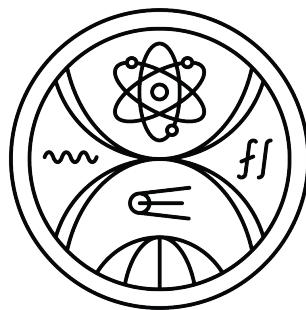


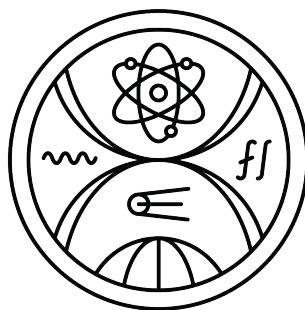
UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



**ŠKÁLOVATEĽNÝ PROCEDURÁLNY
ALGORITMUS PRE TRBLIETKY V REÁLNOM
ČASE**

Diplomová práca

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



ŠKÁLOVATEĽNÝ PROCEDURÁLNY
ALGORITMUS PRE TRBLIETKY V REÁLNOM
ČASE

Diplomová práca

Študijný program: Aplikovaná informatika
Študijný odbor: Informatika
Školiace pracovisko: Katedra aplikovanej informatiky
Školiteľ: prof. RNDr. Roman Ďuríkovič, PhD.



ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Róbert Kica
Študijný program:	aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	slovenský
Sekundárny jazyk:	anglický
Názov:	Škálovateľný procedurálny algoritmus pre trblietky v reálnom čase <i>Scalable procedural algorithm for glinty surface rendering in real-time</i>
Anotácia:	Unity má robustné materiály a shadery, Shadery v Unity (https://docs.unity3d.com/Manual/shader-built-in.html) Zamerali by sme sa na High Definition Rendering Pipeline - HDRP používa Lit, Layered Lit, Unlit a decal shadery. Existuje viacero verzií HDRP. Zameriame sa na verziu 12.1, ktorá je v stabilnej LTS verzii Unity. Existujú materiály s komplexnou mikroštruktúrou, ktoré pod silným svetlom produkujú mnoho drobných odrazov. Takéto trblietavé materiály, sú náročné na reprezentáciu a vykreslovanie nielen v reálnom čase ale aj pri offline vykreslovaní.
Ciel:	Použitím real-time enginu Unity vytvorte materiál s trblietkami. Popíšte použité metódy a porovnajte ich z hľadiska vizuálneho a výkonnostného. Porovnajte ich výhody a nevýhody. Implementujte algoritmy vykreslovania trblietavých materiálov. Navrhnite vlastnú metódu vykreslovania trblietavých materiálov. Porovnajte použité metódy.
Literatúra:	Recent advances in glinty appearance rendering [Junqiu Zhu] hlavne posledná sekcia real-time Modeling Aventurescent Gems with Procedural Textures [Andrea Weidlich] Procedural Generation of Natural Variations in Materials and Surfaces [JOHAN NILSSON] strany 30-49 Real-Time Microstructure Rendering with MIP-mapped Normal Map Samples[Haowen Tan] Junqiu Zhu, Sizhe Zhao, Yanning Xu, Xiangxu Meng, Lu Wang, and Ling-Qi Yan. Recent advances in glinty appearance rendering. Computational Visual Media, 8(4):535–552, 2022. Xavier Chermain, Basile Sauvage, J-M Dischler, and Carsten Dachsbacher. Procedural physically based brdf for real-time rendering of glints. In Computer Graphics Forum, volume 39, pages 243–253. Wiley Online Library, 2020. Tobias Zirr and Anton S Kaplanyan. Real-time rendering of procedural multiscale materials. In Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and games, pages 139–148, 2016



Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Thomas Deliot and Laurent Belcour. Real-Time Rendering of Glinty Appearances using Distributed Binomial Laws on Anisotropic Grids. Computer Graphics Forum, 2023.

Kľúčové

slová: vykreslovanie v reálnom čase, trblietky, Unity 3D

Vedúci: prof. RNDr. Roman Ďuríkovič, PhD.

Katedra: FMFI,KAI - Katedra aplikovanej informatiky

Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 24.10.2022

Dátum schválenia: 24.10.2022

prof. RNDr. Roman Ďuríkovič, PhD.

garant študijného programu

.....
študent

.....
vedúci práce

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím citovaných zdrojov.

Bratislava, 2024

.....
Bc. Róbert Kica

Pod'akovanie

Ďakujem pánovi profesorovi RNDr. Romanovi Ďuríkovičovi, PhD. za odbornú pomoc pri tvorbe tejto práce. Moja vdaka patrí aj mojej rodine a priateľom, ktorí sú mojou veľkou oporou.

Abstrakt

Vizualizácia materiálov sa používa v mnohých odvetviach ako architektúra, interiérový dizajn ale aj vo filmovom a videohernom priemysle. Mnohé materiály majú zložitú mikroštruktúru. Ak ju vieme verne vykresliť, dodáva scéne väčší realizmus. Existuje mnoho techník na dosiahnutie trblietavého vzhľadu pri neinteraktívnych aplikáciách. V tejto práci sa zaoberáme metódami vykresľovania trblietavých povrchov na GPU v reálnom čase. Tieto metódy sú často zjednodušené aby sa stihli vykonať v jednotkách milisekúnd. Implementovali sme niektoré existujúce metódy a navrhli sme vlastnú metódu na vykresľovanie trblietok v reálnom čase. Implementácia je v programe Unity 2022.1 v kanáli HDRP verzie 13.1 s použitím HLSL shader programov. Porovnali sme metódy a vytvorili aplikáciu kde sa dajú metódy vedľa seba porovnávať. Výsledný shader je možné stiahnuť a vložiť do nového projektu v Unity.

Kľúčové slová: vykresľovanie v reálnom čase, trblietky, Unity 3D

Abstract

Visualisation of materials is used in many industries such as architecture, interior design as well as in the film and video game industry. Many materials have complex microstructures. If we can render it faithfully, it adds more realism to the scene. There are many techniques to achieve a glinty appearance in non-interactive applications. In this work, we discuss methods for rendering glinty surfaces on GPUs in real-time. These methods are often simplified to be able to execute in units of milliseconds. We have implemented some existing methods and proposed our own method for real-time glinty surface rendering. The implementation is in Unity 2022.1 in the HDRP version 13.1 channel using HLSL shader programs. We have compared the methods and created an application where the methods can be compared side by side. The resulting shader can be downloaded and added into a new project in Unity.

Keywords: real-time rendering, glints, Unity 3D

Obsah

Zoznam príloh	xviii
1 Úvod	1
1.1 Vykresľovanie	1
1.2 Vykresľovanie v interaktívnych aplikáciách	3
1.3 Offline vykresľovanie	3
1.4 Vykresľovanie v reálnom čase	3
1.4.1 Vykresľovanie vo videohráč	4
1.5 Materiály a shadery	4
1.5.1 Shader program	5
1.5.2 Materiál	5
1.5.3 Grafický softvér na vykresľovanie	6
1.5.4 Vykresľovací kanál	7
1.5.5 Osvetlenie	8
1.5.6 Vykresľovanie v Unity	11
2 Súčasný stav vykresľovania trblietavých materiálov	13
2.1 Správanie sa svetla na povrchu materiálu	13
2.2 Trblietky, trblietavý vzhľad	14
2.3 Problém vykresľovania trblietok	15
2.4 Mikroštruktúra trblietavých materiálov, ich reprezentácia	16
2.5 Explicitná reprezentácia mikroštruktúry	16
2.6 Vykresľovanie trblietavých materiálov	17
2.7 Offline, neinteraktívne metódy	17
2.8 Metódy v reálnom čase	18
2.8.1 Procedurálne generované trblietky	18
2.8.2 Real-time Rendering of Procedural Multiscale Materials	20
2.8.3 Procedural Physically based BRDF for Real-Time Rendering of Glints	22
2.8.4 Real-Time Rendering of Glinty Appearances using Distributed Binomial Laws on Anisotropic Grids	25

3 Implementácia	30
3.1 Vývojárske prostredie	30
3.2 Výber grafického softvéru	30
3.2.1 Rozšírenie HDRP kanálu	30
3.2.2 Modifikácia lokálneho HDRP	31
3.2.3 Tvorba vlastného shaderu	33
3.2.4 Tvorba vlastného editora na parametre trblietok	36
3.3 Implementácia exitujúcich metód vykresľovania trblietok	39
3.4 Návrh a implementácia vlastnej metódy vykresľovania trblietok	39
3.4.1 Návrh	39
3.4.2 Implementácia	40
3.5 Tvorba aplikácie na porovnávanie metód	42
3.6 Stiahnutieľné demo a shader	43
3.7 Práca s Unity HDRP	45
4 Výskum a výsledky	47
4.1 Metodika evaluácie	47
4.1.1 Metodika evaluácie výkonu	47
4.2 Metodika evaluácie	48
4.3 Evaluácia scenárov	50
4.3.1 1. Scenár	50
4.3.2 2. Scenár	52
4.3.3 3. Scenár	53
4.4 Vyhodnotenie	54
4.4.1 Metóda Chermain et al.	54
4.4.2 Metóda Deliot et al.	56
4.4.3 Metóda Zirr et al.	57
4.4.4 Metóda Wang et al.	59
4.4.5 Naša metóda	61
4.4.6 Zhrnutie	62
4.4.7 Ďalšia práca	62
Príloha A - Zdrojové kódy	68
Príloha B - Návod na ovládanie aplikácie	72
Príloha C - Návod na vytvorenie Unity projektu s CustomLit Shaderom v HDRP	75
Príloha D - Použité parametre pri evaluácii	79

Príloha E - Zoznam použitých assetov	81
Príloha F - Úryvok triedy GlintsMethodUIBlock	83

Zoznam obrázkov

1.1	Príklad materiálu v Unity.	2
1.2	Pokrok v grafickom nasvietení naprieč generáciami hardvéru v jednej hre.[26]	5
1.3	Zobrazenie grafického vykresľovacieho kanálu. [13]	8
1.4	Vizuálna reprezentácia renderovacej funkcie. [15]	9
1.5	Pohľad na povrch zblízka, pri štandardnej NDF a NDF vhodnej pre trblietky.	10
2.1	Maki-e technika lakovania. [22]	15
2.2	Ukážka dôležitosti mikroštruktúry. Materiály s jednoduchým hladkým povrhom naľavo. V obrázku napravo bola zachovaná mikroštruktúra povrchov.	16
2.3	Na obrázku naľavo vidíme 2 úrovne mriežky vybraté na výpočet počtu mikrodetailov. Na obrázku napravo je stopa pixelu, podľa ktorého sa vyberú 2 úrovne mriežky. [47]	21
2.4	Produkt 2 1D marginálnych distribúcií. Ukladá sa iba pozitívna časť (v modrých rámčekoch). [19]	23
2.5	Úrovne l SDF lóbov, kde posledná je klasická priemerná NDF. [19] . . .	24
2.6	Slovník marginálnych distribúcií na rôznych škálach, ktoré konvergujú k cieľovej P_{target} - napr. Beckmannovej NDF. [19]	25
2.7	Ilustrácia stopy pixelu v mriežke. Na obrázku hore vidíme LOD mriežku, kde máme iba škálu. Na obrázku v strede vidíme pridanie pomerov. Spodný obrázok ilustruje pridanie orientácií. [18]	27
2.8	Ilustrácia výslednej 3D mriežky. Bod označený červenou bodkou sa nachádza niekde medzi 2 úrovňami škály, 3 orientáciami a 2 pomermi. Preto potrebujeme $2 * 5$ bodov na vypočítanie jedného texelu- jednej bunky mriežky [4]	28
3.1	Jednotlivé priechody pre HDRP kanál s vyznačenými vstupnými bodmi.	31

3.2	Na obrázku naľavo vidíme štruktúru našej implementácie trblietok s upravenými HDRP shader súbormi. Napravo je štruktúra celého balíčka HDRP. Upravovali sme malú časť tohto balíčka, nepotrebovali sme ho kopírovať celý	33
3.3	Podstatná časť kanálu HDRP vykresľovania pre Lit.shader ktorú budeme meniť.	35
3.4	Vlastný editor maeriálu s trblietkami <i>CustomLitMaterialInspector</i> ked' sme pridali parametre trblietok s dedením z LitGUI (naľavo), pridaním vlastného bloku s dedením z LightingShaderGraphGUI (v strede) a opravným pridaním stratených parametrov (vpravo).	38
3.5	UI rozhranie pre samostatnú aplikáciu, kde vieme porovnávať jednotlivé metódy a zároveň ovládať ich parametre.	44
4.1	Ukážka priebehu evaluácie.	49
4.2	Modely použité pri evaluácii scanára 1 a 2.	51
4.3	Vľavo vidíme graf času vykresľovania pre Lit Shader (vľavo) a Custom-Lit Shadera (vpravo).	53
4.4	Vizuál Chermain et al. metódy.	56
4.5	Vizuál Deliot et al. metódy.	57
4.6	Vizuál Zirr et al. metódy.	58
4.7	Vizuál Wang et al. metódy.	60
4.8	Rôzne parametre pri Wang metóde	60
4.9	Vizuál našej metódy.	62

Zoznam tabuliek

4.1	Porovnanie času vykresľovania (v milisekundách) pre rôzne typy modelov a trblietavých metód pre scenár č. 1.	50
4.2	Porovnanie času vykresľovania (v milisekundách) pre rôzne typy modelov a trblietavých metód pre scenár č. 2.	52
4.3	Snímková frekvencia jednotlivých metód v ms pre scenár 3.	53
4.4	Snímková frekvencia jednotlivých metód v FPS pre scenár 3.	54
4.5	Porovnanie času vykresľovania (v milisekundách) našej metódy pri použití šumu vs bez šumu.	54

Zoznam útržkov kódu

2.1	Shopf trblietky	19
3.1	Vlastný editor 1/2	36
3.2	Pseudokód vlastnej metódy	40
3.3	Ward Anizotrpný odlesk	41

Zoznam príloh

- Príloha A - Zdrojové kódy
- Príloha B - Návod na ovládanie aplikácie
- Príloha C - Návod na vytvorenie Unity projektu s CustomLit Shaderom v HDRP
- Príloha D - Použité parametre pri evaluácii
- Príloha E - Zoznam použitých assetov
- Príloha F - Úryvok triedy GlintsMethodUIBlock

Terminológia

Termíny a skratky

- **(APV) Adaptívne svetelné sféry**

Anglicky Adaptive Probe Volumes, je systém globálnej iluminácie založený na zväzkoch sfér, umiestnených v scéne, ktoré vytvárajú predpočítané nepriame osvetlenie.

- **(API) rozhranie pre programovanie aplikácií**

Anglicky Application Programming Interface je súbor pravidiel a nástrojov, ktoré umožňujú komunikáciu rôznych softvérových aplikácií medzi sebou. Definuje najmä metódy a formáty údajov, ktoré môžu aplikácie použiť na výmenu informácií. V grafickej sfére poznáme mnoho API ako napríklad OpenGL, DirectX, Vulkan, Metal, WebGL ale aj OpenCL a CUDA.

- **(BRDF) Obojsmerná funkcia distribúcie odrazu**

Anglicky Bidirectional Reflectance Distribution Function je funkcia ktorá popisuje odraz svetla na povrchu.

- **(FPS) Snímky za sekundu**

Počet snímok za sekundu je miera toho, koľko jednotlivých snímok sa spracuje azobrazí v aplikácii každú sekundu. Predstavuje plynulosť pohybu obsahu na obrazovke. Vyššie hodnoty majú vo všeobecnosti za následok plynulejší pohyb.

- **(HDRP) Vykresľovací kanál s vysokou kvalitou**

Anglicky High Definition Rendering Pipeline je vykresľovací kanál dostupný ako prídavný modul pre Unity, ktorý ponúka vysoko kvalitnú vizuálnu vernosť v reálnom čase.

- **(LOD) úroveň detailov**

Anglicky Level Of Detail, je názov štruktúry, v ktorej každá vyššia úroveň danej štruktúry s istou úrovňou detailov obsahuje ešte viac detailov (respektíve aj naopak, ďalšia úroveň obsahuje menej detailov). Používa sa pri vykresľovaní modelov, kde objekty blízko pri kamere (zaberajú veľkú plochu snímky) majú najvyššiu úroveň detailov a objekty ďaleko od kamery môžu mať menej bodov,

jednoduchší model objektu. Pri trblietkách sa využíva na reprezentáciu a vzorkovanie distribúcie mikroplôšok definovaných na rôznych škálach.

- **(NDF) Normálna distribučná funkcia**

NDF je distribučná funkcia ktorá popisuje koncentráciu plôšok, ktoré sú orientované tak, aby odrážali svetlo smerom do kamery.

- **(PBR) Fyzikálne založené vykresľovanie**

Anglicky Physically Based Rendering. Je to proces ako reprezentovať objekty podľa presných fyzikálnych princípov.

- **(SDF) Sklonová distribučná funkcia**

SDF (anglicky Slope Distribution Function) je distribučná funkcia, podobne ako DNF, ale popisuje distribúciu sklonov, oproti priestoru normálových vektorov alebo sférických súradníc, ktoré sa bežne používajú pri popise povrchov v 3D.

Kapitola 1

Úvod

Materiály v reálnom živote sú komplexné. Čo najvernejšie reprezentovať a zobrazovať komplexné materiály ostáva predmetom skúmania. Materiály, ktoré majú trblietavý vzhľad sú predmetom nášho skúmania. Tieto materiály obsahujú na svojom povrchu drobné plôšky, trblietky, ktoré majú rôznu orientáciu. Pri vhodnom uhlе pozorovania tieto plôšky odrážajú svetlo späť k pozorovateľovi. Pri zmene uhla, miesta pozorovania, alebo pozície svetla rôzne plôšky zhasínajú, iné svietia. Tým vzniká efekt trblietania. Príklady takýchto materiálov sú automobilový metalický lak, trblietky na oblečení a obuvi. Trblietavý efekt môžeme pozorovať aj na prírodných materiáloch ako sú horniny, sneh a ľad.

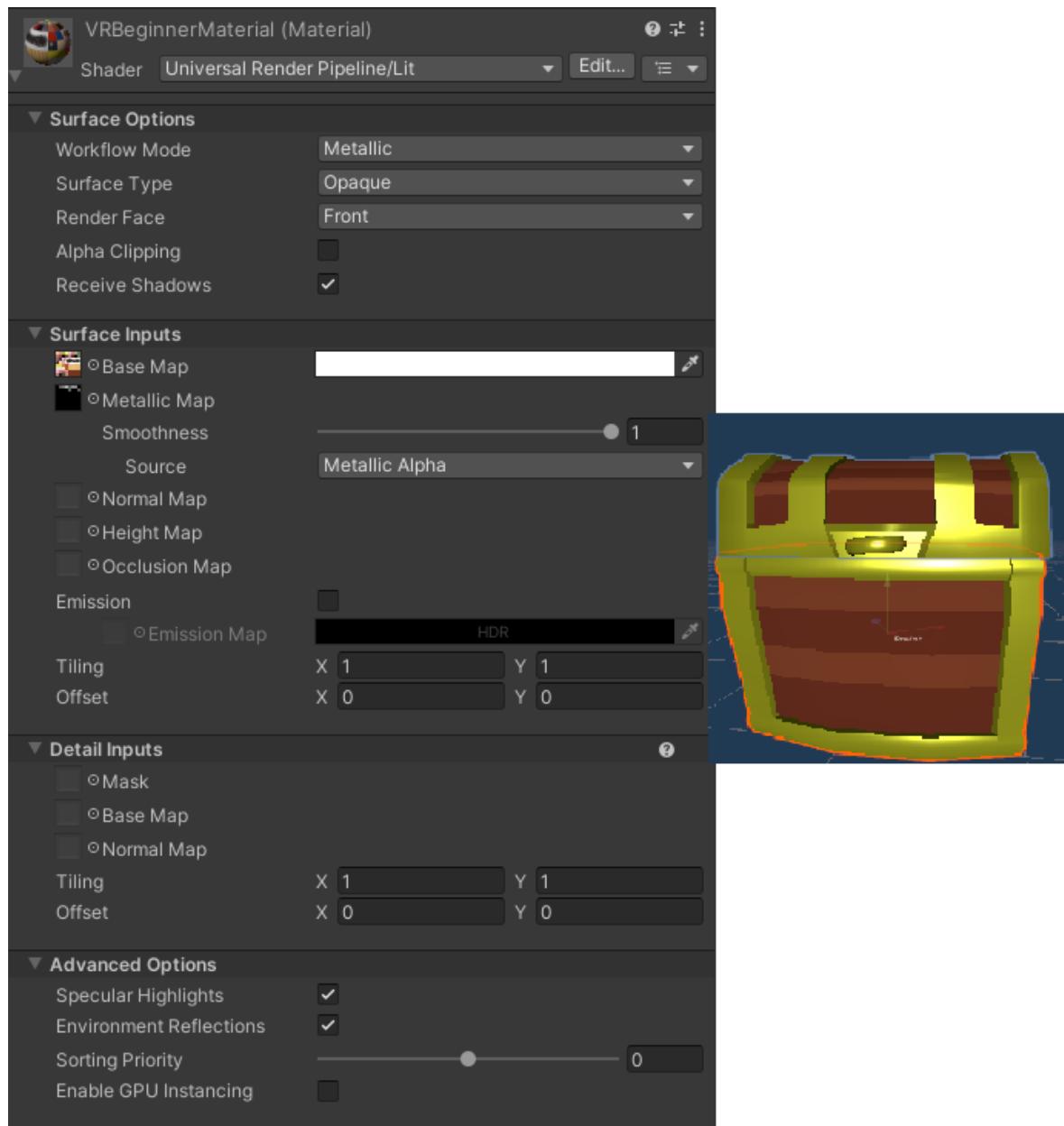
Trblietavé materiály predstavujú najmä dva problémy. Musíme ich najprv vhodne reprezentovať a potom vhodne danú reprezentáciu vykresliť. V prvých kapitolách sa pozrieme na existujúce riešenia. Pozrieme sa na spôsoby reprezentácie mikroštruktúry a jej vykreslovanie v offline a reálnom čase. Analyzujeme ich klady a zápory.

V kapitole Implementácia navrhнемe a implementujeme vlastnú metódu vykreslovania trblietok v reálnom čase v grafickom softvéri Unity. Implementujeme aj niektoré existujúce riešenia na porovnanie. V poslednej kapitole analyzujeme dosiahnuté výsledky. Porovnáme navrhnutú metódu s existujúcimi riešeniami.

1.1 Vykresľovanie

S vykreslovaním sa najčastejšie stretávame pri používaní počítačov a iných elektronických zariadení. Tento proces patrí k základným funkciám moderných počítačov. Na interakciu s takýmito zariadeniami využívame GUI, grafické používateľské rozhranie.

Vykresľovanie vyžívame aj na zobrazenie multimediálneho obsahu ako sú fotky, videá a interaktívne aplikácie. Zobrazovanie takého obsahu sa vykonáva na zobrazovačom zariadení, zväčša na plochom monitore, mobilnej obrazovke alebo aj displeji v náhlavnej súprave VR, na plátne pomocou projektora, alebo v prípade fotky, vytlačený



Obr. 1.1: Príklad materiálu v Unity.

na papieri. Na zobrazovacom zariadení sa zobrazujú snímky, rasterizované, zložené z veľmi drobných pixelov. Fotka je jedna snímka, video je sekvencia snímok, spravidla s 24 snímkami za sekundu. Interaktívne aplikácie ako napr. architektonický softvér alebo videohry používajú väčšinou vyššiu snímkovaciu frekvenciu na zlepšenie plynulosťi.

1.2 Vykresľovanie v interaktívnych aplikáciách

Vykresľovanie v interaktívnych aplikáciách zväčša vyžaduje hardvérovo urýchľované grafické procesory. Využití je naozaj mnoho, v zábavnom biznise sú to animované filmy, VFX efekty a videohry. Taktiež sú využívané na architektonickú vizualizáciu, vizualizáciu produktov na internetových obchodoch, fyzikálne simulácie. GPU môže byť využité aj na strojové učenie, hlboké učenie a veľké jazykové modely. Animované filmy a videohry boli pôvodným zdrojom najväčších inovácií vo vývoji GPU, hlavne v oblasti rasterizácie. Neskôr sa GPU začali používať na všeobecnejšie, paralelizované výpočty, tzv. GPGPU.

1.3 Offline vykresľovanie

Animované filmy vznikajú vytváraním 3D scény s virtuálnymi modelmi a materiálmi ktoré sú potom vykresľované so zadanou snímkovacou frekvenciou. Takto vytvorené snímky predstavujú tzv. offline rendering, teda sú vykreslené v neinteraktívnom čase, čo predstavuje spravidla niekoľko sekúnd na vykreslenie jednej snímky.

Pri offline vykresľovaní je kladený najväčší dôraz na vizuálnu kvalitu. Algoritmy použité na vykresľovanie sú často náročné na výpočet. Snímky sa spravidla renderujú/vykresľujú na rendering farmách, zhlukoch vysoko výkonných počítačov, pričom vykresliť jeden snímok môže trvať minúty.

1.4 Vykresľovanie v reálnom čase

Oproti filmom je pri interaktívnych aplikáciách kladený veľký dôraz na rýchlosť vykresľovania. Čím je vyššia snímkovacia frekvencia, tým je odozva rýchlejšia a interakcia s aplikáciou plynulejšia. Umožňuje to meniť uhol kamery, nasvietenie respektívne meniť modely dynamicky. Táto interaktivita nám umožňuje vizualizovať interiér klientovi, ktorý získa lepší obraz o navrhovanom riešení a rýchlo môžeme napraviť prípadné pripomienky.

1.4.1 Vykresľovanie vo videohráčoch

Pri videohráčoch je snímkovacia frekvencia ešte dôležitejšia. Keďže v nich často vykonávame prudké pohyby, tak snímkovanie musí byť stabilné a aj . Minimálna snímkovacia frekvencia je 30fps alebo 60fps, u niektorých typoch hier aj viac. Vývojári sa snažia vytvoriť vizuálne čo najpôsobivejšie diela. Pri vývoji ale treba myslieť na hardvér na akom to naši zákazníci spustia. V minulosti hardvérový výkon veľmi limitoval vývojárov. Tí museli vytvoriť mnoho metód, ako danú scénu verne zobraziť. Nepoužívali fyzikálne založené osvetľovacie metódy, preto často môžeme hovoriť o trikoch. Tieto metódy komplikovali často aj proces vývoja, keďže pri každej zmene sa muselo opakovať predpočítavanie.

Prostredie bolo málo detailné, zväčša statické a svetlo sa správalo veľmi nerealistiky, bolo len empirické, napríklad Phongov osvetľovací model.

Postupne sa vyvinuli spôsoby, triky ako získať čo najlepší vizuál s akceptovateľným výkonom. Tieto metódy predpočítavali osvetlenie, alebo časti osvetľovacích algoritmov. Nevýhodu majú ale v tom, že predpočítavanie vieme robiť len na primárne statické scény. Dynamické objekty nie sú brané do úvahy a vyčnievajú v prostredí, pretože majú zjednodušený výpočet osvetlenia. Tiež bolo použitých veľa techník ako napríklad techniky pracujúce na snímku, ako napríklad odrazy v obrazovom priestore (SSR) alebo ambientná oklúzia v obrazovom priestore (SSAO).

Predpočítavanie si vyžaduje pamäťový priestor a limituje dynamickosť scény/ prostredia. Vývojári musia pri častých zmenách v projekte tento proces predpočítavania opakovať.

Hardvér sa ale stále vyvíja, čo umožňuje tvorbu čoraz zložitejších, dynamickejších, scén. To vedie aj k použitiu výpočtovo náročnejších techník, ktoré predtým boli príliš náročné. Preferujú sa techniky, ktoré potrebujú čo najmenej predpočítavania, najmä pre rýchlejšie iterovanie a teda menšie náklady na vývoj.

S príchodom najnovšieho hardvéru môžeme vidieť, že je už možné použiť aj techniky, ktoré sú už takmer úplne realistické, pričom umožňujú aj dynamické zmeny. Dá sa teda meniť čas, osvetlenie a netreba nič predpočítavať. Príkladom takýchto metód sú hardvérovou akcelerované sledovanie lúča(ang. ray tracing), resp. sledovanie cesty(ang. path tracing). Technológia Lumen v Unreal Engine 5, má aj softvérovú implementáciu sledovania lúča.

1.5 Materiály a shadery

Objekty v reálnom svetle majú rôzne fyzikálne vlastnosti, ktoré vplývajú na jeho výsledný vzhľad. Reprezentáciou tohto správania vo virtuálnom svete sú materiály, ktoré majú zadaný nejaký shader.



Obr. 1.2: Pokrok v grafickom nasvietení naprieč generáciami hardvéru v jednej hre.[26]

1.5.1 Shader program

Shader program je program bežiaci na GPU, ktorý sa snaží z parametrov zadaných v materiáli vypočítať výslednú farbu. Shadery sa líšia v závislosti od použitej vykresľovacej metódy.

1.5.2 Materiál

Objekt, zväčša polygónový model, má priradený určitý materiál. Tento materiál predstavuje vstupy shaderu.

V interaktívnych aplikáciách sa používa tzv. fyzikálne založený rendering. PBR materiály sa snažia čo najvernejšie opísati, ako výsledný povrch bude vyzerať v scéne. Niektoré materiály sú nepriehľadné, iné priehľadné resp. priesvitné. Môžu nastávať rôzne javy ako rozptyl a lom svetla, difúzia, odraz.

Na vzhľad môže vplývať v rôznej miere aj okolie. Súhrnný názov je globálna iluminácia. Existuje mnoho techník ako approximovať GI.

Materiály poznáme explicitne definované alebo procedurálne.

Explicitné materiály používajú UV mapovanie a textúry, uložené na disku ako obrázky. Polygonálna sieť je namapovaná pomocou UV súradníc a všetky textúry sú na túto sieť nanesené podľa tohto mapovania.

Procedurálne materiály sú, na druhej strane definované parametricky a až pri behu programu môžeme vytvoriť explicitnú textúru.

Materiály v reálnom živote sú veľmi komplexné a často potrebujeme viacero shaderov na zachytenie rôznych objektov. V tom prípade viac inklinujeme k procedurálnym

materiáлом, alebo čisto procedurálnym shaderom.

