukazatelů a offsetů využité v konečných vzorcích. Je nutné hlídat správný posun v poli hodnot dosazovaných do vzorce a upravovat parametr maxOffset. V poli indexů <p> určujících trojúhelníky nejsou hodnoty jdoucí po sobě pro každý typ, ale jsou prokládané, a je nutné je správně přeskakovat podle offsetu, který se ale promítne do pole zdrojových dat <source> s dimenzí stride 3 nebo 2 (vertex, normála a texcoord) a musí se mezi sebou vynásobit. Zároveň se délka pole <p> musí podělit podle maximálního offsetu maxOffset aby byla dosažena skutečná délka pole pro jeden typ vyčítaných hodnot, ne pro všechny dohromady. Díky tomu vznikne poměrně komplikovaný vzorec. Zde je uveden příklad tohoto vzorce pro výpočet vertexů:

```
for (var i = 0; i < triData3.length/maxOffset; i++) {
  for (var j = 0; j < (source[ukazatelVertex].stride); j++) {
    vertexes.push(source[ukazatelVertex].data[triData3[i * inpTri1.length +
    offsetVertex]*source[ukazatelVertex].stride + j ]);
  }
}
```

První for cyklus prochází každý bod indexů tvořících trojúhelníky, druhý for cyklus zajistí vyčtení hodnot podle velikosti dimenze – prvků tvořících jednu hodnotu (3 u vertexů, normál, 2 u texturových koordinátů). Do pole vertexů se postupně nasouvají hodnoty ze zdrojového pole source, přičemž index je dán komplikovanou kombinací právě procházeného indexu z pole <p> u <triangles>, vynásobenou počtem prvků <input>, tedy počtem typů hodnot tvořích trojúhelník (většinou vertex, normála, texcoord - 3) a přičtením posunu offset pro každý typ hodnot. Index je ještě vynásoben dimenzí stride pro každý typ vyčítaného prvku a je přičtena hodnota j – pořadí prvku podle jeho dimenze tvořící celek (u vertexu 3). Obdobný vzorec je použit i pro výpočet normál, se změnou v offsetu a ukazateli na pole dat. U vzorce pro výpočet texturových koordinátů se pak ještě mění dimenze stride na velikost 2. Tím je část věnovaná výpočtu vertexů, normál a texturových koordinátů u konce.

V elementu <library _visual_scenes> mohou být v potomcích <node> umístěny vlastnosti popisující celou scénu i model. Například nastavení kamery, světel, materiálu geometrie, počátečního posunu geometrii atd. Parser se zatím soustředí na získání barvy prvku geometrie pro modely bez textury a pro získání počáteční polohy geometrie (posun, natočení).

Parser pokračuje rutinou pro získání barvy geometrie načtením knihoven <library _visual_scenes> a ve for cyklu je jednotlivě prochází. Aktuálně procházená knihovna je uložena do proměnné libVS. Materiálová data jsou uložena v elementu <instance_geometry>. Opět jsou načteny všechny uzly <instance_geometry> a ve for cyklu procházeny. Jejich atribut „url“ je porovnáván s dříve uloženým id geometrie (if (igUrl == geomID)) a pokud souhlasí, program pokračuje výběrem elementu <instance_material> pokud takový existuje. Jeho atribut „target“, odkaz do další Collada knihovny popisující materiály - <library_materials>, je uložen pro pozdější porovnání (proměnná iMatTarget). Obdobně pomocí funkce getElementsByTagName(); je postupně načtena knihovna <library_materials> a její potomci <material>. Opět jsou ve for cyklu procházeny všechny elementy <material>, ukládány jejich atributy id a porovnávány s dříve uloženým odkazem iMatTarget. Pokud souhlasí, uloží se atribut uzlu <instance_effect> „url“ (proměnná iMatEffect) – odkaz do další Collada knihovny pro efekty <library_effects>. Stejný způsobem jako v předchozích případech je tato knihovna načtena, taktéž její potomci <effect> a opět jsou procházeny for cyklem. Pokud atribut id uzlu <effect> souhlasí (if (iMatEffect == effectID)) je z potomků

<diffuse> a následně <color> získána hodnota barvy geometrie a uložena k patřičné geometrii:

```
if (efDiffuse.getElementsByTagName("color")[0]) {
  difColor = rozdelFloat(efDiffuse.getElementsByTagName("color")[0].textContent);
  geometries[geomID].difColor = difColor;console.log("diffuse barva " +
  geometries[geomID].difColor);
}
```

