Alternativne senčilne metode za računalniško grafiko

Raziskovalna naloga

Področje:

Računalništvo

Avtor:

Alen Cigler, R4a

Mentor:

mag. Boštjan Resinovič, univ. dipl. inž. rač. in inf.

Mestna občina Celje, Mladi za Celje

Celje, april 2021

»Programiranje je umetnost,
ker povezuje abstraktnost in resničnost,
ker zahteva spretnost in iznajdljivost,
predvsem pa zato, ker proizvaja predmete lepote.«
- Donald Knuth

ZAHVALA

Za nasvete in predloge ter za pomoč pri nastajanju naloge se zahvaljujem mentorju, gospodu Boštjanu Resinoviču, ki me je strokovno usmerjal in mi pomagal s konstruktivnimi napotki ter mi omogočil, da sem se naučil nekaj novega.

Alen Cigler

Povzetek

Glavni namen raziskovalne naloge je bil raziskati senčilne tehnike, ki jih lahko uporabijo tako posamezniki kot tudi večja podjetja, ki se ukvarjajo z zabavno industrijo. Poudarek je bil predvsem na alternativnih metodah, ki uporabijo senčilnike za druge namene, kot le navadno senčenje objektov.

Ker je to področje povezano z računalniško grafiko, je najprej predstavljen grafični cevovod in delovanje ter zgradba senčilnikov, nato pa so predstavljene alternativne metode, ki se v industriji vse več uporabljajo ter njihova implementacija v pogonu iger Unity. Vse predstavljene metode se lahko implementirajo tudi v drugih pogonih iger ali aplikacijah za upodabljanje računalniške grafike.

Ključne besede: senčilnik, grafični cevovod, senčilnik fragmentov, senčilnik oglišč, HLSL, Unity.

Abstract

The main purpose of this research project was to explore different shading techniques that can be used by individuals as well as larger companies working in the entertainment industry. The emphasis was mainly on alternative methods, which use shaders for purposes other than just ordinary shading of objects.

Because this area is related to computer graphics, the rendering pipeline and shaders are introduced first, followed by alternative methods that are increasingly used in the industry along with their implementation in the Unity game engine. All researched methods can also be implemented in other game engines or applications for rendering.

Key words: shader, rendering pipeline, fragment shader, vertex shader, HLSL, Unity.

Vsebina

Kazalo slik

1 Uvod

1.1 Opredelitev področja in raziskovalnega problema

Dandanes predstavlja računalniška grafika glavno vlogo pri uspehu projekta v zabavni industriji. Vizualno oblikovanje igra pomembno vlogo v njegovem umetniškem vtisu. Kljub temu da izgled projekta zelo vpliva na uporabnikovo doživljanje, je uresničiti umetniško vizijo za mnoge razvijalce pogosto nemogoče. Predvsem to izhaja iz dejstva, da je za doseganje grafičnih učinkov potrebno dobro poznavanje delovanja računalniške grafike.

Večina uporabnikov se danes tako nauči uporabljati enega izmed mnogih uporabniku prijaznih orodij. Najbolj popularna so tista, ki temeljijo na grafih, predstavljenih z vozlišči in povezavami, kot so Shader Graph v pogonu iger Unity, Material Editor v pogonu iger Unreal Engine in Shader Nodes v naboru grafičnih orodij Blender.

Glavni problem je predvsem, da ta orodja uporabljajo veliko abstrakcij za zakrivanje dejanske zapletenosti grafičnega cevovoda in smo zato z njimi omejeni ter odvisni od drugih, da nadgradijo ta orodja. Zgodi se nam lahko tudi, da ne razumemo delovanja določene abstrakcije, kar vodi v dolgotrajno spreminjanje različnih parametrov in s tem tratenje časa. Poznavanje delovanja grafičnega cevovoda in senčilnikov nam omogoči, da reči vzamemo v svoje roke in dosežemo točno takšno grafiko, kot si jo želimo, ter uporabimo celotno moč grafičnega procesorja.

1.2 Cilji raziskovalne naloge

Cilj je raziskati in predstaviti senčilne metode, ki jih lahko uporabijo tako posamezniki kot tudi večja podjetja, ki se ukvarjajo z zabavno industrijo. Poudarek bo predvsem na alternativnih metodah, ki uporabijo senčilnike za druge namene kot le navadno senčenje objektov in je nekatere zaradi tega težje ali celo nemogoče implementirati v uporabniku prijaznih orodjih. Sprva bom predstavil grafični cevovod in delovanje ter zgradbo senčilnikov, nato pa se osredotočil na alternativne metode in jih brez izgube splošnosti implementiral v pogonu za igre Unity.

Cilji v teoretičnem delu:

- predstaviti osnovne principe računalniške grafike, ki so potrebni za razumevanje senčilnikov;
- predstaviti delovanje grafičnega cevovoda;
- predstaviti matematično ozadje, ki je potrebno za razumevanje senčilnikov;
- predstaviti zgradbo in delovanje osnovnih senčilnikov;
- raziskati in predstaviti alternativne senčilne metode.

Cilji v praktičnem delu:

- implementirati osnovne senčilnike;
- implementirati alternativne senčilne metode;
- ugotoviti in predstaviti uporabnost alternativnih senčilnih metod;
- primerjati alternativne senčilne metode s tradicionalnimi načini, ki dosežejo enake ali podobne rezultate;
- izdelati spletno stran za prikazovanje senčilnikov.

1.3 Hipoteze

H1: Senčilniki se uporabljajo za veliko več, kot le senčenje objektov.

H2: Senčilniki so primeren uvod v računalniško grafiko in jih bi lahko obravnavali tudi pri srednješolskem pouku računalništva.

H3: S senčilniki lahko spreminjamo stil računalniške grafike (bolj ali manj realistična) na parameteriziran način.

H4: Senčilnike lahko uporabimo za upodabljanje zelo velike količine objektov v realnem času.

H5: S senčilniki, s katerimi opišemo volumne, lahko nekatere objekte upodobimo na lažji način, kot če bi jih modelirali.

H6: Senčilniki za opisovanje volumnov so primerni za aplikacije, ki tečejo v realnem času.

1.4 Metodologija

Z raziskovanjem sem začel tako, da sem najprej poiskal in preletel vire, ki obravnavajo raziskovalno tematiko. Na osnovi tega sem zastavil hipoteze in oblikoval cilje raziskovalne naloge.

Sledilo je podrobnejše proučevanje virov, ki so bili v različnih jezikih. Kljub temu da mi je bilo področje znano, sem ga moral podrobneje raziskati, saj se o računalniški grafiki v šoli nismo veliko učili. Sprva se je skozi vire bilo zahtevno prebijati, čez čas pa sem se od njih začel vedno bolj oddaljevati in implementiral tudi lastne rešitve.

Nato sem se lotil praktičnega dela raziskovanja, v katerem sem uporabljal predvsem metodo študija primerov in empiričnega ter eksperimentalnega raziskovanja, se pravi, da sem teorijo poskušal v čim večji meri preizkusiti v praksi. Med delom sem si sproti zapisoval ideje in poskušal razne tehnike v upanju izpopolnjevanja teorije in odrivanju morebitnih napak ter njihovega razreševanja.

Pri raziskavi je bilo tudi nekaj omejitev, in sicer:

- omejeni elektronski in še bolj pisni viri o alternativni uporabi senčilnikov;
- kratek čas za raziskavo, le nekaj mesecev;
- dokumentacija za veliko metod v tej nalogi je zelo skopa in običajno ni vsebovala primerov uporabe;
- občasen problem razumevanja implementacij senčilnih metod zaradi slabših prevodov (nekateri viri so celo v japonščini in kitajščini);
- veliko tehnik je še v razvoju, zato za njih ne obstaja veliko literature izven raziskovalnih inštitucij.

2 Principi računalniške grafike

Da bi lahko razumeli razne senčilne metode, je najprej potrebno dobro razumevanje nekaterih principov računalniške grafike.

2.1 Struktura sveta in koordinatni sistemi[1]

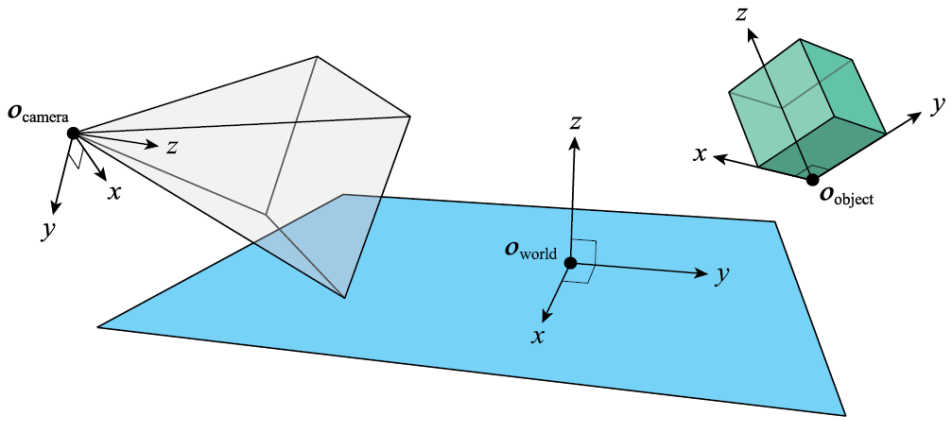
Svet, ki ga želimo upodobiti, je sestavljen iz različnih vrst objektov. Najosnovnejši so trije:

- geometrijski objekti, ki so običajno sestavljeni iz trikotniških mrež in jih želimo upodobiti;
- viri svetlobe, ki jih potrebujemo za osvetljevanje geometrijskih objektov;
- kamera, ki predstavlja pogled nad svetom, ki ga bomo upodobili.

Vsak izmed teh objektov ima svojo pozicijo v svetu, ki pa je lahko predstavljena z različnimi koordinatnimi sistemi.

Objekti imajo svoj lokalni koordinatni sistem, ki se imenuje objektni prostor (angl. object space) (Slika 2.1). Izhodišče in osi tega koordinatnega sistema so določene na smiseln način. Kocka ima na primer izhodišče svojega objektnega prostora v določenem oglišču, osi pa poravnane vzporedno z njenimi stranicami. Kamera ima izhodišče svojega objektnega prostora na sredini pogleda in eno os poravnano s smerjo pogleda.

[1] Povzeto po: Lengyel E. (2019), 14–25



Slika 2.1: Objektni prostori
Vir: (Lengyel, 2019)

Ko govorimo o objektnem prostoru kamere, ga običajno poimenujemo prostor kamere (angl. camera space) oziroma prostor pogleda (angl. view space).

Oglišča geometrijskih objektov, ki jih želimo upodobiti, morajo biti pretvorjena v koordinatni sistem, ki je poravnan s prostorom kamere. Ko se geometrijski objekt nahaja v pogledu kamere, sta ta dva objekta med seboj v interakciji. Če želimo opravljati kalkulacije, ki vsebujejo več objektov, moramo med njimi vzpostaviti prostorski odnos. To se doseže z uvedbo novega globalnega koordinatnega sistema, ki mu pravimo prostor sveta (angl. world space) oziroma realni prostor. Iz linearne algebre vemo, da lahko transformacije iz prostora $\mathbb{R}^n$ v prostor $\mathbb{R}^m$ predstavimo s transformacijskimi matrikami. Vsakemu objektu lahko tako dodelimo transformacijsko matriko $\mathbf{M} \in \mathbb{R}^{4\times4}$, ki predstavlja njegovo pozicijo in orientacijo v realnem prostoru. Matrika $\mathbf{M}$ pretvori vektorje iz koordinatnega sistema objektnega prostora v koordinatni sistem realnega prostora. Prvi trije stolpci matrike $\mathbf{M}$ predstavljajo smeri osi objektnega koordinatnega sistema predstavljene v realnem prostoru, četrti stolpec pa pozicijo objekta v objektnem prostoru predstavljeno v realnem prostoru.

Ko se zgodi interakcija med dvema objektoma A in B, se morajo vse kalkulacije zgoditi v skupnem realnem prostoru. To storimo tako, da pretvorimo vse vektorske količine obeh objektov v realni prostor preko njunih transformacijskih matrik $\mathbf{M_A}$ in $\mathbf{M_B}$, ali pa enega izmed njiju preko realnega prostora pretvorimo v objektni prostor drugega s produktom matrik $\mathbf{M_A}^{-1}\mathbf{M_B}$ ali $\mathbf{M_B}^{-1}\mathbf{M_A}$.

2.2 Transformacije oglišč

Ko grafični procesor upodablja geometrijski objekt, mora njegovo trodimenzionalno pozicijo v objektnem prostoru pretvoriti v dvodimenzionalno pozicijo na prikazovalniku. To je opravljeno skozi zaporedje transformacij, ki pretvarjajo pozicije oglišč iz enega v drug koordinatni sistem, dokler niso pretvorjene v koordinatni sistem pogleda. Vsak objekt ima podatke le o svoji transformaciji $\mathbf{M_{objekt}}$ za pretvorbo iz objektnega v realni prostor. Tudi kamera, skozi katero vidimo svet, ima podatke o svoji transformaciji $\mathbf{M_{kamera}}$ za pretvorbo iz prostora kamere v realni prostor. Ti dve matriki lahko združimo v novo transformacijo, ki pretvori oglišča iz objektnega prostora v prostor kamere preko produkta $\mathbf{M_{kamera}}^{-1}\mathbf{M_{objekt}}$.

Po pretvorbi v prostor kamere potrebujemo dodatno transformacijo, preden lahko grafični procesor določi končne pozicije na prikazovalniku. Ta transformacija se imenuje projekcija in je opravljena z matriko $\mathbf{P} \in \mathbb{R}^{4\times4}$, ki pretvori oglišča iz prostora kamere v rezalni prostor preko produkta $\mathbf{P}\mathbf{M_{kamera}}^{-1}\mathbf{M_{objekt}}$, kjer so oglišča predstavljena v homogenih koordinatah, kar pomeni, da so predstavljena z vektorjem $\mathbf{v} \in \mathbb{R}^4$. Koordinate v tem prostoru predstavljajo tako pozicije oglišč kot tudi globino v smeri pogleda kamere. V naslednjem poglavju bomo govorili o senčilniku oglišč, katerega naloga je skoraj vedno vsaj pretvorba oglišč iz objektnega v rezalni prostor preko množenja matrik.

Rezalni prostor (angl. clip space) je dobil poimenovanje po tem, ker ima pri tem koordinatnem sistemu grafični procesor dovolj podatkov, da določi, kateri trikotniki v mreži segajo izven pogleda kamere. Ti trikotniki so izrezani tako, da so po rezanju v celoti znotraj pogleda kamere. Transformacijska matrika je pravzaprav zasnovana tako, da oglišče v rezalnem prostoru s homogenimi koordinatami

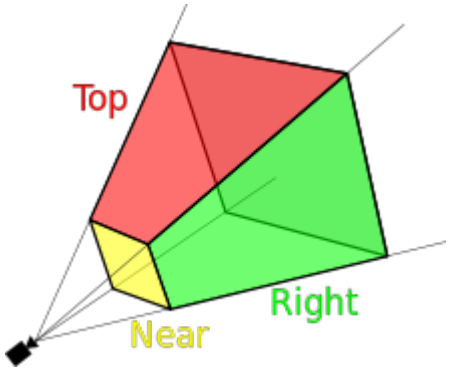
$(x_{rezalna}, y_{rezalna}, z_{rezalna,} w_{rezalna})$ zadovolji sistemu neenačb:

$$-w_{rezalna} \leq x_{rezalna} \leq w_{rezalna}$$

$$-w_{rezalna} \leq y_{rezalna} \leq w_{rezalna}$$

$$0 \leq z_{rezalna} \leq w_{rezalna}$$

Ta sistem neenačb predstavlja volumen znotraj šestih rezalnih ravnin kamere (Slika 2.2). V primeru, da oglišče ne zadovolji sistemu teh neenačb, se z interpolacijo izračuna novo oglišče.



Slika 2.2: Rezalne ravnine kamere
Vir: (Viewing frustum, 2012)

Ko je rezanje opravljeno, so oglišča pretvorjena iz štiridimenzionalnega rezalnega prostora v tridimenzionalne normalizirane koordinate naprave (angl. normalized device coordinates) oziroma prostor naprave. To dosežemo tako, da delimo vsako oglišče z njegovo `w` koordinato. S to operacijo dosežemo perspektivo, kar pomeni, da so objekti daleč od kamere manjši na prikazovalniku kot objekti bližje kamere. V prostoru naprave oglišča oblike $(x_{naprave}, y_{naprave}, z_{naprave})$ zadovoljujejo sistemu neenačb:

$$-1 \leq x_{rezalna} \leq 1$$

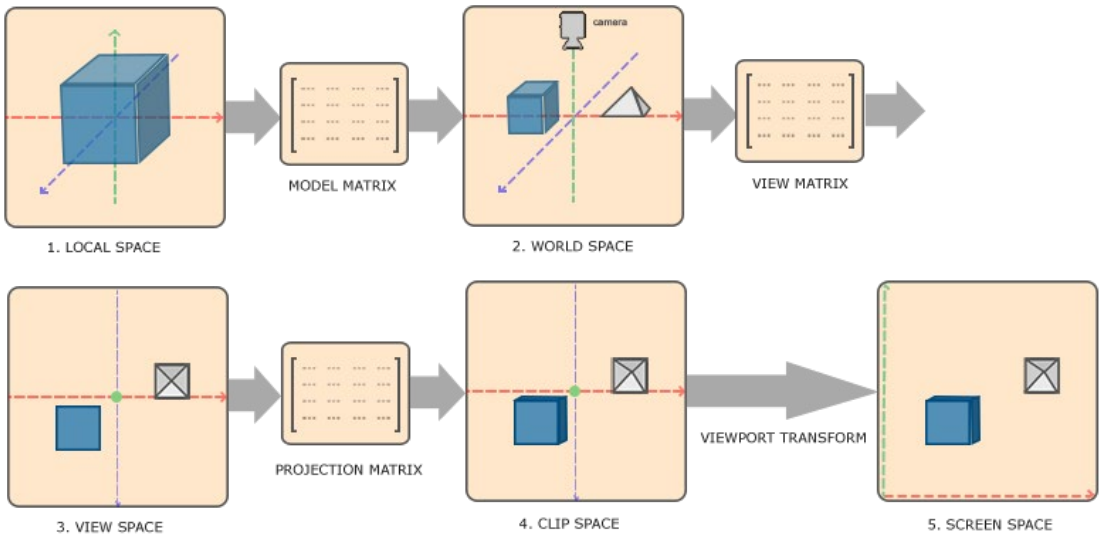
$$1 \leq y_{rezalna} \leq 1$$

$$0 \leq z_{rezalna} \leq 1$$

Ta sistem neenačb predstavlja dimenzije volumna pogleda, ki predstavlja prizmo velikosti $2 \times 2 \times 1$.

