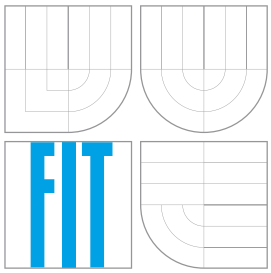
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

# VYUŽITÍ WEBGL ENGINE PRO ZOBRAZOVÁNÍ AUTOMOBILŮ

WEBGL ENGINE EXPLOITATION FOR RENDERING OF AUTOMOBILES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ MACKŮ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL TÓTH

BRNO 2016

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

# Zadání diplomové práce

Řešitel: **Macků Jiří, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Využití WebGL engine pro zobrazování automobilů**
**WebGL Engine Exploitation for Rendering of Automobiles**
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte dostupnou literaturu a dostupný software na téma zobrazování automobilů v prostředí WebGL.
2. Navrhněte postup zobrazování 3D modelu automobilu tak, aby bylo možno automobil zobrazovat realisticky i "stylizovaně" například pro zdůraznění zobrazení některých systémů automobilu.
3. Navrhněte postup a způsob implementace a diskutujte dosažitelné možnosti a vlastnosti implementace.
4. Implementujte WebGL zobrazení automobilů a demonstrujte vlastnosti na vhodném příkladě.
5. Diskutujte výsledky práce a možnosti jejího dalšího pokračování.

Literatura:

- Dle pokynů vedoucího práce

Při obhajobě semestrální části projektu je požadováno:

- Body 1-3 zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Tóth Michal, Ing.**, UPGM FIT VUT
Konzultant: Zemčík Pavel, prof. Dr. Ing., UPGM FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
*vedoucí ústavu*

# Abstrakt

Tato práce se zabývá zobrazováním automobilů za pomoci technologie WebGL. Cílem práce je navrhnout a implementovat způsob realistického i stylizovaného zobrazení 3D modelů automobilů pro interní použití jako jednoduchý náhled automobilu při vývoji a prezentaci nebo pro použití v infotainment systémech přímo v automobilech. Součástí práce je také implementace a nástin dosažitelných možností a vlastností takové implementace. Práce je vytvářena ve spolupráci s firmou ŠKODA AUTO a.s.



# Abstract

This paper deals with rendering of automobiles with use of WebGL technology. Aim of this paper is proposal and implementation of method for realistic and stylized rendering of 3D models of automobiles for internal use as a simple preview of automobile in development and presentation or for use in infotainment systems directly in automobiles. Part of this paper is implementation and outline of reachable options and properties of such implementation. This paper is created in cooperation with company SKODA AUTO a.s.



# Klíčová slova

WebGL, renderování, automobily, javascript, HTML5, infotainment systémy

# Keywords

WebGL, rendering, automobiles, javascript, HTML5, infotainment systems


# Citace

MACKŮ, Jiří. *Využití WebGL engine pro zobrazování automobilů*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Tóth Michal.

# Využití WebGL engine pro zobrazování automobilů

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Tótha. Další informace mi poskytli prof. Dr. Ing. Pavel Zemčík a Mgr. Antonín Míšek, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Macků
25. května 2016


## Poděkování

Chtěl bych zde poděkovat svému vedoucímu Ing. Michalu Tóthovi a konzultantům Prof. Dr. Ing. Pavlu Zemčíkovi a Mgr. Antonínu Míškovi, Ph.D. za odbornou pomoc, praktické připomínky a trpělivost během vzniku této práce. Také bych chtěl poděkovat své rodině za psychickou a materiální podporu.

# Obsah

# Kapitola 1

# Úvod

Zobrazováním automobilů rozumíme určitou vizualizaci počítačových 3D modelů reprezentujících reálné automobily. Tato vizualizace může být realistická nebo stylizovaná (nerealistická).

Při realistické vizualizaci se snažíme dosáhnout co nejvěrohodnějších výsledků. Realistická vizualizace se využívá například ve filmech, ale hojně ji využívají také samotní konstruktéři a designéři v automobilkách při vývoji a výrobě vozů. K realistické vizualizaci používáme 3D modely, které jsou věrnou kopií svých reálných předloh nebo jsou samy předlohou pro budoucí vyráběný automobil. Tyto modely jsou velmi podrobné (desítky až stovky miliónů trojúhelníků), a proto je jejich zobrazování velmi výpočetně náročné.

Při nerealistické vizualizaci můžeme mít různé cíle. Často se snažíme docílit zobrazování v reálném čase nebo se snažíme docílit nějakého uměleckého dojmu (komiksový vzhled apod.). Takováto nerealistická vizualizace se používá například při zobrazení automobilů v počítačových hrách a filmech. Cílem nerealistické vizualizace v průmyslovém prostředí je poté často dosažení větší přehlednosti (zvýraznění křivek, součástek, apod.). K nerealistické vizualizaci používáme 3D modely, které jsou méně podrobné (tisíce až statisíce trojúhelníků), co má za následek snížení výpočetní náročnosti. Modely také nemusí vždy odpovídat přesně reálné předloze, nebo jsou dokonce úplně vymyšlené.

Pro vizualizaci se používají mnohé profesionální nástroje jako například: Autodesk 3Ds Max, Autodesk Maya, CATIA, případně Blender. Tyto nástroje umožňují kompletní návrh, tvorbu i zobrazení modelů. Častěji se však používají pouze pro tvorbu modelů a pro zobrazení se použije nějaký externí nebo proprietární nástroj.

Zobrazování je obecně výpočetně náročný úkol, a proto se k zobrazování používá specializovaný hardware, nejčastěji grafická karta. Pro přístup k tomuto hardware se používají speciální API, např: OpenGL nebo DirectX.

V posledních pár letech vzrůstá také zájem zobrazovat obecně 3D grafiku ve webovém prostředí. Donedávna toto umožňovaly pouze technologie jako Flash, Java, apod. Problém těchto technologií je, že potřebují instalovat doplňky do webových prohlížečů, které představují bezpečnostní hrozbu. Přibližně od roku 2009 je k dipozici nová technologie WebGL, která je přímo součástí téměř všech webových prohlížečů.

Práce je vytvářena ve spolupráci se Škodou Auto a jejím smyslem je předvést možnosti WebGL na poli zobrazování automobilů. V součastnosti je diskutováno využití WebGL ve dvou aplikacích. Prvním je zobrazení automobilu na displeji palubního infotainment systému, kde by bylo možné zobrazit informace o konkrétních částech vozu, případně přímo ovládat některá nastavení. Druhým využitím je zobrazení náhledu automobilu v ovládací aplikaci interního renderovacího sytému VRUT.

Z předchozího odstavce vyplývají požadavky a omezení na navrhovaný WebGL renderer: musí podporovat vstupní data (3D modely) ve formátu pro knihovnu VRUT, měl by být interaktivní, rychlý a nenáročný.

V následující kapitole proberu teoretické základy WebGL, z čeho vychází, co umožňuje, apod. V další kapitole představím knihovnu VRUT, co je jejím smyslem, jak funguje, jaká data umí zpracovat, atd. Poté v následující kapitole představím, co to jsou infotainment systémy a jaké jsou jejich úskalí. V páté kapitole popíšu návrh vlastního řešení zobrazování automobilů ve WebGL. V další kapitole podrobně popíšu samotnou implementaci navrženého řešení a jeho jednotlivých součástí. V sedmé kapitole poté otestuji svoji implementaci hlavně z hlediska kompatibility napříč prohlížeči a na desktopové versus mobilní platformě. V závěrečné kapitole zhodnotím svou práci a navržené postupy.

# Kapitola 2

# WebGL

WebGL, neboli Web Graphics Library, je JavaScriptové API pro renderování 2D a 3D grafiky přímo v okně internetového prohlížeče. Je volně dostupné a k jeho použití není potřeba žádný plugin. WebGL program se skládá z kódu napsaném v JavaScriptu a „shaderů“ napsaných v jazyce GLSL, který je vykonáván přímo na grafické kartě. WebGL je vyvíjeno a udržováno neziskovou organizací Khronos group.

## 2.1 Vlastnosti

WebGL verze 1.0 je založená na OpenGL ES 2.0 API, což je zjednodušená verze OpenGL pro vestavěné zařízení, jako tablety a mobilní telefony. Programátor tedy může použít pouze základní shadery (Vertex shader a Fragment shader) a k vytvoření i jednoduchého programu je zapotřebí mnoho příkazů. WebGL integruje 3D grafiku s DOM (Document Object Model) a výstup vykresluje přímo do HTML5 elementu `<canvas>`. Může být také použito dohromady s jakýmkoliv DOM kompatibilním jazykem [8]. WebGL je uvolněno jako „royalty-free“ specifikace, tedy volně dostupná k použití.

## 2.2 Výhody

- API je založené na známých a široce používaných 3D grafických standardech
- kompatibilita mezi různými platformami a webovými prohlížeči
- integrace s DOM
  - načítání obrázků se provádí jednoduše pomocí rozhraní prohlížeče
  - zpracování událostí (event handling) jako v JavaScriptu
  - možnost kombinování s ostatními HTML elementy a dalšími webovými prvky
  - automatická správa paměti
- hardwarově akcelerovaná 3D grafika ve webovém prohlížeči
- skriptovací prostředí pro jednoduché vytváření 3D grafiky není potřeba cokoliv kompilovat a linkovat

## 2.3 Nevýhody

- možná bezpečnostní rizika vycházející z OpenGL ES specifikace:
  - nedefinované chování kvůli zvýšení výkonu, WebGL se snaží ošetřovat
  - přístup mimo oblast paměti nebo nealokované paměti, WebGL se snaší ošetřovat
  - Denial of Service zablokování systému po dobu vykreslování náročné grafiky
  - validace shaderů nutná před kompilací shaderů
  - načítání obrázků a videí z externích zdrojů – WebGL akceptuje pouze zdroje validované Cross-Origin Resource Sharing
- oproti OpenGL částečně ořezané (bez podpory výchozích zdrojů osvětlení, pouze dva základní shadery, bez typu double `GL_DOUBLE`, ...)
- pro debuggování je potřeba použít další nástroje (WebGL Inspector, ...)

## 2.4 WebGL 2.0

Práce na specifikaci WebGL 2.0 započaly v roce 2013, přesněji v roce 2013 WebGL Working Group zveřejnila první veřejný návrh této specifikace. V poslední verzi (květen 2016) se specifikace nachází ve skoro finální podobě.

Hlavním cílem specifikace WebGL 2.0 je přinést do prohlížečů schopnosti ze specifikace OpenGL ES 3.0. Podpora WebGL 2.0 bude znamenat podstatné rozšíření možností zobrazování 3D grafiky v prohlížeči.

Přesto, že specifikace ještě není kompletní, tak již existují experimentální implementace. Jedná se o implementace v aktuálních verzích webových prohlížečů Mozilla Firefox (`"experimental-webgl2"`) a Chrome Canary (`"webgl2"`). Pro zprovoznění WebGL 2.0 v těchto prohlížečích je nutný OpenGL ES 3.0 kompatibilní hardware a dále je nutné ručně povolit WebGL 2.0.

**Novinky ve WebGL 2.0 oproti WebGL 1.0:**

- více cílů renderování
- instancování
- vertex array objekty
- 3D textury
- transform feedback
- GLSL ES 3.00

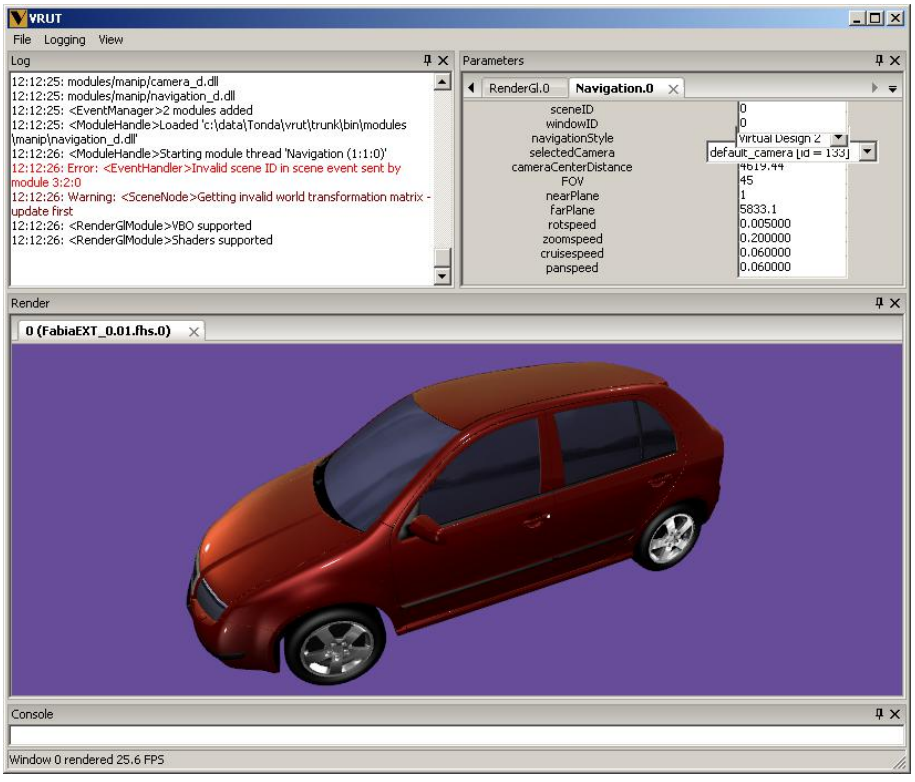
## 2.5 Shrnutí

V této kapitole byla představena technologie WebGL. Byly popsány základní vlastnosti této specifikace. Taktéž byly uvedeny některé výhody a nevýhody, které WebGL přináší. Ke konci byl také představen budoucí nástupce v podobě nové verze WebGL 2.0.

# Kapitola 3

# Knihovna VRUT

Knihovna VRUT [5] (Virtual Reality Universal Toolkit) poskytuje jednoduchý a použitelný nástroj pro práci s grafickými daty, jejich načítání, ukládání a vizualizaci ve 3D. Jako VRUT se označuje samotná knihovna, tak i výsledná aplikace (viz obr. 3.1). Tato knihovna vznikla na základě diplomové práce V. Kyby [4] a dále se rozvíjela za spolupráce firmy Škoda AUTO a.s. a univerzit v České Republice (ČVUT, MU, VŠB a dalších). Univerzitám slouží jako výuková pomůcka ulehčující realizaci projektů, protože studentům odpadá nutnost implementace načítání dat či jejich vizualizace. Společnost Škoda Auto a.s. prostřednictvím knihovny VRUT poskytuje studentům možnost tyto projekty nadále rozvíjet a uplatnit je v automobilovém průmyslu [11].




Obrázek 3.1: Prostředí aplikace VRUT [5]

## 3.1 Architektura knihovny VRUT

Knihovna VRUT by se dala rozdělit na dvě části, hlavní aplikaci nebo jádro a základní balíček modulů. Hlavní aplikace je sama o sobě z uživatelského hlediska nepoužitelná pro jakoukoliv práci; až s pomocí základních modulů jsou zpřístupněny funkce, jako například načtení a zobrazení scény [5].

Moduly jsou v knihovně VRUT rozděleny do několika skupin podle toho, jaké funkce poskytují. Mezi tyto skupiny patří:

- SceneModule – moduly pracující se scénou (například optimalizace scény či detekce kolizí)
- IOModule – moduly umožňující načítání či ukládání grafických dat, jako jsou geometrie či textury
- RenderModule – moduly poskytující vykreslování scény pomocí rasterizace, raytracingu či jiné techniky
- ManipulatorModule – moduly umožňující interakci se scénou
- CameraModule – moduly specializující se na interakci s kamerou
- Module – obecné moduly nespadající do žádné z předešlých kategorií

Tato práce se nezabývá přímo knihovnou VRUT, proto zde nebude podrobněji probírána další její struktura. Navrhovaná aplikace bude pouze využívat data ve formátu VRUT, proto se dále zaměříme pouze na reprezentaci scény ve VRUTu. Z modulů VRUTu nás budou zajímat pouze vstupně-výstupní moduly, které zajišťují načítání a ukládání dat do/z VRUTu.

## 3.2 Reprezentace scény v knihovně VRUT

Objekty, se kterými knihovna VRUT pracuje a které zobrazuje, jsou sdružovány do tzv. scén. Scéna je tvořena skupinou objektů, které spolu nějakým způsobem souvisí. Jedná se například o podrobný model automobilu obsahující jednotlivé díly nebo o scénu zobrazující tento automobil v jeho okolí [11].

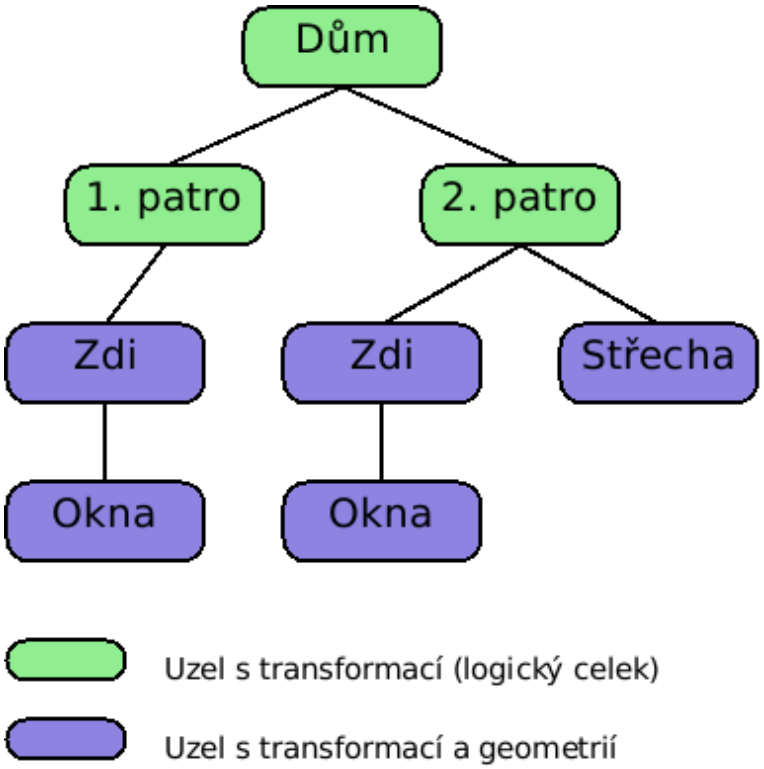
Graf scény představuje hlavní a strukturované úložiště grafických dat scény. N-ární strom reprezentuje vzájemné vztahy mezi logickými celky scény. Základním prvkem grafu scény je uzel. Uzel může obsahovat přímo grafická data a může mít libovolný počet potomků. Graf scény představuje transformační hierarchii (viz obr. 3.2). Každý uzel scény bez ohledu na typ má transformační matici, která definuje umístění podstromu (uzlu a jeho potomků) jako logického celku v lokálním měřítku. Umístění ve světovém měřítku lze získat postupným násobením transformačních matic od kořene stromu po vybraný uzel. Podle druhu objektu se dělí typ uzlu na:

- Assembly – uzel neobsahující nic navíc než své potomky
- Root – vrcholový uzel, reprezentuje soubor na disku
- GeometryNode – uzel obsahující definici geometrie objektu, jejíž součástí je nejen geometrie samotná, ale i materiál s informací o tom, jak má být objekt vykreslen
- BackgroundNode – uzel obsahující pozadí scény

- CameraNode – uzel obsahující kameru s parametry pohledu na scénu
- LightNode – uzel obsahující světlo ve scéně

VRUT podporuje i další druhy uzlů, jako jsou uzly měnící strukturu grafu scény (třídy SwitchNode a LODNode) a uzel se zvuky [5].

Každý SceneNode obsahuje také své jméno, identifikační číslo a příznak indikující, zdali je uzel a jeho potomci aktivní. Jednotlivé podtřídy obsahují další potřebná data, jako je geometrie (v případě GeometryNode), světla (v případě LightNode) a další [11].




