VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zobrazování 3D modelů pomocí WebGL

Displaying 3D Models in Web Pages using WebGL

2015 Jakub Imrich

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jakub Imrich**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Zobrazování 3D modelů pomocí WebGL
Displaying 3D Models in Web Pages using WebGL

Zásady pro vypracování:

V současné době rostou možnosti dnešních moderních webových prohlížečů. Můžeme je využívat nejen k zobrazování textu a obrázk;, ale dnes už plně podporují interaktivní 3D obsah. Cílem této práce je navrhnout a implementovat řešení, při kterém bude uživateli umožněno interaktivně zobrazovat 3D scény s možností pohybu. Používané scény budou vytvářené v externích modelovacích nástrojích (např. Blender, 3D studio apod.), ze kterých se budou ve vhodném formátu exportovat tak, aby je bylo možné následně interaktivně prohlížet ve webovém prohlížeči.

1. Nastudujte problematiku zobrazování 3D objektů v moderních webových prohlížečích. Zaměřte se zejména na využití technologie WebGL.
2. Vyberte vhodný formát pro export modelů a scén, který bude umožňovat následné interaktivní prohlížení ve webových prohlížečích.
3. Vytvořte interaktivní aplikaci využívající technologii WebGL tak, aby bylo možné výslednou scénu načíst a následně interaktivně prohlížet (procházet) v moderních webových prohlížečích.
4. Zaměřte se na možnosti zobrazování 3D scén z oblasti architektury (modely domů, zahrad apod.).
5. Celý postup včetně vývoje aplikace pečlivě zdokumentujte, aby v práci bylo možné dále pokračovat.

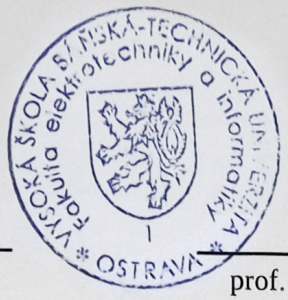
Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Martin Němec, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 07.05.2015



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 07. května 2015 .....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 07. května 2015 .....

Na tomto mieste by som rád poďakoval svojmu vedúcemu bakalárskej práce Ing. Martinu Němcovi, Ph.D za jeho rady, odbornú pomoc a konzultácie, ktoré mi poskytol pri tvorbe tejto práce.

Abstrakt

Hlavným cieľom tejto práce je vytvoriť aplikáciu, ktorá bude zobrazovať interaktívny 3D obsah. Aplikácia je navrhnutá pre webové prostredie a na zobrazenie využíva technológiu WebGL. Hlavné zameranie aplikácie je na zobrazenie scén z oblasti architektúry. V prvej časti práce su opísané možnosti zobrazenia 3D obsahu vo webovom prostredí. V ďalšej časti je teoretický popis použitých algoritmov pre zobrazenie, postprocessing a algoritmov pre identifikáciu objektov. Posledná časť je venovaná návrhu a praktickej implementácii aplikácie.

Klíčová slova: WebGL, 3D, Three.js, identifikácia objektov, postprocessing, architektonické vizualizácie

Abstract

The main objective of this work is to create an application that will display interactive 3D content. The application is designed for web environment, and use WebGL technology. The main focus of the application is to display architecture visualization. First part of work describe the possibility of displaying 3D content in web environment. Below is a theoretical description of the algorithms for displaying, postprocessing and object identification. The last part is dedicated to the development and practical implementation of the application.

Keywords: WebGL, 3D, Three.js, object identification, postprocessing, architectural visualizations

Seznam použitých zkratek a symbolů

HTML	–	Hyper Text Markup Language
API	–	Application programming interface
ASP	–	Active Server Pages
DOM	–	Document Object Model
GLSL	–	OpenGL Shading Language
2D	–	Two dimensional
3D	–	Three dimensional
GPU	–	Graphics processing unit
CPU	–	Central processing unit
GUI	–	Graphical User Interface

Obsah

Zoznam tabuliek

Zoznam obrázkov

Seznam výpisů zdrojového kódu

1 Úvod

V súčasnosti sa integrácia grafických kariet v bežne dostupných zariadeniach stala štandardom. Preto môžme počítať stím, že výkonný hardware pre zobrazovanie grafiky sa nachádza v množstve zariadení. To nám umožňuje vytvárať graficky bohatšie aplikácie. Jedno z ich množstva využití sú interaktívne vizualizácie architektúry nielen ako desktopové aplikácie ale aj ako webové aplikácie. V minulosti boli architektonické vizualizácie na webe prezentované staticky, teda ako obrázky, prípadne ako video. Interaktívne vizualizácie v reálnom čase nemali dostatočnú kvalitu zobrazenia a pre svoj chod potrebovali množstvo pluginov do webového prehliadača. S príchodom WebGL a jeho následným integrovaním do webových prehliadačov sa situácia zmenila a vývojári majú k dispozíci plnohodnotné API pre vývoj grafických aplikácii vo webovom prostredí. Preto som sa rozhodol vytvoriť aplikáciu opísanú v tejto práci. Hlavnou motiváciou bolo vytvoriť aplikáciu umožnujúcu užívateľovi interaktívne si prehliadať dom vo webovom priehliadači.

2 Aktuálny stav

V súčastnej dobe je zobrazovanie interaktívnej grafiky na webe možné pomocou rôznych technológii. V kapitole popíšeme najznámejšie a najpoužívanejšie znich a zameráme sa na ich možnosti pri použití z hľadiska interaktívnej prezentácie architektúry.

2.1 3D vo webovom prehliadači

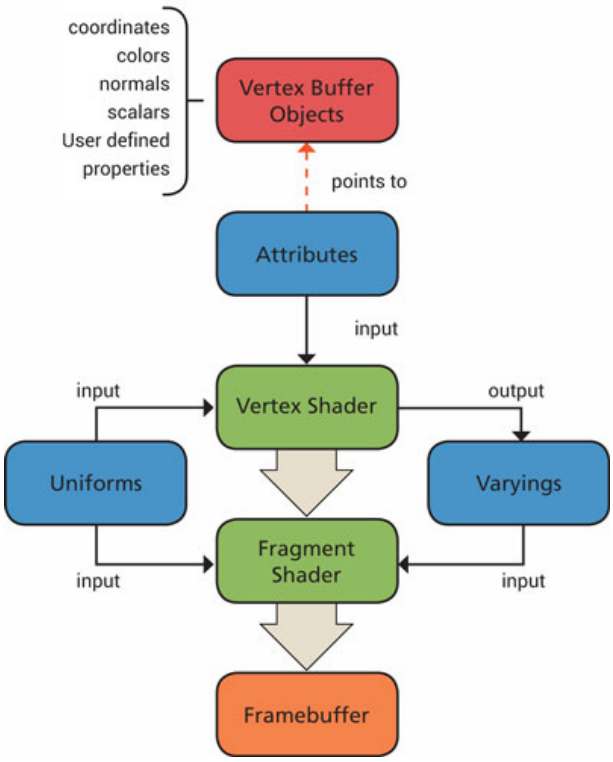
Zobrazenie 3D obsahu vo webovom prehliadači bolo donedávna komplikované. Vzhľadom nato, že HTML štandard priamo nepodporuje 3D obsah bolo potrebné použiť rôzne zásuvne moduly (plugin). Jednou z možností bol modul Adobe Flash [1], ktorý bol primárne určený na zobrazovanie 2D vektorovej grafiky. Postupom času mu bola pridaná možnosť zobrazovať aj 3D grafiku, avšak možnosti, ktoré ponúkal oproti desktopovým aplikáciam boli značne obmedzené. V súčasnej dobe je technológia Flash na ústupe a je nahradzovaná HTML5. Na možnosti technológie Flash odpovedala firma Microsoft vývojom technológie Silverlight [2] pre ASP.Net. Tá taktiež umožnovala zobrazovať 3D obsah, pričom ju postihli rovnaké problémy ako Flash. V súčasnosti je vývoj technológie Silverlight ukončený. Problémom týchto technológii bolo, že fungovali ako pluginy do prehliadačov, čo spôsobovalo problémy užívateľom a aj vývojarom. A preto bolo požadované vytvorenie jednotné štandardu pre zobrazovanie 3D grafiky vo webovom prostredí. To vyústilo k vzniku WebGL.

2.2 WebGL

Web-based Graphics Library, skrátene WebGL je JavaScript API, ktoré nám umožnuje implementovať interaktívnu 3D prípadne 2D grafiku priamo do intrenetového prehliadača bez použitia dodatočných pluginov. WebGL rieši dlhoročný problém ako vyriešiť zobrazovanie interaktívnej grafiky priamo vo webovom prehliadači. V minulosti bolo nutné pre zobrazenie interaktívnej grafiky doinštalovať do priehliadača požadované pluginy a rozšírenia. WebGL je plne integrované do prehliadača a umožnuje komunikáciu a interakciu s inými prvkami HTML pomocou DOM. Kód WebGL aplikácie pozostáva z logickej časti napísanej v JavaScripte a shaderov napísaných v jazyku GLSL, ktoré sa spúšťajú na grafickej karte.

API vychádza z OpenGL ES 2.0. OpenGL ES je adaptácia dlhodobo ustáleného 3D štandardu OpenGL, ES znamená embedded systems, čiže prispôsobené na používanie na „mobilných“ zariadeniach ako sú mobilné telefóny, tablety, prenosné zariadenia... Vývojary WebGL sa rozhodli výchadzať z OpenGL ES aby mohli vytvoriť cross-platform a cross-browser API. WebGL vyvýja Kronos Group[3]. Štandard OpenGL ES 2.0 bol pre WebGl zvolený pre jeho jednoduchosť a hlavne pre jeho možnosť presunúť väčšinu výpočtov a funkcionality na grafickú kartu, a tým ušetriť datové prenosy medzi aplikáciou a grafickou kartou. Preklad WebGL príkazov prebieha v prehliadačoch Chrome a Firefox pomocou implementácie projektu ANGLE (Almost Native Graphics Layer Engine [4]), ktorý na operačných systémoch Windows prekladá OpenGL príkazy na DirectX príkazy.

Prehliadače Internet Explorer a Opera majú implementované svoje vlastné prekladače z OpenGL do DirectX. Na operačnom systéme Linux je využívaná podpora OpenGL.



Obr. 1: WebGL pipeline [3].

2.3 Knižnice pre prácu s WebGL

Keďže WebGL je low-level API je programátor nútený písať množstvo repetitívného kódu. Vývoj interaktívnej grafickej aplikácie je náročný proces, v ktorom treba vyriešiť množstvo problémov, preto pri práci s WebGL je vhodné použiť niektorú s open source knižníc, ktoré zaobaľujú WebGL API a zjednodušujú vývojarovi prácu, aby sa mohol sústrediť na vývoj samotnej aplikácie. S príchodom WebGL v roku 2012 sa začalo z vývojom niekoľkých knižníc. Ako to však býva, prvotné nadšenie rýchlo opadlo a vývoj mnohých knižníc bol ukončený (napríklad CibicVR [6]). Do dnešných dní v podstate prežili dve použiteľné knižnice. V dobe písania bakalárskej práce došlo k zaujímavej situácii, keďže bol Unreal Engine 4 [7] uvoľnený pre nekomerčné použitie zdarma. Jednou z možnosti enginu je aj export do WebGL.