Oglišča so nazadnje še pretvorjena v prostor prikazovalnika, tako da x in y koordinate iz prostora naprave preslikamo na intervala $[0, w]$ in $[0, h]$, kjer sta w in h širina in višina prikazovalnika v zaslonskih pikah, z koordinata pa je preslikana na interval $[g_{min}, g_{max}]$, ki predstavlja minimalno in maksimalno globino volumna kamere.

Vse transformacije so še enkrat prikazane spodaj (Slika 2.3).



Slika 2.3: Transformacije oglišč
Vir: (Vries)

2.3 Grafični cevovod[2]

Ko mora grafični procesor upodobiti objekt, se na njem izvede zaporedje zapletenih ukazov. Ti v več fazah pretvorijo podatke o ogliščih in trikotnikih, ki opisujejo zgradbo objekta v dejanske fragmente na ekranu. Ta skupek faz se imenuje grafični cevovod (angl. rendering pipeline) ali izrisovalni cevovod. Nekateri deli tega cevovoda so programirljivi s programi imenovanimi senčilniki, ki so v ospredju te raziskovalne naloge, nekateri pa so nespremenljivi in predstavljajo specializirane enote v grafičnem procesorju, ki so zadolžene za opravljanje specifične naloge (Slika 2.4).

Izraz senčilnik je včasih predstavljal majhen program, katerega namen je bil opraviti zgolj senčenje in osvetlitev objekta med upodabljanjem njegove površine, dandanes pa je uporabnost teh programov zrasla krepko preko tega osnovnega namena. Tako sedaj izraz senčilnik uporabljamo za kateri koli program, ki poteka na grafičnem procesorju, ne glede na njegovo uporabo.

Večina senčilnikov uporablja le podatke o ogliščih, nekateri pa tudi teksture, barve in druge podatke o objektih.

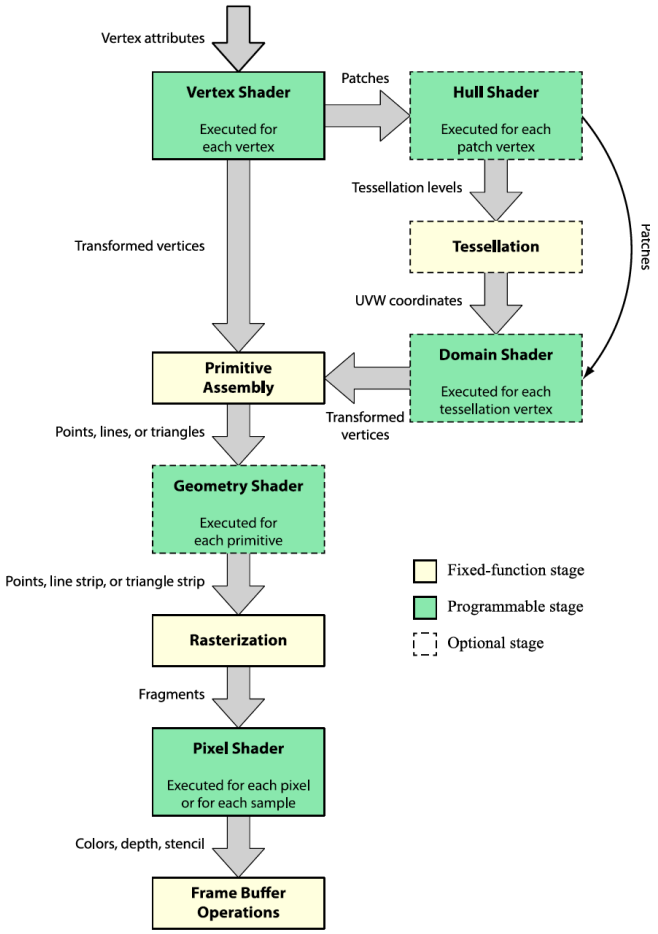
Pred časom so grafični procesorji podpirali le senčilnike oglišč in fragmentov, ki sta tudi edini dve obvezni programirljivi fazi v izrisovanju slike. Danes lahko napišemo tudi

[2] Povzeto po: (Tomas, Haines, & Hoffman, 2018), 14-25

senčilnike geometrij, plašča in domen. V tej raziskovalni nalogi se bomo osredotočili na alternativno uporabo senčilnikov oglišč in fragmentov.

Poleg teh senčilnikov pa lahko uporabimo tudi senčilnik za izračunavanje (angl. compute shader), ki poteka zunaj grafičnega cevovoda in jih uporabimo za pospeševanje računsko zahtevnih del povezanih z računalniško grafiko.

Ko se procesiranje geometrije objekta konča, sledi postopek rasterizacije, kjer so trikotniki razbiti na posamezne fragmente. Po tem procesu so vse operacije izvršene na nivoju fragmentov s senčilnikom fragmentov.



Slika 2.4: Nivoji grafičnega cevovoda
Vir: (Lengyel, 2019)

2.3.1 Senčilnik oglišč

V splošnem je grafična kartica zadolžena za izrisovanje treh tipov geometrijskih primitivov: točk, premic in trikotnikov. Večina izrisanih objektov sestoji iz trikotnikov.

Prvi korak pri izrisovanju objekta je pridobiti polje oglišč, ki jih procesiramo v senčilniku oglišč. Vsako oglišče v trikotni mreži običajno pripada več trikotnikom. Če bi želeli izrisati mrežo, ki vsebuje m trikotnikov, bi lahko neodvisno shranili vsa tri oglišča trikotnika v polju oglišč in ukazali GPU, naj izriše seznam trikotnikov s $3m$ oglišči. Ta pristop pa bi povzročil podvajanje podatkov. Zaradi tega uporablja GPU polje indeksov pri izrisovanju trikotnikov, namesto da bi izrisal vsak trikotnik s tremi zaporednimi oglišči v polju oglišč, se trikotnik izriše s tremi zaporednimi oglišči v polju indeksov, ki so na naključnih mestih v polju oglišč.

Polje oglišč vsebuje enega ali več atributov oglišč in vsak med njimi vsebuje od enega do štiri številske komponente. V skrajnem primeru obstaja vsaj en atribut, ki vsebuje podatke o pozicijah vseh oglišč in je predstavljen kot 3D vektor v objektnem prostoru. Običajno pa polje oglišč vsebuje tudi 2D teksturne koordinate, ki jih uporabimo za izvajanje senčilnih operacij.

Pozicije oglišč morajo biti v fazi rasterizacije predstavljene v štiridimenzionalnih koordinatah rezalnega prostora, zato mora zadnja faza pred njo opraviti pretvorbo iz objektnega prostora v rezalni prostor preko transformacijskih matrik.

2.3.2 Senčilnik fragmentov

Preden oglišča v rezalnem prostoru pridejo do senčilnika fragmentov, morajo najprej biti pretvorjena v fragmente, za kar poskrbi nivo rasterizacije. Operacija, ki se zgodi najprej, je rezanje, saj imamo v tej fazi dovolj podatkov, ker se oglišča nahajajo v rezalnem prostoru.

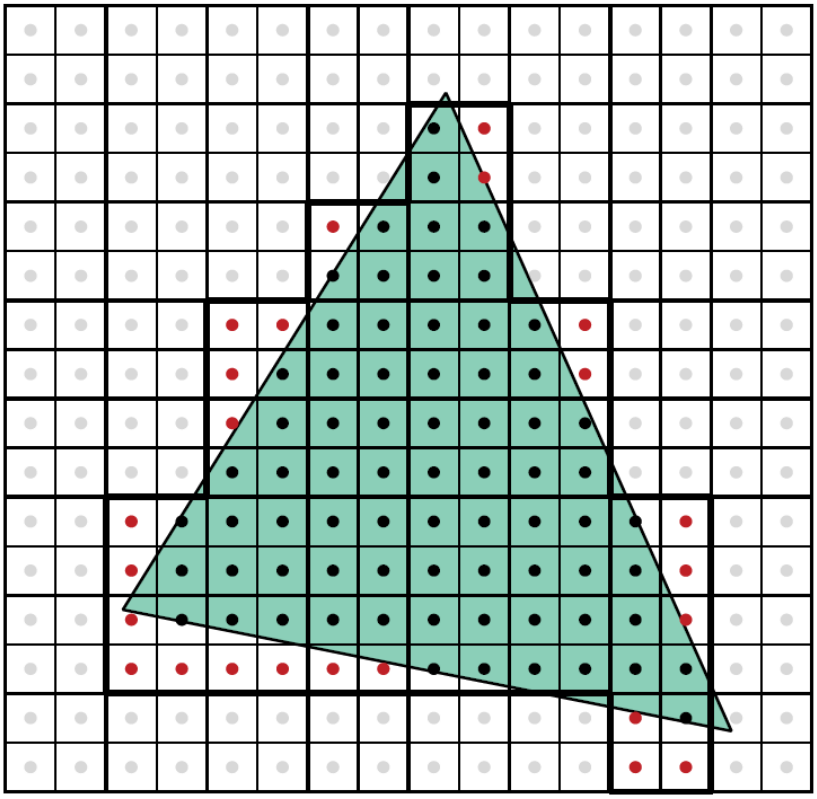
Sledi pomembna operacija zakrivanja trikotnikov, katere namen je izločiti trikotnike, ki so obrnjeni proti pogledu kamere ali pa stran od njega. Določitev opravimo tako, da izračunamo predznačeno ploščino trikotnika z uporabo determinante:

$$S = \frac{1}{2}\begin{vmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{vmatrix}$$

Ko je vrednost te ploščine pozitivna, je trikotnik običajno obrnjen proti nam, to pa je odvisno od tega, kako smo definirali bazo prostora. Operacijo zakrivanja trikotnikov lahko tudi izpustimo. V tej raziskovalni nalogi bomo uporabili vse tri možnosti.

Trikotniki, ki ostanejo vidni po zakrivanju, so nato pretvorjeni v fragmente, le-ti predstavljajo množico zaslonskih točk, ki pripadajo prvotnim primitivom. Fragmenti se

določijo za vse štiri zaslonske točke v 2×2 kvadratih, ki imajo vsaj eno zaslonsko točko vsebovano v primitivu (Slika 2.5). Zaslonska točka je vsebovana, če je njeno središče vsebovano v primitivu. Razlog, da jih generiramo v 2×2 kvadratih, je v tem, da lahko grafični procesor izbere pravilni nivo filtriranja po vzorčenju tekstur, za kar potrebuje odvode v x in y smeri. Najenostavnejši način izračuna teh odvodov je, da odštejemo vrednosti teksturnih koordinat med sosednjima zaslonskima točkama.



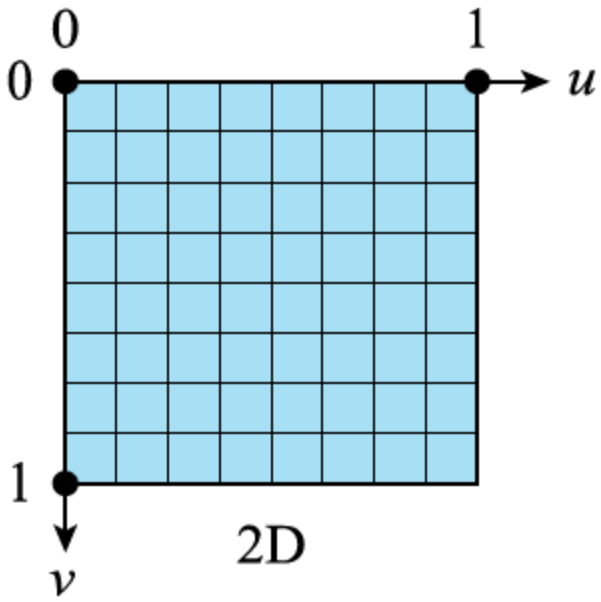
Slika 2.5: Rasterizacija
Vir: (Lengyel, 2019)

Pred in po senčilniku fragmentov se izvede še veliko drugih operacij, ki pa za nas v tej nalogi niso pomembne. Namen vseh je izločitev določenih fragmentov v namene optimizacije ali pa za doseganje posebnih zapletenih učinkov.

Naloga senčilnika fragmentov je, da vsakemu fragmentu določi barvo. Ta barva lahko povozi tisto, ki se že nahaja v grafičnem pomnilniku, ali pa se barvi zmešata po določenih pravilih.

2.4 Teksture

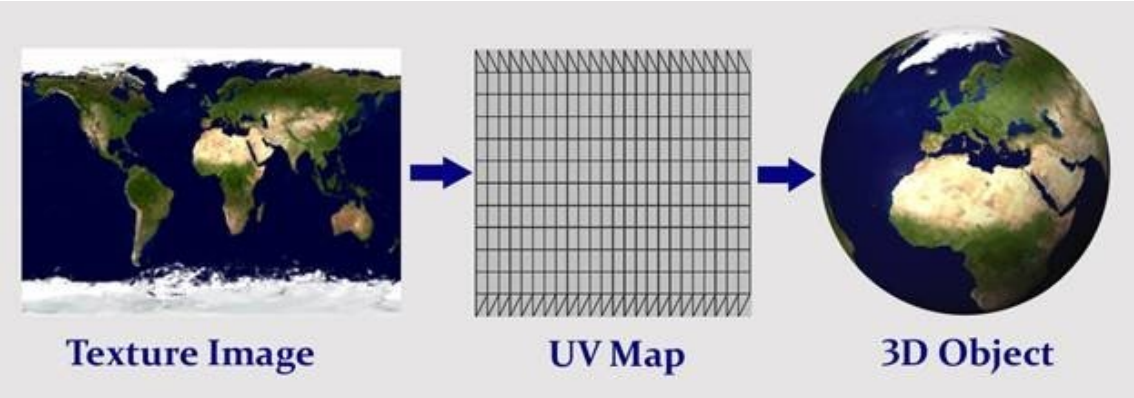
Moderni grafični procesorji omogočajo uporabo tekstur, ki lahko vsebujejo podatke v različnih formatih. Teksture so na trikotnih mrežah uporabljene tako, da jim dodelimo v naprej določene teksturne koordinate (angl. texture coordinates). Običajno so dvodimenzionalne in se imenujejo UV koordinate, ker X in Y že uporabljamo za pozicije objekta (Slika 2.6).



Slika 2.6: Teksturne koordinate
Vir: (Lengyel, 2019)

UV koordinate vsebujejo kakršne koli podatke o površini objekta, kot so barve, normale in podobno.

Najobičajnejša uporaba tekstur je "ovijanje " 2D slike okoli trikotne mreže (Slika 2.7), uporabimo pa jih lahko tudi za kateri koli drug namen.



Slika 2.7: Uporaba teksture
Vir: (Tschmits, 2008)

3 HLSL in implementacija senčilnikov

Senčilniki bodo v nalogi implementirani v pogonu iger Unity 2019.3.5f1 z jezikom HLSL (High-level Shader Language), isti principi pa veljajo tudi za grafične cevovode v drugih aplikacijah.

Kljub temu da cilj raziskovalnega dela ni delovanje pogona iger Unity, bodo vseeno prikazane nastavitve specifične za ta program, v upanju lažjega iskanja alternativ v drugih okoljih.

Unity ima možnost ustvarjanja lastnega grafičnega cevovoda, ali pa uporabo enega izmed štirih že prisotnih grafičnih cevovodov, ki so namenjeni določenemu nivoju grafične zahtevnosti. Ti so: preprost (angl. lightweight), visoko ločljivostni (angl. high definition), univerzalni (angl. universal) in privzet (angl. default).

Zaradi njegove robustnosti bomo v raziskovalni nalogi uporabljali univerzalni grafični cevovod, ki je primeren za vse projekte v Unity in bo v bližnji prihodnosti nadomestil privzet cevovod, ki se ga danes velikokrat označuje kot zastarelega (angl. legacy).

Za začetek se bomo osredotočili na pisanje preprostih dvodimenzionalnih senčilnikov, ki so primerni za prikaz osnovnih lastnosti izbranega jezika za pisanje senčilnikov. Kljub preprostosti samih primerov so v njih zajeti ključni koncepti, ki jih bomo uporabljali tudi pri pisanju naprednejših in uporabnih senčilnikov v naslednjem poglavju.

Za lažje razumevanje bom v vseh primerih v raziskovalni nalogi spremenljivke, lastnosti in funkcije poimenoval slovensko, razen, kjer bi to kršilo kakšen standard.

3.1 Osnovni primer

Pri primerih v tem poglavju se bomo osredotočili le na pisanje senčilnika fragmentov, ki prejme koordinate in vrne barvo, ki jo želimo prikazati na koordinati. Začeli bomo z najbolj osnovnim primerom senčilnika, ki je zapisan z univerzalnim grafičnim cevovodom (Izsek kode 3.1).

```
Shader "PrviPrimer"
{
    Properties
    {
        _Tekstura ("Tekstura", 2D) = "white" {}
    }
    SubShader
    {

        Tags { "RenderType" = "Opaque" "RenderPipeline" =
"UniversalRenderPipeline" }

        Pass
        {

            HLSLPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "Packages/com.unity.render-
pipelines.universal/ShaderLibrary/Core.hlsl"


            struct Attributes
            {
                float4 pozicijeOS   : POSITION;
            };

            struct Varyings
            {
                float4 pozicijeCS  : SV_POSITION;
            };


            Varyings vert(Attributes IN)
            {

                Varyings OUT;
                OUT.pozicijeCS = TransformObjectToHClip(IN.pozicijeOS.xyz);
                return OUT;
            }

            float4 frag() : SV_Target
            {
                float4 poljubnaBarva;
                poljubnaBarva = float4(0.5, 0, 0, 1);
                return poljubnaBarva;
            }
            ENDHLSL
        }
    }
}
```

Izsek kode 3.1: Najbolj osnovni senčilnik v univerzalnem grafičnem cevovodu

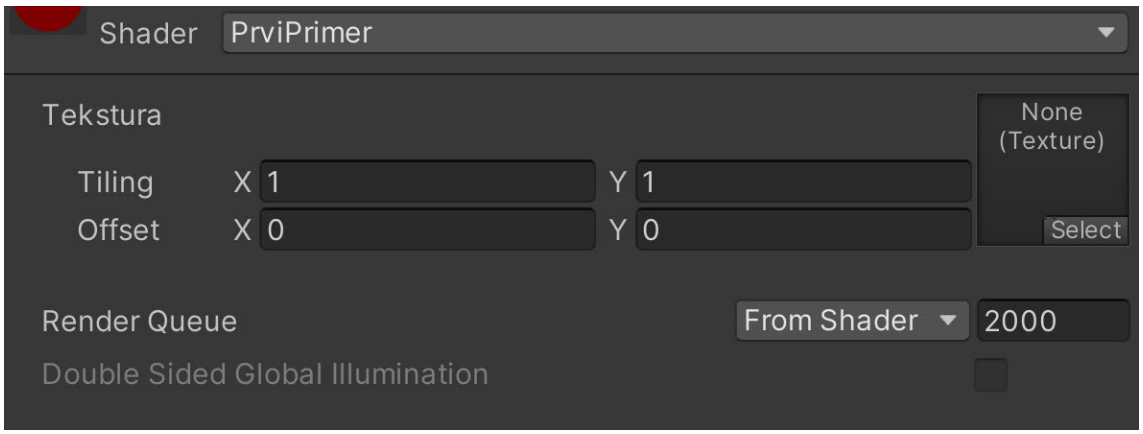
Ta senčilnik bo predstavljal začetno stanje za vse senčilnike v raziskovalni nalogi. Razložil ga bom po vrsti. Naprej navedemo ime senčilnika in lastnosti (Izsek kode 3.2).