Obrázek 3.2: Graf scény s hierarchií transformací [5]

## 3.3 IO moduly

Moduly pro import a export umožňují, jak jejich název napovídá, načítání a ukládání dat scény do souborů s různými formáty. V současné verzi VRUTu existuje několik modulů pro známé formáty:

- IOCSB – import a export formátu CSB, od firmy SGI
- IOFBX – import a export formátu FBX, od firmy Autodesk
- IOOBJ – import a export formátu OBJ
- IOOSB – import a export formátu OSB, z knihovny OpenSG
- IOSTL – import a export formátu STL

- IOVRUT – import a export formátu VRUT, nativní formát VRUTu

Protože specifikace formátů jsou mnohdy nedostupné nebo zastaralé, nemusí moduly podporovat veškeré možnosti daného formátu. Všechny moduly však umožňují načtení a uložení základní trojúhelníkové sítě [5].

Pro navrhovanou aplikaci je nejdůležitější modul `IOVRUT`, který dokáže uložit celý graf scény ve formátu VRUT. Tento formát je interně používaný v celé knihovně a je požadováno, aby tento formát podporovala i navrhovaná WebGL aplikace.

## 3.4 Shrnutí

V této kapitole jsme si představili knihovnu VRUT, která je používána ve firmě Škoda Auto. Podívali jsme se na základní architekturu knihovny. Dále byl popsán způsob reprezentace scény v této knihovně a byly uvedeny některé moduly zajišťující import a export dat do/z knihovny VRUT.

# Kapitola 4

# Infotainment systémy

Výrobci automobilů se neustále snaží vylepšovat komfort řidiče i cestujících v automobilu. V posledních pár letech se velmi rozvíjí oblast takzvaných „infotainment“ systémů. Jedná se v podstatě o pokročilý palubní počítač, který ovládá všemožné palubní systémy a s řidičem komunikuje prostřednictvím dotykového displeje umístěného do palubní desky a dalších ovládacích prvků na palubní desce, středovém panelu nebo volantu. Řidič jej také může ovládat pomocí hlasu, nejčastěji v angličtině. Jak takový infotainment systém může vypadat, je možné vidět na obrázku 4.1.

Infotainment systémy v sobě integrují navigaci, rádio, multimediální přehrávač a mnoho dalších věcí. Navíc tyto systémy také často umožňují propojení s chytrým telefonem. Takové propojení podporují telefony se systémem iOS pomocí platformy Apple CarPlay, android telefony pomocí platformy Android Auto a windows phone, blackberry nebo android telefony také pomocí technologie MirrorLink. Takové propojení dále rozšiřuje možnosti infotainment systémů.


Obrázek 4.1: Infotainment systém COLUMBUS od Škody Auto s 8"LCD displejem, http://www.skoda-auto.cz

Infotainment systémy jsou výborný vynález, je zde však jedno velké nebezpečí: za jízdy takový systém může rozptylovat řidiče. Toto je nutné vzít v úvahu při návrhu aplikace pro takový systém. Prvky grafického uživatelského rozhraní musí být dostatečně velké (aby se

řidič nemusel „trefovat“ prstem do malých tlačítek), takových tlačítek může být v jednu chvíli na displeji jen několik (aby řidič nemusel zkoumat co zmáčknout), taktéž fonty musí být dostatečně velké a čitelné (aby řidič nemusel luštit displej s lupou). Některé aplikace dokonce za jízdy omezují svoji funkcionalitu. Celkový vzhled aplikace a jejího gui by měl být decentní a přehledný, výrazné barvy nebo vyskakovací okénka ponechat pouze pro kritická hlášení.

# Kapitola 5

# Návrh řešení

Cílem práce je navrhnout způsob zobrazení automobilu pomocí WebGL. Toto zobrazení by mělo být využíváno na mobilních zařízeních nebo v infotainment systémech, nelze tedy předpokládat vysoký výkon, spíše naopak. Protože je ze zadání vyžadováno zobrazení realistické i stylizované, bude navrhované řešení rozděleno na dvě části: návrh realistického zobrazení a návrh nerealistického zobrazení.

## 5.1 Realistické zobrazení

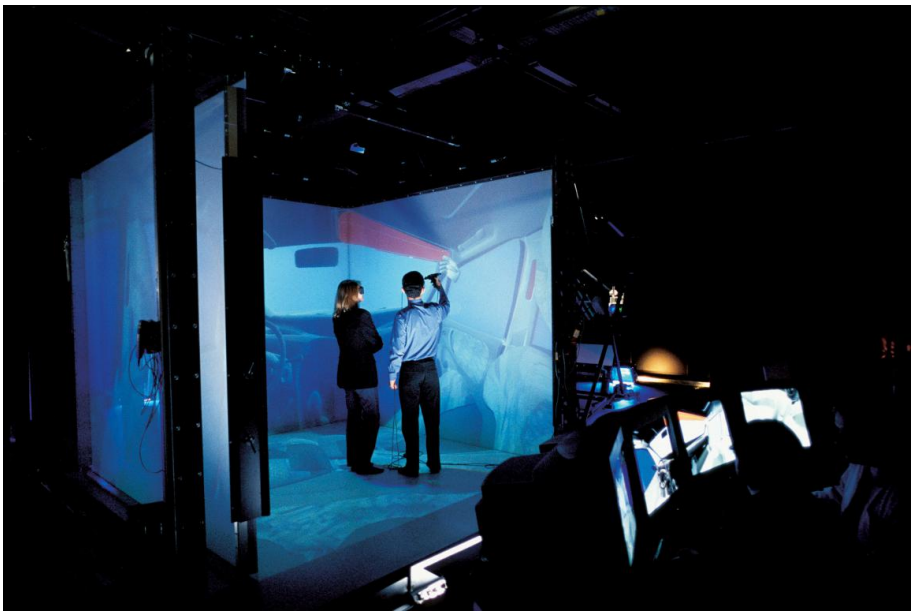
Při realistickém zobrazování je cílem, jak bylo zmíněno v úvodu, dosažení co nejvěrohodnější vizualizace scény. Jako příklad může být obrázek 5.1, lze si zde všimnout odlesků na kapotě, průhledných oken nebo měkkých stínů.

Obrázek 5.1: Škoda Fabia Combi 2008, http://tuning-skoda.blog.cz

K dosažení tohoto výsledku se nejčastěji používají algoritmy ray-tracingu a metody

globální iluminace. Tyto algoritmy jsou velmi výhodné, pokud je scéna složena z malého počtu jednoduchých geometrických objektů. To se však nedá říct o scéně s modelem auta, které může být složeno z mnoha složitých objektů. V takovém případě značně vzrůstá výpočetní náročnost takového zobrazení, protože výpočet průniku paprsku s trojúhelníkovou sítí je zdlouhavý proces. Druhou možností je použít klasickou rasterizaci a například Phongovo stínování. Toto je možné implementovat ve WebGL s použitím vertex a fragment shaderů. S tímto postupem lze dosáhnout pěkných výsledků za kratší dobu.

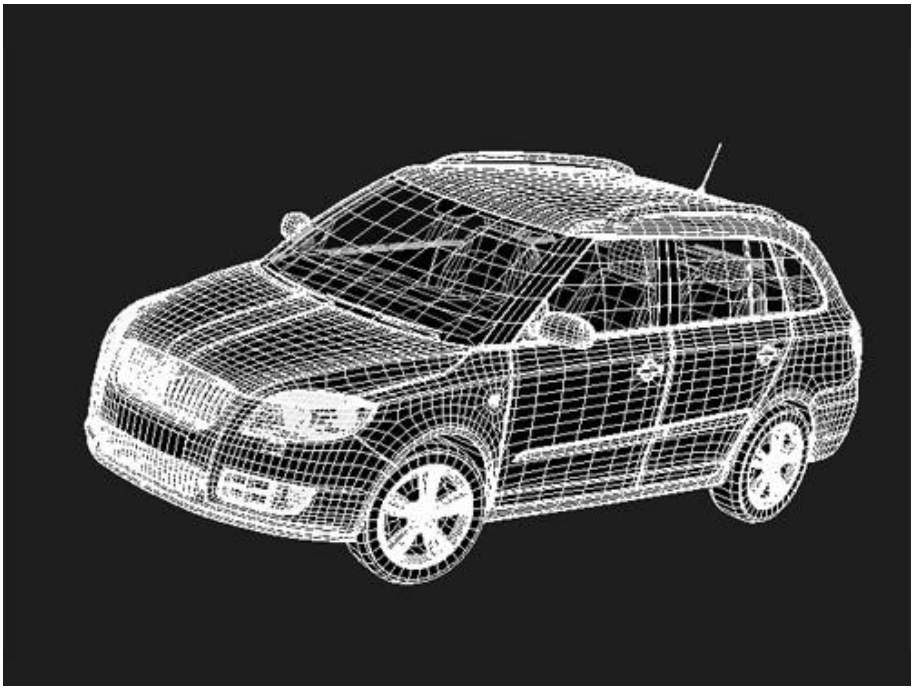
Realistické zobrazení se hodí pro využití jako jednoduchý náhled lepšího renderu přímo z VRUTu. Představa je taková, že VRUT bude například renderovat fotorealistický snímek a uživatel, který VRUT ovládá vzdáleně, vidí na svém zařízení (např. tablet) jednoduchý náhled scény. Tato situace nastává například při prezentaci automobilů v prostředí CAVE (obr. 5.2).



Obrázek 5.2: CAVE – HW pro virtuální realitu [5]

## 5.2 Nerealistické zobrazení

Při nerealistickém zobrazení se snažíme dosáhnout například zvýraznění určitých vlastností nebo součástí zobrazovaného modelu. Příkladem může být obrázek 5.3, kde můžeme vidět takzvané „wireframe“ neboli drátové zobrazení. Toto zobrazení klade důraz na samotný tvar auta, nevidíme zde sice žádné barvy, ale přesto rozpoznáme kola, kapotu, zrcátka, apod.



Obrázek 5.3: Škoda Fabia Combi 2008, http://tuning-skoda.blog.cz

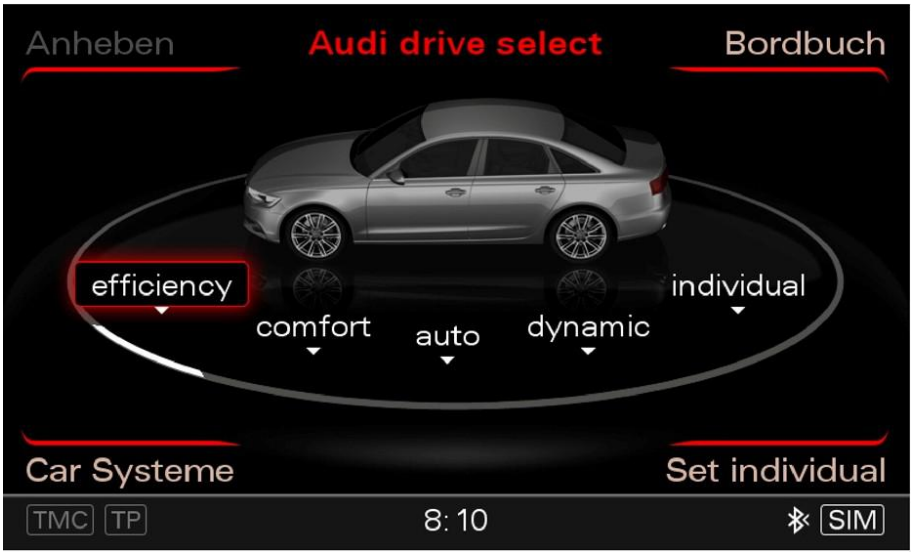
Jiným příkladem je obrázek 5.4. Tento obrázek zobrazuje pohonný systém automobilu jako překryvnou vrstvu přes samotné auto. Zvýrazňuje tím danou část automobilu. Tohoto lze dosáhnout pomocí dvouprůchodového renderu, nejdříve se vykreslí celé auto a poté se přes něj vykreslí pouze vybraný subsystém.

Obrázek 5.4: Škoda Yeti Outdoor, http://www.skoda-auto.cz

Jedno z možných nasazení nerealistického zobrazování automobilů je v infotainment systému. Představa je taková, že by měla aplikace zobrazovat informace o systémech automobilu, jako například poruchy apod. Aplikace by měla také umožňovat ovládání nastavení například audio systému.

## 5.3 Obecný návrh aplikace

Aplikace se bude skládat z několika základních částí. První je zdrojový soubor webové stránky, která bude zobrazovat aplikaci. Struktura tohoto HTML5 dokumentu bude jednoduchá, bude obsahovat jeden element typu `<canvas>` a další elementy pro gui aplikace. Tyto gui elementy budou pravděpodobně generovány dynamicky. Příklad jak by mohlo vypadat gui výsledné aplikace je na obrázku 5.5.




Obrázek 5.5: Audi drive select menu v automobilu Audi A6, http://www.audi-technology-portal.de

Druhou částí jsou soubory se skripty v jazyce javascript. Tyto soubory budou obsahovat většinu funkcionality: zdrojový kód WebGL, kódy shaderů, načítání vstupních dat, ovládání aplikace, atd.

Pro maticové a vektorové operace bude aplikace využívat javascriptové knihovny glMatrix http://www.glmatrix.net.

**Aplikace poté bude fungovat následovně:**

1. po startu aplikace získáme vstupní data obsahující model automobilu
2. zpracujeme získaná data a sestrojíme graf scény
3. spustíme vykreslovací smyčku pro zobrazení scény
   - projdeme graf scény a vykreslíme jednotlivé objekty
   - čekáme na vstup od uživatele, nebo na událost, o které je třeba informovat (např. nějaká porucha)

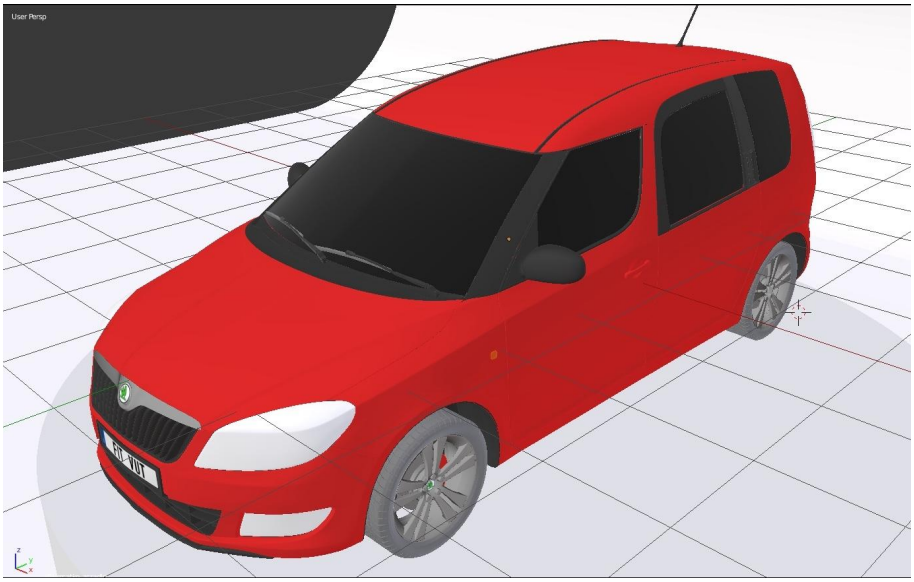
Aplikace je mířená pro použití v mobilních zařízeních nebo infotainment systémech, takže ovládání aplikace bude dotykové. Pro testovací a prezentační účely však bude mít aplikace i klasické ovládání pomocí myši a klávesnice.

## 5.4 Testovací data

Pro otestování správné funkčnosti jsou potřeba nějaká testovací data, konkrétně model nějakého auta. Protože originální modely se nedostanou mimo budovy Škody Auto, tak je možné použít nějaký volně dostupný neoriginální model z internetu nebo si vytvořit vlastní.

Rozhodl jsem se vytvořit si vlastní model. Z vozů značky Škoda jsem si vybral model Roomster 2010, protože daný automobil moje rodina vlastní. Vlastní model jsem vytvořil v programu Blender. Výsledný model je zobrazen na obrázku 5.6.

Pro použití v aplikaci potřebuji model dostat do formátu VRUT. Model jsem tedy z Blenderu vyexportoval do formátu OBJ. Výsledný soubor jsem otevřel v aplikaci VRUT. Protože formát OBJ neuchovává hierarchii objektů, musel jsem zrekonstruovat správně graf scény a nastavit správně materiály. Upravený model jsem vyexportoval ve formátu VRUT. Takto uložený model již lze použít jako testovací model v navrhované aplikaci.




Obrázek 5.6: Model vozu Škoda Roomster v programu Blender

## 5.5 Shrnutí

V této kapitole jsem představil svůj návrh řešení zobrazování automobilů ve WebGL. Zabýval jsem se jak návrhem realistického zobrazení, tak i nerealistického zobrazení. Popsal jsem také návrh struktury výsledné aplikace a uvedl jsem způsob, jakým jsem vytvořil testovací data.

# Kapitola 6

# Implementace

Tato kapitola je hlavní částí práce a proto je rozdělena na další podkapitoly. Na začátku této kapitoly popíšu a zdůvodním určité odchylky od návrhu, ke kterým došlo v průběhu práce. Poté popíšu implementaci základní struktury aplikace. Dále se budu zabývat zpracováním vstupních dat, přípravou scény k vykreslení, implementací vykreslovacího algoritmu, ovládáním a navigací ve scéně a také grafickým rozhraním aplikace.

Zadání práce vznikalo delší dobu při konzultacích s profesorem Zemčíkem a doktorem Míškem ze Škoda Auto. Bohužel při termínu pro zapsání zadání nebylo stále úplně jasné jakým směrem se vyvinou požadavky Škoda Auto, proto bylo zadání vypsáno hodně obecně. S tímto byla také zpracována první polovina této práce, tedy body 1–3 zadání. Krátce po novém roce, během měsíců ledna a února, byly požadavky firmy Škoda Auto vyjasněny a účel práce se přesunul spíše k implementaci realistického rendereru podle C++ rendereru, který obsahuje aplikace VRUT. Hlavní částí této kapitoly tedy bude popis implementace realistického rendereru a jeho součástí. Stylizovanému zobrazení je poté věnována menší samostatná podkapitola dále. Společně s posunem účelu práce se také posunuly požadavky na aplikaci. Aplikace by stále měla být mobilní, ale není přímo určena do infotainment systémů. Při implementaci však bylo na takovéto využití myšleno a s minimálními úpravami je možné aplikaci integrovat do jiné stránky a lze ji tedy využít i jinak. Z tohoto důvodu bude implementováno plné ovládání jak myší a klávesnicí tak dotykem. Podle požadavků byla také zrušena vykreslovací smyčka a vykreslení se provádí takzvaně „on demand“ tj. na vyžádání. Renderer aplikace VRUT se chová stejným způsobem. Z důvodu zatím stále nedokončené specifikace WebGL verze 2.0, bylo taktéž zavrženo použití této verze a je použita verze 1.0.

Implementace probíhala, jak již bylo dříve zmíněno, v jazyce Javascript. Protože WebGL je webová technologie, tak program běží v prostředí nějaké webové stránky. Tato stránka byla napsána v jazyce HTML5 a její vzhled včetně vzhledu prvků aplikace byl definován v jazyce CSS3.

Jazyk javascript je slabě typovaný objektově orientovaný jazyk bez tříd. I přesto je v této práci užíváno terminologie z jazyků s třídami, protože v javascriptu existují způsoby, jak funkčnosti tříd napodobit [9].

## 6.1 Struktura HTML

HTML stránka je pro aplikaci nutnou součástí, jelikož funguje jako ústřední prvek, který spojuje všechny součásti aplikace. HTML stránka definuje objekty, které budou na stránce zobrazeny a jaké další soubory jsou potřeba pro správné zobrazení a funkčnost stránky.

Pro správné zobrazení na mobilních prohlížečích je třeba v hlavičce stránky uvést element `<meta name="viewport»`, který říká prohlížeči, jak ovládat rozměry stránky a její měřítko. Umožňuje například nastavit rozměry stránky tak, aby sledovaly rozměry obrazovky zařízení. Mobilní zařízení totiž umožňují otáčení obrazovky na šířku nebo na výšku. Další možnosti jsou nastavení úvodního měřítka stránky a omezení změny měřítka. Protože aplikace bude zobrazena přes celou stránku, je uživatelská změna měřítka nežádoucí.

Jako další je nutné do hlavičky přilinkovat potřebné javascriptové soubory a soubory s CSS styly. Jedná se o soubor s matematickou knihovnou glMatrix, soubor s kódem aplikace `script.js` a soubor s CSS pravidly `type="style.css"`.