2.3.1 GLGE

GLGE [5] sa snažil jeho vývojar orientovať ako engine orientovaný na tvorbu 3D hier v prehliadači. GLGE umožnuje správu scény, animácie, generovanie terénu, materiály. Me-

dzi jeho hlavné prednosti patrí implementovaná podpora výpočtu fyziky a podpora pre výpočet kvalitných tieňov. Bohužial aktuálne je vývoj enginu pozastavený, a v súčasnosti je len sporadicky udržiavaný komunitou.

2.3.2 Three.js

Three.js [8] je open-source JavaScript 3D Engine. Three.js ponúka high-level prístup pri tvorbe aplikácii založených na WebGL. Three.js zaobaľuje WebGL API a reprezentuje 3D scénu pomocou objektov, materialov a svetiel so zameraním na objektovo-orientované programovanie. Three.js je rýchle a optimalizované, čo umožnuje programátorom vytvorit detailnejšiu a bohatšiu grafiku a prostredie.

Obsahuje podporu pre prácu s množstvo 3D formátov (.3ds, JSON, OBJ, collada...) a obsahuje pluginy pre export modelov a scény s 3D modelovacích nástrojov (Blender, 3Dsmax). Three.js je open-source a je jednoducho rozšíritelný. Three.js momentálne predstavuje najrozšírenejšiu knižnicu pre prácu s WebGL. Pôvodne z jednoduchého projektu sa stal robusný nástroj, ktorý pre svoje projekty využívaju Autodesk, WarnerBros, Denim alebo Microsoft (ukážky projektov sú dostupné na [8]). Pre prácu s WebGL a Three.js bolo napísaných niekoľko publikácii napríklad *Three.js Essentials* [9] alebo *WebGL Up and Running* [10]. Knižnica Three.js vo verzii *r.68* použitej v tejto práci má niekoľko kľúčových vlastností:

- Práca so scénou, umožnujúca pridávanie a odoberanie objektov počas behu aplikácie.
- Kamery: perspektívna, ortografická
- Morfologické a keyframe animácie
- Osvetlenie a tieňe.
- Podpora materiálov a textúr.
- Shadre a postprocessing efekty.

Treba si uvedomiť, že knižnica Three.js nieje plnohodnotný „game engine“, čiže veľa vecí si budeme musieť vytvoriť a implementovať sami. Taktiež neobsahuje podporu pre sieťový kód a fyzikálny engine. Knižnica je stále vo vývoji, a novšie verzie nezaručujú spätnú kompatibilitu.

3 Úvod do teórie

Pri vykresľovaní a spracovaní výsledného obrazu je nutné použiť množstvo algoritmov. V tejto kapitole si objasníme princíp algoritmov a techník, ktoré sú implementované v práci a stručne si priblížime alternatívy k použitým algoritmom.

3.1 Ambient Occlusion

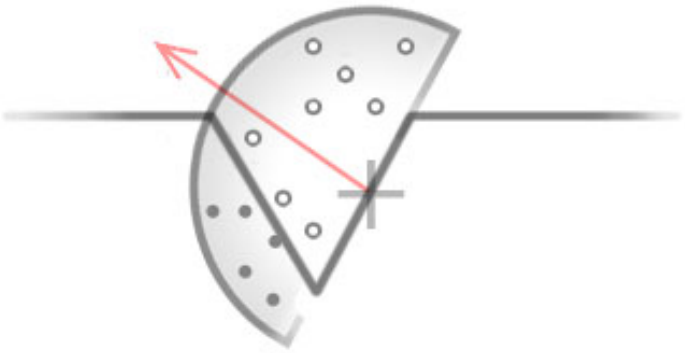
Ambient Occlusion (zatienenie okolia) technika pre určenie útlmu svetla v určitom bode scény. K útlmu dochádza ak je bod zatienený časťou geometrie, ktorá bráni prístupu osvetlenia. Ambient Occlusion pomáha vo vykreslenom obraze lepšie pochopiť hĺbku scény a tvar objektov. Navyše pridáva do obrazu kontaktne tiene a ztmavenie v rohoch geometrie. Algoritmus pre výpočet Ambient Occlusion prebieha tak, že v každom bode P uvažujeme hemisféru ω ktorej plocha tvorí hranicu pre testovanie prítomnosti geometrie. Následne z bodu P vedieme lúče. Ak sa lúč nepreťal so žiadnou geometriou, je bod P v danom smere lúča dostupný osvetleniu. Výslednu hodnotu zatienenia v bode P s normálou n môžme vyjadriť pomocou rovnice:

$$A(P,n) = \frac{1}{\Pi}\int_{\Omega} V(P,\omega)max(n.\omega,0)d\omega \tag{1}$$

ω predstavuje všetky lúče v hemisfére. V je funkcia pre určenie dostupnosti svetla, ktorá vracia hodnotu 0 ak lúč v hemisfére nenarazí na žiadnu geometriu. Hodnotu 1 nadobudne vtedy, ak sa lúč v hemisfére pretne s geometriou. Skalárnym súčinom $n.\omega$ určíme intenzitu zatienenia v závislosti od odklonu lúču od normály.

3.1.1 Screen Space Ambient Occlusion

SSAO je implementácia Ambient occlusion pre real-time zobrazenie. Základná idea SSAO je, že použijeme hĺbku scény ako aproximáciu geometrie a vypočítame AO v screen-space, to znamená že celý výpočet prebehne pomocou fragment shaderu v GPU a je úplne nezávislý od komplexnosti a osvetlenia scény. Implementovaná verzia SSAO používa kelner orientovaný podľa normály v danom fragmente. Hĺbka scény je meraná z pohľadu kamery, preto sa pozícia a velkosť „tieňov“ mení v závislosti od pozície a orientácie kamery. Aby bolo možné dosiahnúť použiteľný framerate, nieje možné vzájomne porovnávať všetky pixely, preto sa používa náhodné vzorkovanie v kelnery (viz. obrázok č.2). Náhodnosť a obmedzený počet vzorkov vedie ku vzniku šumu vo výslednom obraze, ktorý sa často odstraňuje rozmazaním výsledného SSAO efektu. Pre podrobnejšie informácie o Screen Space Ambient Occlusion navštívte web ([13]).



Obr. 2: SSAO kelner orientovaný podľa normály [12].

3.1.2 Horizont-Based Ambient Occlusion

HBAO [16] algoritmus, je podobný algoritmu SSAO, preto pre svoj výpočet potrebuje poznať hodnotu hĺbky a normály. Algoritmus určí hodnotu zatienia pomerom zatienenia hemisféry v aktuálnom bode k celkovej ploche hemisféry. Narozdiel od SSAO následne algoritmus aproximuje vypočítanu hodnotu zatienenia prechádzaním hodnôt hĺbky od aktuálneho bodu v niekoľkých smeroch,a hľadá hodnoty najbližie k pozorovateľovi. Nájdene hodnoty nazývame „horizonty“ a tvoria výseče v hemisfére. Výpočet algoritmu porebieha v view-space priestore, teda musíme aktuálny pixel previest z screen-space pomocou projekčnej matice a uv súradnic do view-space.

3.2 Motion Blur

Motion blur je efekt simulujúci rozmazanie pohybujúcich sa objektov, prípadne rozmazanie obrazu v závislosti od pohybu kamery. Motion blur je jeden z dôležitých efektov, ktoré pomáhajú zvýšiť realizmus vykreslovanej scény. Rozmazanie obrazu pri pohybe je pre ľudské oko prirodzené. Táto „nedokonalosť“ vnímania je spôsobená zotrvačnosťou sietnice v oku. Bunky v sietnici potrebujú určitý čas nato aby zaregistrovali zmenu intenzity svetla. Takže ak sa objekt pohybuje dostatočne rýchlo, sietnica zaregistruje len čiastočnú zmenu intenzity svetla a mozog interprétuje obraz ako rozmazaný. Ak sa objekt pohybuje príliš rýchlo, sietnica zmenu nezachytí a objekt zostane pre nás „neviditeľný“, ako napríklad vystrelený projektil. Motion blur je typický aj pre film, kde vzniká veľmi podobne ako v oku, len stím rozdielom, že sietnica je nahradená čipom, prípadne filmovým pásom. V 3D aplikácii pomáha rovnako ako vo filme pomáha zvýšiť vnímanie plynulosti pohybu, hlavne ak je snímkova frekvencia nižšia ako 30 snímkov za sekundu.

3.2.1 Simulácia Motion blur efektu pomocou akumulácie obrazu

Ak chceme dosiahnúť simuláciu motion blur efektu, tak prirodzene nás napadne napodobniť proces jeho v vzniku v kamere. Vo filmovej kamere vzniká tak, že pohybujúci sa

objekt postupne osvetľuje jednotlivé časti čipu. Množstvo rozmazania obrazu je závisle od dĺžky uzavierky obejktívu. V počítačovej grafike docielime tento proces tak, že jednotlivy snímkok sa bude skladať z podsnímkov. Každý podsnímok vykresľuje snénu v určitom čase. Spriemerovaním hodnôt vykreslených podsnímkov dosiahneme výsledny snímok za daný časový úsek, v ktorom bude zachytený pohyb objektov. Táto technika sa využíva najmä pri renderovaní animovaných filmov, prípadne statických obrazkov. Pre dosiahnutie dobrej kvality rozmazania potrebujeme vykresliť približme 16 až 32 podsnímkov. Veľke množstvo vykreslených podsnímkov je hlavny dôvod prečo sa tento postup často nepoužíva v real-time aplikáciach, pretože ich vykreslenie nám zaberie množstvo výpočtového výkonu.

3.2.2 Motion Blur ako post-processing efekt

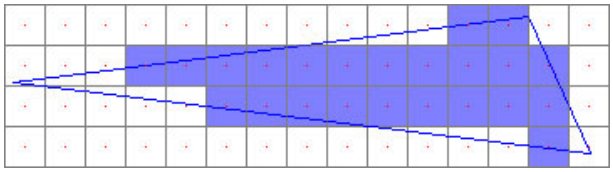
K problému simulácie Motion blur efektu môžme pristupovať ako k post-postprocessing efektu [17] a využiť možnosti moderných grafických kariet. Algoritmus vypočíta world-space pozíciu pre každý pixel, ktorú určí z hĺbky a projekčnej matice aktuálneho snímku. Následne keď poznáme world-space súradnice pre daný pixel môžme ich transformovať pomocou projekčnej matice z predchádzajúceho snímku. To nám umožní vypočítať rozdiel medzi aktuálnym a predchádzajúcim snímkom a určiť ako sa daný pixel pohyboval. Keď poznáme smer a veľkosť pohybu pre daný pixel, tak motiom blur efekt docielime sčítaním farby pixelov v smere pohybu a ich následným spriemerovaním. Výhody techniky: funguje ako post-processing effect , takže sa dá jednoducho implementovať do zobrazovacieho reťazca. Technika je nenáročná na výpočet.