Mapy a textúry

Mapa definuje nejakú vlastnosť povrchu, napríklad farbu, drsnosť alebo nerovnosť. Takýchto máp poznáme veľa, líšia sa v závislosti od zvoleného grafického enginu, a jednotlivých shaderov. Najzákladnejšie mapy sú difúzna mapa, spekulárna/metalická, normálová, výšková mapa, alebo mapa drsnosti či mapa oklúzie. Tieto textúry sa nampujú na modely pomocou UV súradníc a predstavujú rôzne preddefinované vlastnosti materiálov.

Aby bol povrch objektu verne zachytený, potrebujeme veľké textúry na zachovanie malých detailov, alebo opakovane použiť malú textúru po celom povrchu ako dlaždicu.

V praxi je často dostačujúce použiť explicitné mapy na väčšinu povrchov. Tieto mapy sú diskrétné, máme teda určitý rozsah pozorovania, kde je vzorkovanie týchto textúr dostačujúce. Objekty v diaľke využívajú MIP mapovanú textúru a ak sa predpokladá, že objekt môžeme vidieť z veľkej blízkosti, vieme prepnúť na osobitnú detailnú mapu.

Pokiaľ ale chceme zachovať drobné detaily, ako sú napríklad škrabance alebo trblietky, tieto mapy začnú byť nevhodné. Nejakým spôsobom chceme zachovať alebo navodiť dojem úplnej mikroštruktúry.

1.5.3 Grafický softvér na vykreslovanie

Vykreslovanie môžeme vykonávať softvérovo na CPU ale aj použiť hardvérový akcelerátor grafický čip, GPU.

Mnoho vykresľovačov bolo pôvodne navrhnutých pre CPU, voláme ich softvérové vykresľovače. Moderný softvér využíva CPU aj GPU, podľa zvolenej metódy a problému.

Na hardvérovo akcelerované vykreslovanie boli vytvorené programovacie rozhrania (API), softvérové rámce alebo knižnice. Príkladom API je DirectX, OpenGL, Vulkan.

Komerčne dostupný softvér na vykreslovanie v neinteraktívnom čase je napríklad Chaos Corona renderer, Pixar RenderMan alebo Blender (Cycles a Eevee).

Vykreslovanie reálnom čase využíva najmä hardvérovo urýchlené vykreslovanie. Často je vykreslovanie len časť celej aplikácie, ako napríklad videohry. Preto vznikol softvér, ktorého neoddeliteľnou súčasťou je vykreslovanie, ale obsahuje aj fyzikálny softvér, správu audia, vstupných zariadení a mnoho iného. Medzi najpopulárnejšie komerčne dostupné softvéry patrí Unity Engine, Unreal Engine, Godot alebo CryEngine.

1.5.4 Vykresľovací kanál

Vykresľovací kanál je séria fáz alebo krokov, ktorými prechádzajú grafické údaje, aby sa vytvoril konečný obraz - snímka na obrazovke. Je to základný koncept v programovaní počítačovej grafiky. Je teda základným kameňom na tvorbu realistických a interaktívnych vizuálnych zážitkov v aplikáciách, ako sú videohry, animované filmy ale aj vedecké simulácie.

Kanál sa za posledné desaťročia výrazne rozšíril a skomplikoval. Vykresľovací kanál je základom grafických API. Existuje viacero grafických API, ktoré majú mierne rozdiely medzi sebou, ale vychádzajú z rovnakých princípov. Ukážeme zjednodušený proces vykresľovacieho kanála a stručný opis najdôležitejších krokov. Vykresľovací kanál je tiež znázornený na obrázku 1.3.

Aplikačný krok:

V tejto fáze aplikácia bežiaca na CPU posiela geometrické údaje scény a ďalšie relevantné informácie grafickému API. Tieto údaje zvyčajne zahŕňajú vrcholy objektov, textúry, materiály, informácie o osvetlení a vlastnosti kamery.

Krok spracovania geometrie:

Geometrické údaje ktoré boli prijaté z aplikačnej fázy sú transformované - aplikujeme transláciu, rotáciu, škálovanie a rôzne projekčné matice. Vykonáva sa aj orezanie zorného poľa aby sa vylúčili všetky objekty, ktoré nie sú viditeľné.

Krok rasterizácie:

Základné eometrické útvary ako trojuholníky alebo aj mnohouholníky sa konvertujú na fragmenty / pixely, ktoré sa vyfarbia a zobrazia na obrazovke. Pre každý fragment určí rasterizátor jeho polohu, hĺbku a ďalšie iné atribúty.

Vertex Shader:

Vertex shader je programovateľný krok vykresľovacieho kanálu, ktorý spracováva každý vrchol geometrie. Môžeme upravovať pozície vrcholov, počítať osvetlenie jednotlivých vrcholov, aplikovať transformácie a vykonávať ďalšie operácie.

Fragmentový /pixelový shader:

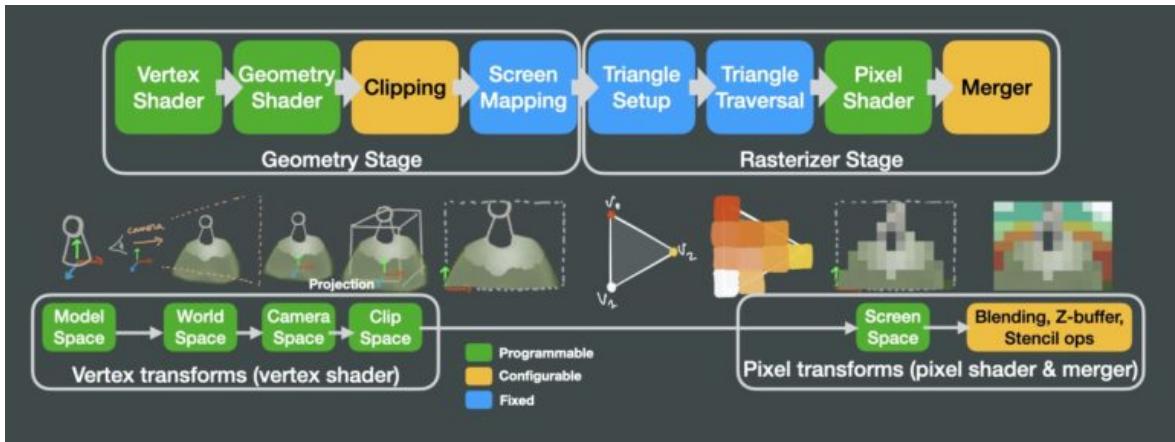
Fragmentový shader je ďalší programovateľný krok. Je zodpovedný za výpočet konečnej farby každého pixelu. Môžeme v ňom počítať osvetlenie, aplikovať textúry, vykonávať operácie zmiešavania a implementovať rôzne zaujímavé vizuálne efekty.

Krok rastrových operácií:

V tomto kroku sa vykonáva test hĺbky, test šablón, alfa zmiešavanie a ďalšie operácie na pixeli. Kombinuje výstup fragmentového shadera s obsahom vyrovnávacej pamäte snímky.

Krok zlúčenia výstupov:

V tejto záverečnej fáze sa do vyrovnávacej pamäte snímky zapíšu konečné farby pixelov.



Obr. 1.3: Zobrazenie grafického vykresľovacieho kanálu. [13]

Vykresľovací kanál je vysoko paralelizovaný a optimalizovaný na efektívne spracovanie veľkého množstva údajov. Grafický hardvér - grafické čipy (GPU) sú špeciálne navrhnuté na urýchlenie rôznych krokov vykresľovacieho kanálu, čo umožňuje vykresľovanie zložitých scén pri interaktívnej snímkovej frekvencii.

1.5.5 Osvetlenie

Poznáme viacero osvetľovacích modelov. Poznáme modely, ktoré aproximujú osvetlenie podľa jednoduchých pozorovaní a experimentov alebo fyzikálne založené modely.

Empirické modely osvetlenia

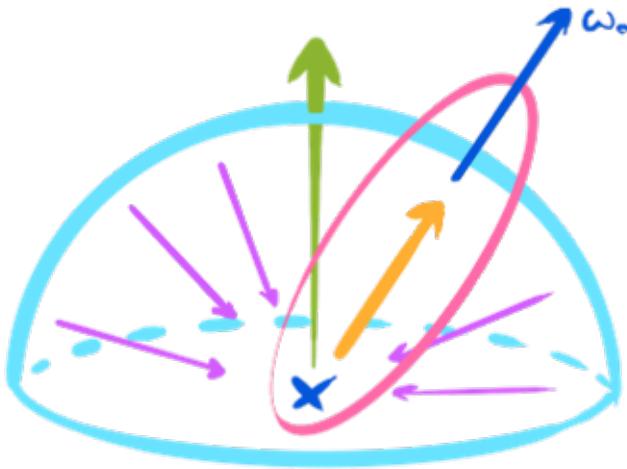
Phongov, respektíve viac využívaný Blinn-Phongov model je empirický model osvetlenia, ktorý zohľadňuje iba lokálnu ilumináciu. Je rýchly na výpočet, nie je fyzikálne presný. Matematicky je vyjadrený ako súčet vplyvu svetiel v scéne s ambientnou, difúznou a spekulárnoch zložkou. Svetlá môžu mať rôzne intenzity a farby. Materiály majú koeficienty odrazivosti, difúzie, pomer odrazu.

$$Blinn - Phong = c_p(c_a * l_a + \sum_{m \in svetla} (c_d * (L_m \cdot N) * I_{m,d} + c_s * (H \cdot N)^\alpha * I_{m,s})) \quad (1.1)$$

Fyzikálne modely osvetlenia

V posledných rokoch sa prešlo z empirických modelov na fyzikálne založené osvetlenie, pretože sú presnejšie a jednoduchšie na prácu. Taktiež hardvér sa vyvinul a nie sme ním tak obmedzovaní.

Poznáme mnoho modelov BSDF resp. BRDF a ich variácií. Variácie sú často vytvárané kombináciami prvkov pôvodných BRDF. Najrozšírenejšie sú modely založené na teórii mikroplôšok.



Obr. 1.4: Vizuálna reprezentácia renderovacej funkcie. [15]

Rovnica vykresľovania, BRDF

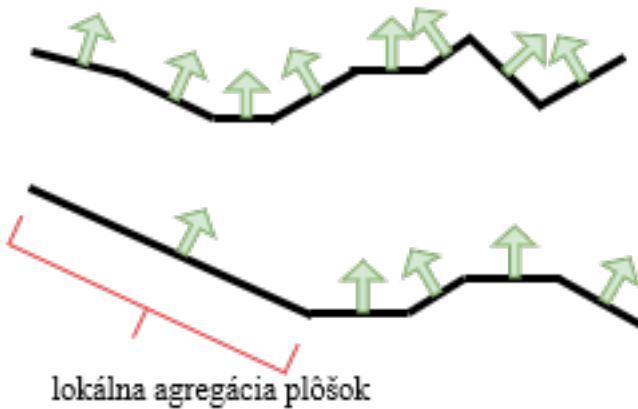
Rovnica vykresľovania je základná rovnica používaná pri modelovaní osvetlenia. Predstavuje teoretický model. Model navrhli David Immel et al. a James Kajiya v roku 1986 súčasne [23] [20].

$$L_{ref}(X, \omega_{ref}) = L_e(X, \omega_{ref}) + \int_{\Omega} f_r(X, \omega_{in} \rightarrow \omega_{out}) * L_{in}(X, \omega_{in}) * \cos\theta_{in} * d\omega_{in} \quad (1.2)$$

$L_{ref}(X, \omega_{ref})$ je žiarenie vychádzajúce z bodu X pod uhlom ω_{ref} . Aby sme našli svetlo smerom k divákovi z určitého bodu X , spočítame svetlo vyžarované z tohto bodu (tzv. emisiu) L_e plus integrál na jednotkovej hemisfére zo súčinu: svetla L_{in} prichádzajúceho zo smeru ω_{in} s geometrickým poklesom pod uhlom vyjadreným povrchovou normálou a smerom prichádzajúceho svetla θ_{in} (Lambertov odraz), vynásobeného interakciou svetla na povrchu $f_r(X, \omega_{in} \rightarrow \omega_{out})$ (túto funkciu nazývame BRDF). Integráciou v podstate dostaneme farbu lúča. L_{in} sa bežne počíta rekurzívne, tento proces nazývame sledovanie cesty, path tracing.

V praxi sa pri PBR sa používa najmä modifikovaná verzia vykresľovacej rovnice, tzv Reflectance equation, kde sa neberie sa do úvahy zložka emisie.

Na tomto teoretickom modeli vidíme, že na výpočet farby v bode X z pohľadu kamery je použité rekurzívne volanie výpočtu farby na ostatné body a svetlá viditeľné z bodu X . Tento výpočet je veľmi náročný. Algoritmy využívané v praxi aproximujú mnohé časti tejto rovnice, limitujú počet odrazov, vyberajú dôležité vzorky ktoré najviac vplývajú na výslednú farbu alebo využívajú odšumovače. Záleží to od využitia a



Obr. 1.5: Pohľad na povrch zblízka, pri štandardnej NDF a NDF vhodnej pre trblietky.

kapacity hardvéru, miery dosiahnutia čo najvernejšej grafiky verus miery nárokov na udržanie interaktívnych snímkov za sekundu.

Teória mikroplôšok

Vo fyzikáne založenom vykreslovaní sa makroskopický vzhľad modeluje stochasticky tak, že pomocou normálovej distribučnej funkcie (NDF) definujeme distribúciu orientácií mikroskopických plôšok.

Vo výpočte pomocou smeru pohľadu NDF vyjadruje koľko svetla sa odrazí smerom k pozorovateľovi. Takúto funkciu nazývame mikroplôšková BSDF. Jej vzorec vyzerá nasledovne:

$$f_r = \frac{F(\alpha) * D(H) * G(L, V, H)}{4|n \cdot l||n \cdot v|} \quad (1.3)$$

Rovnica 1.3 predstavuje Cook-Torrance BRDF, kde D je NDF, F je fresnel funkcia a G je geometrická funkcia. N je normála povrchu, l je smer svetla a v je smer pozorovania.

NDF funkcia D, predstavuje priemerné odrazené svetlo, ktorá je pre materiály s homogénym povrhom dostačujúca. Je modelovaná stochasticky a definuje makroskopický vzhľad povrchu. Rozloženie plôšok na povrhmu môžeme vidieť na obrázku 1.5, hore. Na vykreslenie materiálov s trblietkami potrebujeme upraviť túto funkciu NDF a spojiť ju s vhodnou reprezentáciou trblietok. Trblietka sa prejavuje silným odrazom svetla, čo je zapríčinené lokálnej agregáciou smerov týchto plôšok.

Funkcia BRDF

Bidirectional reflectance distribution function, BRDF, aproximuje, do akej miery každý jednotlivý svetelný lúč ω_{in} prispieva ku konečnému odrazenému svetlu matného povrhmu vzhľadom na vlastnosti materiálu [42]. Poznáme mnoho BRDF funkcií - Cook-Torrance BRDF, Oren-Nayar BRDF, a iné modifikácie [1].

Pre spekulárne modely sa ľahšie vytvárajú funkcie BRDF, ale nájdeme aj dobré approximácie difúznych materiálov.

Cook-Torrance model je jedným z prvých používaných modelov. Je vhodný na spekulárne materiály. Jednotlivé zložky G, D, F sa často upravujú a vznikajú variácie. napríklad pre distribúciu poznáme Beckmann–Spizzichino, GGX alebo GTR (Trowbridge-Reitz) a mnoho ďalších. Keď Eric Heitz v roku 2007 vymyslel GGX BRDF (Trowbridge-Reitz-Generalized) [17], znamenalo to istú revolúciu v grafických softvéroch. Jeho GGX BRDF stavala na Trowbridge-Reitz modeli distribučnej funkcie D_{TR} [29]. Táto generalizovaná distribučná funkcia zlepšila presnosť a flexibilitu, čo umožňovalo použiť ju na väčší rozsah povrchov rôznych drsností.

GGX model sa stal veľmi populárnym vďaka svojej relatívnej presnosti a vizuálnej kvalite pri uspokojivej rýchlosťi výpočtu. Rokmi sa vytvorilo množstvo variácií. Na dosiahnutie fyzikálneho interaktívneho vykresľovania sa s GGX BRDF stále využívajú rôzne triky ako ambientná oklúzia v obrazovom priestore (SSAO) alebo odrazy v obrazovom priestore (SSR). Bežne vykresľovancie softvéry používajú rôzne BRDF (alebo ich variácie) pre čo najvernejšiu reprezentáciu širokej škály materiálov.

Unity

Unity je nástroj primárne určený na tvorbu multiplatformových videohier. Je vhodný hlavne pre menšie a stredne veľké tímy, ktoré vytvárajú menej náročné hry ako mobilné hry, 2D hry a jednoduchšie 3D hry. V Unity je ale možné vytvárať aj väčšie komplikované videohry. Vieme v ňom tvoriť rôzne vizualizácie, napríklad pre automobilový priemysel pomocou AxF shadera [30].

Je možné pozorovať približovanie sa filmového a herného priemyslu, unity v tomto ponúka riešenia v reálnom čase, napríklad Wētā Tools na tvorbu virtuálnych postáv[34], speedtree na prostredia[39], nástroje na kompozíciu [40]. Simulačný balík Ziva [41], napríklad Ziva VFX na simuláciu ľudí a zvierat, Ziva fur na srst' a vlasy.

Unity tiež poskytuje riešenia aj pre architektonický, automobilový, výrobný, energetický a maloobchodný priemysel, hlavne pomocou vizualizácie, simulácie, počítačového videnia a rôznych techník umelej inteligencie [35].

1.5.6 Vykresľovanie v Unity

Unity, podľa dokumentácie [36], ponúka vstavaný vykresľovací kanál BRP. Tento kanál má limitovanú modifikateľnosť, preto Unity vytvorilo skriptovateľný vykresľovací kanál (SRP). Unity ponúka 2 skriptovateľné vykresľovacie kanály HDRP, URP ako stiahnuťné prídavné moduly, balíčky. Je možné vytvoriť aj vlastný vykresľovací kanál od základov.

URP kanál je určený pre menej výkonný hardvér ako sú mobily a prenosné konzoly, respektíve je vhodný aj keď nie je prioritou dosiahnuť najväčšiu vizuálnu vernosť ale rýchlejšie vykreslovanie .

Unity HDRP

HDRP to balíček ktorý umožňuje vytvárať vysoko detailné prostredia pre najnovší hardvér s použitím náročných ale vizuálne kvalitnejších, vernejších, metód.

HDRP kanál je multiplatformový, určený primárne pre nové herné konzoly a výkonné počítače. Písaný je v HLSL jazyku a funguje na rôznych grafických API ako DirectX 11, DirectX 12, Vulkan. V HLSL kóde sa teda nachádza veľa makier ktoré potom v závislosti od hardvéru a zvolenej grafickej API spustí ten správny kód, určený na danú platformu.

HDRP vyžaduje podporu Shader Model 5.0 na GPU. Využíva aj compute shadery. Nepodporuje teda OpenGL a OpenGL ES.

HDRP je vo vývoji viacero rokov a v čase písania je na verzii 17.0.1. Keďže HDRP musí komunikovať s Unity základom, ktorý sa tiež mení, rôzne verzie HDRP vyžadujú určitú verziu Unity. Často je to z dôvodu využitia novo pridanej funkcionality Unity, ale aj kvôli odoberaniu zastaralej funkcionality. Pri začiatku vývoja je preto vhodné vybrať vhodnú verziu Unity a HDRP. pretože prechod na novšie verzie môže vyžadovať veľa roboty.

Funkcionality HDRP zahŕňajú:

- Dynamické rozlíšenie - FSR, DLSS, TAAU
- Odraz, lom, ambientná a spekulárna oklúzia, globálna iluminácia v obrazovom priestore
- Adaptívne svetelné sféry
- Materiály a shadery podporujúce priehľadné/ priesvitné povrchy, iridescentiu(dúhovatenie), podpovrchový rozptyl, materiál s povrchovou vrstvou, shader na vlasy, čiarový vykreslovač na srsť, shader na oči, látkový shader, X-Rite AxF shader
- Tieňové kaskády, kontaktné tiene, mikrotiene
- Metóda vykreslovania pomocou sledovania lúčov alebo sledovania cesty
- Volumetrické oblaky a hmla, realistický povrch vody
- HDR, Post-processing

Kapitola 2

Súčasný stav vykresľovania trblietavých materiálov

V tejto kapitole stručne popíšeme jav trblietania. Predstavíme existujúce prístupy a metódy vykresľovania trblietavých materiálov. Prejdeme najstaršími a najjednoduchšími metódami a na koniec predstavíme najnovšiu metódu.

2.1 Správanie sa svetla na povrchu materiálu

Interakciu svetla vieme popisovať na viacerých úrovniach/škálach. Na týchto škálach existujú rôzne optické vlastnosti. Najčastejšie sa zaoberáme optickými vlastnosťami na mikroskopickej, mesoskopickej, a makroskopickej škále. V týchto rôznych škálach preto vieme optické vlastnosti materiálov popísť odlišne. Boli navrhnuté rôzne metódy, ktoré opisujú povrch materiálu na jednej z týchto škál. Na mikroskopickej škále to môžeme modelovať explicitnou geometriou a cez drobné častice. Na mesoskopickej a makroskopickej úrovni používame triedu matematických funkcií BSDF, BRDF resp. BSSRDF. V literatúre nájdeme napríklad prehľad metód na získavanie optických vlastností priesvitných materiálov [11].

Existujú aj viacškálové metódy. Tieto metódy zachytávajú a vykresľujú povrch objektu naprieč 2 a viac škálami. Na to akú škálu je vhodné použiť, záleží od požiadaviek. Pokiaľ potrebujeme realistickú reprezentáciu, môžeme siahnuť po výpočtovo náročných technikách. Príkladom viacškálových modelov je vykreslenie piesku v Disney Research článku Multi-Scale Modeling and Rendering of Granular Materials [25]. Pre vykresľovanie v rálnom čase poznáme napríklad metódu vykresľovania trblietok Real-time Rendering of Procedural Multiscale Materials [47].

Bežne v fyzikálne založenom vykresľovaní sa makroskopický vzhľad, ktorý je zapríčinený mikroskopickou štruktúrou modeluje stochasticky. Funkcia NDF definuje distribúciu orientácií mikroskopických plôšok. Distribúcia definuje časť makroskopickej oblasti, ktorá je okupovaná mikroskopickými segmentami pre každú možnú orientáciu. Najčastejšie sú NDF definované v slope space (priestore sklonu) a potom sa transformujú do priestoru orientácií a do korešpondujúcej oblasti povrchu.

2.2 Trblietky, trblietavý vzhľad

Niekteré materiály ako sú automobilové nátery, brúsený kov, sneh a trblietky vytvárajú ostré svetelné odrazy. Tento fenomén nazývame trblietanie. Tento jav je zapríčinený štruktúrou povrchu materiálu, resp., povrchovou úpravou, ktorá interaguje s dopadajúcimi lúčmi. Trblietanie v automobilových lakocho vzniká vďaka rozloženiu drobných častíc v povrchu materiálu. Tieto drobné častice sa správajú ako malé zrkadielka, ktoré odrazia svetlo smerom k pozorovateľovi pri veľmi špecifickom uhle pohľadu. V článku [6] je popísaný goniochromatický lak, ktorý vo farebnej vrstve obsahuje trblietavé častice, ktoré majú normálu odchýlenú do približne 5 ° od normály povrchu.

Príkladom takýchto lakov je aj japonská technika Maki-e. Tvorbu maki-e efektu môžeme vidieť na obrázku 2.1. Táto technika je dekoračné umenie z približne desiateho storocia, kde sú najprv na povrch objektu nalakované slová, obrázky či vzory. Na lak je potom posypaný kovový prášok, napríklad zlato alebo striebro. Keď lak zaschnie tak môžeme pozorovať odrazy produkované týmito drobnými čiastočkami. Tvorba maki-e je náročná z časového hľadiska a ceny surovín. Ryou Kimura a Roman Ďuríkovič [5], [7] navrhli metódu vykresľovania, vďaka ktorej vieme rýchlo vytvoriť vzorky a prispôsobiť produkt zákazníkovi. Tým dokážeme urýchliť proces návrhu a prototypovania.

Efekt lakovania dosiahli použitím sférických máp pre každý pigment. Tie následne pomocou alfa masiek zmiešali na dosiahnutie finálneho obrázku.

Problém trblietania je aplikovateľný aj na hrčky, vločky, zrná kože, ryhy a škrabance. Na obrázku 2.2, od autorov článku Recent advances in glinty appearance rendering [46], pekne ukazujú rozdiel medzi tradičným vykresľovaním a vykresľovaním s komplexnou mikroštruktúrou.

Na ľavej strane je použitý algoritmus, ktorý používa tradičnú teóriu o mikroplôškach, teda pomocou NDF. Na pravom obrázku je definovaná mikroštruktúra pomocou normálovej mapy, pomocou metódy [45]. Vidíme 4 rôzne typy trblietavých odleskov. Textúra pohovky (1), nožičky pohovky (2), nerovný povrch podstavca (3) a poškria-



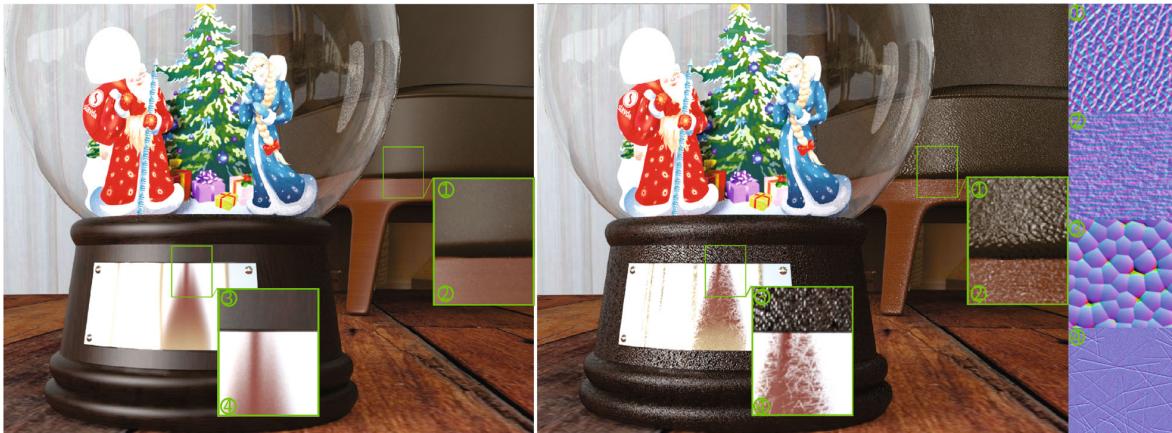
Obr. 2.1: Maki-e technika lakovania. [22]

baný kov (4). Modely majú všetky rovnakú polygonálnu siet, líšia sa len pridaným detailom mikroštruktúry, ktorá viditeľne zlepšuje vizuálnu kvalitu.

Efekt trblietania je zaujímavý aj v oblasti virtuálnej reality. V článku [8] autori vytvorili aplikáciu, so stereo vykresľovaním. Vedia tým simulovať efekt trblietok v rôznej hĺbke.

2.3 Problém vykresľovania trblietok

Vykresľovanie trblietavých materiálov má dva zložité problémy. Prvým je reprezentácia mikroštruktúry a druhým je vykreslenie danej mikroštruktúry. Spravidla sú tieto metódy náročné na výpočet, ale poznáme i metódy, ktoré s určitými obmedzeniami a zjednodušeniami vieme použiť v interaktívnom vykresľovaní v reálnom čase. Ak pôjde o takúto rýchlu metódu, tak to zdôrazníme, ináč pôjde o neinterakívnu metódu. Článok Recent advances in glinty appearance rendering [46] ponúka prehľad existujúcich metód, kategorizovaných podľa podobnosti metód.



Obr. 2.2: Ukážka dôležitosti mikroštruktúry. Materiály s jednoduchým hladkým povrchom naľavo. V obrázku napravo bola zachovaná mikroštruktúra povrchov.

2.4 Mikroštruktúra trblietavých materiálov, ich reprezentácia

Drobné povrchové detaily sú najlepšie viditeľné pod silným svetlom a na lesklých povrchoch. Majú veľkú komplexnosť a sú veľmi malé. Preto reprezentovať tieto detaily nie je jednoduché. Musíme teda myslieť na kompaktnú reprezentáciu, ktorú vieme efektívne vyhodnotiť pri vykreslovaní.

Reprezentácia mikroštruktúry by mala čo najvernejšie zachytiť reálnu mikroštruktúru. Nemala by ale zaberat' príliš veľa priestoru, pretože v komplexných scénach máme mnoho materiálov s rozličnou štruktúrou. Vtedy by sme potrebovali veľa úložného priestoru. Navyše vykreslenie mikroštruktúry by bolo tiež spomaľované, pretože s týmto rozsiahlym množstvom dát musíme vedieť efektívne pracovať.

Reprezentovať mikroštruktúru vieme explicitne alebo implicitne.

2.5 Explicitná reprezentácia mikroštruktúry

Explicitne reprezentovať mikroštruktúru môžeme viacerými spôsobmi. Explicitná reprezentácia je vhodná najmä pre offline vykreslovanie. Problémom explicitných reprezentácií je veľká pamäťová náročnosť. Dokážu ale reprezentovať viac druhov trblietania ako hrčky, preliačiny, vločky, škrabance alebo vzorky kože.

Na začiatok môžeme celú mikroštruktúru explicitne definovať pomocou explicitnej geometrie modelu. Získame tak najkvalitnejší výsledok, za cenu úložiska a vykreslovania takého veľkému počtu polygónov.

Ako lepšie a stále jednoduché riešenie sa javí reprezentovať mikroštruktúru v normálnej mape. Reprezentujeme teda odlesky ako normály. Toto naivné riešenie, má ale množstvo nedostatkov. Síce normálová mapa s dostatočne vysokým rozlíšením (zvy-

čajne sú potrebné 10-tky až 100-vky tisíc pixelov na oboch osiach) nám umožňuje verne reprezentovať detaily mikroštruktúry, má ale množstvo nevýhod. Jednou z nich sú vysoké pamäťové nároky na uloženie takejto mapy a následne jej posun do pamäte GPU VRAM, ktorá je tiež limitovaná. Taktiež nám to neumožňuje použiť importance sampling [46].