Lastnosti definiramo z imenom, ki se običajno začne s podčrtajem, nato pa v oklepajih navedemo ime, ki se prikaže v Unity urejevalniku ter tip podatka. Lastnosti lahko dodelimo tudi privzeto vrednost. V raziskovalni nalogi so vse spremenljivke, ki imajo pripadajoče lastnosti, poimenovane s predpono "_". Definicije lastnosti v nadaljevanju večinoma ne bodo prikazane, saj jim kasneje v senčilniku potrebujemo določiti še spremenljivke. Vse spremenljivke v raziskovalni nalogi, ki imajo predpono "_", imajo tudi svoje pripadajoče lastnosti.

```
Shader "PrviPrimer"
{
    Properties
    {
        _Tekstura("Tekstura", 2D) = "white" {}
    }
```

Izsek kode 3.2: Določanje imena senčilnika in lastnosti

V pogonu iger Unity se nam lastnosti pokažejo v Unity prikazovalniku (Slika 3.1).



Slika 3.1: Lastnosti prikazane v Unity prikazovalniku

Sledi oznaka `SubShader`, ki vsebuje oznake, ki definirajo kdaj in kako bodo potekali posamezni grafični prehodi oziroma sloji (Izsek kode 3.3).

```
SubShader
{

    Tags { "RenderType" = "Opaque" "RenderPipeline" =
"UniversalRenderPipeline" }
```

Izsek kode 3.3: Določanje oznak

V posameznem prehodu označimo mesto začetka HLSL programa z oznako `HLSLPROGRAM`, nato pa uporabimo ukaze za določitev imen senčilnika oglišč in senčilnika fragmentov. Standardno ime senčilnika oglišč je `vert`, standardno ime senčilnika fragmentov pa `frag` (Izsek kode 3.4).

```
Pass
{
    HLSLPROGRAM

    #pragma vertex vert

    #pragma fragment frag
```

Izsek kode 3.4: Določanje imen senčilnikov

Sledi vključitev datoteke `Core.hlsl`, ki vsebuje implementacije pogosto uporabljenih HLSL funkcij (Izsek kode 3.5).

```
#include "Packages/com.unity.render-
pipelines.universal/ShaderLibrary/Core.hlsl"
```

Izsek kode 3.5: Vključitev datoteke `Core.hlsl`

Nato definiramo dve strukturi, ki bosta vsebovali podatke, ki prehajajo med različnimi nivoji grafičnega cevovoda (Izsek kode 3.6). Struktura, ki predstavlja vhodne podatke za senčilnik oglišč, se običajno imenuje `Attributes`, struktura, ki predstavlja izhodne podatke za senčilnik oglišč in vhodne podatke za senčilnik fragmentov, pa `Varyings`. Podatki, ki jih smemo uporabljati, so definirani v Unity, vendar se ne razlikujejo med grafičnimi cevovodi. V vsakem primeru potrebujemo v strukturi `Attributes` vsaj podatke o pozicijah oglišč, ki so podana v objektnem prostoru, pri strukturi `Varyings` pa vsaj podatke o pozicijah v rezalnem prostoru za rasterizacijo objekta na zaslon. Ta dva podatka določimo z uporabo semantičnih oznak `POSITION` in `SV_POSITION`.

```
struct Attributes
{
    float4 pozicijeOS   : POSITION;
};

struct Varyings
{
    float4 pozicijeCS  : SV_POSITION;
};
```

Izsek kode 3.6: Definiranje struktur s podatki

Za konec sledita še dve metodi, ki predstavljata senčilnik oglišč in senčilnik fragmentov. Trenutno bomo pri senčilniku oglišč le pretvorili pozicije iz objektnega prostora v rezalni prostor z uporabo metode, ki jo omogoča `Core.hlsl`. To bi lahko opravili tudi sami tako, da bi pozicijo v objektnem prostoru množili z dvema transformacijskima matrikama. V raziskovalni nalogi bomo vedno uporabili vgrajene metode za transformacijo, saj je to priporočljivo.

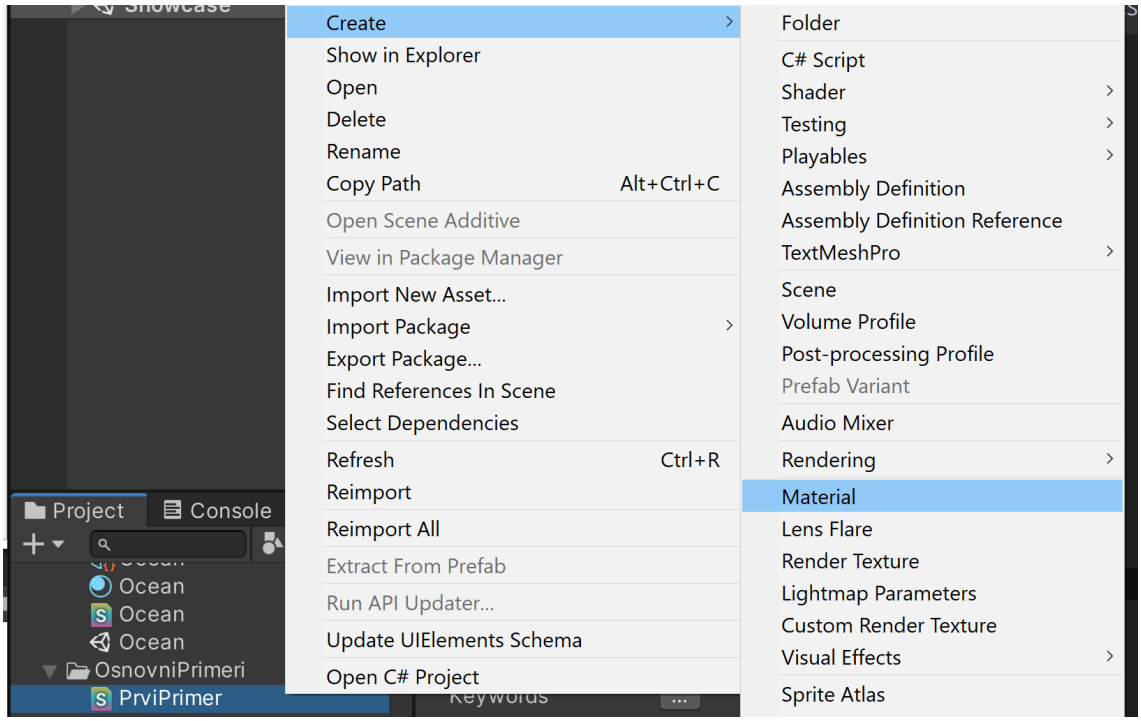
V senčilniku fragmentov pa za ta preprosti primer ne bomo uporabili vhodnih podatkov in bomo za vse fragmente določili enako barvo ne glede na njihovo pozicijo (Izsek kode 3.7). Izhodni podatek senčilnika fragmentov je običajno le barva posameznih slikovnih točk. S semantično oznako `SV_Target` prevajalniku povemo, da bo vrnjena vrednost uporabljena za upodobitev.

Po zapisu vse HLSL kode še zaključimo z `ENDHLSL` oznako.

```
Varyings vert(Attributes IN)
{

    Varyings OUT;

    OUT.pozicijeCS = TransformObjectToHClip(IN.pozicijeOS.xyz);

    return OUT;
}


float4 frag() : SV_Target
{
    float4 poljubnaBarva;
    poljubnaBarva = float4(0.5, 0, 0, 1);
    return poljubnaBarva;
}
ENDHLSL
```

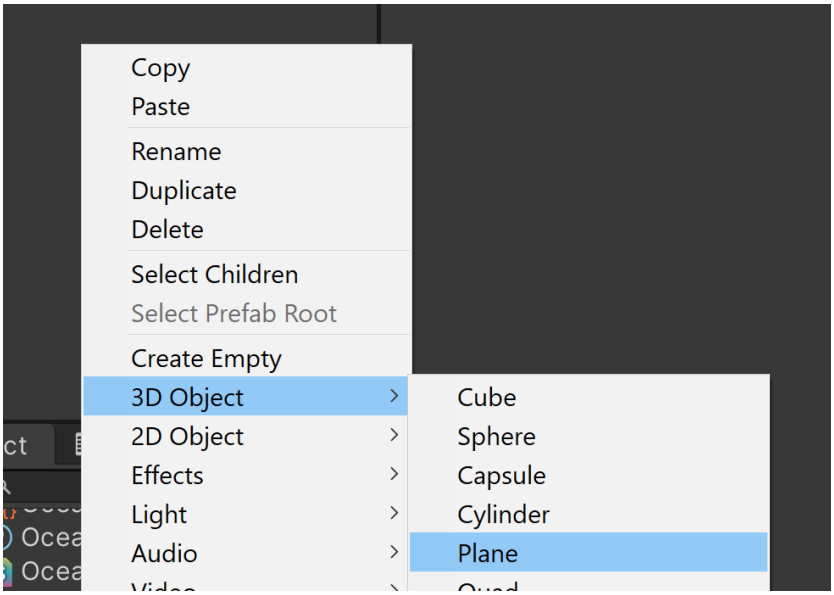
Izsek kode 3.7: Zapis senčilnikov

V Unity urejevalniku bomo ustvarili primerek tega senčilnika, ki se v pogonih iger in drugih programih za upodabljanje grafike običajno imenuje material (Slika 3.2).



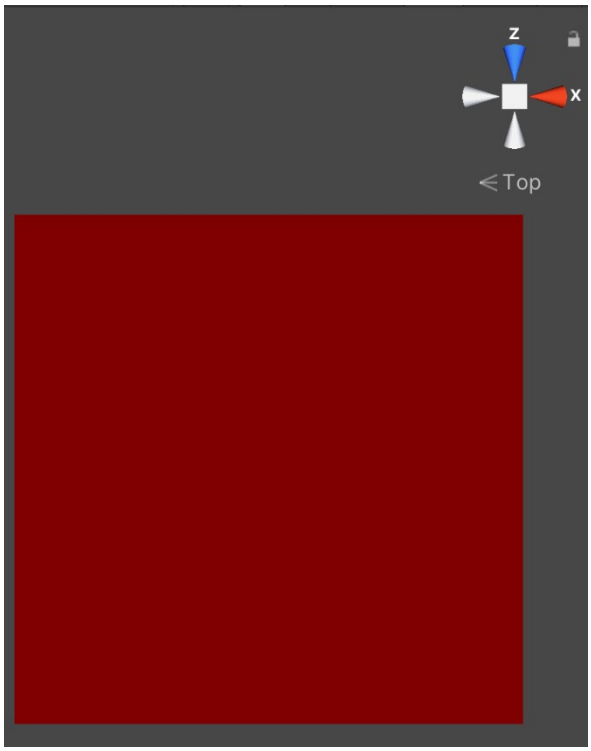
Slika 3.2: Ustvarjanje primerka senčilnika

Desno kliknemo na hierarhijo naših objektov in ustvarimo novo ravnino (Slika 3.3).



Slika 3.3: Ustvarjanje ravnine

Z miško lahko nanjo povlečemo material. Rezultat je rdeča ravnina (Slika 3.4).



Slika 3.4: Rezultat senčilnika

3.2 HLSL

Sedaj ko smo dobili občutek za HLSL, poglejmo nekaj njegovih lastnosti. Predvsem se bomo osredotočili na njegovo sintakso.

3.2.1 Podatkovni tipi

V HLSL lahko ustvarimo vektorje do vključno velikosti $\mathbb{R}^4$, kot tudi transformacijske matrike do vključno velikosti $\mathbb{R}^{4\times4}$ ter uporabljamo tako cela kot necela števila in logične operatorje. V zgornjem primeru se v `poljubnaBarva` shrani `vec4` s komponentami RGBA. Do komponent vektorja lahko dostopamo preko spremenljivk `x`, `y`, `z`, `w` ali `r`, `g`, `b`, `a`. Vidimo tudi, da so vrednosti barve senčilnika fragmentov normalizirane, kar pomeni, da so v intervalu od 0 do 1. Normalizacija je pogost pojav v senčilnikih in jo bomo velikokrat uporabili.

Podatkovna tipa `float` in `half` sta ključna pri pisanju senčilnikov, saj velikokrat želimo ustvariti navidezno zveznost med diskretnimi podatki. Če programiramo senčilnik za računalniške grafične procesorje med njima ni razlik, saj ti vedno delujejo na visokem nivoju natančnosti, medtem ko se na drugih napravah razlikujeta v nivoju natančnosti. Manjša natančnost pomeni hitrejše izrisovanje, vendar slabšo kvaliteto. V primeru iz

prejšnjega poglavja smo uporabili tip `float`, ki predstavlja visok nivo natančnosti, lahko pa bi uporabili tudi `half`, ki predstavlja srednjega. Običajno se podatkovni tipi `half` uporabljajo za vse razen za pozicije in UV koordinate.

Na začetku primera iz prejšnjega poglavja smo pod lastnosti navedli teksturo, ki je tudi podatkovni tip. Teksture je potrebno vzorčiti s posebnim podatkovnim tipom `sampler2D` oziroma `samplerCUBE` odvisno od tipa strukture. V raziskovalni nalogi so uporabljene le 2D teksture.

Zadnja pomembna podatkovna tipa sta generična medpomnilnika `StructuredBuffer<T>` in `RWStructuredBuffer<T>`, ki se uporabljata predvsem pri senčilnik za izračunavanje. Razlika med njima je, da je prvi samo bralni, drugi pa bralno-pisalni. Te medpomnilnike vedno ustvarimo v kodi izven HLSL.

Poleg teh podatkovnih tipov lahko definiramo tudi svoje strukture na enak način kot definiramo strukturi `Varyings` in `Attributes`.

3.2.2 Potek senčilnika

HLSL podpira vse običajne funkcionalnosti, ki jih poznamo v jezikih, kot so C. Med temi so podpora za `for` in `while` zanke, `if` ter `switch` stavke, pozna pa tudi poseben stavek `discard`, ki zavrže vrednost trenutnega fragmenta in je uporaben le v senčilniku fragmentov.

3.2.3 Funkcije in oznake

Funkcije definiramo na enak način kot pri jeziku C. Najprej navedeno podatkovni tip, ki ga funkcija vrne oziroma `void`, nato ime funkcije in nazadnje njene parametre navedene v oklepajih.

Funkcijam in spremenljivkam lahko dodamo semantične oznake. Te so potrebne na vseh podatkih, ki potujejo med nivoji grafičnega cevovoda. Vse oznake se dodajo tako, da se zapiše dvopičje in v naprej določeno ime oznake.

Jezik omogoča tudi uporabo komentarjev z `//` oziroma `/*` in `*/`, enako kot pri jeziku C.

Jezik podpira tudi predprocesorke makro ukaze, le-ti se izvršijo pred prevajanjem programa. Z njimi je mogoče definirati globalne spremenljivke, vključiti nove datoteke in razvejati program (z `#ifdef` in `#endif`). Vsi makro ukazi se začnejo z znakom `#`.

3.3 Enostavni primeri

Implementirajmo še nekaj enostavnih primerov, ki so z malo kreativnosti že dejansko uporabni.

3.3.1 Animirano 2D srce

Namen primera je, da spoznamo upodabljanje 2D likov oziroma funkcij.

Najprej bomo vključili še eno datoteko, ki (med drugim) vsebuje matematične funkcije, ki jih bomo uporabili za upodobitev srca (Izsek kode 3.8).

```
#include "Packages/com.unity.render-
pipelines.universal/Shaders/LitInput.hlsl"
```

Izsek kode 3.8: Vključitev dodatne datoteke

Tokrat bomo uporabili tudi teksturne UV koordinate, ki jih lahko pridobimo z uporabo semantične oznake `TEXCOORD0` znotraj strukture `Attributes`. Potrebovali jih bomo znotraj senčilnika fragmentov, zato jih bomo definirali tudi v strukturi `Varyings` in jih nastavili v senčilniku oglišč (Izsek kode 3.9).