Nevýhody techniky: motion blur funguje len pri pohybe kamery. Ak je kamera staticka a objekt sa pohybuje, efekt sa nedostaví.

Obr. 3: Motion Blur

3.3 Techniky vyhladzovania hrán v obraze

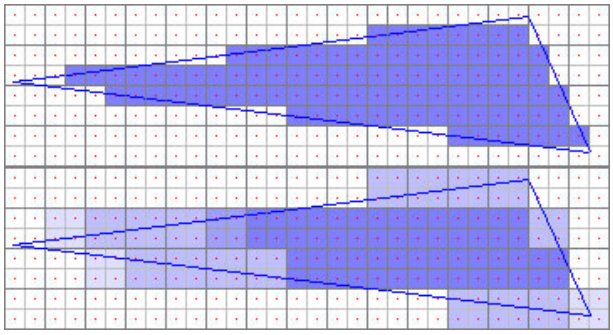
Pri prevode trojrozmerného obrazu na dvojrozmerný, tak aby mohol byť vykreslený na monitore, sa používa rasterizácia. Rasterizácia je reprezentácia trojrozmerných objektov pomocou množiny pixelov. Jedná sa o diskretizáciu útvarov, pri ktorej je potrebné nájsť všetky pixely, ktoré daný útvar reprezentujú, a následne im priradiť farbu daného útvaru. Keďže množina pixelov je konečná, nedokážeme dostatočne presne zobraziť daný útvar a preto dochádza k deformáciam a nedokonalostiam. Napríklad iné než horizontálne a vertikálne krivky sa v procese rasterizácie znehodnotia do typických „schodov“. Existuje niekoľko techník a algoritmov ako tieto nedostatky odstrániť, prípadne zamaskovať aby vo výslednom obraze nepôsobili rušivo.



Obr. 4: Princíp rasterizácie obrazu

3.3.1 Oversampling a Supersampling

Oversampling je základnou metódou antialiasingu. Z princípu rasterizácie obrazu vyplýva, že čím budeme mať vyšie rozlíšenie, tím presnejší bude výsledný obraz. Vyššie rozlíšenie dosiahneme buď hardwarovo, napríklad výmenou monitora alebo použijeme Oversampling algoritmus, ktorý softwarovo navýši rozlíšenie. Algoritmus rozdelí každý pixel na štvrtiny a celú scénu vyrenderuje v dvojnásobnom rozlíšeni do pamäte. Vo vykreslenom obraze sa stále nachádzajú nedostatky, ale už niesu tak závažné. Následne algoritmus spriemeruje farby štyroch susedných pixelov a scénu prevedie späť do pôvodného rozlíšenia.



Obr. 5: Aplikácia Oversampling algoritmu

Vďaka spriemerovaniu hodnôt pixelov je výsledný obraz (viz. obrázok č.5) vernejší oproti pôvodnému obrazu (viz. obrázok č.4). Hlavnou nevýhodou Oversamplingu je jeho

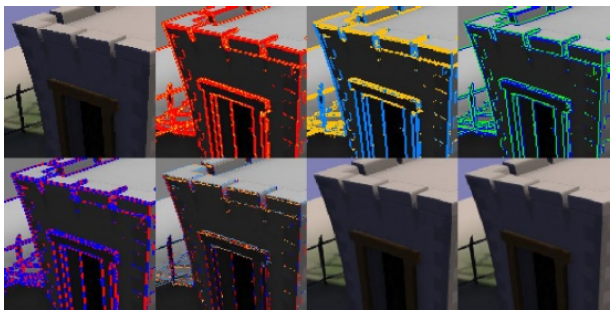
harwarová náročnosť, kedže sa obraz vykresľuje v dvojnásobnom rozlíšeni, čo má za následok zníženie počtu vykreslených snímkov za sekundu. Aby sa harwarové nároky znížili musí sa použiť optimalizácia algoritmu. Jednou z možných optimalizácii je Supersampling, ktorý vychádza z princípu, že nieje nutné počítať farbu každého subpixelu, ale postačí ak sa vypočítaju hodnoty pixelov po uhlopriečke. Znížením počtu počítaných pixelov na polovicu docielime značné úspory výpočtového výkonu, pri zachovaní porovnateľnej kvality obrazu.

3.3.2 Multisampling

Multisampling je ďalšou možnosťou optimalizácie Oversampling algoritmu, ktorý vo vysokom rozlíšení renderuje celú scénu. Oproti tomu Multisampling renderuje vo vysokom rozlíšení iba blízke oklie hrán, čím dochádza k úspore výpočtového výkonu. Nevýhodou algoritmu je, že nevyhladzuje textury. Tento nedostatok sa prejaví najmä pri transparentných textúrach, napríklad zobrazujúcih vegetáciu alebo pletivo plotu. Rôzne derivácie Multisampling algoritmu sa snažia odstrániť jeho nedostatky, medzi najznámejšie patria Coverage Sampling Antialiasing (CSAA [14]) pre grafické karty nVidia alebo veľmi podobný Custom Filter Antialiasing (CFAA) pre grafické karty od AMD.

3.3.3 Fast approximate anti-aliasing

FXAA [15] predstavuje uplne rozdielný prístup k riešeniu problému aliasingu. Bežné Antialiasing(viz. kapitolu č.3.3.1 , 3.3.2) algoritmy vstupujú do procesu tvorby obrazu už v priebehu rasterizácie, pričom odstránenie aliasingu dosiahnú pomocou vyrendrovania obrazu vo vyššom rozlíšeni. Oproti nim je FXAA morfologický algoritmus, ktorý spracuvá výsledný obraz pomocou fragment shaderu na GPU. Algoritmus funguje ako hranový detektor, ktorý vyhľadáva v obraze hrany typické pre aliasing a následne ich vyhladí (rozmaže). Výhodou techniky je, že je plne nezávisla od počtu polygonov v scéne a je nenáročna na množstvo použitej pamäte.

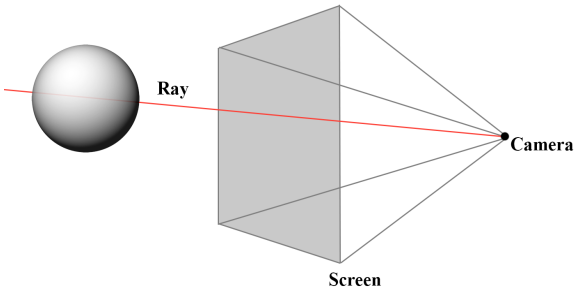


Obr. 6: FXAA [15]

3.4 Identifikácia objektov

V interaktívnych grafických aplikáciach, musíme riešiť problematiku interakcie užívateľa s 3D objektami v scéne, ktorá najčastejšie prebieha pomocou myši, klávesnice alebo dotykovej obrazovky. Problém je, že obraz ktorý užívateľ vidí je dvojrozmerný, teda aj súradnice kurzora myši sú dvojrozmerné. Aby sme zistili kde užívateľ klikol musíme proces vykreslovania (projection 3D to 2D) vykonať odzadu (uprojection 2D to 3D). Existuje niekoľko metód a postupov, ktoré nám to umožnia.

3.4.1 Raycasting



Obr. 7: Raycasting

Raycasting je metóda, ktorá pri zisťovaní výberu objektu používa lúče (rays). Z pozície kde užívateľ klikol (2D súradnice kurzora myši) vyšle lúč v smere pohľadu kamery. Následne sa hľadá priesečník medzi objektom v scéne a lúčom.

Výhody:

- Vieme zistiť presnú pozíciu priesečníku medzi objektom a lúčom, teda nielež vieme, že sa na objekt kliklo ale vieme aj presne kde.
- Scéna može obsahovať neobmedzený počet objektov. THREE.js obsahuje implementovaný raycaster a potrebné funkcie pre zistenie priesečníkov.

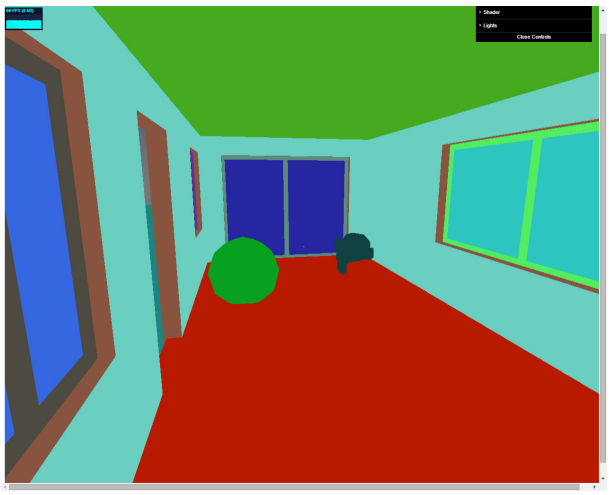
Nevýhody:

- Raycasting je výpočetne náročný, a to hlavne pri komplexných scénach.
- Náročnejšia implementácia.
- Výpočet vykonáva CPU.

Aj pre zjavné výhody sa „čistý“ raycasting v realtime aplikáciach pre výber objektov nepouživa, práve kôli svojej náročnosti na výpočtový výkon. Ak sa ho rozhodneme použiť je nutné implementovať urýchlovacie algoritmy.

3.4.2 Color picking

Každému objektu v scéne sa na základe jeho ID priradí unikátna farba. Následne sa celá scéna vykreslí do buffru, a pomocou funkcie *glReadPixel()* z pozície pod kurzorom myši sa prečíta farba pixelu. Spätne z farby pixelu určíme ID objektu. Color picking je oveľa vhodnejšia technika na výber objektov než raytracing, pretože jej nároky na výkon a implementáciu sú nízke.