Velmi podobná procedura je použita i pro získání počáteční polohy geometrie. Stejně jako při získávání barvy jsou procházeny knihovny <library_visual_scenes>, z nichž jsou načteny všechny uzly typu <node>. Ve for cyklu se hledá v každém uzlu <node> potomek typu <translate>, obsahující hodnoty počáteční polohy geometrie. While cyklem jsou postupně testovány všechny potomci <node> za pomocí dvou podmínek. Pokud je splněna podmínka if (diteVS.tagName == "translate"), tedy byl nalezen <translate>. Z <translate> je uložen jeho obsah – počáteční souřadnice geometrie. Druhá podmínka if (diteVS.tagName == "instance_geometry") zajišťuje uložení atributu „url“, ukazatele na geometrii, ke které posun patří. Pokud jsou obě podmínky splněny a obě proměnné nejsou prázdné, je do patřičné geometrie uložena počáteční hodnota polohy:

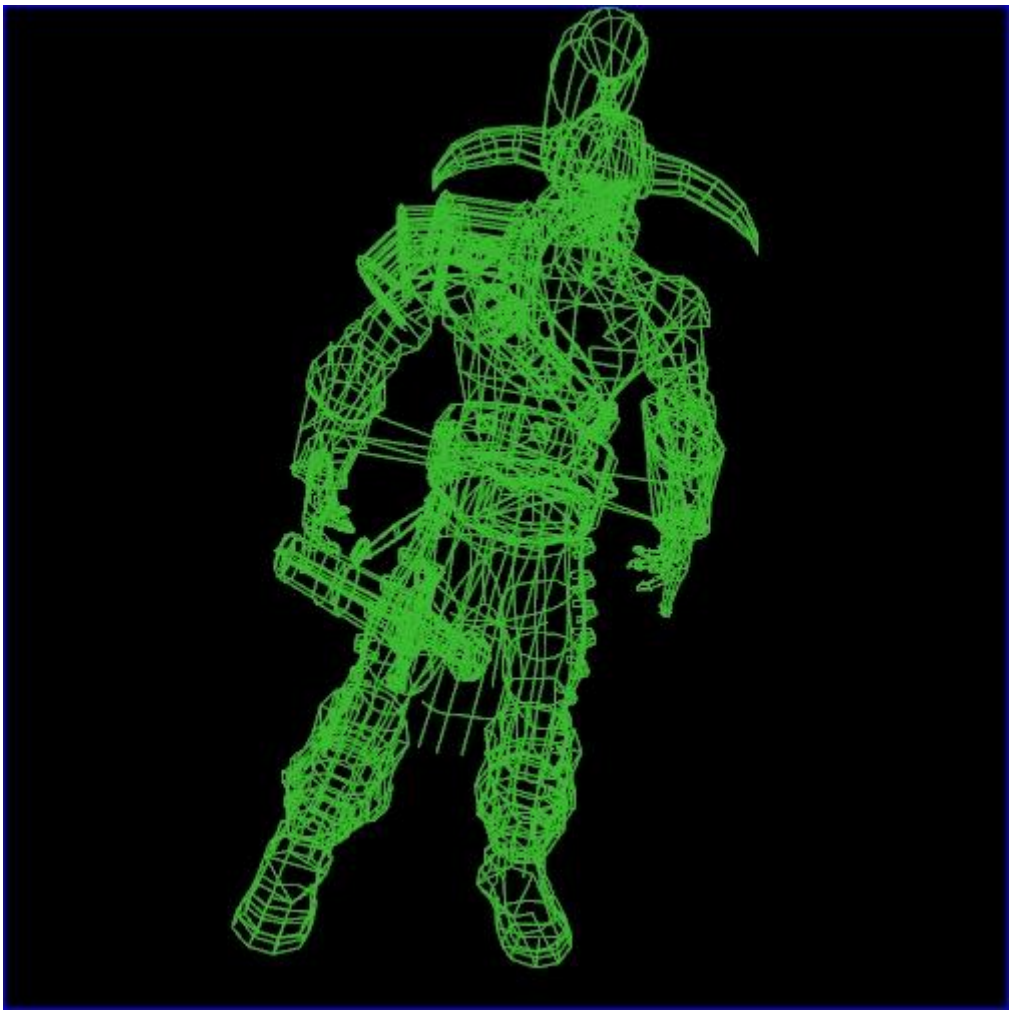
```
if (ukazIG && trData) {
  geometries[ukazIG].translSC = rozdelFloat(trData);
}
```

Funkce parse je uzavřena uložením získaných dat do proměnných a jsou připravena pro vyvolání a využití v hlavním souboru celé aplikace – Viewer.php.

3.2.3 Výsledný datový objekt

Výsledný datový objekt Collada vypadá následovně:

- Funkce:
  - Collada.load(„model.dae“); - načtení souboru modelu a získání datového objektu
  - Collada.parse(„xml“) – zpracování a uložení dat z modelu
- Data:
  - Collada.img
    - Collada.img.src
  - Collada.geomData[]
    - Collada.geomData[].vertexes
    - Collada.geomData[].normals
    - Collada.geomData[].texcoords
    - Collada.geomData[].difColor
    - Collada.geomData[].translSC
- Implementace do html, php (Viewer.php)
  - <script type="text/javascript" src="Collada.js">
- Vytvoření nového objektu
  - c = new Collada();



Obr. 8: Zobrazení geometrie modelu Collada.