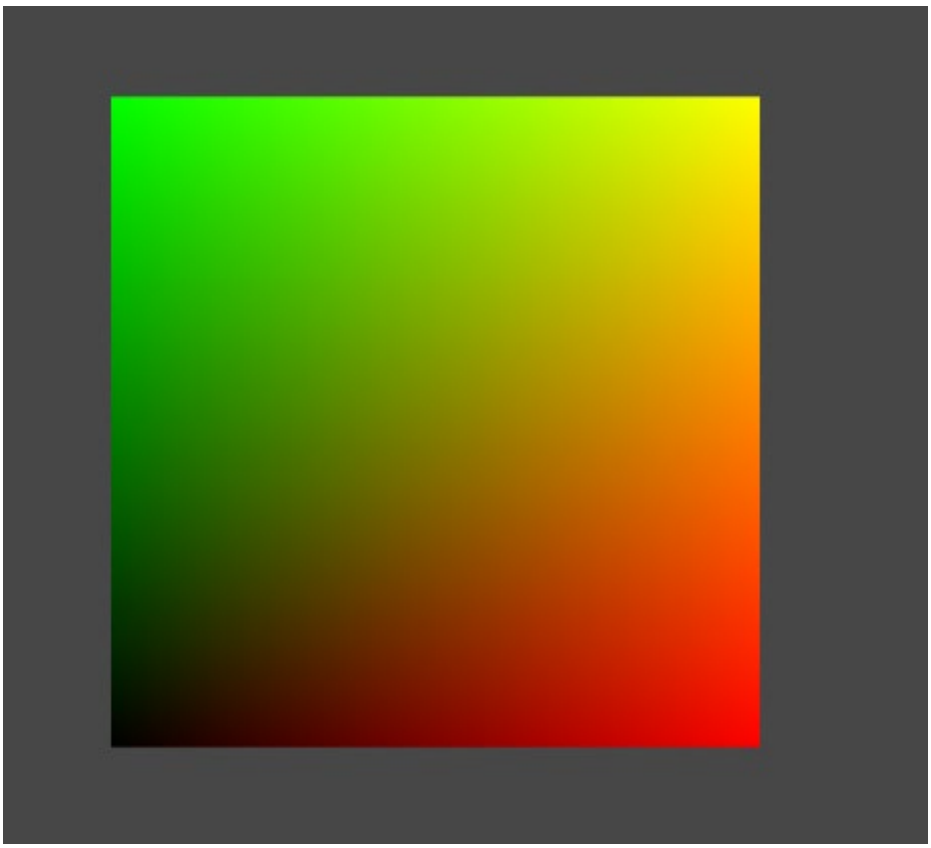
```
struct Attributes
{

    float4 pozicijeOS   : POSITION;
    float2 uv           : TEXCOORD0;
};

struct Varyings
{
    float2 uv          : TEXCOORD0;
    float4 pozicijeCS  : SV_POSITION;
};

Varyings vert(Attributes IN)
{

    Varyings OUT;

    OUT.pozicijeCS = TransformObjectToHClip(IN.pozicijeOS.xyz);
    OUT.uv = IN.uv;
    return OUT;
}
```

Izsek kode 3.9: Dodajanje UV koordinat in nov senčilnik oglišč

Tokrat želimo strukturo `Varyings` uporabiti tudi v senčilniku fragmentov, zato jo definiramo kot parameter (Izsek kode 3.10). Za preizkus delovanja vrnimo UV koordinate (Slika 3.5).

```
float4 frag(Varyings IN) : SV_Target
{
    return float4(IN.uv.x,IN.uv.y,0,1);
}
```

Izsek kode 3.10: UV koordinate v senčilniku fragmentov



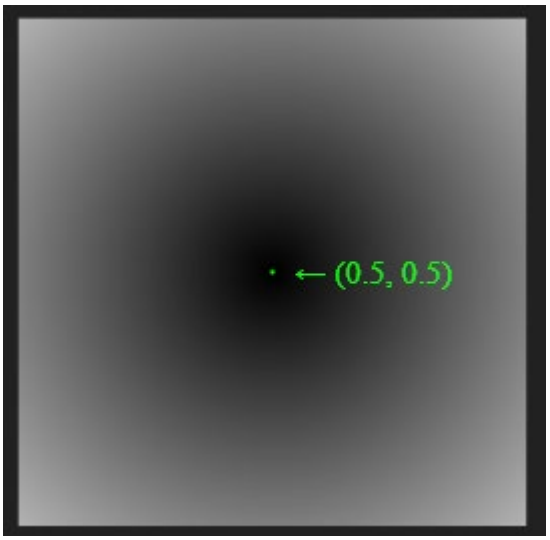
Slika 3.5: UV koordinate

Prva funkcija, ki jo bomo uporabili, nam vrne razdaljo med dvema vektorjema (Izsek kode 3.10).

```
float4 frag(Varyings IN) : SV_Target
{
    float razdalja = distance(float2(0.5,0.5),IN.uv);
    return razdalja;
}
```

Izsek kode 3.10: Uporaba razdalje

UV koordinate so normalizirane po obeh oseh, zato predstavlja vektor $\mathbf{v} = (0{,}5; 0{,}5)$ središče ravnine. Dobimo prehod (Slika 3.6).

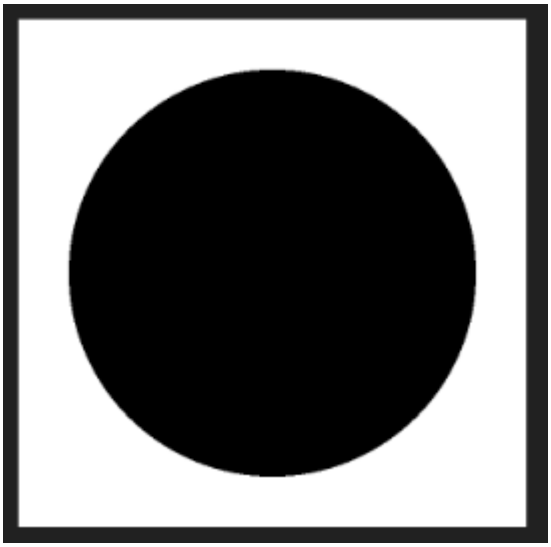


Slika 3.6: Dobljeni prehod

Sedaj bomo uporabili funkcijo `step(a, x)`, ki vrne vrednost 0, če je prvi argument večji od drugega, sicer vrne 1. Tako lahko ustvarimo krog, ki ima polmer enak prvemu argumentu funkcije (Izsek kode 3.11 in Slika 3.7).

```
float4 frag(Varyings IN) : SV_Target
{
    float razdalja = distance(float2(0.5,0.5),IN.uv);
    float polmer = 0.4;
    return step(polmer, razdalja);
}
```

Izsek kode 3.11: Uporaba funkcije `step`



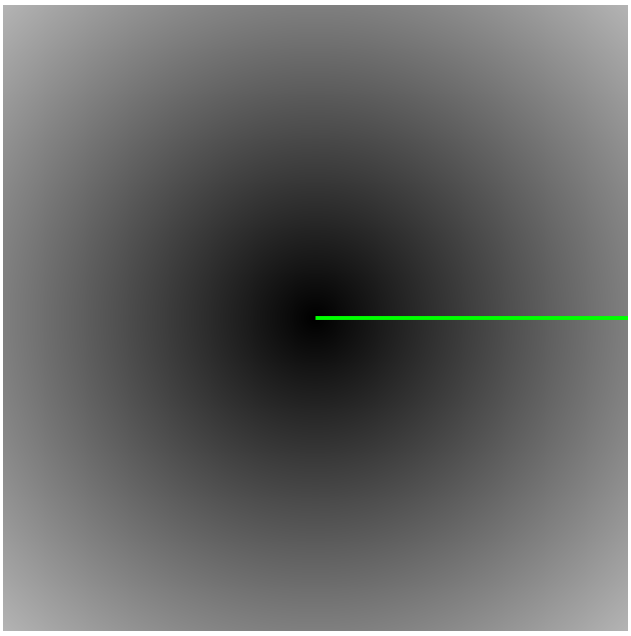
Slika 3.7: Ustvarjen krog

Uporabimo lahko tudi globalno Unity spremenljivko `_Time`, katere y komponenta vsebuje pretečen čas v sekundah. Če ga vnesemo v funkcijo sinus, in obdamo z absolutno vrednostjo, dobimo periodično funkcijo, ki predstavlja utripanje (Izsek kode 3.12).

```
float4 frag(Varyings IN) : SV_Target
{
    float razdalja = distance(float2(0.5,0.5),IN.uv);
    float polmer = abs(sin(_Time.y)) * 0.4;
    return step(polmer, razdalja);
}
```

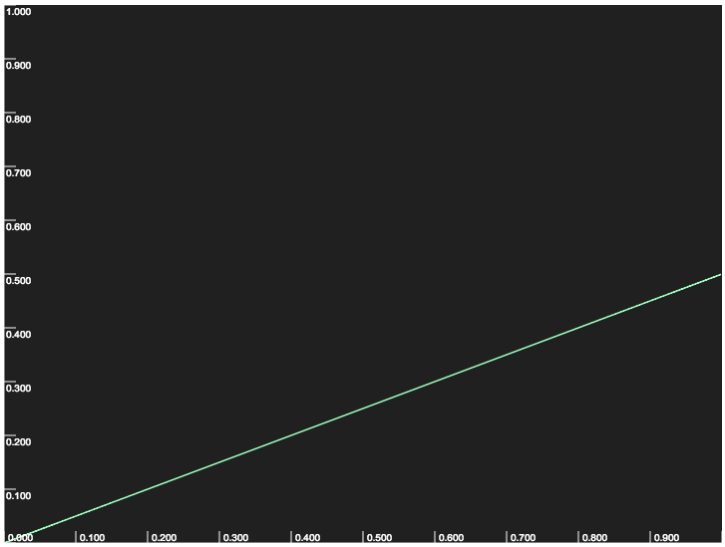
Izsek kode 3.12: Uporaba spremenljivke `_Time`

Še zadnji učinek, ki ga bomo uporabili, preden preidemo na oblikovanje srca, je premikajoča obroba. Trenutno naša spremenljivka `razdalja` izgleda tako (Slika 3.8).



Slika 3.8: Spremenljivka `razdalja`

Predstavimo jo lahko z naslednjim grafom (Slika 3.9).



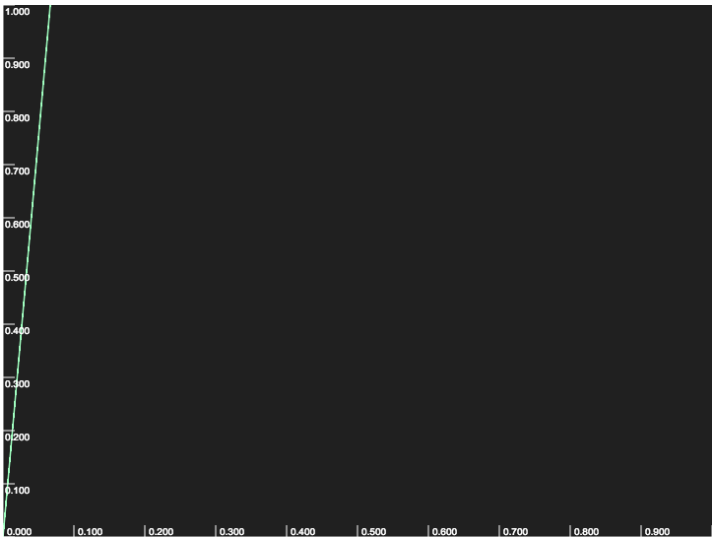
Slika 3.9: Graf v odvisnosti od razdalje

Če jo močno povečamo, je rezultat sledeč (Izsek kode 3.13 in Slika 3.10).

```
razdalja *= 30;
```

Izsek kode 3.13: Povečava po y osi

Slika 3.10: Močno povečana spremenljivka `razdalja`

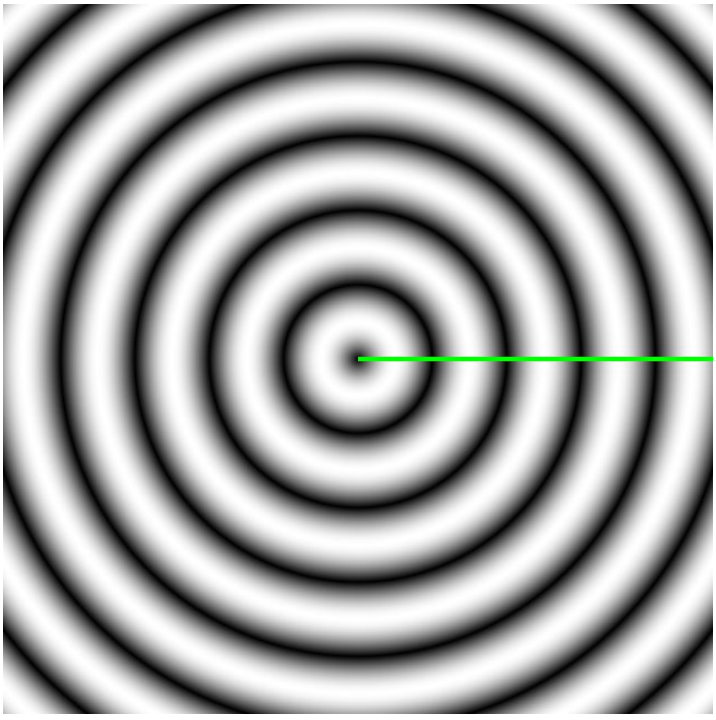


Slika 3.11: Graf v odvisnosti od močno povečane razdalje

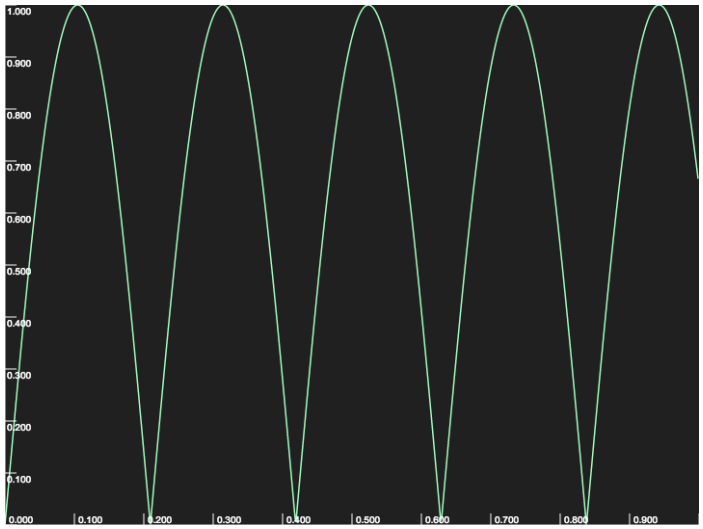
Nato lahko to razdaljo vnesemo v funkcijo sinus in jo obdamo z absolutno vrednostjo, da dobimo učinek obrobe (Izsek kode 3.14 in Sliki 3.12 ter 3.13).

```
razdalja = abs(sin(razdalja));
```

Izsek kode 3.14: Uporaba funkcije sinus

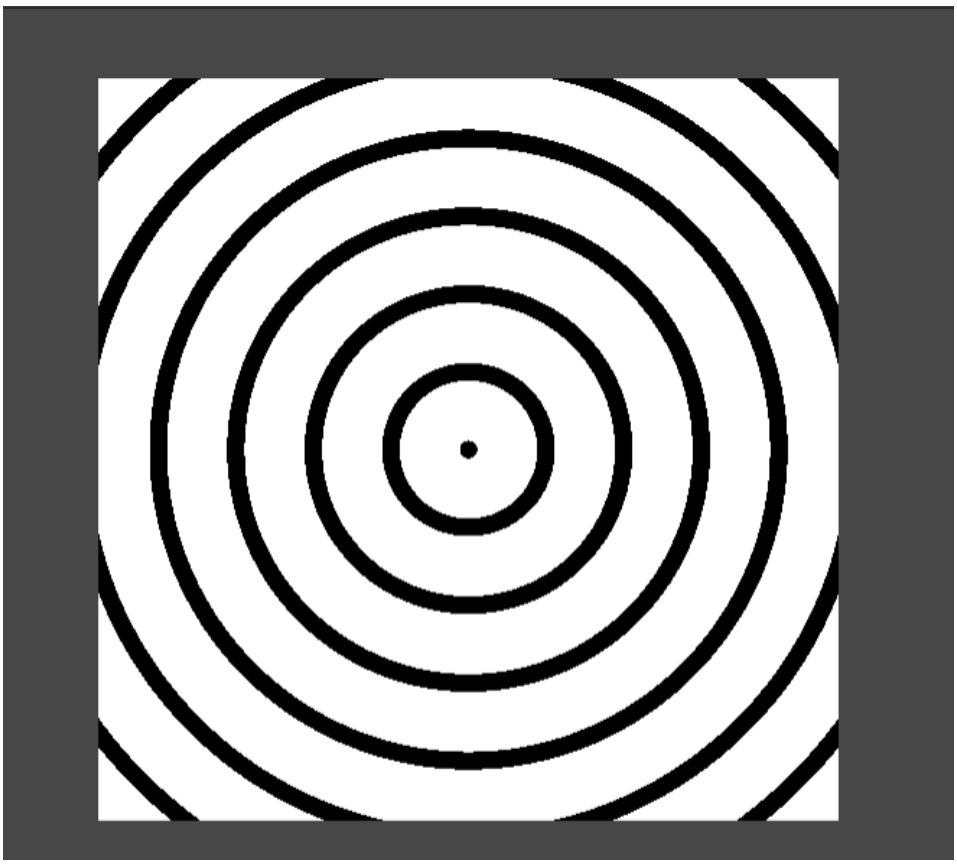


Slika 3.12: Učinek obrobe



Slika 3.13: Graf po uporabi absolutne vrednosti in funkcije sinus

Če to razdaljo vključimo v prejšnjo kodo, je rezultat sledeč (Slika 3.14).



Slika 3.14: Obroba brez gladkega prehoda

Na spletni strani (Project, 2021) sem našel sledečo implicitno enačbo srca:

$$x^2 + (y - \sqrt{|x|})^2 = 1$$

Zapišimo funkcijo, ki UV koordinate preslika v srce in jo uporabimo v senčilniku fragmentov (Izsek kode 3.15 in Slika 3.15).

```
float srce(float2 uv)
{
    uv = (uv - float2(0.5,0.38)) * float2(2.1,2.8); // centriranje srca

    return pow(uv.x,2) +
           pow(uv.y - sqrt(abs(uv.x)),2);
}


float4 frag(Varyings IN) : SV_Target
{
    float razdalja = srce(IN.uv);

    return step(razdalja, abs(sin(razdalja * 8 - _Time.y * 2)));
}
```

Izsek kode 3.15: Uporaba implicitne enačbe srca

Slika 3.15: Animirano srce

Za konec pomnožimo vse skupaj še s poljubno barvo (Izsek kode 3.16 in Slika 3.16).

```
float4 frag(Varyings IN) : SV_Target
{
    float razdalja = srce(IN.uv);

    return step(razdalja, abs(sin(razdalja * 8 - _Time.y * 2))) * float4(0.5,
0, 0, 1);

}
```

Izsek kode 3.16: Dodajanje barve

Slika 3.16: Dodajanje barv

To srce lahko že uporabimo kot animacijo nad igralcem v igri. Če želimo odstraniti ozadje, lahko uporabimo funkcijo `clip()`, ki zavrže fragmente, če je vrednost argumenta manjša od nič. Funkcijo lahko torej uporabimo na naslednji način (Izsek kode 3.17).

```
float4 frag(Varyings IN) : SV_Target
{
    float razdalja = srce(IN.uv);
    float barva = step(razdalja, abs(sin(razdalja * 8 - _Time.y * 2)));
    clip(barva - 0.00001);
    return barva * float4(0.5, 0, 0, 1);
}
```

Izsek kode 3.17: Odstranitev ozadja

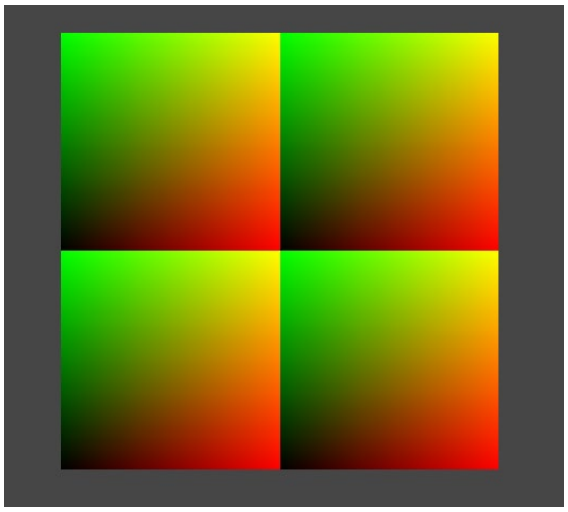
3.3.2 Preprost prehod

S senčilniki lahko ustvarimo tudi prehode, ki so primerni za uporabo tako v igrah kot v animacijah. Namen poglavja je spoznavanje novih funkcij, kot tudi pridobivanje občutka za praktično uporabo teh metod.