Obr. 8: Ukážka algoritmu Color picking

1. Výhody
   - Jednoduchá implemetácia.
   - Vysoký výkon aj pri komlexnej scéne.
   - Možnosť vybrať viacero objektov (area selection) bez dopadu na výkon.
2. Nevýhody
   - Každý pixel nesie informáciu len o jednom objekte.
   - Počet objektov je obmedzený farebnou hĺbkou, štandardne na 255^3

3.4.3 Priesečník priamky a trojuholníka

Algoritmus prechádza geometriu každého objektu v scéne, pričom prejde každú stenu (face) objektu a vyhodnotí či došlo k pretnutniu lúčom. V prvom bode algoritmu zistíme z akých vrcholov sa stena (face) skladá, a následne prevednieme súradnice vrcholov z Object-Space do World-Space. Z vrcholov vytvoríme vektory $\vec{u}$, $\vec{v}$, ktoré nám definujú hrany trojuholníka. Ich vektorovým súčinom získame normálu $\vec{n}vbodeV_0$.

$$\vec{n} = \vec{u} \times \vec{v} \tag{2}$$

V dalšom kroku definujeme vektor $\vec{dir}, ktorurujesmerlu(vnaomprpadesajednosmerovvektorkamery).Nsle$

$$\vec{w_0} = origin - V_0 \tag{3}$$

Pomocou skalárneho súčinu normály $\vec{n}$ a vektoru $\vec{w}0$ vypočítame skalár a.

$$a = \vec{n}.\vec{w}_0 \tag{4}$$

Skalárnym súčinom normály a smerového vektoru lúču vypočítame skalár b.

$$b = \vec{n}.\vec{dir} \tag{5}$$

Z definície skalárneho súčinu

$$\vec{u}.\vec{v} = |\vec{u}||\vec{v}| \cos a \tag{6}$$

nám v algoritme vyplývajú tieto skutočnosti:

- Ak je skalár b rovný nule tak lúč je paralelný s rovinou tvorenou trojuholníkom a nachádza sa mimo trojuholníku.
- Ak sú skalár b a skalár a rovné nule tak je lúč je rovnobežný s rovinou trojuholníku a prechádza celou rovinou.

V prvom prípade k pretnutiu nikdy nedôjde a môžme ukončiť iteráciu cyklu. V druhom prípade dôjde k pretnutiu nekonečne krát, a v takejto situácii sa ukončí iterácia cyklu, pretože algoritmus nedokáže jednoznačne určiť kde k pretnutiu došlo.

Z predchádzajúcich výpočtov dokážeme určiť či náš lúč pretne rovinu definovanú vektormi $\vec{u}$, $\vec{v}$. Uvažujme náš lúč ako polpriamku zo začiatkom v bode $origin$ a smerom $\vec{dir}$. Parametricky to môžme vyjadrit ako

$$P(r) = origin + r(\vec{dir}) \tag{7}$$

Parameter r v bode priesečníku s plochou vypočítame ako

$$r_i = (\vec{n}.a)/(\vec{n}.b) \tag{8}$$

Ak je $r_i < 0$ iterácia cyklu končí pretože nedôjde k pretnutiu lúča s plochou. Keď poznáme parameter r_i pre bod priesečníku s plochou vieme dosadením do parametrickej rovnice polpriamky vypočítať súradnice priesečníku.

Následne keď poznáme súradnice priesečníku s plochou, musíme vypočítať či daný priesečník leží vo vnútri trojuholníka. Na výpočet využijeme barycentricke súradnice [11], ktoré určujú bod v rovine pomocou lineárnej kombinácie troch dalších bodov ležiacich v rovnakej rovine. Pre náš bod priesečníku platí

$$I = a_1 * V_0 + a_2 * V_2 + a_3 * V_3 \tag{9}$$

pričom platí $a_1 + a_2 + a_3 = 1$ a $0 \leq a_1, a_2, a_3 \leq 1$.

Pomocou parametrickej rovnice

$$V_(s,t) = V_0 + s(V_1 - V_0) + t(V_2 - V_0) = V_0 + s\vec{u} + t\vec{v} \tag{10}$$

môžme definovať akýkoľvek bod ležiaci v rovine. Ak dodržíme $s + t \leq 1$ a $s, t <= 1$ tak môžme definovať akýkoľvek bod v trojuholníku danom vrcholmi V_0, V_1, V_2.

Na výpočet barycentrických súradnic pre bod I si musíme najskôr definovať vektor $\vec{p}$, ktorý sa nachádza v rovine tvorenou trojuholníkom a je kolmný na vektor $\vec{v}$ a normálu $\vec{n}$ pre ktoré plati $\vec{n}.\vec{v} = 0$

Vektor $\vec{p}$ definujeme ako

$$\begin{aligned} \vec{p} &= \vec{n} \times \vec{v} && (11)\\ \vec{p}.\vec{v} &= 0 && (12) \end{aligned}$$

Vektor $\vec{q}$ definujeme ako

$$\begin{aligned} \vec{q} &= \vec{n} \times \vec{u} && (13)\\ \vec{q}.\vec{u} &= 0 && (14) \end{aligned}$$

Následne si definujeme vektor $\vec{w}$

$$\vec{w} = I - V_0 \quad (15)$$

a môžme riešiť rovnicu pre vektor $\vec{w}$, v ktorej na obe strany dosadíme $\vec{p}$ a dostaneme

$$\begin{aligned} \vec{w} &= s\vec{u} + t\vec{v} && (16)\\ \vec{w}.\vec{p} &= s\vec{u}.\vec{p} + t\vec{v}.\vec{p} && (17)\\ \vec{w}.\vec{p} &= s\vec{u}.\vec{p} && (18) \end{aligned}$$

Postup opakujeme pre $\vec{q}$ a dostaneme

$$\begin{aligned} \vec{w} &= s\vec{u} + t\vec{v} && (19)\\ \vec{w}.\vec{q} &= s\vec{u}.\vec{q} + t\vec{v}.\vec{q} && (20)\\ \vec{w}.\vec{q} &= t\vec{v}.\vec{q} && (21) \end{aligned}$$

Z rovníc si vytkneme parametre s,t.

$$\begin{aligned} s &= \frac{\vec{w}.\vec{p}}{\vec{u}.\vec{p}} && (22)\\ s &= \frac{\vec{w}.(\vec{n} \times \vec{v})}{\vec{u}.(\vec{n} \times \vec{v})} && (23) \end{aligned}$$

$$\begin{aligned} t &= \frac{\vec{w}.\vec{q}}{\vec{v}.\vec{q}} && (24)\\ t &= \frac{\vec{w}.(\vec{n} \times \vec{u})}{\vec{v}.(\vec{n} \times \vec{u})} && (25) \end{aligned}$$

Dvojnásobný vektorový súčin vektorov $\vec{a}$, $\vec{b}$ a $\vec{c}$ je vektor definový ako:

$$\vec{a} \times (\vec{b} \times \vec{c}) = \vec{b}(\vec{a}.\vec{c}) - \vec{c}(\vec{a}.\vec{b}) \quad (26)$$

Dosadením (26)do nášho výpočtu dostaneme:

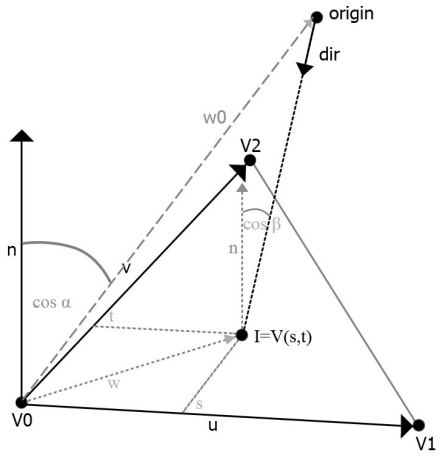
$$\vec{q} = \vec{n} \times \vec{u} \tag{27}$$
$$\vec{q} = (\vec{u} \times \vec{v}) \times \vec{u} \tag{28}$$
$$\vec{q} = (\vec{u}.\vec{u})\vec{v} - (\vec{u}.\vec{v})\vec{u} \tag{29}$$

$$\vec{p} = \vec{n} \times \vec{v} \tag{30}$$
$$\vec{p} = (\vec{u} \times \vec{v}) \times \vec{v} \tag{31}$$
$$\vec{p} = (\vec{u}.\vec{v})\vec{v} - (\vec{v}.\vec{v})\vec{u} \tag{32}$$

Teraz už môžme vypočítať parametre s,t len s použitím skalárneho súčinu.

$$s = \frac{(\vec{u}.\vec{v})(\vec{w}.\vec{v}) - (\vec{v}.\vec{v})(\vec{w}.\vec{u})}{(\vec{u}.\vec{v})^2 - (\vec{u}.\vec{u})(\vec{v}.\vec{v})} \tag{33}$$

$$t = \frac{(\vec{u}.\vec{v})(\vec{w}.\vec{u}) - (\vec{u}.\vec{u})(\vec{w}.\vec{v})}{(\vec{u}.\vec{v})^2 - (\vec{u}.\vec{u})(\vec{v}.\vec{v})} \tag{34}$$



Obr. 9: Priesečník priamky a trojuholníka

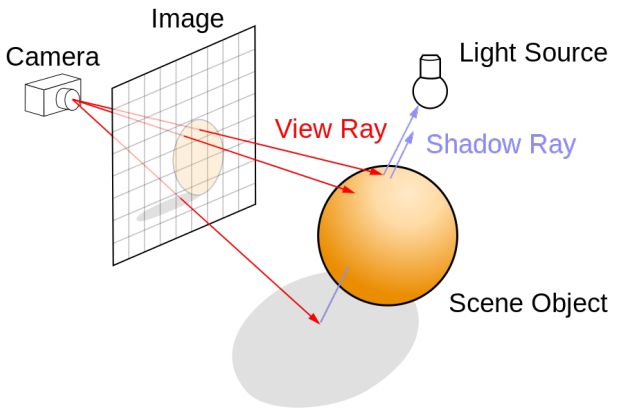
3.5 Ray tracing vs Graphic pipeline

Ray tracing je technika vytvarajúca obraz pomocou sledovania lúča. Jedná sa o veľmi efektívny algoritmus pre rendering scén v počítačovej grafike. Pomocou raytracingu vieme vypočítať farbu, odlesky, tiene, priesvitnosť, lom a iné artefakty, ktoré vykreslovaný obraz priblížia realite. K vykresleniu obrázu sa dá dopracovať dvoma prístupmi:

Dopredná metóda: Tak ako v prírode fotóny vychádzajú zo svetelného zdroja, širia sa priestorom a po mnohých odrazoch od objektov skončia na sietnici oka (v kamere). Z daného pricípu je zrejme, že implemntovaním daného postupu by sme dostali absurdne vysoké renderovacie časy a väčšina počítaných lúčov by bola úplne zbytočná pretože by sa nikdy nedostala do kamery. Napriek tomu má dopredná metóda využite v počítačovej grafike, napríklad pri fotónových mapách.