Ak si pomocou normálovej mapy zadefinujeme naozaj malé odlesky, tak táto mapa môže dosahovať gigabajty. To predstavuje problém, pretože tých materiálov chceme ideálne mnoho a pamäť je limitovaná nielen na harddisku ale aj VRAM na grafickej karte.

Diskrétne trblietky vieme reprezentovať ako náhodné normály s pozíciami ako napríklad [9], kde trblietanie vyskúšali na automobilových lakoch a lakovacej technike Nashiji.

Existujú aj inverzné techniky, ktoré z odmeranej BRDF laku nájdu kompozíciu pigmentov vo danom laku. Následne ju napasujú na jednoduchý 2 vrstvový model farby [10]. Metódami strojového učenia vieme inverzne získať z nameranej BRDF explicitné parametre.

2.6 Vykresľovanie trblietavých materiálov

Komerčný softvér na offline vykresľovanie využíva hlavne metódy sledovania lúča, respektíve sledovania cesty. Pri sledovaní cesty vzorkujeme zasiahnutý povrch, na ktorom sa ale môže nachádzať veľmi veľa (tisícky) malých plôšok. Z nich iba niektoré môžu vplývať na výslednú farbu pixelu. Prejsť všetky plôšky je veľmi náročné a nepraktické. Naproti tomu, vylúčenie prispievajúcich plôšok vo vyhodnocovaní vedie k zašumenému obrazu. Pri pohybe kamery sa tiež vytvára šum, pretože sa prehľadajú úplne iné plôšky ako snímku predtým. Je teda dôležité vybrať tie, ktoré najviac prispievajú k finálnej farbe, tzv. importance sampling, aby sa predišlo šumu, temporálnej nekoherencii. Predchádza sa tomu tak, že sa zvýši počet vzoriek na jeden pixel, čo al významne zvyšuje náročnosť výpočtu. V súčasnosti sa používajú tzv. odšumovače (angl. denoiser) na zníženie potrebných vzoriek. Často sú tieto odšumovače trénované pomocou strojového učenia prípadne, inou technikou umelej inteligencie.

2.7 Offline, neinteraktívne metódy

Základná idea vykresľovania spočíva v spracovaní plôšky povrchu P na jednom pixeli. Túto plôšku videnú cez jeden pixel môžeme predfiltrovať, čo ale vedie k hladším výsledkom, kde sa znižuje variácia mikroštruktúry. Najpresnejšie výsledky dosiahneme, keď zoberieme danú celú plôšku a vypočítame diskrétnu NDF [46].

Jedna z predfiltrovacích stratégií je vytvoriť dataset nameraných odrazov pri rôznych uhloch pozorovania a rôznych smeroch svetla. Táto metóda vytvára 6D tabuľku, nazvanú bidirectional texture function BTF. Metódou BTF sa ľahko a rýchlo vykreslí virtuálny objekt. Problémom je, že nie je preneseľná na iné materiály. Každý iný materiál si teda vyžaduje nové nameranie. Materiály so zložitým optickými vlastnosťami, ako napríklad vysoko anisotropické alebo nehomogénne materiály sa ľahko merajú.

Ďalšie existujúce metódy využívajú gausiánske laloky. Použitím viacerých lalokov získame komplexnejšiu NDF. Táto NDF je ale stále príliš hladká oproti reálnemu povrchu [46].

Metóda od autorov Jakob et al. [21] je stochastická metóda, ktorá používa diskrétnu NDF. Stochastickou metódou sa vyhli explicitnej reprezentácii všetkých častíc. Táto metóda produkuje kvalitné výsledky a podporuje úpravu parametrov. Nevieme s ňou zobraziť anisotropické trblietky alebo drážky/škrabance, ako poukazujú autori T. Zirr a A. S. Kaplanyan [47]. Autori distribuujú plôšky pomocou 4D hierarchického prechádzania, ktoré je časovo náročné.

2.8 Metódy v reálnom čase

Metódy vykresľovania trblietavých materiálov v reálnom čase musia svoj výpočet niekoľkonásobne urýchliť. Vykrešlenie jedného snímku by nemalo presiahnuť 33ms - teda, to predstavuje 30 snímkov za sekundu. Samotná metóda vykreslenia trblietok je len časť by mala trvať jednotky milisekúnd, aby zvýšil čas na ostatné časti vykresľovacieho procesu.

Ako sme spomenuli v predchádzajúcej sekcii, už pri neinteraktívnom vykresľovaní máme problém vhodne reprezentovať a verne vykresliť komplexnú mikroštruktúru. Metódy v reálnom čase preto musia zjednodušovať reprezentáciu ako aj vykresľovanie.

Dôležitým problémom je koherencia, stabilita obrazu. Pri posunutí kamery alebo trblietavého objektu sa často vyskytuje problém preblikavania, šumu trblietok.

2.8.1 Procedurálne generované trblietky

Procedurálne generované materiály a efekty sú zaujímavé kvôli nízkej pamäťovej náročnosti a flexibilite. S pomocou šumu, napríklad perlinovho šumu, vieme pridať náhodnosť, ktorá je hladká a konzistentná. Túto ideu použil Jeremy Shopf [28] na vytvorenie jednoduchého efektu trblietok na povrchu materiálu. Vytvorí sa 3D mriežka na ktorej sú rovnomerne rozmiestnené trblietky. Následne, tam kde sa povrch trblietavého objektu pretne s mriežkou trblietok vykreslíme trblietku. Mriežka je perturbovaná perlinovým šumom aby sa zakryl vzhľad uniformnej mriežky.

Takto dostaneme pomerne jednoducho efekt trblietok. Je to jednoduchý algoritmus

```

1 float specBase = saturate(dot(reflect(-normalize(viewVec), normal),
2 lightDir));
3 // Perturb a grid pattern with some noise and with the view—vector
4 // to let the glittering change with view.
5 float3 fp = frac(0.7 * pos + 9 * Noise3D( pos * 0.04).r + 0.1 * viewVec);
6 fp *= (1 - fp);
7 float glitter = saturate(1 - 7 * (fp.x + fp.y + fp.z));
8 float sparkle = glitter * pow(specBase, 1.5);

```

Kód 2.1: Shopf trblietky

ktorý funguje na jednoduché scény. Má veľa nedostatkov. Nie je fyzikálne presný, ale je efektívny. Vzniká tu aliasing trblietok. Tvar trblietok znásobuje tento efekt, pretože sú špicaté. Trblietky sú navyše jednej veľkosti a teda blízko pozorovateľa sú veľké a v diaľke zas príliš malé. Nefunguje dobre na príklade rozsiahleho zasneženého terénu. Táto metóda produkuje len riedke trblietky. Pri veľkej hustote je až moc viditeľná uniformnosť mriežky. Pod malým uhlom pozorovania (rovina povrchu a smer pozorovania je blízko 0°) tiež vzniká aliasing.

Metóda Bebei Wang a Huw Bowles [43] stavia na algoritme od Shopfa. Je robustnejšia, ale stále relatívne veľmi jednoduchá.

Wang a Bowles zmenili tvar trblietky zo špicatej na okrúhlu. Takto sa ľahšie predchádza aliasingu. Aby dosiahli správny vzhľad trblietok pri meniaci sa vzdialenosť od kamery, zmenili výpočet veľkosti trblietky a hustoty mriežky. Veľkosť trblietky je závislá od hustoty mriežky. Mriežka je škálovaná logaritmicky v závislosti vzdialenosťi povrchu od kamery tak, že vytvárajú skoky/ úrovne škálovania. Je teda viditeľný rez kde sa zmení škála, trblietky sú hustejsie a menšie oproti predchádzajúcemu levelu, resp. naopak, sú väčšie a redšie. Podľa autorov to nebolo až príliš rušivé a pôsobilo to ako efekt ztmavenia a shasnutia trblietky. Pri hustejsiej mriežke to ale prekáža oveľa viac.

Shopf používal 3D perlinov šum pri získavaní mriežky(presnejšie ked' do nej vstupujú, prehľadávajú ju) aby sa trblietanie menilo s pohľadom. To ale prinášalo problém zmenšovania a zväčšovania tvaru trblietky. Wang a Bowles namiesto toho perturbujú stred mriežky namiesto perturbovania vyhľadávania do mriežky. Nakoniec, trblietky sú na povrchu roztiahnuté a preblikávajú pri ostrom uhle pozorovania. Na odstránenie tohto problému sa inšpirovali algoritmom Elliptical Weighted Average texture filtering[16]. Predlžujú kernel trblietky zo sféry na elipsu, kde hlavná os je dotyčnica k povrchu, ktorá je v smere podvzorkovania.

Oproti Shopfovmu riešeniu funguje táto metóda po celej hĺbke pozorovania, nevytvára veľký aliasing, vďaka čomu trblietky môžu zmeniť na len 1-2 pixely. Metóda je stále celkom nenáročná na výpočet ale stále sa s ňou ľažko zobrazujú husto rozmiestnené trblietky.

2.8.2 Real-time Rendering of Procedural Multiscale Materials

Tobias Zirr a Anton S. Kaplanyan vychádzajú z metódy navrhnutej Jakobom et.al [21] Discrete Stochastic Microfacet Models. Modifikujú algoritmus aby nevyžadoval náročné 4D hierarchické prehľadávanie a dosiahlo rýchlosť vykresľovania potrebnú na interaktívne vykresľovanie.

Uvažujú o mikrodetailoch s vlastnou NDF, ale metóda Jakob et al. predpokladala iba jednu odrážajúcu plôšku.

Nahrádzajú explicitné počítanie počtu odrážajúcich sa plôšok štatistickým priemierom pomocou binomického zákona.

Teoretické východisko, mikrodetailsy

Pri pozorovaní objektu zblízka, využívajú lokálnu NDF. Tá sa mení vzhľadom na polohu. Ak sa pozérame z väčšej dĺžky, tak NDF konverguje na makroskopickú / globálnu NDF. Globálnu NDF konštruujujú pomocou spojenia viacerých lokálnych NDF, aby dosiahli lepšiu kontrolu nad trblietkami a celkovým výzorom. Povrch na makroskopickej úrovni nazývajú mikrodetailsy. Mikrodetailsy sú plochy s určitou lokálnou NDF, $D_l(m)$. Tieto kúsky mikrodetailov x potom náhodne inšancujú pomocou ďalšej funkcie mesoscale distribution of microdetails (MDDF) $D_\mu(x)$. Zhlukovanie mikrodetailov teda vytvára globálny mikropovrch, globálnu NDF, ktorú získame konvolúciou $D_g(m) = \int D_\mu(x)D_l(m-x)dx$.

Diskretizácia

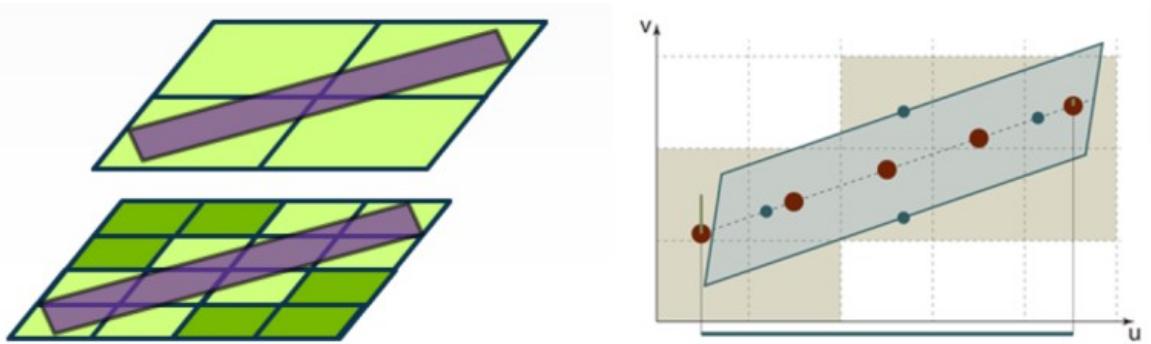
Oproti metóde Jakob et.al, majú mikrodetailsy ľubovoľný rozsah lesku, ktorý nevplýva na makroskopický vzhľad.

To zlepšuje výzor mikrodetailov ale sťaže diskretizáciu na počítanie odrážaných mikrodetailov.

Na vyriešenie tohto problému sa rozhodli previesť spojité odrazivosť každého jednotlivého kúsku mikrodetailu na diskrétnu pravdepodobnosť tak, že každý mikrodetail buď prispeje svojou maximálnou odrazivosťou definovanou $D_l(0)$, alebo vôbec neprispeje. Odrazivost x je pravdepodobnosť $P_x(m) = \frac{D_l(m-x)}{D_l(0)}$ x -ka, že prispieva maximálnou intenzitou.

Stochastický proces

Počítanie založili na diskrétnej pravdepodobnosti pre každý mikrodetail, teda je využadený ako hierarchický multinomiálny sčítavací proces. Tento proces, vykonávaný na každý pixel je príliš náročný pre interaktívne aplikácie, preto zjednodušili spočítavanie prispievajúcich mikrodetailov aby dosiahli lepšiu efektívnosť. Presnejšie, zjednodušili



Obr. 2.3: Na obrázku naľavo vidíme 2 úrovne mriežky vybraté na výpočet počtu mikrodetaľov. Na obrázku napravo je stopa pixelu, podľa ktorého sa vyberú 2 úrovne mriežky. [47]

počítanie na jednu binomickú premennú. Znamená to, že šanca, že mikrodetail bude odrážať je rovnaká pre každý mikrodetail. Tak dostaneme jedno jednoduché binomické rozdelenie počtu mikrodetaľov, ktoré rozdelí celkový počet mikrodetaľov na ploche na odrážajúce(s plnou intenzitou) a na neodrážajúce. Pri vzorkovaní binomickej distribúcie limitujú počet pokusov na malý počet čísel. Po dosiahnutí tejto hranice používajú gaussovú distribúciu s prvým a druhým momentom binomickej distribúcie na fitting.

Koherencia

Plocha pixelu sa medzi snímkami veľmi mení, čo môže viesť k nestabilnému obrazu, mihotaniu, nekoherentným trblietkam. Riešenie problému spočíva vo vytvorení vnořenej mriežky mocnín 2^{ky} v priestore textúry, pomocou UV súradníc. Každá bunka pokrýva unikátnu časť uv textúry. Počiatočnú hustotu zadá používateľ, každá ďalšia úroveň mriežky predstavuje dvojnásobne hustejšiu mriežku. Keďže sú to mocniny 2^{ky} vieme hierarchiu efektívne prechádzať pomocou bitových operácií.

Pixel môže pokrývať viacero buniek mriežky/texelov. Vyberú sa 2 susedné mriežky ktoré najviac pokrývajú pixel. Pre každú bunku v mriežke spočítajú celkový počet obsiahnutých mikrodetaľov a aj stabilnú počiatočnú hodnotu pre generátor, ktorý môžu použiť na priamy binomický výber počtu odrážajúcich mikrodetaľov. Potom môžu aplikovať techniku anizotropického filtrovania a trilineárnu interpoláciu výsledkov na každú bunku v stope pixelu a 2 susedné úrovne mriežky, ktoré pasujú ploche pixelu najlepšie.

V stope pixelu vyberú 2 susedné úrovne mriežky, ktoré pasujú ploche pixelu najlepšie. Potom podobne ako pri anisotropnom filtrovaní textúr vypočítajú tieňovanie na každej bunke v oboch mriežkach a zmiešajú ich.

Pridanie závislosti od uhlu pozorovania

Priestor prehľadávania odražajúcich mikrodetailov je v skutočnosti 4D, pretože počítanie sa mení pre rôzne orientácie mikrodetailov. Diskretizujú polovičný vektor H zo stredu pixelu na plochu pomocou paraboloidného mapovania. Takto približne zachovávajú hustotu polovičných vektorov približne rovnomernú pri pevnom uhle. Túto plochu potom diskretizujú do viacerých buniek a indexy používajú na počiatočnú hodnotu generátora mikrodetailov. Potom, počiatočná hodnota generátora, seed, je určená indexom bunky textúry a indexom mriežky polovičného vektora na vyvodenie binomickej náhodnej premennej. Tieto polovičné vektory ešte perturbujú pre každý pixel, aby eliminovali problém zmeny všetkých trblietok naraz pri zmene indexu.

Kohерентný šum

Nakoniec potrebujú správne zmiešať vypočítané náhodné premenné z rôznych úrovní vnorenej mriežky textúr, aby nedochádzalo k rozmažaniu. Vyriešili to tak, že levej mriežky spravili medzi sebou korelované - každá hrubšia úroveň vždy obsahuje člena s jemnejšou úrovňou.

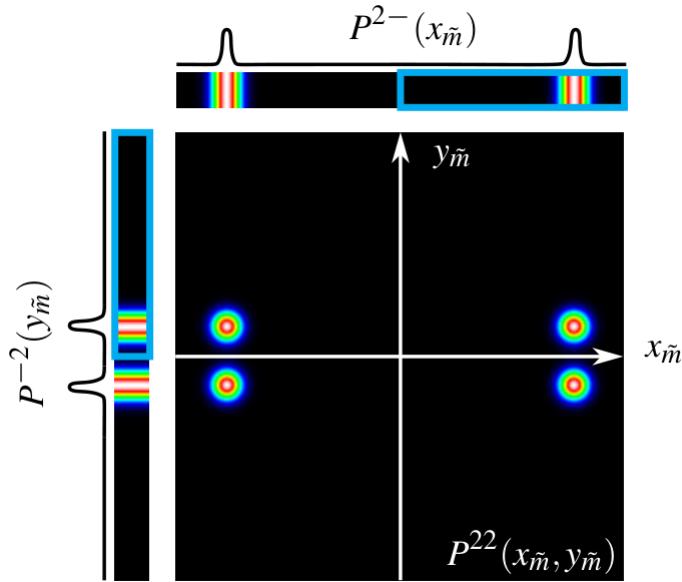
Tento algoritmus poskytuje kontrolu nad makroskopickými a lokálnymi detailmi. Je priestorovo stabilný. Nevýhodou je závislosť rýchlosťi výpočtu od povrchu pixela, pokiaľ prekrýva veľa buniek mriežky. Je stále relatívne náročný na výpočet, je lineárne závislý od počtu svetiel. Anizotropické filtrovanie stopy pixelu je výpočtovo náročné a limitujú ho na úroveň 16.

2.8.3 Procedural Physically based BRDF for Real-Time Rendering of Glints

Metóda od autorov Chermain et al. je fyzikálne založená. Zachováva zákon konzervácie energie. Oproti metóde Zirr a Kaplanyana ktorí používali binomické pokusy, používajú autori normalizovanú NDF evaluáciu. Tiež využívajú viaceré úrovne, hierarchie, konkrétnie ju nazývajú MIP hierarchia.

Základné kroky metódy sú:

1. Filtrujú procedurálnu MIP hierarchiu. Každá bunka hierarchie nemá explicitne danú NDF.
2. Každá bunka svojim indexom nastaví počiatočnú hodnotu generátora, ktorý náhodne vyberie 2 1D marginálne distribúcie zo slovníka.



Obr. 2.4: Produkt 2 1D marginálnych distribúcií. Ukladá sa iba pozitívna časť (v moderných rámčekoch). [19]

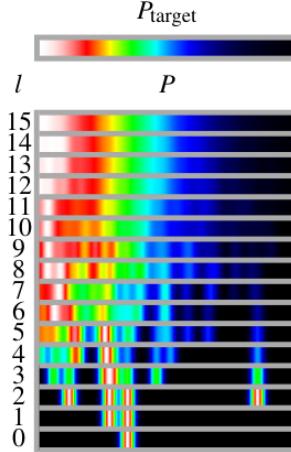
3. Produkt týchto 2 distribúcií je 2D NDF. Vďaka randomizácii sa vyhnú opakujúcim sa vzorom a podobným artefaktom. Vďaka tomu, že neukladajú NDF v hierarchii, má algoritmus nízke pamäťové nároky.
4. Pre každú NDF sa ešte aplikuje náhodná rotácia a parameter škálovania, zadaný používateľom.
5. Váhovaná suma na bunkách ktoré pokrýva stopu pixelu vytvorí NDF, definovanú v diskrétnej škále - LOD.
6. Nakoniec, pomocou interpolácie naprieč škálami dostanú finálnu NDF, ktorá je použitá v BRDF.

Odvodenie viacškálovej SNF

Mikroštruktúru charakterizujú ako štatistickú distribúciu orientácií mikroplôšok. Môže byť definovaná ako distribúcia mikronormál ω_m . - $D(\omega_m)$ (NDF) . Alternatívne ju definujú pomocou distribúcie sklonov / svahov mikroplôšok. $\tilde{m} = (x_{\tilde{m}}, y_{\tilde{m}}) = (\frac{-x_m}{z_m}, \frac{-y_m}{z_m})$ To je teda sklonová distribučná funkcia, SDF, \tilde{m} je mikrosklon. Obe so sebou súvisia:

$$D(\omega_m) = \frac{P^{22}(\tilde{m})}{(\omega_m \cdot \omega_g)^4}, \text{ } \omega_g \text{ je geometrická normál povrchu, stredná hodnota mikronormál.}$$

Aby bola metóda fyzikálne správna, distribúcie musia byť normalizované (plocha premietnutá mikroplochou musí byť rovná makroploche). $\int_{\Omega} (\omega_m \cdot \omega_g) D(\omega_m) d\omega_m = 1$, čo je ekvivalentné rovnici $\int_{R^2} P^{22}(\tilde{m}) d\tilde{m} = 1$.



Obr. 2.5: Úrovne l SDF lóbov, kde posledná je klasická priemerná NDF. [19]

Definícia P^{22} je spojenie nezávislej funkcie hustoty pravdepodobnosti, ktorú získame pomocou dvoch jednorozmerných marginálnych distribúcií P^{2-} a P^{-2} po osiach x a y. $P^{22}(\tilde{m}) = P^{2-}(x_{\tilde{m}})P^{-2}(y_{\tilde{m}})$. P^{22} je normalizovaná ak sú obe zložky normalizované. Používajú symetrické marginálne funkcie hustoty, ktorými sa vyhnú problému, kde stredná hodnota mikronormál je odlišná od geometričkej normály. Navyše tak ušetria polovicu pamäte, keďže do slovníka uložia iba pozitívnu časť funkcie.

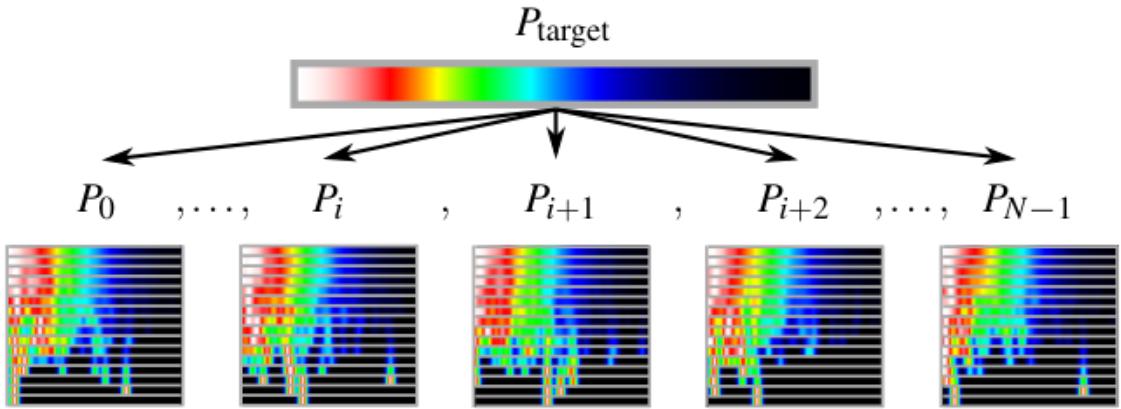
Teraz pridajú vlastnosť viacškálovosti do ich SDF. Pridajú parameter l - level v LOD hierarchii. $P^{22}(\tilde{m}, l) = P^{2-}(x_{\tilde{m}}, l)P^{-2}(y_{\tilde{m}}, l)$. Level 1 je daný ako rozsah od 0 (najjemnejšia škála) - 16 (najhrubšia škála). Na najvyššom leveli $l = 16$ vynútili konvergenciu k Cook-Torrance BRDF. To znamená, že $P^{22}(\tilde{m}, \max L) = P_{target}^{22}(\tilde{m})$. Beckmannovú NDF zvolili z dôvodu jej oddeliteľnosti - jej výsledok je produkt marginálnych distribúcií. Z tohto dôvodu nepodporujú GGX BRDF, pretože jej GTR NDF nie je separovateľná.

Pri prechode medzi susednými LOD sa početlóbov štvornásobní. Na úrovni l viacškálová SDF je mix 4^{l+1} lóbov a ich marginálne distribúcie sú zmes 2^{l+1} lóbov. Na poslednej najväčšej úrovni je NDF charakterizovaná jedným lóbom, predstavuje neko-nečný počet mikroplôšok.

Generovanie slovníka

Pomocou gausovských lóbov s malou smerodajnou odchýlkou generujú jednu viacškálovú SDF. Pozície - stredné hodnoty gausových lóbov sú získané importance samplingom P_{target} . Prvá náhodná vzorka definuje lób na $l = 0$, druhá náhodná vzorka definuje $l = 1$ lób atď.

Na trblietky potrebujeme veľa takýchto SDF. Vytvorením slovníka N marginálnych distribúcií $P_i, 0 \geq i \geq N$ dosiahnu potrebnú variáciu. Zvolením väčšieho N dosiahnu väčšiu variáciu a pri vykresľovaní budeme menej pozorovať vzory za cenu väčších



Obr. 2.6: Slovník marginálnych distribúcií na rôznych škálach, ktoré konvergujú k cieľovej P_{target} - napr. Beckmannovej NDF. [19]

pamäťových nárokov.

Aby sme vedeli ovládať drsnosť a anisotropiu p^{22} , nezávisle po osi x aj y, museli by sme mať slovník na jednu os. Autori to spravili tak, že majú jeden slovník, ale škálujú ho a rotujú pri evaluácii.

Priestorovo premenlivá a viacškálová SDF

Povrch rozdelili do štvorcových mriežok, z ktorej má každá svoju SDF. Na LOD používajú podobne ako pri MIP hierarchiách bunka $l + 1$ reprezentuje 4 susedné bunky na leveli l .

$$P^{22}(\tilde{m}, l, s_0) = P^{2-}(x_{\tilde{m}}, l, s_0)P^{-2}(y_{\tilde{m}}, l, s_0).$$

Aby zabezpečili koherenciu medzi levelmi a bunkami, tak musí platiť, že SDF úrovne $l + 1$ sa rovná priemeru štyroch susedných SDF úrovne l . To by znamenalo predpočítať všetky kombinácie. So stúpajúcou úrovňou l by to viedlo k exponenciálnemu nárastu. Použijú aproximáciu: SDF na úrovni $l + 1$ získava lób iba jedného SDF na úrovni l . Počet lóbov sa štvornásobí preto, že sa dostaneme na vyššiu úroveň v hierarchii slovníka a nie preto, že berieme do úvahy 4 spojené susedné bunky.

Autori pridali variáciu rotáciami, aby predišli zjavným artefaktom, ktoré sa prejavia na povrchu. Pridali ovládanie celkovej drsnosti povrchu : $P_{\theta,\alpha}^{22}(\tilde{m}, l, s_0) = \frac{\alpha_{dict}^2}{\alpha_x \alpha_y} P^{22}(S_\alpha^{-1} R_\theta^{-1} \tilde{m}, l, s_0)$, kde $S_\alpha = \frac{1}{\alpha_{dict}} \begin{bmatrix} \alpha_x & 0 \\ 0 & \alpha_y \end{bmatrix}$

2.8.4 Real-Time Rendering of Glinty Appearances using Distributed Binomial Laws on Anisotropic Grids

Metóda od autorov Deliot a Belcour [4] je ku dňu písania najnovšou metódou vykreslovania trblietok v reálnom čase. Ich práca rozširuje metódu Zirr a Kaplanyana.

Autori, podobne ako predchádzajúca metóda, náhodne evaluujú NDF pomocou stochastického procesu počítania. Merajú podiel vločiek/mikroplôšok v rámci stopy pixelu, ako keby boli náhodne rozmiestnené po povrchu objektu.

Najväčším prínosom tejto metódy je, že vymysleli spôsob ako napasovať parametrizovanie textúry, ktoré sa používajú na náhodné čísla v stope pixelu a zabezpečili, že na každej z mriežok úrovni detailov sa stopa namapuje na konštantný počet texelov. Takto zabezpečia, že výpočet bude rýchly a stabilný.

Zmiešavanie výsledkov binomických pokusov

Oproti metóde Zirr et al. neinterpolujú výsledky binomických pokusov 2 levelov, ale interpolujú priamo binomické parametre.

Metóda Zirr a Kaplanyana má niekoľko nedostatkov. Prvým je, že ak stopa pixelu zaberá viaceré texely, proces spočítavania výpočtovo rastie, pretože binomický pokus musíme opakovať pre každý texel.

Spočítavacia metóda Zirr a Kaplanyana tiež neráta s priestorovou pozíciou mikroplôšok v stope pixela. Presnejšie, mikroplôška má náhodnú pozíciu na povrchu a je trblietkou ak platí že, je v priestore stopy pixelu a odráža svetlo. Tieto 2 podmienky sú binomické pokusy s rôznymi pravdepodobnosťami. Kombinovaný výsledok týchto 2 pokusov nazývajú autori priestorový bernoulliho pokus.

Najprv definujú priestorovú distribúciu bodov. Z 2 spomenutých možností, uniformne distribuovaných na povrchu alebo náhodne perturbované v pravidelnej mriežke si vybrali druhú možnosť pre lepšiu numerickú presnosť.