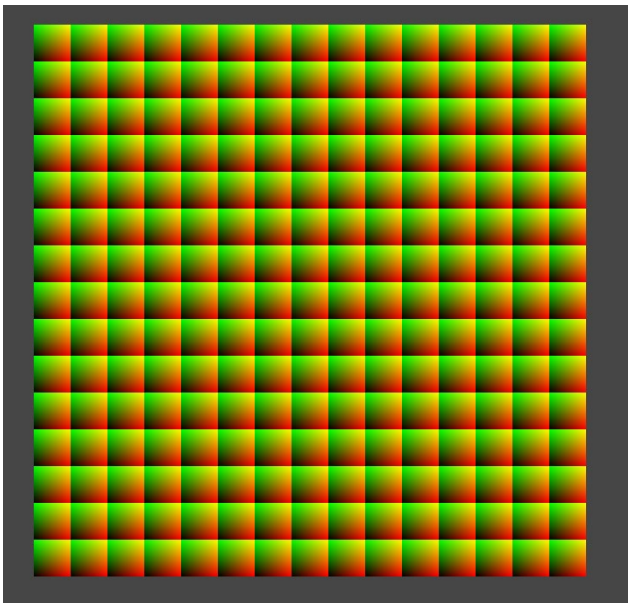
Pri tem primeru bomo prav tako uporabili UV koordinate, zato jih definiramo enako kot pri prejšnjem primeru (Izsek kode 3.8). Kot dodatek bomo uporabili funkcijo `frac(x)`, ki vrne decimalni del parametra. Pogosta tehnika s senčilniki je ponavljanje, ki se običajno doseže s periodičnimi funkcijami, kot so tudi na primer trigonometrijske funkcije in funkcija modulo. Ker vemo, da so UV koordinate normalizirane, jih lahko "razrežemo" na n^2 enakih delov tako, da jih pomnožimo s poljubno vrednostjo n in izvršimo funkcijo `frac` (Izsek kode 3.18 in Sliki 3.17 ter 3.18).

```
float4 frag(Varyings IN) : SV_Target
{
    float n = 2;
    float2 ostanek = frac(IN.uv * n);
    return float4(ostanek.x, ostanek.y,0,1);
}
```

Izsek kode 3.18: Ponavljanje UV koordinat



Slika 3.17: Ponavljajoče UV koordinate

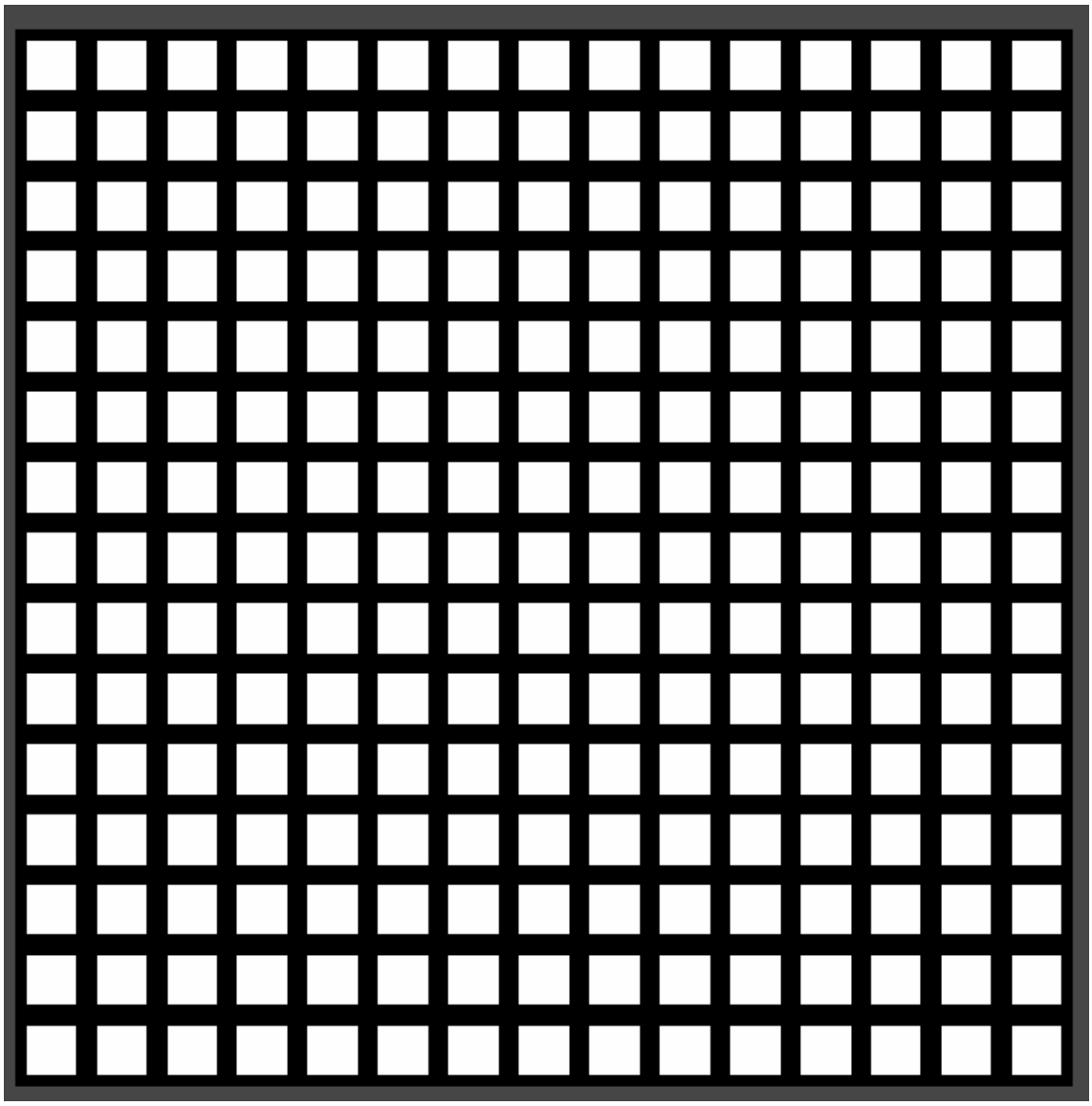


Slika 3.18: Ponavljanje pri vrednosti $n = 15$

Nastale kvadratke bomo uporabili za prehod. Želimo jim določiti poljubno barvo in jih povečati ter pomanjšati skozi trajanje prehoda. Za boljšo preglednost bomo definirali funkcijo `kvadrat`, ki bo realizirala implicitno enačbo kvadrata na podoben način, kot smo pri prejšnjem primeru definirali srce. Uporabili bomo tudi funkcijo `step`, da se znebimo gladkega barvnega prehoda (Izsek kode 3.19 in Slika 3.19).

```
float kvadrat(float2 uv, float stranica)
{
    stranica = 0.5 + stranica * 0.5; // centriranje kvadrata
    uv = step(uv, stranica) * step(1.0 - uv, stranica);
    return uv.x * uv.y;
}

float4 frag(Varyings IN) : SV_Target
{
    float n = 15;
    float2 ostanek = frac(IN.uv * n);
    return kvadrat(ostanek, 0.8);
}
```

Izsek kode 3.19: Definiranje funkcije `kvadrat`



Slika 3.19: Dobljeni kvadratki

Velikost stranice lahko nato spreminjamo skozi čas z uporabo funkcije sinus (Izsek kode 3.20).

```
float stranica = abs(sin(_Time.y));
return kvadrat(ostanek,stranica);
```

Izsek kode 3.20: Spreminjanje velikosti stranice skozi čas

Učinek lahko naredimo veliko zanimivejši tako, da dodamo malo rotacije in funkciji sinus dodamo fazni zamik, ki je odvisen od oddaljenosti od središča UV koordinat. Prvi učinek dosežemo tako, da definiramo novo funkcijo `rotiraj` in nad UV koordinatami opravimo rotacijo tako, da jih pomnožimo z rotacijsko matriko, ki jo poznamo iz linearne algebre. Paziti moramo, da koordinate pravilno centriramo, saj želimo, da se rotacija izvede okrog

središča. To funkcijo nato uporabimo med definicijo kvadrata tako, da kot spreminjamo v odvisnosti od dolžine stranice (Izsek kode 3.21).

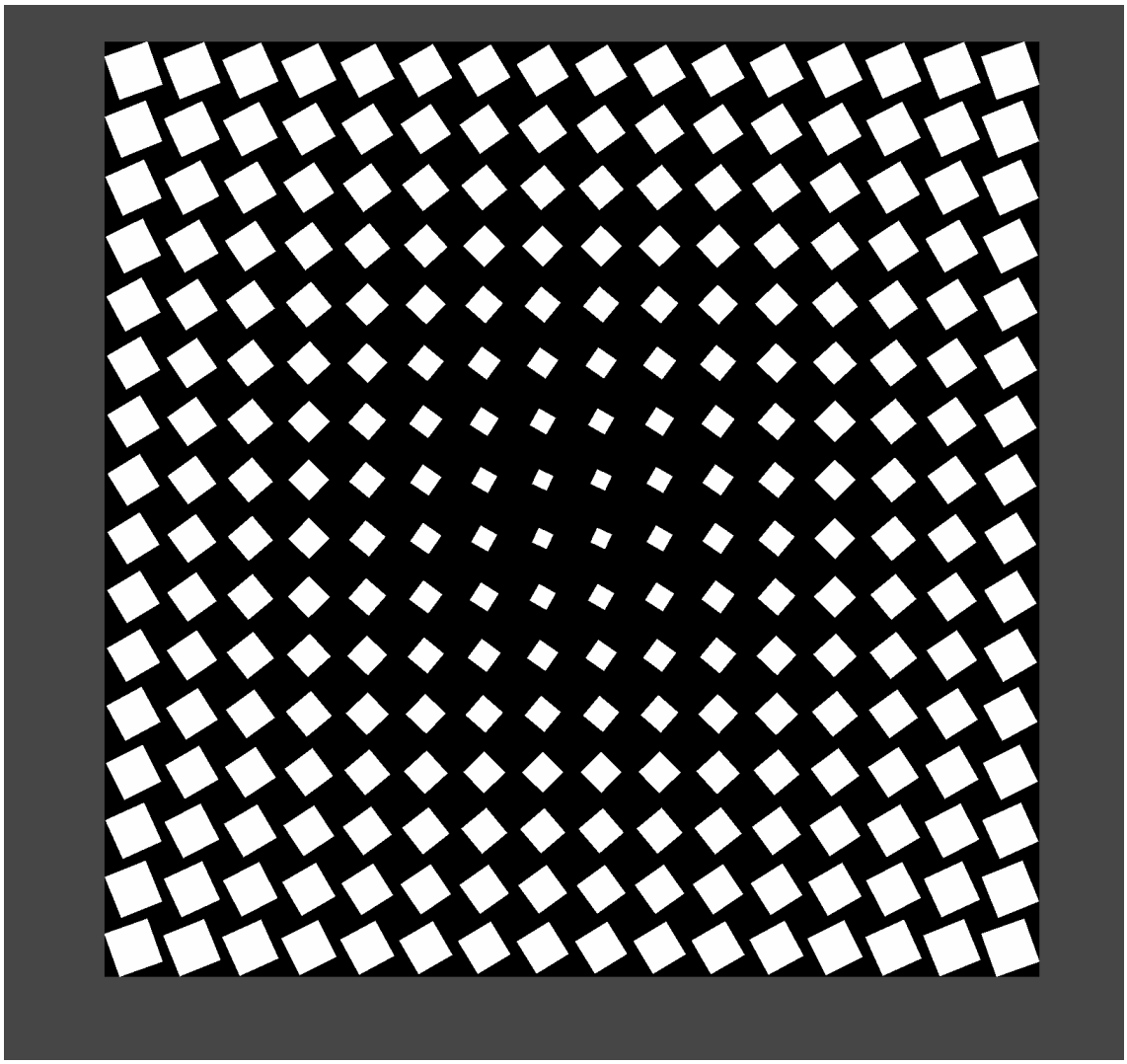
```
#define PI 3.14159265359

float2 rotiraj(float2 uv, float kot)
{
    float2x2 matrika = float2x2(cos(kot), -sin(kot),
                    sin(kot), cos(kot));
    uv -= 0.5;
    uv = mul(matrika, uv);
    uv += 0.5;
    return uv;
}


float kvadrat(float2 uv, float stranica)
{
    uv = rotiraj(uv, stranica * PI / 2);
    stranica = 0.5 + stranica * 0.5; // centriranje kvadrata
    uv = step(uv, stranica) * step(1.0 - uv, stranica);
    return uv.x * uv.y;
}
```

Izsek kode 3.21: Uporaba rotacijske matrike

Za konec še implementirajmo fazni zamik. Ta učinek dosežemo tako, da uporabimo funkcijo `length`, ki vrne dolžino vektorja. Uporabimo tudi funkcijo `floor`, ki vrne decimalno število zaokroženo navzdol. Na ta način lahko posameznim stranicam, ki pripadajo enemu kvadratku, dodelimo enako dolžino med trajanjem transformacije (Izsek kode 3.22 in Slika 3.20).

```
float4 frag(Varyings IN) : SV_Target
{
    float n = 16;
    float2 ostanek = frac(IN.uv * n);
    float zamik = length(0.5 - (floor(IN.uv * n) + 0.5) / n);
    float stranica = abs(sin(_Time.y + zamik));
    return kvadrat(ostanek,stranica);
}
```

Izsek kode 3.22: Implementacija faznega zamika



Slika 3.20: Dokončan prehod

3.3.3 Deformacija definicijskega območja

Kot zadnji 2D učinek si poglejmo obliko deformacije, ki se pogosto uporablja s senčilniki. Imenuje se deformacija definicijskega območja (angl. domain warping), omogoča pa nam dodajanje organske kompleksnosti teksturam. Njena naloga je, da zakrijemo urejenost šuma (angl. noise), ki ga uporabimo. Glavni namen poglavja je, da spoznamo še preostale pomembne funkcije in uporabo naključnosti v senčilnikih. Ta učinek je bolj kompleksen, zato bomo njegovo implementacijo razdelili na štiri dele. Najprej bomo implementirali funkcijo za naključne vrednosti.

V jeziku HLSL ne obstaja funkcija `random()`, zato bomo zapisali svojo (Izsek kode 3.23). Predstavljala bo razpršilno funkcijo, ki je psevdonaključna. Naključnost je v programiranju obširno področje, zato bom to funkcijo razložil na kratko.

```
float random(float2 uv)
{
    return frac(sin(dot(uv.xy, float2(12.9898, 78.233))) * 43758.5453123);
}
```

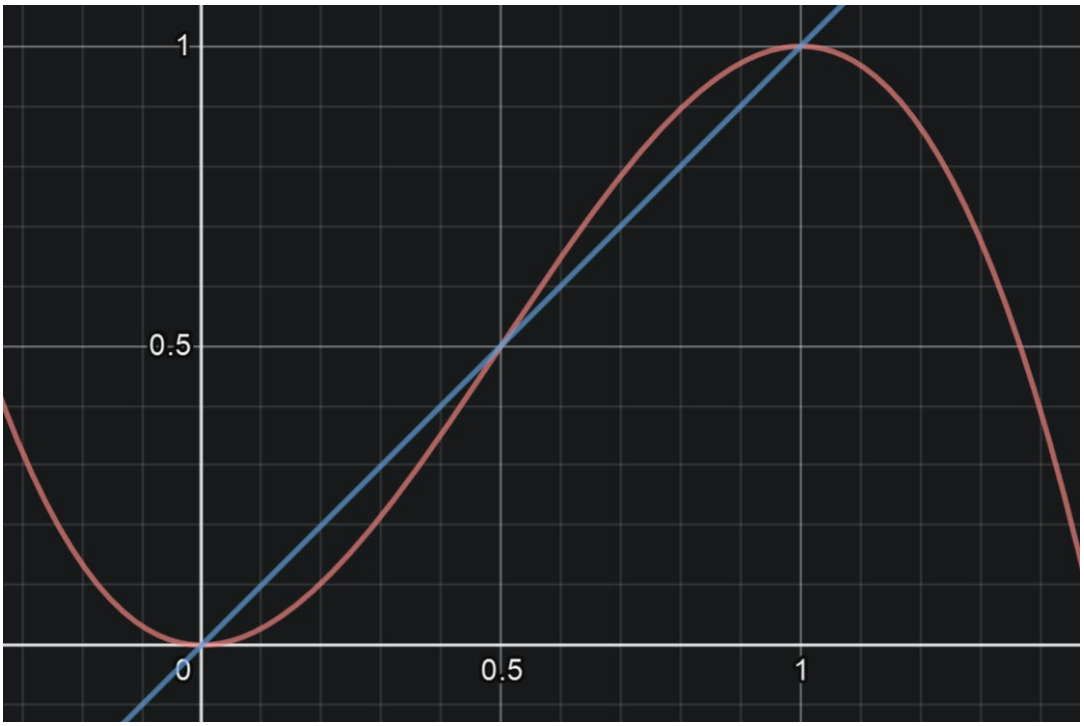
Izsek kode 3.23: Funkcija za naključne vrednosti

Uporaba funkcij `frac` in `sin` služi kot dobra aproksimacija naključnosti pri dovolj velikih vrednostih. Če je perioda sinusne funkcije dovolj velika, bo ob vzorčenju kompozitum prestavljal psevdonaključna števila. Skalarni produkt služi predvsem za pretvorbo dvodimenzionalnega vektorja UV koordinat v število. Poleg tega preslikamo funkcijo iz intervala (-1,1) na veliko večji interval, saj bi sicer lahko opazili ponavljanje. Števila uporabljena v funkciji so poljubna, vendar je priporočljivo, da so zelo blizu praštevil. Običajno se uporabijo števila, ki so prikazana zgoraj (Izsek kode 3.23), zaradi raznih podrobnosti iz teorije števil.

Nato bomo implementirali šum. Zanj bomo uporabili pogosto uporabljeno funkcijo vrednostnega šuma. Na ta način bomo uporabili odvode, kar je bolj natančno in hitrejše od drugih metod za ustvarjanje šumov. Uporabili bomo odvod polinoma:

$$p(x) = 3x^2 - 2x^3$$

Ta polinom izberemo zato, ker njegova zaloga vrednosti vsebuje vrednosti $\frac{1}{2}$ in 1, njegov odvod pa predstavlja polinom sode stopnje z ničlama 0 ter 1 in temenom pri $x = \frac{1}{2}$. Te lastnosti zagotavljajo nelinearno simetrično rast od 0 do 1, kar vodi v večjo kompleksnost, ki bolj spominja na resnično življenje. Na Sliki 3.21 modra barva predstavlja funkcijo $y = x$, rdeča pa funkcijo $y = 3x^2 - 2x^3$. Razliko v izgledu lahko vidimo na spodnjih slikah (Sliki 3.22 in 3.23).