Použité funkce a metody DOM pro získávání dat

- .documentElement – Vrací kořenový element dokumentu (například <html> u HTML souboru).
- .getElementsByTagName() – Metoda vrací pole všech elementů s vybraným jménem i s jejich substromem. Vyhledává se v substromu od elementu, u kterého je metoda definována. Strom je živý, což znamená, že změny v dokumentu se projeví i ve výsledku.
- .getElementById() – Vrací objekt – element se specifikovaným Id i s jeho substromem.
- .getAttribute() – Vrátí hodnotu vybraného atributu ze specifikovaného elementu. Pokud atribut neexistuje, vrátí 0 nebo “”.

- .childNodes – Vrací kolekci potomků vybraného elementu, pokud nějaké má.
- .firstChild – Vrací uzel prvního potomka vybraného elementu, pokud existuje, jinak vrací hodnotu null.
- .nextSibling – Vrací následující uzel po vybraném, ze stejné úrovně stromu, tedy následující sourozenec. Pokud je vybraný uzel poslední nebo jediný, vrací hodnotu null.
- nazevUzlu[0] – Parametr [0] vybere z kolekce uzlů (vybraných například metodou getElementsByTagName() nebo childNodes první uzel. Tento parametr je nutno použít, i pokud dokument obsahuje pouze jeden element vybraného názvu, protože funkce getElementsByTagName() vrací vždy pole.
- .tagName – Vrací název vybraného elementu. Použitý například při výběru elementu podle id – není znám název, nebo pro testování správného typu uzlu mezi potomky.
- .nodeValue – Vrací hodnotu uzlu, podle jeho typu. Pro uzly typu text, CDATA, comment vrací obsah uzlu, pro uzly typu atribut hodnotu atributu.
- .textContent – Vrací nebo nastaví text podle typu vybraného elementu a jeho potomků. Při nastavení textu odebere všechny potomky a nahradí je jediným textovým uzlem.
- .baseURI.toString() – Kombinace dvou funkcí. baseURI vrací základní URI objektu, tedy jednoznačnou adresu objektu v síťovém prostředí. Funkce toString převede objekt na textový řetězec. Výsledkem je absolutní adresa URI zvoleného objektu ve formě textového řetězce.
- parseInt() – Vestavěná funkce JavaScriptu. Vezme argument, převede jej do typu string, a pokud je numerického typu vrátí jej jako celočíslený integer. Pokud první znak nelze převést na číslo, funkce vrací hodnotu NaN.
- Console.log() – Velmi důležitá funkce z hlediska testování a hledání chyb. Vyhledávání chyb v JavaScriptu není snadná záležitost a i když je kód správný, vývojář se často potřebuje podívat co se v programu děje a jak vypadají výsledky skriptu, objekty atd. Moderní prohlížeče mají často implementovány podporu zobrazení a práce s JavaScriptovou konzolí, která debuggování značně usnadňuje. Umožňuje nejen zobrazení chybových hlášení ale v podstatě jakéhokoliv objektu, hodnoty, vlastnosti v JavaScriptu, v podstatě čehokoliv co je zadáno jako parametr funkce.

Vytvořené pomocné funkce

- rozdelFloat(), rozdelInt() – Hodnoty geometrických dat typu float a integer uvnitř formátu Collada nejsou odděleny čárkou, jak potřebuje JavaScript pro další zpracování, ale mezerou. Funkce rozdelFloat a rozdelInt od sebe oddělí hodnoty mezi kterými je mezera a uloží do klasického pole, s kterým se bude dále pracovat.

3.3 Zobrazovač – Viewer.php

Jádrem celé aplikace je dynamická webová stránka Viewer.php. Plní celou řadu funkcí. Poskytuje základní UI (uživatelské rozhraní) aplikace složené z několika html elementů, přičemž klíčový element pro zobrazení WebGL je plátno <canvas> se zobrazeným modelem, který je možné pomocí myši ovládat. Další elementy slouží především k nastavení scény (shaderů). V hlavičce <head> jsou importovány rozšiřující javascriptové soubory s pomocnými knihovnami pro maticové výpočty (sylvester.js, glMatrix.js), knihovny zaměřené na funkce WebGL (webgl-utils.js, glUtils.js) a také vytvořený parser pro získání dat z modelu Collada (Collada.js). Poměrně velký prostor hlavičky se věnuje definici shaderů (vertex a fragment), se kterými pracuje samotný skript WebGL. Podstatnou část Viewer.php tvoří WebGL kód psaný v JavaScriptu. Kód zahrnuje inicializaci WebGL, inicializaci shaderů a jejich načtení do objektů (shaderProgram). Dále je definována projekční matice pMatrix (zobrazování 3D objektů na 2D zařízení, nastavení kamery) a modelview matice mvMatrix (transformace scény) a funkce pro operace s maticemi a jejich spolupráci s shadery. Data modelu, vyparsovaná pomocí parseru Collada.js, jsou načtena do bufferů (metoda initBuffers). Po inicializaci všech prvků stránky a načetní potřebných dat je spuštěna sekvence opakujícího se vykreslování. V následujícím textu budou podrobněji popsány jednotlivé součásti aplikace Viewer.php.

