UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

REAL-TIME PATH-TRACING PRE PROCEDURÁLNE MODELY VO WEBGL

Bakalárska práca

Bratislava, 2016 **Róbert Sarvaš**

UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

REAL-TIME PATH-TRACING PRE PROCEDURÁLNE MODELY VO WEBGL

Bakalárska práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	2511 Aplikovaná informatika
Školiace pracovisko:	FMFI.KAGDM - Katedra algebry, geometrie a didaktiky matematiky
Vedúca bakalárskej práce:	RNDr. Ivana Ilčíková

Bratislava, 2016 **Róbert Sarvaš**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Róbert Sarvaš
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: aplikovaná informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Real-time path-tracing pre procedurálne modely vo WebGL
Real-time Path-tracing for Procedural Models in WebGL

Cieľ: Procedurálne generované modely sa vyznačujú vysokou úrovňou detailov a teda aj veľkým objemom geometrie. Pre takéto dáta má študent v prostredí WebGL implementovať real-time fotorealistické zobrazovanie, založené na path-tracingu.

Vedúci: RNDr. Ivana Ilčíková
Katedra: FMFI.KAGDM - Katedra algebry, geometrie a didaktiky matematiky
Vedúci katedry: prof. RNDr. Pavol Zlatoš, PhD.

Dátum zadania: 26.10.2014

Dátum schválenia: 06.11.2014

doc. RNDr. Mária Markošová, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Čestné vyhlásenie

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne s použitím uvedených zdrojov.

V Bratislave, dňa 29.5.2016 .....

Róbert Sarvaš

Poďakovanie

Ďakujem svojej vedúcej bakalárskej práce, že sa podujala na tak ťažkú úlohu akou bolo viesť mi bakalársku prácu, ďalej ďakujem Martinovi Ilčíkovi, ktorý si mnohokrát našiel čas aby mi pomohol a poradil pri implementácii aplikácie. Ďakujem aj mojej rodine a mojej priateľke, ktorí ma podporovali a vytvárali mi ideálne prostredie pre prácu.

Abstrakt

SARVAŠ, Róbert: Real-time path-tracing pre procedurálne modely vo WebGL [Bakalárska práca] - Univerzita Komenského v Bratislave. Fakulta Matematiky, Fyziky a Informatiky. Katedra Aplikovanej informatiky. Vedúca bakalárskej práce: RNDr. Ivana Ilčíková. Bratislava 2016.

Cieľom bakalárskej práce je preštudovať algoritmus Path-Tracing pre vizualizáciu 3D modelov. Ďalším cieľom je vytvorenie samostatného funkčného modulu vo webovom prehliadači, ktorý bude používať Path-Tracing algoritmus a bude vizualizovať procedurálne modely v reálnom čase. Procedurálne modely sú modely, ktoré sa vyznačujú vysokým objemom geometrie. Tento modul bude implementovaný pomocou WEBGL a GLSL.

Kľúčové slová: WebGL, vizualizácia 3D modelov, Path-Tracing, GLSL, web.

Abstract

SARVAŠ, Róbert: Real-time path-tracing for procedural models in WebGL [Bachelor thesis] - Comenius University in Bratislava. Faculty of Mathematics, Physics and Informatics. Department of Applied Informatics. Bachelor thesis supervisor: RNDr. Ivana Ilčíková. Bratislava 2016.

The aim of this bachelor thesis is to study Path-Tracing algorithm that visualize 3D models. Another part of aim is to implement self-standing plug-in in web browser which uses Path-Tracing algorithm for real-time visualization of procedural models. Procedural Models are significant with huge amount of data which they are containing. This plug-in will be implemented in WebGL and GLSL.

Key words: WebGL, visualization 3D models, Path-Tracing, GLSL, web.

Predhovor

V dnešnej dobe je svet IT technológii asi jedným z najprogresívnejších odborov. Napreduje takou rýchlosťou, že samotní programátori musia vynakladať množstvo úsilia na sledovanie najnovších trendov. Pre mňa predstavuje jadro tohto sveta práve internet. Internet je jednoznačne najväčšou knižnicou informácií ale takisto aj najsilnejším komunikačným tokom dnešnej doby.

Každým dňom na internete pribúda čoraz viac multimediálneho obsahu, ktorý má poskytnúť používateľovi rozsiahle informácie o danej téme. A nie je to vôbec náhodou, že multimediálny obsah je tak preferovaný. Predstavte si, koľko riadkov textu je potrebných na opísanie jednej scény a koľko času to zaberie každému, kto si to prečíta a porovnajte to s alternatívou že na webe je fotografia tej scény. Ak sa pozrieme na fotografiu, okamžite si uvedomíme všetky nadväznosti a nemusíme zdĺhavo čítať, že nejaká vec je napravo od inej a vytvárať si tak obraz postupne. Je dokázané, že človek dokáže najviac informácii a najrýchlejšie získať vizualizáciou a práve preto je multimediálny obsah internetu tak dynamicky rozrastajúci sa.

Multimediálny obsah je kombináciou obsahu pozostávajúceho z textu, obrázkov, zvuku, animácií, video sekvencií a interaktívnych prvkov. Dnes je už takýto obsah bežnou súčasťou web stránok a do popredia sa dostáva idea vytvárania 3D webu. Web3D bola spočiatku myšlienka zobrazovať celý obsah internetu v 3D. Dnes tento termín opisuje celý interaktívny obsah, ktorý je vsadený do HTML web stránok. Myšlienka webu3D otvorila nové možnosti v rôznych oblastiach od on-line počítačových hier, cez simulácie rôznych vedeckých javov, vizualizácie 3D máp, napríklad [GMAPS] až po detailné vizualizácie ľudského tela a procesov v ňom pre medicínske účely, napríklad [ZGOTE]. V mojej práci sa venujem tvoreniu práve takéhoto obsahu, konkrétne možnostiam vizualizácie 3D scén vo webovom prehliadači.

Túto tému bakalárskej práce som si vybral, pretože si myslím, že v blízkej budúcnosti vzrastie obsah 3D webu a stane sa pre ľudí príťažlivým a to hlavne vďaka nasledujúcim dôvodom. Pohyb v 3D priestore bude pre ľudí oveľa viac intuitívny, než dnes na klasických 2D HTML stránkach. Vďaka takémuto pohybu dokážu nielen efektívne vyhľadávať potrebné informácie ale 3D web im umožní pomocou 3D vizualizácií aj oveľa rýchlejšie pochopiť daný problém.

Obsah

1. Úvod do problematiky 11
1.1 Cieľ práce 13
1.2. Úvod do WebGL 13
1.3. Úvod do vykresľovania vo WebGL 15
1.3.1. Vertex Buffer Objects (VBOs) 16
1.3.2. Vertex Shader 16
1.3.3. Fragment Shader 16
1.3.4 Framebuffer 17
1.3.5. Attributes, Uniforms a Varyings 17
2. Súvisiace práce 18
2.1 WebGL Path Tracing 18
2.2 More Spheres 18
2.3 Jednoduchý Path-Tracer 19
2.4 Brigade Renderer 19
3. Teória 20
3.1 Obojsmerná odrazová distribučná funkcia 20
3.2 Zobrazovacia rovnica 20
3.3 Monte Carlo metódy 21
3.4 Sledovanie ciest *(ang.Path - Tracing)* 22
3.4.1 Obojsmerný Path-Tracing 23
4. Metodika 25
4.1 Technické riešenie algoritmu sledovania ciest 25
4.1.1 Vzorkovanie 26
4.2 Algoritmus Path-Tracing vo WebGL 26
4.3 Analýza kódu 27
4.3.1 Hlavná časť 27
4.3.2 Triedy Ui , Renderer a Pathtracer 27
4.3.3 Objekty 29
4.3.4 Utility funkcie 29
4.3.5 Kód v GLSL 29
4.3.5.1 Funkcie MakeCalculateColor, makeShadow, makeMain a makeTracerFragmentSource 30

5. Riešenie a Implementácia 31
5.1 Zefektívnenie algoritmu 32
5.2 Objekty 33
5.2.1 Mapovanie objektov do textúr: 34
5.2.2 Lúč- objekt prieniky 35
5.2.3 Objekty z .obj súborov 36
6. Záver 39
6.1 Dosiahnuté ciele 39
6.2 Zhodnotenie a Návrh ďalšej práce 39
6.3 Výsledné obrázky 39
Bibliografia: 43
Prílohy: 44

1. Úvod do problematiky

Vykresľovanie scén pomocou pokročilých softvérov sa dnes využíva v mnohých odvetviach, akými sú najmä herný a filmový priemysel, ale tiež aj pri tvorení návrhov a dizajnu rôznych produktov, tvorby reklám a podobne. Pri takomto vykresľovaní sa kladie veľký dôraz hlavne na realistické zobrazovanie týchto scén. Odpoveďou na takéto hodnoverné zobrazovanie je simulácia globálneho osvetľovania. Na desktopových aplikáciách zameraných na kvalitné vizualizácie je dnes už simulácia globálneho osvetľovania bežným štandardom. Ako príklad uvádzam *Cycles renderer* pre Blender, ktorý je založený na algoritme sledovania lúčov *(ang. Ray-Tracing)*, ktorého scénu môžeme vidieť vykreslenú na obrázku 1.

Obrázok 1
prebratý z: https://www.blender.org/manual/render/cycles/introduction.html

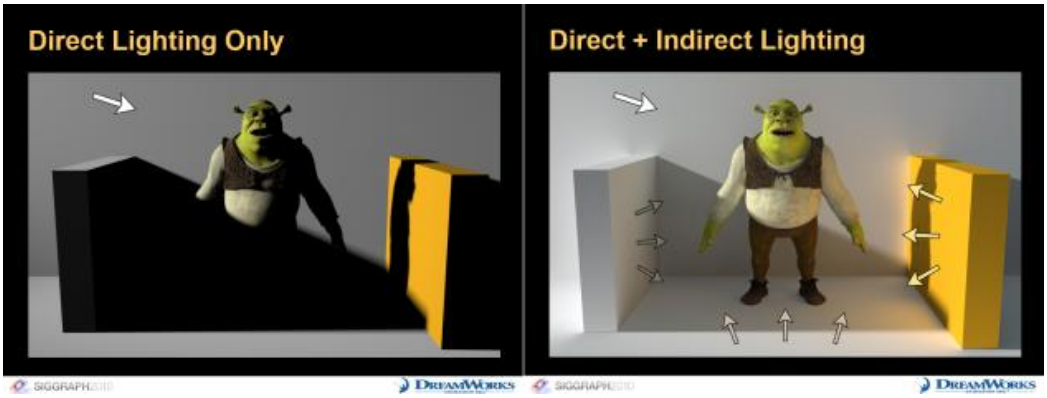
Mnohí vývojári takýchto pokročilých vykresľovacích softvérov si uvedomujú, aké množstvo výhod by priniesla možnosť pracovať s týmto druhom softvéru priamo vo webovom prehliadači. Jedným z hlavných dôvodov je dostupnosť takéhoto druhu softvéru pre akékoľvek zariadenia s podporou webových prehliadačov. Ďalšou silnou motiváciou je možnosť pracovať kooperačne na jednej scéne.

Vďaka tejto potrebe vzniklo za uplynulé roky množstvo rôznych projektov zameraných na vytvorenie kvalitných vizualizácií na webe. Vznikli webové štandardy umožňujúce pracovať s grafikou. Typickými predstaviteľmi projektov implementovaných na GPU sú Adobe Shockwave, Java3D, X3D, 3DMLW a WebGL.

Po implementácii týchto štandardov vzniklo množstvo aplikácií zameraných na modelovanie na webe. Príkladmi sú SculptGL, Clara.io, 3Dtin, ThreeFAB

[SCULPT], [CLAR], [3Dtin], [THREE]. Komunita dizajnérov modelujúcich v takýchto aplikáciách narastá, a s ňou aj potreba vykresľovať takéto modely čo najrealistickejšie.

Pri implementácií algoritmov ktoré riešia tento problém však narazili vývojári na niekoľko problémov súvisiacich s výkonnosťou zobrazovania takejto formy dát na internete, alebo s obmedzeným prístupom k GPU alebo CPU. Preto sa vo webových aplikáciách takéhoto typu využíva hlavne priame osvetľovanie, ktorého algoritmy sú výpočtovo jednoduchšie. Na obrázku číslo 2 vidíme rozdiel medzi priamym osvetľovaním a globálnym osvetľovaním, ktoré v sebe spája priame a nepriame osvetľovanie scén.