Spätná metóda: Lúče vychádzajú z kamery v inverznom poradí k svetelnému zdroju => počítamé len tie lúče, ktoré skutočne potrebujeme. Pipeline raytracingu: Vygenerujeme lúč, ktorý začína v kamere a prechádza cez pixel v priemetne, následne prechádzame po lúči v scéne, ktorá je rozdelená do buniek. Zistujeme či v danej bunke sa náchadza objekt. Ak nájdeme bunku, v ktorej sa objekt nachádza tak následne hľadáme priesečník s daným objektom. V priesečníku generujeme dalšie lúče: Tieňový lúč: z priesečníku vyšleme lúč smerom k svetelnému zdroju, a opakujeme celý algoritmus sledovania lúča. Ak lúč narazí na objekt znamená to, že pôvodný priesečník sa nachádza v tieni. Ak lúč dorazí do svetelného zdroja, v povodnom priesečníku možeme vypočítať farbu pomocou shading modelu. *Odrazený lúč* (reflection): Začíname v priesečníku a smer lúču je daný odrazom v danom bode na základe materiálu alebo normály. Následne opakujeme celý algoritmus pre tento lúč.

Lomený lúč (refraction): Ak máme priehľadný materiál, tak z priesečníku vychádza lúč, ktorý sa láme do vnútra objektu na základe indexu lomu.



Obr. 10: Ray tracing

Graphic pipeline (rendering pipeline) je postupnosť krokov používaných na vytvorenie 2D reprezentácie 3D scény. V OpenGL vyzerá rendering pipeline nasledovne.

1. Príprava a špecifikácia dát vrcholov, ktoré sa následne odošlú do pipeline. Vrcholy definujú ohraničenie primitív(trojuholník, úsečka, bod).
2. Spracovanie vrcholov. Každý vrchol z predchádzajúceho kroku je zpracovaný pomocou Vertex shaderu na výstupný vrchol. Mapovanie vstupného vrcholu na výstupný je 1:1, čiže každý vstupný vrchol musí byť namapovaný na výstupný vrchol.

3. Post-procesing vrcholov. Séria operáciu slúžiacich ako príprava pre zostavenie primitív a rasterizáciu
4. Zostavenie primitív. Výstupom procesu je usporiadaná postupnosť primitív(trojuholníky, úsečky, body).
5. Rasterizace. Primitíva sa rasterizované (prevedené do 2D). Výstupom je postupnosť fragmentov (pixelov).
6. Fragment postprocesing. Každý fragment je spracovaný pomocou Fragment shaderu. Výstupom každého Fragment shaderu je zoznam farieb pre každy color buffer, hĺbka a hodnota pre stencil buffer.
7. PerSample Processing: Séria operácii a testov nad výstupom z spracovania fragmentov. Scissor test, stencil test, depth test, blending. Po vykonaní operácii sú fragmenty zapísane do framebuffru.

Pri použití rendering pipeline musí byť povrch objektov rozdelený na primitíva ešte pred rendrovaním. Vplyv osvetlenia je pre každé primitívu počítané samostane, teda tienie, odrazy, lomy musia byť emulované. Metóda je nenáročná na výpočet, vhodná do interaktívnych aplikácii. Ku každému objektu sa pristuje len raz. Zložitosť výpočtu závisí od zložitosti scény a veľkosti vykreslovaného obrázku, ktorý sa musí vtesnať do pamäte.

Oproti graphic pipeline je raytracing všeobecnejšie riešenie, vhodné pre vykreslenie fotorealistických obrázkov. Z princípu výpočtu získame tiene, odrazy a lomy „zadarmo“. Metóda má vysokú náročnosť na výpočet. K objektom sa pristupuje mnohokrát a celá scéna sa musí zmestiť do pamäte. Zložitosť výpočtu je logaritmicky závisla od zložitosti scény.

4 Praktická implementácia

V tejto kapitole popíšeme vývoj vlastnej aplikácie založenej na WebGL. Opíšeme proces od návrh aplikácie až po implementovanie pokročilých shaderov pre vylepšenie kvality vykresľovania.

4.1 Návrh aplikácie

Praktická časť bakalárskej práce sa zaoberá vývojom webovej aplikácie s použitím WebGL a Three.js . Hlavným cieľom vyvýjanej aplikácie je priniesť užívateľovi možnosť interaktívne si prehliadať 3D model domu. Pri vývoji sa prihliadalo nato, že aplikácia bude v budúcnosti umiestnená na firemnú webstránku a bude slúžiť ako jeden z nástrojov určených na prezentáciu projektov.

4.1.1 Koncept

Pri návrhu aplikácie si musíme v prvom rade určiť kto bude s našou aplikáciou pracovať. Prvý kontakt s aplikáciou bude mať dizajnér alebo architekt, ktorý bude pripravovať model do aplikácie. Preto musíme aplikáciu pripraviť tak aby bol import modelov do aplikácie čo najjednoduchší. Keď bude model do aplikácie naimportovaný môžme ho zobraziť užívateľovy. Hlavným cieľom je umožniť užívateľovy si prezrieť model domu v pohľade z prvej osoby a poskytnúť mu predstavu ako bude dom v skutočnosti vyzerať. Keďže je aplikácia určená širokej verejnosti, treba tomu prisbôsobiť aj ovládanie, tak aby bolo jednoduché a intuitívne. Aby vykreslený obraz pôsobil krajším a reálnejším dojmom, rozšírime aplikáciu o niekoľko grafických efektov, ktoré využijú možnosti WebGL a programovateľných shaderov.

4.2 Príprava modelu a import do aplikácie

Model do aplikácie vytvoríme pomocou štandardných 3D modelovacích nástrojou ako je Autodesk 3Ds Max alebo Blender. V tomto sa proces tvorby v ničom nelíši od tvorby modelov pre architektonické vizualizácie. Preto môžu architekti a dizajnéri použiť aj svoju predchadzajúcu prácu. Pravdaže treba mať na pamäti, že model bude zobrazený v reálnom čase, preto je vhodné doržať vhodnú topológiu objektu a počet polygónov. Taktiež treba prisbôsobiť veľkosť textúr s ohľadom nato, že ich užívateľ bude načítať z webu. Preto sa odporúča použiť textúry v komprimovanom formáte a nižšsom rozlíšeni.

Pri exporte samotného modelu z 3D modelovacieho programu máme dve možnosti. Môžme použiť export do formátu .OBJ, ktorý podporuje väčšina programov, alebo môžme použiť export do Three.js formátu. Samotný export do Three.js formátu sa uskutoční pomocou pluginu, ktorý pridáme do 3D modelovacieho nástroju. V súčastnosti existuje plugin pre Blender, 3Dsmax a Mayu.

Načítanie vyexpotovaného modelu do aplikácie možme vykonať, priamim zadaním cesty do zdrojového kódu aplikácie alebo použiť tlačítko v GUI. Aplikácia na základe prípony načítaného súboru sama vybere vhodný loader a vykoná načítanie modelu.

Pri tvorbe a exporte modelu je vhodné dodržať správnu veľkosť objektov, taktiež musíme používať len štandardné materiály, pretože exporter si neporadí s MentalRay materiálom alebo s Cycles. Aplikácia sa dá do budúcna jednoducho rozšíriť o podporu viacerích formátov. V súčasnosti existujú loadre pre gITF, Collada, VRML ...

4.3 Scéna

Aby bolo možné naimportovaný model zobraziť, musíme najskôr pripraviť scénu. Aplikácia obsahuje predpripravenú scénu, v ktorej sa nachádza kamera, obloha v podobe hemisféry, ambientne osvetlenie a svetlo simulujúce slnko. Užívateľ si môže pomocou GUI nastaviť pozíciu slnka na oblohe a pozorovať ako je model domu osvetlený v rôzne časti dňa. Kamera v scéne je defaultne umiestnená na súradnice [0,0,10]. Aplikácia momentálne nepodporuje vloženie vlastnej scény s osvetlením a kamerou. O vytvorenie predpripravenej scény sa stará knižnica *SetupScene.js*. Samotné vytvorenie scény v aplikácii prebieha nasledovne:

```
var container = document.getElementById( 'container' );
 init () ;
animate();

function  init ()  {
 renderer = new THREE.WebGLRenderer();
 container.appendChild( renderer.domElement );

 camera = new THREE.PerspectiveCamera( 80, window.innerWidth / window.innerHeight, 1, 800 );
 controls  = new THREE.PointerLockControls( camera );
 scene = new THREE.Scene();
 setupscene=new THREE.SetupScene(scene);

 initPostProcessing() ;
 initGUI() ;

}

function  animate() {
 requestAnimationFrame( animate );
 render() ;
 controls.update();
}

function  render()  {
 renderer.render( scene, camera, composer.renderTarget2, true );
 composer.render( );
}
```

Výpis 1: Základná kostra apklikácie

V prvom kroku si vytvoríme objekt *renderer*, ktorý zaobstaráva vykresľovanie scény pomocou WebGL. Následne priradíme *renderer* objektu *container*, ktorý reprezentuje *div* element s názvom container v HTML stránke. Toto priradenie určuje do akého elementu na

stránke sa nám WebGL obsah vykreslí. Následne si vytvoríme objekty pre kameru, ovládanie a scénu. Vytvorením objektu *setupscene* naplníme scénu predpripraveným prostredím a načítame objekty zo súboru. Funkcie *initPostrocessing()* a *initGUI()* zaobstarávajú počiatočnú inicializáciu pre shadery a menu. Vo funkcii *animate()* pomocou volania funkcie *requestAnimationFrame()* umožníme prehliadaču ovládať a spravovať vykreslovaciu slučku. Vo funkcii *render()* volaním funkcie *renderer.render()* vykreslíme scénu do buffru určeného pre *composer*. *Composer* je objekt spravujúci shadery, ktoré vykresluje v poradí v ako mu boli pridané.

4.4 Ovládanie

Pre aplikácia je ovládanie jeden z klúčových prvkov. Ovládanie musí byť jednoduché a intuitívne. Keďže bol zámer implementovať ovládanie z pohľadu prvej osoby, bola pre ovládanie zvolená kombinácia myši a klávesnice. Aby sme mohli použiť takýto typ ovládania priamo v prehliadači musíme použiť *Pointer Lock API* [18]. Jedná sa o vstupnú metódu založenú na pohybe myši v priebehu času. API schová kurzor myši a odstráni obmedzenia pri pohybe myšou. V podstate API poskytuje rovnaký typ ovládania aký je známy napríklad z 3D hier. Implementácia *Pointer Lock API* v prehliadačoch sa momentálne nachádza v stave „Candidate Recommendation“ a je podporovaný v prehliadčoch Chrome, Firefox a Opera. V podporovaných prehliadačoch funguje spoľahlivo, jediný bug nastane pri ukončení API a jeho opätovnom zapnutí. V tedy ojedinele nastane mierny posun pohľadu kamery. Pre pohyb kamery po scéne slúži kombinácia kláves *WSAD* prípadne šípky na klávesnici.