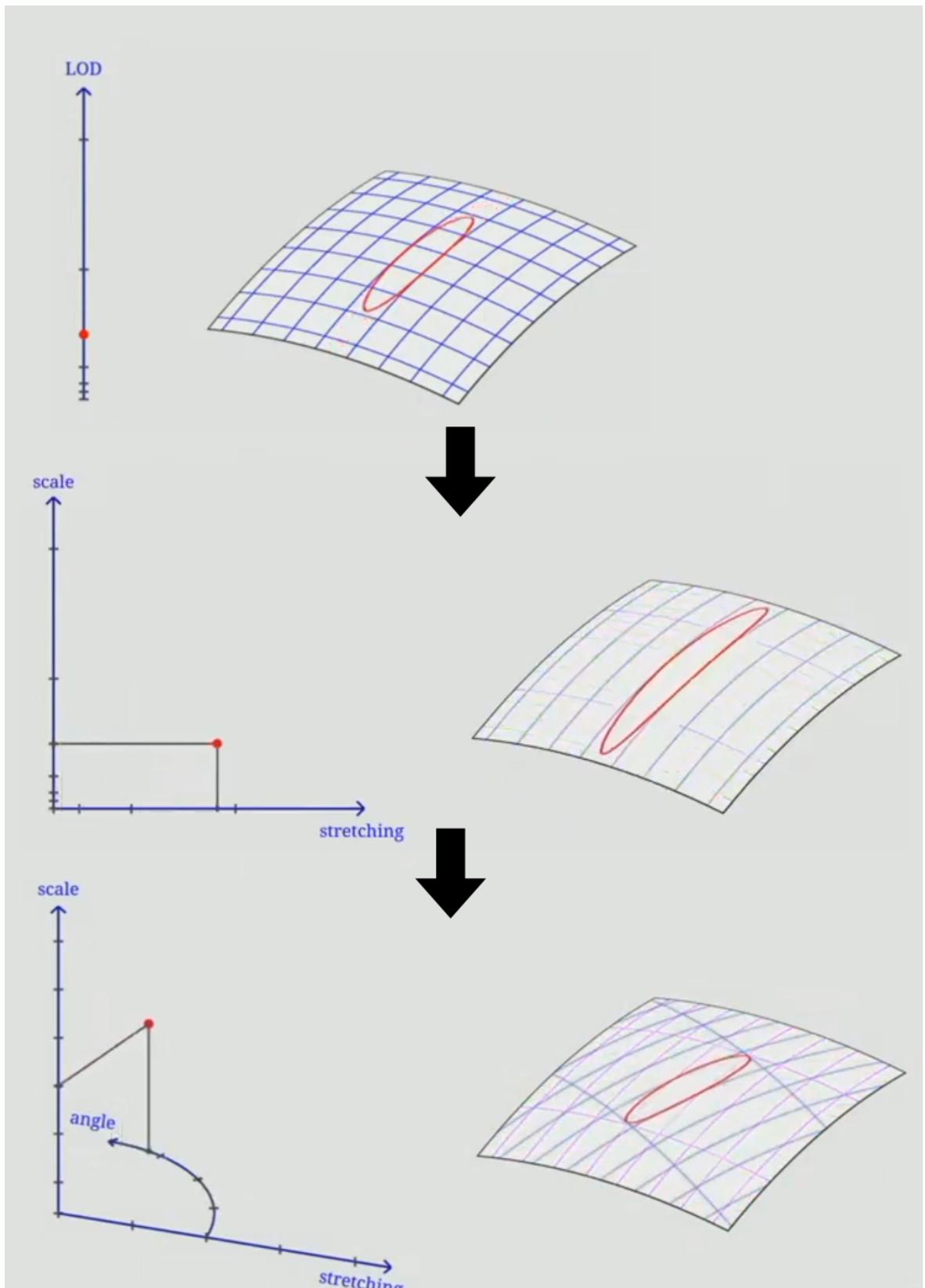
Tvorba anisotropických virtuálnych mriežok

Tak ako aj predošlé 2 metódy, aj v tejto metóde vytvárajú autori viaceré mriežky na povrchu objektu. Autori rozšírili túto myšlienku o prekrytie objektov mriežkami, ktoré sú lineárnymi transformáciami uniformnej mriežky. Tako vedia napasovať texely tak, aby pokrývali 1 stopu pixelu. Tým vedia dosiahnuť konštantnú výpočtovú náročnosť.

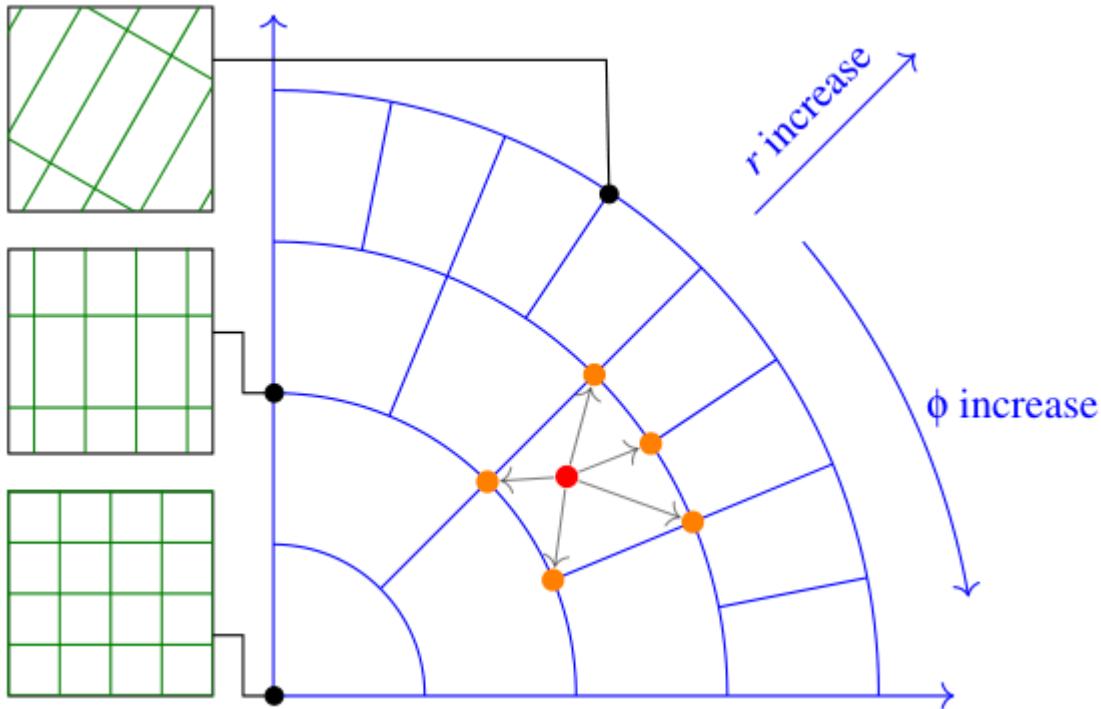
Zirr a Kaplanyn škálovali UV súradnice uniformne, mriežky boli rovnako veľké: $\theta(uv')$ kde $uv' = S_s(uv)$. Deliot a Belcour pridali rotáciu a pomery: $uv' = R_\phi S_{1,r}(S_s(uv))$ K diskrétnym mriežkam škály pridali mriežku pomerov a mriežku rotácií. Pixel premetnu na povrch objektu a extrahujú hlavnú a vedľajšiu os jeho asociovanej elipsy. Cez ne získajú rotáciu ϕ a pomer r .

Implementácia

Máme 3D mriežku, kde jedna os predstavuje diskrétne veľkosti-škály, druhá os predstavuje diskrétne pomery a tretia os predstavuje diskrétne uhly. Každá bunka v 3D mriežke obsahuje 3 rôzne orientácie a 10 bodov na výpočet binomického pokusu (5



Obr. 2.7: Ilustrácia stopy pixelu v mriežke. Na obrázku hore vidíme LOD mriežku, kde máme iba škálu. Na obrázku v strede vidíme pridanie pomerov. Spodný obrázok ilustruje pridanie orientácií. [18]



Obr. 2.8: Ilustrácia výslednej 3D mriežky. Bod označený červenou bodkou sa nachádza niekde medzi 2 úrovňami škály, 3 orientáciami a 2 pomermi. Preto potrebujeme $2 * 5$ bodov na vypočítanie jedného texelu- jednej bunky mriežky [4] .

bodov na pomery, 2 body pre škálu). Keďže potrebujeme stabilný, koherentný obraz, vykonávame interpoláciu medzi 4 texelmi a to ako pre orientáciu aj pomery. Takáto implementácia si teda vyžaduje 160 evaluácií binomického zákona. $2 * 5$ anizotropická mriežka $* 4$ lineárne zmiešané priestorové texely $* 4$ orientačné texely.

Počet potrebných binomických vzoriek vedia znížiť optimalizáciou. Každá bunka bude štvorsten. Mriežku, ktorej každá bunka má 4 vrcholy premenia na mriežku zloženú z trojuholníkov. Keď ho samplujú, pri odbere vzoriek z povrchu diskretizujú priestor povrchu do 2D simplexovej siete namiesto pravidelnej štvorcovej siete podľa simplexovej metódy šumu. Takto zo 160 evaluácií potrebujú len $4 * 4 * 4 = 64$ evaluácií na pixel. Potrebujú teda stále 4 texely pre orientáciu a 4 texely pre pomery na zmiešavanie, ale stačia už len 4 vrcholy štvorstenu, namiesto pôvodných 10tich.

Aproximácia binomického zákona

Posledná optimalizácia je urýchlenie výpočtu binomického zákona, ktorú potrebujeme pri každom výhodnotení. Zirr a Kaplanyan metóda používa strednú hodnotu a rozptyl z gaussovej approximácie binomického zákona. Deliot a Belcour namiesto toho používa len 2 náhodné čísla na vzorkovanie a bez cyklických pokusov. Ich metóda spresňuje štatistiku pri nízkom počte pokusov pomocou pridania jedného uzatvárajúceho bino-

mického pokusu - pravdepodobnosť, že bude aspoň jeden úspech. Takto potrebujú len 2 náhodné čísla na vzorkovanie guasovej distribúcie pomocou uzatvorenia binomického procesu $p = P_{\geq 1}$

Zhrnutie

Táto metóda má stále limitácie. Metóda nie je úplne fyzikálne presná, nemusí byť dodržaný zákon zachovania energie. Nepodporuje anizotropné trblietky, autori v tomto prípade radšej odporúčajú použiť metódy škrabancov. Proces počítania je zadefinovaný v závislosti s NDF. Tá ale neráta so svetlom, preto táto metóda vie vykresľovať trblietky iba s bodovými svetlami. Je závislá od UV súradníc a nespojité UV mapy produkujú artefakty.

Taktiež evaluácia binomického zákona je použiteľná iba pre konkrétné svetlo. Náročnosť výpočtu preto stúpa lineárne s počtom svetiel ktoré osvetľujú povrch.

Veľkou výhodou je, že umožňuje použitie pri GGX NDF a má konštantnú výpočtovú náročnosť.

Kapitola 3

Implementácia

V tejto sekcií predstavíme v akom prostredí budeme implementovať vlastný algoritmus vykresľovania trblietok. Zhrnieme výhody a limity použitého prostredia, ukážeme implementáciu už existujúcich metód.

3.1 Vývojárske prostredie

Pracovať budeme v prostredí OS Windows 10 s Intel i7-9750H procesorom a NVIDIA GTX 1660ti - laptop grafickou kartou.

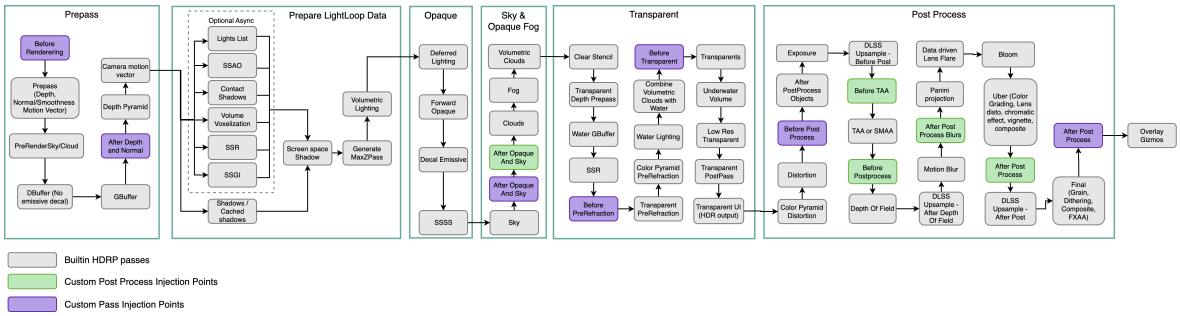
3.2 Výber grafického softvéru

Na trhu sú viaceré grafické programy na vykresľovanie v reálnom čase. My sme sa rozhodli použiť Unity, verzie 2022.1. Pracovať budeme s balíčkom HDRP 13.1.8. Pre Unity HDRP sme sa rozhodli preto, že je to balíček určený pre najmodernejší a najvýkonnejší hardvér s cieľom dosiahnuť vysokú vizuálnu úroveň. Je to hardvérovo najnáročnejší a vizuálne najlepší vykresľovač, čo ponúka Unity. Trblietky sú komplexný efekt, ktorý má nezanedbateľný vplyv na výkon. Taktiež sa naučíme veľa o fungovaní HDRP kanála, multiplatformového vývoja a vykresľovania v HLSL.

Našu implementáciu vykresľovania trblietok môžeme zabaliť ako stiahnutelný balíček a poskytnúť na Unity obchode Asset Store. Unity je populárny nástroj na tvorbu videohier, pridaním do obchodu môžeme prispieť k väčšiemu povedomiu a použitiu vykresľovania trblietok.

3.2.1 Rozšírenie HDRP kanálu

Najprv musíme zistiť, ako sa dá rozšíriť HDRP vykresľovač, aby sme mohli pridať vlastnú funkciu.



Obr. 3.1: Jednotlivé priechody pre HDRP kanál s vyznačenými vstupnými bodmi.

HDRP sa dá rozširovať viacerými spôsobmi:

- Vieme vložiť vlastné post-procesing efekty.
- Vieme vytvoriť vlastný vykresľovací priechod.
- Použijeme ShaderGraph.
- Modifikujeme zdrojový kód HDRP.

Pomocou preddefinovaných vstupných bodov na Obr. 3.1 zvýraznenými fialovou a zelenou farbou môžeme vidieť, kde môžeme vkladať vlastné efekty a priechody.

Pomocou vlastných priechodov vieme spraviť zmenu vzhľadu materiálov v scéne, zmeniť poradie, v akom Unity vykresľuje objekty a umožňuje Unity poslať vyrovnávaciu pamäť kamery do shaderov.

Tento spôsob úpravy ale pre nás nie je dostatočný, pretože my musíme meniť priamo správanie sa svetla v napríklad, prechode Odložené nasvietenie (Deferred Pass), alebo v doprednom nepriehľadnom nasvietení (Forward Pass).

ShaderGraph

ShaderGraph je vizuálny nástroj na tvorbu zložitejších efektov a materiálov. Je určený hlavne pre umelcov a používateľov, ktorí nie sú technicky zdatní na prispôsobenie tieňovania. Používateelia vizuálne spájajú uzly s rôznou funkcionalistou. Každý shader začína s hlavným uzlom, ktorý definuje výstup. Je možné vytvárať vlastné uzly pomocou API. Bohužiaľ ani tento prístup nám neponúka prístup k spôsobu osvetľovania.

My potrebujeme prístup k ešte nižšej úrovni HLSL kódu, ktorý nie je prístupný cez vstupné body ani ShaderGraph.

3.2.2 Modifikácia lokálneho HDRP

Posledná možnosť je skopírovať balíček HDRP lokálne do projektu, kde budeme vykonávať požadované zmeny. Tento postup nie je často odporúčaný, pretože je HDRP kanál moc zložitý, ale je to možné.

Musíme si spraviť kópiu HDRP balíčka, ktorú po stiahnutí cez manažér balíčkov v Unity nájdeme na našom disku v lokálnom úložisku balíčkov Unity. Skopírujeme a pridáme ju do projektu ako lokálny balíček. Nastavíme manifest aby využíval tento upravený balíček.

Takáto modifikácia nie je úplne ideálna. Vytvárame si vlastnú kópiu balíčka. Z hľadiska aktualizácie HDRP na novšiu verziu sa môže zmeniť kód ktorý sme upravili a teda nemusí byť triviálne prejsť na novú verziu. Bude potrebné preniesť uskutočnené zmeny z nášho modifikovaného balíčka HDRP. Teda v budúcnosti môže byť aktualizácia na novšiu verziu HDRP trochu nákladná, ale vhodnejší spôsob sme nenašli.

Unity Shadery a ShaderLab

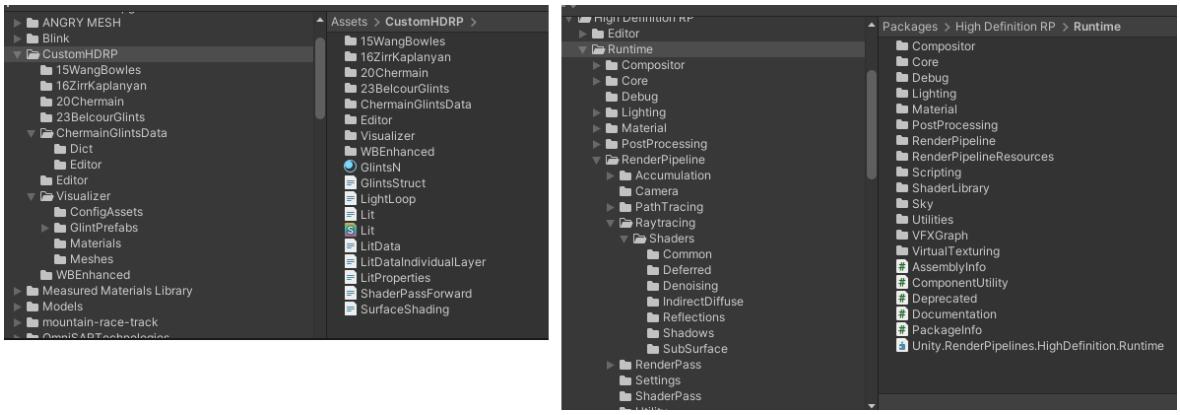
Unity je program, ktorý podporuje viacero platforem, s rôznymi hardvérovými schopnosťami. Aby dosiahlo takúto podporu, musí implementovať viacero vykresľovacích API. Zároveň pre používateľa Unity by malo byť čo najjednoduchšie pracovať s materiálmi. Nad všetkými týmito API je preto tzv. ShaderLab, shaderový jazyk špecifický pre Unity. Je to deklaratívny jazyk, definujú sa v ňom parametre, vlastnosti materiálov. ShaderLab Shader obaľuje shader programy a parametre. ShaderLab je podporovaný vo všetkých vykresľovacích kanáloch Unity, teda aj HDRP. V Shadere definujeme vlastnosti materiálu, rôzne SubShadery, pridanie vlastného editoru a aj tzv. záložný shader kód [38].

Jazyk HLSL v HDRP

Unity shadery sú písané v jazyku HLSL, pretože HLSL umožňuje lepšiu štruktúru a hierarchizáciu shader programov. Následne sú pomocou HLSL cross komplátora preložené do iných API [37]. HLSL je podobný objektovo orientovanému jazyku ako C++. Vieme oddelovať kód do viacerých súborov a vkladať dané súbory pomocou `#include` príkazu. Množstvo funkcií bolo pomocou makier odseparovaných od jednotlivých platforem. Je to z dôvodu, aby sme písali jeden kód. Unity samo za nás potom cez tieto makrá vytvorí varianty shadera pre rôzne platformy/ API. Príkladom je napríklad vzorkovanie textúr. Pri programovaní použijeme Unity makro `SAMPLE_TEXTURE2D_ARRAY_LOD`, ktoré vzorkuje zoznam 2D textúr s LOD štruktúrou. Používame pritom samostatne definovanú textúru a vzorkovač(sampler). V GLSL to nie je oddelené a používa sa iba vzorkovač.

HDRP je veľká knižnica, z ktorej budeme upravovať len niekoľko potrebných súborov, ktoré nutne potrebujeme zmeniť. Pôvodný HLSL kód teda nebudeme meniť priamo celý. Vytvoríme si kópiu pôvodných súborov. Až tú danú kópiu budeme modifikovať.

V HDRP sa nachádza viacero typov shaderov. My budeme upravovať nasvietený shader(Lit.hssl).



Obr. 3.2: Na obrázku naľavo vidíme štruktúru našej implementácie trblietok s upravenými HDRP shader súbormi. Napravo je štruktúra celého balíčka HDRP. Upravovali sme malú časť tohto balíčka, nepotrebovali sme ho kopírovať celý.

3.2.3 Tvorba vlastného shaderu

Začneme vytvorením prázdnego Unity projektu s HDRP balíčkom.

Potom vytvoríme vlastný Shader, ktorý sme nazvali CustomLit tak, že z duplikujeme Lit.hlsl a Lit.shader súbory z centrálneho repozitára balíčkov do nášho lokálneho **Assets/CustomHDRP** priečinka. Zachovávame pôvodné súbory, aby sme zachovali pôvodný Lit shader a modifikujeme ich kópiu, ktorá bude reprezentovať náš vlastný shader s pridanými metódami a inými zmenami. Takto zduplicovanému Shaderu stačí zmeniť názov z Lit na CustomLit a môžeme ho používať v editore a priraďovať materiálom. Stále totiž obsahuje direktívy, ktoré zahŕňajú využitie metód v pôvodnom balíčku HDRP.

Nepotrebujeme teda skopírovať celý HDRP balíček do projektu, postačí mať v manažérovi balíčkov (*Package Manager*) naimportovaný pôvodný HDRP balíček a časti, ktoré budeme potrebovať zmeniť nakopírujeme do projektu.

Nastavenia Unity a prvý Shader

Najskôr zmeníme niekoľko nastavení v Unity editore. Používať budeme dopredný prechod(tzv. Forward Rendering), preto v Unity nastaveniach prepneme zvolenú metódu osvetlenia z pôvodného, predvoleného, odloženého(deffered) prechodu.

Ešte vypneme vykreslovanie pomocou relatívnej pozície kamery (camera relative position rendering), ktorý pri veľmi veľkých scénach zlepšuje presnosť výpočtov tak, že všetky objekty posunie aby kamera bola vždy na súradničiach (0,0,0). Túto zmenu vieme vykonáť iba ak si spravíme tentoraz úplnú lokálnu kópiu pomocného konfiguračného balíčka pre HDRP (tzv. High Definition RP Config). V súbore *ShaderConfig.cs* potom prepneme premennú *CameraRelativeRendering* na = 0 - nebude sa používať. V budúcnosti by sme mohli toto nastavenie zapnúť ale, musíme toto nastavenie zohľadniť

pri výpočtoch prekonvertovať pozície kamery a objektov.

CustomLit Shader

Máme vytvorený vlastný Lit.shader a Lit.hlsl. Môžeme teda modifikovať kód.

Lit shader má v sebe odkazy na časti kódu umiestnené v iných súboroch, napríklad SurfaceShading.hlsl. Pokiaľ chceme vykonať zmenu v danom súbore, modifikujeme jeho kópiu. Skopírujeme z originálneho balíčka súbory do nášho projektu. Následne zmeníme pôvodné #include príkazy aby využívali kópiu.

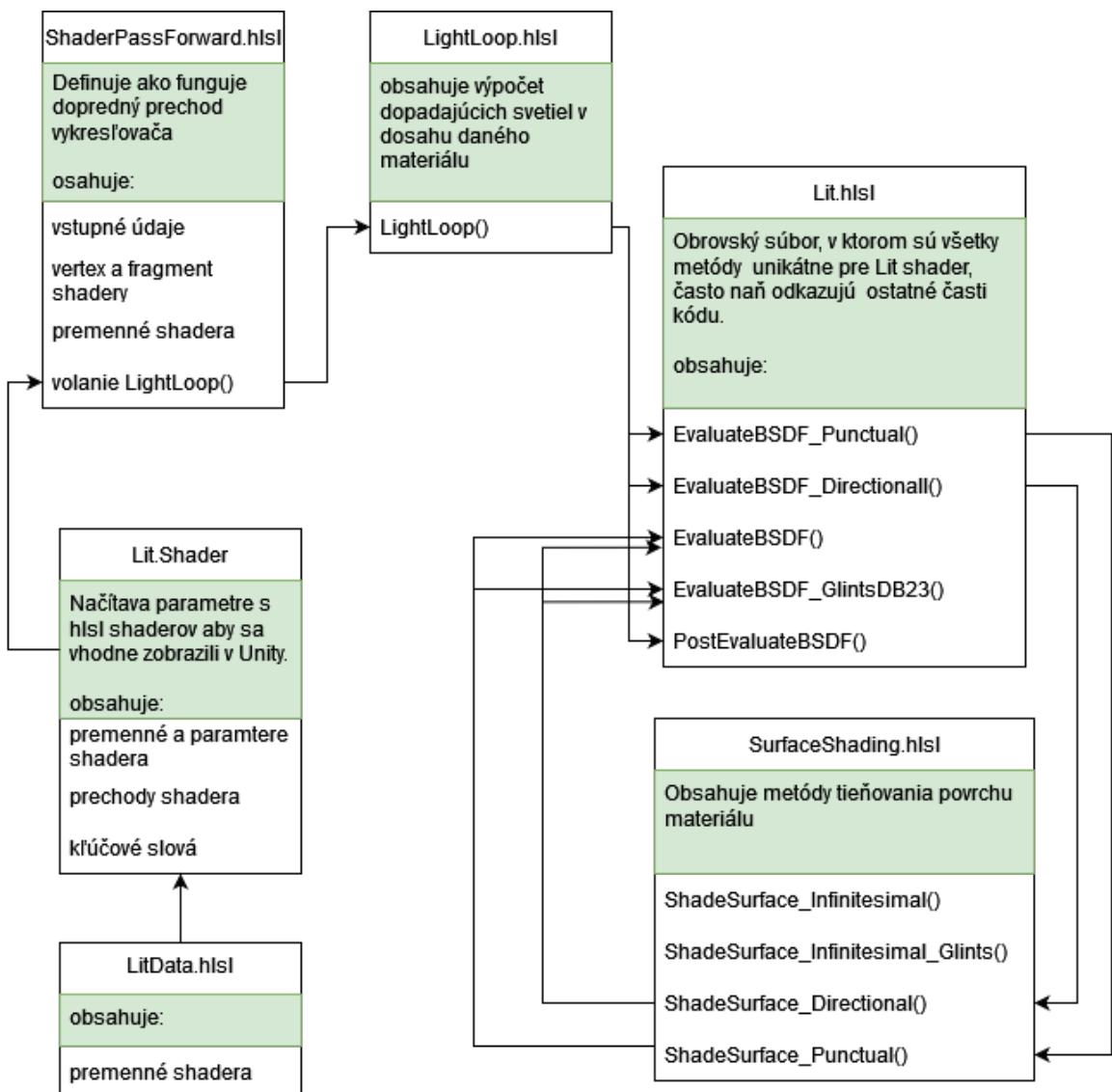
Súbory, ktoré sme modifikovali a už existovali v pôvodnom balíčku sú LightLoop.hlsl, LitData.hlsl, LitDataIndividualLayer.hlsl, LitProperties.hlsl, ShaderPassForward.hlsl, SurfaceShading.hlsl, Lit.hlsl a Lit.shader.

Lit.shader je Shaderlab kód, obsahuje názov nášho nového shaderu - CustomLit, a pridané vlastnosti trblietavých materiálov pre jednotlivé metódy. Tieto vlastnosti sú číselné, vektorové parametre, potrebné pomocné textúry a prepínače. Tieto parametre sa napájajú na definície v HLSL kóde, ktoré sú v súbore *LitProperties.hlsl*. Pokiaľ ich nedefinujeme aj v *Lit.shader* nebudú dostupné v editore pri tvorbe a úprave materiálu. Keď sú definované v *LitProperties.hlsl*, môžeme ich používať iba v ostatných súboroch, teda v kóde HLSL ale aj C# Unity skriptoch, čo nie je ideálne.

Pridali sme aj Enum na prepínanie typu metódy trblietania, vrátane možnosti bez trblietok. Pôvodne sme používali aj premennú typ materiálu (*_MaterialID*) nastavenú na "Glints na označenie, či sa majú používať trblietky, ale to spôsobovalo konflikty v neupravených súboroch HDRP(kontrolné súbory nepoznali dané ID a zbytočne vyhľadzovali chyby). Preto sme to vymenili za prepínač *_UseGlints* a typ materiálu nie je nový Glinty ale pôvodný Standard.

Nakoniec, tento súbor obsahuje direktívy s cestami na ďalšie použité súbory. Ako sme postupne modifikovali vyššie spomenuté súbory, tak sme menili tieto direktívy, aby používali našu lokálnu verziu. Definujú sa tu aj kľúčové slová, ktorými sa zapínajú vlastnosti materiálov napríklad goniachromizmus alebo aké prechody používa Shader.

ShaderPassForward.hlsl Shader je prechod dopredného vykresľovania, volajú sa tu metódy Vertex a Fragment shadery. Vo Fragment shaderi sa aplikuje finálne difúzne a zrkadlové-spekuláne osvetlenie. Nastavujú sa tu povrchové dátá, dátá pre BSDF a volá sa tu LightLoop. Funkcia LightLoop sa nachádza v *LightLoop.hlsl*. V tejto enormouskej funkcií sa počítá svetlo všetkých svetiel, počítajú sa tu tiene, environmentálne mapy a mnoho ďalšieho. Nás zaujímajú metódy, ktoré sa volajú pri osvetľovaní a to *EvaluateBSDF_Directional* a *EvaluateBSDF_Punctual*. Tieto dve metódy sa zase nachádzajú v *Lit.hlsl* a volajú funkcie *ShadeSurface_Punctual* a *ShadeSurface_Directional*.



Obr. 3.3: Podstatná časť kanálu HDRP vykresľovania pre Lit.shader ktorú budeme meniť.

Tu je presne miesto, kde musíme spraviť zmeny, aby sme správne vykreslili trblietky. Nastavuje sa tu totiž zrkadlová zložka farmy pre matariál pre každé svetlo.

Upravíme metódy *ShadeSurface_Punctual* a *ShadeSurface_Directional* tak aby kontrolovali, či je zapnutý prepínač na použitie trblietok. Vypočítame si potrebné parametre pre trblietky podľa metódy a postupne spúšťame metódu na vykreslenie trblietok. V metóde *ShadeSurface_Infinitesimal_Glints* potom vypočítame BSDF a aplikujeme výslednú farbu. Metóda *EvaluateBSDF* sa nachádza v *Lit.hsls*. Počítajú v nej Disney Diffuse BRDF, Smith Joint GGX a Fresnelová zložka. V prípade metódy od autorov Deliot a Belcour [4] nepotrebujeme priamo nahradzať spekulárnu zložku svetla, ale priamo v *EvaluateBSDF* môžeme nahradiať NDF funkciu (D zložku v BSDF) výpočtom trblietok.