Obrázok 2
prebratý z: https://colinbarrebrisebois.com/category/rendering/global-illumination/

Na nasledujúcich riadkoch si stručne vysvetlíme tieto pojmy a ich koncepty. Priame osvetľovanie (*ang. direct illumination*), taktiež nazývané lokálny osvetľovací model, simuluje svetlo dopadajúce k pozorovateľovi po jednom odraze v scéne. Najtypickejším príkladom takéhoto osvetľovania je Phongov osvetľovací model. [*ZAR10*]. Globálne osvetľovacie *(ang. Global illumination)* metódy počítajú taktiež s odrazenými lúčmi a lomenými lúčmi v scéne. Spájajú teda priame a nepriame osvetľovanie, znamená to, že jeden objekt v scéne ovplyvňuje vykresľovanie iného objektu. Podstatou týchto metód je nájdenie riešenia zobrazovacej rovnice pre konkrétnu scénu. Analytické riešenie je vo väčšine prípadov nemožné, preto sa využívajú Monte Carlo Metódy. [ZAR10]

Medzi algoritmy riešiace problém globálneho osvetľovania patria intenzita žiarenia *(ang.Radiosity)*, sledovanie lúčov *(ang. Ray-Tracing)*, sledovanie ciest *(ang. Path-Tracing)*, metóda Metropolis Light Transport a takisto zatienenie okolím *(ang. Ambient occlusion)*

Intenzita žiarenia je metóda globálneho osvetľovania, ktorá je založená na šírení svetelnej energie. Táto metóda je založená na princípe zachovávania energie. Bola predstavená v roku 1984 výskumníkmi na Corrnell University. Sledovanie lúčov je metóda, ktorá sleduje odrazy lúčov v scéne. Bola prezentovaná v roku 1968. Algoritmus bol pôvodne označovaný ako algoritmus vrhania lúčov do scény *(ang. Ray-Casting)*. Idea algoritmu spočíva vo vystreľovaní lúčov z oka pozorovateľa, jeden pre každý pixel, a nájdení najbližšieho prieniku lúča s objektom. Sledovanie ciest je metóda rozširujúca metódu sledovania lúčov, ktorá sleduje možnosť cesty lúča, ktorý vychádza zo svetla a po interakcií so scénou dopadá do daného pixlu virtuálnej kamery. V 1997 bola popísaná Metropolis Light Transport metóda, ktorá využíva už nájdené cesty z kamery do svetelných zdrojov za účelom zvýšenia výkonu v náročných scénach. Táto metóda využíva metódu obojsmerného algoritmu sledovania ciest. Zatienenie okolím je metóda tieňovania, ktorá započítava tlmenie svetla zatienením.

1.1 Cieľ práce

Mojim cieľom je vytvorenie skriptu, ktorý na webe vykresľuje procedurálne scény v reálnom čase. Procedurálne scény sa vyznačujú vysokým objemom geometrie, ktorý sa získa zadaním určitého algoritmu pre pár objektov, a tie sú ďalej duplikované v scéne pomocou tohto algoritmu s určitou modifikáciou. Typickými scénami pre procedurálne modelovanie je napríklad scéna mesta z vtáčej perspektívy, les, alebo akákoľvek scéna obsahujúca množstvo podobných objektov. Takéto scény tvorím v aplikácií pomocou algoritmu sledovania ciest vo WebGL a GLSL. WebGL je javascriptové API pre vykresľovanie 3D a 2D grafiky bez inštalovania akýchkoľvek pluginov kompatibilné s webovými štandardmi. OpenGL Shading Language skrátene GLSL je vysokoúrovňový jazyk na implementáciu grafiky založený na syntaxe jazyka C.

1.2. Úvod do WebGL

WebGL (Web Graphics Library) je vytvorené neziskovou organizáciou Khronos Group, a pozostáva z javascriptových knižníc a kódu shaderov, ktorý je vykonávaný priamo na grafickej karte, teda patrí do kategórie hardvérovo založeného renderingu [CANT12]. K tomuto popisu je potrebné povedať ešte pár zásadných vlastností tejto technológie. WebGL je API (*ang. Application Programming Interface*), a teda je to

rozhranie určené k programovaniu aplikácii a v prehliadači je sprístupnené pomocou súboru Javascript-ových knižníc. WebGL je založené na OpenGL ES 2.0. OpenGL je priemyslový štandard špecifikujúci multiplatformové rozhranie API pre tvorbu aplikácii počítačovej grafiky . OpenGL ES *(OpenGL for Embedded Systems)* je odľahčená verzia OpenGL, ktorá sprístupňuje 3D grafiku pre tablety a mobilné zariadenia pod operačnými systémami Android a iPad. WebGL je cross-platformové a taktiež je ho možné kombinovať so zvyšným obsahom web stránky tzn. 3D canvas môže pokryť celú plochu stránky alebo len jej časť. WebGL je webový štandard, ktorý je voľný k používaniu pre každého. [PAR12]

K správnej predstave o WebGL považujem za adekvátne opísať ešte štruktúru WebGL aplikácie. Ku korektnému vykresleniu WebGL na webovej stránke je potrebné prejsť minimálne nasledovnými krokmi. [PAR12]

1.) Vytvoriť HTML5 element Canvas.

2.) Celý WebGL rendering sa nachádza v Javascriptovom DOM objekte *context.* Táto štruktúra odzrkadľuje 2D vykresľovanie ktoré je vykonávané v HTML5 elementom *canvas.* V tomto kroku je potrebné prepojiť *canvas* a *context.*

3.) Inicializovať Viewport a určiť hranice vykresľovacieho okna.

4.) Vytvoriť štruktúry pre dáta, ktoré budú vykresľované. WebGL vykresľuje obraz pomocou takzvaných primitív, patria medzi ne napríklad vrcholy, čiary, trojuholníky *(ang. triangle sets)* a pásy trojuholníkov *(ang. triangle strips).* Uchovávajú sa v poliach a zásobníkoch (*ang. buffer*).

5.) Aplikovať matice transformácii na tieto dáta. Väčšinou ide o transformáciu, rotáciu a škálovanie.

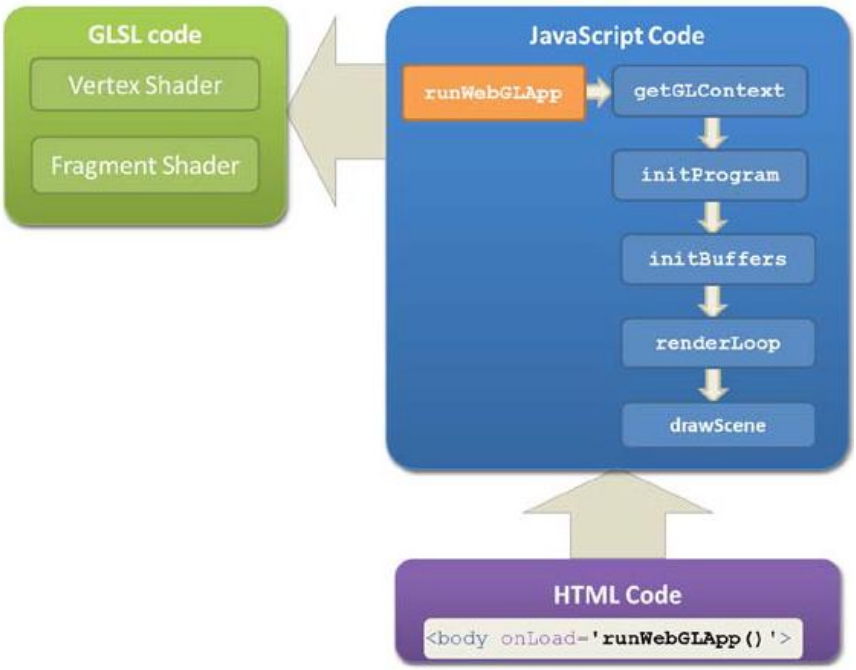
6.) Inicializovať shadere. Shader je počítačový program, slúžiaci k riadeniu jednotlivých častí programovateľného grafického reťazca grafickej karty (tzn. GPU).

7.) Vykresliť.

Po splnení týchto krokov a po načítaní stránky v internetovom prehliadači by sa nám malo zobraziť okno s vykreslenou scénou.

1.3. Úvod do vykresľovania vo WebGL

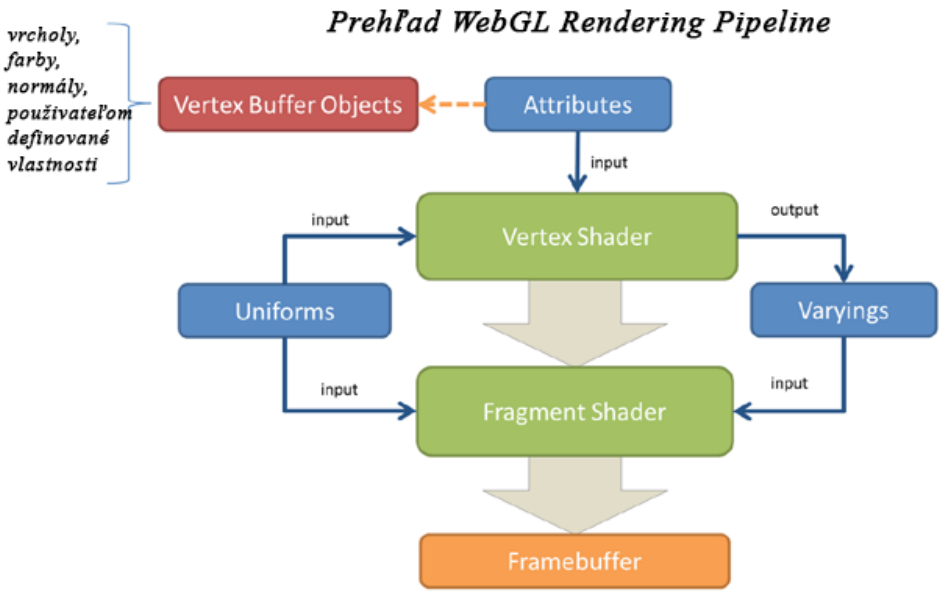
V tejto časti si priblížime štruktúru WebGL aplikácie, jej implementáciu a vykreslenie so základnými procesmi prebiehajúcimi na jej pozadí. Obrázok 3. popisuje základnú štruktúru WebGL aplikácie.



Obrázok 3: Štruktúra WebGL aplikácie

Inicializácia WebGL aplikácie začína pri HTML elemente canvas odkiaľ vchádzame do WebGL, kde inicializujeme všetky objekty a vlastnosti našej scény. Následne tieto hodnoty vchádzajú do GLSL kódu, kde sa vo vertex shaderi a fragment shaderi prepočítajú a odkiaľ vychádzajú ako výstup tzn. vykreslenie scény. Tento proces prebiehajúci medzi WebGL aplikáciou a shaderami sa nazýva aj WebGL Rendering Pipeline.

Keďže celá moja práca súvisí s vykresľovaním vo WebGL, je nesmierne dôležité, aby sme WebGL Rendering Pipeline proces pochopili správne. Názorne je tento proces opísaný na obr.2 kde sú na začiatku vstupom vrcholy, farby, normály a ďalšie nastavenia [CANT12]



Obr.4 WEBGL Rendering Pipeline
https://www.safaribooksonline.com/library/view/webgl-beginners-guide/9781849691727/ch02s02.html

1.3.1. Vertex Buffer Objects (VBOs)

VBOs obsahujú dáta, ktoré WebGL potrebuje na popísanie scény, ktorú ide vykresliť. Patria sem hlavne súradnice vrcholov, normály vrcholov, farby, súradnice textúr ale aj ďalšie používateľom nastavené parametre.

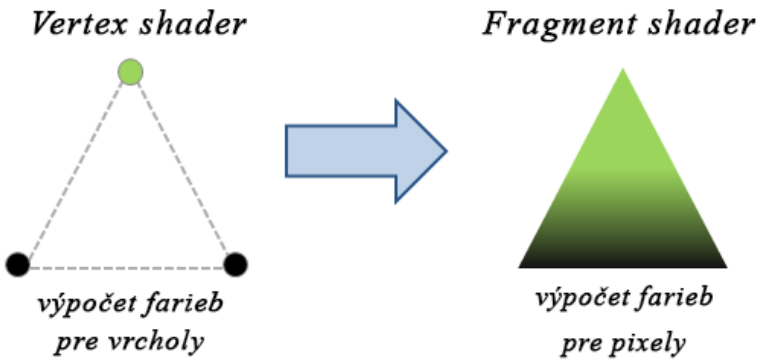
1.3.2. Vertex Shader

Vertex shader je program, ktorý sa vykoná pre každý vrchol geometrie scény. Medzi najčastejšie funkcie tohto programu patrí transformácia do súradnicového systému kamery a následná projekcia do zobrazovacej roviny. Vždy do programu vstupuje jeden vrchol, je upravený a vystúpi z programu. [CANT12]

1.3.3. Fragment Shader

Každá množina troch vrcholov definuje trojuholník a každý element na jeho povrchu musí mať priradenú farbu, inak by bol priehľadný. Každý element na povrchu je nazývaný fragment a pokiaľ hovoríme o fragmentoch, ktoré sú zobrazené na našom

displeji, môžeme ich nazývať pixely. Hlavnou úlohou Fragment shadera je teda vypočítanie farby pre každý pixel. Táto idea je vystihnutá aj na obr. 5.