V hlavičce jsou také uvedeny další elementy `<script>`, které obsahují kód shaderů pro grafickou kartu. Těmto se budu věnovat v dále v textu.

Tělo stránky je poté velmi jednoduché. Obsahuje pouze jeden element `<div id="app»` jako obalující element celé aplikace. Tento element dále obsahuje tři potomky. Elementy `<div id="log»` a `<div id="info»`, se používají pro výpis stavových informací nebo chybových hlášek. Element `<canvas id="main»` je nejdůležitější, protože do něj bude vykreslován obraz pomocí WebGL.

Důležité je také nastavení parametru `onload` prvku `<body>`, který definuje javascriptový kód, který bude spuštěn po načtení stránky.

### 6.1.1 CSS styly

CSS styly se používají pro definici vzhledu prvků ve stránce. Styly se zapisují ve formě pravidel. Pravidla umožňují definovat vzhled pouze pro určité vybrané prvky. Prvky lze vybírat podle názvu elementu, identifikátoru nebo třídy.

Element aplikace je pomocí stylů roztažen, aby zabíral celý prostor okna. Další elementy aplikace mají pomocí stylů určené rozměry a jsou rozmístěny v okně.

Změna zobrazení prvků je typicky řešena změnou třídy daného elementu. Pro tuto třídu jsou definovány jiné styly a tomu odpovídá změna vzhledu prvku. Vysouvací menu je řešeno pomocí nastavení hodnoty `transition`, která udává, jak dlouho trvá změna jednoho stylu na jiný.

Barevné schéma stránky je laděno do barev typických pro Škodu Auto: zelené a bílé. Konkrétní RBG hodnoty barev byly zjištěny přímo ze stránek `www.skoda-auto.cz`.

## 6.2 Základní struktura aplikace

Aplikace je dle požadavků firmy Škoda vyvíjena podle jejich C++ aplikace VRUT. Zde popisovaná aplikace tak používá stejné názvosloví. Implementace se také snaží napodobit chování aplikace VRUT, toto se týká hlavně způsobu vykreslování a ovládání navigace ve scéně.

Ve třetí kapitole byla popsána knihovna VRUT převážně z hlediska reprezentace dat scény. Pro implementaci bylo nadále nutné prostudovat další části této knihovny, jako například modul Navigation nebo RenderGlModule.

Aplikace se skládá z následujících tříd:

- AABB – osově zarovnaný obalový kvádr
- Camera – kamera sledující scénu
- GUI – grafické uživatelské rozhraní

- Geometry – definice vrcholů, normál, trojúhelníků
- Light – světlo ve scéně
- Material – definice materiálových vlastností
- Navigation – pohyb a navigace ve scéně
- RenderModule – renderování
- Scene – graf scény s materiály a geometriemi
- ScenNode – uzel scény
- TextParser – zpracování vstupních dat

Dále jsou v aplikaci různé pomocné funkce, převážně pro načítání dat a podobně. Velmi důležitou funkcí je funkce `render()`, která se volá při překreslení scény. Na začátku prací na implementaci aplikace byla většina vykreslovacích příkazů použita právě v této funkci. Po implementaci renderovacího modulu (třída `RenderModule`) se značně zkrátila na pouhé volání funkce `RenderModule.drawScene()`.

Překreslení scény je závislé na čase příštího překreslení obsahu okna prohlížeče. Proto není dobré vykreslovat obraz automaticky. Nejdříve je potřeba požádat prohlížeč o vykreslení. K tomu slouží funkce `requestAnimationFrame()`, které se v parametru předá název „callback“ funkce, kterou prohlížeč vykoná, až bude připraven k vykreslení [6].

Z důvodu zrušení renderovací smyčky je třeba o vykreslení žádat z různých částí aplikace. K tomu je potřeba, aby „callback“ funkce pro `requestAnimationFrame()` byla globální. Proto nelze funkci `render()` úplně zrušit. Bylo by možné jako „callback“ funkci použít přímo `RenderModule.drawScene()`, ale při změně názvu nebo při použití jiné funkce, by bylo nutné všechny výskyty volání `requestAnimationFrame()` upravit. Takto stačí změnit pouze jeden řádek v globální funkci `render()`.

### 6.2.1 Funkce onLoad

Nejdůležitější funkcí aplikace je funkce `onLoad()`, která se spouští po načtení stránky a má na starosti vytvoření WebGL kontextu, načtení dat, vytvoření gui, atd.

Nejdříve je třeba v HTML dokumentu nalézt element `<canvas>`, do kterého budeme kreslit. Nastavíme mu velikost podle velikosti obalujícího divu `<div id="app»`. Pomocí funkce `getContext("webgl")` získáme WebGL kontext. Protože WebGL v základu podporuje pouze buffer elementů o rozsahu typu `unsigned byte` nebo `unsigned short`, což může být pro větší modely nedostatečné, tak potřebujeme použít rozšíření `OES element index uint`. Toto rozšíření umožní použít i větší datový typ `unsigned int`. Rozšíření získáme pomocí funkce `getExtension()`. Zkontrolujeme, zda se podařilo získat kontext i rozšíření a že můžeme pokračovat. Pokud se kontext nepodařilo získat, tak pravděpodobně zařízení nepodporuje WebGL nebo zmíněné rozšíření a zobrazíme informační okénko s chybovou hláškou.

Když máme kontext, můžeme vytvořit potřebné struktury pro aplikaci. Vytvoříme grafické menu, prázdnou scénu, modul navigace a renderovací modul. Pro renderovací modul přeložíme program z shaderů definovaných v HTML hlavičce. Modulu navigace řekneme, že má sledovat události nad elementem `<canvas>`.

Dále je třeba nastavit funkci při změně velikosti okna, která se postará o korektní změnu velikosti canvasu. Funkci `onResize()` nastavíme pomocí `addEventListener()` k objektu `window`. Funkce bere dva parametry: element canvas a obalující element aplikace. Funkce vezme velikost divu s aplikací a nastaví ji jako velikost canvasu. Tímto je zajištěno, že canvas vyplňuje celý vymezený prostor pro aplikaci a je správně vzorkován. Pokud je již načtena scéna, je nutné upravit perspektivní projekční matici kamery a následně překreslit scénu.

Poté zbývá pouze načíst a zpracovat soubor se scénou. Po úspěšném zpracování dat scény ještě inicializujeme renderovací modul `RenderModule` a aplikace je plně spuštěna.

## 6.3 Načtení dat

Soubor s daty scény je načítán asynchronně díky technologii AJAX. V javascriptu stačí vytvořit objekt `XmlHttpRequest`, nastavit mu správně typ a pomocí jeho metody `open()` zadat url a poté zavolat metodu `send()`, která spustí načítání. Poté stačí k události `onload` tohoto objektu nastavit, co se má udělat po kompletním načtení dat. Pokud například url neexistuje, je nutné odchytit výjimku metody `send()`. Chyba může nastat i při přenosu, který tedy nebude dokončen, proto je třeba nastavit k události `onerror` funkci, která tuto situaci ošetří [6].

O všechno toto se v aplikaci postará funkce `loadScene()`, která očekává 3 parametry: url souboru se scénou, objekt grafu scény (třída `Scene`) a callback funkci, která se má vykonat po načtení a zpracování scény. Chybové stavy ošetřuje funkce `AJAXOnError()`, která zobrazí chybovou hlášku.

Pokud je soubor z url načten v pořádku, je zavolána funkce `AJAXOnLoadInitScene()`. Tato funkce vezme odpověď na odeslaný požadavek (načtená data) a předá je dále funkci `parseVrut()`, která data zpracuje a naplní jimi graf scény.

### 6.3.1 VRUT parser

Parser dat ve VRUT formátu se skládá s třídy `TextParser` a funkce `parseVrut()`. Třída `TextParser` obsahuje ukazatel na aktuální pozici v souboru a metody pro čtení různých datových typů. Jádrem parseru je poté právě funkce `parseVrut()`, která řídí zpracování dat a využívá k tomu metody třídy `TextParser`. Zpracovaná data funkce přímo plní do grafu scény (třída `Scene`). Většina kódu tohoto parseru je dílem doktora Míška, který zná formát VRUT a byl tak laskav a dal mi tento parser k dispozici. Já jsem do parseru později doplnil určité potřebné části. Například výpočet osově zarovnaného obalového kvádru, přepočet trojúhelníkových indexů na hranové (pro drátové zobrazení), načítání textur a další.

## 6.4 Třída Scene

Třída `Scene` reprezentuje graf scény. Obsahuje tři pole pro uložení uzlů scény, materiálů a geometrií. Dále obsahuje metody pro vkládání nových uzlů, materiálů nebo geometrií, metody pro práci s těmito objekty a metody pro práci s celým grafem scény nebo jeho podstromem. Graf scény je zde tvořen definovanými potomky a rodičem v jednotlivých uzlech grafu. Kořen stromu je reprezentován speciálním typem uzlu s identifikátorem 0.

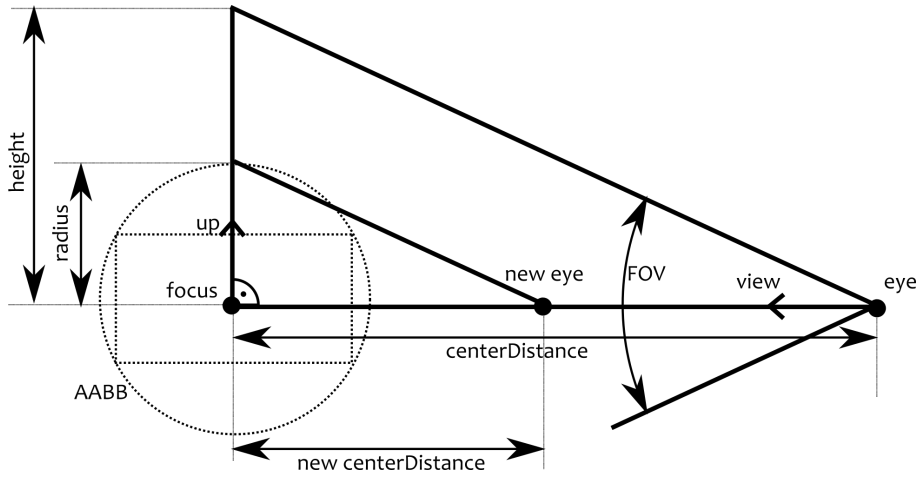
Metody třídy:

- AddGeometry(geometry) – metoda pro vložení nové geometrie. Vstupním parametrem je objekt třídy `Geometry`. Objekt je vložen do seznamu geometrií na správné místo

dle svého identifikátoru `gid`.

- AddMaterial(material) – metoda pro vložení nového materiálu. Vstupním parametrem je objekt třídy `Material`. Objekt je vložen do seznamu materiálů na správné místo dle svého identifikátoru `mid`.
- AddSceneNode(node) – metoda pro vložení nového uzlu grafu scény. Vstupním parametrem je objekt třídy `SceneNode`. Objekt je vložen do seznamu uzlů na správné místo dle svého identifikátoru `id`. Pokud má uzel definovaného rodiče, tak se identifikátor nového uzlu vloží do seznamu potomků svého rodiče. Pokud je rodič uzel typu `switch`, tak je třeba vyřešit aktivitu vkládaného uzlu. Podíváme se na jeho rodiče, jakou hodnotu má parametr `activeChild` a dle toho, zda tato souhlasí s novým uzlem, tak uzel nastavíme jako aktivní nebo neaktivní.
- Fit(camera, nodeID) – nastaví kameru (`camera`), aby se dívala na celý podstrom grafu scény, definovaný uzlem s identifikátorem `nodeID`. Nejdříve spočítá osově zarovnaný obalový kvádr celého podstromu pomocí funkce `Scene.GetAABB()`. Pomocí metody `Camera.lookAt()` otočí kameru, aby směřovala na střed spočítaného kvádru. Podle trojúhelníkové podobnosti spočítá, jak je třeba kameru oddálit nebo přiblížit, aby byl celý podstrom vidět (Obrázek 6.1). Na základě rozměrů obalového kvádru nastaví blízkou a vzdálenou rovinu kamery. Poté už zbývá upravit perspektivní projekční matici kamery (`Camera.updatePersective()`) a překreslit scénu.
- GetAABB(nodeID) – spočítá osově zarovnaný obalový kvádr podstromu grafu scény určeného uzlem s identifikátorem `nodeID`. Vytvoří si nový objekt třídy `AABB`. Prochází postupně celý strom/podstrom scény a počítá transformační matice uzlů. Pokud je aktuální uzel typu `Geometry`, tak vezme jeho obalový kvádr, transformuje jeho okrajové body podle aktuální transformační matice a rozšíří jimi svůj obalový kvádr. Po průchodu celým podstromem spočítá střed a poloměr obalového kvádru (ve výsledku tím dostaneme obalovou kouli) a vrátí objekt tohoto obalového kvádru.
- GetNodesAABB(node) – vrací objekt osově zarovnaného obalového kvádru daného uzlu.
- GetNodesTransformation(nodeID) – spočítá transformační matici uzlu s id `nodeID`, pro převod modelových souřadnic do světových. Vezme lokální matici uzlu a zleva ji vynásobí lokální maticí rodiče. Takto ji postupně násobí maticemi dalších předků, dokud nedojde do kořene stromu. Výslednou matici vrátí.
- GetSceneNode(nodeID) – vrací uzel grafu scény specifikovaný identifikátorem `nodeID`.
- GetSceneNodesByType(type) – vrací pole uzlů se zadaným typem. Projde seznam všech uzlů scény a pokud typ uzlu odpovídá zadanému, tak jej uloží do pole, které na konci vrací.
- IsNodeReflective(node) – test zda je materiál daného uzlu odrazivý.
- IsNodeTransparent(node) – test zda je materiál daného uzlu průhledný.
- SetActive(nodeID, active) – nastaví uzlu s id `nodeID` aktivitu.
- SetGeometry(nodeID, geometryID) – nastaví uzlu s id `nodeID` geometrii s id `geometryID`.

- SetGeometry(nodeID, light) – nastaví uzlu s id `nodeID` světlo `light` = objekt třídy `Light`. Uzel by měl být pouze typu `light`.
- SetMaterial(nodeID, materialName) – nastaví uzlu s id `nodeID` materiál s názvem `materialName`.
- SetSwichActiveChild(nodeID, activeChild) – nastaví uzlu typu `switch` s id `nodeID` aktivního potomka. Projde všechny potomky, nastaví jim vlastnost `active` na hodnotu `false`, pouze pro daného potomka ji nastaví na `true`. Pokud je parametr `activeChild` roven `-1`, tak všechny potomky deaktivuje. Pokud je parametr `activeChild` roven `-2`, tak všechny potomky naopak aktivuje.
- SetTransformation(nodeID, localMatrix) – nastaví uzlu s id `nodeID` lokální transformační matici `localMatrix`.




Obrázek 6.1: Schéma výpočtu přiblížení kamery v metodě Scene.Fit(), pohled z boku

### 6.4.1 Třída Material

Třída `Material` reprezentuje materiál použitý pro zobrazení objektů ve scéně. Neobsahuje žádné metody, pouze informace o vlastnostech materiálu.

Parametry třídy:

- mid – identifikátor materiálu
- name – název materiálu
- ambient – ambientní barva
- diffuse – difuzní barva
- specular – barva odlesků
- emission – vyzařující barva
- shininess – lesklost materiálu

- reflectivity – odrazivost materiálu
- imageName – relativní cesta k obrázku textury

Třída `Material` obsahuje další parametry, které vycházejí z formátu VRUT, v této aplikaci však nejsou využity. Jedná se například o nastavení textur nebo vlastní shadery.

Pokud je parametr `Material.imageName` neprázdný, znamená to, že materiál má definovanou texturu. Tuto texturu je nutno vytvořit a naplnit daty obrázku. K tomuto jsou určené funkce `loadTexture()` a `handleTextureLoaded()`. První funkce vytvoří WebGL texturu a spustí asynchronní nahrávání obrázku. Druhá funkce se spustí po úspěšném načtení obrázku a naplní jeho data do textury. Podrobnější popis, co dál tato funkce řeší, je v kapitole o texturách.

### 6.4.2 Třída Geometry

Třída `Geometry` reprezentuje geometrii objektu ve scéně. Taktéž nemá žádné metody, obsahuje většinou WebGL buffery s příslušným obsahem.

Parametry třídy:

- gid – identifikátor geometrie
- vertices – WebGL buffer vrcholů
- normals – WebGL buffer normál
- texCoords – WebGL buffer texturovacích souřadnic
- indices – WebGL buffer indexů trojúhelníků
- indicesLength – počet indexů trojúhelníků
- indicesWF – WebGL buffer indexů hran
- indicesWFLength – počet indexů hran
- firstTri – vrcholy prvního trojúhelníka
- firstNorm – normála prvního trojúhelníka,
- aabb – osově zarovnaný obalový kvádr, objekt třídy `AABB`

Z důvodu úspory místa nejsou při zpracování dat ukládány data jednotlivých bufferů. Data jsou pouze nahrána do bufferu na grafické kartě. Pro další funkčnost je ale potřeba uložit alespoň některá data. Pro odrazivé objekty je důležitý první trojúhelník a jeho normála. Pro drátové zobrazení je potřeba při zpracování přepočítat trojúhelníkové indexy na hranové indexy. Takto stačí pro drátové zobrazení použít ve funkci `DrawElements()` místo módu `TRIANGLES` mód `LINES` a připojit správný element buffer.

### 6.4.3 Třída AABB

Třída `AABB` reprezentuje osově zarovnaný obalový kvádr. Obsahuje minimální a maximální souřadnice v podobě dvou bodů. Navíc obsahuje také souřadnice středu a poloměr, čímž může reprezentovat také obalovou kouli.

Metody třídy:

- SetInvalid() – nastavuje inicializační hodnoty mezí, středu a poloměru.
- ExpandWithPoint(x, y, z) – rozšíří meze obalového kvádru o bod se souřadnicemi `x`, `y`, `z`. Střed ani poloměr tato metoda neupravuje.
- ComputeCenter() – vypočítá střed a poloměr obalové koule. Metoda by měla být volána vždy až po konečném rozšíření mezí. Střed je vypočítán jako lineární interpolace mezi oběma mezemi obalového kvádru. Poloměr je poté vzdálenost středu od jedné z mezí.

Protože třída Geometry neukládá souřadnice vrcholů geometrie, a z WebGL bufferů se nedá číst zpět, je nutné tento objekt naplnit již při zpracování vstupních dat. Pro novou geometrii je vytvořen nový objekt pomocí `SetInvalid()`. S každým nově načteným bodem geometrie je také zavolána metoda `ExpandWithPoint()`. Po načtení celé geometrie se ještě zavolá metoda `ComputeCenter`.

## 6.5 Třída SceneNode

Třída `SceneNode` reprezentuje uzel grafu scény. Tato třída také nemá žádné metody, uchovává pouze informace o uzlu scény.

V knihovně VRUT se uzly mohou lišit svým typem, přičemž každý typ je definován ve své vlastní podtřídě. Po konzultaci s doktorem Míškem bylo domluveno, že je to pro tuto aplikaci zbytečné. Vzhledem k implementovaným typům uzlů, které se vzájemně liší jen v pár parametrech, to není potřeba řešit. Uzly jsou stále rozlišeny typem, ale jejich parametry jsou sloučené do jedné třídy. Určitý typ uzlu má vždy nastavené pouze potřebné parametry, ostatní jsou s implicitními hodnotami, případně nedefinované.

Podporované typy uzlů jsou následující:

- 0 ASSEMBLY – agregační uzel
- 1 FILE_ROOT – uzel kořene stromu
- 2 CAMERA – uzel kamery
- 3 GEOMETRY – uzel s geometrií
- 4 LIGHT – uzel se světlem
- 6 SWITCH – přepínací uzel
- 7 BACKGROUND – uzel pozadí

Parametry třídy:

- id – identifikátor uzlu

- name – název uzlu
- type – typ uzlu, číslo 0–8
- parentID – identifikátor přímého předka uzlu
- active – příznak aktivity uzlu
- children – pole potomků uzlu
- gid – identifikátor geometrie, pouze pro uzly typu 3
- mid – identifikátor materiálu, pouze pro uzly typu 3 a 7
- localMatrix – lokální transformační matice
- light – světlo, objekt třídy `Light`, pouze pro uzly typu 4
- activeChild – pořadové číslo aktivního potomka, pouze pro uzly typu 6
- camera – kamera, objekt třídy `Camera`, pouze pro uzly typu 2

### 6.5.1 Třída Light

Třída `Light` reprezentuje světlo ve scéně. Také nemá žádné metody, uchovává pouze informace o typu a vzhledu světla.