3.2.4 Tvorba vlastného editora na parametre trblietok

Unity v dokumentácii [32] uvádzá, ako sa má vytvárať vlastný editor-inšpektor pre upravené materiály. Ukazujú ako spraviť upravený editor pre Lit shader pomocou vlastnej UIBlock triedy, ktorá dedí z triedy *MaterialUIBlock*. Tu sa nachádzajú metódy *LoadMaterialProperties* a *OnGUI*, ktoré preťažíme a načítame svoje upravené parametre trblietok. Všetky tieto parametre vložíme do rozkladacieho bloku. Keď už tento UIBlock máme, je potrebné vytvoriť triedu *CustomLitMaterialInspector*, ktorá v konštruktore načíta všetky požadované *MaterialUIBlock-y*. Originálny Lit.shader má editor v triede *LitGUI*. Keď sme ale dedili z tejto triedy a upravovali ju, nefungovalo to. Unity z nejakého dôvodu v inšpektore vypísalo všetky parametre materiálu, vrátane pôvodných aj tých našich a nebolo to prehľadné. Dedením z triedy *LightingShaderGraphGUI*, ktorá je používaná pre všetky ShaderGraph shadery využívajúce Lit shader sme dosiahli želaný výsledok. Aj keď ShaderGraph nepoužívame.

Posledný krok bolo pridať pôvodné *MaterialUIBlock-y*, ktoré sme museli na začiatku odstrániť, aby sa tam nenachádzali duplikovane. Všimnime si v 3.1, že *AdvancedOptionsUIBlock* nebolo potrebné pridávať. To preto, že sa tam nachádzal aj keď sme predtým všetky odstránili. Tento postup je sice mätúci, ale tak to funguje, ako má. Máme začiarkavacie poličko, ktorým povieme shaderu či chceme používať trblietky, rozbalovaciu ponuku na výber metódy trblietok a k vybranej metóde všetky relevantné parametre, vhodne pomenované, s rozsahmi hodnôt(ak sú potrebné) a priraditeľnými textúrami. Na obrázku 3.4 môžeme vidieť editor materiálu s CustomLit Shaderom.

V prílohe F je aj ukážka druhej časti editora s triedou *GlintsMethodUIBlock*.

```

1 public class CustomLitMaterialInspector : LightingShaderGraphGUI
2 {
3     public CustomLitMaterialInspector()

```

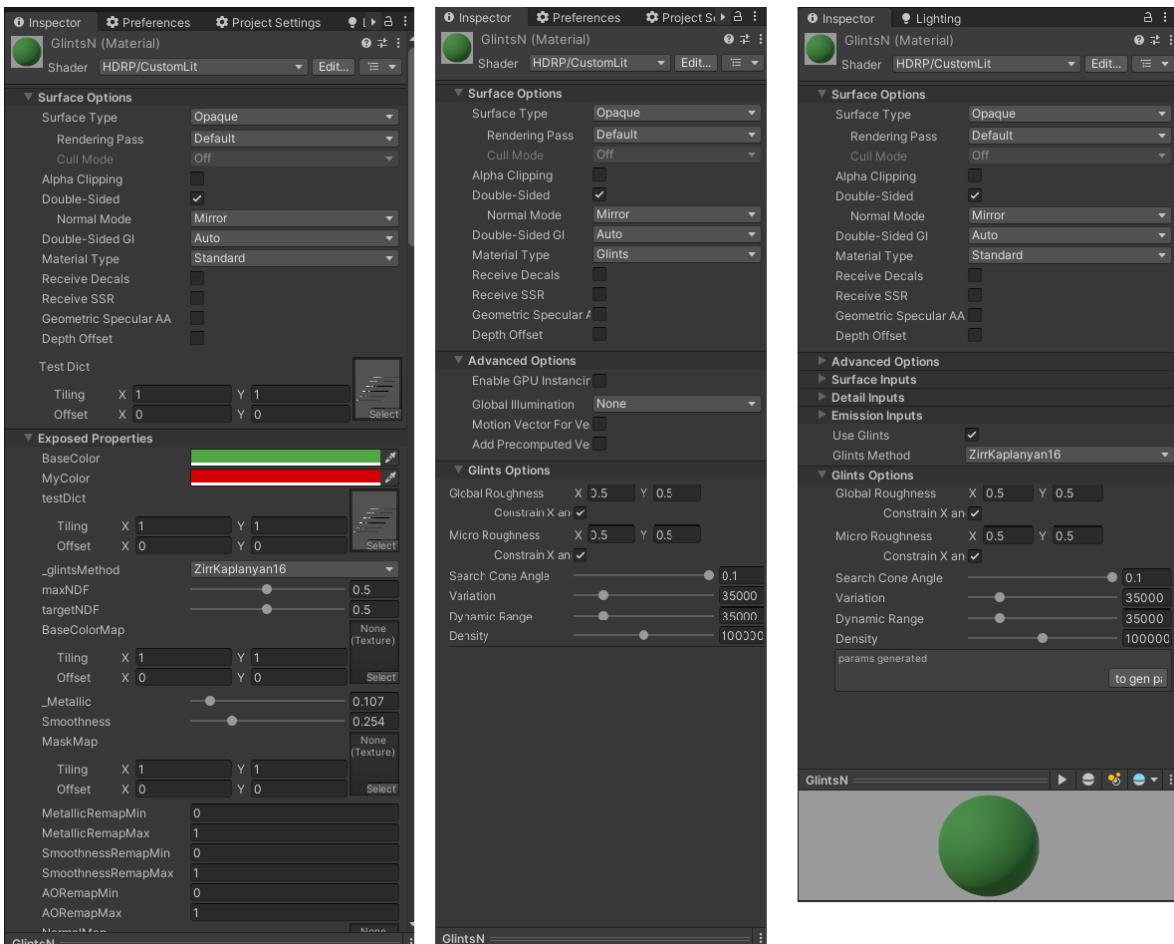
```

4  {
5      // Remove the ShaderGraphUIBlock to avoid having duplicated properties in the UI.
6      uiBlocks.RemoveAll(b => b is ShaderGraphUIBlock);
7      // Now we need to recreate missing blocks
8      const LitSurfaceInputsUIBlock.Features litSurfaceFeatures =
9          LitSurfaceInputsUIBlock.Features.All ^
10         LitSurfaceInputsUIBlock.Features.HeightMap ^
11         LitSurfaceInputsUIBlock.Features.LayerOptions;
12
13     var uiBlocks2 = new MaterialUIBlockList
14     {
15
16         new TessellationOptionsUIBlock(MaterialUIBlock.ExpandableBit.Tessellation),
17         new LitSurfaceInputsUIBlock(MaterialUIBlock.ExpandableBit.Input, features:
18             litSurfaceFeatures),
19         new DetailInputsUIBlock(MaterialUIBlock.ExpandableBit.Detail),
20         new TransparencyUIBlock(MaterialUIBlock.ExpandableBit.Transparency,
21             TransparencyUIBlock.Features.All & ~TransparencyUIBlock.Features.
22             Distortion),
23         new EmissionUIBlock(MaterialUIBlock.ExpandableBit.Emissive)
24
25     };
26     uiBlocks.AddRange(uiBlocks2);
27     MaterialUIBlock gMat = new GlintsMethodUIBlock(MaterialUIBlock.ExpandableBit.
28         User0);
29     uiBlocks.Add(gMat);
30 }
31 }

```

Kód 3.1: Vlastný editor 1/2

Aby sme v inšpektore zobrazili 2 zložkový vektor, musíme vytvoriť vlastný *MaterialPropertyDrawer*, ktorý sme nazvali *ShowAsVector2Drawer*. Vďaka tomuto “šuflíku” vieme zobraziť len 2 zložky zo štvorzložkového vektora (ShaderLab podporuje iba Vector4 vektory) ktoré reálne používame. Vieme napríklad pridať prepínač na kontrolu oboch zložiek vektora duplikovaním hodnoty x do y a zjednodušiť tým nastavovanie hodnôt.



Obr. 3.4: Vlastný editor maériálu s trblietkami *CustomLitMaterialInspector* ked' sme pridali parametre trblietok s dedením z LitGUI (naľavo), pridaním vlastného bloku s dedením z LightingShaderGraphGUI (v strede) a opravným pridaním stratených parametrov (vpravo).

3.3 Implementácia existujúcich metód vykresľovania trblietok

Implementácia existujúcich metód prebiehala vo viacerých fázach. Autori metód poskytli zdrojové kódy na svojich weboch. Spočiatku sme sa s metódami zoznámili a to tak, že čo najrýchlejšie ich spustiť, preskúmať čo robia parametre. Na to sme využili náš existujúci projekt-vykresľovač v OpenGL. Spravili sme to z toho dôvodu, že Unity je veľký softvér a spočiatku sme nevedeli kde máme robiť zmeny, navyše komplikácia shaderov zaberala podstatný čas, čo znižovalo čas našej iterácie, pokiaľ sme hľadali chyby. Navyše, shadery sa ľahko debugujú/ladia a v Unity je to ešte o niečo zložitejšie, pretože je potrebné písat veľa kódu (tzv. boilerplate kód). Takto vieme lepšie pochopiť dané metódy a upraviť ich.

Metódu od Chermain et al. [19] a metódu Zirr a Kalanyan [47] bolo celkom priamočiare použiť v našom OpenGL vykresľovači. Autori ich priamo písali v OpenGL. Zirr metóda sa dá vyskúšať i v prehliadači pomocou ShaderToy [48]. Pri týchto metódach bolo potrebné spraviť zopár zmien aby sme ho adaptovali do nášho programu.

Metódu Wang a Bowles [43] sme preložili do OpenGL, odstránili niekoľko chýb.

Metóda autorov Deliot a Belcour bola písaná v Unity, teda v HLSL, preložiť do GLSL bolo relatívne náročné. Bolo treba dbať na rozdielnu funkcionality GLSL a HLSL, napríklad, v GLSL sú matice reprezentované ako stĺpcové a v HLSL sú riadkové. Niektoré metódy v GLSL sa ani nenachádzali. Napríklad vybaľovanie 16bit floatov z 32bit hodnoty pomocou byte posunov, pre 4 zložkový vektor naraz.

Ked' sme boli komfortní s metódami v OpenGL, pridali sme ich do Unity. Museli sme dbať na to ako funguje HLSL a špecifické veci pre Unity. Potrebovali sme zistiť, kde je najvhodnejšie miesto na zapojenie trblietok. *SurfaceShading.hsls* je miesto, z ktorého voláme väčšinu metód na vykresľovanie trblietok. metóda Deliot a Belcour [4] sa používa v *Lit.shader* - *EvaluateBSDFGlints()*, pretože priamo modifikujeme funkciu *D* vo výpočte BSDF.

Samostatné metódy trblietania sú v samostatných súboroch. Pomocné metódy alebo definície štruktúr sú tiež oddelené v samostatných súboroch.

3.4 Návrh a implementácia vlastnej metódy vykresľovania trblietok

3.4.1 Návrh

Našim cieľom bolo vytvoriť trblietavý materiál. Ideálne by bolo dosiahnuť dobrú vizuálnu kvalitu a zároveň nízku hardvérovú náročnosť. Vizuálne by mali na povrchu byť

viditeľné trblietky, ktoré pri pohybe kamery alebo meniacom sa uhle dopadu svetla na objekt sa zhasínali a rozsvecovali jednotlivé trblietky. Malo by to zároveň byť koherentné, teda nemali by silno prepínať pri malej zmene uhla a vytvárať tak rušivý, chvejúci sa obraz.

S poznatkom ako fungujúce existujúce metódy, sme sa rozhodli, že vytvoríme empirickú metódu. Nebudeme sa teda snažiť štatisticky modelovať veľké množstvo trblietok, a riešiť problémy ktoré zapríčiníme prílišným zjednodušovaním náročnej techniky.

Rozhodli sme sa rozšíriť metódu Wang a Bowles [43], ktorá celkom efektívne vytvárala riedke trblietky bez preblikávania - mihotania.

Trblietky sú generované na rovnomernej 3D mriežke na ktorej sú poukladané malé guľky, ktoré sa namapujú na povrch materiálu. Veľkosť a hustota mriežky záleží od vzdialenosť od kamery. Pridáme niekoľko vylepšení, tak aby sme vedeli reprezentovať hustejšie trblietanie a priblížiť sa výsledkom najmodernejšej metódy Deliot a Bencour [4].

Hustotu trblietok sme museli mať nízku, pretože sa ináč začali prejavovať rôzne nepríjemné vzory. Na odstránenie tohto problému sme použili viacero rovnomených mriežok. používateľ vie nastaviť koľko ďalších mriežok sa má vygenerovať a ako moc sa majú ich stredy náhodne posunúť, aby sa trblietky neprekryvali. Týmto spôsobom vieme vytvoriť materiál s hustejšími trblietkami bez zjavného vzoru. Najlepšie to vidno na objektoch ako rovná plocha alebo guľa. Oproti pôvodným trblietkam ale počítame viac krát mriežku, preto je i náročnejšia na výpočet.

Ďalším vylepšením je pridanie škál. Trblietke sa pri výpočte určí škála podľa vzdialenosť od kamery. Následne sa zmieša s o úroveň vyššou a nižšou reprezentáciou. Toto vylepšenie pridáva ďalšiu variáciu, ktorá nám umožní ešte viac zhustiť mriežku.

Pôvodné trblietky sú vždy biele. Upravili sme ich, aby farba trblietania závisela od farby svetla.

Posledným vylepšením je pridanie vlastnosti globálnej drsnosti povrchu. V ostatných metódach bolo možné kontrolovať v akej oblasti sa trblietky vyskytujú, presnejšie zrkadlového zosvetlenia. Pridali sme model anizotropného spekulárneho osvetlenia Ward [44], ktorým definujeme ako sa správa zrkadlový odraz.

3.4.2 Implementácia

Celý kód metódy sa nachádza v prílohe A. Uvedieme pseudokód.

```
1 float3 GlintFade()
2 {
3     float level = CalculateLevelBasedOnDistance();
4     sparkleGridDensity *= level;
5     noiseDensity *= level;
6
7     float3 randomPosition = randomVec3(vObjPos.xy, inoise);
```

```

8     float3 positionPlusView = objPos * sparkleGridDensity + wbeStruct.viewAmount *
9         normalize(warpedViewDir + randomPosition);
10
11    float3 gridCenter = GenerateGridCenter();
12    float3 newGridOffset = positionPlusView - floor(positionPlusView);
13    newGridOffset += JitterGrid();
14    newGridOffset = Anisotropy();
15
16    return float3(CalculateFinalContribution());
17}
18 float3 CalcGrid(int i)
19 {
20     float level0 = -1.0f;
21     float level1 = 0.0f;
22     float level2 = 1.0f;
23
24     float3 resultC = GlintFade(level1);
25     if(useScales)
26         resultC += GlintFade(level0)+glintFade(level2);
27 }
28 float3 WBEnhancedGlints(float3 vObjPos, float3 vNormal, float3 lightPos, float3 vViewVec,
29                          float3 tangentVS, WBEstruct wbeStruct)
30 {
31     float3 glittering = float3(0.0f, 0.0f, 0.0f);
32     float specularity = WardAniso(ldir, vNormal, -n_view_dir, tangentVS);
33
34     UNITY_LOOP for (int i = 1; 0 < _wbGridAmount; i++)
35     {
36         glittering += CalcGrid(i);
37     }
38     return glittering * specularity;
39 }

```

Kód 3.2: Pseudokód vlastnej metódy

Pridali sme Ward spekulárny model, aby sme vedeli ovládať globálnu spekulárnosť materiálu.

```

1 float WardAniso(float3 L, float3 N, float3 V, float3 tangentVS)
2 {
3     float alphaX = _wbRoughness.x;
4     float alphaY = _wbRoughness.y;
5     float ro = 0.045; //we could expose this as param too
6
7     float NdotV = dot(N, V);
8     float NdotL = dot(N, L);
9     float3 halfV = normalize(L + V);
10
11    float3 binormal = cross(N, tangentVS);
12
13    float HdotT = dot(halfV, tangentVS);
14    float HdotB = dot(halfV, binormal);
15    float HdotN = dot(halfV, N);
16
17    if (NdotL <= 0.0 || NdotV <= 0.0)
18        return 0;
19
20    float a = sqrt(max(0.0, ro * NdotL / NdotV));
21    float b = 1 / (4 * 3.14159 * alphaX * alphaY);

```

```

22     b = max(0.0, b * NdotL);
23
24     float t1 = HdotT / alphaX;
25     float t2 = HdotB / alphaY;
26
27     float term = -2.0 * ((t1 * t1 + t2 * t2) / (1 + HdotN));
28     float e = 2.71828;
29     float c = pow(e, term);
30
31     // combine to get the final spec factor
32     return a * b * c;
33 }
```

Kód 3.3: Ward Anizotrpny odlesk

Generovanie šumu

Na generovanie šumu používame perlinov 3D šum. Mohli sme si vybrať medzi 2mi spôsobmi: vygenerovať šum do 3D Textúry, ktorú budeme potom vzorkovať, alebo priamo pri behu programu generovať šum. Zvolili sme druhý spôsob, aj keď to spomalí výpočet pri hustých mriežkach. Skúšali sme použiť aj dopredu vygenerovanú 3D textúru. Prvý spôsob sme sa rozhodli nepoužiť, pri vzorkovaní textúry sa šum neprenášal správne. Je to pravdepodobne nesprávnym použitím vzorkovacieho Unity makra, ale kvôli nedostatočnej dokumentácii sme sa rozhodli radšej zostať pri evaluácii šumu počas vykreslovania. Toto riešenie je samozrejme výpočtovo náročnejšie, čo ukážeme aj v nasledujúcej kapitole.

Na výpočet šumu sme použili open source implementáciu *FastNoiseLite* [27] respektíve môžeme použiť i implementáciu klasickej metódy vytvorenej Stefanom Gustavsonom [14]. Tieto metódy vieme použiť aj na vygenerovanie 3D Textúry šumu.

3.5 Tvorba aplikácie na porovnávanie metód

Na porovnávanie jednotlivých metód a ich modifikateľnosti pomocou dodaných parametrov sme vytvorili aplikáciu na porovnávanie. Návod a link na aplikáciu sú v prílohe B. Používateľ vie vedľa seba zobraziť naraz dva rovnaké objekty, ktorým môže zmeniť parametre materiálu. Používateľské rozhranie obsahuje výber metód trblietania, voľbu, či chceme mať na obrazovke iba jeden objekt alebo 2 vedľa seba. Používateľ môže zadávať parametre a pozorovať správanie sa metód. Vieme zapnúť pomalé otáčanie objektu a ovládať priblíženie a rotáciu kamery pomocou tlačidiel a koliečka myši. Súčasťou vykreslovača je počítadlo snímkov v pravom hornom rohu. Zobrazuje priemerné, aktuálne, minimálne a maximálne snímky za sekundu a čas v ms.

Materiál obsahuje veľa premenných ktoré chceme dovoliť upraviť používateľovi. Bohužiaľ Unity neposkytuje také dobré nástroje na ovládanie parametrov materiálov ako je to možné spraviť cez Editor.

Najprv musíme rozumným spôsobom získať všetky relevantné parametre. V ShaderLab shaderoch sú definované rôzne premenné, ktoré sú ale v mennom priestore UnityEditor. To znamená, že pokiaľ ich nevytiahneme do vlastnej reprezentácie, nebudú dostupné počas behu programu, respektíve sa s nimi bude veľmi ťažko pracovať.

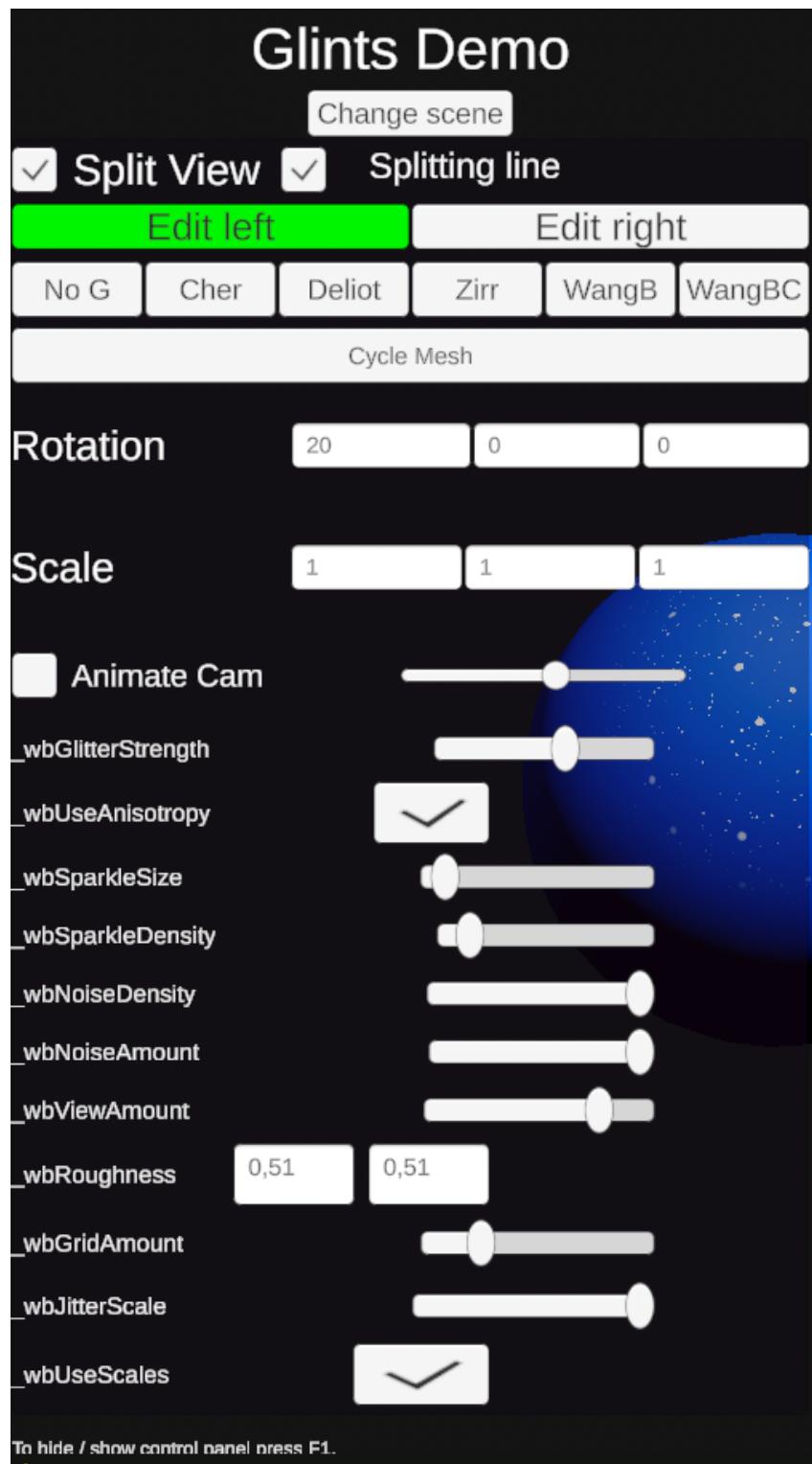
Vytvoríme si Scriptable Object, špeciálny Unity objekt, určený na ukladanie dát, ktorý môžeme potom použiť vo finálnej aplikácii. V tomto objekte, nazvanom *MaterialPropertyData* máme zoznam tried typu *SerializedGlintsMaterialProperty*. Obsahujú potrebné parametre, ktoréj metóde patria, akého sú typu, jej meno, ktoré použijeme na zmenu daného parametra, zobrazovaný názov, ktorý je v čitateľnejšej forme, 2 zložkový vektor rozsah pre float a následne premenné ako int, float, 4 zložkový vektor. Triedu *GlintsMethodUIBlock* rozšírimo tak, aby pri zobrazovaní metódy (pomocou *materialEditor.ShaderProperty(matProp, name)*) vložila tento parameter do zoznamu potrebných parametrov typu *SerializedGlintsMaterialProperty*. Tento list sa následne uloží na disk ako .asset súbor, ktorý môžeme používať v iných častiach programu a nemá závislosť na *UnityEditor* mennom priestore. Nevýhodou je, že nemôžeme zistiť, či je napríklad premenná float typu prepínač, podobne či je to 2 zložkový vektor. Po každom vytvorení .asset súboru budeme musieť manuálne skontrolovať a zaškrtnúť prepínač, určujúci využitie float premennej na prepínanie. Keďže využívame iba 2 zložkové vektory, túto informáciu, koľko zložiek potrebujeme si neukladáme.

Skript *ObjectViewer.cs* sa stará o správne nastavenie porovnávača. Pri zapnutí sa načíta .assset, dokončí sa UI pridaním trblietavých parametrov materiálu a zo scény sa načítajú potrebné informácie ako kamery určené na vykreslovanie ľavej strany resp. pravej.

Rozhranie UI, ktoré je dostupné pri samostatnej aplikácii nie je úplne ideálne z hľadiska používateľskej prívetivosti. Napríklad, do políčok, kde majú ísť iba čísla sa môže napísať aj text s písmenami. To nepokazí program, je to ošetrené. Dôležitejšie je, že pri práci v Editore inšpektor správne zobrazuje tieto premenné a je užívateľsky prívetivé na úpravu materiálov.

3.6 Stiahnutelné demo a shader

Vytvorili sme samostatnú aplikáciu pre Windows v ktorej môžeme porovnávať jednotlivé metódy. K aplikácii sme vytvorili návod - viď. príloha B. V prílohe B sa nachádza aj



Obr. 3.5: UI rozhranie pre samostatnú aplikáciu, kde vieme porovnávať jednotlivé metódy a zároveň ovládať ich parametre.

link na stiahnutie balíčka. V deme sme používali na vizualizáciu aj modely od rôznych autorov. Všetky cudzie použité assety, sme vypísali v prílohe E.

CustomLit Shader sme exportovali ako stiahnuťelný balíček. Jeho vloženie do Unity je jednoduché ale je potrebné predtým spraviť pár krokov naviac. Preto sme vytvorili návod ako vytvoriť Unity projekt a vložiť tam balíček. Návod je dostupný v prílohe C. V prílohe C sa nachádza aj link na stiahnutie aplikácie.

3.7 Práca s Unity HDRP

Na záver implementácie by sme chceli zhrnúť prácu v Unity a naše skúsenosti s vykresľovacím balíčkom HDRP.

Ked' Unity Technologies oznámilo vytvorenie Skriptovateľného vykresľovacieho kanála (SRP) v Unity 2018.1 [3], v ktorých boli vytvorené tieto HDRP a URP kanály, ohlasy boli zmiešané. Otázne bolo ako sa podarí spoločnosti udržiavať 3 odlišné vykresľovacie kanály, vrátane zabudovaného (BRP). Od zverejnenia HDRP uplynulo 6 rokov a medzitým bolo vydaných 17 hlavných verzií.

Jednotlivé verzie HDRP si vyžadujú konkrétnu minimálnu verziu Unity tak ako bola pridávaná funkcionálita do jadra softvéru. URP a HDRP sa čím ďalej tým viac vzdaľovali implementáciou a funkciami. Veľa grafických efektov aj tak pridávali do HDRP aj URP, striedavo s rôznymi rozostupmi do oboch balíčkov. Unity Technologies ani v dokumentácii neodporúčajú modifikovať priamo kód HDRP ale využiť ShaderGraph. URP má lepšiu podporu modifikácie kódu. URP je pre vývojárov vhodnejší kanál, pretože je menej náročný na hardvér a umožňuje malým tímom vývojárov vytvárať mobilné a VR hry, kde je efektivita veľmi dôležitá.

HDRP má dosahovať vysokú vizuálnu kvalitu, ale vývojárov pravdepodobne nepresvedčili a tí radšej volia konkurenciu, ako Unreal Engine 5.x.

Nedávno vydaná predbežná verzia Unity 6 (pôvodne mala mať názov 2023.3) priniesla niekoľko vylepšení v HDRP, ale najviac noviniek dostane kanál URP [24]. Týmto potvrdzujú naše obavy, že HDRP upadá do úzadia.

Tu je niekoľko problémov na ktoré sme narazili pri tvorbe projektu:

- Pády aplikácie: Používame verziu Unity 2022.1.5, čo nie je dlhodobo podporovaná verzia LTS. 2022.3 LTS vydali až po začiatku tvorby práce a rozhodli sme sa neaktualizovať verziu pre potenciálne problémy s HDRP.

- Zložitosť kódu: Ako je viditeľné aj z Obr. 3.3, tak v kóde sa neorientovalo ľahko, veľa vecí bolo medzi sebou poprepájaných. Modifikácie spočiatku boli celkom náročné.
- Čas kompliacie: Čas kompliacie pri každej zmene kódu sa zväčšoval až kým sme niekedy nečakali až 10 minút. Spravili sme potom niekoľko úprav aby sme zrýchlili výpočet, napríklad vypnutím nepoužívaných funkcionálít shadera. Stále to ale dostatočne neznížilo náš iteračný čas. Preto sme často iterovali v našom OpenGL vykresľovači a potom prenesli už hotové zmeny do HLSL.

Niekedy sa stalo, že sa proces komplikovania prosté pokazil. Najjednoduchšie riešenie vtedy bolo uložiť spravené zmeny, vrátiť vykonané zmeny pomocou gitu a nechať Unity znova načítať projekt. Ak to nepomohlo, ešte sme museli vymazať *Library* priečinok ktorý slúži ako cache.

- Chyby HDRP: Vo verzii HDRP 13.1 sme narazili na niekoľko nepríjemných problémov. Ich odstránenie niekedy nebolo triviálne alebo sa odrazu prestali vyskytovať. Raz sa nám stalo, že trblietavý materiál so správnym shaderom sa úplne prestal zobrazovať, ako v Editore tak aj v samostatnej aplikácii. Doteraz nevieme ako sa nám podarilo vrátiť ich, ale skúšali sme toho veľa.