3.3.1 Shadery

Shadery se píší ve svém speciálním jazyce GLSL, jedná se o programy vykonávané přímo na grafické kartě a specializují se na grafické operace, výpočty, transformace a další funkce ovlivňující scénu před vykreslením. Umožňují jednu z největších výhod WebGL, a to přesunout výpočetní výkon pro grafické operace přímo

na GPU a tím ulevit CPU. Grafická karta se přímo specializuje na práci s grafikou, vykreslováním, oproti CPU tedy vykonává grafické operace rychleji a efektivněji.

Definovány jsou dva typy shaderů. Fragment shader (neboli pixel) a vertex shader. Fragment shader se stará o efekty pracující s pixely ať už se jedná o barvu, osvětlení, stíny, odrazivé světlo a mnoho dalších efektů. Inicializaci shaderů provede v initShaders();.

V metodě initShaders(); se využívá funkce getShader pro vyčtení fragment a vertex shaderu. Funkce getShader hledá na webové stránce element, jehož id souhlasí s parametrem funkce, vybere jeho obsah, z něj vytvoří podle typu vertex nebo fragment shader. Oba shadery jsou přiřazeny programu (shaderProgram) náležícímu WebGL. Program kompiluje kód do formy, které rozumí grafická karta a umožňuje jeho vykonávání na grafické kartě. Programu je možné přiřadit vždy jeden fragment a jeden vertex shader. Po nastavení programu a přiřazení shaderů je vytvořena reference na atribut pozice a uloží se do nového pole objektu program nazvané vertexPositionAttribute. VertexPositionAttribute je využíván v každém volání funkce drawScene pro vykreslení obrazovky, pro nastavení pozic vertexů trojúhelníků. Bufferu v drawScene je přiřazen právě tento atribut. Vertex shader je volán pro každý vertex, ten je předán do jeho kódu jako aVertexPosition, díky využití zmíněného vertexPositionAttribute v drawScene při asociaci s atributu s bufferem. Uvnitř shaderu se s vertexy mohou provádět nejrůznější operace, transformace, například základní vynásobení pozice vertexu s modelView maticí (perspektivou) a je vrácena výsledná pozice vertexu ve scéně. Další funkce initShaders se týkají tzv. uniform proměnných, do kterých se ukládají ostatní informace využité v shaderu. Funkce setMatrixUniforms přiřadí uniform proměnným reference na modelView a projekční matici. Kompletní kód shaderů je možné nalézt ve zdrojovém souboru Viewer.php.

Atributy shaderProgramu:

uMVMatrix, uPMatrix – modelView a projekční matice

uUseTestures, uSampler – vlastnosti textur

uNMatrix – normály

uColor, uAmbientColor, uDirectionalColor – barvy

uUseLightning, uLightningDirection – osvětlení

3.3.2 Ovládání a prvky webové stránky

Po načtení modelu a jeho zobrazení na canvasu je možné ho částečně ovládat pomocí myši. Levým tlačítkem je možné otáčet modelem, prostřední tlačítko umožňuje posun modelu a točení kolečka ovládá přiblížení modelu. Pomocí elementu <select> umístěného pod plátnem je možné přepínat zobrazované modely, uložené ve složce „./modely“ v kořenovém adresáři aplikace. Další ovládací prvky jsou pole pro zadání hodnot souvisejících s nastavením shaderů. V současnosti je možné nastavit pouze jednotlivé složky (RGB) difuzního osvětlení, ale do budoucna je možné přidat ovládání dalších parametrů a vlastností podle zvoleného shaderu. Posledním typem ovládacích prvků jsou zaškrtávací checkboxy. První slouží pro povolení osvětlení (prozatím není implementováno), druhý pro přepínání barvy pozadí mezi černou a bílou a třetí checkbox přepíná mezi zobrazením modelu nebo jeho kostrou, tvořenou fragmenty.