Slika 3.21: Primerjava grafov

To seveda ni edini polinom s temi lastnostmi, vendar ima najbolj preprosto obliko ter najnižjo stopnjo in ga zato najhitreje izračunamo.

Z izbranim polinomom bomo izvedli bilinearno interpolacijo po UV koordinatah.

Funkcija `lerp(x, y, s)` linearno interpolira vrednost prvega parametra do drugega na podlagi skalarja. Definirana je kot:

$$lerp(x, y, s) = x + s(y - x)$$

V našem primeru potrebujemo bilinearno interpolacijo, saj bomo šum uporabili z UV koordinatami (Izsek kode 2.24). To pomeni, da bomo izvedli linearno interpolacijo najprej po smeri x in nato še po y. Podobno kot pri prejšnjem primeru, bomo UV koordinate tudi tokrat omejili na interval (-1,1) z uporabo funkcije `frac`. Šum bomo nanesli v več slojih z metodo fBm (Fractional Brownian Motion), in sicer tako, da bomo za vsak sloj povečali število kvadratkov. Naključne vrednosti šuma bomo izračunali le pri ogliščih posameznega kvadrata in jih bilinearno interpolirali po celotnem kvadratu, zato uporabimo funkcijo `floor` (Izseka kode 3.24 in 3.25).

```
float noise(float2 uv)
{
    float2 uvNavzdol = floor(uv);
    uv = frac(uv);

    float a = random(uvNavzdol);
    float b = random(uvNavzdol + float2(1, 0));
    float c = random(uvNavzdol + float2(0, 1));
    float d = random(uvNavzdol + float2(1, 1));

    float2 polinom = uv * uv * (3 - 2 * uv);

    return lerp(lerp(a, b, polinom.x), lerp(c,d, polinom.x),polinom.y);
}
```

Izsek kode 3.24: Implementacija funkcije za šum

```
float4 frag(Varyings IN) : SV_Target
{

    return noise(IN.uv*20);
}
```

Izsek kode 3.25: Uporaba funkcije šuma

Slika 3.22: Šum z linearno funkcijo

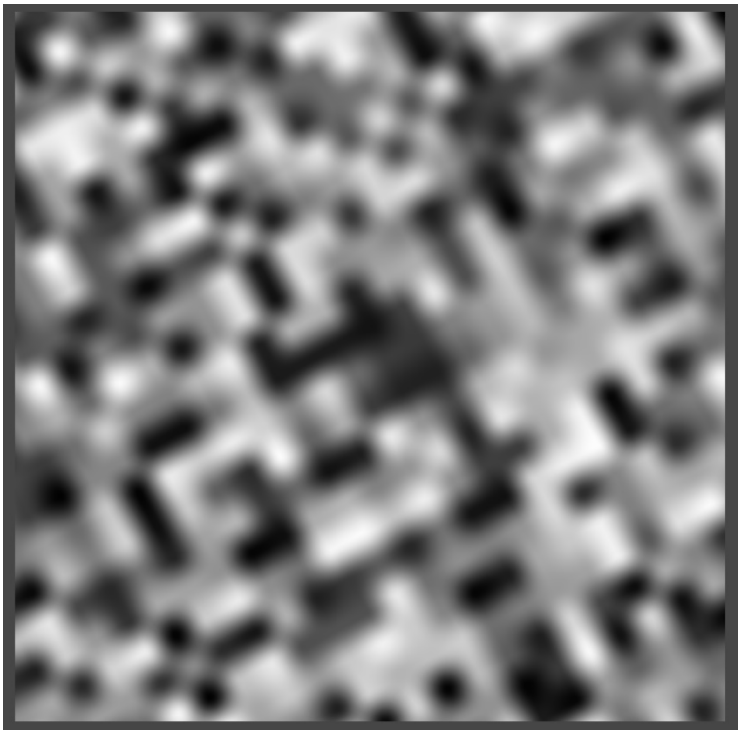
Slika 3.23: Šum z izbranim polinomom

Vrednosti so pravilne, vendar so zelo izrazite, saj smo uporabili le eno plast. Za bolj organski prikaz šuma bomo sedaj uporabili metodo fBm[3], ki predstavlja integral nad plastmi šuma. Plasti se formalno imenujejo oktave, saj tukaj podobno, kot pri tonih, med dvema zaporednima oktavama podvojimo frekvenco. Plastem bomo tudi eksponentno manjšali amplitudo šuma tako, da bomo vsako plast pomnožili s skalarji. Ker vemo, da so vrednosti barve senčilnika fragmentov normalizirane, bomo za skalarje izbrali vrednosti, ki pripadajo geometrijski vrsti $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\cdots$, saj vrsta konvergira k 1. Vsaki plasti tudi povečamo frekvenco tako, da povečamo število kvadratkov oziroma povečamo UV koordinate. Nanesli bomo sedem oktav, saj so po tej vrednosti skalarji preveč blizu 0, da bi opazili razliko. Da dodamo še več naključnosti našemu šumu, bomo za vsak sloj UV koordinate tudi premaknili in rotirali (Izseki kode 3.26, 3.27 ter 3.28 in Sliki 3.24 ter 3.25).

[3] Implementirano po: Quilez (2021)

```
float4 frag(Varyings IN) : SV_Target
{
    float2x2 matrika = float2x2(cos(0.5), sin(0.5), -sin(0.5), cos(0.5));
    return noise(mul(matrika,IN.uv*20));
}
```

Izsek kode 3.26: Implementacija rotacije



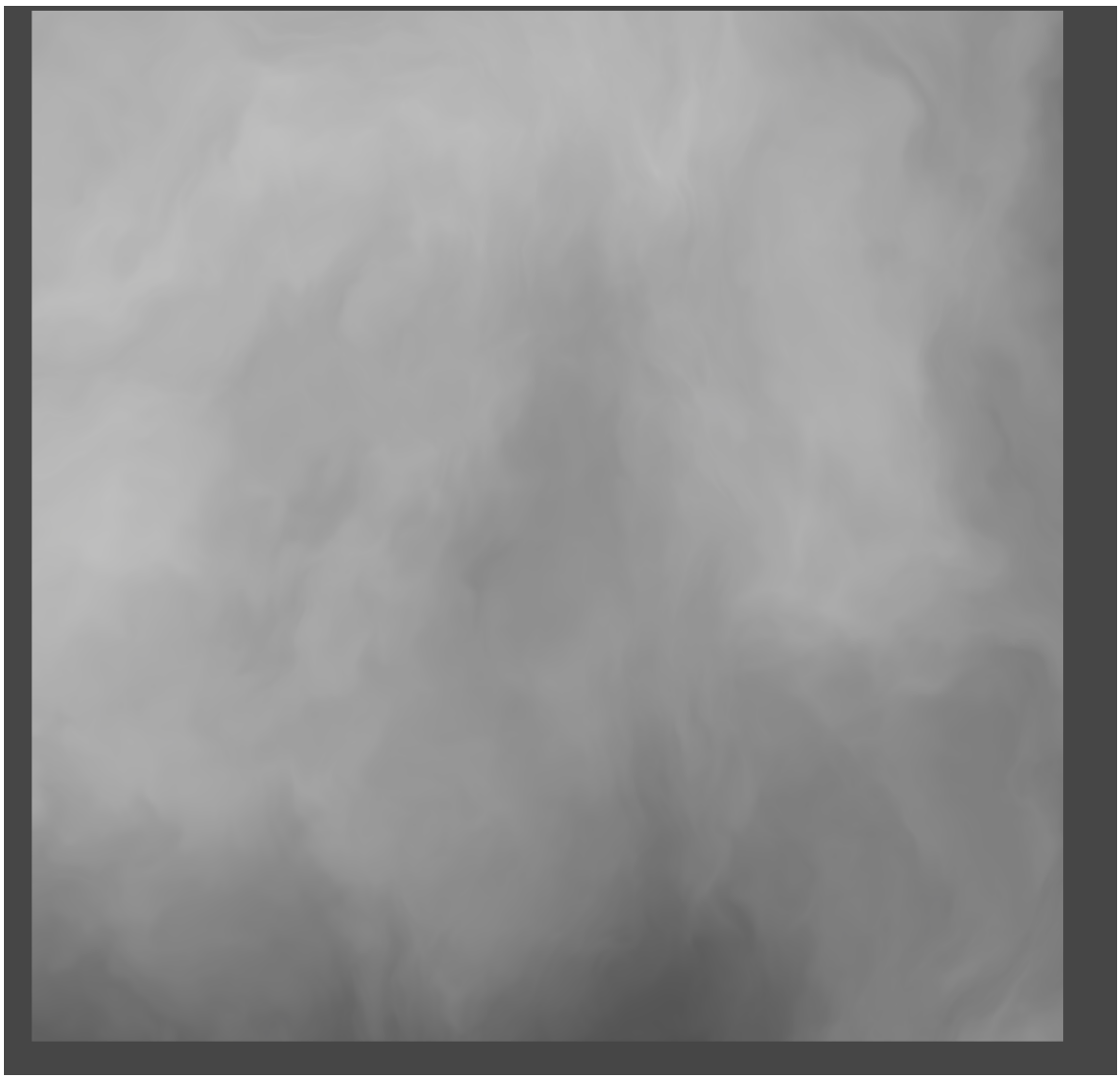
Slika 3.24: Rotacija nad šumom

```
float fbm(float2 uv)
{
    float integral = 0.0;
    float skalar = 0.5;
    float2 premik = 100.0;

    float2x2 matrika = float2x2(cos(0.5), sin(0.5), -sin(0.5), cos(0.5));

    for (int i = 0; i < 7; ++i)
    {
        integral += skalar * noise(uv);
        uv = mul(matrika, uv) * 2.0 + premik;
        skalar *= 0.5;
    }
    return integral;
}
```

Izsek kode 3.27: Funkcija fBm

```
float4 frag(Varyings IN) : SV_Target
{
    return fbm(IN.uv);
}
```

Izsek kode 3.28: Uporaba funkcije fBm



Slika 3.25: Prikaz slojev šuma

Za konec bomo še implementirali deformacijo definicijskega območja[4] funkcije fBm, in sicer v treh slojih. Kot pove že samo ime, je cilj te metode deformirati definicijsko območje tako, da uporabimo čim bolj naključne vrednosti, zato bomo izvedli kompozitum funkcij fBm, pri čemer bomo vsakemu klicu dodali poljuben zamik. Deformacijo bomo izvajali tudi v odvisnosti od časa (Izsek kode 3.29). Za zanimivejši učinek bomo linearno

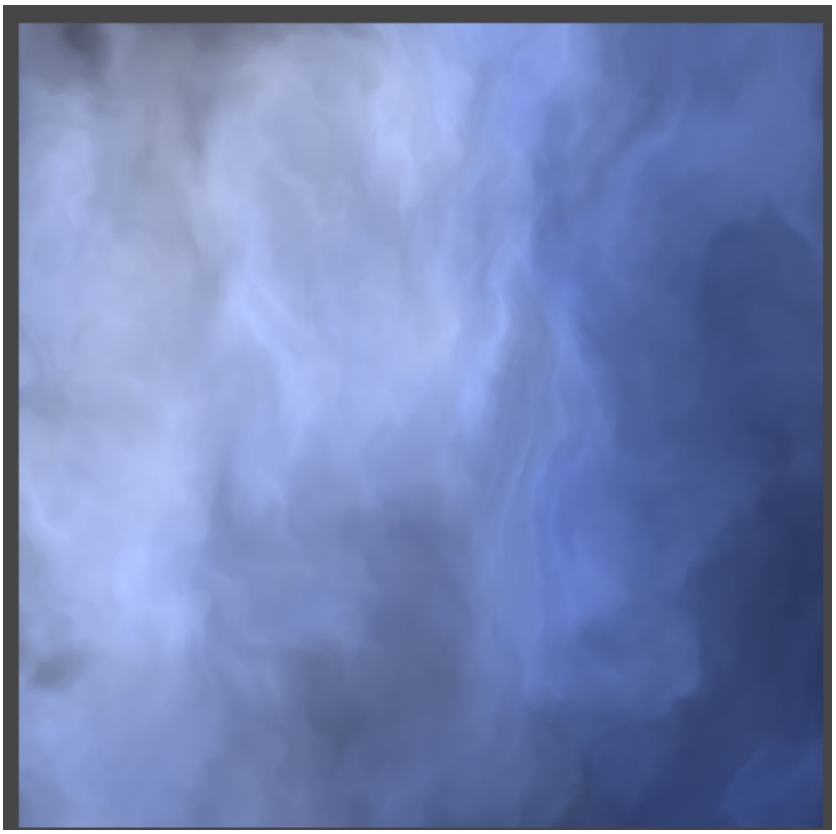
[4] Implementirano po: Quilez (2021)

interpolirali med dvema barvama na podlagi šuma in za konec še potencirali, da bo učinek izrazitejši (Sliki 3.26 in 3.27).

```
float4 _Barva;
float4 _Barva2;

float4 frag(Varyings IN) : SV_Target
{
    float t = _Time.y;


    float2 prvaDeformacija = 0;
    prvaDeformacija.x = fbm(IN.uv);
    prvaDeformacija.y = fbm(IN.uv + 1);

    float2 drugaDeformacija = 0;
    drugaDeformacija.x = fbm(IN.uv + 2.0 * prvaDeformacija + float2(1.7, 9.2)
+ 0.15 * t);
    drugaDeformacija.y = fbm(IN.uv + 2.0 * prvaDeformacija + float2(8.3, 2.8)
+ 0.126 * t);

    float koncnaDeformacija = fbm(IN.uv + drugaDeformacija);

    float3 barva1 = lerp(_Barva,
                _Barva2,
                saturate(koncnaDeformacija * koncnaDeformacija * 4));

    float3 barva2 = lerp(_Barva,
                barva1,
                saturate(length(prvaDeformacija)));

    float3 barva3 = lerp(_Barva,
                barva2,
                saturate(length(drugaDeformacija.x)));

    return float4(saturate((koncnaDeformacija * koncnaDeformacija *
koncnaDeformacija * koncnaDeformacija + 0.6 * koncnaDeformacija *
koncnaDeformacija + 0.5 * koncnaDeformacija) * barva3),1);
}
```

Izsek kode 3.29: Implementacija deformacije definicijskega območja

Slika 3.26: Prvi prikaz deformacije definicijskega območja



Slika 3.27: Drugi prikaz deformacije definicijskega območja

4 Tradicionalne senčilne metode

Najobičajnejša uporaba senčilnika je simuliranje interakcije svetlobe z objektom. Kljub temu da je osvetljevanje že implementirano v večini aplikacijah, je razumevanje principov osvetljevanja uporabno, kot bomo lahko videli kasneje pri alternativnih metodah.

4.1 Senčilnik osvetlitve

Za demonstracijo bomo implementirali preprost senčilnik osvetlitve. Najprej bomo vključili novo datoteko, ki vsebuje podatkovno strukturo `Light` (Izsek kode 4.1).

```
#include "Packages/com.unity.render-
pipelines.universal/ShaderLibrary/Lighting.hlsl"
```

Izsek kode 4.1: Posodobljene strukture

Pri tem primeru bomo potrebovali podatek o normalah objekta, zato jih definiramo v naših strukturah (Izsek kode 4.2).

```
struct Attributes
{
    float4 pozicijaOS   : POSITION;
    float3 normalaOS    : NORMAL;
};

struct Varyings
{
    float4 pozicijaCS    : SV_POSITION;
    float3 normalaWS     : NORMAL;
};
```

Izsek kode 4.2: Posodobljene strukture

Ker osvetljevanje poteka v realnem prostoru, bomo normale v senčilniku oglišč pretvorili v realni prostor preko metode podobno, kot to storimo s pozicijami (Izsek kode 4.3).

```
Varyings vert(Attributes IN)
{

    Varyings OUT;
    OUT.pozicijaCS = TransformObjectToHClip(IN.pozicijaOS.xyz);
    OUT.normalaWS = TransformObjectToWorldNormal(IN.normalaOS);
    return OUT;
}
```

Izsek kode 4.3: Posodobljen senčilnik oglišč

V senčilniku fragmentov potrebujemo informacije o svetlobnih virih sveta. Uporabili bomo le en svetlobni vir, ki ga lahko shranimo v podatkovni tip `Light` preko metode `GetMainLight()`.

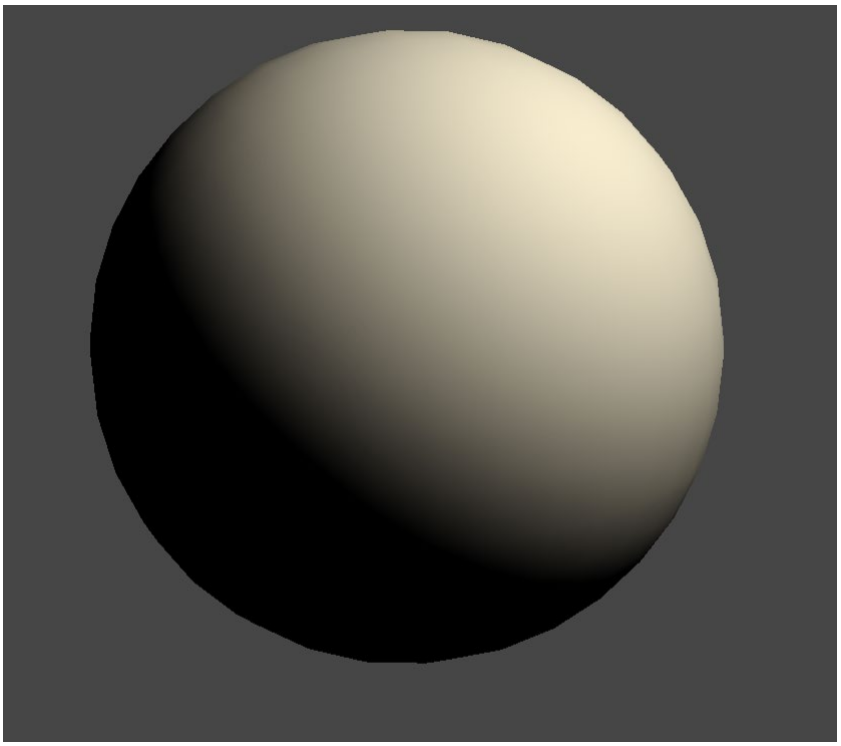
Zapisati moramo še funkcijo za osvetlitev. Uporabili bomo najpreprostejši model osvetljevanja, ki se imenuje Lambert, pri katerem bomo upoštevali le razpršen odboj svetlobe po površini objekta oziroma difuzni odboj (angl. indirect diffuse). Lambertov model se imenuje po Lambertovem kosinusnem izreku, ki pravi, da je odboj od površine sorazmeren s kosinusom med normalo površine in vektorjem smeri svetlobe od površine objekta. Pri tem modelu predpostavljamo, da je odboj svetlobe po površini enakomeren, se pravi, da površina ni gladka.