Iný problém bol s nefunkčnými shadermi ShaderGraph v samostatnej aplikácii. Tento problém sa nám nepodarilo vyriešiť. V editore všetko fungovalo. Z toho dôvodu sme z finálnej verzie odstránili napríklad stromy, ktoré boli, naštastie, len na skrášlenie prostredia.

- Inšpektor s vlastným editorom materiálu sa pokazil a museli sme ho trochu šíkovne obíť a zdelenie ShaderGraph Editor, ako sme ukázali v Obr. 3.4.

Kapitola 4

Výskum a výsledky

V tejto kapitole ukážeme vykreslenie niektorých materiálov s použitím implementovaných metód trblietania. Vykreslíme ich na viacerých modeloch s rôznymi svetelnými podmienkami. Porovnáme ich z hľadiska výpočtovej náročnosti a vizuálnej kvality. Zhrnieme výhody a nevýhody jednotlivých metód, kde fungujú najlepšie a čo je ešte potrebné vylepšiť.

4.1 Metodika evaluácie

V predchádzajúcej kapitole sme implementovali metódu trblietania a už 4 existujúce metódy. V tejto kapitole ich porovnáme na základe viacerých kritérií. Na evaluáciu potrebujeme mať vhodnú metodiku. Najdôležitejšie je vizuálna kvalita a efektivita.

Prostredie evaluácie

Základné rozlíšenie sme zvolili 4K rozlíšenie, ktoré má $3840 \times 2160 = 8\ 294\ 400$ pixelov. Toto rozlíšenie sme zvolili aby GPU nebolo obmedzované CPU, hlavne v situáciách keď má GPU ľahkú záťaž sa môže stať, že hlavné vlákno na CPU sa stane obmedzením. Na evaluáciu použijeme notebook s procesorom i7-9750H a grafikou NVIDIA GTX 1660ti Laptop. Grafika je porovnatelná s desktopovou GTX 1060, ktorá je v dnešnej dobe radená medzi hardvér nízkej triedy v rámci výkonu. Tento hardvér je z roku 2020. Moderné desktopové čipy od Nvidie dosahujú násobne väčší výkon. Evaluovali sme v Unity editore pomocou interných nástrojov.

4.1.1 Metodika evaluácie výkonu

Unity má robustné nástroje na profilovanie aplikácie. Použijeme 2 nástroje a to *Profiler* a *Profile analyzer*. Na obrázku 4.1 môžeme vidieť ako taká evaluácia prebiehala. *Profiler*, na obrázku vľavo, zaznamenáva výkon aplikácie a to vo viacerých moduloch.

Nás zaujíma hlavne modul využitia GPU. Nepoužívame nič, čo by zbytočne zaťažovalo CPU, pretože trblietky sú v shaderi výlučne spúštané na GPU. Vpravo dole vidíme *Profile analyzer*. V ňom vieme získať zvolením viacerých snímok strednú hodnotu a priemernú hodnotu, v milisekundách. Milisekundy sú všeobecne lepšia metrika na porovnávanie výkonu oproti snímkam za sekundu. Dôležitá je aj konzistentnosť snímkovania, teda ako moc v čase sa mení čas vykreslenia 1 snímky. Tieto nástroje majú určité pridané režijné náklady, ale to platí pre všetky scenáre, takže sme konzistentní. *Profiler* má sice oddelenú verziu, ktorá beží oddelene od hlavného vlákna Unity, ale ten sme nepoužívali, pretože neumožňoval export dát do *Profile analyzer-a* a takmer vôbec nechcelo zachytávať dátu v module využitie GPU, ktoré bolo pre nás kľúčové.

Existuje ešte nástroj *Frame debugger*, ktorý nám umožňuje pozrieť sa na jeden snímok a zistíť, čo sa približne vykonáva vo vykresľovači. Neposkytuje ale tak detailné informácie aby sme vedeli, čo presne spomaľuje vykresľovanie. Ladenie a schopnosť získať presný obraz o tom čo presne GPU vykonáva je bežný problém vykresľovania na GPU. Nevieme napríklad krokovat kód ako pri bežných programoch. Najbližšie k ladiacim nástrojom, ktoré poznáme sú RenderDoc a NVIDIA Nsight Graphics.

Limitácie evaluácie výkonu

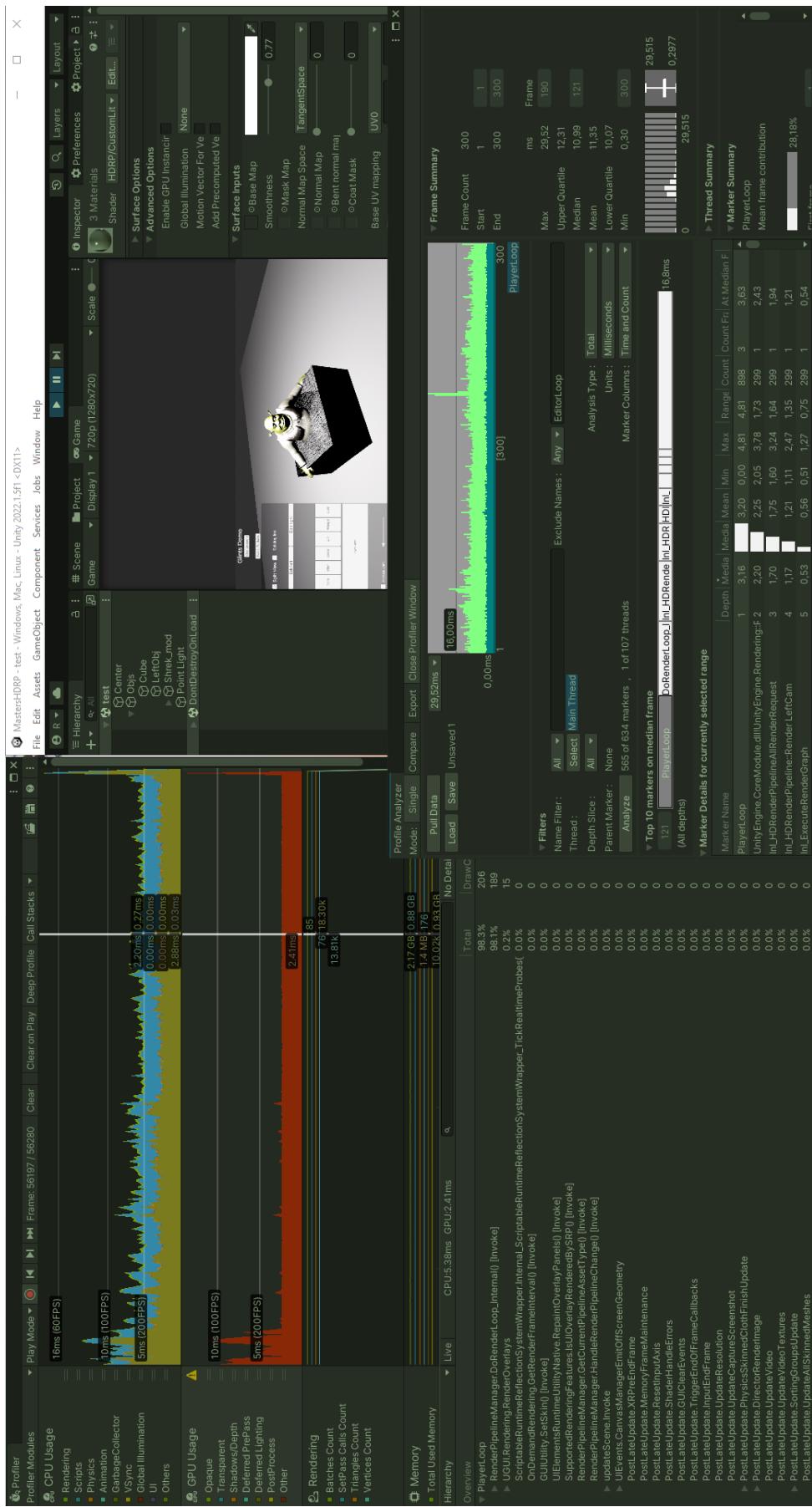
V prvých dvoch scenároch evalujeme 5 rôznych konfigurácií objektov a svetiel bez pohybu. Metódy môžu mať rôzne silné stránky, ktoré záležia od použitia parametrov. Preto použité parametre uvádzame v prílohe D.

Demo scéna s animovanou kamerou je reprezentatívnejší príklad. nachádza sa tam viacero objektov používajúcich tú istú metódu pri jednom behu evaluácie ale parametre sa mierne líšia objekt od objektu. Parametre všetkých týchto objektov nebudeme uvádzat.

4.2 Metodika evaluácie

Vyhodnocovali sme implementované metódy na základe týchto parametrov:

- Jednoduchosť použitia: Ako ľahko sa používateľovi pracuje s danou metódou, či robí to čo očakáva.
- Zložitosť implementácie: Aké veľké boli potrebné úpravy aby metóda fungovala.
- Obmedzenia: Čo nemôžeme robiť s danou metódou.
- Výkonnostná náročnosť: Aký dopad na výkon majú jednotlivé metódy a ktoré scenáre majú najväčší dopad na výkon.



Obr. 4.1: Ukážka priebehu evaluácie.

- Parametre: Aké máme parametre a čo robia. Sú parametre rozumné a ako ovplyvňujú finálny vzhľad, poprípade výkon.
- Vizuálny výsledok: Je vizuál kvalitný? Sú tam viditeľné artefakty alebo nekonsistentnosť?
- Ďalšie poznámky: Všetko čo sa nehodí do vyššie spomínaných kategórií.

Naša evaluácia bude pozostávať z viacerých scenárov. Tieto 3 scenáre boli vytvorené tak aby sme zachytili čo najviac relevantných oblastí použitia a mali kontrolu nad experimentami. Keď píšeme Lit.shader, máme na mysli neupravený Lit.shader z HDRP. Všetky metódy trblietania používajú CustomLit.shader. CustomLit.shader má aj možnosť nepoužiť trblietky, to v tabuľkách označujeme ako *CustomLit Bez*.

Pri skúmaní chceme odpozorovať, či sa nemihotajú trblietky pri pohybe kamery alebo objektu. V scenároch je používaná ambientná oklúzia a používame nastavenie grafických možností HDRP *High Fidelity*.

4.3 Evaluácia scenárov

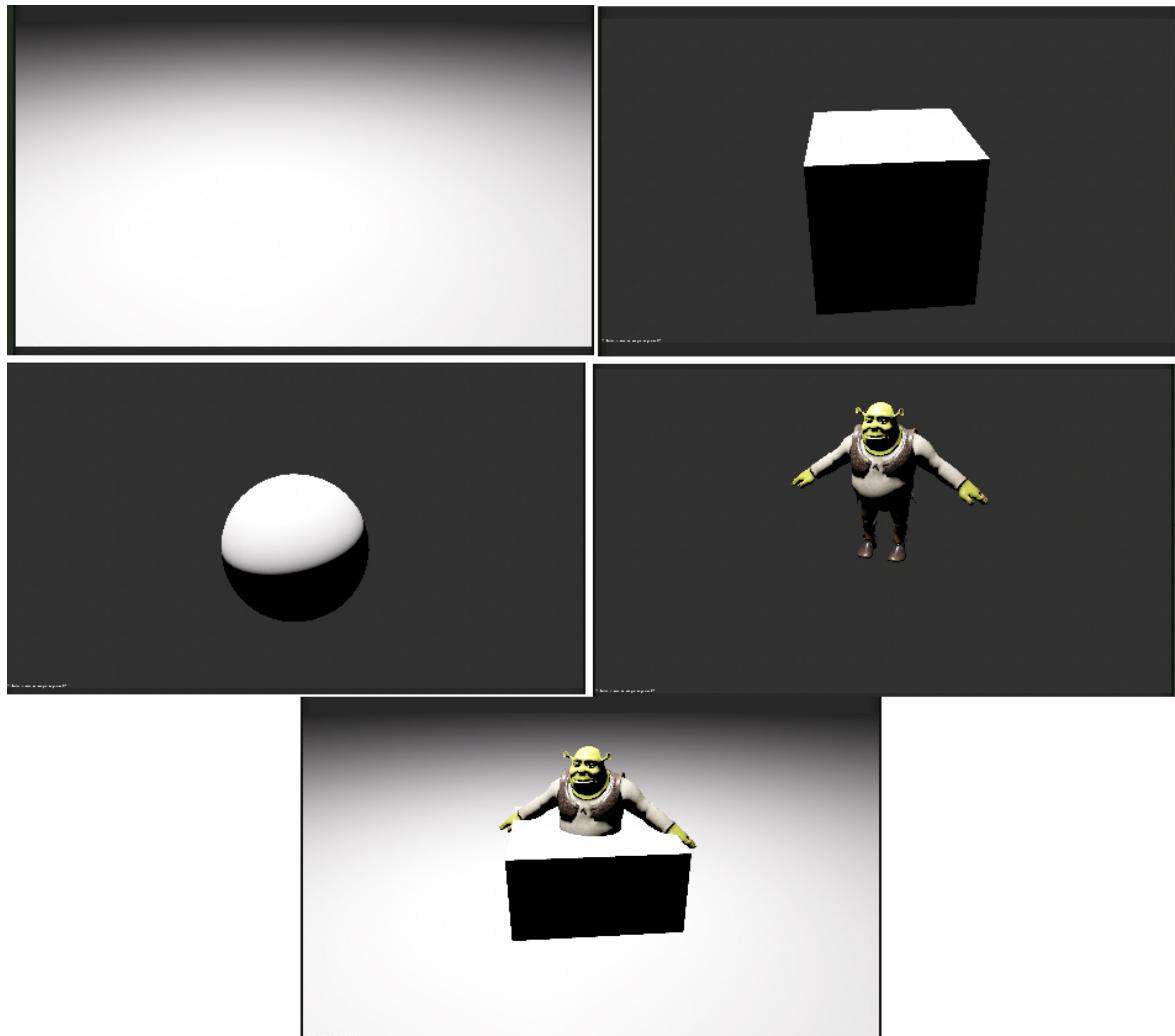
4.3.1 1. Scenár

Tento prvý scenár je jednoduchý. Máme jedno bodové svetlo s dosahom 200m(od stredu) umiestnené nad objektom. Na obrázku 4.2 sú znázornené jednotlivé snímky s použitím materiálu používajúceho štandardný Lit shader. Postupne vystriedame nasledovné objekty: rovinu, kocku, sféru a zložitejší model. Posledný, piaty objekt bude zjednotenie všetkých predchádzajúcich modelov. Spravíme to z toho dôvodu aby dochádzalo k oklúzii, nevadí ako to vyzerá.

Zaznamenávame čas na výpočet snímky v milisekundách, poprípade pridáme aj počet snímok za sekundu - FPS.

Typ objektu	Lit.shader	CustomLit Bez	Chermain	Deliot	Zirr	Wang	Naša
Rovina	12.41	16.29	41.90	49.95	39.87	19.33	25.66
Kocka	8.85	9.73	15.06	12.40	12.03	10.60	10.97
Sféra	8.63	9.37	19.70	13.65	12.09	10.11	11.09
Shrek	8.60	8.60	13.50	12.14	11.70	9.80	10.50
Všetko	12.41	16.42	50.48	50.48	50.48	50.48	50.48

Tabuľka 4.1: Porovnanie času vykresľovania (v milisekundách) pre rôzne typy modelov a trblietavých metód pre scenár č. 1.



Obr. 4.2: Modely použité pri evaluácii scanára 1 a 2.

Nárast náročnosti

V tabuľke 4.1 vidíme, že keď zvolíme náš upravený CustomLit Shader, náročnosť sa mierne zvýši, aj keď nevoláme funkcie na výpočet trblietania. Najzarážajúcejší je nárast výpočtu pri rovine. Snažili sme sa nájsť dôvod tohto spomalenia. Najväčším prírastkom bolo RenderLoop.ScheduleDraw ktoré sa zvýšilo z 2ms na 8ms. Pomocou programu RenderDoc sme identifikovali taktiež RenderLoop.ScheduleDraw a RenderLoop.DrawSRPBatcher ktoré majú najväčší nárast v ms. Toto znamená, že CustomLit shaderu trvá dlhšie vykonanie na GPU. Pri takomto nastavení by sa mal shader správať úplne totožne ako neupravený, v HLSL sa nachádza len podmienka, či je zapnutá možnosť trblietania.

Kolísanie času vykreslovania snímky

Ďalšia vec čo sme si všimli je konzistencia dodávky snímok. Na obrázku 4.3 môžeme vidieť CustomLit a Lit shadery. Ich náročnosť by mala byť totožná, pretože je nastavené nepoužívať trblietky. Napriek tomu vidíme, že väčší počet snímok používa konzistentne viac času na vykreslovanie a niekedy to klesne na viac očakávanú hodnotu. Dôvod tohto kolísania sa nám podarilo odhaliť až po vykonaní evaluácie výkonu.

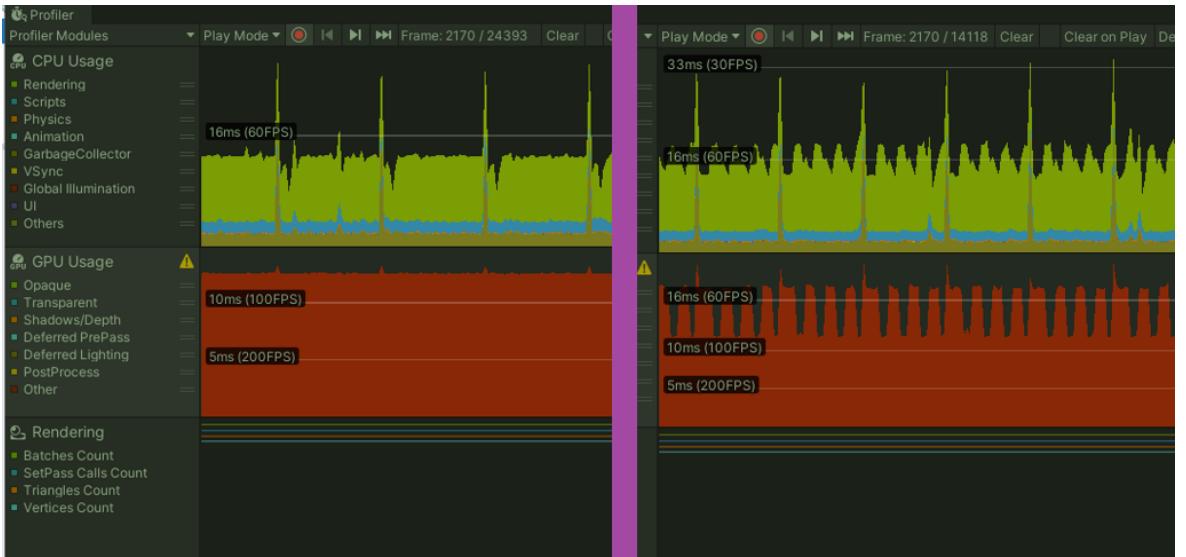
Dôvod tohto problému je dynamická vetva, ktorá kontrolovala či sa má použiť štandardná metóda `ShadeSurface_Infinitesimal` alebo `ShadeSurface_Infinitesimal_Glints`. Konkrétnie to bolo zapríčinené tzv. dynamickým vetvením. To znamená, že priamo na GPU sa testuje podmienka či sa má použiť trblietanie. Aby sme to odstránili musíme prerobiť voľbu metód podľa premennej (`_glintsMethod`). Podľa [31] ak pri dynamickom vetvení je jedna vetva oveľa zložitejšia, môže viest' k menšiemu paralelnému volaniu shader programov, čo znižuje výkon. Tento kolísavý stav sa prejavoval aj pri metódach s trblietaním. Síce sa náročnejšia vetva aj naozaj spúšťala, ale boli tam ďalšie vetvy ktoré volali správnu metódu.

4.3.2 2. Scenár

Oproti scenáru 1 sme tentoraz použili 3 bodové svetlá, tiež s dosahom 200m umiestnené vedľa seba, smerom od kamery.

Typ objektu	Lit.shader	CustomLit	Bez	Chermain	Deliot	Zirr	Wang	Naša
Rovina	13.68	17.90	107.66	106.61	84.74	21.84	44.07	
Kocka	8.85	9.83	23.93	14.87	15.13	10.71	12.06	
Sféra	8.61	9.36	15.09	13.51	13.19	10.17	11.33	
Shrek	8.49	9.01	15.03	12.22	13.76	9.83	10.95	
Všetko	14.33	20.55	114.19	98.65	85.44	22.08	42.75	

Tabuľka 4.2: Porovnanie času vykreslovania (v milisekundách) pre rôzne typy modelov a trblietavých metód pre scenár č. 2.



Obr. 4.3: Vľavo vidíme graf času vykresľovania pre Lit Shader (vľavo) a CustomLit Shadera (vpravo).

4.3.3 3. Scenár

Komplexná scéna, veľa rôznych objektov použitím jednej metódy. Táto scéna sa nachádza aj v samostatnom deme, ktorého link na stiahnutie je v prílohe B.

Aby sme lepšie ukázali aká náročná je táto scéna, pre tento scenár pridáme tabuľku aj v snímkach za sekundu.

Táto scéna obsahuje mnoho objektov. Všetky svetlá sú bodové a v reálnom čase sa počíta osvetlenie, nie je tam nič predpočítané. Svetiel je v scéne 135. Samozrejme nie všetky sú blízko kamery a Unity ich vylúči z cyklu pri počítaní osvetlenia. Počas animácie vplyva na objekty s trblietkami svetiel. V scéne je aj slabé mesačné osvetlenie-smerové svetlo. To nepridáva trblietky. 35 osobitných objektov má v jednom čase trblietavý vzhľad. Svetlá nevrhajú tiene.

Vykonáva sa priemerne 450 *drawcalls*- volanie CPU na GPU s požiadavkou vykresliť objekt. Maximum bolo v cca. 4 sekunde animácie kamery. Vytvorilo sa približne 683 *drawcall-ov*. To predstavovalo približne 204 900 trojuholníkov a približne 153 600 bodov.

Metóda	priemerné ms	min ms	max ms	1% nízke ms	0.1% nízke ms
Lit	17.2	19.8	14.1	20.2	20.7
CustomLit Bez	17.5	21.1	13.0	22.3	22.8
Chermain	47.2	65.8	28.6	67.1	67.2
Deliot	48.1	66.2	32.5	67.0	67.0
Zirr	42.0	62.8	26.7	65.2	66.5
Wang	18.5	23.7	13.5	23.7	106.8
Naša	44.8	65.8	32.4	67.0	76.6

Tabuľka 4.3: Snímková frekvencia jednotlivých metód v ms pre scenár 3.

Metóda	priemerné FPS	min FPS	max FPS	1% nízke FPS	0.1% nízke FPS
Lit	58.0	50.5	71.1	49.4	48.3
CustomLit Bez	57.1	47.3	76.6	44.8	43.8
Chermain	21.2	15.2	35.0	14.9	14.8
Deliot	20.8	15.1	30.8	14.9	14.9
Zirr	23.8	15.9	37.4	15.3	15.0
Wang	54.0	42.1	74.3	42.1	9.2
Naša	22.3	15.2	30.9	14.9	13.1

Tabuľka 4.4: Snímková frekvencia jednotlivých metód v FPS pre scenár 3.

Šum v našej implementácii

Kedže sme 3D Perlinov šum generovali priamo pri behu programu, chceli sme zistiť, aký veľký vplyv má tento spôsob na výkon. V tabuľke 4.5, vidíme, že čas potrebný na výpočet jedného snímku sa znížil o približne 43,5%. To znamená, že pridanie procedurálne počítaného šumu v tejto scéne celkom výrazne zvýšilo náročnosť.

Konizstentnosť

Naša metóda a väčšina ostatných (vid. tab. 4.3, min hodnoty) mala tendenciu kolísat s frekvenciou snímkovania. Je možné, že to súvisí s dynamickým vetvením, ako sme spomínali v 4.3.1.

Metóda	priemer	min	max	1% nízke	0.1% nízke
Naša s procedurálnym šumom	44.8	65.8	32.4	67.0	76.6
Naša bez šumu	25.3	32.4	19.5	35.2	37.9

Tabuľka 4.5: Porovnanie času vykresľovania (v milisekundách) našej metódy pri použití šumu vs bez šumu.

4.4 Vyhodnotenie

4.4.1 Metóda Chermain et al.

Jednoduchosť použitia:

Pri tejto metóde potrebujeme dodať slovník. V našej implementácii sme vytvorili jeden slovník a neponúkame spôsob ako vygenerovať detailnejší alebo menej detailný slovník. Tento slovník, ale úplne postačuje. Nejaké trblietky sa objavia aj keď slovník nepriradíme, ale budú utlačené, nesprávne vyzerajúce. Bolo bu vhodné na tento fakt upozorniť napr. v konzole.

Zložitosť implementácie:

Implementácia je mierne zložitá, je potrebné dodávať slovník predpočítaných SDF. Musíme koherentne indexovať zo slovníka aby nevznikali artefakty.

Parametre:

- Material Alpha: Spekulárna oblasť trblietok.
- Log Microfacet Density: Nastavuje hustotu trblietok. Vysoká hodnota konverguje k štandardnej NDF.
- Dict N Levels: Koľko levelov LOD máme v slovníku.
- Max Anisotropy: Limituje počet prechodov pri evaluácii stopy pixelu.
- Microfacet Relative Area: Čím vyššia, tým je distribúcia trblietok viac sústredená v strede highlightu.
- Dictionary Alpha: Cieľová drsnosť na ktorej bol vypočítaný slovník.

Obmedzenia:

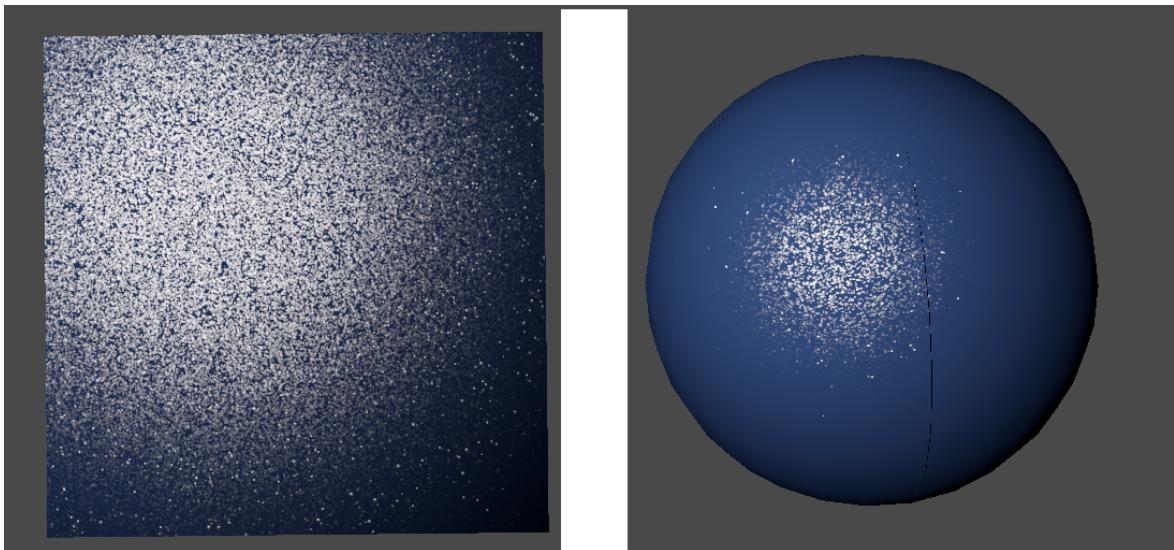
Táto metóda nemá oddeliteľný výpočet NDF, preto sa nedá použiť priamo v HDRP funkcií na výpočet BSDF zmenou parametra D. Musíme nahradíť spekulárnu zložku pri výpočte farby svetla. Ak plocha pixelu zaberá veľkú časť povrchu, musíme spočítať veľa texelov, čo spomaľuje výpočet. Limitujeme anizotropiu na 16 alebo 8 aby sa výpočet príliš nespomaľoval. Vtedy sa trblietky môžu mierne roziahnuť. Nepodporujú enviromentálne mapy ani plošné svetlá.

Výkonnostná náročnosť:

Táto metóda patrí k tým najnáročnejším, ako je aj vidieť z 4.4. Jej náročnosť sa zvyšuje podľa toho akú veľkú plochu zaberajú trblietky na obrazovke. Každé svetlo lineárne zvyšuje náročnosť.

Vizuálny výsledok:

Vizuálne výsledky sú pôsobivé, vieme si vzhľad upraviť. Nedá sa priamo nastaviť veľkosť trblietok, ale dalo by sa to dorobiť. Nevidíme žiadne veľké artefakty. Máme Je možné natiahnuť trblietky - spraviť anizotropické trblietky. Túto možnosť sme ale nepovolili v editore.



Obr. 4.4: Vizuál Chermain et al. metódy.

4.4.2 Metóda Deliot et al.

Jednoduchosť použitia:

Metóde stačí pridať predpočítanú textúru a ponechať základné parametre.

Zložitosť implementácie:

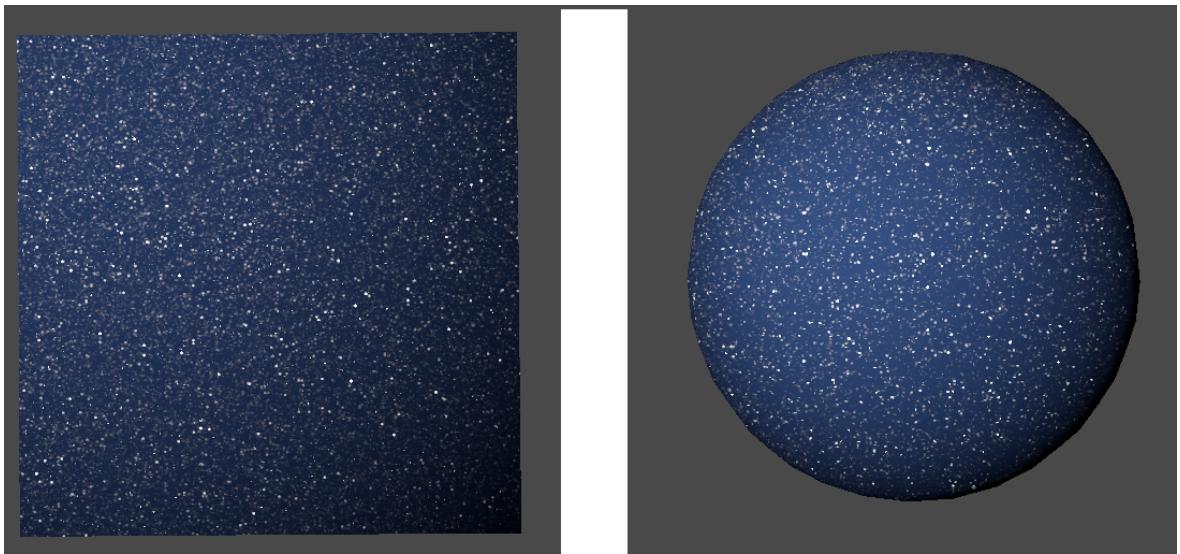
Implementácia je veľmi zložitá s niekoľkými optimalizáciami. Povrch materiálu inde-xujú do mriežky, ktorá má škálu, rotáciu a pomery. Aby nemuseli evaluovať binomický zákon medzi 10mi bodmi vypočítajú štvorsteny, do ktorých patrí daný fragment. Viac sme sa tomu venovali v časti 2.8.4.