3.3.3 Zpracování modelu, vykreslení

Po načtení webové stránky Viewer.php se volá metoda handleLoad(), která nuluje využívané objekty, načte vybraný model z comboboxu webové stránky a do vytvořeného Collada objektu načte vyparsovaná data modelu zavoláním funkce parseru load(). Po načtení modelu je zavolána callback metoda parserCallback. Jsou vytvořeny objekty pro uložení dat do bufferů, je vytvořen objekt elementu canvas a následuje inicializace:

```
initGL(canvas);
initShaders();
initBuffers();
if (c.img)
    initTexture();
```

V několika inicializačních metodách se postupně zpřístupní WebGL získáním contextu canvasu, vytvoří se objekty shaderů a výsledný shaderProgram, bufferům se přiřadí získaná geometrická data a pokud model obsahuje texturu, dojde k jejímu načtení.

Provede se základní nastavení WebGL (barva pozadí, hloubka scény, atd.), přiřadí se události pro obsluhu myši a konečně začne každých 15ms volat funkce metoda tick a z ní pak vykreslovací metoda drawScene:

drawTimer = setInterval(tick, 15);

Pokaždé, když je volána funkce drawScene, jsou data předané javascriptovým funkcím převedena na pixely a zobrazena na WebGL canvasu na obrazovce. Vykreslovací funkce drawArrays, WebGL zpracuje dříve poskytnutá data ve formě atributů (z bufferů) a uniform proměnných (projekční, modelview matice,…) a předá je vertex shaderu. Vertex shader je volán pro každý předaný vertex s atributy nastavenými pro daný vertex. Parametry uniform při každém volání nemění. Vertex shader zpracuje poskytnutá data, např. vypočítá hodnotu výsledného vertexu s hodnotami projekční a modelview matice, vertexy tedy budou v perspektivním zobrazení v poloze odpovídající aktuálnímu stavu modelView matice. Výsledek je předáván pomocí tzv. varying proměnných (variable). Těch může být vyšší počet avšak jedna je povinná – gl_Position, která obsahuje hodnoty výsledné souřadnice vertexu. Po vertex shaderu WebGL převede z varying proměnných 3D scénu na 2D scénu pro obrazovku a zavolá fragment shader pro každý pixel tohoto obrázku, který neobsahuje vertex (pixely mezi vertexy). Výplň pixelů mezi vertexy je určena lineární interpolací. Fragment shader vrací barvu pro každý takový bod opět v proměnné typu varying variable gl_FragColor. Výsledek je opět předán WebGL aplikaci pro další zpracování. Po dokončení všech operací je výsledek zapsán do frame bufferu a představuje v podstatě to, co je vidět na obrazovce. Pomocí WebGL je takto možné dostat data z JavaScriptu díky do shaderu i bez přímého propojení, za pomoci varying proměnných. Cyklické volání funkce drawScene každých 15ms zajistí promítnutí změn ve scéně na obrazovku.