Obr.5 Prevod medzi Vertex shaderom a Fragment shaderom

1.3.4 Framebuffer

Framebuffer je 2D pamäť *(ang. buffer),* ktorá na displej prenáša hodnoty fragmentov z Fragment shadera pre jednotlivé pixely. Obsah buffera je obvykle lineárny úsek operačnej pamäte, ktorý sa skladá z číselných hodnôt farieb (odtieňov šedej, RGB) pre každý pixel. Framebuffer je výstup Rendering pipeline. [CANT12]

1.3.5. Attributes, Uniforms a Varyings

Attributes sú vstupné hodnoty používané vo vertex shaderi ako napríklad súradnice vrcholov a farby vrcholov. Z dôvodu, že vertex shader je volaný pre každý vrchol, atribúty sú vždy hodnoty daného vrcholu. *Varying* sú ich obdobou pre fragmenty. Ich hodnoty sa získajú tak, že atribúty vrcholov každého trojuholníka v scéne sú v jeho vnútri interpolované pomocou barycentrických súradníc. *Uniforms* sú spoločné vstupné hodnoty pre vertex shader aj fragment shader. Narozdiel od *attributes* a *varyings*, sú *uniforms* konštantné hodnoty počas celého procesu vykresľovania *(ang. rendering cycle)*. Sú to hodnoty, ktoré popisujú nastavenia, ktoré sú pre celú scénu konštantné napr. pozícia svetiel. [CANT12]

2. Súvisiace práce

Aj napriek možným problémom implementovania globálneho osvetľovania je na internete množstvo projektov, ktoré dokazujú, že takáto implementácia nie je nemožná. Väčšina z nich však vznikla len pred pár rokmi, čo je zapríčinené aj donedávna obmedzeným prístupom k tvorbe grafiky na webe. Mojimi príkladmi aplikácií využívajúcich tento algoritmus riešenia zobrazovacej rovnice sú dve riešenia implementované vo WebGL a ďalšie dve implementované v OpenGL.

2.1 WebGL Path Tracing

Evan Wallace vytvoril v roku 2010 aplikáciu na vizualizáciu 3D scény pomocou metódy sledovania ciest, ktorú nájdete na adrese [WEBGL]. Evanova aplikácia vykresľuje scénu do HTML *canvas* elementu v rozlíšení 500x500 pixelov. Scéna obsahuje kváder, guľu a jedno svetlo vnútri známeho Cornell box modelu. Cornell box model sa stal často používaným 3D modelom na testovanie fotorealistických rendering metód. Jeho výhodou je ľahké vytvorenie si takéhoto modelu v reálnom svete a následné porovnanie fotorealistických techník s modelom vytvoreným našim softvérom. Možnosti výberu materiálu pre objekty sú tu tri: difúzny, zrkadlový a lesklý materiál. Scénu je možné rotovať okolo jej globálneho súradnicového systému a objekty je možné presúvať, odstraňovať, pridávať a meniť ich materiály. Celá scéna je dynamicky kompilovaná v GLSL shaderi.

2.2 More Spheres

More Spheres je shader ktorý vytvoril Reinder Nijhoff v roku 2013. Tento algoritmus sledovania ciest vizualizuje scénu s množstvom skákajúcich gúľ, okolo ktorých rotuje kamera. Na zefektívnenie vizualizácie je použitá priestorová dátová štruktúra Mriežka. Vďaka tejto dátovej štruktúre je možné vizualizovať veľké scény nakoľko náš algoritmus nemusí vyhodnocovať všetky objekty na nájdenie prieniku pre jeden lúč. [NIJ13]

2.3 Jednoduchý Path-Tracer

Iñigo Quílez vytvoril v roku 2012 experiment v ktorom vykresľoval Sierpinski fraktál pomocou Path-Tracing metódy s tromi odrazmi a 256 vzorkami pre každý pixel v rozlíšení 1280x720 pixelov. Tento algoritmus bol implementovaný v GLSL na OpenGL. [QUI12]

2.4 Brigade Renderer

Brigade je pokročilý renderer pre počítačové hry na GPU. Táto aplikácia nie je implementovaná na webe, napriek tomu ju uvádzam pretože používa vykresľovanie scén pomocou Path-tracing algoritmu v reálnom čase. Používa unbiased path tracing algoritmus s ruskou ruletou. Algoritmus spočíva v nájdení počtu ciest, ktoré spájajú kameru so svetelným zdrojom cez žiadny alebo niekoľko odrazov lúčov od objektov v scéne, použitím náhodných ciest. V algoritme sa používajú dve rozšírenia. Prvou je ruská ruleta, ktorá sa používa k zníženiu množstva veľmi dlhých ciest a druhým rozšírením je, že pri každom prieniku s nezrkadlovým materiálom, priame osvetľovanie *(ang. direct light)* je explicitne použité. Takýto renderer musí spĺňať mnoho veľmi špecifických požiadaviek, z ktorých pravdepodobne najvyššiu prioritu má požiadavka na real-time vykreslenie scény. Štandardom pri moderných hrách sa stalo 60 snímok za sekundu *(ang. fps).* [BIK13]

3. Teória

Väčšina svetla ktoré vnímame, je svetlo odrazené od povrchu objektov. Svetlo dopadajúce do ľudského oka priamo zo svetla je skôr výnimkou. Farba objektu je daná spektrálnou charakteristikou svetla, ktoré na objekt dopadá, ale predovšetkým vlastnosťami jeho povrchu, hlavne vlnovými dĺžkami a smerom v ktorom ich odráža. Je zrejmé, že potrebujeme formálny aparát, ktorý nám umožňuje popísať schopnosti materiálu, svetlo odrážať alebo absorbovať. Neexistuje ale žiadna formálna rovnica, ktorá by platila pre všetky druhy povrchov a zachovávala všetky vlastnosti, ktoré daný povrch alebo svetlo môže mať. Preto si definujeme nasledujúce výnimky: 1.) Svetlo, ktoré sa od povrchu odrazí, sa odrazí okamžite v súlade s pravidlom nekonečnej rýchlosti šírenia svetla. Kvôli tomu ale nebudeme môcť simulovať javy ako fosforescencia *(ang. Fosforescence)*, teda "nabitie" materiálu svetlom a jeho oneskorené vyžiarenie. 2.) Fotón s vlnovou dĺžkou bude odrazený od objektu s rovnakou vlnovou dĺžkou. 3.) Svetlo, ktoré dopadne do nejakého bodu, bude z tohto bodu aj odrazené. Svetlo, ktoré je odrazené z iného bodu ako z bodu kam dopadá musí cestovať pod povrchom materiálu. Táto vlastnosť sa anglicky nazýva aj *Subsurface Scattering*.[ZAR10]

3.1 Obojsmerná odrazová distribučná funkcia

BRDF *(ang. Bidirectional Reflectance Distribution Function)* je označenie obojsmernej distribučnej funkcie odrazu svetla. Prvý krát bola vyjadrená Fredom Nicodemusom v roku 1965. Táto funkcia sa používa ako vyjadrenie vlastnosti povrchu. Udáva subkritickú hustotu pravdepodobnosti, že sa svetlo, ktoré na povrch dopadne, odrazí daným smerom..

3.2 Zobrazovacia rovnica

Kajiya formuloval zobrazovaciu rovnicu *(ang. rendering equation)* [KAJI86], ktorá je matematickou formuláciou problému prenosu svetla v rámci scény. Jej riešenie udáva pre každý bod povrchu každej plochy a pre každý smer vychádzajúcu intenzitu žiarenia. Vychádzajúcim kritériom zobrazovacej rovnice je zákon zachovania energie. Intenzita žiarenia opúšťajúca bod povrchu musí byť niekde odrazená alebo pohltená. [ZAR10]

Matematická formulácia osvetľovacieho problému umožňuje jeho riešenie ale nanešťastie zobrazovacia rovnica je zložitým integrálnym vzťahom, ktorého analytické riešenie je vo všeobecnosti nemožné a musí sa preto hľadať aproximujúce riešenie. Všetky globálne osvetľovacie metódy sa dajú popísať ako čiastočné riešenie zobrazovacej rovnice [ZAR10].

Pri výpočte zobrazovacej rovnice musíme brať do úvahy nasledovné:

- rozdiel medzi lúčom, ktorý dopadá na objekt a lúčom, ktorý je odrazený sa rovná rozdielu medzi energiou lúča, ktorá je pohltená objektom a energiou, ktorá je vyžarovaná do okolia.[PHAR10]

Zobrazovacia rovnica je potom nasledovná:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{s^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |cos\theta_i| d\omega_i$$

obr.7 Zobrazovacia rovnica
Prevzaté z Physically Based Rendering - From theory to implementation

Kde L_o je energia odrazeného lúča, L_i je energia dopadajúceho lúča, L_e je energia emitovaná do prostredia, p je bod z ktorého lúč vychádza, ω je smer lúča, S^2 je hemisféra v smere ω_o z bodu p, θ_i je veľkosť uhla medzi ω_i a normálou plochy, na ktorej sa nachádza bod p, funkcia je distribučná funkcia odrazu (BRDF) v bode p zo smeru ω_o *do* ω_i .

Potom podľa rovnice je náš odchádzajúci lúč L_e definovaný ako súčet L_e žiarenia emitovaného z tohto bodu a integrálu, ktorý je derivovaný podľa smeru odchádzajúceho lúča. Tento integrál je zložený zo súčinu dopadajúceho svetla L_i , funkcie odrazu $f(p, \omega_o, \omega_i)$ a veľkosti uhla $cos\ \theta_i$ medzi bodom dopadu p a normálou na tento bod.

3.3 Monte Carlo metódy

Podstatou Monte Carlo metód je aproximácia riešenia problému stochastickým vzorkovaním. Hľadá sa náhodná premenná, ktorej stredná hodnota je riešením problému. Najčastejšie rozdelenie Monte Carlo metód je podmienené spôsobom vypočítavania trajektórie svetla. To sa vykonáva buď od pozorovateľa, od zdroja, alebo v oboch smeroch naraz. [ZAR10]

Medzi hlavné výhody týchto metód patrí používanie bodových vzorkovaní tzn. sledovanie lúča putujúceho scénou. Geometria scény môže byť procedurálna. Nemusíme mať k dispozícii trojuholníky aproximujúce pôvodný povrch. Objekty nie je potrebné deliť na menšie časti môžeme s nimi pracovať ako s celkami. Môžu pracovať s akoukoľvek BRDF. Ďalšími výhodami sú: jednoduchý výpočet pre zrkadlový odraz, nízka pamäťová náročnosť. Presnosť riešenia je daná počtom pixelov. To je závislé od pohľadovej závislosti riešenia. Vypočítavame obraz, ktorého presnosť je v prvom rade závislá od jeho rozlíšenia. Empirická zložitosť týchto metód je O(log n), kde n je počet objektov v scéne. Najrýchlejšie metódy založené na konečných prvkoch majú zložitosť *O(n*log n)*. Riešenie nie je ovplyvnené systematickou chybou *(ang. unbiased).*

Jednou z hlavných chýb týchto metód je šum, ktorý môže byť veľmi silný, a pre niektoré prípady ťažko odstrániteľný. Ďalšou z nevýhod metód Monte Carlo je ich pomalá konvergencia. Presnosť riešenia rastie s $\sqrt{n}$, kde n je počet vzoriek. Na zdvojnásobenie presnosti riešenia je teda potrebných štyrikrát viac vzoriek. Monte Carlo metódy poskytujú pohľadovo závislé riešenie.