Parametre:

- Max NDF: Použitá na preškálovanie NDF, sila odrazu trblietky.
- Target NDF: Pomer k Max NDF, koľko trblietok bude vyžarovať maximálne.
- Density Randomization: Množstvo pridanej variácie hustoty trblietok.
- Log Microfacet Density: Hustota trblietok na povrchu.
- Microfacet Roughness: Drsnosť trblietok, pri nízkej hodnote sú odrazy hladké.
- Screen Space Scale: Zväčší trblietky, Bohužiaľ ich aj rozmaže.

Obmedzenia:

Je vhodná iba na bodové svetlá. Nedá sa meniť veľkosť trblietky, je potrebné upraviť kód. Nevieme zmeniť globálnu drsnosť.



Obr. 4.5: Vizuál Deliot et al. metódy.

Výkonnostná náročnosť:

Má relatívne vysokú náročnosť, ale konzistentnejšiu, pretože napasovali pomocou parameterov, rotácií a škály stopu pixelu aby nevyžadovala viacero evaluácií.

Vizuálny výsledok:

Táto metóda vyzerá najkrajšie. Nemá ale takú variabilitu vo výzore trblietok ako iné metódy.

4.4.3 Metóda Zirr et al.

Jednoduchosť použitia:

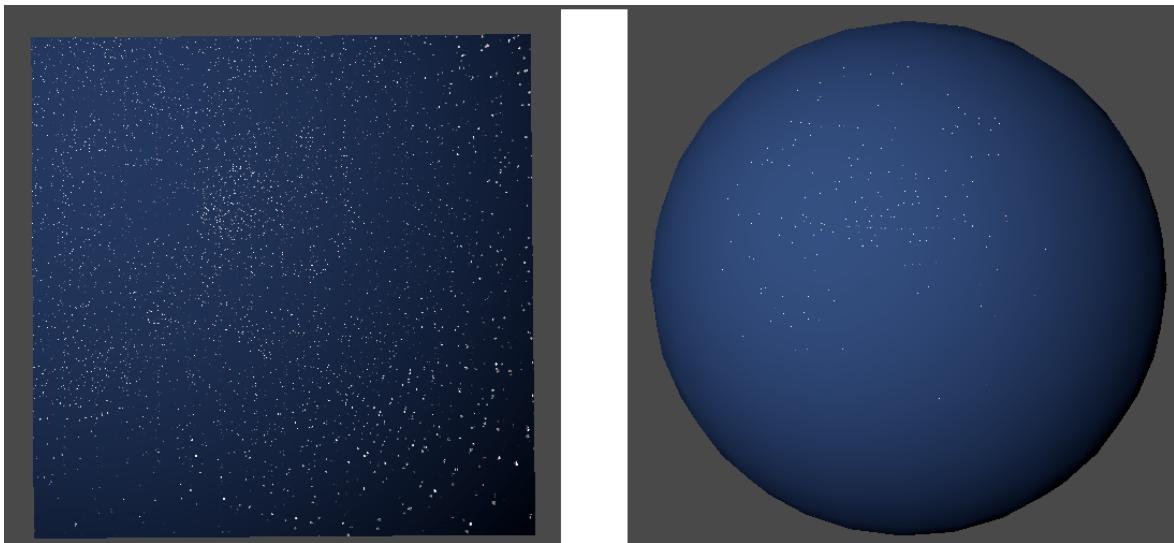
Metóda má iba 6 parametrov. Nedajú sa zväčsiť trblietky. Parameter *Micro Roughness* musí byť menší ako *Global Roughness* a nie je úplne jasné ako presne musíme nastaviť tieto vektoru.

Zložitosť implementácie:

Implementácia je relatívne zložitá.

Parametre:

- Global Roughness: Drsnosť povrchu ako celku.
- Micro Roughness: Drsnosť drobných trblietok.
- Search Cone Angle: Rozsah v ktorom sa počítajú trblietky.



Obr. 4.6: Vizuál Zirr et al. metódy.

- Variation: Metóda produkuje veľmi podobné mikrodetaily, zväčšením variácie trblietok dosiahneme prirodzenejší vzhľad.
- Dynamic Range: Kontroluje svietivosť trblietok.
- Density: Hustota trblietok.

Obmedzenia:

Limitujú anisotropické filtrovanie na 16. Preto napríklad na rovine môžeme vidieť v diaľke rozmazané trblietky. Je limitovaná na použitie oddeliteľnej NDF ako napr. Beckmann NDF, podobne ako metóda Chermain et al.

Výkonnostná náročnosť:

Metóda je relatívne náročná na výpočet.

Vizuálny výsledok:

Ďalšie poznámky:

Výsledky sa nám nezdajú byť správne. Myslíme si, že sme niekde spravili chybu a metóda nefunguje úplne správne. Nedarí sa nám dosiahnuť husté trblietky bez toho aby bolo vidno artefakty. Tie sa prejavujú tak, že trblietky vyzerajú byť na mriežke. Niekoľko sú trblietky aj roztiahanuté.

Ak sa pozeráme na povrch z diaľky, vyzerá to v poriadku, problém nastáva ak sme veľmi blízko pozorovaného objektu.

4.4.4 Metóda Wang et al.

Jednoduchosť použitia:

Niektoré parametre nie sú jasné čo presne robia.

Zložitosť implementácie:

Metóda je jednoduchá na implementáciu.

Parametre:

- Glitter Strength: Výsledná sila trblietok, použitá na doladenie.
- Use Anisotropy: Pri ostrom ulhe pozorovania môže vznikať aliasing ak je vypnutá.
- Sparkle Size: Mení veľkosť trblietky.
- Sparkle Density: Mení hustotu trblietok.
- Noise Density: Modifikuje ako silno aplikujeme šum.
- Noise Amount: Sila šumu.
- View Amount Jitter: Sila perturbácie trblietky, ktorá pomáha dodať efekt zhasívania.

Obmedzenia:

Pokiaľ nastavíme vysokú hustotu trblietok, je možné pozorovať vzory. Tieto vzory sa dajú rozbiť zvýšením *Noise amount* a *Noise Density*. *Noise Density* len zjemňuje silu parametra *Noise amount*. Možno by sa to dalo zlúčiť do jedného parametra. Pokiaľ je šum vysoký, pomôže to pri špecifických uhloch. Potom ale zmeníme napríklad hustotu a tieto vzory sa znova objavia. Vyžadujú si teda veľa dolaďovania. Pri hustej mriežke nepomáha skryť vzory ani na rovine, resp. strane kocky.

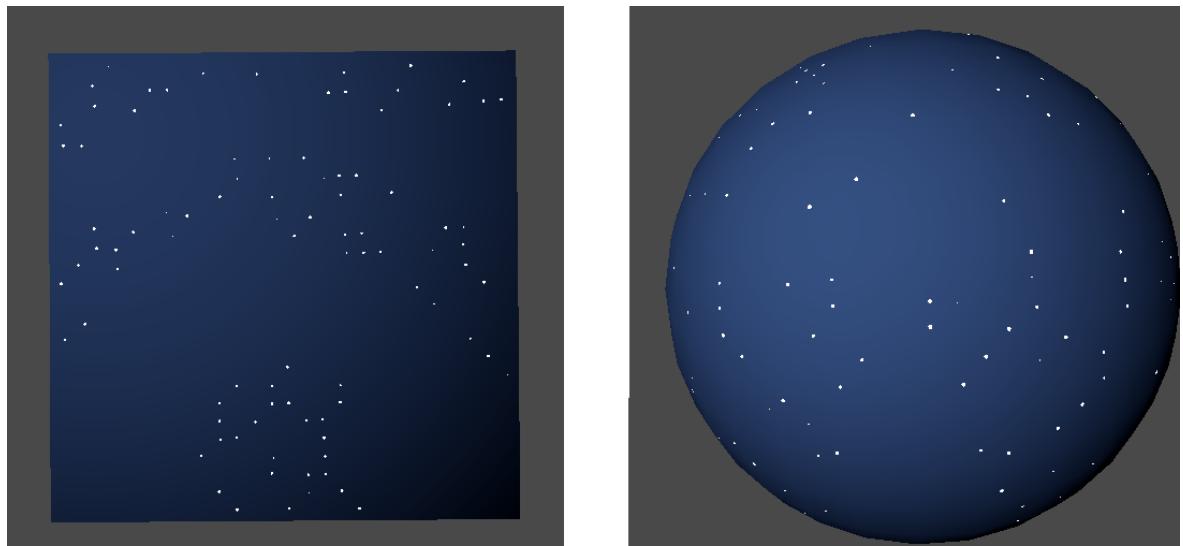
Na obrázku 4.8 môžeme vidieť metódu Wang bez šumu (naľavo), so šumom (v strede), so šumom (vpravo) na inom objekte.

Výkonnostná náročnosť:

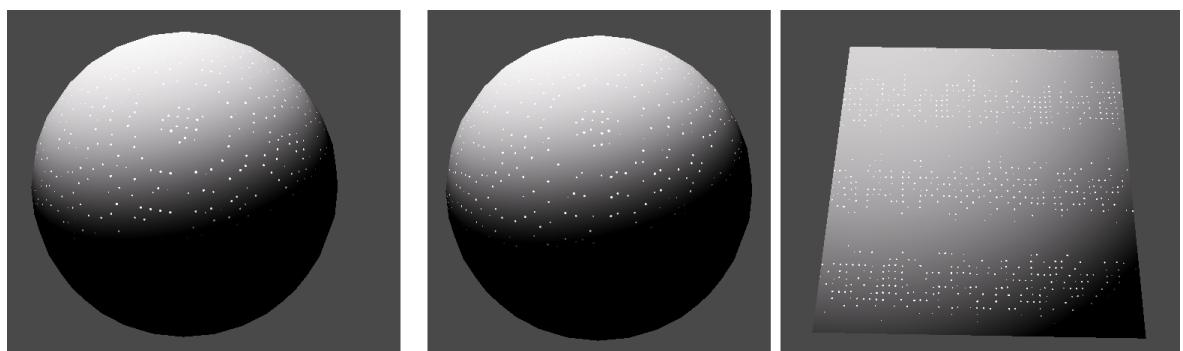
Metóda je nenáročná.

Vizuálny výsledok:

Metóda je čisto empirická pretože vytvára uniformné mriežky guličiek ktoré ked' pretínajú povrch tak tvoria trblietku. Metóda produkuje veľmi riedke trblietky. Pri väčšej hustote vidno vzory a to limituje hustotu. Veľmi malé trblietky



Obr. 4.7: Vizuál Wang et al. metódy.



Obr. 4.8: Rôzne parametre pri Wang metóde

4.4.5 Naša metóda

Jednoduchosť použitia:

Je komplikovanejšia na použite vďaka veľkému počtu parametrov. Podľa situácie - tvar objektu, pozícia kamery je nutné dolaďovať parametre aby metóda mala dobré vizuálne výsledky.

Zložitosť implementácie:

Implementácia je tiež nenáročná.

Parametre:

Kedže naša metóda stavia na metóde Wang et al., niektoré parametre sú totožné s metódou Wang. Pridané parametre pomáhajú odstrániť vzory a umožniť hustejšie trblietky.

- Global Roughness: Ward implementácia odleskov.
- Grid Loops Amount: Počet generovaných mriežok.
- Perlin Jitter Scale: Škála perlinového šumu.
- Toggle to use scales: Zapneme škálovanie mriežok.

Obmedzenia:

Kvalitnejšie trblietky sme získali na úkor výkonu. Je potrebné dolaďovať parametre na získanie pekných výsledkov.

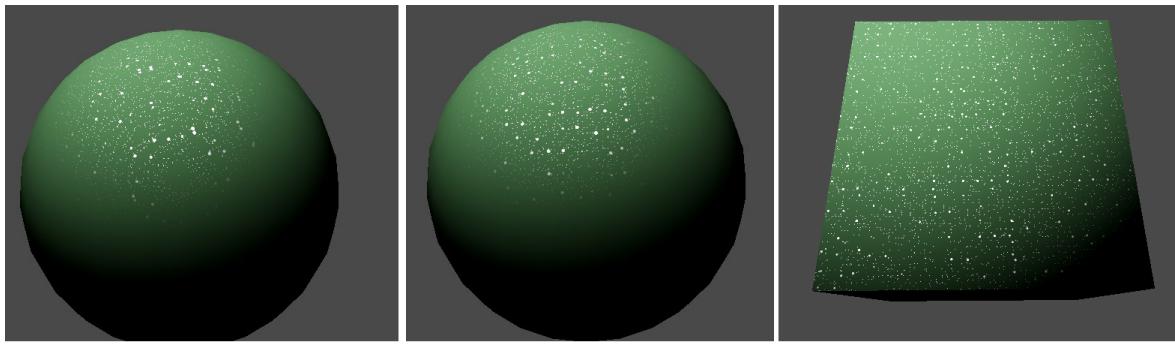
Výkonnostná náročnosť:

Výkonnostná náročnosť je rovnaká ako Wang, pokiaľ ponecháme počet mriežok na 1 a nepoužijeme škály. Viaceré mriežky a škály podstatne zvyšujú náročnosť.

Vizuálny výsledok:

Metóda dosahuje priemerné výsledky pri pohľade z blízka. Z väčšej diaľky je vizuál krajší, pokiaľ sa trblietky nestanú veľmi malými, vtedy vzniká nežiadúce mihotanie. Preto je dôležité vhodne zvoliť parametre.

Na porovnanie v obrázku 4.9 vidíme objekt s nezmenenými parametrami (vľavo), pridanie škál (v strede) a (vpravo) našou metódou. Umožňuje nám to reprezentáciu hustejších trblietok za cenu väčšej výpočtovej náročnosti.



Obr. 4.9: Vizuál našej metódy.

4.4.6 Zhrnutie

Všetky metódy majú svoje pozitíva a negatíva. Najlepšie výsledky podľa nás produkuje metóda Deliot et al.

Veľa metód je založených na UV súradničiach. V evaluácii sme používali objekty s dobrým UV mapovaním. Môžeme sa ale stretnúť s objektami ktorých mapovanie bude nerovnomerné. Vtedy môžu byť trblietky inej veľkosti a hustoty na iných častiach povrchu. Sú to ale okrajové prípady ktorým sa všeobecne tvorcovia snažia vyhnúť. Pravdepodobne problémy budú pri UV spojoch, kde sa stretávajú jednotlivé mapovania (UV ostrovy), a pri ostrých hranách. Viditeľné je to napríklad na obrázku 4.4 pri sfére sa to prejavilo čiernym pásikom.

4.4.7 Ďalšia práca

Podľa grafov a vizuálnych výsledkov navrhujeme ako ešte vieme vylepšiť vykreslovanie trblietavého vzhľadu. Naša metóda bola relatívne dostatočne výkonná, funguje na rôzne druhy svetiel ale pri pohľade zblízka vidno rôzne nedostatky ako aj mihotanie sa pri veľmi nízkej veľkosti trblietky.

Metóda Deliot a Belcour je vizuálne dosť pôsobivá. Jej nevýhodou je limitácia použitia na bodové typy svetiel a relatívne vysoká výpočtová náročnosť. Avšak je oveľa konzistentnejšia po výkonnostnej stránke pri meniacich sa uhloch pohľadu.

Preto navrhujeme, že metódy by sa mohli skombinovať, a to tak, že pre blízke objekty sa použije nákladnejšia Deliot metóda a pre vzdialenejšie objekty/povrhy a ostatné typy svetiel môžeme použiť našu implementáciu. Je potrebné eliminovať drobné trblietky a zmeniť počítanie šumu priamo za behu programu. Tiež bude potrebné zabezpečiť, aby pri prechode z jednej metódy a druhej neboli veľký vizuálny rozdiel, pretože by to pôsobilo rušivo. Toto je však téma na budúcu prácu.

Záver

V našej diplomovej práci sme sa zaoberali vykresľovaním trblietavých materiálov. V Kapitole 1 sme stručne priblížili problematiku vykresľovania a priblížili softvér Unity.

V Kapitole 2 sme sa do hĺbky venovali existujúcim metódam vykresľovania trblietavých materiálov pre interaktívne aplikácie. Stručne sme spomenuli aj offline vykresľovacie metódy. Vysvetlili sme ako vzniká trblietavý jav a aké výzvy prináša reprezentovanie povrchu s komplexnou mikroštruktúrou.

Implementácii vlastného shadera v Unity sme venovali kapitolu 3. Popísali sme ako funguje vykresľovací kanál HDRP a ako treba modifikovať tento kanál. Pridali sme popis tvorby vlastného editora aplikácie na porovnanie jednotlivých metód. Výslednú aplikáciu a samostatne použiteľný shader sme sprístupnili na githube, ktorý zároveň obsahuje celý zdrojový kód. Nachádza sa tam celý projekt v ktorom sme pracovali.

V poslednej kapitole 4 sme evaluovali našu metódu oproti existujúcim. Hodnotili sme jednoduchosť použitia daných metód, ich hardvérovú náročnosť, vizuálnu kvalitu, zložitosť implementácie a ich obmedzenia.

Myslíme si, že sme splnili cieľ vytvoriť vlastnú metódu tvorby trblietok. Pri vhodných parametroch je dobrá vizuálna kvalita. Vytvorili sme aj samostatne stiahnuteľný shader vo forme balíčka *glintsShader.unitypackage*, ktorého návod na použitie je

V budúcej práci by sme sa chceli zamerať na zlúčenie našej metódy a Deliot et al.[4] ako sme spomínali v časti 4.4.7. Taktiež chceme opraviť nedostatky. Náš shader mal určité nedostatky a to nekonzistentný čas snímkovania, ktorý bol zapríčinený dynamickým vetvením na GPU. Opraviť by sme chceli aj generovanie šumu a to použitím 3D textúry namiesto počítania šumu pri behu programu. V našej metóde sa prejavovalo mihotanie, ktoré nastávalo keď boli trblietky príliš malé. Metóda má ešte potenciál na zefektívnenie, niektoré výpočty sa dajú zrýchliť. Vhodné by bolo pozrieť sa na odložené vykresľovanie. Náš shader teraz používa dopredné vykresľovanie. Odloženým vykresľovaním potenciálne vieme zrýchliť výpočet pri veľkom počte svetiel.

Bibliografia

- [1] Akenine-Möller Tomas, Haines Eric a Hoffman Naty. *Real-time rendering*. CRC Press, 2018.
- [2] Belcour Laurent. „Efficient rendering of layered materials using an atomic decomposition with statistical operators“. V: *ACM Transactions on Graphics* 37.4 (2018), s. 1.
- [3] Cooper Tim a Unity Blog. *Scriptable Render Pipeline Overview*. <https://blog.unity.com/technology/srp-overview>. 2018. Dátum prístupu: 05.05.2024.
- [4] Deliot Thomas a Belcour Laurent. „Real-Time Rendering of Glinty Appearances using Distributed Binomial Laws on Anisotropic Grids“. V: *Computer Graphics Forum* 42.8 (2023). ISSN: 1467-8659. DOI: 10.1111/cgf.14866.
- [5] Ďuríkovič Roman, Kimura Ryou a Tsuruga. „Real-Time Rendering of Japanese Lacquerware“. V: (2003).
- [6] Ďuríkovič Roman a Ágošton Tomáš. „Prediction of optical properties of paints“. V: *Open Physics* 5.3 (2007), s. 416–427.
- [7] Ďuríkovič Roman a Kolchin Konstantin. „Real-time visualization of Japanese arcraft“. V: *Proceedings Computer Graphics International 2003*. IEEE. 2003, s. 184–189.
- [8] Ďuríkovič Roman, Kunovská Lucia a Mihálik Andrej. „Sparkling Effect in Virtual Reality Device“. V: *Mathematical Insights into Advanced Computer Graphics Techniques*. Singapore: Springer Singapore, 2019, s. 51–58. ISBN: 978-981-13-2850-3. DOI: 10.1007/978-981-13-2850-3_4. URL: https://doi.org/10.1007/978-981-13-2850-3_4.
- [9] Ďuríkovič Roman a Martens William L. „Simulation of sparkling and depth effect in paints“. V: *Proceedings of the 19th spring conference on Computer graphics*. 2003, s. 193–198.
- [10] Ershov Sergey, Ďuríkovič Roman, Kolchin Konstantin a Myszkowski Karol. „Reverse engineering approach to appearance-based design of metallic and pearlescent paints“. V: *The Visual Computer* 20 (2004), s. 586–600.

- [11] Frisvad Jeppe Revall, Jensen Søren Alkærsig, Madsen Jonas Skovlund, Correia António, Yang Li, Gregersen Søren K. S., Meuret Youri a Hansen Poul-Erik. „Survey of models for acquiring the optical properties of translucent materials“. V: *Computer graphics forum*. Zv. 39. 2. Wiley Online Library. 2020, s. 729–755.
- [12] Gardner Alice a Unity Technologies. *Find out how Unity Wētā Tools is transforming workflows for artists*. <https://blog.unity.com/news/unity-weta-tools-for-artists>. Aug. 2023. Dátum prístupu: 10.10.2023.
- [13] Gavilan David. *The Stencil Buffer and how to use it to visualize volume intersections*. <http://tech.meta1.com/tag/rendering/>. Júl 2020. Dátum prístupu: 12.03.2024.
- [14] Gonzalez Patricio. <https://github.com/Auburn/FastNoiseLite>. <https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83>. 2024.
- [15] Hayes Isaac. *Wallpaper Generator - Volumetric light scattering, 1 of 2*. <https://chuckleplant.github.io/2017/05/28/light-shafts.html>. Máj 2017. Dátum prístupu: 12.03.2024.
- [16] Heckbert Paul S. „Fundamentals of texture mapping and image warping. Master’s thesis“. V: *University of California, Berkeley* 2.3 (1989).
- [17] Heitz Eric. „Understanding the masking-shadowing function in microfacet-based BRDFs“. V: *Journal of Computer Graphics Techniques* 3.2 (2014), s. 32–91.
- [18] High-Performance Graphics a Deliot Thomas. *Real-Time Rendering of Glinty Appearances using Distributed Binomial Laws on Anisotropic Grids*. https://youtu.be/EB7qa4aL_XI&t=560. Júl 2023. Dátum prístupu: 1.12.2023.
- [19] Chermain Xavier, Sauvage Basile, Dischler J.-M. a Dachsbacher Carsten. „Procedural Physically based BRDF for Real-Time Rendering of Glints“. V: *Computer Graphics Forum*. Zv. 39. 7. Wiley Online Library. 2020, s. 243–253.
- [20] Immel David S., Cohen Michael F. a Greenberg Donald P. „A radiosity method for non-diffuse environments“. V: *Acm Siggraph Computer Graphics* 20.4 (1986), s. 133–142.
- [21] Jakob Wenzel, Hašan Miloš, Yan Ling-Qi, Lawrence Jason, Ramamoorthi Ravi a Marschner Steve. „Discrete stochastic microfacet models“. V: *ACM Transactions on Graphics (TOG)* 33.4 (2014), s. 1–10.
- [22] Japan400. *Into the Incandescent: Artworks of traditional and innovative Japanese maki-e*. <https://japan400.org/event/into-the-incandescent-artworks-of-traditional-and-innovative-japanese-maki-e/>. 2013. Dátum prístupu: 10.02.2024.

- [23] Kajiya James T. „The rendering equation“. V: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, s. 143–150.
- [24] LaRue Nancy a Blog Unity. *Unity 6 Preview is now available*. <https://blog.unity.com/engine-platform/unity-6-preview-release>. 2024. Dátum prístupu: 05.05.2024.
- [25] Meng Johannes, Papas Marios, Habel Ralf, Dachsbacher Carsten, Marschner Steve, Gross Markus a Jarosz Wojciech. „Multi-scale modeling and rendering of granular materials“. V: *ACM Trans. Graph.* 34.4 (júl 2015). ISSN: 0730-0301. DOI: 10.1145/2766949. URL: <https://doi.org/10.1145/2766949>.
- [26] Open Surprise. *The Last of Us Part I - PS3 Original vs. PS4 Remastered vs. PS5 Remake*. <https://youtu.be/fQliNIFkMIM&t=145>. Sept. 2022. Dátum prístupu: 12.03.2024.
- [27] Peck Jordan. <https://github.com/Auburn/FastNoiseLite>. <https://github.com/Auburn/FastNoiseLite>. 2024.
- [28] Shopf Jeremy. *Gettin' procedural*. <https://web.archive.org/web/20221009225825/http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Shopf-Procedural.pdf>. Okt. 2012. Dátum prístupu: 19.1.2024, prístupné cez službu Wayback Machine.
- [29] Trowbridge T. S. a Reitz Karl P. „Average irregularity representation of a rough surface for ray reflection“. V: *JOSA* 65.5 (1975), s. 531–536.
- [30] Unity Technologies. *AxF Material Inspector Reference*. <https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.0/manual/index.html>. 2023. Dátum prístupu: 10.10.2023.
- [31] Unity Technologies. *Branching in shaders*. <https://docs.unity3d.com/Manual/shader-branching.html#dynamic-branching>. 2024. Dátum prístupu: 4.05.2024.
- [32] Unity Technologies. *HDRP custom Material Inspectors*. <https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@13.1/manual/hdrp-custom-material-inspector.html>. 2024. Dátum prístupu: 10.04.2024.
- [33] Unity Technologies. *High Definition Render Pipeline overview*. <https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.0/manual/HDRP-Features.html>. 2023. Dátum prístupu: 10.10.2023.
- [34] Unity Technologies. *Characters*. <https://unity.com/solutions/characters>. 2023. Dátum prístupu: 10.10.2023.

- [35] Unity Technologies. *Industry solutions from Unity*. <https://unity.com/industry>. 2023. Dátum prístupu: 10.10.2023.
- [36] Unity Technologies. *Introduction to render pipelines*. <https://docs.unity3d.com/Manual/render-pipelines-overview.html>. 2023. Dátum prístupu: 10.10.2023.
- [37] Unity Technologies. *Shader compilation*. <https://docs.unity3d.com/Manual/shader-compilation.html>. 2024. Dátum prístupu: 10.04.2024.
- [38] Unity Technologies. *ShaderLab: defining a Shader object*. <https://docs.unity3d.com/Manual/SL-Shader.html>. 2024. Dátum prístupu: 10.04.2024.
- [39] Unity Technologies. *SpeedTree*. <https://unity.com/products/speedtree>. 2023. Dátum prístupu: 10.10.2023.
- [40] Unity Technologies. *VFX compositing tools*. <https://unity.com/solutions/compositing-tools>. 2023. Dátum prístupu: 10.10.2023.
- [41] Unity Technologies. *ZIVA*. <https://unity.com/products/ziva>. 2023. Dátum prístupu: 10.10.2023.
- [42] Vries Joey de. *Learn OpenGL, graphics programming: learn modern OpenGL graphics programming in a step-by-step fashion*. Kendall & Welling, 2020.
- [43] Wang Beibei a Bowles Huw. „A robust and flexible real-time sparkle effect“. V: *EGSR 2016 E&I-Eurographics Symposium on Rendering-Experimental Ideas & Implementations*. The Eurographics Association. 2016, s. 49–54.
- [44] Ward Gregory J. „Measuring and modeling anisotropic reflection“. V: *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*. 1992, s. 265–272.
- [45] Yan Ling-Qi, Hašan Miloš, Marschner Steve a Ramamoorthi Ravi. „Position-normal distributions for efficient rendering of specular microstructure“. V: *ACM Transactions on Graphics (TOG)* 35.4 (2016), s. 1–9.
- [46] Zhu Junqiu, Zhao Sizhe, Xu Yanning, Meng Xiangxu, Wang Lu a Yan Ling-Qi. „Recent advances in glinty appearance rendering“. V: *Computational Visual Media* 8.4 (2022), s. 535–552.
- [47] Zirr Tobias a Kaplanyan Anton S. „Real-time rendering of procedural multiscale materials“. V: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 2016, s. 139–148.
- [48] Zirr Tobias a Kaplanyan Anton S. *ShaderToy Glints shader example*. <https://www.shadertoy.com/view/1dVGRh>. 2024. Dátum prístupu: 12.10.2023.