3.4 Spuštění aplikace, použité nástroje a ladění

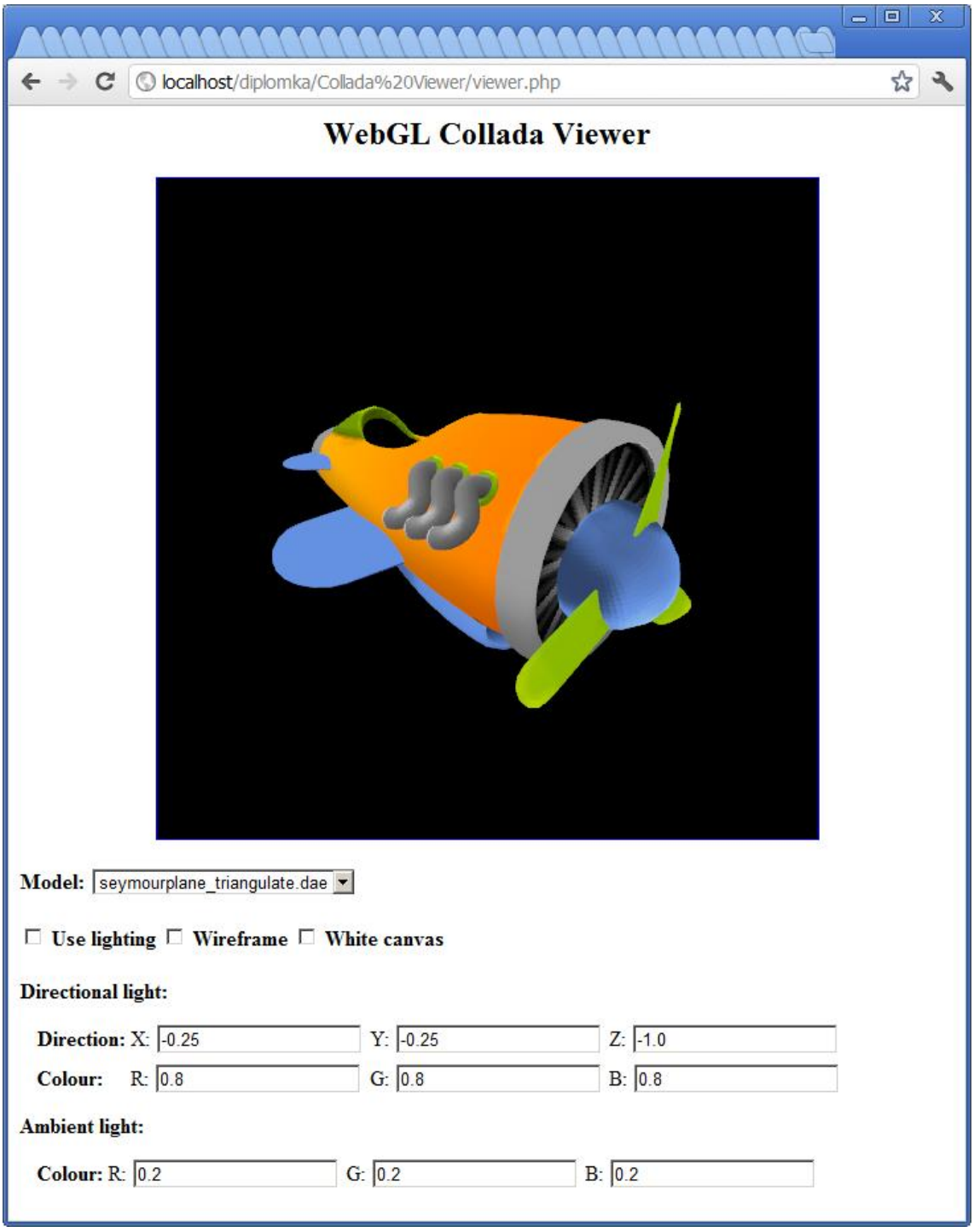
Aplikace se spouští otevřením souboru Viewer.php ve webovém prohlížeči. Správná funkce aplikace a její korektní spuštění je podmíněno podporou WebGL v použitém prohlížeči (Google Chrome, Mozilla Firefox,…) na systému s kompatibilní grafickou kartou. Aplikaci netvoří jen soubor Viewer.php ale i pomocné JavaScriptové knihovny, parser Collada.js a složka ./modely obsahující modely s texturami (pokud nějaké mají). Vzhledem k tomu, že se jedná o webovou aplikaci a přistupuje se k různým souborům v jejím adresáři, je nutné celou její složku nakopírovat na webový server a z něj potom spouštět. Při tvorbě aplikace byly využívány dva způsoby spouštění. První možnost je nakopírovat celou složku na webový server, druhá možnost je spustit lokální server na svém počítači například pomocí balíčku Xampp (Apache server s podporou PHP).

Kromě zmíněného balíčku Xampp se serverem Apache a PHP byly při vývoji aplikace použity další nástroje jako editory HTML, PHP, JavaScriptu, prohlížeč XML souborů. Pro testování aplikace byl zvolený prohlížeš Google Chrome, jehož podpora WebGL je na velmi vysoké úrovni a poskytuje v podstatě jediné nástroje pro efektivní debugování a testování aplikace. Jedná se o konzoli JavaScriptu, do které lze vypisovat v podstatě veškeré objekty a hodnoty. Konzole se spouští z menu prohlížeče nebo klávesovou zkratkou CTRL + SHIFT + J. Konzole je součástí ostatních nástrojů pro vývojaře s jejichž pomocí je možné přímo editovat HTML kód, prohlížet veškeré zdrojové soubory, debugovat skripty a sledovat běh aplikace v čase. Nástroje pro vývojaře se opět dají vyvolat z menu nebo pomocí klávesové zkratky CTRL + SHIFT + I.

Pro testování bylo vybráno několik různých typů modelů. Modely pochází z internetových zdrojů, většina z banky testovacích modelů [6].

Některý modelovací software umožňuje export modelu ve formátu Collada. Vhledem k rozsáhlosti formátu ale dochází k odlišnému způsobu ukládání a zpracování modelů. Může být velmi složité rozlišit, zda model opravdu odpovídá Collada schématu a bude zobrazen správně. V současné době je aplikace schopna zobrazit modely, jejichž geometrie je uložena v triangulárním tvaru (model je složen z trojúhelníků), modely složené z více geometrických prvků, modely bez textury a modely s jednou texturou. Složitější modely pravděpodobně nebudou zobrazeny správně.

Na následujícím obrázku Obr. 9 je zobrazena výsledná aplikace.



Obr. 9: Vzhled výsledné aplikace

Závěr

Cílem práce bylo seznámit se s problematikou vykreslování 3D scén v prostředí WebGL, jazykem JavaScript, který je pro práci s WebGL využíván, a dalšími souvisejícími technologiemi. Bylo třeba prozkoumat existující řešení, porovnat jejich vlastnosti a na základě získaných informací navrhnout a realizovat vlastní knihovnu umožňující vykreslení modelů ve formátu Collada pomocí WebGL v různých webových prohlížečích.