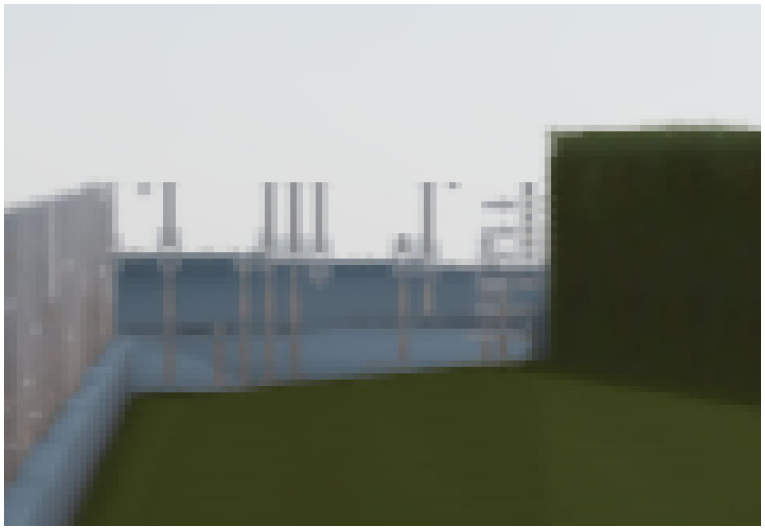
4.5 Implementácia FXAA algoritmu

Z princípu rasterizácie vznikajú v obraze „zubaté“ hrany a dalšie artefakty, ktoré si objasníme. Aby sme ich mohli odstrániť potrebujeme použiť Antialiasing. Jeden z možných postupov Antialiasingu je prepočítať scénu do vyšieho rozlíšenie a následne spriemerovaním hodnôt ju vykresliť v pôvodnom rozlíšeni. Tento prístup má nevýhodu v značne navýšených nárokoch na výpočtový výkon. Preto som sa rozhodol použiť algoritmus FXAA (viz. kapitolu č.3.3.3), ktorý pristupuje k odstráneniu aliasingu až po vykreslení scény. Preto nemusíme scénu renderovať do vyšieho rozlíšenia čo nám ušetrí hardwarové nároky a výpočet vykonáme pomocu fragment shaderu. Ako následne uvidíme ani FXAA nieje dokonalý a nedokáže odstrániť všetky prvky aliasingu.

Najcharakteristickejší prvok aliasingu sú „zubaté“ hrany, v angličtine označované ako jagged edges. Prejavujú sa najmä na diagonálych hranách objektov. Pri pohybe kamery okolo pozorovaného objektu sa mení aj uhol pod akým je objekt pozorovaný a teda sa mení aj uhol zobrazených hrán. To spôsobuje efekt „pohybu zubov“, ktorý pôsobí rušivo. FXAA dokáže úspešne detekovať a hrany objektu a tento prejav aliasingu potlačiť.

Obr. 11: Aplikácia FXAA filtru na obraz.

V porovnaní s metódami antialiasingu ako je napríklad Oversampling alebo Multisampling nedokáže FXXA zrekonštruovať deformácie vzniknuté pri rasterizácie objektu. K tomúto javu dochádza najmä keď sa vykresľujú tenké objekty ako sú laná alebo pletivo. Dôvodom vzniku deformácii je, že vykreslený objekt je tenší ako jeden pixel a na obrazovku sa nevykreslí. To vedie k vzniku nesúvislého celku.



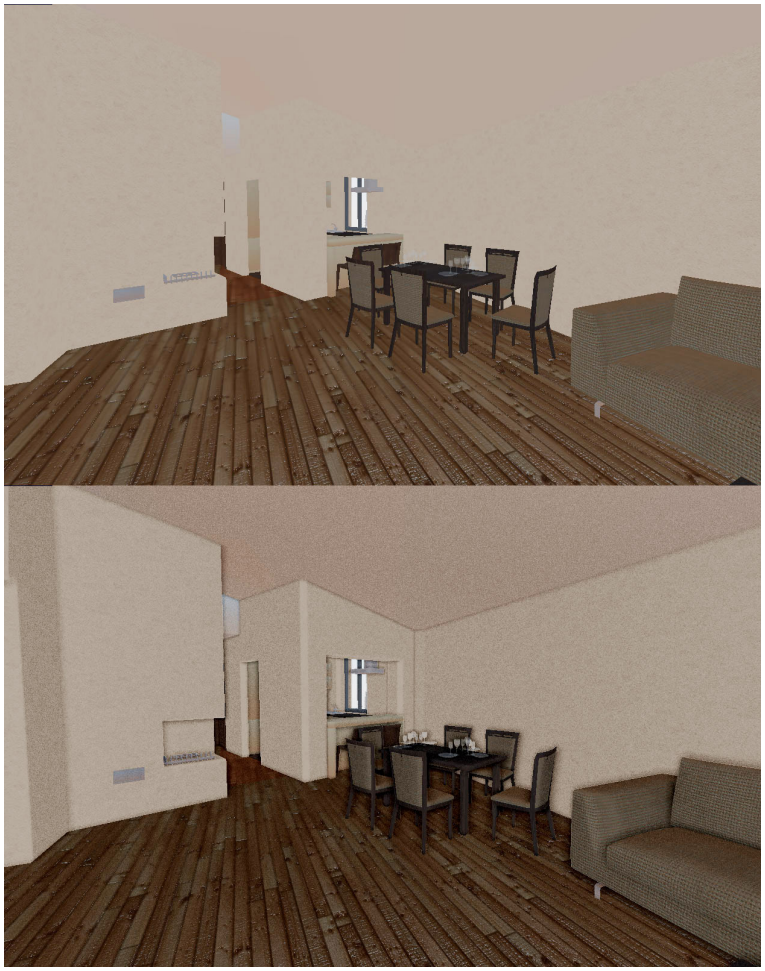
Obr. 12: Deformácie obrazu pri vykreslení plotu.

Hlavným nedostatkom FXAA je že dochádza k neželanému vyhladzovaniu prvkov v obraze, ako je napríklad text alebo textúry. Tím strácame určite množstvo detailov v textúrach a celý obraz pôsobí mierne rozmazane.

4.6 Implementácia SSAO algoritmu

Vo vyvýjanej aplikácii zobrazujeme architektonické modely, ktoré obsahujú interiérové a exteriérové scény. Pri zobrazení interiérových scéne narazíme na problém, že vyzerajú plocho, a užívateľ nevie presne rozlíšiť priestor a tvary objektov. To vedie k jeho dezorientácii v priestore a následne k neuspokojivému „zážitku“ z prezentácie. Tak ako v reálnom svete tak aj v aplikácii je zdrojom svetla slnko a čiastočne aj obloha. Priame svetlo vstupuje do interieru cez okná a vačia časť interiéru je v „tieni“ a je osvetlená nepriamo. Výpočet nepriamého osvetlenia je náročný na výkon a implementáciu. Preto sa musíme efektu nepriameho osvetlenia priblížiť pomocou inej metódy. Jednou z možností je vypočítať nepriame osvetlenie scény v externom 3D editore (Blender, 3Dsmax), pomocou pokročilých renderovacích nástrojov a výsledok „zapiecť“ do textúr objektov. Tento prístup poskytuje vizuálne vysokú kvalitu. Nevýhoda spočíva v statickosti vygenerovaných textur, čiže pri každej zmene geometrie a polohy objektov je nutné ju prepočítať a znova „zapiecť“ do textúr.

Po uvážení som sa rozhodol pre použitie Ambient Occlusion algoritmu. Ten sám o sebe v dnešnej dobe nieje vhodný pre implementovanie do interaktívných aplikácii pre svojú vysokú výpočtovú náročnosť. Aby sme tento problém vyriešili musíme preniesť výpočet zatienenia z procesoru na grafickú kartu. Preto som sa rozhodol implementovať SSAO algoritmus (3.1.1). Jeho kombináciou s ambientným osvetlením docielime prekreslenie časti interiéru, ktorý sú v „tieni“.



Obr. 13: (*hore*)Scéna bez použitia SSAO. (*dole*) Scéna po použití SSAO.

Pôvodný SSAO algoritmus, ktorý v roku 2007 vyvinul Vladimir Kajalin z CryTeku, používal pre testovanie vzorkov celú sféru okolo daného bodu. To malo za následok stmavnutie obrazu, pretože aj pri rovných plochách je minimálne polovička sféry zatienená. Algoritmus bol následne vylepšený a na zistenie zatienienia používa hemisféru orientovanú poľa normály. To nám pomôže odstrániť problém šedých plôch.

```
fragmentShader: [
    .....
    for(int i=0; i<samples;++i)
    {
        //urcime luc z kernelu a nahodne ho usmernime. velkost kernelu v danom bode urcujeme
            pomocu radD
        ray = radD*reflect(sample_sphere[i],fres);

        //ak luc smeruje mimo polgule kernelu, otocime ho.
        se = ep + sign(dot(ray,norm)) *ray;
```

```
            //urcenie hlbky bodu v kernely
            depth_temp = readDepth( se.xy );

            //rekonstrukcia normaly bodu v kernely
            occNorm = normal_from_depth(depth,se.xy);

            //urcenie rozdielu  normal (cosinus uhlu aky zvieraju))
            normDiff = (1.0−dot(occNorm,norm));

            //rozdiel  hlbok bodu v kernely a aktualneho fragmentu
            depthDifference =(currentPixelDepth−depth_temp);

            if (depthDifference∗normDiff>=rad∗0.2){bl+=1.0;}
        };
         .....
]
```

Výpis 2: Implementácia SSAO s orientovanou hemisférou

Ďalší nedostatok algoritmu, je vo vytváraní šumu v obraze. Intenzita šumu je závisla od počtu vzorkov v hemisfére, čiže čím máme viac vzorkov v hemisfére tak šum bude menej postrehnuteľný. Aby sme šum potlačili potrebujeme relatívne vysoké množstvo vzorkov, čo sa nám negatívne prejaví na výkone aplikácie. Iná možnosť odstránenia šumu je, že výstup SSAO algoritmu vyrendrujeme do textúry, ktorú následne rozmažeme a až potom zmiešame s pôvodným obrazom. Takto sa však pripravíme o určité množstvo vykreslených detailov.