Príloha A - Zdrojové kódy

```
1 //improved Method of Beibei Wang and Huw Bowles from the paper "A Robust and Flexible
  Real-Time Sparkle Effect"
2
3 #include <HLSLSupport.cginc>
4 #ifndef __GLINTSWBE__
5 #define __GLINTSWBE__
6 #include "PerlinNoiseLite.hlsl"
7 #ifndef LAYERED_LIT_SHADER
8 #define "LitProperties.hlsl"
9 #endif
10
11 float WardAniso(float3 L, float3 N, float3 V, float3 tangentVS)
12 {
13     float alphaX = _wbRoughness.x;
14     float alphaY = _wbRoughness.y;
15     float ro = 0.045;
16
17     float NdotV = dot(N, V);
18     float NdotL = dot(N, L);
19     float3 halfV = normalize(L + V);
20
21     float3 binormal = cross(N, tangentVS);
22
23     float HdotT = dot(halfV, tangentVS);
24     float HdotB = dot(halfV, binormal);
25     float HdotN = dot(halfV, N);
26
27     if (NdotL <= 0.0 || NdotV <= 0.0)
28         return 0;
29
30     float a = sqrt(max(0.0, ro * NdotL / NdotV));
31     float b = 1 / (4 * 3.14159 * alphaX * alphaY);
32     b = max(0.0, b * NdotL);
33
34     float t1 = HdotT / alphaX;
35     float t2 = HdotB / alphaY;
36
37     float term = -2.0 * ((t1 * t1 + t2 * t2)
38         / (1 + HdotN));
39     float e = 2.71828;
40     float c = pow(e, term);
41
42     // combine to get the final spec factor
43     return a * b * c;
44 }
45
46 float CalculateLevelBasedOnDistance(float3 ViewVec)
```

```

47 {
48     float zBuf = length(ViewVec);
49     float z_exp = log2(0.3 * zBuf + 3.0f) / 0.37851162325f;
50     float floorlog = floor(z_exp);
51     return 0.1f * pow(1.3f, floorlog) - 0.2f;
52 }
53
54 float3 GenerateGridCenter(float3 gridIndex, float sparkleGridDensity, float3 randPos)
55 {
56     float grid_length = 1.0f / sparkleGridDensity;
57     //add random position too
58     return gridIndex * grid_length + randPos;
59 }
60
61 float3 JitterGrid(float noiseAmount, float noise)
62 {
63     float jitter_noisy = noiseAmount * noise;
64     float3 jitter_grid = float3(jitter_noisy, jitter_noisy, jitter_noisy);
65     jitter_grid = 0.5f * frac(jitter_grid + 0.5f) - float3(0.75f, 0.75f, 0.75f);
66
67     return jitter_grid;
68 }
69
70 float CalculateFinalContribution(float3 newGridOffset, float newSparkleSize)
71 {
72     float l2 = dot(newGridOffset, newGridOffset);
73     float m_ss = 1.0f - newSparkleSize;
74
75     if (m_ss > l2)
76     {
77         float lend = ((1.0f - l2) - newSparkleSize) / (1.0f - newSparkleSize);
78         return 20.0f * lend;
79     }
80
81     return 0;
82 }
83
84 float3 GlintFade(WBEStruct wbeStruct, float3 viewDir, float noiseDensity, float
85     sparkleGridDensity,
86             float newSparkleSize,
87             float inoise, float floorLogPlus,
88             float3 objPos, float3 normal, float3 viewVec)
89 {
90     float levelZBuffer = CalculateLevelBasedOnDistance(viewVec);
91     //make the levels denser so there is less pop
92     float level = (0.12f + 0.5 * floorLogPlus) / (levelZBuffer);
93
94     sparkleGridDensity *= level;
95     noiseDensity *= level;
96
97     float3 warp = viewDir * levelZBuffer;
98     float3 warpedViewDir = sign(warp) * frac(abs(warp));
99
100    float3 randomPosition = randomVec3(objPos.xy, inoise);
101    float3 positionPlusView = objPos * sparkleGridDensity + wbeStruct.viewAmount *
102        normalize(
103            warpedViewDir + randomPosition);
104
105    float3 gridIndex = floor(positionPlusView);

```

```

104 float3 gridCenter = GenerateGridCenter(gridIndex, sparkleGridDensity, randomPosition);
105 float3 newGridOffset = positionPlusView - gridIndex;
106
107 float noise = 0;
108 if(_wbeNoise)
109     noise = PerlinNoise(noiseDensity * gridCenter * _wbJitterScale);
110
111 newGridOffset += JitterGrid(wbeStruct.noiseAmount, noise);
112
113
114 float dotvn = dot(viewVec, normal);
115 float3 large_dir = viewDir - (dotvn * normal);
116 if (wbeStruct.useAnisotropy)
117 {
118     newGridOffset += (abs(dotvn) - 1.0f) * dot(newGridOffset, large_dir) * large_dir /
119         dot(large_dir, large_dir);
120 }
121
122 float glittering = CalculateFinalContribution(newGridOffset, newSparkleSize);
123
124 return float3(glittering, glittering, glittering);
125
126 float3 CalcGrid(WBEstruct wbeStruct, float3 vViewVec, float3 vObjPos, float3 vNormal,
127     float1 inoise)
128 {
129     float3 n_view_dir = normalize(vViewVec);
130     float noiseDensity = wbeStruct.noiseDensity;
131     float sparkleDensity = wbeStruct.sparkleDensity * 15.0f;
132     float newSparkleSize = 1.0f - 0.2f * wbeStruct.sparkleSize * wbeStruct.sparkleDensity *
133         wbeStruct.sparkleDensity;
134     // Middle level
135     float level1 = 0.0f;
136     // Level below
137     float level0 = -1.0f;
138     float level2 = 1.0f;
139
140     float3 resultC = GlintFade(wbeStruct, n_view_dir, noiseDensity, sparkleDensity,
141         newSparkleSize, inoise,
142             level1,
143             vObjPos, vNormal, vViewVec);
144     if (_wbUseScales)
145         resultC +=
146             GlintFade(wbeStruct, n_view_dir, noiseDensity, sparkleDensity, newSparkleSize,
147                 inoise,
148                 level0,
149                 vObjPos, vNormal, vViewVec) +
150             GlintFade(wbeStruct, n_view_dir, noiseDensity, sparkleDensity, newSparkleSize,
151                 inoise,
152                 level2,
153                 vObjPos, vNormal, vViewVec);
154
155     return resultC;
156 }
157
158 float3 WBEnhancedGlints(float3 vObjPos, float3 vNormal, float3 lightPos, float3 vViewVec,
159     float3 tangentVS, WBEstruct wbeStruct)

```

```

156 {
157     float3 lightDir = normalize(lightPos.xyz - vObjPos);
158     float3 n_view_dir = normalize(vViewVec);
159
160     // test the noise pattern
161     //float test = PerlinNoise(vObjPos);
162     //return float3(test,test,test);
163
164     float3 glittering = float3(0.0f, 0.0f, 0.0f);
165     float specularity = WardAniso(lightDir, vNormal, -n_view_dir, tangentVS);
166
167
168     UNITY_LOOP for (int i = 1; i <= _wbGridAmount + 1; i++)
169     {
170         glittering += CalcGrid(wbeStruct, vViewVec, vObjPos, vNormal, i);
171     }
172     return glittering * specularity;
173 }
174 #endif

```

Pseudokód vlastnej metódy

Príloha B - Návod na ovládanie aplikácie

Link na stiahnutie aplikácie

Aplikácia je dostupná na adrese: <https://github.com/RoiIam/MastersHDRP/releases/tag/release>.

Hardvérové požiadavky

Aplikácia je určená pre Operačný systém Windows. Bola navrhnutá a testovaná v DirectX11 API. Bola testovaná na integrovanej grafike Intel UHD 630 a dedikovanej GPU Nvidia RTX 1660ti. Aplikácia by ale mala byť spustiteľná na hardvéri podporujúcom Shader model 5.0. Tieto zariadenia sú na trhu už viac ako dekádu.

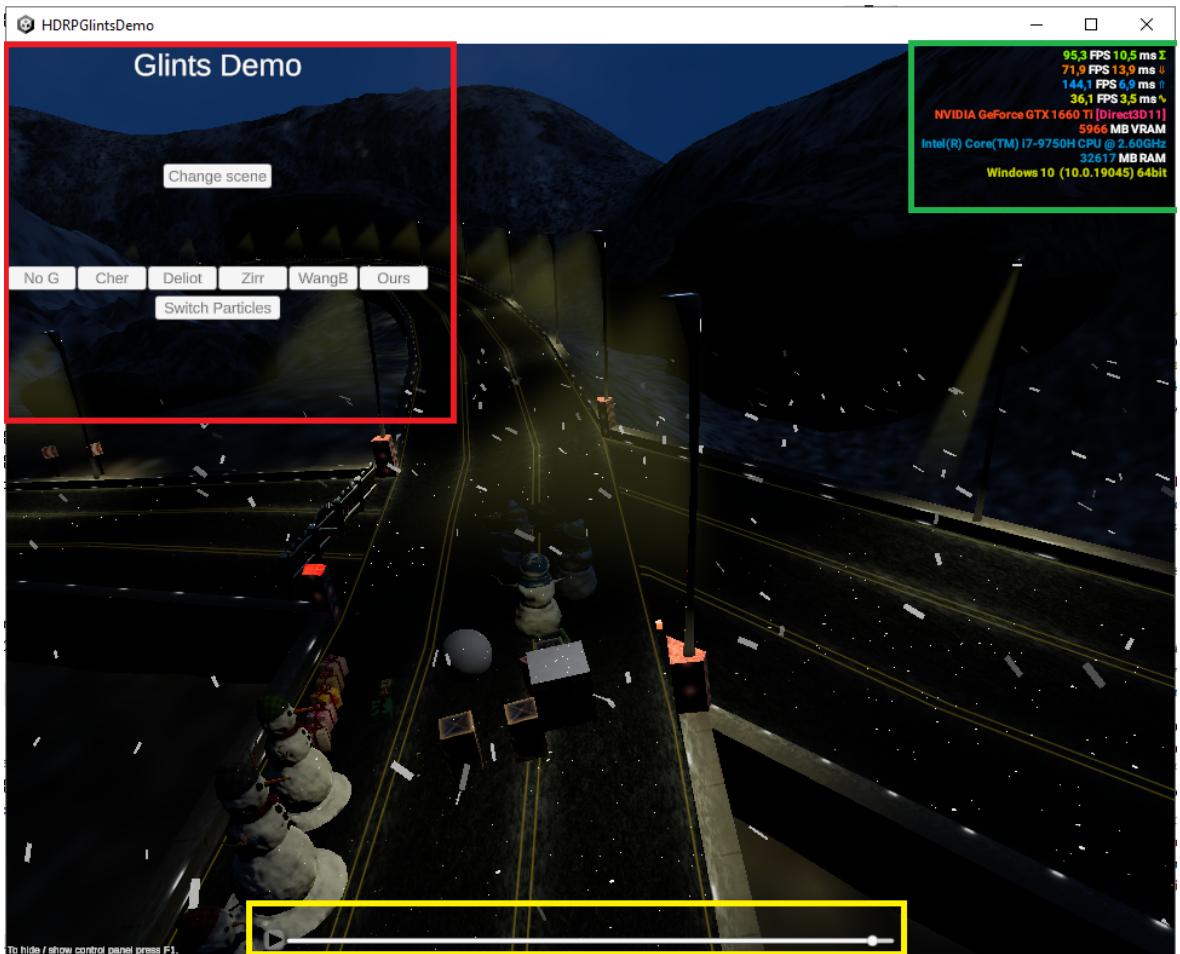
Spustenie aplikácie

Aplikácia sa spúšta na celú obrazovku otvorením .exe súboru. Neobsahuje upraviteľné grafické nastavenia, rozlíšenie sa prispôsobí veľkosti obrazovky. Používateľ vie aplikáciu minimalizovať do okna a zmeniť jeho veľkosť, napríklad vo Windows použítím skratky Win+Enter a následným rozťahnutím okna.

Po spustení aplikácie sa automaticky spustí demo scéna, ktorá slúži ako prezenácia metód.

Ovládanie aplikácie v prvej nočnej scéne:

- Pomocou klávesnice a myši: Kláves *medzerník* pozastaví respektíve obnoví animáciu kamery. Kláves *F1* zobrazí/schová UI. Kláves *Win+Enter* a následné rozťahnutie okna zmení rozlíšenie aplikácie.
- Pomocou myši ovládaním UI: V pravej hornej časti obrazovky, označenej zeleným rámčekom, je štatistika snímok za sekundu(FPS). Zobrazuje sa priemerné FPS, minimálne, maximálne a skutočné FPS. Obsahuje aj inofrmáciu o používanom hardvéri, OS a grafickej API. V spodnej časti obrazovky, označenej žltým



Obr. 10: Aplikácia s ovládacími prvkami

rámčekom sú ovládacie prvky kamery. Kliknutím a potiahnutím na spodný posuvník(nielen presne na bodku) sa presunieme na určitý čas animácie kamery. Kliknutím tlačidla naľavo vieme tiež pozastaviť/obnoviť animáciu kamery.

V ľavej hornej časti obrazovky označenej červeným rámčekom, sú prvky ovládane myšou: Kliknutím na *Change scene* sa prepne scéna na *Porovnávač*. Kliknutím na *Switch Particles* vypneme/zapneme efekt padania snehových vločiek.

Prepínaním pomocou zvyšných 5 tlačidiel zmeníme typ trblietok na niektorých materiáloch. V poradí zľava, doprava sú to: materiál bez trblietok, trblietky pomocou metódy Chermain et al., trblietky pomocou metódy Deliot et al., trblietky pomocou metódy Zirr et al., trblietky pomocou metódy Wang et al., a naša metóda.

Ovládanie aplikácie v porovnávacej scéne:

Podobne ako v prvej scéne, vieme pomocou UI meniť rôzne parametre. Kliknutím na *Change scene* sa prepne scéna na *nočnú scénu, demo*. Ak zaškrtneme *Split View*, zobrazí sa kópia objektu, na začiatku s rovnakým materiálom. Pomocou tlačidiel *Edit*

a *Edit right* zvoléme parametre ktorého objektu chceme zmeniť. Vieme tiež zmeniť typ objektu pomocou tlačidla *Change Mesh*, nastaviť auto otáčanie kamery pomocou prepínača *Animate cam* a rýchlosť otáčania pomocou slideru vpravo od neho.

Do políčok sa môže vložiť akýkoľvek text ale akceptované sa iba čísla. Reálne čísla môžu byť zapísané s bodkou alebo čiarkou. Text v políčku sa dá naraz vymazať aj prilepiť text. Pomocou myši:

- Ľavým tlačidlom myši meníme vertikálny sklon kamery.
- Pravým tlačidlom myši meníme orientáciu kamery okolo osi Y.
- Koliečkom myši meníme priblíženie.

Príloha C - Návod na vytvorenie Unity projektu s CustomLit Shaderom v HDRP

Link na stiahnutie aplikácie

Aplikácia a balíček s CustomLit trbieltavým shaderom je možné stiahnuť na adrese: <https://github.com/RoiIam/MastersHDRP/releases/tag/release>.

Inštalácia Unity

Nainštalujeme Unity Hub a príslušnú verziu Unity pomocou návodu: <https://unity.com/download#how-get-started>. Podľa HDRP dokumentácie: <https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.0/manual/System-Requirements.html> je HDRP 13.1 (v ktorom sme využívali a robili zmeny) vhodné len pre Unity verzie 2022.1. Odporúčame preto len verziu Unity 2022.1. Verzie 2022.x by mali tiež fungovať, ale môžu tam nastať problémy s kompatibilitou HDRP, repsectívne je možné, že Unity ani nedovolí nainštalovať verziu 13.1 HDRP na novšej verzii editora.

Tvorba projektu

1. V Unity Hub zvolíme tlačidlo *New Project* vpravo hore.
2. Vyberieme verziu Unity. Zvolíme šablónu **High Definition 3D Core** a nastavíme meno a umiestnenie projektu. V prípade, že sa chceme vyhnúť automatickému použitiu zlej verzie, použijeme šablónu *3D Built-in render pipeline* a manuálne cez menu *Window/PackageManager* pridáme vlastnú verziu. Stlačíme + tlačidlo vľavo hore a zvolíme *add package by name* a špecifikujme meno a verziu- meno: *com.unity.render-pipelines.high-definition* a verzia: *13.8.1*.
3. Stlačíme tlačidlo *Create Project*. Toto nastavenie použije najvyššiu možnú verziu HDRP. V našom prípade to bude vždy 13.1.x pre verziu Unity 2022.1. Teraz

Unity začne vytvárať projekt a kompilovať štandardne používané shadery. Ak sa kompilujú shadery, objekt, ktorý ho používa je zafarbený tyrkysovou farbou.

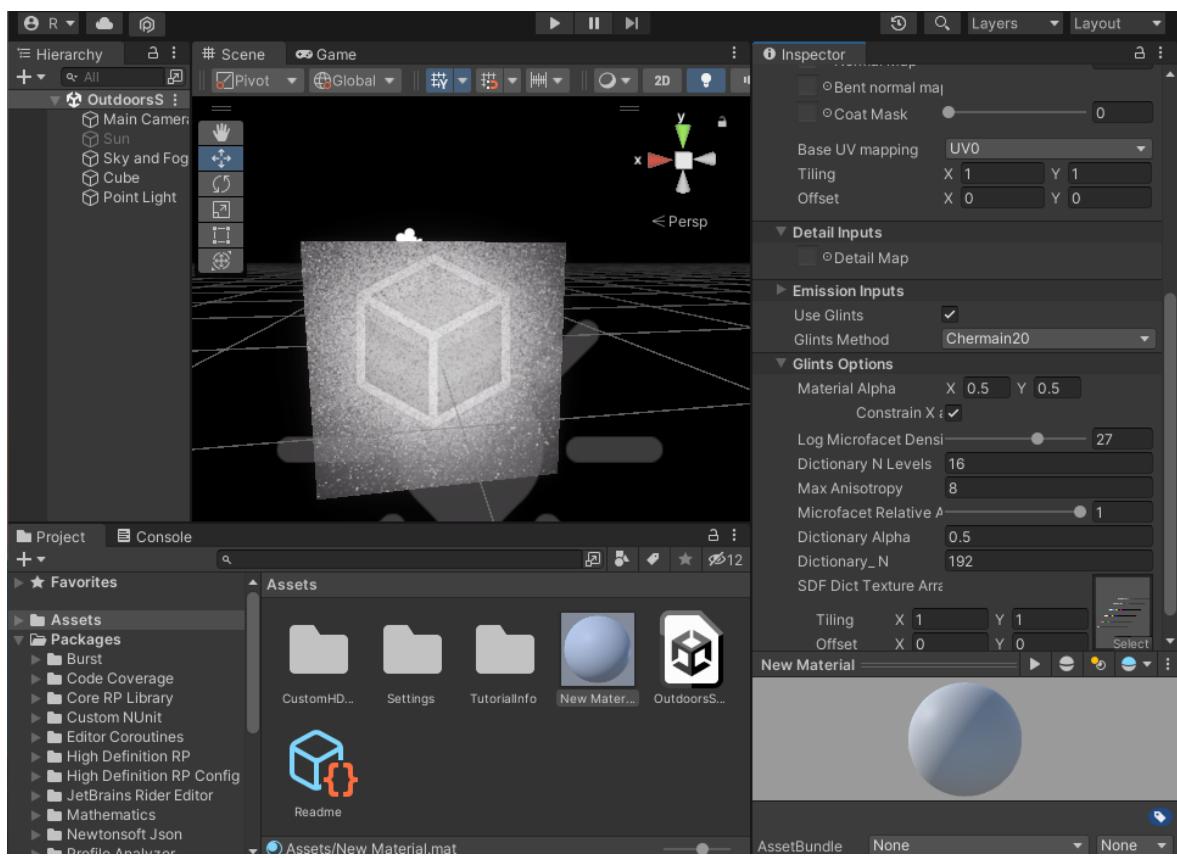
Nastavenie Unity

4. Keď sa vytvorí projekt, respektíve popri jeho vytváraní, stiahneme UnityPackage *glintsShader.unitypackage* obsahujúci vlastný trblietavý shader z linku: <https://github.com/RoiIam/MastersHDRP/releases/tag/release> a zároveň stiahneme aj balíček *high-definition-config1318.zip* z rovnakej url.
5. V pravo dole budeme vidieť prípadnú prácu Unity na pozadí. Keď je Unity priprané na použitie, zatvoríme prípadné vyskočené okná. Zavrieme aj Unity.
6. Rozbalíme *high-definition-config1318* a výsledný priečinok nakopírujeme do `rootOfProject/Packages/`.
7. Spustíme Unity projekt a počkáme kým sa dokončí import.
8. Môžeme v scéne vytvoriť prázdný objekt v Hierarchii pomocou kliku pravého tlačidla myši a zvolením . Priradíme mu vlastný materiál, ktorý vytvoríme v *Project* okne klikom pravej myši a zvolením *Create/Material*. Materiál potiahneme priamo na objekt v scéne aby ho používal.
9. Vytvoríme bodové svetlo v scéne. Dbajme na to aby intenzita bodového svetla bola približne 70 Lumenov a svetlo sa nenachádzalo v objekte ktorý chceme osvetliť.
10. Scénu uložíme stlačením *Ctrl+S*, respektíve potvrďme uloženie ak nás vyzve Unity napr. pri zatváraní projektu.
11. Objekty ako Sun môžeme vymazať. Kamera môže zostať v scéne.
12. Aby boli trblietky viditeľné potrebujeme zmeniť globálne nastavenia projektu.
13. Zvolíme *Edit/ProjectSettings*.
14. Zvolíme možnosť *Graphics* a klikneme na objekt *HDRenderPipelineAsset*. To nám v *Projekte* zvýrazní danný asset.
15. Kliknime naň v okne *Projekt* aby sa zobrazil v *Inšpektore*.
16. V sekcií *Rendering* musíme nastaviť *Lit Shader Mode* na *Forward Only* alebo *Both*. Pri zvolení možnosti *Both* bude chvíľu dlhšie kompilovať shadery.
17. Poslednú úpravu, ktorú musíme vykonať je zmeniť predvolenú kvalitu.

18. Konkrétnie v okne *Edit/Project Settings* zvolíme tentoraz *Quality*. Vidíme 3 levele: High Fidelity, Balanced a Performant.
19. Zvolíme ten ktorý je jemne zvýraznený, pretože ten profil je aktívny v okne editora.
20. Následne sa nám nižšie v sekciu *Rendering* zobrazí položka *Render Pipeline Asset*, napríklad *HDRP High Fidelity.asset*.
21. Kliknime naň aby sa zobrazil v *Inšpektore* podobne ako sme spravili pri *HDRenderPipelineAsset*.
22. Musíme zmeniť *Lit Shader Mode* na *Forward Only* alebo *Both*. Unity začne kompilovať shader, čo môže chvíľu trvať.

Pridanie balíčka

23. V ďalšom kroku zvolíme v ponuke menu **Assets/Import Package/Custom Package...** a zvolíme v kroku 4 stiahnutý *glintsShader.unitypackage*. Importujeme celý balíček tak, že v okne zvolíme *All* vpravo hore a stlačíme *Import*.
24. Počkáme kým Unity naimportuje a skompiluje shadery.
25. Vyberieme náš vytvorený materiál a v *Inšpektore* zvolíme *HDRP/CustomLit*.
26. Unity začne kompilovať shader, dovtedy bude objekt zafarbený tyrkysovou farbou.
27. Po skompilovaní by mal mať objekt na povrchu trblietky. Dôležité je pre metódy Chermain et al. a metódu Deliot et al. priradiť textúry a to stlačením *Select* a vybratím *Dictionary2D* pre parameter *SDF Dict Texture Array* resp. *glint2023noise.asset* pre parameter *Glint2023NoiseMap* pri vybranej metóde Deliot.



Obr. 11: Výsledná scéna po úspešnom pridaní trblietavého Shaderu v novom projekte.

Príloha D - Použité parametre pri evaluácii

- Parametre Lit shadera: predvolené parametre, ktoré sa nastavia pri vytvorení Lit materiálu v editore.
- Parametre CustomLit shadera bez trblietok: Úplne rovnaké ako pre materiál s Lit shaderom.
- Parametre Chermain shadera:
 - Material Alpha: (0.5 , 0.5)
 - Log Microfacet Density: 27
 - Dict N Levels: 16
 - Max Anisotropy: 8
 - Microfacet Relative Area: 1.0
 - Dictionary Alpha: 0.5
- Parametre Deliot shadera:
 - Max NDF: 0.5
 - Target NDF: 0.5
 - Density Randomization: 2
 - Log Microfacet Density: 16
 - Microfacet Roughness: 0.005
 - Screen Space Scale: 1.5
- Parametre Zirr shadera:
 - Global Roughness: (0.5 , 0.5)
 - Micro Roughness: (0.5 , 0.5)
 - Search Cone Angle: 0.02

- Variation: 35000
- Dynamic Range: 35 000
- Density: 100 000

- Parametre Wang shadera:

- Glitter Strength: 1000
- Use Anisotropy: False
- Sparkle Size: 0.015
- Sparkle Density: 5
- Noise Density: 0.1
- Noise Amount: 100
- View Amount Jitter: 8

- Parametre Nášho shadera: Zdieľa rovnaké parametre ako Wang metóda plus:

- Global Roughness: (0.5 , 0.5)
- Grid Loops Amount: 3
- Perlin Jitter Scale: 75
- Sparkle Density: 5
- Toggle to use scales: True

Príloha E - Zoznam použitých assetov

- **Názov:** Mountain Race Track - Night **Autor:** AndreiNi **Zdroj:** <https://assetstore.unity.com/packages/3d/environments/roadways/mountain-race-track-night-68199>
- **Názov:** FREE Christmas Gift from Blink - 2021 **Autor:** Blink **Zdroj:** <https://assetstore.unity.com/packages/2d/textures-materials/free-christmas-gift-from-blink-2021-209810>
- **Názov:** FREE Snowman **Autor:** ANGRY MESH **Zdroj:** <https://assetstore.unity.com/packages/3d/props/free-snowman-105123>
- **Názov:** Snowed Fence **Autor:** justtwo **Zdroj:** <https://assetstore.unity.com/packages/3d/environments/snowed-fence-6722>
- **Názov:** Lite FPS Counter **Autor:** OmniSAR Technologies **Zdroj:** <https://assetstore.unity.com/packages/tools/integration/lite-fps-counter-probably-the-world-s-fastest-fps-counter-132638>
- **Názov:** Unity texture packer **Autor:** andydbc **Zdroj:** <https://github.com/andydbc/unity-texture-packer.git>
- **Názov:** Suzanne **Autor:** Model z modelovacieho programu Blender.
- **Názov:** Christmas and Birthday Presents Pack **Autor:** Robot Skeleton **Zdroj:** <https://assetstore.unity.com/packages/3d/props/interior/christmas-and-birthday-presents-pack-157090>
- **Názov:** Presents **Autor:** Maria K **Zdroj:** <https://assetstore.unity.com/packages/3d/props/presents-135907>
- **Názov:** PBR Snowmen **Autor:** Ferocious Industries **Zdroj:** <https://assetstore.unity.com/packages/3d/props/pbr-snowmen-251152>
- **Názov:** Present **Autor:** FunFant **Zdroj:** <https://assetstore.unity.com/packages/3d/props/present-60575>

- **Názov:** PBR Christmas Gifts **Autor:** Ferocious Industries **Zdroj:** <https://assetstore.unity.com/packages/3d/props/pbr-christmas-gifts-237877>
- **Názov:** Shrek **Autor:** HarrisonHag1 **Zdroj:** <https://sketchfab.com/3d-models/shrek-ee9fbba7e7a841dbb817cc6cec678355>

Príloha F - Úryvok triedy GlintsMethodUIBlock

```
1 public class GlintsMethodUIBlock : MaterialUIBlock
2 {
3     private readonly ExpandableBit foldoutBit;
4     private MaterialProperty glintsMethod;
5     public List<MaterialPropertyType> activeGlintProperties = new();
6     private MaterialProperty dbDensityRandomization;
7     private MaterialProperty dbLogMicrofacetDensity;
8     ...
9     private MaterialProperty chDictionary_Alpha;
10    private MaterialProperty chDictionary_N;
11    ...
12    private MaterialProperty wbRoughness;
13    private MaterialProperty wbUseScales;
14    private MaterialProperty wbViewAmount;
15    ...
16    public GlintsMethodUIBlock(ExpandableBit expandableBit)
17    {
18        foldoutBit = expandableBit;
19    }
20
21    public override void LoadMaterialProperties()
22    {
23        activeGlintProperties.Clear();
24
25        glintsMethod = FindProperty("_glintsMethod");
26        useGlints = FindProperty("_UseGlints");
27
28        //cher20
29        chMaterial_Alpha = FindProperty("_chMaterial_Alpha");
30        ...
31        //Deliot23
32        dbMaxNDFBlock = FindProperty("_dbMaxNDF");
33        ...
34    }
35
36    public void ShaderProperty2(MaterialProperty m, string name,
37        MaterialPropertyData.GlintsType t, bool matches, bool include = true)
38    {
39        if (matches)
40            materialEditor.ShaderProperty(m, name);
41        if (include)
42            activeGlintProperties.Add(new MaterialPropertyType(m, t));
43    }
}
```

```

44
45     public void ShowChermainParams(MaterialPropertyData.GlintsType t)
46     {
47         var type = MaterialPropertyData.GlintsType.Cher;
48         var matches = t == type ? true : false;
49         ShaderProperty2(chMaterial_Alpha, "Material Alpha", type, matches);
50         ShaderProperty2(chLogMicrofacetDensity, "Log Microfacet Density", type,
51             matches);
52         ShaderProperty2(chDictionary_NLevels, "Dictionary N Levels", type, matches);
53         ShaderProperty2(chMaxAnisotropy, "Max Anisotropy", type, matches);
54         ShaderProperty2(chMicrofacetRelativeArea, "Microfacet Relative Area", type,
55             matches);
56         ShaderProperty2(chDictionary_Alpha, "Dictionary Alpha", type, matches);
57         ShaderProperty2(chDictionary_N, "Dictionary N", type, matches, false);
58         ShaderProperty2(chSDFDictBlock, "SDF Dict Texture Array", type, matches,
59             false);
60     }
61     ...
62     public void ShowDeliotParams(MaterialPropertyData.GlintsType t){}
63     ...
64     ...
65     public override void OnGUI()
66     {
67         materialEditor.ShaderProperty(useGlints, "Use Glints");
68
69         if (useGlints.floatValue != 1.0)
70             return;
71         //Debug.Log("type for blockui",type);
72         //Debug.Log(activeGlintProperties.Count);
73         materialEditor.ShaderProperty(glintsMethod, "Glints Method");
74
75         using (var header = new MaterialHeaderScope("Glints Options", (uint)foldoutBit,
76             materialEditor))
77         {
78             if (header.expanded)
79             {
80                 var type = (MaterialPropertyData.GlintsType)glintsMethod.floatValue;
81                 ShowChermainParams(type);
82                 ShowDeliotParams(type);
83                 ShowZirrParams(type);
84                 ShowWangParams(type);
85                 ShowWBEnhancedParams(type);
86                 ...
87             }
88         }
89     }

```

Vlastný editor - trieda GlintsMethodUIBlock