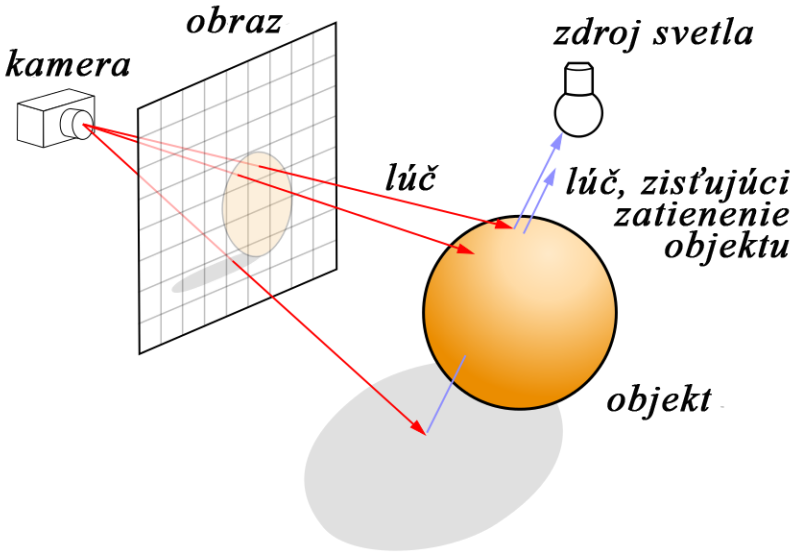
3.4 Sledovanie ciest *(ang.Path - Tracing)*

Sledovanie ciest je algoritmus, ktorý pomocou Monte Carlo integrovania rieši zobrazovaciu rovnicu a verne tak simuluje globálne osvetľovanie. Táto metóda bola prvýkrát popísaná v rovnakom článku ako zobrazovacia rovnica [Kaji86]. Algoritmus pre každý pixel výsledného obrazu opakovane simuluje tok svetla v scéne pomocou sledovania ciest lúčov. Táto metoda poskytuje kompletné riešenie zobrazovacej rovnice, aj keď za cenu veľkého výpočtového výkonu.

Tento algoritmus vie simulovať množstvo fotorealistických efektov, ako napríklad jemné tiene *(ang. soft shadows)*, hĺbka ostrosti *(ang. depth of field)*, rozmazanie pohybom *(ang. motion blur)*, kaustiky, prenos farby difúznym odrazom *(ang. color bleeding)*, zatienenie okolím *(ang. ambient occlusion)* a nepriame osvetľovanie .

Pre jednoduchosť algoritmus počíta s kamerou ako s bodom. Odtiaľ sa cez jednotlivé pixely vedie veľké množstvo lúčov do scény. V každom priesečníku lúča s objektom sa vypočíta lokálny osvetľovací model a detekcia tieňa. To znamená, že v každom odraze sa vyšle tieňový lúč do svetelného zdroja. V ďalšom kroku sa pomocou náhodného vzorkovania BRDF v priesečníku určí smer ďalšieho sledovania. Tento bod je

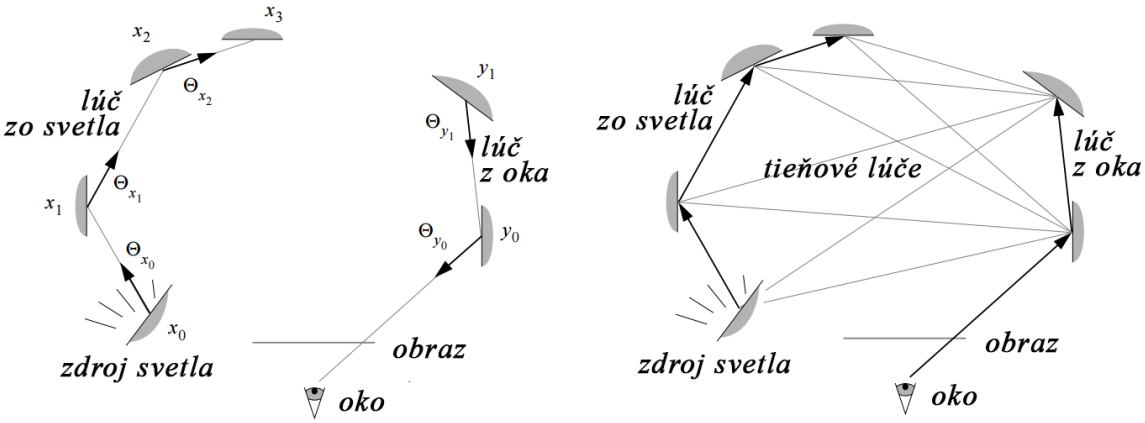
zásadným rozdielom oproti metóde sledovanie lúčov (*ang. Ray-Tracing*). Pokiaľ v metóde sledovania lúčov sa vypočítavajú zrkadlové odrazy, v metóde sledovania ciest sa vypočítava náhodný odraz. Pri použití metódy sledovania lúčov bude teda trajektória vždy rovnaká, pričom v prípade sledovania ciest bude trajektória pre každý lúč rôzna. Lúče sa v scéne odrážajú od objektov a algoritmus zisťuje, či nedopadnú do svetla. Ak áno, vypočíta príspevok v danom bode a vráti sa naspäť cez odrazené miesta postupne až ku kamere, pričom v odrazených miestach eliminuje intenzitu svetla v závislosti na vlastnostiach materiálu konkrétneho objektu v scéne. Hore uvedený algoritmus platí pre jeden pixel. Výsledná scéna vznikne, ak sa spustí niekoľkokrát pre každý pixel a v rámci jedného pixelu sa hodnoty spriemerujú. Tento algoritmus mám vizuálne znázornený na obrázku 7 [ZAR10].



Obr.7 Algoritmus sledovania ciest
prevzaté z https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg

3.4.1 Obojsmerný Path-Tracing

Obojsmerný Path-Tracing *(ang. Bi-Directional Path-Tracing)* je rozšírením algoritmu sledovania ciest. Hlavnou ideou je, že lúče sú vystrelené do scény z pohľadu kamery a zo svetla. Všetky body prienikov lúčov s objektmi sú potom spojené tzv. tieňovými lúčmi (*ang. shadow rays*) a primerané príspevky z nich sú pripočítané do výsledného vyhodnotenia farby každého pixelu. Tento algoritmus je popísaný aj na obrázku 8. [LAF93]



obr.8 Bi-Directional Path-Tracing
Prevzaté z Bi-directional Path Tracing [LAF93]

4. Metodika

4.1 Technické riešenie algoritmu sledovania ciest

Zatiaľ budem predpokladať že 3D scénu už mám namodelovanú v mojej aplikácií. Mojim cieľom je jej fotorealistická vizualizácia na displeji monitora, to znamená, že v scéne mám virtuálnu kameru. Okrem kamery a objektov sa v scéne nachádza zdroj svetla. Zo svetla dopadajú lúče na objekty, určitá energia sa pohltí a určitá energia je znovu odrazená. Časť lúčov dopadá do kamery. V mojom algoritme budeme túto cestu lúčov sledovať spätne, z dôvodu optimalizácie algoritmu. Pri vystreľovaní lúčov zo svetla, len malé percento dopadá do kamery a zvyšné lúče nemajú vplyv na výslednú vizualizáciu. Preto je efektívne sledovať cesty spätne z kamery do svetla.

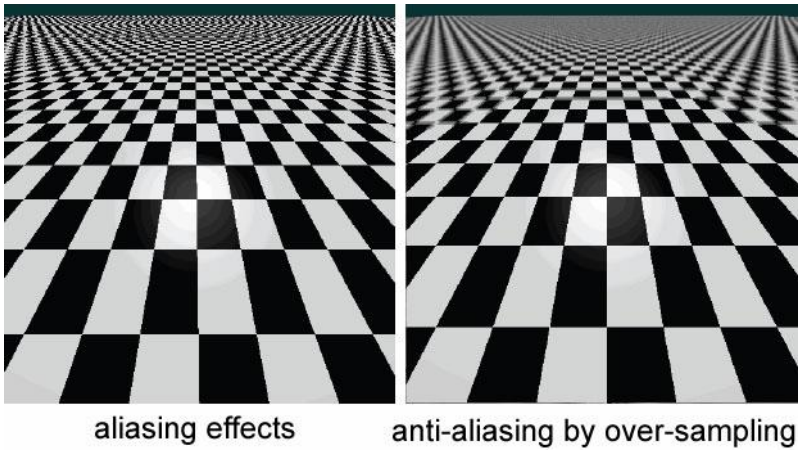
Výstup na našej kamere je Framebuffer v GLSL. Framebuffer je 2D pole pixlov. Každý z pixlov je popísaný pomocou farby a intenzity svetla dopadajúcich doň. Tieto pixle budem spracovávať vo WebGL Fragment shaderi. Z kamery budú vystreľované lúče do scény cez pixle. Pre každý pixel jeden lúč. Lúč bude prechádzať scénou a zastaví sa iba pri kolízii s nejakým objektom. V bode prieniku lúča s objektom sa vypočíta lokálny osvetľovací model. Podľa materiálu objektu a jeho vlastností sa vypočíta náhodný uhol odrazu pomocou BRDF a lúč sa odrazí týmto smerom. Okrem toho sa v danom bode vystrelí do svetla ešte takzvaný tieňový lúč na zistenie či je objekt zatienený. Tento postup sa opakuje až kým lúč nedopadne do svetla alebo nie je počet odrazov maximálny. Ak lúč dopadne do svetla farba pixlu sa vypočítavá pomocou akumulácie farieb všetkých povrchov cez ktoré sa lúč do svetla odrazil. Obraz s implementovanou množinou funkcií ktoré sa vykonávajú pre každý pixel sa nazýva akumulačná pamäť. Typickými predstaviteľmi operácií vykonávajúcich sa s akumulačnou pamäťou sú, pričítanie obrazu alebo výpočet priemeru z obsahu pamäti a nového obrazu.

Správnosť intenzity a farby uloženej v pixely je však priamoúmerne závislá od počtu lúčov vystrelených cez pixel pod rôznymi uhlami. Výsledná hodnota pre jeden pixel sa teda vypočíta pomocou stochastického sledovania všetkých možných ciest svetelných trajektórií. Celý tento proces sa nazýva vykresľovací prechod *(ang. Rendering pass)*. K získaniu hodnovernej vizualizácie scény musíme tento proces opakovať niekoľko krát. Takýto proces opakovania sa nazýva vzorkovanie.

4.1.1 Vzorkovanie

Vzorkovanie *(ang. Sampling)* je bežná technická záležitosť a vyskytuje sa všade okolo nás. Príkladom je videokamera snímajúca scénu v diskrétnych intervaloch napr. 24fps - vzoriek za sekundu. Hudba uložená ako súbor MP3 je postupnosť čísiel reprezentujúcich úroveň signálu snímaná s relatívne vysokou frekvenciou. Pri vzorkovaní však vzniká chyba *Alias* [ZAR10].

Alias vzniká pri rekonštrukcií signálu vzorkovaného pod Nyquistovým limitom a prejavuje sa ako nová nízkofrekvenčná informácia, ktorá nebola v pôvodnom signály prítomná. Ide o prípad, keď pôvodná funkcia obsahuje funkcie, ktoré nie je možné v rastrovanom výstupe zobraziť. Vyskytuje sa najmä pri vzorkovaní textúr a pri zobrazovaní objektov na ich hranách. Hranatému zobrazeniu čiar a okrajov polygónov *(ang. jaggies).* Táto chyba sa odstraňuje pomocou *antialiasingu* ktorý túto chybu odstraňuje napríklad rozostrením alebo zašumením. Príklad na aliasing a antialiasing je na obrázku číslo 9.[ZAR10]



obrázok číslo 9:Aliasing a Antialiasing
prevzaté z: http://people.eecs.berkeley.edu/~sequin/CS184/IMGS/anti-aliasing.jpg

4.2 Algoritmus Path-Tracing vo WebGL

Za základ mojej práce som si vybral už spomínanú implementáciu *(viď kapitola 2.1)* vytvorenú Evanom Wallaceom a to hlavne z nasledovných dôvodov. Základný algoritmus v mojej implementácii by bol veľmi podobný tomu Wallaceovmu. Navyše pri tvorení tohto modulu úplne nanovo by som stratil veľa času implementáciou rovnakých častí, ktoré by boli takmer totožné s Wallaceovými.

V nasledujúcich riadkoch si stručne opíšeme a analyzujeme implementáciu Wallaceovej aplikácie a v druhej časti tejto kapitoly budem popisovať moju modifikáciu tohto kódu pre potreby vytvorenia rendering modulu pre procedurálne modely.

4.3 Analýza kódu

Naša aplikácia sa nachádza v *webgl-path-tracing.js* súbore a s hlavným HTML kódom stránky je prepojená pri inicializácii vo funkcii *window.onload* pomocou *canvas* elementu. V tejto funkcií sa inicializuje trieda *Ui(),* hodnoty prostredia, materiálu a prednastavená scéna z HTML tlačidlových elementov. Zaujímavá a dôležitá je *setInterval* funkcia, v ktorej sa spúšťa funkcia *tick(timeSinceStart)* v intervale 60 krát za sekundu. V nej je nastavená pomyselná kamera a volané funkcie na obnovenie materiálu, prostredia a *ui.update(timeSinceStart)* a *ui.render().*

Pre ďalšiu analýzu si rozdelíme Wallaceov kód na 5 celkov podľa tried alebo časti kódu. Prvým celkom je ***GLSL kód,*** nasledujú funkcie, ktoré môžeme popísať ako ***Utility(),*** ďalej ***Triedy objektov***, triedy ***Ui, Renderer a Pathtracer*** a ***Hlavná časť***.