Najprej izračunamo skalarni produkt med normalo in smerjo svetlobe, ki jo pridobimo iz podatkovnega tipa `Light`. Na podlagi tega izračuna vemo, kdaj je posamezni fragment v senci. Z metodo `saturate` bomo omejili vrednosti skalarnega produkta med 0 in 1. To je potrebno storiti, saj lahko imamo probleme pri množenju z negativnimi vrednostmi. Dobljeno vrednost pomnožimo še z vrednostmi, ki predstavljajo slabljenje svetlobe in senc. S to kalkulacijo zagotovimo pravilna bela in črna območja, pri katerih je predmet neposredno osvetljen ali v senci, tudi ko spreminjamo nastavitve svetlobnega vira v urejevalniku. Za konec še vrednost pomnožimo z barvo svetlobe in funkcijo uporabimo v senčilniku fragmentov (Izsek kode 4.4).

```
half3 Osvetlitev(Light svetloba, half3 normala)
{
    half neposrednaRazprsenost = saturate(dot(normala, svetloba.direction));
    half3 osvetlitev = svetloba.color * svetloba.shadowAttenuation *
svetloba.distanceAttenuation;
    half3 rezultat = neposrednaRazprsenost * osvetlitev;
    return rezultat;
}

float4 frag(Varyings IN) : SV_Target
{
    Light svetloba = GetMainLight();
    return half4(Osvetlitev(svetloba, IN.normalaWS), 1);
}
```

Izsek kode 4.4: Implementacija funkcije za osvetlitev in senčilnik fragmentov

Rezultat je popolnoma enak kot privzeti senčilnik osvetlitve v Unity, le, da je naš preprostejši (Slika 4.1).



Slika 4.1: Rezultat senčilnika za osvetlitev

4.2 Bleščice[5]

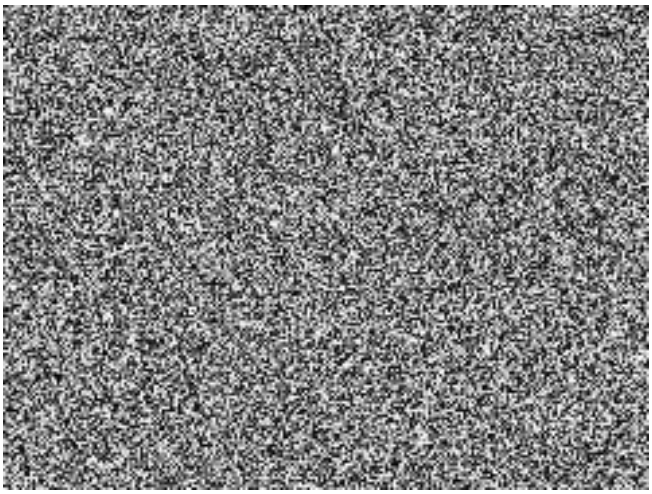
Da spoznamo več o osvetljevanju s senčilniki, bomo implementirali še senčilnik za upodobitev objekta posutega z bleščicami, pri čemer bomo uporabili popolnejši model osvetljevanja, ki se imenuje Blinn-Phong. Ta model upošteva zrcalni odboj (angl. specular reflection) in ambientno svetlobo (angl. ambient lighting).

Zrcalni odboj je v resničnem svetu značilen za vse objekte, ki imajo gladko površino. Za razliko od difuznega je zrcalni odboj odvisen še od smeri gledanja.

Z ambientno svetlobo se lahko znebimo temnih delov objektov. V našem primeru bomo izvedli linearno interpolacijo med barvo sence, ki predstavlja ambientno svetlobo in barvo objekta, ki se včasih imenuje tudi albedo, kar po latinsko pomeni belina.

Za bleščice bomo vzorčili teksturo belega šuma, saj je najbolj podoben bleščicam (Slika 4.2).

[5] Implementirano po: (WALKINGFAT, 2018)



Slika 4.2: Tekstura belega šuma

Beli šum bi lahko proizvedli sami tako, da bi kodi dodali en sloj šuma z močno povečanimi UV koordinatami, vendar sem želel prikazati tudi, kako se vzorči teksturo, saj si v praksi običajno vrednosti določenega šuma shranimo v teksturo za bodočo uporabo. V Unity moramo poleg tipa sampler2D dodati še 4D vektor, ki ima enako ime kot tekstura in pripono `_ST`, kar označuje stanje med vzorčenjem (angl. sampler state).

Najprej definirajmo vse potrebne spremenljivke, ki jih bomo potrebovali (Izsek kode 4.5).

```
sampler2D _NoiseTexture;
float4 _NoiseTexture_ST;
float4 _Barva, _BarvaSence, _FresnelBarva, _BarvaBlescic;
float _MocZrcalnegaOdboja, _Sijaj, _NoiseSize, _HitrostBlescic;
float _MocFresnela, _IntenzivnostFresnela, _HitrostBlescicZrcalnegaOdboja,
_HitrostBlescicFresnel, _HitrostBlescicDifuznegaOdboja, _IntenzivnostBlescic;
```

Izsek kode 4.5: Definirane spremenljivke

Kot smo omenili, je za ta osvetljevalni model pomembna tudi smer pogleda od objekta do kamere, zato jo izračunamo v senčilniku oglišč (pazimo na pravilno odštevanje vektorjev) (Izsek kode 4.7). Vektor normaliziramo, saj nas zanima le smer. Rezultat želimo poslati v senčilnik fragmentov, zato je potrebno spremeniti strukturo (Izsek kode 4.6).

```
struct Attributes
{
    float4 pozicijaOS : POSITION;
    float3 normalaOS : NORMAL;
    float2 uv : TEXCOORD0;
};

struct Varyings
{
    float4 pozicijaCS : SV_POSITION;
    float3 normalaWS : NORMAL;
    float2 uv : TEXCOORD0;
    float3 smerPogleda : TEXCOORD1;
};
```

Izsek kode 4.6: Spremenjene strukture

```
Varyings vert(Attributes IN)
{
    Varyings OUT;
    OUT.pozicijaCS = TransformObjectToHClip (IN.pozicijaOS);
    OUT.normalaWS = TransformObjectToWorldNormal(IN.normalaOS);
    OUT.uv = TRANSFORM_TEX(IN.uv, _NoiseTex);

    OUT.smerPogleda = normalize(_WorldSpaceCameraPos.xyz -
TransformObjectToWorld(IN.pozicijaOS.xyz));

    return OUT;
}
```

Izsek kode 4.7: Izračun smeri pogleda

Slabljenje svetlobe bomo tokrat definirali kot samostojno spremenljivko, saj bomo izračun večkrat uporabili (Izsek kode 4.8).

```
float3 Osvetlitev(Light svetloba, half3 normala, half3 smerPogleda, float2
uv)
{
    float slabljenje = svetloba.shadowAttenuation *
svetloba.distanceAttenuation;
    float3 barvaSvetlobe = svetloba.color * slabljenje;
```

Izsek kode 4.8: slabljenje svetlobe

Koda za difuzni odboj ostaja enaka, kot pri prejšnjem primeru, le da na koncu linearno interpoliramo od barve sence do barve objekta (Izsek kode 4.9).

```
float skalarniProdukt = saturate(dot(normala, svetloba.direction));
float3 neposrednaRazprsenost = skalarniProdukt * barvaSvetlobe;
float3 difuzniOdboj = lerp(_BarvaSence, _Barva, neposrednaRazprsenost);
```

Izsek kode 4.9: Difuzni odboj

Zrcalni odboj zahteva drugačno kalkulacijo. Izračunali bomo vektor, ki leži na polovici med smerjo svetlobe in smerjo pogleda. To storimo zato, ker v primerih, ko je kot med

smerjo svetlobe in smerjo kamere večji kot 90 stopinj, dobimo negativni skalarni produkt, kar lahko vodi do neželenih in nepredvidljivih rezultatov. Dobljen produkt večkrat pomnožimo, da je rezultat bolj izrazit. Parameteriziramo lahko tudi velikost območja zrcalnega odboja, kar v spodnjem izseku (Izsek kode 4.10) dosežemo s spremenljivko `_Sijaj` pri izračunu eksponenta.

```
float mocZrcalnegaOdboja = exp2((1 - _Sijaj) * 10.0 + 1.0);
float3 barvaZrcalnegaOdboja = float4(_MocZrcalnegaOdboja,
_MocZrcalnegaOdboja, _MocZrcalnegaOdboja, 1);

float3 polovicniVektor = normalize(svetloba.direction + smerPogleda);
float3 neposredenZrcalniOdboj = pow(max(0, dot(polovicniVektor, normala)),
mocZrcalnegaOdboja) * barvaZrcalnegaOdboja;
float3 zrcalniOdboj = neposredenZrcalniOdboj * barvaSvetlobe;
```

Izsek kode 4.10: Zrcalni odboj

Na ukrivljenih površinah je vpadni kot strmejši proti robovom objekta. To vodi do tega, da so zrcalni odsevi najbolj vidni ob robovih. Temu pravimo Fresnelov učinek (angl. Fresnel effect). Implementiramo ga tako, da opravimo skalarni produkt med normalo in smerjo pogleda ter ga odštejemo od 1, saj želimo imeti zrcalne odseve na robovih.

Če vrednost večkrat pomnožimo dobimo tanjši pas zrcalnega odboja. Na koncu še vrednost pomnožimo s poljubno barvo in še skalarjem za intenzivnost (Izsek kode 4.11).

```
float fresnel = 1.0 - saturate(dot(normala, smerPogleda));
float3 fresnelBarva = _FresnelBarva.rgb * pow(fresnel, _MocFresnela) *
_IntenzivnostFresnela;
```

Izsek kode 4.11: Fresnelov učinek

Bleščice vzorčimo z metodo tex2D, ki prejme teksturo in UV koordinate. Bleščice bomo nanesli v treh slojih, da bo učinek bolj naključen, z metodo deformacije definicijskega območja (Izsek kode 4.12).

```
float noise1 = tex2D (_NoiseTexture, uv * _NoiseSize + float2 (0, _Time.x *
_HitrostBlescic)).r;
float noise2 = tex2D (_NoiseTexture, uv * _NoiseSize * 1.4 + float2 (_Time.x
* _HitrostBlescic, 0)).r;
float blescice1 = pow (noise1 * noise2 * 2, _IntenzivnostBlescic);


noise1 = tex2D (_NoiseTexture, uv * _NoiseSize + float2 (0.3, _Time.x *
_HitrostBlescic)).r;
noise2 = tex2D (_NoiseTexture, uv * _NoiseSize * 1.4 + float2 (_Time.x *
_HitrostBlescic, 0.3)).r;
float blescice2 = pow (noise1 * noise2 * 2, _IntenzivnostBlescic);


noise1 = tex2D(_NoiseTexture, uv * _NoiseSize + float2 (0.6, _Time.x *
_HitrostBlescic) ).r;
noise2 = tex2D(_NoiseTexture, uv * _NoiseSize * 1.4 + float2 (_Time.x *
_HitrostBlescic, 0.6)).r;
float blescice3 = pow (noise1 * noise2 * 2, _IntenzivnostBlescic);
```

Izsek kode 4.12: Bleščice

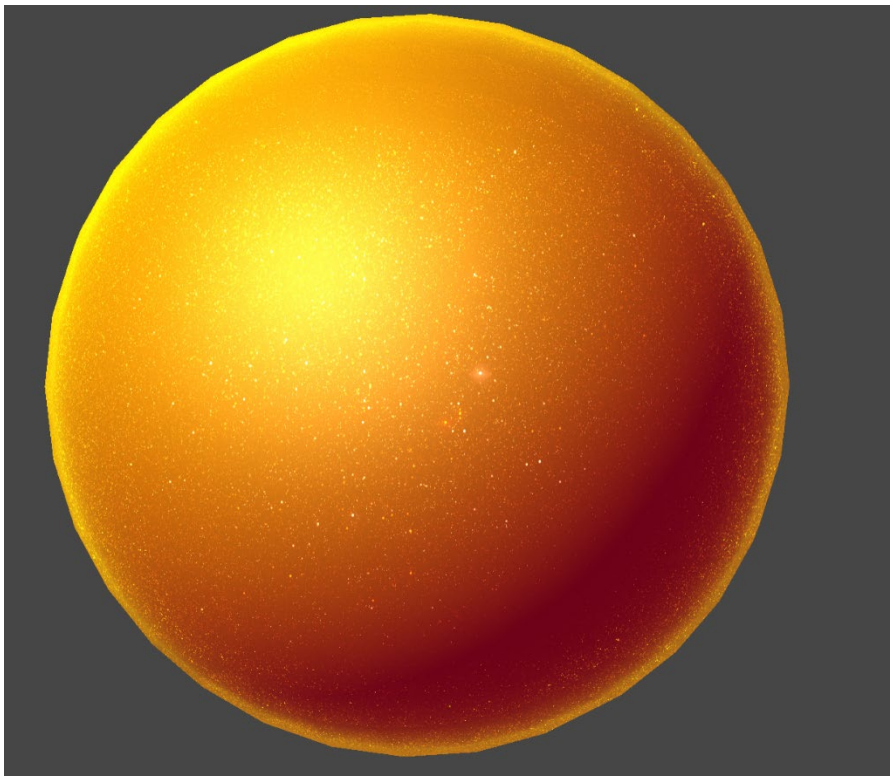
Za konec še izračunamo barve in intenzivnost na podlagi mesta bleščic in zapišemo podoben senčilnik fragmentov, kot pri prejšnjem primeru (Izseka kode 4.13 in 4.14). Z lastnostmi lahko nato ustvarimo različne izglede, eden od njih je prikazan spodaj (Slika 4.3).

```
float3 barveBlescic1 = blescice1 * (zrcalniOdboj *
_IntenzivnostBlescicZrcalnegaOdboja + neposrednaRazprsenost *
_IntenzivnostBlescicDifuznegaOdboja + fresnelBarva *
_IntenzivnostBlescicFresnel) * lerp(_BarvaBlescic, float3(1,1,1), 0.5);
float3 barveBlescic2 = blescice2 * (zrcalniOdboj *
_IntenzivnostBlescicZrcalnegaOdboja + neposrednaRazprsenost *
_IntenzivnostBlescicDifuznegaOdboja + fresnelBarva *
_IntenzivnostBlescicFresnel) * _BarvaBlescic;
float3 barveBlescic3 = blescice3 * (zrcalniOdboj *
_IntenzivnostBlescicZrcalnegaOdboja + neposrednaRazprsenost *
_IntenzivnostBlescicDifuznegaOdboja + fresnelBarva *
_IntenzivnostBlescicFresnel) * 0.5 * _BarvaBlescic;

float4 koncnaBarva = float4 (difuzniOdboj + zrcalniOdboj + barveBlescic1 +
barveBlescic2 + barveBlescic3 + fresnelBarva, 1);
return koncnaBarva;
```

Izsek kode 4.13: Izračun končne barve

```
float4 frag(Varyings IN) : SV_Target
{
    Light svetloba = GetMainLight();


    return half4(Osvetlitev(svetloba, IN.normalaWS, IN.smerPogleda, IN.uv),
1);
}
```

Izsek kode 4.14: Senčilnik oglišč



Slika 4.3: Rezultat bleščic

Podroben pogled v osvetljevanje objektov ni v sklopu te raziskovalne naloge, dobro pa je vedeti, da se Lambertov in Blinn-Phongov model posplošijo z BRDF (Bidirectional Reflection Distribution Funciton) modelom.

5 Premik od tradicionalnih metod

Sedaj senčilnike poznamo dovolj dobro, da lahko preusmerimo našo pozornost na jedro raziskovalne naloge.

5.1 Risana grafika

Danes je vedno več grafike risane, saj na tak način umetniki ustvarijo grafične stile, ki bolj izstopajo. Takšna grafika je v zadnjih letih bila deležna velike pozornosti, kot lahko danes vidimo pri nekaterih popularnih igrah. Običajno izgleda podobno kot grafika, ki jo zasledimo v risankah ali animiranih filmih. Zanjo je značilno, da prehod med barvami ni tako gladek kot pri realistični. Velikokrat vidimo tudi obrobe, kar vodi v večje izstopanje posameznih objektov.

Senčilnik za upodabljanje risane grafike bo naš prvi senčilnik iz dveh slojev. Najprej bomo upodobili objekt, nato pa še njegovo obrobo. Za senčilnike iz več slojev Unity potrebuje dodatno informacijo o tem, kateri sloj predstavlja glavno osvetljevanje objekta. To definiramo v dodatni oznaki znotraj sloja (Izsek kode 5.1).

```
Pass
{

    Tags
    {
        "LightMode" = "UniversalForward"
    }
```

Izsek kode 5.1: Dodatna oznaka v sloju

Strukture pri tem primeru so podobne kot pri prejšnjih primerih osvetljevanja (Izsek kode 5.2).

```
struct Attributes
{
    float4 pozicijaOS   : POSITION;
    float3 normalaOS    : NORMAL;
};

struct Varyings
{
    float4 pozicijaCS    : SV_POSITION;
    float3 smerPogleda : TEXCOORD0;
    float3 normalaWS     : NORMAL;
};
```

Izsek kode 5.2: Strukture pri risani grafiki

Tudi senčilnik oglišč je podoben (Izsek kode 5.3).

```
Varyings vert(Attributes IN)
{

    Varyings OUT;
    OUT.pozicijaCS = TransformObjectToHClip(IN.pozicijaOS.xyz);
    OUT.normalaWS = TransformObjectToWorldNormal(IN.normalaOS);
    float3 pozicijaWS = TransformObjectToWorld(IN.pozicijaOS.xyz);
    OUT.smerPogleda = normalize(_WorldSpaceCameraPos.xyz - pozicijaWS.xyz);
    return OUT;
}
```

Izsek kode 5.3: Senčilnik oglišč pri risani grafiki

Razlika je pravzaprav le v načinu osvetljevanja, ki ga uporabimo. V poglavju 3.3.1 smo spoznali funkcijo step, ki nam omogoči, da se znebimo gladkega barvnega prehoda. Tu bomo uporabili funkcijo `smoothstep(min, max, x)`, ki vrne vrednost 0, če je `x` manjši od min, 1 če je `x` večji od max in interpolirano vrednost med 0 in 1, če je `x` v

intervalu $[\min, \max]$. Ta funkcija izvrši interpolacijo z enakim polinomom, kot smo ga uporabili v poglavju 3.3.3.