4.7 Implementácia Motion Blur algoritmu

Aplikácia obsahuje post-process implementáciu Motion blur efektu (viz. kapitolu č.3.2.2). Aby sme mohli vypočítať rozmazanie obrazu musíme najskôr určiť pre daný pixel súradnice objektu vo world-space. Hodnota uchovaná v depth buffry , je podiel z súradnice trojuholníka a w súradnice trojuholníka potom ako sú vrcholy trojuholníka transformované promocou world-view-projection matice. Vyredrovanie depth buffra do textúry nám umožní zístať world-space súradnice objektu pre daný pixel. Súradnice získame vynásobemím aktuálnej pozície pixelu inverznou projekčnou maticou a vynásobením w. Proces môžme definovať ako:

$$H = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1) \tag{35}$$

$$H \times M^{-1} = \frac{wX}{wW}, \frac{wY}{wW}, \frac{wZ}{wW}, wW = D \tag{36}$$

$$W = \frac{D}{D.w} \tag{37}$$

kde x, y sú súradnice pixelu , z je hĺbka daného pixelu. M je projekčná matica a W sú world-space súradnice pre daný pixel. Keď poznáme world-space súradnice pre daný pixel, môžme ho transformovať pomocou projekčnej matice z predchádzajúceho snímku a ich rozdielom určiť smer a veľkosť rozmazania. Následne môžme obraz rozmazať v smere vektoru. Implementácia fragment shaderu vyzera nasledovne:

```
fragmentShader: [
        .....
                        float zOverW = readDepth( vUv );
    vec4 H = vec4( vUv.x * 2. - 1., vUv.y * 2. - 1., zOverW, 1. );
    vec4 D = H * viewProjectionInverseMatrix;
    vec4 worldPos = D / D.w;

    vec4 currentPos = H;
    vec4 previousPos = worldPos * previousViewProjectionMatrix;
    previousPos /= previousPos.w;
    vec2 velocity = velocityFactor * ( currentPos.xy - previousPos.xy ) * 0.5;

    for( int i = 0; i < samples; i++ ) {
        offset = velocity * ( float( i ) / ( float( samples ) - 1. ) - .5 );
        vec4 c = texture2D( tColor, vUv + offset );
        finalColor += c;
    }

    finalColor /= float( samples );
    gl_FragColor = vec4( finalColor.rgb, 1. );
    ]
```

Výpis 3: Implementácia Motion blur efektu.

Pri implementácii je dôležite správne nastaviť hodnotu $velocity$ defaultne nastavená na 1, ktorá určuje celkové rozmazanie obrazu. Ak je hodnota nízka efekt je zanedbatelný. Naopak ak je hodnota príliš vysoká tak sa obraz rozmaže až príliš a celkový dojem je nereálny. V implementácii je počet vzorkov nastavený na 20 čo predstavovalo dobrý kompromis medzi kvalitou obrazu a vplyvom na výkon. Taktiež nesmieme zabudnúť že Motion blur efekt zarad'ujeme až na koniec post-proces reťazca efektov. Implementovaný efekt zlepšuje plynulosť obrazu pri pohybe a dodáva zobrazenej scéne na reálnosti. Jediná nevýhoda efektu je, že nedokáže aplikovať rozmazanie na pohybujúci sa objekt ak je kamera statická. Tento nedostatok však v našom prípade môžme považovať za nepodstatný, pretože zobrazujeme hlavne scény kde sú objekty statické.

4.8 Implementácia Color picking algoritmu

Color picking algoritmus používame v aplikácii na detekciu kolízii kamery s objektami. To nám umožní aby užívateľ nemohol prechádzať napríklad cez steny domu. V prvom kroku implementácie Color picking algoritmu musíme každému objektu v scéne priradiť unikátnu farbu. Pôvodný material, ktorý mal objekt si dočasne uchovávame aby sme ho mohli na konci algoritmu vrátiť späť a scénu vyrenderovať. Následne si vytvoríme buffer do ktorého vyrendrujeme scénu s unikátnimi farbami pre objekty. Keď je scéna vykreslená do buffru môžme pomocou funkcie *gl.readPixels()* určiť farbu pixelu na daných súradniciach. Zo získanej farby spätne určíme *ID* objektu, a každému objektu vrátime jeho pôvodný materiál. Implementácia algoritmu vyzerá následovne:

```
for ( meshsId in meshs)
    {
```

```
    var temp_id = parseInt(meshsId)+1;
    color.setHex(temp_id);

     tempmat[meshsId] = meshs[meshsId].material;
     meshs[meshsId].material = new THREE.MeshBasicMaterial({color:color});
}

var gl = self.renderer.getContext();
var rttFramebuffer;

rttFramebuffer = this.pickingTexture._webglFramebuffer;
gl.bindFramebuffer(gl.FRAMEBUFFER, rttFramebuffer);
gl.viewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
var pixelBuffer = new Uint8Array(4);
renderer.render( scene, camera, rttFramebuffer );

gl.readPixels( mousex, mousey, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, pixelBuffer );
gl.bindFramebuffer(gl.FRAMEBUFFER, null);
var id_temp = ( pixelBuffer[0] << 16 ) | ( pixelBuffer[1] << 8 ) | ( pixelBuffer[2] );
if (id_temp===0){var id = null; }
  else{var id = id_temp-1; }

for ( meshsId in meshs) {
meshs[meshsId].material = tempmat[meshsId];
}

return id;
```

Výpis 4: Implementácia Color picking algoritmu.

Aby algoritmus fungoval správne je potrebné renderovať scénu bez tieňov a post-process efektov. Algoritmus má tú výhodu, že pristupuje k riešeniu problému detekcii kolízii v 2D. To znamená, že je plne nezávislý od zložitosti objektov v scéne. V našej aplikácii pre určenie kolízi objektu s kamerou potrebujeme vedieť ako daleko je od nás objekt vzdialený. Údaj o vzdialenosti objektu od kamery môžme získať z depth buffru. Bohužial WebGL neumožnuje priamy prístup do depth buffru pomocou *GLDEPTHCOMPONENT* , preto si pomôžeme jednoduchým trikom. Predtím než scénu vykreslím si nastavíme dohľadnosť kamery na požadovú vzdialenosť kde chceme detekovať kolíziu. To znamená, že všetky objekty, ktoré su dalej než je hraničná vzdialenosť sa nevykreslia. Vykreslia sa len tie objekty, ktoré sú bližšie než je hraničná vzdialenosť. Ak z daného pixelu odčítame farbu a určíme ID objektu, vieme s istotou, že objekt je za hranicou a ku kolízii došlo. Algoritmus je dostatočne rýchly aj pre náročne scény, jeho jediný problém je, že v aktuálnej implementácii dochádza k problému synchronizácie CPU a GPU pri volaní funkcie *gl.readPixels()*, čo vedie k miernemu trhaniu obrazu.

4.9 Implementácia výpočtu priesečníku priamky s trojuholníkom

V aplikácii používame tento algoritmus rovnako ako Color picking algoritmus (viz. kapitola č.4.8) na detekciu kolízii s kamerou. Oproti nemu má tento algoritmus výhodu vtom, že vieme presne určiť kde v priestore ku kolízii došlo. V prvom bode algoritmu

si musíme zostaviť vrcholy trojuholníku a následne ich transformovať z object-space do world-space. Z takto pripravených vrcholov možme vytvoriť vektory u,v a znich určiť normálu n (viz. rovnica č.2).

```
fa=meshs[meshsId].geometry.faces[i].a;
fb=meshs[meshsId].geometry.faces[i].b;
fc=meshs[meshsId].geometry.faces[i].c;

v0 = new THREE.Vector3(meshs[meshsId].geometry.vertices[fa].x, meshs[meshsId].
    geometry.vertices[fa].y, meshs[meshsId].geometry.vertices[fa].z);
v1 = new THREE.Vector3(meshs[meshsId].geometry.vertices[fb].x, meshs[meshsId].
    geometry.vertices[fb].y, meshs[meshsId].geometry.vertices[fb].z);
v2 = new THREE.Vector3(meshs[meshsId].geometry.vertices[fc].x, meshs[meshsId].
    geometry.vertices[fc].y, meshs[meshsId].geometry.vertices[fc].z);

v0.applyMatrix4(meshs[meshsId].matrixWorld);
v1.applyMatrix4(meshs[meshsId].matrixWorld);
v2.applyMatrix4(meshs[meshsId].matrixWorld);

u.subVectors(v1,v0);
v.subVectors(v2,v0);
n.crossVectors(u,v);
```

Výpis 5: Implementácia výpočtu priesečníku priamky s trojuholníkom, prvá časť.

Vektor dir definujeme ako smerový vektor kamery. Vektor $w0$ určíme z bodu kde sa nachádza kamera ($origin$) a z vrcholu $V0$. Následne podľa (viz. rovnice č.4 a č.5) vypočítame skalár a a b, ktoré otestujem či spĺňaju podmienky podľa (viz. rovnica č.6).

```
dir=controls.getDirection(new THREE.Vector3(0, 0, 0)).clone();
dir.applyMatrix4(this.rMatrix);


w0 = new THREE.Vector3(origin.x−v0.x,origin.y−v0.y,origin.z−v0.z);
a=(n.dot(w0))∗(−1);
b=n.dot(dir);

  if (Math.abs(b) < eps)
  {
      if (a===0)
      {
          continue;
      }
      else
      {
          continue;
      }
  };
```

Výpis 6: Implementácia výpočtu priesečníku priamky s trojuholníkom, druhá časť.

Ak a,b spĺňajú potrebné podmienky určíme si podľa skalár r (viz. rovnica č.8). Ak je skalár r väčší ako 0 môžme si podľa (viz. rovnice č.9 a č.15) definovať vektor I a vektor w. V poslednom kroku si pripravíme pomocné premenné pre výpočet parametrov s,t

podľa (viz. rovnice č.33 a č.34). Ak vypočítané parametre spĺňajú podmienky, z istotou vieme, že lúč pretťal daný objekt.

```
I=new THREE.Vector3(origin.x + (r*dir.x),origin.y + (r*dir.y), origin.z + (r*dir.z));
w=new THREE.Vector3(I.x-v0.x,I.y-v0.y,I.z-v0.z);

uu = u.dot(u);
uv = u.dot(v);
vv = v.dot(v);


wu= w.dot(u);
wv= w.dot(v);
D=(uv*uv) - (uu*vv);

s = ((uv * wv) - (vv * wu)) / D;

if (s<0.0 || s>1.0)
{
    continue;
}
else{

t = ((uv * wu) - (uu * wv)) / D;

if (t<0.0 || (s+t)>1.0)
{
    continue;
}
    else{
      distance_temp=Math.sqrt(((origin.x-I.x)*(origin.x-I.x))+((origin.y-I.y)*(origin.y
           -I.y))+((origin.z-I.z)*(origin.z-I.z)));

        if (distance_temp<distance)
        {
          distance=distance_temp;
          intersection_position=I;
        }


      continue;

        }
  }
```

Výpis 7: Implementácia výpočtu priesečníku priamky s trojuholníkom, tretia časť.