Parametry třídy:

- type – typ světla
- diffuse – barva světla
- attenuation – útlum světla

Třída `Light` obsahuje další parametry, které vycházejí z formátu VRUT, v této aplikaci však nejsou využity. Jedná se například o úhel a intenzitu světla nebo IES soubor s definicí světla.

## 6.6 Třída Camera

Třída `Camera` reprezentuje kameru, která snímá scénu. Kamera je přímo součástí grafu scény (`ScenNode` typ 2). Díky tomu je možné kameru vložit pod určitý uzel stromu a tento uzel tak kamerou sledovat. Chování kamery je inspirované chováním kamery z knihovny VRUT, ale je více oddělena od scény. Objekt třídy `Camera` lze takto využít i mimo graf scény.

Třída `Camera` obsahuje různé parametry, které určují pozici, orientaci a vlastnosti kamery:

- fov – zorný úhel kamery
- nearPlane – blízká rovina kamery
- farPlane – vzdálená rovina kamery

- width – šířka projekční roviny kamery
- height – výška projekční roviny kamery
- center – střed rotace kamery
- focus – bod na který se „dívá“ kamera; počátek souřadnic prostoru kamery
- eye – pozice kamery ve světových souřadnicích
- up – normalizovaný up vektor kamery určuje vertikální osu kamery
- view – normalizovaný pohledový vektor kamery, směr od `Camera.eye` ke `Camera.focus`, je kolmý na up vektor
- centerDist – vzdálenost kamery od bodu `Camera.focus`
- vmatrix – pohledová matice kamery, transformuje vrcholy ze světových souřadnic do souřadnic prostoru kamery
- pmatrix – projekční matice kamery, matice simulující chování takzvané „pinhole“ kamery, je určena rozměry projekční roviny, zorným úhlem a blízkou a vzdálenou rovinou

Třída `Camera` dále obsahuje několik metod pro manipulaci s kamerou a práci s jejími maticemi. Většina těchto metod je ve VRUTu součástí modulu `Navigation`. Zde jsou přesunuty do třídy `Camera` z důvodu oddělení kamery od scény.

Metody třídy:

- getCenterDist() – metoda vrací vzdálenost kamery od bodu `focus`.
- updatePerspective() – metoda přepočítá novou perspektivní projekční matici. Tato metoda je volána vždy, když je změněna hodnota některého z těchto parametrů: `Camera.fov`, `Camera.width`, `Camera.height`, `Camera.nearPlane` nebo `Camera.farPlane`. Při změně velikosti okna prohlížeče je třeba získat a nastavit nové rozměry projekční roviny. Tyto rozměry získáme z rozměrů canvasu. Poté použijeme funkci `mat4.perspective()`, která z aktuálních parametrů kamery vytvoří perspektivní projekční matici.
- lookAt(newFocus) – metoda přesune kameru, tak aby se „dívala“ na nový bod `newFocus`. Spočítá vektor posunu starého bodu `Camera.focus` do nového bodu `newFocus`. O tento vektor poté posune body `Camera.focus` a `Camera.eye`. Pohledový a up vektor kamery nejsou ovlivněny a pomocí funkce `mat4.lookAt` je spočítána nová pohledová matice kamery. Projekční matice není ovlivněna a není třeba ji měnit.
- pan(dx, dy) – metoda pro posun kamery v osách x a y. Tyto osy nejsou osy světového prostoru, ale spíše osy kamery transformované do světového prostoru. Osy jsou tedy reprezentovány up vektorem a bočním vektorem. Posun v ose y získáme vynásobením up vektoru velikostí posunu `dy`. Pro posun v ose x potřebujeme nejdříve boční vektor kamery. Ten získáme vektorovým součinem up vektoru a pohledového vektoru. Takto získaný vektor by měl být již normalizovaný. Vynásobíme ho velikostí posunu `dx`. Nyní jsme získali dva vektory pro posun v jednotlivých osách kamery. Oba vektory sečteme a vyjde nám jeden vektor posunu. O tento vektor poté posuneme body `Camera.focus` a `Camera.eye` a pomocí funkce `mat4.lookAt` spočítáme novou pohledovou matici kamery. Projekční matice není ovlivněna a není třeba ji měnit.

- zoom(dz) – metoda pro přiblížení nebo oddálení kamery. Tato metoda v podstatě posunuje kameru ve směru pohledového vektoru dopředu a dozadu. Parametr `dz` určuje, o kolik se má kamera posunout. Pro posunutí ve správném směru vynásobíme pohledový vektor velikostí posunu `dz`. Tímto vektorem posuneme bod `Camera.eye`, ostatní parametry není třeba měnit. Opět pomocí funkce `mat4.lookAt` spočítáme novou pohledovou matici kamery a upravíme proměnnou vzdálenosti kamery od středu `Camera.centerDist`.

- rotate(ax, ay) – metoda pro rotaci kamery kolem středu `Camera.center`. Parametry `ax` a `ay` určují úhel rotace v radiánech. Pro tuto rotaci potřebujeme dvě rotační matice pro rotaci kolem x-ové a y-ové osy kamery. Osa y je prostý up vektor kamery, osu x získáme opět vektorovým součinem up vektoru a pohledového vektoru. Dvojitým použitím funkce `mat4.rotate()` vypočítáme výslednou rotační matici. Touto maticí transformujeme up vektor `Camera.up` a pohledový vektor `Camera.view`. Pro transformaci bodů `Camera.eye` a `Camera.focus` musíme vzít v úvahu, že střed rotace `Camera.center` nemusí ležet v počátku souřadného systému kamery. Musíme proto vypočítat vektor posunu z počátku do bodu `Camera.center`. Poté vytvoříme novou matici rotace jako jednotkovou matici. Tuto matici vynásobíme translační maticí vytvořenou z vektoru posunu. Teprve teď ji vynásobíme původní rotační maticí. Následně ji vynásobíme translační maticí vytvořenou z obráceného vektoru posunu. Výslednou maticí již můžeme transformovat body `Camera.eye` a `Camera.focus`. Poté přepočítáme novou pohledovou matici (`mat4.lookAt()`). Takto dosáhneme rotace kamery kolem určitého bodu ve scéně místo rotace kolem středu kamery.

Na začátku implementace byla uvažována také metoda `Camera.roll()` pro rotaci kamery kolem z-ové osy, ale nakonec nebyla potřeba. V kódu je však ponechána zakomentovaná rozpracovaná verze, pokud by byla v budoucnu potřeba.

## 6.7 Třída Navigation

Třída `Navigation` se stará o ovládání kamery pomocí myši nebo dotyku. Tato třída svým účelem odpovídá modulu `Navigation` v knihovně VRUT. Ve VRUTu je vztah mezi modulem `Navigation` a třídou `Camera` trochu jiný. Modul `Navigation` totiž řeší více způsobů navigace inspirovaných různými 3D nástroji (Catia, Cinema 4D, Maya, apod.). Tato třída `Navigation` řeší pouze navigaci ve stylu aplikace VRUT, což odpovídá třídě `Navigation::NavigationStyleVRUTSpecific`. Zde jsou tedy jasně odděleny funkce kamery a funkce navigace, proto jsou některé metody modulu `Navigation` přesunuty oproti VRUTu do třídy `Camera`.

Implementace se také značně liší z důvodu různého zpracování událostí v jazyce Javascript oproti C++. Zde je navíc řešeno dotykové ovládání, které v knihovně VRUT není přítomno. Třída `Navigation` tedy obsahuje parametry pro úpravu chování navigace a parametry uchovávající informace nutné ke správnému fungování dotykového ovládání i ovládání myší.

Parametry třídy:

- camera – kamera; objekt třídy `Camera` s kterým navigace manipuluje
- rotSpeed – rychlost rotace kamery
- panSpeed – rychlost posunu kamery

- zoomSpeed – rychlost přibližování a oddalování kamery
- speedModifier – koeficient rychlosti, je závislý na vzdálenosti kamery od středu
- rotX, rotY – rotace v osách X a Y
- panX, panY – posun v osách X a Y
- mouseX, mouseY – předchozí souřadnice myši na obrazovce v osách X a Y
- leftDrag, middleDrag, rightDrag – proměnné pro uložení stisknutých tlačítek myši
- leftDragOut, middleDragOut, rightDragOut – proměnné pro uložení stavu, když ukazatel myši vyjede mimo okno aplikace
- ongoingTouches – seznam aktuálních dotyků
- dblTapTimer – časovač dvojitého dotyku
- pressTap – proměnná pro stav dotykového gesta „press-and-tap“
- pressTapTimer – časovač gesta „press-and-tap“

Třída obsahuje dvě obecné metody pro inicializaci:

- initHandlers(canvas) – metoda nastaví potřebné ovladače událostí pro element specifikovaný v parametru `canvas`. Pokud zařízení podporuje dotykové ovládání, tak nastaví i ovladače dotykových událostí, jinak nastaví pouze ovladače událostí myši.
- changeCamera(camNode) – metoda nastaví s jakou kamerou bude třída manipulovat. V parametru očekává uzel grafu scény, který je typu 2, tedy objekt třídy `SceneNode` a nikoli `Camera`. Po uložení kamery je třeba zjistit vzdálenost kamery od středu a upravit podle toho parametr `Navigation.speedModifier`. Poté je možné překreslit scénu.

Třída `Navigation` implementuje ovládání šesti základních úkonů. V následujícím seznamu jsou uvedeny tyto úkony včetně příslušných ovládacích gest myši a dotykových gest [10]:

- rotace – tažení levým tlačítkem; tažení jedním prstem („drag“ )
- posun – tažení pravým tlačítkem; tažení dvěma prsty („two-finger drag“)
- zoom – kolečko myši nebo tažení prostředním tlačítkem; stažení/roztažení 2 prsty („pinch/spread“)
- výběr objektu – dvojklik levým tlačítkem; dvojdotyk jedním prstem („double tap“)
- výběr středu rotace – dvojklik pravým tlačítkem; dotykové gesto není
- výběr středu rotace a vycentrování – dvojklik prostředním tlačítkem; podržení jedním prstem a dotyk druhým prstem („press and tap“)

Způsob implementace těchto úkonů bude popsán v následujících podkapitolách.

### 6.7.1 Ovládání myší

Implementace ovládání myší v Javascriptu spočívá v definici funkcí (takzvaných „handlerů“ nebo ovladačů), které se mají vykonat vždy, když nastane určitá událost. Události, které mohou nastat, jsou předem určené a známé, například: `onclick`, `onmousedown`, `onwheel`, a podobně. Pro správnou funkčnost je třeba určit, nad kterým elementem stránky se mají události sledovat a jaká funkce se má spustit při které události. K tomu slouží funkce HTML elementu `addEventListener()`. O toto nastavení se stará metoda `Navigation.initHandlers()`, která byla popsána dříve.

- rotace – při stisknutí levého tlačítka myši se vyvolá událost `mousedown` a v jejím ovladači se zaznamenají souřadnice myši do proměnných `mouseX` a `mouseY`. Také se změní hodnota proměnné `leftDrag` na `true`. Při pohybu myši se vyvolá událost `mousemove` v jejímž ovladači se otestuje tato hodnota, a pokud je nastavena na `true`, tak se z aktuálních a předchozích souřadnic myši vypočítá relativní změna. Tato změna představuje rotaci ve stupních. Pro převod na radiány slouží funkce `Navigation.toRadians()`. Hodnota v radiánech je ještě vynásobena proměnnou `rotSpeed`. Výsledné úhly rotací předáme funkci `Camera.rotate()` a následně překreslíme scénu. Tento výpočet se opakuje tak často, jak je prohlížeč schopen zpracovávat vstup z myši. Po uvolnění tlačítka se ještě spustí událost `mouseup`, která změní hodnotu proměnné `leftDrag` na `false`.

- posun – základní koncept zpracování této akce je totožný s rotací. Pouze s tou změnou, že ovladače reagují na stisk pravého tlačítka a pracují s proměnnou `rigthDrag`. Jednotlivá tlačítka jsou rozlišena díky proměnné `button` objektu události. Výpočet hodnot posunu je poté během události `mousemove` podobný. Relativní změna souřadnic se jen vynásobí proměnnou `panSpeed` a také `speedModifier`, čímž docílíme pomalejšího posunu při větším přiblížení a naopak. Výsledný posun v obou osách předáme funkci `Camera.pan()` a překreslíme scénu.

- zoom – tato akce má dvě různá řešení. První je shodné s rotací a posunem. Ovladače reagují na stisk prostředního tlačítka. Pro změnu přiblížení se používá pouze relativní změna v ose y. Tato je opět vynásobena `zoomSpeed` a `speedModifier`. Výslednou změnu přiblížení předáme funkci `Camera.zoom()` a překreslíme scénu. Protože tímto měníme vzdálenost kamery od středu, je třeba aktualizovat proměnnou `speedModifier`. Druhým řešením je místo tažení myši rolování kolečkem. Při rolování kolečkem myši se vyvolá událost `mousewheel` a v jejím objektu se nachází proměnná `deltaY`, která určuje, o kolik jednotek se posunulo kolečko. Zde je potřeba dávat pozor, v jakých jednotkách je tento posun udáván. To lze zjistit z proměnné `deltaMode`, která se u různých prohlížečů liší. Výpočet změny přiblížení je stejný jako při předchozím řešení, pouze hodnota `deltaY` je vynásobena koeficienty podle proměnné `deltaMode`, aby byla zajištěno stejné chování napříč prohlížeči. Opět zavoláme funkci `Camera.pan()`, překreslíme scénu a aktualizujeme proměnnou `speedModifier`.

- výběr geometrie – pro tuto akci je použit dvojklik levým tlačítkem. Pro toto by šlo využít události `dblclick`, ta je však vyvolána pouze po dvojkliku levým tlačítkem. U dalších akcí budeme potřebovat detekovat i dvojklik ostatními tlačítky. Proto zvolíme obecnější řešení. Objekt všech událostí myši obsahuje proměnnou `detail`, která udává, kolikrát bylo na myš kliknuto. Toto funguje již i pro ostatní tlačítka. Ovladač nastavíme na událost `mouseup`, aby se náš kód spustil až po dokončení dvojkliku. Při uvol-

není tlačítka myši tedy zkontrolujeme, zda má proměnná `detail` hodnotu 2 (dvojklik). Poté se zavolá funkce `RenderModule.pickingDraw()`, které předáme souřadnice myši. Tato funkce vrátí identifikátor uzlu s geometrií, na kterou bylo kliknuto. Funkcí `Scene.GetSceneNode()` získáme uzel s daným identifikátorem. Pro výběr daného uzlu a jeho případné zobrazení zavoláme funkci `RenderModule.setSelectedNode()`.

- výběr středu rotace – tato akce je vyvolána dvojklikem pravým tlačítkem. Řešení je totožné jako u předchozí akce až po získání uzlu grafu scény. Pro zjištění přesného bodu, na který bylo kliknuto, by bylo potřeba provést další renderovací průchod a vykreslit nějakým způsobem očíslované trojúhelníky. Z těchto bodů vybrat ten správný a s ním vypočítat průnik paprsku vrženého z kamery skrze projekční plochu v místě kliknutí. Toto řešení bylo po konzultaci s doktorem Míškem zavrženo jako výpočetně náročné. Místo toho je z vybraného uzlu získán obalový kvádr a jeho střed je použit jako střed rotace. Toto řešení je pro většinu případů dostatečné a značně zjednodušuje výpočet. Obalový kvádr uzlu získáme funkcí `Scene.GetNodesAABB()`. Tento kvádr je však definován v modelových souřadnicích, proto ještě potřebujeme získat transformační matici pro převod z modelových do světových souřadnic. K tomuto slouží funkce `Scene.GetNodesTransformation()`. Transformované souřadnice středu poté nastavíme jako nový střed rotace kamery `Camera.center`.

- výběr středu rotace a vycentrování – tato akce je vyvolána dvojklikem prostředním tlačítkem. Řešení je shodné s předchozí akcí, pouze je zde navíc vycentrování scény na obrazovce. Po nastavení středu rotace stačí zavolat funkci `Scene.lookAt()`, která přesune kameru tak, aby byl vybraný střed uprostřed obrazu. Protože je tímto pohnuto s kamerou je třeba znovu překreslit scénu.

Při vyjetí myši z prostoru aplikace je třeba ošetřit situaci, že událost `mouseup` se nespustí nad správným elementem. Po navrácení myši zpět je v aplikaci stále nastaveno, že tlačítko je stisknuté. Proto je třeba sledovat událost `mouseout`, při které si zapamatujeme stavy stisknutých tlačítek a objektu `window` nastavíme při události `mouseup` stejnou funkci, jako je u canvasu. Takto je tato událost zaznamenána i mimo prostor aplikace. Při návratu myši se spustí událost `mouseenter`, při které odebereme z objektu `window` ovladač události `mouseup` a navrátíme správné hodnoty stisknutých tlačítek.

### 6.7.2 Dotykové ovládání

Základní princip implementace dotykového ovládání je stejný jako u ovládání myší, pouze se používají jiné události. Rozdíl je v tom, jaké události jsou k dispozici. V Javascriptu existují v zásadě tyto tři dotykové události: `touchstart`, `touchmove` a `touchend`. Dalším rozdílem je, že myš je jen jedna, ale dotyků může být více. Z tohoto důvodu je třeba uchovávat seznam aktuálních dotyků a při událostech kontrolovat, které dotyky se změnily.

Implementace dotykových gest se vzájemně liší, pro gesta tažení stačí sledovat počet dotyků a při události `touchstart` zjistit posun. Pro gesta „double tap“ a „press and tap“ je třeba nastavit časovače pro testování, zda byl dotyk ukončen dostatečně rychle a rozlišit tak jednoduchý dotyk(„tapnutí“) od tažení. Princip časovače je následovný: nastavíme nějaký příznak a s ním spustíme časovač, po vypršení časovače příznak zase zrušíme. Pokud tedy nějaká událost nastane včas, je příznak nastaven; pokud však událost nastane později, je již příznak zrušen.

- touchstart – tato událost se spustí při přiložení prstu na obrazovku. Zde je nutné uložit nové dotyky do pole. K tomuto je použita funkce `copyTouch()`, která do pole dotyků uloží potřebná data: identifikátor a souřadnice [6]. Pokud byl v poli pouze jeden dotyk a nyní přibyl jeden nový a je nastaven příznak `pressTap`, tak nastavíme časovač pro gesto „press and tap“. Pokud žádný dotyk nebyl a teď jeden přibyl, spustíme časovač dvojitého dotyku a nastavíme příznak pro gesto „press and tap“. Pokud byl časovač již nastaven, ale nevypršel, tak jsme zaznamenali gesto „double tap“. Zavoláme funkci `RenderModule.pickingDraw()`, které předáme souřadnice dotyku a získáme identifikátor uzlu, kterého se uživatel dotkl. Funkcí `Scene.GetSceneNode()` získáme uzel s daným identifikátorem. Zavoláme funkci `RenderModule.setSelectedNode()` a nastavíme vybraný uzel.
- touchmove – tato událost se spustí při pohybu prstu po obrazovce. Nejdříve zrušíme příznak `pressTap` pro ukončení gesta „press and tap“. Pokud je v poli aktivních dotyků pouze jeden dotyk, tak probíhá gesto „drag“, tedy musíme rotovat kamerou. V poli dotyků je uložena předchozí pozice dotyku, v objektu této události je zase uložena aktuální pozice dotyku, z těchto informací zjistíme relativní posun. Výpočet úhlů rotace a další postup je potom stejný jako při ovládání myší. Pokud jsou v poli dva aktivní dotyky, jedná se o gesta „two-finger drag“ a „pinch/spread“. Tyto gesta nejsou pevně oddělena. Nejdříve najdeme souhlasné dotyky mezi aktuálními a uloženými. K tomu slouží funkce `ongoingTouchIndexById()`, díky které zjistíme, které dotyky patří k sobě podle jejich identifikátorů [6]. Nyní můžeme pro oba dotyky vypočítat jejich relativní posun. Zprůměrováním posunů obou dotyků získáme výsledný posun, který vynásobíme proměnnými `panSpeed` a `speedModifier` a výsledek použijeme ve funkci `Camera.pan()`. Rozdílem vzdáleností původních dotyků a nových dotyků zjistíme, o kolik přiblížit nebo oddálit kameru. Tuto hodnotu taktéž vynásobíme proměnnými `zoomSpeed` a `speedModifier` a výsledek použijeme ve funkci `Camera.zoom()`. Na závěr této události aktualizujeme uložené dotyky.
- touchend – tato událost se spustí, když je prst zvednut z obrazovky. Pokud je nastaven časovač gesta „press nad tap“, v poli aktuálních dotyků jsou dva dotyky a byl zvednut druhý prst, tak se jedná o gesto „press and tap“ a akci výběru středu rotace a vycentrování. Zavoláme funkci `RenderModule.pickingDraw()`, které předáme souřadnice druhého dotyku („tapnutí“) a získáme identifikátor uzlu, kterého se uživatel dotkl. Funkcí `Scene.GetNodesAABB()` získáme obalový kvádr uzlu s daným identifikátorem a dále pokračujeme stejně jako při ovládání myší a vybereme střed rotace a vycentrujeme scénu. Na závěr události odebereme všechny ukončené dotyky z pole aktuálních dotyků.