4.3.1 Hlavná časť

V tejto časti sa nachádza inicializácia našej aplikácie, funkcie inicializácie objektov v scéne, už spomínaná *window.onload* funkcia a ďalšie funkcie zaznamenávajúce vstupy či už z klávesnice alebo z myši, ako napríklad *document.onkeydown* alebo *document.onmousedown* funkcie.

4.3.2 Triedy Ui , Renderer a Pathtracer

Na začiatku je dôležité poznamenať, že tieto triedy sú prepojené a to nasledovne. V triede *Ui* sa nachádza trieda *Renderer* a v nej trieda *Pathtracer*.

Trieda *Ui* vykonáva interakciu počítačovej myši a klávesnice s aplikáciou. Obsahuje funkcie, ktoré pracujú s triedou *Renderer* a vypočítavajú pre ňu vstupné hodnoty na vykresľovanie. Jednou z takýchto funkcií je aj *mouseDown* funkcia, ktorá zisťuje, či sme pri kliknutí do vykresľovacieho okna klikli na objekt. Ak áno, načítajú sa pozície

vrcholov tohto objektu a vo funkcií *mouseMove* sa tento objekt presúva a volajú sa funkcie na opätovné vykresľovanie.

Ak sme neklikli na objekt, mení sa pozícia kamery a prepočítava sa bod, z ktorého sú vystreľované lúče do scény. Ďalšími funkciami tejto triedy sú funkcie *SelectLight, addSphere, addCube, deleteSelection, updateMaterial, updateEnviroment* a *updateGlossiness*, ktoré podľa vstupov z myši, z klávesnice alebo z HTML tlačidlových elementov pridávajú objekty, mažú ich alebo menia materiály a prostredie.

Okrem už vyššie spomínaných vertex a fragment shaderov potrebných na vykresľovanie grafiky na GPU, sa v aplikácii používa aj Line shader. Tento program vykresľuje WebGL primitívy, ktorými sú čiary, ktoré ohraničujú objekt po kliknutí naň. Trieda *Renderer* pokrýva inicializáciu a vykresľovanie práve týchto primitív. Pri jej inicializácii sa vytvorí pole súradníc vrcholov, pole hrán a inicializuje sa GLSL kód na vykreslenie týchto primitív.

Pri inicializácií triedy Pathtracer sa vytvorí pole súradníc vrcholov. Vytvoria sa dve rovnako veľké textúry pre *framebuffer*. S týmito textúrami budeme ďalej pracovať vo funkcií *Pathtracer.update()*. Inicializujú sa GLSL funkcie *renderVertexSource a renderFragmentSource.* Trieda Pathtracer má 3 funkcie *setObjects(), update()* a *render().*

Vo funkcií setObjects() sa prepájajú naše objekty popísané a inicializované zatiaľ len v javascripte s funkciami napísanými v GLSL. Na tomto mieste sa kompiluje náš shader pre tieto objekty.

V *Pathtracer.update()* prebieha vykresľovanie do textúr framebuffera a to nasledujúcim spôsobom. Pri inicializácií tejto triedy Wallace vytvoril dve rovnako veľké textúry s rovnakým vnútorným formátom popisujúcim typ dát ukladajúcich sa do jedného pixelu. Po prejdení algoritmu sledovania ciest pre všetky pixely raz, tzn. jeden vystrelený lúč pre každý pixel, na prvú textúru sa načítajú hodnoty FBO (Frame bufffer objects). Keďže GLSL Fragment shader má problém čítať z jednej textúry a vzápätí na ňu hodnoty zapisovať, použije sa na zápis druhá textúra. Z našej prvej textúry sa budú hodnoty FBO načítavať a do druhej sa hodnoty zapíšu. Potom sa textúry vymenia. Táto technika sa volá prepínanie textúr *(ang. ping ponging textures)* a využíva sa v algoritmoch, pri ktorých potrebujeme hodnoty po jednom vykreslenom výstupnom obraze znovu načítať a počítať s nimi pri ďalšom obraze.

4.3.3 Objekty

Objekty *kváder* a *guľa* majú svoje vlastné triedy a do aplikácie vstupujú po inicializácií jednej zo scén vo funkcii *window.onload()*. V týchto objektoch sa nachádzajú funkcie, ktoré prepájajú javascript kód s GLSL kódom a vypočítavajú lúč - objekt prieniky, výpočet normál týchto objektov pre správne nastavenie odrazu lúčov od objektov, funkcia zisťujúca nachádzanie sa objektu v tieni, funkcie prevádzajúce javascript súradnice vrcholov objektov do GLSL kódu a funkcie na vykreslenie ohraničenia vybraného objektu (*ang. bounding box*) pri jeho presúvaní. Do tejto skupiny zaradíme aj objekt *Light*. Jeho funkcie a odlišujú od funkcií zvyšných dvoch objektov a obsahuje len funkcie prevodu súradníc do GLSL kódu a funkciu *clampPosition.*

4.3.4 Utility funkcie

Utility funkcie sú funkcie, ktoré Wallace implementoval pre zjednodušenie ostatných algoritmov a funkcií. Medzi tieto funkcie patria napríklad vytvorenie 3D a 4D vektorov pre súradnice vrcholov a matematické operácie s týmito vektormi. Okrem týchto funkcií sa tu nachádzajú ale aj veľmi dôležité funkcie, a to sú *compileSource* a *compileShader*. Pri každej inicializácii aplikácie sa nami vytvorený shader v javasripte spája s shaderom vytvoreným v GLSL a overí sa, či nenastala nejaká neočakávaná chyba. Ak áno, túto chybu program vypíše.

V utility funkciách sa nachádza aj funkcia *concat*, ktorá prechádza určitú množinu (tzn. v našom prípade množinu objektov), ktoré majú totožné vlastnosti a niektoré funkcie a tieto funkcie pre daný objekt zavolá. *setUniforms* je funkcia, ktorá spája Javascript hodnoty, ktoré majú byť v GLSL uniformami teda nemennými hodnotami.

4.3.5 Kód v GLSL

Funkcie v GLSL sú jadrom aplikácie. Pre každý pixel sa v nich vykonáva výpočet cez vertexshader do fragmentshaderu a tento prevod je ovplyvnený path-tracing algoritmom, ktorý je v nich simulovaný. V našom súbore sú zabalené do javascriptových funkcií. Tieto funkcie by sme mohli rozdeliť do piatich kategórii podľa ich významu.

Do prvej kategórie spadajú funkcie, ktoré inicializujú vertex shader, fragment shader, line shader a funkcie, ktoré vykonávajú vykresľovanie cez tieto shadere. Sú nimi

renderVertexSource, renderFragmentSource, lineVertexSource, lineFragmentSource, tracerVertexSource a *tracerFragmentSource*.

Druhú kategóriu tvoria funkcie na výpočet lúč - objekt prienikov a výpočet ich normál. V týchto funkciách sa zisťuje, či náš vystrelený lúč dopadá do objektu alebo ho minie, tak isto sa tu prepočítavajú normály objektov. Tieto funkcie sú volané z funkcií jednotlivých objektov.

Treťou kategóriou sú funkcie, ktoré zabezpečujú dostatočne náhodné uhly odrazov lúčov. Sú nimi *randomSource, cosineWeightedDirectionSource, uniformlyRandomVectorSource a uniformlyRandomDirectionSource.*

Štvrtá skupina funkcií obsahuje funkcie *specularReflection, newDiffuseRay, newReflectiveRay, newGlossyRay, yellowBlueCornellBox a redGreenCornellBox.* Z ich názvov vyplýva, že sú to funkcie, ktoré nastavujú ako sa budú lúče odrážať od jednotlivých materiálov. Práve pri rôznych materiáloch využívame funkcie pseudonáhodných odrazov z predchádzajúcej skupiny.

Do piatej skupiny spadajú zvyšné funkcie GLSL kódu a to sú *makeCalculateColor, makeShadow* a *makeMain*. Môžeme povedať, že práve tieto funkcie tvoria jadro našej aplikácie. Detailne ich opisujem v nasledujúcej kapitole.

4.3.5.1 Funkcie MakeCalculateColor, makeShadow, makeMain a makeTracerFragmentSource

Funkcia *makeTracerFragmentSource* je funkcia, v ktorej sa zavolajú všetky funkcie v ktorých sú GLSL funkcie popisujúce a pracujúce s našimi vertex a fragment shaderami a s pathtracing algoritmom. Táto funkcia je volaná v triede *Pathtracer.setObjects.* Vo funkcii *makeMain* je inicializované svetlo v GLSL a pripojená GLSL textúra, ktorú následne prechádzame po pixeloch a ukladáme do nich *gl_FragColor* premennú, do ktorej sa ukladá výsledok našej *makeCalculateColor* funkcie.

V *makeCalculateColor* sa vykonáva celý algoritmus pathtracing. Pseudokód je nasledovný:

```
calculateColor (počiatok, lúč, svetlo) {

    for (odraz=0; odraz  < max.odraz; odraz++) {
        vypočítaj prieniky so všetkym;
```

```
        nájdi najbližší prienik;
        hit = počiatok + lúč * prienik;
        if (najbližší prienik == CornellBox) {
            vypočítaj normálu CornellBox;
            lúč = newDiffuseRay;
         } else if (prienik == nekonečno) {
            break;
         } else if (všetky ostatné objekty) {
           vypočítaj normálu objektu;
           lúč = luč podľa materiálu objektu;
        }
           nájdi normálu k svetlu;
           zisti či je objekt zatienený;
           akumulovaná farba = farba povrchu + intenzita svetla + intnzita tieňa ;
           počiatok = hit;
    }
    return akumulovaná farba;
}
```

5. Riešenie a Implementácia

Na začiatok je dôležité povedať, že moju aplikáciu som vytvoril a testoval v prehliadači *Google Chrome verzii 50.0.2661.94 m* a na vykresľovanie vo WebGL je potrebné na adrese *chrome://settings* kliknúť na rozšírené nastavenia a skontrolovať, či je odkliknutá možnosť Používať hardvérovú akceleráciu. Ďalej na adrese *chrome://flags* skontrolujeme, či nie je odkliknuté Disable WebGL. Nakoniec skontrolujeme na adrese *chrome://gpu* pod Graphics Feature Status, či sú jednotlivé funkcie Hardware accelerated alebo Software only, hardware acceleration unavailable. Po overení týchto nastavení by sa nám mal WebGL obsah vykresľovať v Chrome prehliadači. Moju aplikáciu som testoval na grafickej karte NVIDIA GeForce 840M.

V nasledovnej časti popisujem všetky moje zásahy do kódu Wallacea a implementáciu mojich algoritmov na vykresľovanie procedurálnych modelov.

Jednou z mojich prvých modifikácii bola zmena okna na vykresľovanie mojej scény z 512x512 pixlov na celú plochu prehliadača, v ktorom je moja aplikácia spustená.

Na zrealizovanie vykresľovania na celú plochu prehliadača som musel upraviť kód na niekoľkých miestach. Najprv som musel zmeniť veľkosti textúr, na ktorých naše vykresľovanie prebieha. Vo funkcií *makeMain* a v triede *Pathtracer*. Ďalej som zmenil funkciu *onmousedown* aby po kliknutí na ktorékoľvek miesto na textúre zaznamenala pozíciu počítačovej myši a *v Ui.mouseDown, Ui.mouseMove* a *Ui.mouseUp* sa vypočítaval lúč na ktoromkoľvek mieste v textúre.

Ďalším zásahom bola zmena pohybu kamery v scéne. Okrem rotácie okolo stredu pomocou počítačovej myši je možné približovať a vzďaľovať sa pomocou šípok na klávesnici.

5.1 Zefektívnenie algoritmu

Wallaceová aplikácia je neefektívna najmä kvôli funkciám vykresľovania, ktoré spúšťa v cykle počas celej doby, kým má používateľ aplikáciu otvorenú v prehliadači. Kód som modifikoval vo *window.onload* funkcii tak aby vykreslenie nastalo len po interakcii s aplikáciou.

Ďalej som sa venoval zefektívneniu samotného path-tracing algoritmu v *makeCalculateColor* a to z dôvodu, že môj algoritmus modifikujem pre procedurálne modely, čo sú modely s vysokým obsahom dát. Skúsme odhadnúť výpočtovú zložitosť tohto algoritmu. Vo funkcií *makeMain* najprv prechádzame pixel po pixely po našej obrazovke v časti, kde je *canvas* element, zložitosť môžeme odhadnúť ako *O(n).* Pre výpočet hodnoty farby pixelu vchádzame do *makeCalculateColor* kde pre každý odraz vypočítavame prienik s *CornellBox* a potom so všetkými objektmi a tieto prieniky porovnávame pre nájdenie toho najbližšieho, na ktorom vypočítame farbu a vlastnosti prieniku a znovu prechádzame všetkými objektmi pri hľadaní toho, pre ktorý sme našli prienik a vypočíta sa jeho normála a zistí sa, či je zatienený alebo nie.