V implementácii musíme ošetriť situáciu, keď lúč pretne viacero objektov. V takom prípade by dochádzalo k nesprávnej detekcii kolízii pretože v pamäti by ostali uchované informácie o poslednom prieniku lúču s objektom a nie o najbližšiom prieniku vzhľadom ku kamere. Preto pre každý vypočítaný prienik musíme určiť jeho vzdialenosť od kamery, a údaje o danom prieniku uchovávame len vtedy ak je jeho vzdielenosť voči kamere menšia než bola vzdialenosť predchádzajúceho prieniku. Takto docielime, že v pa-

mäti budeme mať uchované informácie o najbližšom prieniku, ktorý je v našej aplikácii pre detekciu kolízii podstatný. Pri imlementácii musíme ošetriť aj situáciu kedy užívateľ sa pomocou kláves pohybuje do strán, prípadne dozadu. Vtedy by detekcia kolízii pomocou tohto algoritmu nefungovala pretože smerový vektor kamery nieje zhodný zo smerom pohybu. Preto si pre každý smer pohybu *vľava,vpravo,vzad* pripravíme rotačnú maticu, ktorou prenásobime vektor dir. To nám zabezpečí detekciu kolízii s objektami aj keď ich užívateľ priamo nevidí. Algoritmus dokáže presne určiť pozíciiu kolízie, ale je priamo závislý od zložitosti scény a objektov. Aby bolo možné algoritmus použiť aj v zložitejších scénach je potrebné implementovať urýchlovacie algoritmy. Druhá možnosť je použiť pre výpočet kolízii zjednodušenú geometriu scény, pretože vo väčšine prípadov postačuje ak zložitý objekt „zaobalíme“ do jednoduchej geometrie, nad ktorou budeme počítať kolízie.

4.10 Testovanie aplikácie

Počas vývoja bola aplikácia testovaná na niekoľkých zariadeniach aby sa overila jej funkčnosť. Výkon aplikácie je priamo závislý od výkonu grafickej karty a procesora. Pre úspešné spustenie aplikácie musí užívateľ vlastniť zariadenie schopné spracovať programovateľne shadery.

Názov	CPU	GPU
Desktop Windows 8.1	Intel Core2Quad Q9550	Nvidia GTX 650Ti
Lenovo IdeaPad S8	Intel Atom Z3745	Intel HD Graphics
Lenovo IdeaPad B5400	Intel Pentium 3550 Haswell	NVIDIA GeForce GT 820M

Tabuľka 1: Testované zariadenia

Názov	Priemerný počet snímkov
Desktop Windows 8.1	60
Lenovo IdeaPad S8	22
Lenovo IdeaPad B5400	52

Tabuľka 2: Namerané hodnoty

Výsledky testovania potvrdili závislosť aplikácie od výkonu grafickej karty. Aplikácia fungovala plynule na desktope a notebooku. Na tablete bol výkon aplikácie uspokojivý a na prezentačné účely dostačujúci. Testovacie scény obsahovaly v priemere pól milióna polygonov. Pri webovej aplikácii je dôležité zabezbečiť podporu čo najširšieho spektra webových prehliadačov.

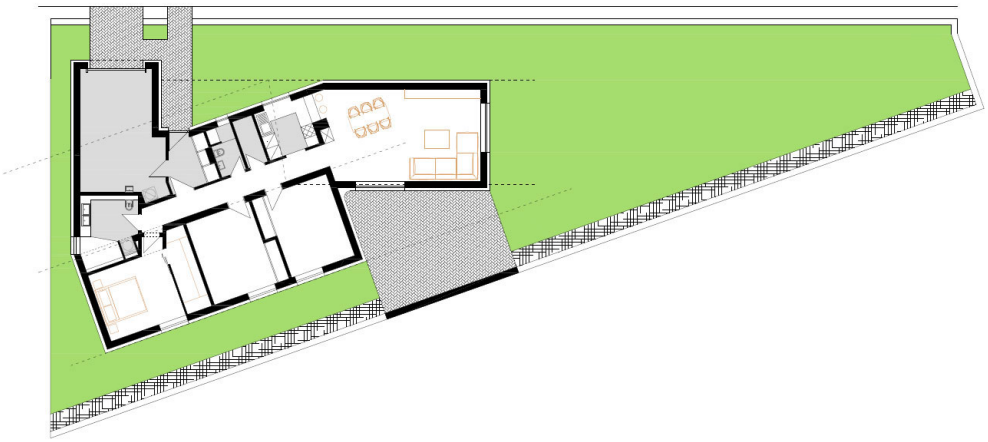
	Chrome 42	FireFox 37	Internet Explorer 11	Opera 29	Android 4.4
WebGL	Podporuje	Podporuje	Podporuje	Podporuje	Podporuje
PointerLockAPI	Podporuje	Podporuje	Nepodporuje	Podporuje	Nepodporuje

Tabuľka 3: Podpora aplikácie webovými prehliadačmi

WebGL je štandard a preto je podporovaný vo všetkých testovaných prehliadačoch. Funkčnosť zobrazenia aplikácie bola bezproblémová. Funkcia PointerLockAPI v prehliadači Internet Explorer nieje implementovaná, takže v tomto prehliadači nieje možné aplikáciu ovládať. Microsoft v poslednom roku v tichosti ukončil vývoj Internet Exploreru a všetky sily sústredí do vývoja nového prehliadču Edge ktorý má IE nahradiť. Edge má byť predstavený spolu s Windows 10. Bohužial PointerLockAPI taktiež nieje implementovaný v prehliadačoch pre platformu Android. Avšak pre použitie aplikácie na mobilných zariadeniach by bolo potrebné navrhnúť a implemntovať spôsob ovládania, ktorý by viac vyhovoval dotykovým zariadeniam. Aplikáciu je potrebné ešte pred nasadením otestovať na širšom spektre zariadení a operačných systémov, najmä na zariadeniach s iOS.

4.11 Ukážka aplikácie

V tejto kapitole sa nachádza ukážka z vytvorenej aplikácie. Na obrázkoch je vidieť porovnanie s pôdorysom domu a s výstupom z aplikácie. Zobrazenie v 3D oproti pôdorysu prináša užívateľovy možnosť pozrieť sa na dom z novej perspektívy a lepšie si predstaviť celkový vzhľad domu.



Obr. 14: Pôdorys domu nachádzajúceho sa v katastri obce Zlatovce.

Obr. 15: Zobrazenie modelu domu v aplikácii.

5 Záver

V tejto práci bol popísaný postup práce pri vytváraní interaktívnej webovej prezentácie s použitím WebGL a rozšírením pomocou knižnice Three.js. Vyskúšaný bol postup práce od tvorby modelu, jeho úpravu pre real-time zobrazenie až po import do aplikácie. Vytvorená aplikácia obsahuje podporu pre načítanie modelov z dvoch podporovaných formátov. Taktiež sa podarilo úspešne implementovať pokročilé techniky pre postprocess úpravu obrazu, ktoré vylepšili vizuálnu kvalitu výsledného obrazu. Aplikácia bola rozšírene o niekoľko techník identifikácie objektov, ktoré boli použité na detekci kolízii.

Táto práca mi pomohla pochopiť základné princípy a postupy pri tvorbe interaktívnej grafickej aplikácie, taktiež rozšírila moje znalosti JavaSciptu a WebGL. Pri vývoji aplikácie bolo nutné reagovať na vývoj nových verzii internetových prehliadačov, ktoré postupne implementovali možnosti WebGL. Tieto zmeny niekedy viedli k nefunčnosti aplikácie v určitých verziách prehliadačov, preto bolo nutné operatívne hľadať príčinu a odstrániť ju.

V aktuálnom stave je aplikácie schopná plniť svoj účel, ale pre nasadenie vo firemnej sfére by bolo nutné uskutočniť rozsiahlejšie testovenie aby sa predišlo prípadným chybám na rôznych platformách a prehliadačoch. V budúcnosti by som chcel na aplikácii naďalej pracovať a rozšíriť ju o urýchľovacie algoritmy, ktoré pomôžu zvýšiť výkon pri detekci kolízii v náročnejších scénach. Dalej rozšíriť a vylepšiť postproces efekty a implementovať metódu pre vykreslenie kvalitnejších tieňov. Podporu formátov, ktoré možu byť naimportované do aplikácie by som chcel rozširiť najmä o binárne formáty, ktoré urýchlia načítanie scény a obmedzia prenos dát medzi užívateľom a serverom.

WebGL definovalo štandard pre 3D grafiku na webe, a preto sa sním budeme v budúcnosti stretávať čoraz častejšie.

6 Literatúra

[1] *Adobe Flash Player Specification [online],* [cit.2015-05-01].
Dosputpné z: http://www.adobe.com/products/flashplayer.html

[2] *Microsoft Silverlight [online],* [cit.2015-05-01].
Dosputpné z: http://www.microsoft.com/silverlight/

[3] *WebGL Specification [online],* [cit.2015-04-22].
Dostupné z: https://www.khronos.org/registry/webgl/specs/1.0/

[4] *ANGLE: Almost Native Graphics Layer Engine [online],*[cit.2015-01-05].
Dostupné z: https://code.google.com/p/angleproject/

[5] *GLGE [online],* [cit.2015-04-27].
Dostupné z: http://www.glge.org/

[6] *CubicVR 3D Engine [online],* [cit.2015-04-27].
Dostupné z: http://www.cubicvr.org/

[7] *Unreal Engine [online,]* [cit.2015-04-27].
Dostupné z: https://www.unrealengine.com

[8] *Three.js [online],* [cit.2015-04-27].
Dostupné z: http://threejs.org/

[9] Dirksen, Jos.
Three.js Essentials. Birmingham, Packt Publishing, 2014, ISBN:978-1-78398-086-4

[10] Parisi, Tony.
WebGL:Up and Running. California, O'Reilly Media, 2012, ISBN:978-1-44932-357-8

[11] Weisstein, Eric W. *Barycentric Coordinates [online],* [cit.2015-04-17].
Dostupné z: http://mathworld.wolfram.com/BarycentricCoordinates.html

[12] Chapman, John. *SSAO [online],* [cit.2015-04-18].
Dostupné z: http://john-chapman-graphics.blogspot.sk

[13] Bavoli, Louis. *Nvidia Screen Space Ambient Occlusion [online,]* [cit.2015-04-18].
Dostupné z: http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/ScreenSpaceAO/doc/ScreenSpaceAO.pdf

[14] Young, Peter. *CSAA (Coverage Sampling Antialiasing) [online],* [cit.2015-04-20].
Dostupné z: http://www.nvidia.com/object/coverage-sampled-aa.html

[15] Lottes, Timothy. *FXAA [online],*[cit.2015.04-20].
Dostupné z: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

[16] Bavoli,Louis. *Horizon-Based Ambient Occlusion [online],*[cit.2015-04-16].
Dostupné z: http://developer.download.nvidia.com/presentations/2008/SIGGRAPH/HBAO_SIG08b.pdf

[17] Gilberto, Rosado. *Motion Blur as a Post-Processing Effect [online]*,[cit.2015-04-20]. Dostupné z: `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch27.html`

[18] Mozzila developer network. *Pointer Lock API [online]*, [cit.2015-04-27]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Pointer_Lock_API`