## 6.8 Třída RenderModule

Třída `RenderModule` má na starosti vykreslování scény. Jak název napovídá, tak třída odpovídá modulu `RenderGLModule` v knihovně VRUT. Princip vykreslování je prakticky totožný s implementací v knihovně VRUT s tím, že některé části jsou zjednodušeny (vykreslování odrazivých objektů), přepracovány (zobrazení pozadí), nebo nově přidány (výběr geometrie a její zvýraznění).

Třída obsahuje řadu různých parametrů, které uchovávají informace nutné k vykreslení nebo řídí způsob vykreslení. Mnohé další parametry jsou vytvořeny až dynamicky podle potřeby (lokace uniforem a atributů shaderů apod.).

V následujícím seznamu jsou uvedeny základní parametry třídy `RenderModule`:

- gl – renderovací kontext WebGL
- scene – scéna; objekt třídy `Scene`, který chceme zobrazovat
- program – realistický shader, WebGL program pro grafickou kartu
- program2 – stylizovaný shader, WebGL program pro grafickou kartu
- activeProgram – identifikace aktivního programu
- camera – uzel s kamerou; objekt třídy `SceneNode`
- cameraPosition – pozice kamery ve světových souřadnicích
- viewMatrix – pohledová matice kamery
- perspMatrix – perspektivní projekčni matice kamery
- lights – seznam světel ve scéně
- activeLight – index aktivního světla
- lightPosition – pozice světla ve světových souřadnicích
- backgrounds – seznam pozadí
- defaultBackground – barva výchozího pozadí
- activeBackground – index aktivního pozadí
- defaultMaterial – výchozí materiál
- shadowQueue – fronta uzlů při průchodu scény
- itemsList – seznam neprůhledných objektů
- blendItemsList – seznam průhledných objektů
- reflectiveItemsList – seznam odrazivých objektů
- backgroundItemsList – seznam objektů ve scéně
- selectedItem – seznam vybraných objektů
- selectedNode – vybraný uzel v grafu scény
- drawSelected – přepínač zvýraznění vybraného objektu
- wireframe – přepínač drátového zobrazení
- lastRenderTime – čas posledního vykreslení

Třída využívá několika obecných funkcí, které nejsou součástí třídy, ale jsou definovány globálně. Tyto funkce mají smysl i při použití mimo tuto třídu. Protože jsou převážně využívány právě touto třídou, tak jsou popsány zde:

- isMat4Unit() – funkce testuje zda je matice jednotková. Jednotková matice má na hlavní úhlopříčce jedničky a jinde nuly. Test se používá při násobení matic, kdy při násobení jednotkovou maticí není druhá matice změněna a tudíž není potřeba násobit.

- isMatrixMirror() – funkce testuje, zda matice obsahuje transformaci zrcadlení. Test se využívá při ořezání zadních stěn, protože při zrcadlení se mění pořadí vrcholů trojúhelníků z „proti směru hodinových ručiček“ na „ve směru hodinových ručiček“. Test se provádí podle rovnice 6.1, kde symbol $m_{xy}$ značí hodnotu matice na řádku x a ve sloupci y.

$$((m_{11}, m_{12}, m_{13}) \times (m_{21}, m_{22}, m_{23})) \cdot (m_{31}, m_{32}, m_{33}) < 0.0 \tag{6.1}$$

  Výpočet levé strany nerovnice je ekvivalentní výpočtu determinantu matice $3 \times 3$ .

- programFromHTMLScripts() – tato funkce vytvoří WebGL program. V parametrech je třeba předat identifikátory elementů `<script>` které obsahují kód vertex a fragment shaderů. Funkce vytvoří příslušné shadery, naplní je zdrojovým kódem ze zadaných elementů `<script>` a zkompiluje tyto shadery. Zkompilované shadery následně slinkuje do nového WebGL programu. Tento program pot vrací jako návratovou hodnotu.

- programFromJSVariables() – tato funkce vytvoří WebGL program. Chování této funkce je stejné jako `programFromHTMLScripts()`, pouze zdrojový kód shaderů vezme ze zadaných javascriptových proměnných. Tyto proměnné musí obsahovat zdrojový kód zapsaný ve formě textového řetězce.

### 6.8.1 Inicializace modulu

Po vytvoření nového objektu této třídy musíme tento objekt inicializovat. Nejdříve je třeba zkompilovat a slinkovat potřebné shader programy. Když máme načtenou a zpracovanou scénu, tak ji přiřadíme objektu `RenderModule`, aby ji začal vykreslovat. K tomuto slouží metoda `initScene()`, která volá další metody, které se postarají o správnou inicializaci:

- initScene() – metoda pro inicializaci renderovacího modulu. V parametru očekává scénu (objekt třídy `Scene`), kterou má modul vykreslovat. Metoda dále vytvoří novou kameru a hlavní světlo („headlight“) a vloží je do scény. Dále najde všechna světla a pozadí ve scéně a nastaví aktuálně používané světlo a pozadí. Poté inicializuje shadery. Na závěr správně nastaví kameru, aby byla vidět celá scéna (`Scene.Fit()`) a předá kameru modulu navigace (`Navigation.changeCamera()`).

- initCamera() – metoda vytvoří novou kameru. Nejdříve vytvoří nový uzel scény `SceneNode`, nastaví mu identifikátor, název, a typ. Poté vytvoří novou kameru (třída `Camera`) a vloží ji do vytvořeného uzlu. Tento uzel následně vloží do scény pomocí funkce `Scene.AddSceneNode()`. Zároveň nastaví tento uzel do proměnné `RenderModule.camera`. Ukládá se celý uzel `SceneNode` a nejenom kamera `Camera`, aby se uchovala reference přímo do grafu scény. To je vhodné pro výpočet transformace kamery.

- initHeadlight() – metoda vytvoří nové světlo. Toto světlo je svázané s kamerou a funguje jako takzvaná „čelovka“. Scéna je tak vždy nasvícena z pohledu kamery. Nejdříve se vytvoří nový uzel grafu scény `SceneNode`, kterému se nastaví identifikátor, název

a typ. Důležité je také správně nastavit identifikátor rodiče, v tomto případě identifikátor kamery `RenderModule.camera.id`. Tímto zajistíme, že uzel světla bude vložen do grafu pod uzel kamery a bude tak transformován spolu s kamerou. Dále se vytvoří nové světlo `Light` a nastaví se mu typ, barva a tlumení. Světlo se vloží do uzlu a celý uzel je vložen do grafu scény (`Scene.AddSceneNode()`).

- findLights() – metoda nalezne všechny uzly světel ve scéně a uloží je do seznamu `RenderModule.lights`. Metoda je jednoduchá stačí zavolat metodu `Scene.GetSceneNodesByType()`, která nám vrátí přímo pole uzlů s daným typem, v tomto případě je to typ 4.

- switchLight() – metoda přepíná index aktuálního světla mezi všemi světly ve scéně. V parametru očekává index světla, které se má stát aktuálním. Index musí být v rozsahu 0 – N-1, kde N je počet světel ve scéně. Při použití metody v inicializaci použijeme index N-1, protože to je index nově vytvořeného světla z metody `InitHeadlight()`.

- findBackgrounds() – metoda nalezne všechny uzly pozadí ve scéně a uloží je do seznamu `RenderModule.backgrounds`. Zavoláme metodu `Scene.getSceneNodesByType()` s parametrem 7 (typ uzlu pozadí), která nám vrátí přímo pole uzlů s pozadími.

- switchBackground() – metoda přepíná index aktuálního pozadí mezi všemi pozadími ve scéně. Parametr metody musí být v rozsahu 0 – N-1, kde N je počet světel ve scéně, a určuje index nového aktuálního pozadí.

- initShaders() – metoda inicializuje potřebné proměnné pro uložení lokací parametrů WebGL programu. Jedná se o lokace atributů a uniforem. K nalezení těchto lokací jsou ve WebGL k dispozici funkce `gl.getAttributeLocation()` a `gl.getUniformLocation()`. Atributy jsou myšleny vlastnosti vrcholů geometrie, tedy pozice, normálový vektor a texturovací souřadnice. Uniformy jsou ostatní parametry jako vlastnosti materiálů, světel a podobně. Tato metoda na závěr nastaví používaný program (`gl.useProgram()`) a povolí z-test (`gl.enable(gl.DEPTH_TEST)`)

.

### 6.8.2 Vykreslení scény

O vykreslení scény se stará metoda `RenderModule.drawScene()`. Tato metoda nemá žádné parametry.

Před vykreslením je třeba nastavit správně parametry shaderů. Ty jsou rozděleny do třech kategorií: globální uniformy, lokální uniformy a atributy. Globální uniformy jsou společné pro všechny vykreslované objekty, lokální uniformy a atributy můžou naproti tomu být pro každý objekt jiné. Globální uniformy tedy nastavíme hned. Lokální uniformy a atributy musíme nastavit až před vykreslením každého objektu.

Dále je třeba projít graf scény a spočítat transformační matice pro každý uzel, který se má vykreslit. Toto řeší funkce `RenderModule.sceneCull()`. Tato funkce zároveň roztřídí objekty na neprůhledné, průhledné a odrazivé a uloží je do příslušných seznamů včetně jejich matic a případně vzdálenosti od kamery.

Když máme vypočítané transformace, můžeme přejít k vykreslování. Nejdříve nastavíme transformaci normalizovaných souřadnic zařízení do souřadnic obrazovky (canvasu). K tomuto slouží funkce `gl.viewport()`, které předáme počáteční bod souřadnic (`[0,0]`)

a rozměry. Rozměry nastavíme podle rozměrů projekční roviny kamery. Dále vymažeme framebuffer a depth buffer pomocí funkce `gl.clear()`. Pro vymazání framebufferu slouží příznak `gl.COLOR_BUFFER_BIT` a pro depthbuffer `gl.DEPTH_BUFFER_BIT`. Když chceme použít oba zároveň, stačí je zkombinovat pomocí bitového operátoru `OR`.

Dalším krokem je vykreslení pozadí `RenderModule.drawBackground()`. Poté nastavíme potřebné vlastnosti WebGL kontextu. Povolíme ořez stěn (`gl.enable(gl.CULL_FACE)`), nastavíme směr vrcholů v přivrácené stěně (`gl.frontFace(gl.CCW)`) a zapneme ořez odvrácených stěn (`gl.cullFace(gl.BACK)`).

Nyní můžeme vykreslovat objekty. Nejdříve vykreslíme odrazivé objekty (zrcátka), které jsou uložené v seznamu `RenderModule.reflectiveItemslist`. Tento seznam předáme funkci `RenderModule.drawReflectiveGeometry()` a ta se postará o vykreslení objektů včetně odrazů. Dále vykreslíme neprůhledné objekty. Funkci `RenderModule.drawSolidGeometry()` předáme seznam `RenderModule.itemsList`. Jako poslední vykreslíme průhledné objekty ze seznamu `RenderModule.blendItemslist` pomocí funkce RenderModule.drawTransparentGeometry().

### 6.8.3 Příprava scény

Příprava scény spočívá v průchodu grafu scény, spočítání transformačních matic a roztřídění objektů podle průhlednosti a odrazivosti. Toto řeší funkce `RenderModule.sceneCull()`. V knihovně VRUT této funkci odpovídá funkce `Scene::Cull()`. Zde byla přesunuta do třídy `RenderModule`.

Na začátku funkce vyprázdní všechny seznamy objektů včetně fronty uzlů. Z grafu scény získá kořenový uzel a uloží ho do proměnné s aktuálním uzlem `node`. Lokální transformační matici tohoto uzlu si uloží do proměnné `worldMat`, která bude postupně násobena dalšími maticemi. V tuto chvíli se spustí cyklus průchodu frontou. Tento cyklus je řešen pomocí konstrukce `while (true)`, takže je v podstatě nekonečný, ale uvnitř cyklu je podmínka pro ukončení cyklu. Fronta uzlů je ve skutečnosti fronta objektů typu `pair [node, matrix]`, kde node je samotný uzel a matrix je součin matic od kořene až k rodičovskému uzlu.

Cyklus probíhá následovně:

- Pokud je aktuální uzel neplatný nebo neaktivní, tak vybereme z fronty další dvojici uzlu a matice. Pokud je fronta prázdná, tak cyklus ukončíme.
- Proměnnou `worldMat` vynásobíme lokální maticí uzlu a uložíme do proměnné `nodeWorldMat`. Pokud je lokální matice uzlu jednotková, tak pouze zkopírujeme `worldMat`.
- Vytvoříme nový objekt typu `itemInfo`, což je trojice ve tvaru `[distance, node, matrix]`. Distance představuje vzdálenost od kamery, zatím ji nastavíme na nulu, do `node` vložíme aktuální uzel a do matrix vložíme kopii `nodeWorldMat`. Tento objekt v sobě uchovává všechny potřebné informace pro správné vykreslení objektu.
- Pokud je uzel typu 3, tedy obsahuje geometrii, tak zjistíme jeho vzdálenost od kamery. Vezmeme střed jeho obalového kvádru (`AABB`), transformujeme maticí `nodeWorldMat` a spočítáme jeho vzdálenost od `RenderModule.cameraPosition`. Tuto vzdálenost uložíme do objektu `itemInfo`. Pokud je objekt reflektivní, tak objekt `itemInfo` vložíme do seznamu `reflectiveItemsList()`. Pokud je objekt průhledný tak `itemInfo` vložíme do seznamu `blendItemsList()`. Jinak `itemInfo` vložíme do seznamu `itemsList()`.

- Pokud je uzel typu 7, tedy pozadí, tak objekt `itemInfo` vložíme do seznamu `backgroundsItemsList()`.
- Projdeme seznam potomků uzlu a pro každého potomka vytvoříme nový objekt `pair`, do kterého vložíme uzel potomka a kopii aktuální matice `nodeWorldMat`. Tento nový objekt vložíme do fronty `shadowQueue`.
- Pokud aktuální uzel má nějaké potomky, tak matici aktuálního uzlu `worldMat` přepíšeme maticí `nodeWorldMat`. Do proměnné aktuálního uzlu `node` vložíme uzel prvního potomka.
- Pokud aktuální uzel potomka nemá, tak vybereme z fronty další dvojici uzlu a matice. Pokud je fronta prázdná, tak cyklus ukončíme.

Tento algoritmus provádí takzvaný „preorder“ průchod stromem.

Na závěr ještě seřadíme seznam `blendItemsList()` podle vzdálenosti od kamery tak, aby byly průhledné objekty vykresleny od nejvzdálenějších k nejbližším. Takto zajistíme v drtivé většině případů korektní vykreslení. Chybné zobrazení by mohlo nastat například, kdyby nějaké dva objekty měly neprázdný průnik. Protože předpokládáme použití průhledných objektů pro okna v autě, můžeme průnik objektů vyloučit.

### 6.8.4 Realistický shader

Vykreslování na grafické kartě je řešeno skrze takzvanou programovatelnou „pipeline“. Ve WebGL se programovatelné části týkají zpracování vrcholů a zobrazování fragmentů. Tyto programovatelné bloky se nazývají „vertex shader“ a „fragment shader“, přeneseně se tak říká i zdrojovým kódům, které tyto bloky vykonávají. Pro funkční renderování je tedy potřeba dodat WebGL tyto zdrojové kódy, přeložit je a spojit („slinkovat“) dohromady do takzvaného programu. V aplikaci jsou pro tento účel dvě funkce, které byly již dříve popsány (`programFromHTMLScripts()` a `programFromJSVariables()`).

Realistický shader program má na starosti zobrazovat objekty scény co možná nejvěrohodněji. Základ programu je založen na známém Phongově modelu [2]. Shader zvládne zobrazit najednou pouze jedno světlo, když ale scéna obsahuje více světel, je možné mezi nimi přepínat.

Obrázek 6.2: Testovací model vozu zobrazený realisticky

Vertex shader zpracovává vstupní data na úrovni vrcholů. Transformuje vrcholy podle modelové, pohledové a projekční matice podle vzorce 6.2.

$$gl\_Position = M_P * M_V * M_M * vec4(pos, 1.0); \tag{6.2}$$

Dále počítá transformaci normálových vektorů (normál). Pro transformaci normál je potřeba takzvané normálové matice. Tu získáme z funkce `getNormalModelMatrix()`. Normálová matice se počítá následovně: vezmeme první tři řádky a sloupce z modelové matice a vznikne matice $3 \times 3$, tuto matici invertujeme a transponujeme a výsledkem je matice pro transformaci normál. Transformace normály v shaderu je popsána rovnicí 6.3.

$$tNormal = M_N * normal; \tag{6.3}$$

Pro osvětlení je důležitý výpočet intenzity světla. Intezita je převrácená hodnota útlumu světla. Tlumení je vyjádřeno třemi koeficienty: konstantním, lineárním a kvadratickým. Dohromady tyto koeficienty určují jak světlo ztrácí na intenzitě se vzrůstající vzdáleností. Výpočet intenzity je popsán v rovnici 6.4, kde symbol $d$ představuje vzdálenost světla od fragmentu.

$$intensity = \frac{1.0}{Att_C + Att_L * d + Att_Q * d^2}; \tag{6.4}$$

Další výpočty ve vertex shaderu zahrnují: směr od vrcholu ke světlu, směr odraženého světla, směr od vrcholu ke kameře. Výsledné vypočtené a transformované hodnoty jsou pomocí proměnných deklarovaných jako `varying`, předány fragment shaderu.

Fragment shader má na starosti výpočet barvy konkrétních fragmentů. V tomto shaderu jsou nadeklarovány stejné proměnné typu `varying`, ze kterých získáme hodnoty vypočtené ve vertex shaderu a interpolované pro jednotlivé fragmenty.

Výpočet barvy fragmentu je popsán v rovnicích 6.5, 6.6, 6.7, 6.8 a 6.9 [2]. Znaky použité v těchto rovnicích mají následující význam: $M$ – materiál, $L$ – světlo, $I$ – intenzita světla,

$C$ – kamera, $N$ – normála, $R$ – vektor odrazu světla, $D$ – difuzní složka, $S$ – spekulární složka.

$$gl\_FragColor = M_{emission} + M_{ambient} + D + S; \tag{6.5}$$

$$D = M_{diffuse} * L_{color} * L_{angle} * I; \tag{6.6}$$

$$S = M_{specular} * L_{color} * L_{angle} * pow(C_{angle}, M_{shininess}) * I; \tag{6.7}$$

$$L_{angle} = max(dot(L_{direction}, N), 0.0); \tag{6.8}$$

$$C_{angle} = max(dot(R, C_{direction}), 0.0); \tag{6.9}$$

V shaderu je také řešeno texturování. Pokud má materiál objektu nastavenou a načtenou texturu, tak se místo difuzní barvy materiálu použije barva z textury (`texture2D(sampler, coords)`).