Pri desiatich objektoch to nespôsobí spomalenie, ale ak by sme chceli vykresľovať scénu zloženú z 500 objektov spomalenie by bolo výrazné.

Zefektívnil som to nasledovne. Pre každý objekt som inicializoval novú premennú *mid* ktorá popisuje jeho ID. Vo funkcií *makeCalculateColor* som nastavil premennú *hitID,* ktorá má na začiatku hodnotu prieniku s *CornellBox* modelom. Táto premenná je vstupno-výstupnou premennou pre všetky funkcie prienikov objektov. Vo vnútri týchto funkcií ju

porovnávam s momentálne vypočítaným prienikom. Táto implementácia nám ušetrila ďalšie prechádzanie všetkými objektami pre zistenie najbližšieho prieniku.

5.2 Objekty

Pod pojem procedurálne modelovanie spadá niekoľko techník tvorby scén s 3D objektov alebo textúr pomocou určitej množiny pravidiel. Typickými predstaviteľmi týchto techník sú L-systémy, fraktály a generované modelovanie, keďže tieto techniky používajú algoritmy pre tvorbu scén. Výhodou tvorby týchto modelov je, že tieto scény nemusia byť ukladané pomocou pozícií objektov a ich ďalších vlastností ale pomocou algoritmov a pravidiel. Vďaka tomu môžeme mať pomerne rozsiahlu scénu uloženú v pár kilobyteoch.

Vytvorenie takýchto algoritmov ale nie je cieľom mojej bakalárskej práce a spôsob, akým sú objekty v scéne tvorené, nemá vplyv na algoritmus sledovania ciest. Na otestovanie môjho algoritmu tak nepotrebujem bezpodmienečne procedurálne modely ale postačujú mi scény s veľkým počtom objektov. Ak môj algoritmus bude zvládať vykresľovanie takýchto scén, bude zvládať aj vykresľovanie procedurálnych modelov s rovnakým objemom geometrie.

Aj napriek faktu, že by som mohol scénu tvoriť len z dvoch objektov, kvádra a gule, rozhodol som sa do nej implementovať ďalšie objekty.

Z hľadiska implementácie nových modelov do mojej scény som vytvoril dva koncepty. Keďže moja scéna je zatiaľ tvorená len z dvoch objektov, prvým konceptom je vytvorenie ďalších objektov, ktoré dokážeme popísať analytickou rovnicou a dokážeme tak vypočítať ich prienik s lúčom. Do tejto kategórie patria jednoduché objekty ako napríklad kváder, kužeľ, valec, prstenec. Výhodou tejto skupiny modelov je ich nízka pamäťová náročnosť. Napríklad na vykreslenie gule zloženej z polygónov by sme ich museli použiť niekoľko stoviek, guľa sa dá ľahko analyticky popísať a preto mi na jej veľmi presné zobrazenie postačí pár parametrov a správny výpočet jej povrchu.

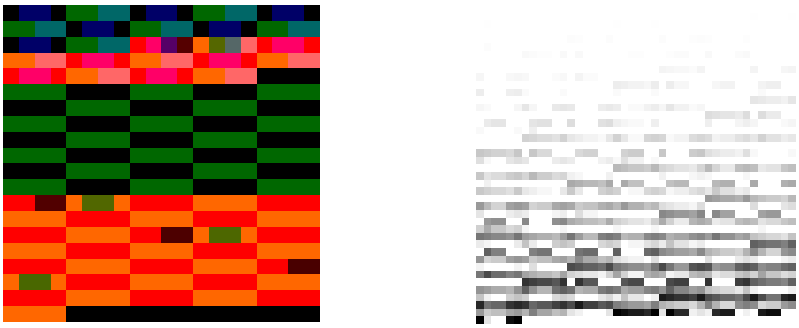
Druhým spôsobom je importovanie objektov do môjho modulu. Po konzultácii zo zadávateľkou sme zvolili importovanie objektov z OBJ. súborov. Tieto modely sa skladajú z malých polygónov. Výhodou je, že takto vieme vykresľovať akýkoľvek objekt pomocou rovnakej funkcie na výpočet prieniku pre každý segment. Segmentom pri týchto modeloch je mnohokrát trojuholník alebo iný mnohouholník. Hlavnou nevýhodou tohto typu objektov je, že priamoúmerne s kvalitou výsledného modelu sa zvyšuje počet polygónov čo priamo súvisí s narastaním pamäťovej náročnosti.

5.2.1 Mapovanie objektov do textúr:

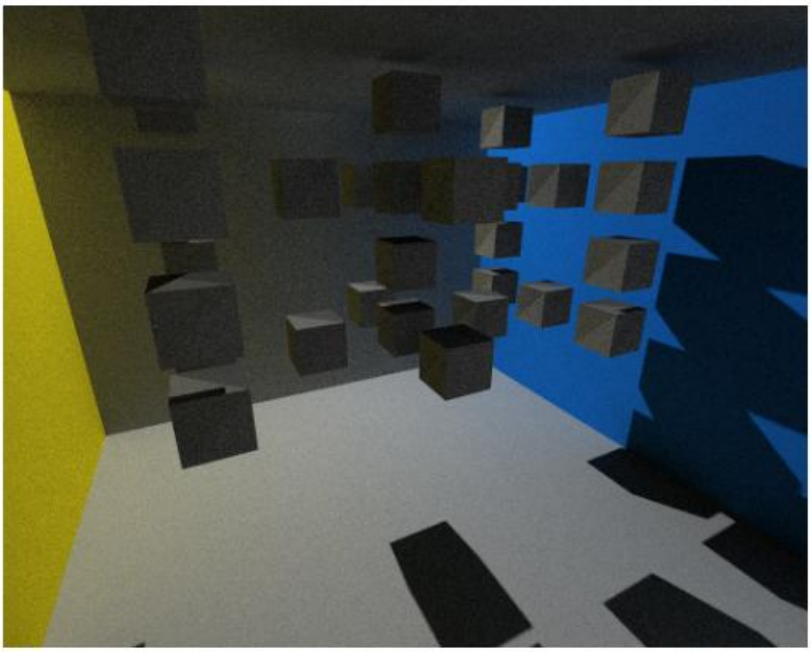
Pre vizualizáciu veľkého počtu objektov som musel vyriešiť zásadný problém. Ako som už uvádzal náš algoritmus sa nachádza na Fragment shaderi v GLSL. Fragment shader má ale limitovaný počet premenných. Musel som preto vyriešiť otázku ako čítať vrcholy a vlastnosti našich objektov vo Fragment shaderi bez ukladania ich do premenných. Wallace tvoril malú scénu z pár objektov a nevyčerpal tento limit. Pre procedurálne modely ale musím nájsť iný spôsob tieto VBO ukladať.

Textúra je vo WebGL špeciálny objekt do ktorého je možný veľmi rýchly zápis aj čítanie. Je to 2D pole v ktorom sú uložené hodnoty farieb. Možnosti formátov pre jeden pixel je vo WebGL niekoľko, sú nimi RGB, ALPHA, RGBA, LUMINANCE, LUMINANCE_ALPHA. RGB formát obsahuje tri zložky, každú v rozsahu <0,255> typu Byte pre celé čísla. Po hlbšej analýze textúr v GLSL som zistil že je možné nastaviť týmto hodnotám aj typ FLOAT.

Vytvoril som teda dve textúry pre mapovanie trojuholníkov pri importe objektov z .obj súborov. Štruktúra obj. súboru je uložená nasledovne. V poradí za sebou sa nachádzajú zoznamy pre vrcholy, normály, textúry a indexy trojuholníkov. Ja budem pracovať len s vrcholmi a s indexmi trojuholníkov. Do jednej textúry načítavam vrcholy trojuholníkov. Do druhej textúry načítavam indexy vrcholov. Na obrázku 10 môžeme vidieť textúru v ktorej máme uložené pozície vrcholov a pozície indexov a na obrázku 11 je vizualizácia objektov mapovaných do textúr.



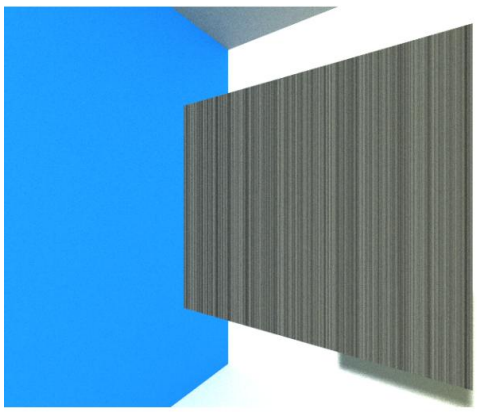
Obrázok 10: Textúra vrcholov veľkosti 20x20 pixlov, textúra indexov susedných vrcholov veľkosti 42x42 pixlov



obrázok 11: Vizualizácia objektov z .obj súboru namapovaných do textúr, rozlíšenie: 509x409px, počet odrazov lúčov: 2, materiál: difúzny, vyhladzovanie: 94

5.2.2 Lúč- objekt prieniky

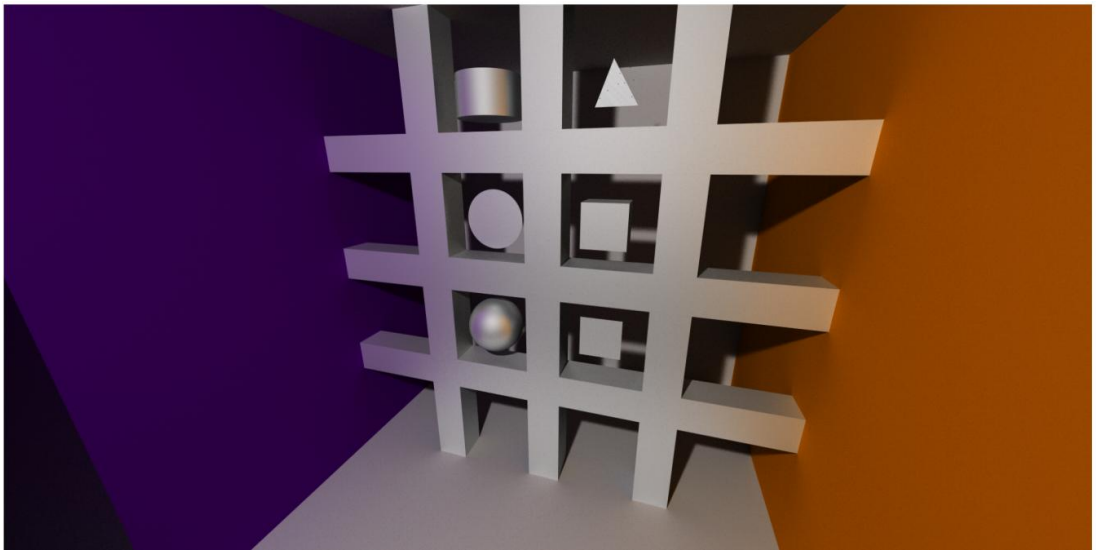
Do scény som sa rozhodol implementovať niekoľko nových objektov, ktoré je možné analyticky popísať a neskôr sa dajú použiť pri tvorení zaujímavých scén. Prvý objektom je objekt plocha ktorú som odvodil z Wallaceovej implementácie kvádra, pričom som vyhodnocoval ktoré dva rozmery sú nenulové. Tretí rozmer ale tiež nie je nulový, ale len blížiaci sa nule. Keď som tento rozmer nastavil rovný nule vznikol na mojom objekte takzvaný Z-Fighting. Z-Fighting je problém kedy sa viac primitív alebo stien primitív prelína cez seba. Tento problém je možné vidieť na obrázku 12.



Obrázok 12: Chyba vytvorená prelínaním dvoch stien objektu

Pre trojuholník som vytvoril triedu a popísal som jeho dva typy. Niekedy je zbytočne zaťažujúce pre náš algoritmus vykresľovať zadnú stranu objektu, v našom prípade trojuholníka, teda stranu, ktorá je pri pohľade kamery odvrátená. Takýto typ objektu sa anglicky nazýva *back-face culling*. Práve tento typ trojuholníkov budeme neskôr využívať pri vykresľovaní objektov zložených z polygónov.

Ďalším objektom ktorý som popisoval je valec. Pri ňom som musel vyhodnocovať aj možnosti kedy môže mať lúč až štyri prieniky cez objekt. Posledným objektom bol kruh ktorý som odvodil od triedy valec podobne ako už vyššie plochu od kvádra. Pri výpočte prienikov lúčov s našimi objektmi som vychádzal z analytického vyjadrenia týchto objektov. Na obrázku môžeme vidieť mnou implementované objekty. Všetky objekty ktoré som analyticky popísal môžete vidieť aj s Wallaceovými objektmi, kvádrom a guľou na obrázku 13.