Zpracování a zobrazení 3D grafiky je velmi náročné na hardware, proto jsou specifické výpočetní operace přesunuty z procesoru počítače na grafickou kartu. WebGL umožňuje pro práci s 3D grafikou využít grafický hardware prostřednictvím programovatelných shaderů. Programovatelné shadery přinášejí možnost optimalizovat jejich funkci a efektivitu přesně podle potřeb aplikace. Velkou výhodou WebGL je, že oproti jiným způsobům zobrazení 3D grafiky ve webových stránkách není potřeba instalace žádných pluginů a díky multiplatformnosti může standard WebGL konkurovat ostatním technologiím. Bohužel, stále se jedná o vyvíjený standard. Jeho funkčnost a podpora není stoprocentní, aplikace se mohou chovat různě v závislosti na použitém webovém prohlížeči, optimalizaci kódu a v neposlední řadě podpoře grafické karty. Implementace WebGL se stále mění, ať už se jedná o zmíněné prohlížeče, propojovací prvek s webovou stránkou – HTML 5 element canvas a nebo samotné JavaScriptové rozhraní.

V teoretické části jsou popsány nejdůležitější nástroje a technologie nutné pro realizaci výsledné aplikace. Kromě WebGL a formátu Collada se jedná také o jazyk HTML5 a jeho nový element canvas umožňující propojení s webovou stránkou, programovací jazyk WebGL – JavaScript. Důležitou část teorie tvoří programovatelné shadery a jejich souvislost s grafickým hardware. Z rozboru nejrozšířenějších prohlížečů vyplývá, že podpora WebGL je v současnosti již na dobré úrovni. Vedoucí postavení má v tomto ohledu Google Chrome, na druhé straně Internex Explorer naopak s podporou WebGL nepočítá vůbec. První část diplomové práce uzavírá test existujících knihoven pro zobrazení modelů Collada ve WebGL. Existujících metod (nástrojů, knihoven) není mnoho, jejich vlastnosti se velmi liší a ve většině případů jsou v rozporu s účelem tvorby výsledné aplikace. Nástroje, které se podařilo zprovoznit, měly celou řadu nevýhod – nebyly optimalizovány pro zobrazování modelů Collada, jejich funkce nebyla korektní, jsou pomalé, komerční prostředí jsou naopak zbytečně obsáhlá a

komplexní pro jednoduché potřeby aplikace, nehledě na jejich pořizovací cenu. Pokus o zobrazení modelu ve většině testovaných knihoven nedopadl dobře. Z těchto důvodů testovaná prostředí spíše posloužila jako inspirace a návrh pro tvorbu vlastní knihovny v praktické části.

Praktická část se věnuje tvorbě vlastní aplikace pro zobrazení Collada modelů pomocí WebGL. Skládá se ze dvou hlavních částí – JavaScriptového parseru Collada.js a webové stránky Viewer.php. Parser slouží pro vyseparování dat z modelu Collada, potřebných pro jeho zobrazení. Je mu věnován poměrně velký prostor, jeho implementace zabrala značné množství času při tvorbě aplikace. Rozsáhlost Collady a snaha o vytvoření univerzálního formátu má za následek komplikovanější zpracování. Parser má dvě hlavní funkce a to načtení modelu load(); a vyseparování potřebných dat do výsledného datového objektu parse();. Webová stránka Viewever.php pak využívá po výběru zobrazovaného modelu právě parser Collada.js a předaná data se pokusí pomocí WebGL zobrazit na hlavní komponentu stránky – element canvas. Viewer.php je v podstatě jednoduchá webová stránka s několika elementy. Kromě zobrazovacího plátna obsahuje prvky pro nastavení modelu a ovládání zobrazení pomocí myši. Sdružuje všechny knihovny a funkce, které aplikace využívá a významnou část stránky tvoří WebGL kód cyklicky zobrazující zvolený model.

Výsledná aplikace je schopna zobrazit několik typů Collada modelů s určitým omezením. Parser je schopný zpracovat trianglulární modely, tedy modely tvořené trojúhelníkovými plochami. Aplikace je schopna zobrazit model skládající se z více jednotlivých částí a také model s texturou. Složité a komplexní modely zatím není možné zpracovat, což je také námět pro možné vylepšení aplikace. Kromě toho by také bylo možné vylepšit vzhled webové stránky, modifikovat ovládací prvky a přidat další nastavení podle potřeby uživatele. Využití programovatelných shaderů přináší další možnosti optimalizace výpočtu a vykreslení scény. Vývoj aplikace je velmi náročný z hlediska testování. V podstatě jediný nástroj na ověřování funkce je výpis do konzole prohlížeče. Hledání chyb při práci s grafickou informací je náročné.