Při vykreslování odrazu scény u zrcadlových objektů je třeba scénu ořezat podle roviny odrazu, aby se objekty reálně umístěné za zrcadlem nezobrazovaly před zrcadlem. Proto je v shaderu také výpočet polohy fragmentu vůči rovině (rovnice 6.10). Proměnná *Plane* obsahuje koeficienty obecné rovnice roviny.

$$dot(Position, Plane) > 0.0; \tag{6.10}$$

Pokud je výsledek výpočtu kladný, fragment se nachází před zrcadlem a je zahozen (`discard;`).

V následujícím seznamu jsou uvedeny všechny parametry programu, které je třeba nastavit před vykreslováním:

- attribute vec3 pos – pozice vrcholu
- attribute vec3 normal – normálový vektor
- attribute vec2 texCoord – texturovací souřadnice
- uniform mat3 NMatrix – normálová matice
- uniform mat4 MMatrix – modelová matice
- uniform mat4 VMatrix – pohledová matice
- uniform mat4 PMatrix – projekční matice
- uniform vec3 cameraPosition – pozice kamery
- uniform vec3 lightPosition – pozice světla
- uniform vec3 lightAttenuation – útlum světla
- uniform vec4 lightColor – barva světla
- uniform vec4 matAmbient – ambientní barva materiálu
- uniform vec4 matDiffuse – difuzní barva materiálu
- uniform vec4 matSpecular – spekulární barva materiálu
- uniform vec4 matEmission – emisní barva materiálu

- uniform float matShininess – lesklost materiálu
- uniform bool mirrorClipping – přepínač ořezové roviny
- uniform vec4 mirrorClipPlane – ořezová rovina
- uniform bool textureEnable – přepínač texturování
- uniform sampler2D texSampler – sampler textury

O nastavení těchto parametrů se starají následující funkce:

- updateGlobalUniforms() – nastavuje uniformy, které jsou společné pro všechny objekty ve scéně. To se týká světla, kamery a ořezové roviny. Před nahráním hodnot do uniforem je nutné aktualizovat parametry renderovacího modulu, aby se projevily změny kamery a světla. K tomu slouží metody `updatePerspectiveMatrix()`, `updateViewMatrix()` a `updateLightPosition()`.
- updatePerspectiveMatrix() – metoda zkopíruje perspektivní projekční matici kamery do proměnné `RenderModule.perspMatrix`.
- updateViewMatrix() – metoda aktualizuje pohledovou kameru a pozici kamery. Nejdříve se invertuje pohledová matice kamery, tím získáme lokální transformační matici kamery. Poté pomocí `Scene.GetNodesTransformation()` získáme globální transformační matici kamery. Touto maticí vynásobíme pohledovou matici kamery a výsledek uložíme do `RenderModule.viewMatrix`. Globální maticí také transformujeme `Camera.eye` a výsledek uložíme do `RenderModule.cameraPosition`.
- updateLightPosition() – metoda aktualizuje polohu světla. Pokud je jako aktuální světlo nastaveno světlo „headlight“, tak stačí použít pozici kamery. Jinak získáme globální transformační matici uzlu světla a s ní spočítáme pozici světla.
- updateLocalUniforms() – metoda nastavuje uniformy, které jsou specifické pro každý objekt ve scéně. Jedná se o modelovou a normálovou matici, vlastnosti materiálu a texturu (pokud je již načtena). Pokud uzel grafu nemá nastaven materiál, tak se použije `RenderModule.defaultMaterial`.
- updateAttributes() – metoda připojuje(„binduje“) správné array buffery (`gl.bindBuffer()`) a nastavuje ukazatele atributů (`vertexAttribPointer()`). Podle přepínače drátového zobrazení připojuje také správný buffer elementů.

### 6.8.5 Jednoduché materiály

Jednoduchými materiály jsou myšleny materiály, které jsou neprůhledné a neodrazivé. O jejich vykreslení se stará metoda `RenderModule.drawSolidGeometry`. Metoda očekává v parametru seznam objektů, které se mají vykreslit. Seznam musí obsahovat prvky typu itemInfo `[distance, node, matrix]`. Seznam se projde for cyklem:

- Otestujeme, zda matice `matrix` aktuálního prvku obsahuje zrcadlení pomocí metody `isMatrixMirror()`. Pokud ano, je třeba změnit orientaci vrcholů přední stěny (`gl.frontFace()`) pro správné ořezání zadních stěn.
- Zavoláme metoda `updateLocalUniforms()` pro nahrání lokálních uniforem na grafickou kartu.

- Funkcí `updateAttributes()` připojíme správné buffery atributů.
- Podle stavu přepínače drátového zobrazení `RenderModule.wireframe` zavoláme funkci `gl.drawElements()` s příslušnými parametry. Funkce `gl.drawElements()` očekává tyto parametry: typ primitiva (například: `POINTS`, `TRIANGLES`, `LINES`, atd.), velikost indexového bufferu, datový typ indexů a offset, odkud se mají začít číst indexy. Tímto jsme zadali grafické kartě vše potřebné pro vykreslení aktuálního objektu a můžeme přejít na další prvek seznamu.


Obrázek 6.3: Testovací model vozu v drátovém zobrazení

Funkce `gl.drawElements()` v základu podporuje největší datový typ `unsigned short`, který je typicky 16bitový. Maximální počet indexů je tak omezen na hodnotu 65536. V této aplikaci je však předpokládáno, že geometrie můžou mít i více indexů. Z tohoto důvodu je využito rozšíření `OES_element_index_uint`, které povoluje použití datového typu `unsigned int`. Tento datový typ je typicky 32bitový nebo větší. Toto umožňuje rozsah indexů až 4 miliardy, což je více než dostatečné.

### 6.8.6 Průhledné materiály

Objekty s průhledným materiálem jsou typicky okna nebo jiné skleněné prvky. O jejich vykreslení se stará metoda `RenderModule.drawTransparentGeometry`. V parametru očekává seznam stejného typu jako metoda `RenderModule.drawSolidGeometry`. Seznam objektů by měl být seřazen podle vzdálenosti objektů od kamery, jinak se můžou objekty vykreslovat chybně.

Průhlednost objektů je simulována pomocí takzvaného „blendingu“. Při blendingu nejsou hodnoty ve framebufferu přepisovány novými, ale nové fragmenty (zdrojové) jsou se starými (cílovými) zkombinovány. Jak má blending tyto fragmenty kombinovat, můžeme nastavit pomocí funkcí `blendEquation()` a `gl.blendFuncSeparate()`. První určuje, jak oba fragmenty zkombinovat: sečíst, odečíst nebo odečíst převrácené hodnoty. Druhá funkce ur-

čuje koeficienty, kterými se mají fragmenty vynásobit před smícháním. Ještě existuje varianta `gl.blendFunc()`, tato nastavuje multiplikativní koeficienty celým fragmentům, tedy RGBA. Rozšířená verze umožňuje nastavit koeficienty zvlášť pro RGB a zvlášť pro A. Toto je výhodné, pokud chceme blendovat více objektů přes sebe. Alfa kanál fragmentu většinou určuje právě tyto multiplikativní koeficienty, a proto by neměl být alfa kanál násobený spolu s RGB kanály [2].

Nejdříve musíme povolit blending. To uděláme pomocí `gl.enable(gl.BLEND)`. Funkce `blendEquation()` je implicitně nastavena na sčítání, což nám vyhovuje a nemusíme nic měnit. Zbývá jen správně nastavit parametry funkce `gl.blendFuncSeparate()`, jsou čtyři. První určuje koeficient kanálů RGB zdrojového fragmentu, tento nastavíme na hodnotu `gl.SRC_ALPHA`. Druhý parametr určuje koeficient kanálů RGB cílového fragmentu, ten nastavíme na `gl.ONE_MINUS_SRC_ALPHA`. Takto zajistíme, že po sečtení budou hodnoty barevných kanálů ve správném rozsahu a nepřetečou. Třetí a čtvrtý parametr určují koeficienty alfa kanálů zdrojového a cílového fragmentu, ty chceme ponechat beze změny, a proto nastavíme do obou parametrů hodnotu `gl.ONE`.

Průchod skrze seznam prvků je úplně stejný jako u `RenderModule.drawSolidGeometry`. Po průchodu celým seznamem jen zase vypneme blending `gl.disable(gl.BLEND)`.

### 6.8.7 Odrazivé materiály

Odrazivý materiál je typicky použit u zrcadel nebo v případě auta u zpětných zrcátek. O vykreslení odrazivých materiálů se stará metoda `RenderModule.drawReflectiveGeometry`. V parametru očekává seznam stejného typu jako metoda `RenderModule.drawSolidGeometry`.

Pro simulaci odrazu se využívá dvojitého renderovacího průchodu. Jednou vykreslíme scénu normálně a poté ji vykreslíme zrcadlenou podle určité plochy. Tento postup je nutné použít pro každé zrcadlo ve scéně. Takže s růstem počtu zrcadel roste úměrně počet renderovacích průchodů. Pokud bychom chtěli renderovat i dvojité odrazy, počet průchodů by teoreticky rostl kvadraticky s počtem zrcadel. V tomto rendereru si vystačíme s jednoduchými odrazy.

Aby se zrcadlená scéna vykreslovala pouze do části obrazu, kterou zabírá zrcadlo, se používá takzvaný „stencil test“. Stencil test potřebuje takzvaný stencil buffer, ve kterém můžeme nastavit různé hodnoty pro pixely obrazu a tento buffer pak použít jako masku. Pokud hodnota ve stencil bufferu nesplňuje podmínku stencil testu, fragment se zahodí.

Pro práci se stencil bufferem exustuje několik funkcí: `gl.stencilFunc()` nastavuje, kdy a co se má zapsat do stencil bufferu, `gl.stencilOp()` specifikuje, co se má stát s hodnotou ve stencil bufferu v závilosti na výsledku stencil a depth testu, `gl.stencilMask()` zase definuje masku pro zablokování nebo povolení zápisu hodnot na bitové úrovni [7].

Ve WebGL není stencil buffer implicitně vytvořen, musíme o něj explicitně požádat při vytváření kontextu. Ve funkci `OnLoad()` při stratu aplikace musíme upravit získání kontextu: `getContext("webgl", stencil:true)`. Po získání kontextu je třeba zkontrolovat, že se stencil buffer opravdu vytvořil. Pomocí funkce `gl.getContextAttributes()` získáme objekt s atributy kontextu. Pokud má tento objekt nastaven příznak `stencil` na hodnotu `true`, tak byl stencil buffer úspěšně vytvořen.

Princip vykreslení zrcadla je takový, že nejdříve do stencil bufferu vykreslíme objekt zrcadla. Poté zrcadlíme celou scénu a vykreslíme ji pouze do prostoru, vymezeného maskou ve stencil bufferu. Na závěr vykreslíme objekt zrcadla pouze do depth bufferu, aby vykreslený odraz nebyl později překreslen objekty za zrcadlem.

Renderer umí zobrazovat pouze rovinná zrcadla, toto klade speciální požadavky na jejich geometrii. Vrcholy geometrie zrcadla by měly všechny ležet pouze v jedné rovině. Geometrie tak nebude uzavřená ani 2-manifold. Nerovinná geometrie vyústí v nekorektní zobrazení. Stejné požadavky má i knihovna VRUT. Výsledné zobrazení odrazivého materiálu je ukázáno na detailu zrcátka testovacího modelu vozu na obrázku 6.4.

Obrázek 6.4: Detail zrcátka testovacího modelu vozu

Funkce opět vykresluje objekty ze seznamu v cyklu:

- Nejdříve nastavíme správně orientaci přední stěny (`gl.frontFace()`) pro ořezání zadních stěn.
- Z objektu třídy `Geomtetry` získáme první vrchol a první normálu a transformujeme je do světových souřadnic. Tyto objekty nyní určují rovinu zrcadla.
- Vypočteme vektor z prvního bodu směrem ke kameře a znormalizujeme ho. Pomocí skalárního součinu tohoto vektoru s normálou zjistíme orientaci kamery vůči rovině. Pokud je znaménko skalárního součinu záporné, tak se kamera nachází za zrcadlem. V takovém případě převrátíme normálu a přepneme ořez zadních stěn na ořez předních stěn (`gl.cullFace(gl.FRONT);`). Dále nastavíme příznak `flip`, který indikuje, že jsme převrátili normálu.
- Povolíme stencil test `gl.enable(gl.STENCIL_TEST)`, nastavíme funkci stencil testu pro zápis: `gl.stencilFunc(gl.ALWAYS, 1, 0xFF)`, nastavíme akci při stencil testu `gl.stencilOp(gl.KEEP, gl.KEEP, gl.REPLACE)` a povolíme všechny bity k zápisu `gl.stencilMask(0xFF)`.
- Protože teď nechceme kreslit ani do color bufferu ani do depth bufferu, tak je deaktivujeme (`gl.colorMask()` a `gl.depthMask()`).
- Vymažeme stencil buffer (`gl.clear(gl.STENCIL_BUFFER_BIT)`).

- Nastavíme uniformy a atributy a vykreslíme geometrii zrcadla (`gl.drawElements()`).
- Nastavíme funkci stencil testu pro testování zapsané hodnoty: `gl.stencilFunc( gl.EQUAL, 1, 0xFF)` a zakážeme zápis do stencil bufferu `gl.stencilMask(0x00)`.
- Povolíme zápis do depth bufferu a color bufferu (`gl.colorMask()` a `gl.depthMask()`).
- Pokud byla převrácena normála, tak vrátíme zpátky ořez zadních stěn.
- Spočítáme rovnici roviny a matici zrcadlení. (metody `getPlaneEquation()` a `mirrorMat()` budou popsány dále)
- Vykreslíme zrcadlenou scénu pomocí funkce `RenderModule.drawSceneReflection()`.
- Vypneme zápis do stencil bufferu a color bufferu a zapneme zápis do depth bufferu.
- Pokud byla převrácena normála, tak opět nastavíme ořez předních stěn.
- Nastavíme uniformy, atributy a znovu vykreslíme geometrii zrcadla (`gl.drawElements()`), tentokrát pouze do depth bufferu.
- Zapneme zápis do color bufferu a vypneme stencil test pomocí funkce `gl.disable( gl.STENCIL_TEST)`.

O vykreslení zrcadlené scény se stará funkce `RenderModule.drawSceneReflection`. Funkce očekává v parametrech matici zrcadlení, koeficienty rovnice roviny zrcadla a uzel zrcadla, jehož odraz má být vykreslen.

- Na úvod se vytvoří kopie seznamů neprůhledných, průhledných i odrazivých objektů. Každému prvku v těchto seznamech je jeho matice vynásobena maticí zrcadlení. V seznamu odrazivých objektů je vynechán objekt, jehož odraz se právě řeší. U průhledných objektů musí být přepočítána vzdálenost od kamery a jejich seznam je znovu seřazen.
- Zrcadlíme pozici kamery a světly pro správný výpočet osvětlení.
- Nastavíme přepínač ořezu podle roviny a koeficienty ořezové roviny pošleme na grafickou kartu.
- Vykreslíme odrazivé objekty jako obyčejné. Nepotřebujeme vícenásobné odrazy. Vykreslíme neprůhledné objekty a potom i průhledné.
- Vypneme přepínač ořezu rovinou a vrátíme kameru i světlo na původní pozice.

Vykreslení zrcadel používá také tyto další metody:

- getPlaneEquation() – metoda vrátí čtyřprvkový vektor s koeficienty rovnice roviny. V parametrech očekává bod v rovině a její normálový vektor. Rovina je definovaná rovnicí 6.11, kde $n$ značí normálový vektor a $p$ bod ležící v rovině.

$$ax + by + cz + d = 0; n = (a, b, c); d = -n \cdot p \tag{6.11}$$

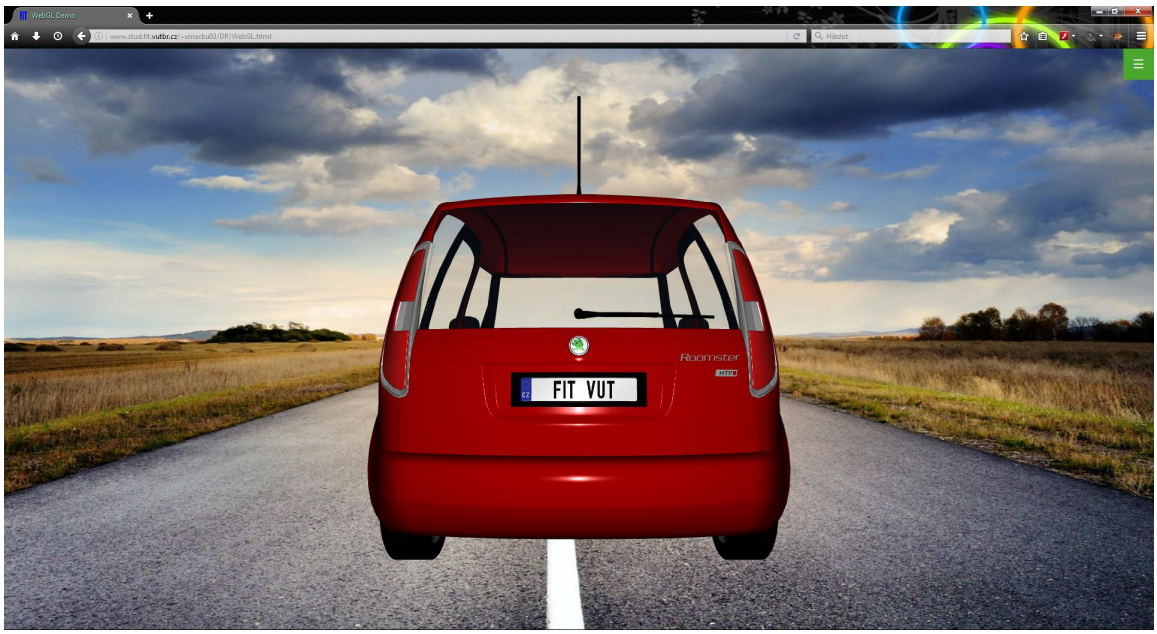
- mirrorMat() – vrací transformační matici zrcadlení, pro zrcadlení kolem libovolné roviny. V parametrech očekává bod v rovině a její normálový vektor. Matice je poté vypočítána podle rovnice 6.12 [3], kde $n$ značí normálový vektor a $p$ bod ležící v rovině.

$$M_R = \begin{vmatrix} -2aa+1 & -2ab & -2ac & 0 \\ -2ba & -2bb+1 & -2bc & 0 \\ -2ca & -2cb & -2cc+1 & 0 \\ -2da & -2db & -2dc & 1 \end{vmatrix}; n = (a,b,c); d = -n \cdot p \tag{6.12}$$

### 6.8.8 Textury

Textury ve WebGL se používají jako v běžném OpenGL. Před vykreslením musíme připojit texturovací jednotku a v shaderech definovat proměnou typu sampler. Rozměry textur nejsou v podstatě omezeny, ale je velmi výhodné používat textury s rozměry v mocninách dvou. Pokud je textura jiných rozměrů, tak pro ni nelze vytvořit MIP mapu ani nejde texturu opakovat [2].

Pokud má nějaký materiál definovanou texturu, tak je při zpracování dat tato textura načtena prostřednictvím funkce loadTexture(). Tato funkce vytvoří v objektu materiálu nový objekt textury (`gl.createTexture()`) a spustí asynchronní načítání obrázku. Po úspěšném načtení obrázku se spustí „callback“ funkce `handleTextureLoaded()`. Tato druhá funkce naplní texturu daty obrázku a nastaví parametry textury. Provede se test, zda má textura rozměry v mocninách dvou [1]. Pokud ano, tak se nastaví správné filtrování textury a vygeneruje se MIP mapa. Pokud má textura jiné rozměry, tak se vypne filtrování a opakování(„wrapping“). Poté se materiálu nastaví příznak, že textura byla úspěšně načtena. Pokud je to možné, tak se také překreslí scéna, aby se načtení textury projevilo.




Obrázek 6.5: Testovací model vozu v pohledu zezadu, na znaku a spz jsou použité textury

### 6.8.9 Pozadí

Graf scény může obsahovat uzly s pozadím. Takový uzel má svůj materiál, ale nemá geometrii. Z materiálu se používá difuzní barva jako barva pozadí, případně textura jako obrázek na pozadí. Protože graf scény může obsahovat více uzlů s pozadím, tak je v `RenderModule` seznam uzlů s pozadím a index aktuálního pozadí. V menu je možné mezi nimi přepínat.