Obrázok 13: Sprava z hora: trojuholník, valec, kváder, kruh, plocha a guľa, rozlíšenie: 1897x951px, počet odrazov lúčov: 3, materiál: lesk, vyhladzovanie: 128

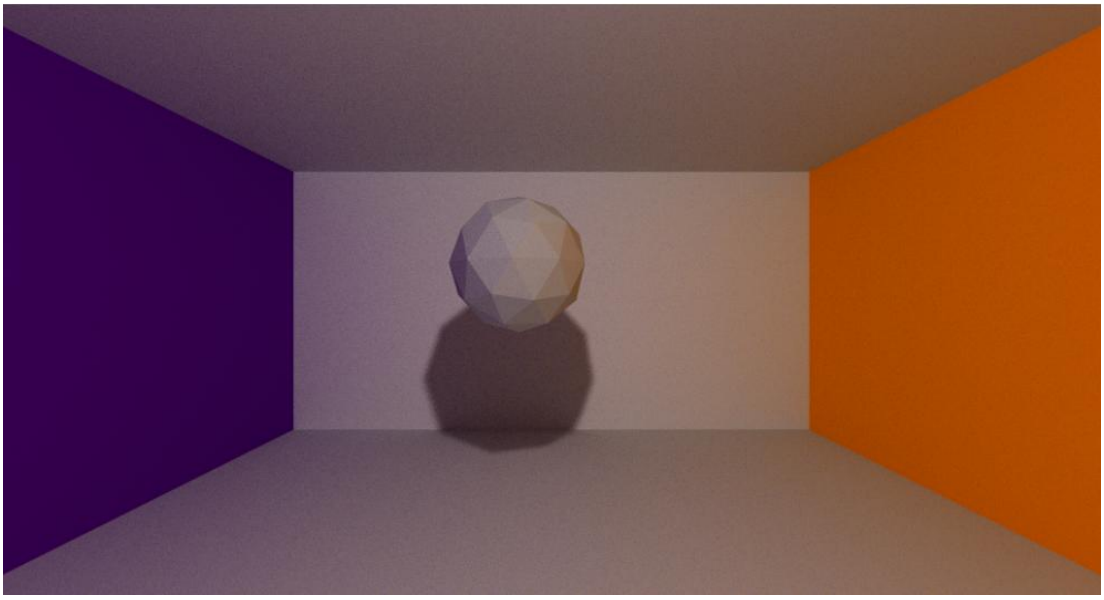
5.2.3 Objekty z .obj súborov

Keďže scény zložené z doteraz definovaných objektov nie sú veľmi zaujímavé doplnil som moju aplikáciu o možnosť načítania objektov z obj. súborov.

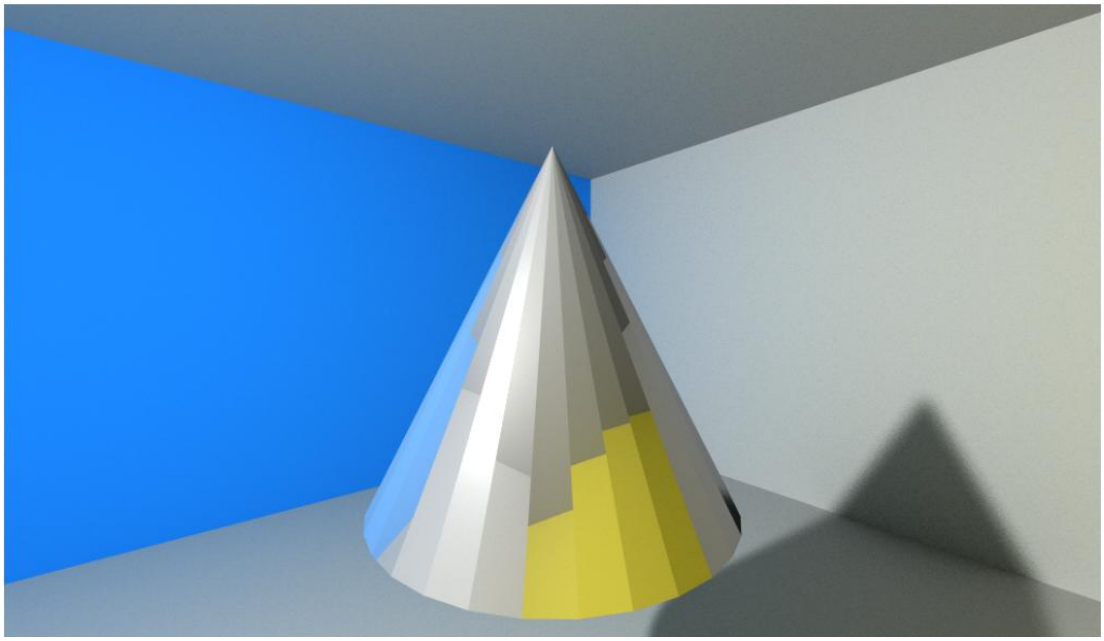
V mojej aplikácií som využil webgl-obj-loader [WOL]. Vo funkcií *Ui.loader()* načítam pomocou tohto skriptu .obj súbor z ktorého vyberám pozície vrcholov a ich indexy. Podľa veľkosti týchto dvoch polí si nastavím veľkosť našich bufferov tak aby simulovali štvorcové 2D pole a hodnoty vkladám do bufferov rovnomerne vzhľadom na

ich šírku a výšku. Keď prejdem celým poľom vrcholov a hrán na zvyšné prázdne miesta do textúr doplním nulové hodnoty.

Pre textúru vrcholov som zvolil formát pixlov v *gl.RGB* type a pole ktoré napájam na textúru je *Float32Array*. Pre textúru indexov som zvolil *gl.LUMINANCE_ALPHA* typ a pole je typu *Uint8Array*. S takto načítanými textúrami ďalej pracujem v *Pathtracer.update()* kde ich pripájam na naš Framebuffer a vo funkcií *makeMain(),* a *makeCalculateColor* kde z nich čítam pozície trojuholníkov a vyhodnocujem funkciu prieniku lúča s trojuholníkom, výpočtu normály a funkciu zatienenia. Na obrázkoch 14 a 15 môžeme vidieť objekty načítané z .obj súboru.

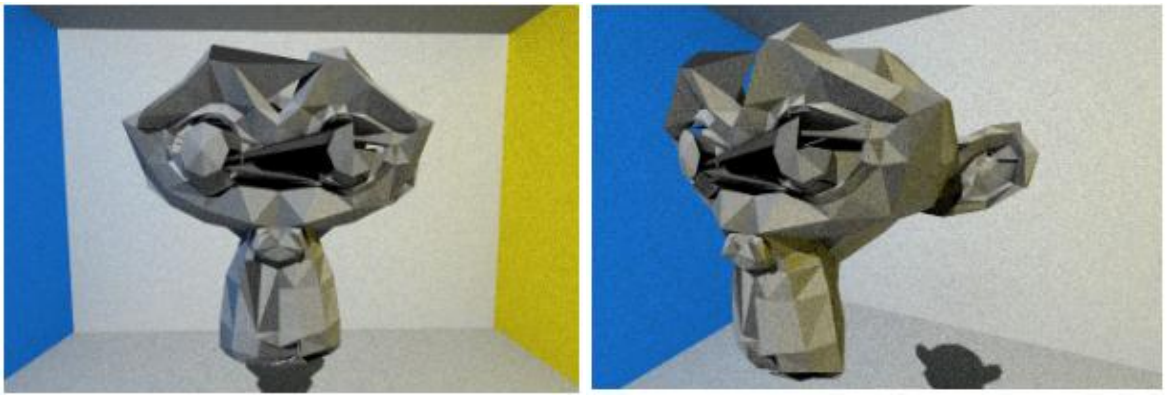


Obrázok 14: IcoSphere z .obj súboru, rozlíšenie: 992x535px, počet odrazov lúčov: 3, materiál: difúzny, vyhladzovanie: 128



Obrázok 15:Kužel z .obj súboru, rozlíšenie: 1017x602, počet odrazov lúčov: 3, materiál: zrkadlo, vyhladzovanie: 128

Pri niektorých .obj súboroch vznikala po načítaní z .obj určitá chyba. Nevedel som ju dostatočne dobre identifikovať a tak som ju nemohol odstrániť. Skúsil som niekoľko možností medzi ktorými som zisťoval či niektoré vrcholy trojuholníkov nie sú načítavané v smere hodinových ručičiek, keďže moja implementácia načítava trojuholníky len v smere proti hodinovým ručičkám. Takýmto objektom bol aj *Suzanne* objekt exportovaný z Blenderu. Ak by bol zlý smer načítavania vrcholov trojuholníkov, tak po vypnutí *backface culling* by sa mi niektoré trojuholníky objektu nevykreslili. Tento problém vidíme na obrázku 16.



Obrázok 16: Chyba vznikajúca na objekte Suzanne načítanom z .obj súboru , rozlíšenie: 304x205 a 306x206, počet odrazov lúčov: 2, materiál: difúzny, vyhladzovanie: 94

6. Záver

V nasledujúcej časti zhodnotím dosiahnuté ciele. Popíšem hlavné výhody mojej modifikácie kódu a zosumarizujem moju prácu. Navrhnem aj prácu ktorá by mohla byť implementovaná do tohto algoritmu do budúcna.

6.1 Dosiahnuté ciele

Cieľom našej práce bolo vytvoriť vykresľovanie procedurálnych 3D modelov pomocou metódy path-tracing vo WebGL a GLSL v reálnom čase. Podarilo sa mi modifikovať už existujúci path-tracing algoritmus od Evana Wallacea v nasledovných bodoch.

- zmenil som veľkosť vykresľovacieho okna pre zvýšenie kvality výsledného obrazu.
- rozšíril som funkcionalitu o pohyb kamery v scéne
- zefektívnil som Path-tracing algoritmus.
- nastavil som vykresľovanie len v prípade zmeny pohľadu kamery, do momentu, kým nie je výsledný obraz dostačujúcej kvality, čím som odľahčil vyrovnávaciu pamäť.
- rozšíril som počet objektov v scéne o ďalšie 4 objekty.
- do scény je možné importovať modely v .obj formáte.
- implementoval som spôsob ako načítavať do scény viac objektov, ako dovoľovala Wallaceová implementácia.

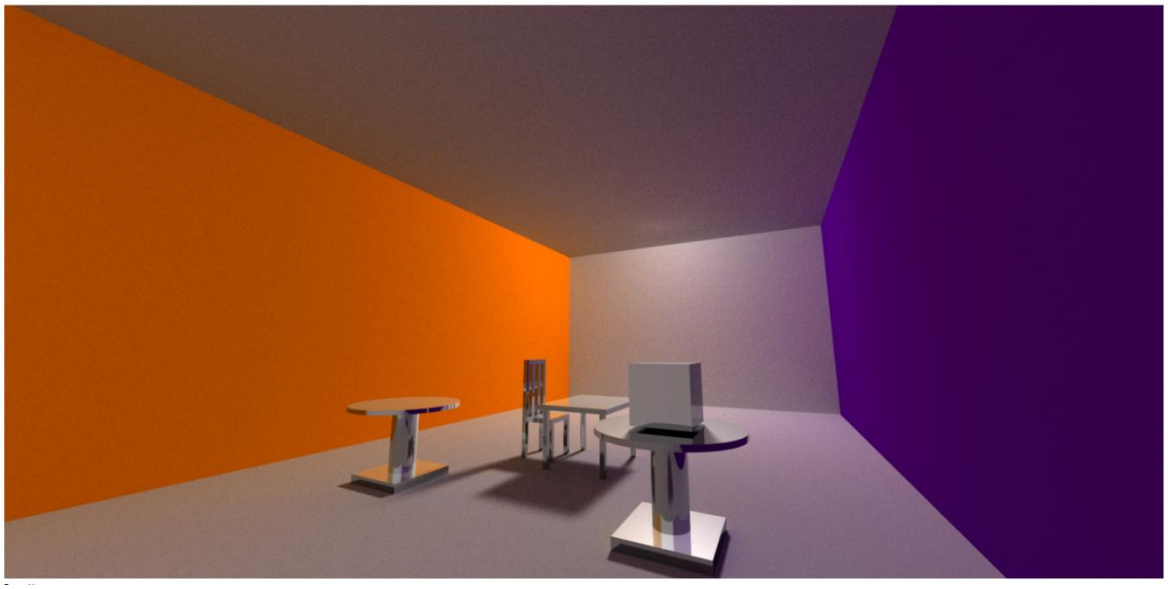
6.2 Zhodnotenie a Návrh ďalšej práce

V Bakalárskej práci sa mi podarilo vizualizovať 3D scény s dvomi typmi objektov. .Jedným typom boli 4 objekty ktoré som popísal pomocou ich analytických rovníc. Ďruhým typom objektov sú objekty importované z .obj súboru. Ďalšia práca, venujúca sa tejto aplikácií, by mohla byť zameraná na implementáciu nejakej dátovej štruktúry (KDstrom, Octree) kvôli urýchleniu výpočtov algoritmu sledovania ciest vo webovom prehliadači.