Formát Collada poskytuje velmi obsáhlý kontejner pro ukládání nejen geometrických dat modelu, ale i dalších vlastností souvisejících se zobrazením scény (fyzikální vlastnosti, animace, světelné efekty, atd.). Jeho komplexnost však znamená vyšší nároky a složitější zpracování. Nevýhodou je nedostatek modelů Collada na

internetu a jejich odlišná skladba. Software schopný exportovat modely ve formátu Collada často ukládá modely jiným způsobem než konkurence.

WebGL, díky přenosu zpracování grafické informace na grafický akcelerátor pomocí programovatelných shaderů a dalším výhodám, je jistě technologie hodná dalšího zkoumání, ačkoliv množství informací, dokumentace, podpora prohlížečů a samotná funkce není z důvodu probíhajícího vývoje ideální. Dá se předpokládat, že do budoucna se tyto nedostatky budou stále zlepšovat.

Seznam použité literatury

[1] Khronos Group. WebGL - OpenGL ES 2.0 for the Web [online]. 2012. [cit. 12.9.2011].
URL: <www.khronos.org/webgl>

[2] Khronos Group. WebGL - COLLADA - 3D Asset Exchange Schema [online]. 2012. [cit. 12.9.2011].
URL: <www.khronos.org/collada>

[3] WebGL Public Wiki [online]. 2012. [cit. 12.9.2011].
URL: <www.khronos.org/webgl/wiki/Main_Page>

[4] Khronos Group. OpenGL ES - The Standard for Embedded Accelerated 3D Graphics [online]. 2012. [cit. 12.9.2011].
URL: <www.khronos.org/opengles>

[5] COLLADA - Digital Asset and FX Exchange Schema [online]. 27.1.2011. [cit. 12.9. 2011].
URL: <www.collada.org>

[6] COLLADA Test Model Bank [online]. [cit. 12.9. 2011].
URL: <www.collada.org/owl>

[7] WHATWG. Web Hypertext Application Technology Working Group [online]. [cit. 12.9. 2011].
URL: <www.whatwg.org>

[8] W3C. World Wide Web Consortium (W3C) [online]. 2012. [cit. 12.9. 2012].
URL: <http://www.w3.org>

[9] W3Schools. W3Schools Online Web Tutorials [online]. 2012. [cit. 12.9. 2011].
URL: <www.w3schools.com>

[10] Learning WebGL. Learning WebGL …lessons 'n' links… [online]. 2012. [cit. 10.1. 2012].
URL: <http://learningwebgl.com>

[11] Planet WebGL [online]. 2012. [cit. 10.1. 2012].
URL: <http://planet-webgl.org/>

[12] Mozilla Developer Network. Mozilla Developer Center - MDN [online]. 2.5.2011. [cit. 10.1. 2012].
URL: <https://developer.mozilla.org>

[13] Mozilla Developer Network. WebGL – MDN [online]. 2012. [cit. 10.1. 2012].
URL: <https://developer.mozilla.org/en/WebGL>

[14] Google. Google Chrome - Google Developers. [cit. 10.1. 2012].
URL: <https://developers.google.com/chrome>

[15] Tvorba-webu.cz. Tvorba webu: tvorba www stránek [online]. 2008. [cit. 10.1. 2012].
URL: <www.tvorba-webu.cz>

[16] Fyrd. When can I use WebGL? [online]. 2012. [cit. 10.1. 2012].
URL: <http://caniuse.com/webgl>

[17] Does My Browser Support WebGL? [online]. 2011. [cit. 10.1. 2012].
URL: <http://doesmybrowsersupportwebgl.com>

[18] AnalyticalGraphicsInc. WebGL Report [online]. 2011. [cit. 10.1. 2012].
URL: <http://webglreport.sourceforge.net>

[19] Xeolabs. SceneJS - WebGL Scene Graph Library [online]. 2009. [cit. 10.1. 2012].
URL: <http://scenejs.org>

[20] Canvas 3d JS Library. Canvas 3d JS Library [online]. 2010. [cit. 10.1. 2012].
URL: <http://www.c3dl.org>

[21] Paul Brunt. GLGE WebGL Library/Framework [online]. 2010. [cit. 10.1. 2012].
URL: <http://www.glge.org>

[22] Visual Computing Laboratory. SpiderGL [online]. [cit. 10.1. 2012].
URL: <http://www.spidergl.org>

[23] WHATWG. FAQ - WHATWG Wiki [online]. 23.11.2011. [cit. 12.9. 2011].
URL: <www.whatwg.org>

[24] Tantek Çelik. HTML5 Now with Tantek Çelik [online]. 2010. [cit. 10.1. 2012].
URL: <http://tantek.com/presentations/2010/04/html5-now>

[25] Khronos Group. OpenGL ES 2_X - The Standard for Embedded Accelerated 3D Graphics [online]. 2012. [cit. 12.9.2011].
URL: <http://www.khronos.org/opengles/2_X>