O vykreslení pozadí se stará funkce `RenderaModule.drawBackgrund`:

- Pokud ve scéně není pozadí nebo není žádné pozadí aktivní, tak se nastaví základní barva pozadí (`RenderModule.defualtBackground`). Barva se nastaví jednak jako `gl.clearColor()` a jako barva pozadí canvasu.
- Když ve scéně nějaké pozadí je, tak zkontrolujeme, zda je v aktivním podstromu grafu. Proto ve funkci `sceneCull()` řešíme i uzly pozadí.
- Pokud má materiál pozadí pouze barvu, tak ji nastavíme jako pozadí.
- Jestli má materiál definovanou texturu, a ta je načtena, tak ji nastavíme jako pozadí canvasu. Barvu pozadí canvasu v takovém případě nastavíme na průhlednou. Toto je změna oproti knihovně VRUT, která musí pozadí vykreslit jako obdélník přes celý obraz a otexturovat jej.

Knihovna vrut navíc u pozadí podporuje takzvané „cube mapy“, což je šest textur namapovaných na stěny krychle. Takto je simulováno panoramatické pozadí. Díky cube mapám fungují v knihovně VRUT i odlesky prostředí.

Ve WebGL lze cube mapy také použít. Během implementace jsem zjistil, že v materiálech nejsou uloženy adresy obrázků cube mapy. Dotázal jsem se tedy doktora Míška, jak z VRUT souboru tyto informace zjistit. Na základě mého dotazu doktor Míšek objevil chybu v knihovně VRUT. Tato chyba způsobuje, že cube mapy se neuloží do souboru se scénou. Do odevzdání práce nebyla chyba zatím odstraněna a tak tato funkcionalita nemohla být implementována.

### 6.8.10 Výběr geometrie

Když uživatel v aplikaci pokliká na nějaký objekt, je tento objekt zvolen. Implementace této funkčnosti je v knihovně VRUT řešena pomocí vrhání paprsků (funkce `Scene::CastRay()` a `Geometry.CastRay()`). V javascriptu by byla takováto implementace výpočetně a časově náročná, proto bylo zvoleno jiné řešení, takzvaný „picking“.

Princip pickingu spočívá v očíslování objektů. Objekty se vykreslí do textury a místo barvy se při vykreslování použije toto číslo. Z textury se na souřadnicích kliknutí přečte hodnota, která se převede na číslo geometrie.

Toto řešení je implementováno v následujících metodách:

- initPicking() – inicializace pickingu. Vytvoří nový shader program pro vykreslení identifikátorů objektů. Vertex shader potřebuje znát pouze pozici vrcholu a transformační matice, transformovaný vrchol pošle dál. Fragment shader pouze na místo fragmentu zapíše barvu, v které je zakódován identifikátor objektu. Dále funkce uloží lokace všech uniforem a atributu pozice. Poté vytvoří nový framebuffer, nastaví mu rozměry a připojí k němu texturu jako color buffer. Jako další je potřeba vytvořit renderbuffer pro použití jako depth buffer. Tento renderbuffer je taktéž připojen k framebufferu.

- pickingDraw() – metoda vykreslí scénu do vedlejšího framebufferu. V parametrech je třeba zadat souřadnice kliknutí, funkce potom vrátí identifikátor uzlu, na který bylo kliknuto. Nejdříve je přepnut program a je připojen nový framebuffer. Nastaví se viewport a vymaže se color buffer i depth buffer. Nastaví se potřebné uniformy a pro každý objekt ze seznamů neprůhledných, průhledných i odrazivých se spustí metoda `pickingDrawItem()`. Z framebufferu se pomocí `gl.readPixels()` přečte `ArrayBuffer()` s hodnotou barvy na souřadnicích klinutí. Odpojí se framebuffer a nastaví se původní program. Přečtená hodnota se předá funkci `vec4ToNodeID()`, která dekóduje z barvy identifikátor.

- pickingDrawItem() – metoda vykreslí zadaný objekt. Převede identifikátor objektu na barvu pomocí funkce `nodeIDToVec4()` a spolu s modelovou maticí ji pošle na grafickou kartu. Připojí správné buffery s vrcholy a indexy a vykreslí geometrii.

- nodeIDToVec4() – převede číselný identifikátor na barvu ve formátu RGBA. K převodu využívá čtyřbytový `ArrayBuffer` a různé pohledy [6]. Skrze pohled `Uint32Array()` do bufferu nahraje identifikátor. Skrze pohled `Uint8Array()` ze stejného bufferu přečte čtyři osmibitová čísla. Tato čísla poté normalizuje a vznikne vektor s RGBA barvou.

- vec4ToNodeID() – převede barvu na identifikátor. Funkce `readBackPixels()` umí automaticky převést hodnotu pixelu na 4 osmibitová čísla, která vrátí jako `ArrayBuffer()`. Stačí tedy z tohoto bufferu vytvořit pohled typu `Uint32Array()` a přečíst první hodnotu, čímž dostaneme zpátky identifikátor [6].

### 6.8.11 Zvýraznění geometrie

Zvýrazněním geometrie je myšleno vizuální odlišení vybraného uzlu od zbytku scény, podobně jako například na obrázku 5.4.

Zvýraznění implementované v aplikaci spočívá ve vykreslení obrysu a následném vykreslení objektu. Zvýrazněný objekt je vykreslen jako poslední, aby se zobrazil přes původní obraz. Takto je dosaženo, že objekt je vidět ze všech úhlů, i skrze ostatní objekty ve scéně. Obrys vznikne tak, že vrcholy geometrie trochu vytáhneme ve směru normály.

Zvýraznění geometrie je implementováno následujícími metodami:

- initOutline() – inicializace kreslení obrysu. Vytvoří nový shader program pro vykreslení obrysu objektu. Vertex shader potřebuje znát pouze pozici vrcholu, normálu, transformační matice a velikost posunu. Vrchol je posunut ve směru normály a poté transformován pohledovou a projekční maticí. Fragment shader pouze na místo fragmentu zapíše barvu. Dále funkce uloží lokace všech uniforem a atributů, definuje barvu obrysu a spočítá vytažení vrcholů jako setinu poloměru obalového kvádru celé scény.

- drawSelectedItem() – metoda zvýrazní vybraný objekt. Nejdříve přepne program, pak vypne depth test a vyčistí depth buffer. Nastaví se všechny potřebné uniformy a buffery a vykreslí se zvolená geometrie. Podle přepínače drátového zobrazení se vykreslí plně nebo drátově. Poté je zapnut depth test a je přepnut původní program. Na závěr je vykreslen zvolený objekt jako neprůhledný objekt (`drawSolidGeometry()`).

- setSelectedNode() – metoda nastaví uzel jako vybraný a překreslí scénu. V parametru je očekáván zvolený uzel.

Pro správnou funkcionalitu je třeba trochu upravit metodu `RenderModule.sceneCull()`. Když se při průchodu scénou narazí na zvolený uzel, tak jej vložíme do speciálního seznamu vybraných objektů. Tento seznam má maximální velikost jedna, protože je podporován výběr jen jednoho prvku. Snadno lze aplikaci rozšířit, aby šlo vybrat více prvků.


Obrázek 6.6: Testovací model vozu se zvýrazněnými levými předními dveřmi

### 6.8.12 Stylizovaný shader

Stylizovaný shader má zobrazovat scénu jiným než realistickým způsobem. Jako stylizované zobrazení lze považovat již drátové zobrazení nebo zvýraznění geometrie. Tento stylizovaný shader je ještě o krok dál. Tento shader již nerozlišuje objekty na průhledné, neprůhledné a odrazivé. Všechny objekty ve scéně vykresluje stejně. Objekty jsou nejdříve vykresleny tmavě modrou barvou a poté jsou světle modrou barvou zvýrazněny hrany trojúhelníků.

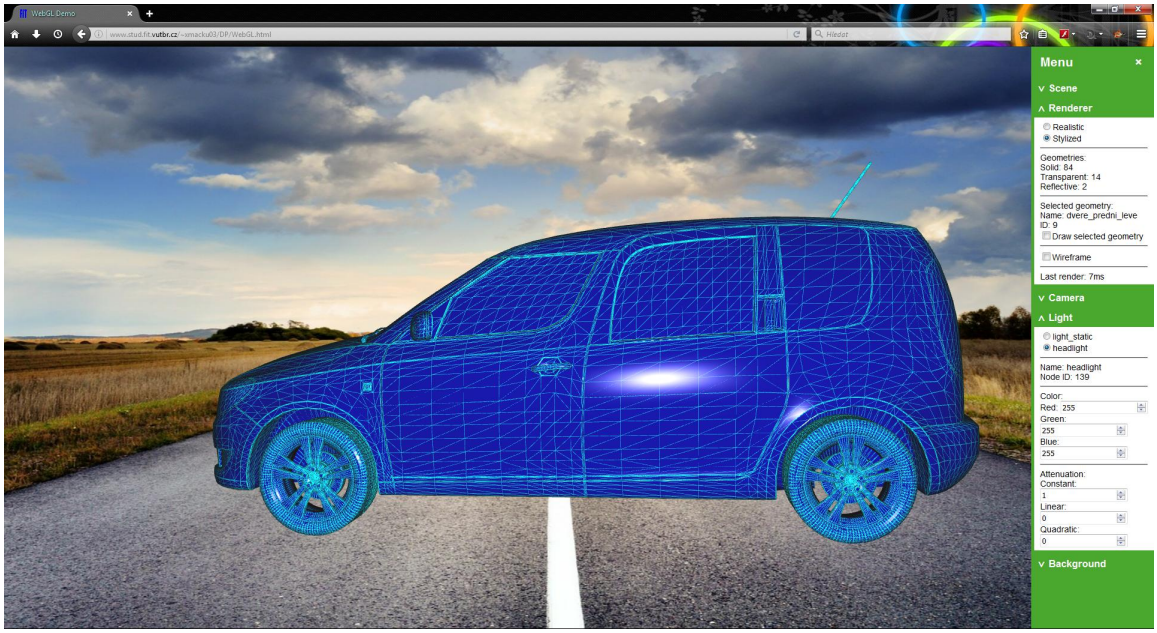
Stylizovaný shader je odvozen z realistického. Bylo v něm odstraněno texturování, clipping. Místo barvy z materiálu je použita stejná barva pro všechny objekty.

Pro snadné přepínání mezi realistickým a stylizovaným programem byla implementována metoda `switchProgram`, která přepne program pomocí `gl.useProgram()` a povolí správné vertexové atributy.

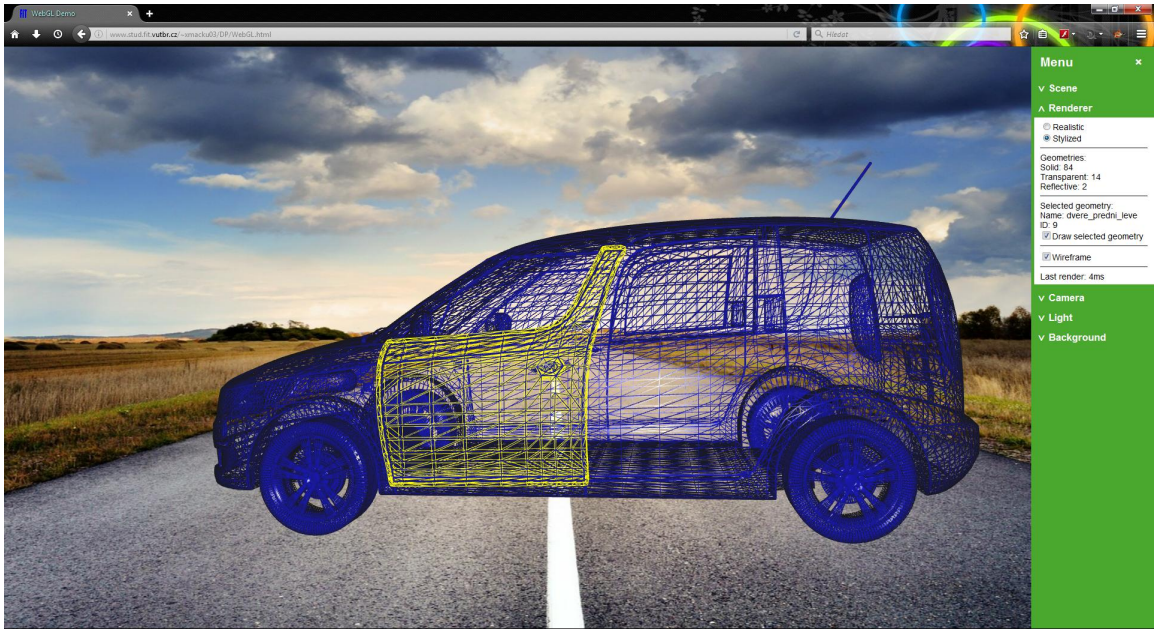
Aby vykreslení fungovalo, musely být upraveny některé metody renderovacího modulu. Většinou se jednalo o vložení testu na aktuální shader, podle kterého se vybírá původní nebo alternativní chování funkce.

- initShaders() – stylizovaný shader program má jinou skladbu prametrů a je potřeba je také inicializovat.
- updateGlobalUniforms() – pro stylizovaný shader program není třeba nastavovat uniformy pro ořez rovinou.
- updateLocalUniforms() – pro stylizovaný shader program stačí nastavit modelovou a normálovou matici.

- updateAttributes() – pro stylizovaný shader program stačí připojit buffer vrcholů a normál a nastavit jejich ukazatele atributů (`gl.vertexAttribPointer()`).
- drawSolidGeometry() – pro stylizovaný shader program nastavíme barvu a vykreslíme trojúhelníky, poté vyměníme barvu a vykreslíme hrany; pokud je přepnuto drátové vykreslování, tak vykreslíme pouze hrany.
- sceneCull() – v této funkci není změna, i když by mohly být všechny objekty vloženy do jednoho seznamu, tak je třídění objektů ponecháno beze změny.
- drawScene() – zde se liší vykreslování seznamů objektů, všechny tři seznamy objektů (neprůhledné, průhledné i odrazivé) vykreslíme jako obyčejné neprůhledné pomocí drawSolidGeometry().
- pickingdDraw() – zde je třeba přepnou správný program po ukončení pickingu.
- drawSelectedItem() – stejně tak i zde je třeba přepnou správný program po vykreslení obrysu vybraného objektu.




Obrázek 6.7: Testovací model vozu ve stylizovaném zobrazení



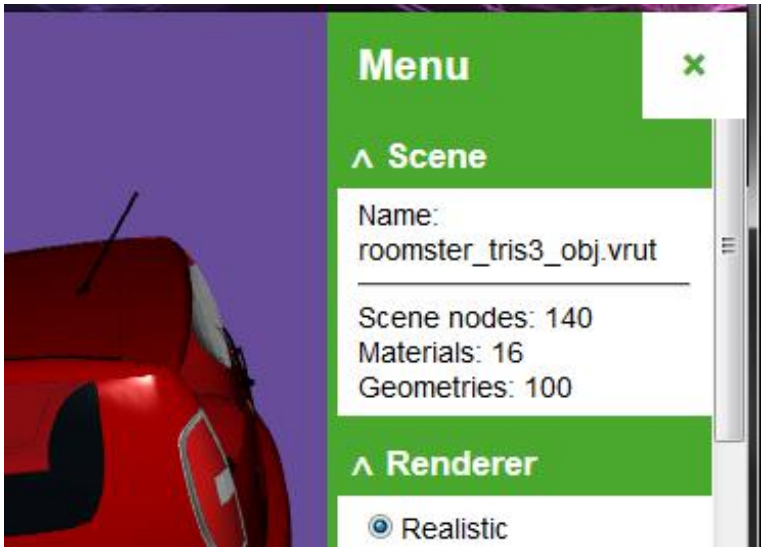

Obrázek 6.8: Testovací model vozu ve stylizovaném zobrazení se zapnutým drátovým zobrazením a se zvýrazněnými levými předním dveřmi

## 6.9 Třída GUI

Třída `GUI` obsahuje funkce pro sestrojení grafického menu a pro ovládání aplikace skrze toto menu. Menu je rozděleno do sekcí, které obsahují informace a ovládací prvky určité části aplikace.

Při spuštění aplikace je v pravém horním rohu zobrazeno tlačítko, kterým se otevírá menu. Po otevření menu je na stejném místě vpravo nahoře tlačítko pro zavření menu. V menu je seznam názvů sekcí, které fungují jako tlačítka. Po kliknutí na nějaký název se pod ním rozbalí příslušná sekce. Dalším kliknutím se tato sekce zase sbalí.

Tlačítka pro otevření/zavření jsou odlišena, pro jednodušší řešení překrytí tlačítka a menu. S jedním tlačítkem byl problém, pokud bylo otevřeno menu a jeho vnitřní obsah byl větší než výška stránky. V menu se správně zobrazil posuvník, ale v horním rohu byl částečně tlačítkem překryt (obr. 6.9).




Obrázek 6.9: Chybné zobrazení menu, při použití pouze jednoho tlačítka

Otevírání a zavírání menu a sekcí je řešeno pomocí tříd. Element může mít třídu `"opened"` nebo `"closed"`. Podle těchto tříd prohlížeč pozná, jaký má použít styl pro daný element. V CSS pravidlech je nastavena pro obě třídy rozdílná šířka (menu) nebo výška (sekce). Použitím CSS vlastnosti `transition` je dosaženo animace přechodu mezi styly.

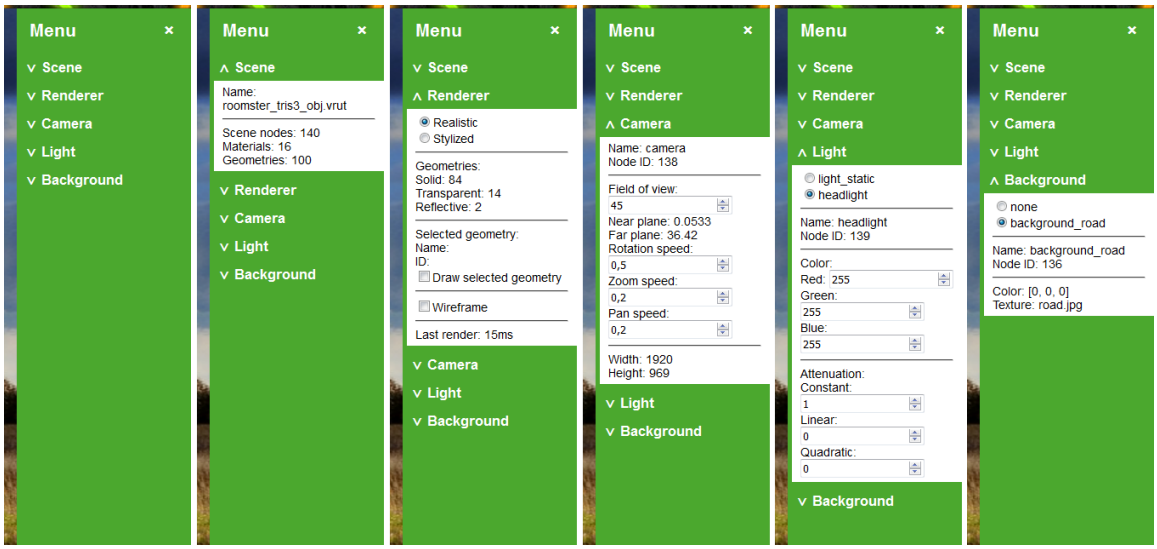
Třída `GUI` obsahuje následující metody:

- createMenu() – vytvoří HTML strukturu menu. Prvně vytvoří tlačítko pro oteření a vloží ho do divu aplikace. Dále vytvoří div menu, do něj vloží název, tlačítko pro zavření a vytvoří jednotlivé sekce menu.
- create[Name]Section() – vytvoří konkrétní sekci menu. Použije metodu `createMenuSection()` a obsah nové sekce vytvoří metodou `set[Name]SecContent()`. `[Name]` je zástupný symbol pro název sekce. Pro každou sekci existuje speciální metoda (`createSceneSection()` a podobně).
- createMenuSection() – obecná metoda pro tvorbu nové sekce menu. Potřebuje znát tři parametry: název sekce, obsah sekce, a funkci pro aktualizaci obsahu. Každá sekce se skládá z hlavního elementu, který sekci drží pohromadě. Potomky tohoto elementu jsou: název sekce a její obsah, který je možné skrýt.
- set[Name]SecContent() – metoda vytvoří obsah konkrétní sekce a nastaví všechny potřebné ovládací funkce aktivním elementům. Tato metoda je volána před otevřením sekce, aby se aktualizoval obsah.
- setContent() – metoda aktualizuje obsah sekce zadané v parametru. Podle hodnoty parametru volá některou z metod `set[Name]SecContent()`. Tato metoda se využívá při rozbalení sekce pomocí `toggleSection()`, kdy známe název sekce a potřebujeme zavolat správnou metodu pro aktualizaci obsahu rozbalované sekce.
- openMenu() – otevře menu.
- closeMenu() – uzavře menu, včetně všech sekcí.