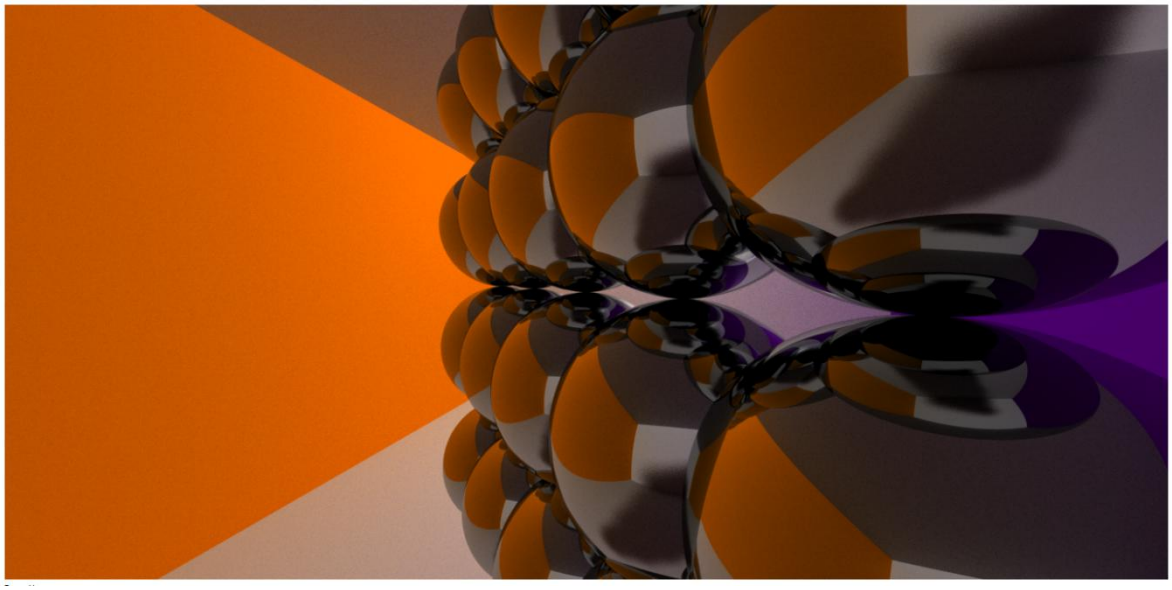
6.3 Výsledné obrázky

V tejto časti nájdete výsledné obrázky s informáciou o parametroch algoritmu sledovania ciest.

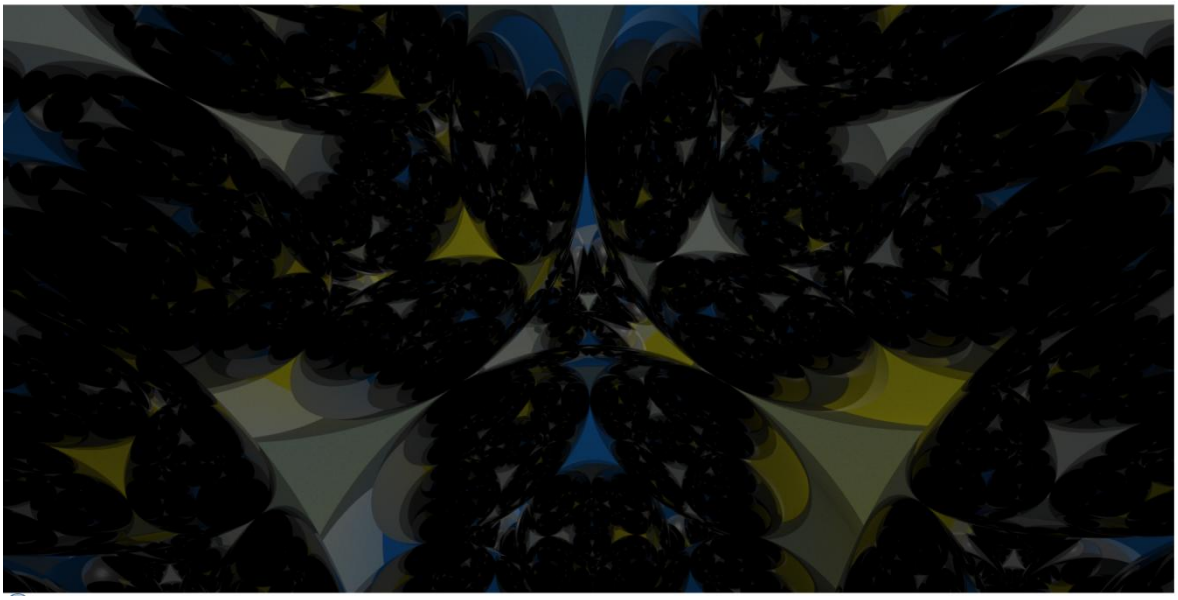
obrázok 17: Stôl, stolička a podstavce, rozlíšenie: 1438x719px, počet odrazov lúčov: 3, materiál: difúzny, vyhladzovanie: 94



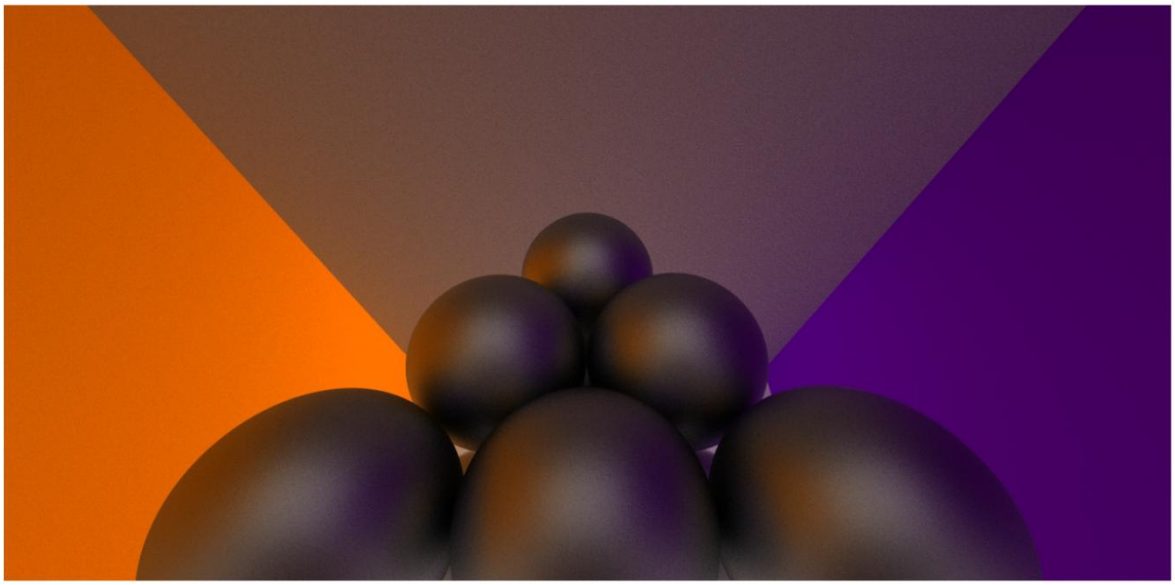
obrázok 18: Stôl, stolička a podstavce, rozlíšenie: 1445x719px, počet odrazov lúčov: 3, materiál: zrkadlo, vyhladzovanie: 94



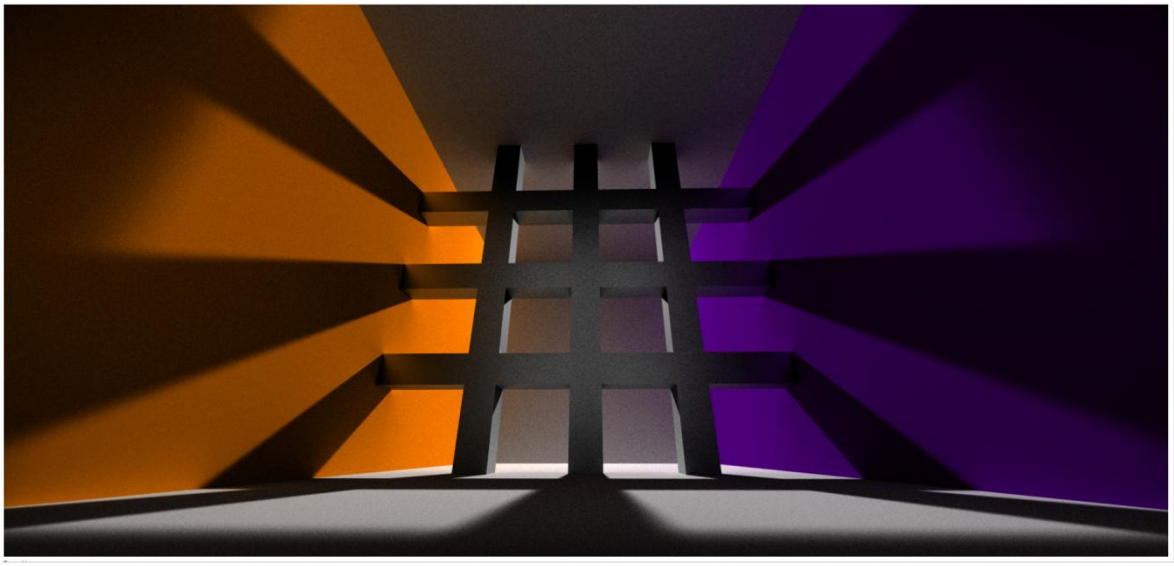
obrázok 19: Stena z gúľ,: 1445x719px, počet odrazov lučov:5,
materiál: zrkadlo, vyhladzovanie: 128



obrázok 20: Zrkadlové gule, rozlíšenie:1443x719px, počet odrazov lúčov: 5,
materiál: zrkadlo, vyhladzovanie: 128



obrázok 21: Matné gule, rozlíšenie:1443x719px, počet odrazov lúčov: 4,
materiál: lesklý, vyhladzovanie: 128



obrázok 22: Mreža rozlíšenie:1443x719px, počet odrazov lúčov: 3,
materiál: difúzny, vyhladzovanie: 94

Bibliografia:

[KAJI86] - J.T.Kajiya. The Rendering Equation. Na konferencí SIGGRAPH '86 Conference Proceedings, str. 143-150, August 1986

[ZAR10] - Moderní počítačová grafika, Jiří Žára, Bedřich Beneš, Jiří Sochor, Peter Felkel, 2010, str.413

[PAR12] Tony Parisi. WebGL: Up and Running, 2012, str. 10,

[CANT12] Diego Cantor, Bradon Jones, WebGL Beginner's Guide, Packt Publishing, 2012

[LAF93] Eric P. Lafortune, Yves D. Willems, Bi-directional Path Tracing, Department of Computing Science Katholieke Universiteit Leuven, Belgium

[BIK13] Jacco Bikker and Jeroen van Schijndel Research Article: A Path Tracer for Real/Time Games, str.2, ADE/IGAD, NHTV Breda University of Applied Sciences, The Netherlands

[PHAR10] Matt Pharr, Greg Humphreys, Physically Based Rendering - From theory to implementation (Second Edition), 2010,

[GMAPS] odkaz na stránku: https://www.google.com/maps , 25.5.2016.

[ZGOTE] odkaz na stránku: https://zygotebody.com/ , 25.5.2016.

[WEBGL] odkaz na stránku: http://www.madebyevan.com/webgl-path-tracing/index.html, 25.5.2016.

[WOL] odkaz na stránku: https://github.com/frenchtoast747/webgl-obj-loader , 25.5.2016.

[NIJJ] odkaz na stránku http://reindernijhoff.net/2015/04/realtime-webgl-path-tracer/ , 25.5.2016.

[SCULPT] odkaz: http://stephaneginier.com/sculptgl/ , 25.5.2016.

[CLAR] odkaz: https://clara.io/ , 25.5.2016.

[3Dtin] odkaz: http://www.3dtin.com/ , 25.5.2016.

[THREE] odkaz: http://blackjk3.github.io/threefab/ , 25.5.2016.

Prílohy:

1.) CD ktoré obsahuje:

- Zdrojové kódy
- Aplikáciu
- Dokument Bakalárska práca vo formáte .pdf
- README.txt súbor v ktorom je popísaný spôsob ako používať aplikáciu
- Výsledné obrázky v plnom rozlíšení.