- toggleSection() – otevírá a zavírá konkrétní sekci, na jejíž název bylo kliknuto.

V menu je konkrétně těchto pět sekcí:

- Scene – obsahuje pouze informace o scéně: název, počet uzlů v grafu scény, počet materiálů a počet geometrií.
- Renderer – umožňuje přepínat mezi realistickým a stylizovaným zobrazením. Zobrazuje počet neprůhledných, průhledných a odrazivých geometrií ve scéně. Dále je zde možnost přepínat drátové zobrazení a zobrazení vybrané geometrie. Na posledním řádku zobrazuje průměrný čas 100 posledních vykreslení.
- Camera – zobrazuje informace o kameře: název, ID, blízkou a vzdálenou rovinu a rozměry projekční roviny. Dále je zde možné měnit zorný úhel a rychlosti pohybu kamery.
- Light – umožňuje přepínat mezi světly ve scéně. Zobrazuje informace o aktuálním světle (název, ID) a dokáže měnit barvu a tlumení aktuálního světla.
- Background – umožňuje přepínat mezi pozadími ve scéně. Zobrazuje informace o použitém pozadí: název, ID, barvu a jméno textury.




Obrázek 6.10: Menu aplikace s rozbalenými sekcemi

## 6.10 Shrnutí

V této kapitole jsem popsal implementaci aplikace podle návrhu z předchozí kapitoly. Zabýval jsem se tím, jak aplikace funguje a co dokáže. Velká část kapitoly byla věnována řešení realistického zobrazení. Stylizované zobrazení jsem popsal ve srovnání s realistickým: co bylo potřeba doplnit, upravit a podobně. V kapitole jsem také popsal ovládání aplikace a grafické uživatelské rozhraní.

# Kapitola 7

# Testování

Když je aplikace implementovaná, je třeba ji otestovat, zda funguje jak má. Aplikaci jsem nahrál na školní server a je dostupná na adrese:
http://www.stud.fit.vutbr.cz/~xmacku03/DP/WebGL.html.

Testování probíhalo primárně v prohlížeči Mozilla Firefox ve verzi 46. Obrázky použité v předešlém textu jsou všechny vygenerované ve Firefoxu. Dále byla aplikace testována v prohlížečích Google Chrome 50 a Opera 37. Testování v těchto prohlížečích bylo prováděno pouze pro ověření kompatibility implementace. Při testování tedy bylo kontrolováno, zda je scéna správně vykreslována, zda funguje ovládání a menu. Snímky obrazovky z tohoto testování jsou na obrázcích B.1, B.2, B.3 a B.4.

Dalším krokem bylo testování, zda aplikace funguje i na mobilních zařízeních. Zde byly k testování použity mobilní verze stejných prohlížečů, konkrétně Firefox for Android 46, Chrome for Android 50 a Opera Mobile 36. Snímky obrazovky z těchto prohlížečů jsou na obrázcích B.5, B.6, B.7, B.8 a B.9.

Během testování byl zaznamenáván čas vykreslení při pohybu kamery. Bylo zaznamenáno 100 hodnot, které byly zprůměrovány. Porovnání jednotlivých prohlížečů je v tabulce 7.1. Časy byly naměřeny na počítači s procesorem AMD Athlon II X4 3 GHz a grafickou kartou AMD Radeon R7, a na mobilním telefonu Samsung Galaxy S4 mini.

| Prohlížeč/ Zobrazení | Firefox | Chrome | Opera | Firefox for Android | Chrome for Android | Opera Mobile |
|---|---|---|---|---|---|---|
| Realistické | 7,18 | 6,91 | 6,92 | 323 | 357 | 241 |
| Drátové | 7,95 | 6,64 | 6,57 | 356 | 358 | 137 |
| Stylizované | 3,63 | 2,24 | 2,27 | 35 | 56 | 47 |
| Stylizované drátové | 2,79 | 2,30 | 2,25 | 32 | 47 | 41 |

Tabulka 7.1: Srovnání časů vykreslení mezi prohlížeči, časy jsou uvedeny v ms

## 7.1 Shrnutí

Při testování jsem zjistil, že aplikace funguje ve všech nejpoužívanějších prohlížečích včetně jejich mobilních verzí. Z tabulky 7.1 vyplývá, že na desktopových prohlížečích je aplikace

zhruba stejně rychlá, stylizované zobrazení je asi 2x až 3x rychlejší a zapnutí drátového zobrazování nemá na rychlost vykreslení příliš vliv. U mobilních prohlížečů jsou výsledky zajímavější. Realistické zobrazení testovacího modelu bylo pro všechny mobilní prohlížeče až příliš náročné. Mobilní opera se s tímto úkolem poprala nejrychleji. Stylizované zobrazení zrychlilo vykreslení 3x až 7x. Nejzajímavější výsledek je zrychlení v mobilním Firefoxu, který při zapnutém drátovém zobrazení dokázal scénu vykreslit až 11x rychleji než při realistickém zobrazení.

# Kapitola 8

# Závěr

Cílem této práce bylo navrhnout a implementovat zobrazování automobilů ve WebGL. Zobrazování mělo být jak realistické tak stylizované. Práci jsem vypracoval ve spolupráci s firmou Škoda Auto. Z tohoto směru byly na výslednou aplikace kladeny další požadavky: zobrazovat modely ve formátu VRUT, struktura a fungování aplikace by mělo odpovídat aplikaci VRUT, atd.

V úvodu jsem tedy nastudoval technologii WebGL a poté jsem se probíral rozsáhlou knihovnou VRUT. Pro implementaci jsem musel prostudovat jak VRUT reprezentuje scénu a objekty v ní, jak je tato scéna zpracována a jak jsou vykreslovány jednotlivé objekty.

V aplikaci jsem implementoval Phongův model stínování, kreslení průhledných objektů, vykreslení libovolně natočených rovinných zrcadel, texturování a výběr objektu kliknutím (picking). Jako stylizované zobrazení jsem implementoval několik metod: drátové zobrazení, jednobarevné stínované zobrazení s viditelnými hranami a zvýraznění vybraného objektu.

Pro pohodlné ovládání aplikace na desktopu i na mobilním zařízení jsem implementoval jak ovládání pomocí myši tak i dotykové ovládání. Pro grafické rozhraní jsem implementoval jednoduché vysouvací menu, které umožňuje měnit některé parametry kamery, světla a renderovacího modulu.

Implementovaná aplikace byla testována na testovacím modelu a následně také, ve spolupráci s Mgr. Antonínem Míškem, Ph.D., přímo ve firmě Škoda Auto. Implementovaná aplikace je plně funkční jak v desktopových tak i v mobilních prohlížečích a splňuje všechny požadavky ze zadání i požadavky firmy Škoda Auto.

## Možná rozšíření

Pokud se podaří opravit chybu v knihovně VRUT, která znemožňuje export cube map do VRUT souboru, tak bude možné implementovat efekt odlesků prostředí.

Další možnosti rozšíření aplikace jsou: implementace ořezání pohledovým jehlanem, vrhání paprsku pro přesný výběr středu rotace nebo implementace vizuálních markerů pro zájmové objekty scény a lepší interakci mezi uživatelem a modelem.

# Literatura


[1] Anderson, S. E.: Bit Twiddling Hacks. [online], 2005-05-05 [cit. 2016-05-15], Dostupné z: http://graphics.stanford.edu/~seander/bithacks.html.

[2] Giles, T., Parisi, T.: Learning WebGL. [online], 2013 [cit. 2016-05-15], Dostupné z: http://learningwebgl.com.

[3] Kovács, E.: Rotation about an arbitrary axis and reflection through an arbitrary plane. *Annales Mathematicae et Informaticae*, ročník 40, 2012: str. 75–186, dostupné z: http://ami.ektf.hu/uploads/papers/finalpdf/AMI_40_from175to186.pdf.

[4] Kyba, V.: *Modulární 3D prohlížeč*. Diplomová práce, České vysoké učení technické v Praze, Fakulta elektrotechnická, 2009 [cit. 2016-01-08], dostupné z: http://dip.felk.cvut.cz/browse/pdfcache/kybav1_2009dipl.pdf.

[5] Kyba, V., Míšek, A., aj.: Dokumentace aplikace VRUT. 2011, [svn databáze projektu].

[6] Mozilla Foundation: Mozilla Developer Network. [online], 2015-2016 [cit. 2016-05-15], Dostupné z: http://developer.mozilla.org/cs/.

[7] Overvoorde, A.: OpenGL. [online], 2012-2016 [cit. 2016-05-15], Dostupné z: http://open.gl/.

[8] Parisi, T.: *WebGL: Up and Running*. O'Reilly Media, 2012, iSBN 978-1-449-32357-8.

[9] Stefanov, S.: 3 ways to define a JavaScript class. [online], 2006-09-29 [cit. 2016-05-15], Dostupné z: http://www.phpied.com/3-ways-to-define-a-javascript-class/.

[10] Villamor, C., Willis, D. a Wroblewski, L.: Touch Gesture reference guide. [online], 2010-04-15 [cit. 2016-05-15], Dostupné z: http://static.lukew.com/TouchGestureGuide.pdf.

[11] Čejka, J.: *Novodobé OpenGL a jeho implementace do knihovny VRUT*. Diplomová práce, Masarykova univerzita, Fakulta informatiky, Brno, 2013 [cit. 2016-01-08], dostupné z: http://is.muni.cz/th/324987/fi_m/.

# Přílohy

# Seznam příloh

# Příloha A

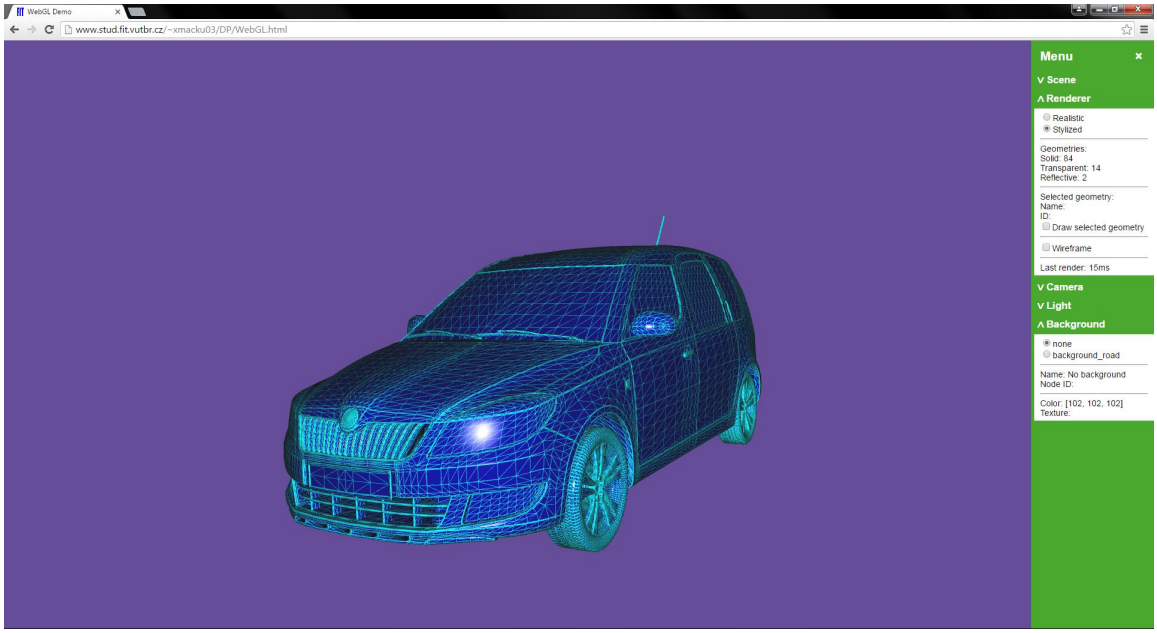
# Obsah CD

- source\– složka se zdrojovými soubory aplikace
- tex\– složka se zdrojovými soubory technické zprávy
- screens\– složka se snímky aplikace
- WebGL.wmv – ukázkové video aplikace
- diplomova_prace.pdf – elektronická verze technické zprávy

# Příloha B

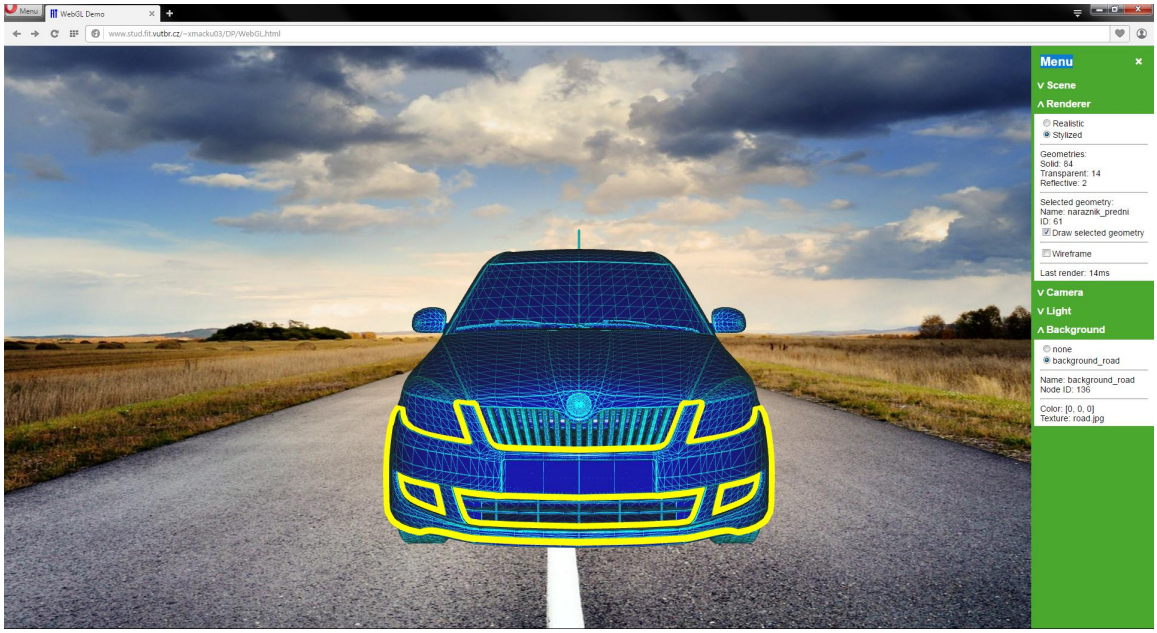
# Snímky z testování


Obrázek B.1: Realistické zobrazení v prohlížeči Google Chrome 50

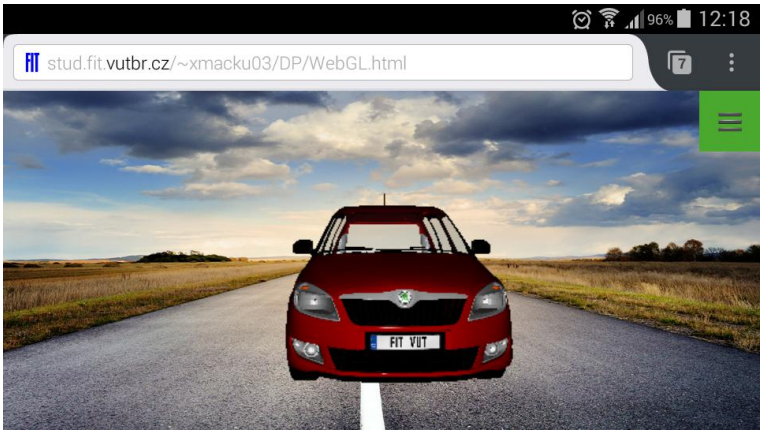



Obrázek B.2: Stylizované zobrazení v prohlížeči Google Chrome 50


Obrázek B.3: Realistické zobrazení v prohlížeči Opera 37

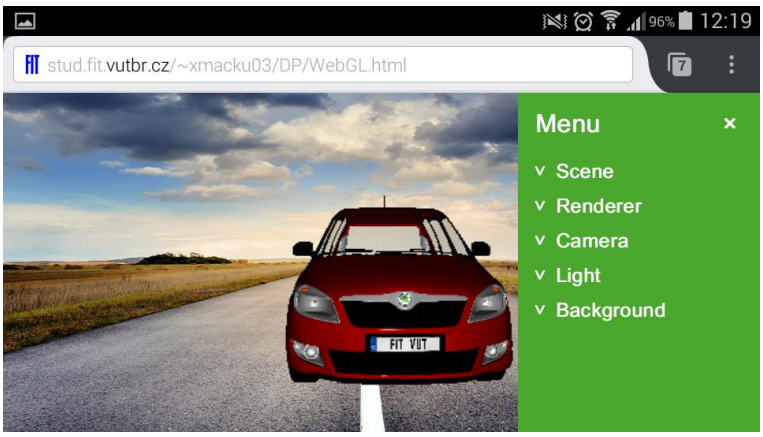



Obrázek B.4: Stylizované zobrazení v prohlížeči Opera 37

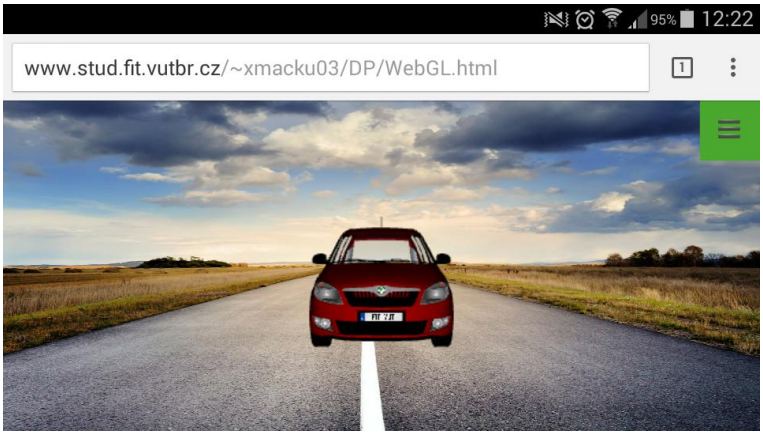



Obrázek B.5: Aplikace spuštěná v prohlížeči Firefox for Android 46, orientace na šířku

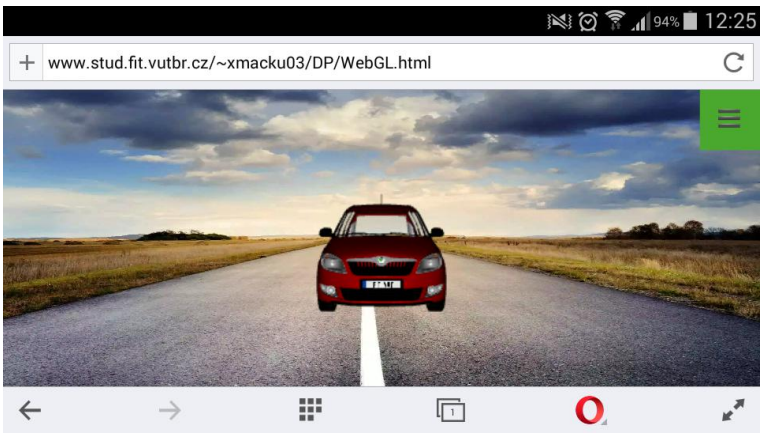

Obrázek B.6: Aplikace spuštěná v prohlížeči Firefox for Android 46, orientace na výšku




Obrázek B.7: Aplikace spuštěná v prohlížeči Firefox for Android 46, s otevřeným menu




Obrázek B.8: Aplikace spuštěná v prohlížeči Chrome for Android 50




Obrázek B.9: Aplikace spuštěná v prohlížeči Opera Mobile 37