Na ta način lahko izberemo, kako gladek prehod želimo imeti. Za ta primer bomo implementirali le difuzni odboj, na podoben način pa lahko z metodo `smoothstep` implementiramo tudi zrcalni odboj. Dodanih je še nekaj spremenljivk za dodatno uravnavanje intenzivnosti učinka (Izsek kode 5.4).

```
float4 _Barva;
float  _MejaBarve;
float  _GladkostPrehoda;
float _MocSvetlobe;
float3 _KonstantnaBarvaDifuznegaOdboja;
float _MocDifuznegaOdboja;

float3 Osvetlitev(Light svetloba, float3 normala)
{

    float skalarniProdukt = dot(normala,svetloba.direction);

    float slabljenje = svetloba.shadowAttenuation *
svetloba.distanceAttenuation;

    float3 neposrednaRazprsenost = smoothstep(_MejaBarve-
_GladkostPrehoda,_MejaBarve+_GladkostPrehoda, skalarniProdukt);

    neposrednaRazprsenost *= _MocSvetlobe;

    return _Barva * svetloba.color * neposrednaRazprsenost + _Barva *
(_KonstantnaBarvaDifuznegaOdboja * _MocDifuznegaOdboja);

}

float4 frag(Varyings IN) : SV_Target
{
    Light svetloba = GetMainLight();
    return half4(Osvetlitev(svetloba, IN.normalaWS), 1);
}
```

Izsek kode 5.4: Spremenljivke in senčilnik fragmentov pri risani grafiki

Kot drugi sloj bomo implementirali obrobe z najpopularnejšim in preprostim algoritmom IH (Inverted Hull). Deluje tako, da še enkrat upodobimo celoten objekt, pri čemer izbočimo vsa oglišča v smeri normal in opravimo zakrivanje trikotnikov, ki so obrnjeni proti nam. Na ta način upodobimo tako objekt iz prvega sloja, kot tudi njegovo obrobo. Izračun determinante in zakrivanje se izvrši samodejno ob uporabi stavka `Cull Front` pred oznako `HLSLPROGRAM` (Izsek kode 5.5).

```
Pass
{

    Cull Front

    HLSLPROGRAM
```

Izsek kode 5.5: Zakrivanje trikotnikov

Definiramo še preproste strukture in spremenljivko, s katero bomo določili debelino obrobe (Izsek kode 5.6).

```
struct Attributes
{
    float4 pozicijaOS   : POSITION;
    float3 normalaOS    : NORMAL;
};

struct Varyings
{
    float4 pozicijaCS    : SV_POSITION;
};

float _DebelinaObrobe;
```

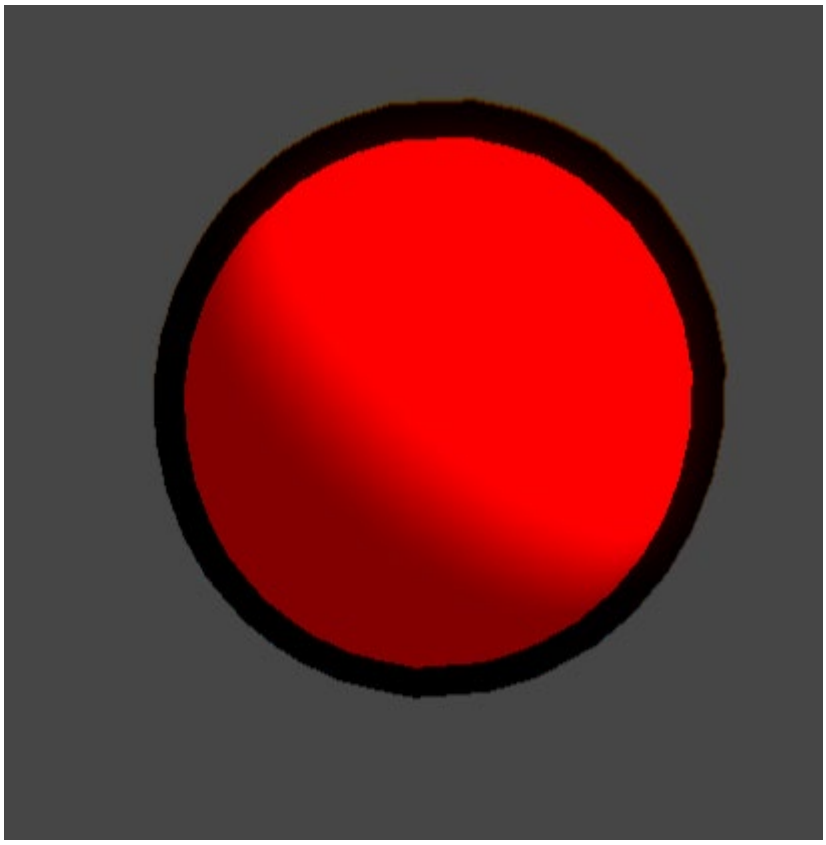
Izsek kode 5.6: Strukture pri obrobi

V senčilniku oglišč nato izbočimo oglišča v smeri normal ter zapišemo preprost senčilnik oglišč, v katerem lahko izberemo poljubno barvo (Izsek kode 5.7).

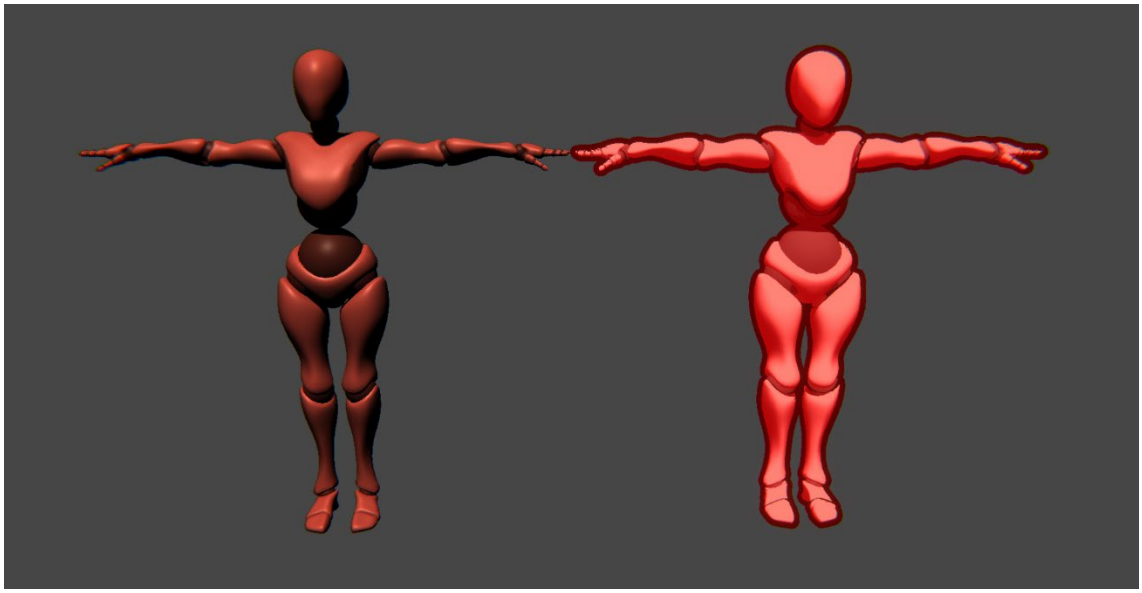
```
Varyings vert(Attributes IN)
{
    Varyings OUT;
    OUT.pozicijaCS = TransformObjectToHClip(IN.pozicijaOS.xyz +
_DebelinaObrobe * normalize(IN.normalaOS));
    return OUT;
}

float4 frag(Varyings IN) : SV_Target
{
    return half4(0,0,0,0);
}
```

Izsek kode 5.7: Implementacija obrobe

Rezultat učinka lahko vidimo na spodnjih slikah (Slika 5.1 in 5.2).



Slika 5.1: Risana grafika



Slika 5.2: Primerjava realistične in risane grafike

Za obrobe uporabljamo algoritem, ki deluje pri gladkih površinah (natančneje pri gladkih mnogoterostih). Naprednejša metoda se imenuje JFA (Jump Flooding Algorithm), ki jo opisuje spletna stran (Golus, 2020), za katero potrebujemo več znanja o alternativni uporabi tekstur.

5.2 Tekočina[6]

Hitro simuliranje tekočine je še danes zapleten problem v računalniški grafiki, zato večina implementacij uporablja superpozicijo sinusoid. Dandanes je popularna tehnika, s katero dosežemo tekočino v steklenici, ki se odziva na premike in rotacije. Poskusili bomo poustvariti ta učinek.

Tudi ta senčilnik bo sestavljen iz dveh slojev, v prvem bomo upodobili tekočino, v drugem pa steklenico.

Pri tem primeru bo potreben zapis C# kode, ki bo služila le kot vmesnik, ki bo posredoval spremembe pozicije in rotacije našemu senčilniku. Podatki o trenutnem materialu objekta se nahajajo v njegovi komponenti Render. To komponento pridobimo pri zagonu aplikacije z Unity dogodkom.

Tekom delovanja aplikacije spremljamo razliko med trenutno in prejšnjo pozicijo objekta, kar predstavlja njegovo hitrost ter razliko med trenutno in prejšnjo orientacijo, kar predstavlja njegovo kotno hitrost. Na podlagi teh dveh podatkov izračunamo količino nihanja v X in Z smeri. Nato bomo to količino manjšali v odvisnosti od časa ter jo uporabili skupaj s sinusno funkcijo, ki realizira učinek nihanja. Velikost same količine bomo tudi omejili na izbran interval, da zagotovimo pravilno delovanje v primeru velikih sunkov. Te informacije posredujemo senčilniku s pomočjo metode `material.SetFloat`, ki ji podamo ime lastnosti in vrednost, na katero jo želimo nastaviti (Izsek kode 5.8). Teh vrednosti ne želimo prikazovati v urejevalniku, zato jim v samem senčilniku dodamo atribut `[HideInInspector]` (Izsek kode 5.9).

[6] Implementirano po: (Art, 2018)

```
using UnityEngine;

public class Tekocina : MonoBehaviour
{
    Renderer renderer;
    Vector3 zadnjaPozicija;
    Vector3 zadnjaRotacija;
    public float MaxNihanje = 0.03f;
    public float hitrostNihanja = 1f;
    public float pojemanje = 1f;
    float dodanaKolicinaNihanjaX;
    float dodanaKolicinaNihanjaZ;
    float cas = 0.5f;
    // pridobitev komponente
    void Start()
    {
        renderer = GetComponent<Renderer>();
    }

    private void Update()
    {
        cas += Time.deltaTime;

        // zmanjšaj nihanje skozi čas
        dodanaKolicinaNihanjaX = Mathf.Lerp(dodanaKolicinaNihanjaX, 0,
Time.deltaTime * pojemanje);
        dodanaKolicinaNihanjaZ = Mathf.Lerp(dodanaKolicinaNihanjaZ, 0,
Time.deltaTime * pojemanje);

        // ustvari sinusno krivuljo nihanja
        float perioda = 2 * Mathf.PI * hitrostNihanja;
        float velikostNihanjaX = dodanaKolicinaNihanjaX * Mathf.Sin(perioda *
cas);
        float velikostNihanjaZ = dodanaKolicinaNihanjaZ * Mathf.Sin(perioda *
cas);
        // pošlji podatke senčilniku
        renderer.material.SetFloat("_NihanjeX", velikostNihanjaX);
        renderer.material.SetFloat("_NihanjeZ", velikostNihanjaZ);

        // izračunaj hitrost
        Vector3 hitrost = (zadnjaPozicija - transform.position) /
Time.deltaTime;
        Vector3 kotnaHitrost = transform.rotation.eulerAngles -
zadnjaRotacija;


        // dodaj hitrost nihanju
        dodanaKolicinaNihanjaX += Mathf.Clamp((hitrost.x + kotnaHitrost.z *
0.2f) * MaxNihanje, -MaxNihanje, MaxNihanje);
        dodanaKolicinaNihanjaZ += Mathf.Clamp((hitrost.z + kotnaHitrost.x *
0.2f) * MaxNihanje, -MaxNihanje, MaxNihanje);

        // shrani zadnjo pozicijo
        zadnjaPozicija = transform.position;
        zadnjaRotacija = transform.rotation.eulerAngles;
    }
}
```

Izsek kode 5.8: C# koda za implementacijo tekočine

```
[HideInInspector] _NihanjeX ("NihanjeX", Range(-1,1)) = 0.0
[HideInInspector] _NihanjeZ ("NihanjeZ", Range(-1,1)) = 0.0
```

Izsek kode 5.9: Lastnosti senčilnika, ki niso vidne v urejevalniku

V senčilniku bomo definirali še nekaj spremenljivk (Izsek kode 5.10).

```
float _NivoTekocine, _NihanjeX, _NihanjeZ;
float4 _PovrsinskaBarvaTekocine, _FresnelBarva, _BarvaPene, _BarvaTekocine;
float _DebelinaPene, _MocFresnela, _IntenzivnostFresnela;
```

Izsek kode 5.10: Spremenljivke za implementacijo tekočine

Tekočina mora imeti neodvisno orientacijo od objekta, zato bomo oglišča pretvorili v realni prostor. Vendar pa ne moremo uporabiti do sedaj vedno uporabljene funkcije `TransformObjectToWorld`, saj želimo, da se tekočina nahaja znotraj objekta. V prvem poglavju smo omenili, da prvi trije stolpci transformacije matrike $\mathbf{M} \in \mathbb{R}^{4\times4}$ iz objektnega v realni prostor predstavljajo smeri osi v realnem prostoru, četrti stolpec pa pozicijo v realnem prostoru. Zaradi tega vemo, da lahko željen učinek dosežemo, če sami pomnožimo transformacijsko matriko in pozicijo oglišč v objektnem prostoru tako, da četrti stolpec matrike pomnožimo z 0. Na ta način ignoriramo pozicijo v realnem prostoru, še vedno pa upoštevamo orientacijo glede na realni prostor. To je eden od primerov, ko nam dejansko poznavanje grafičnega cevovoda zelo pomaga.

Transformacijsko matriko pridobimo z metodo `GetObjectToWorldMatrix()`. Nihanje bomo izvedli v X in Z smeri, zato bomo pretvorjene koordinate rotirali okrog X osi na podlagi vrednosti iz C# kode (izsek kode 5.11). Za Z rotacijo le spremenimo vrstni red koordinat iz X rotacije (oziroma pomnožimo s permutacijsko matriko) (Izsek kode 5.12). Več podrobnostih o rotacijah v treh dimenzijah je opisanih v (Rotation formalisms in three dimensions).

```
float3 rotirajOkoliX(float3 oglisce, float kot)
{
    float2x2 matrika = float2x2(cos(kot), -sin(kot), sin(kot), cos(kot));
    return float3(oglisce.y, mul(matrika, oglisce.xz));
}
```

Izsek kode 5.11: Funkcija za rotacijo okoli x osi

```
Varyings vert(Attributes IN)
{

    Varyings OUT;

    OUT.pozicijaCS = TransformObjectToHClip (IN.pozicijaOS);
    OUT.normalaWS = TransformObjectToWorldNormal(IN.normalaOS);
    float3 pozicijaWS = mul(GetObjectToWorldMatrix(),
float4(IN.pozicijaOS.xyz,0)).xyz;

    float3 pozicijaWSX = rotirajOkoliX(pozicijaWS, PI);

    float3 pozicijaWSZ = float3(pozicijaWSX.y,pozicijaWSX.z,pozicijaWSX.x);



    float3 koncnaPozicijaWS = pozicijaWS + (pozicijaWSX * _NihanjeX) +
(pozicijaWSZ * _NihanjeZ);
    OUT.pozicijaWSY = koncnaPozicijaWS.y + _NivoTekocine;


    OUT.smerPogleda = normalize(_WorldSpaceCameraPos.xyz -
TransformObjectToWorld(IN.pozicijaOS.xyz));

    return OUT;

}
```

Izsek kode 5.12: Senčilnik oglišč pri tekočini

Senčilniku moramo dodati še dva ukaza, saj želimo, da je volumen nad nivojem površine izrezan. V poglavju 3.3.1 smo na koncu to storili s funkcijo `clip`, lahko pa uporabimo tudi `AlphaToMask On`, ki ga definiramo pred začetkom HLSL programa.

Poleg tega ukaza bomo še izklopili zakrivanje skritih primerkov z ukazom `Cull Off`, saj želimo upodobiti tudi trikotnike na notranji strani volumna za prikaz površine tekočine (Izsek kode 5.13).

```
Cull Off
AlphaToMask On
```

Izsek kode 5.13: Dodatna ukaza pri tekočini

Parameter s semantično oznako `VFACE` ima pozitivno vrednost za fragmente, ki pripadajo trikotnikom, ki so obrnjeni proti pogledu in negativno za tiste stran od njega. Na podlagi tega lahko ločimo površinski sloj tekočine (ki v resnici predstavlja le notranjo stran votlega volumna) in mu določimo svojo barvo. Z uporabo funkcije `step` implementiramo tudi vmesni sloj pene (Izsek kode 5.14).

```
float4 frag(Varyings IN, float orientacija : VFACE) : SV_Target
{
    Light svetloba = GetMainLight();


    float fresnel = 1.0 - dot(IN.normalaWS, IN.smerPogleda);
    float4 fresnelBarva = _FresnelBarva * pow(fresnel,_MocFresnela) *
_IntenzivnostFresnela;


    float4 pena = step(IN.pozicijaWSY, 0.5) - step(IN.pozicijaWSY, (0.5 -
_DebelinaPene));
    float4 barvaPene = pena * _BarvaPene;

    float4 preostalaTekocina = step(IN.pozicijaWSY, 0.5) - pena;
    float4 barvaTekocine = preostalaTekocina * _BarvaTekocine;

    float4 preostalaBarva = barvaTekocine + barvaPene;
    preostalaBarva.rgb += fresnelBarva;

    float4 povrsinskaBarva = _PovrsinskaBarvaTekocine * (pena +
preostalaTekocina);

    return orientacija > 0 ? preostalaBarva : povrsinskaBarva;

}
```

Izsek kode 5.14: Senčilnik fragmentov pri tekočini

V drugem sloju bomo upodobili steklenico. Ker bo delno prozorna, moramo zapisati oznako, ki bo označila, kako bomo novo barvo združili z barvo, ki se že nahaja na pripadajočem fragmentu (Izsek kode 5.15).

```
Blend SrcAlpha OneMinusSrcAlpha
```

Izsek kode 5.15: Združevanje barv

Ta oznaka predstavlja spodnjo kodo, ki se izvrši takoj po zapisanem senčilniku fragmentov (Izsek kode 5.16).

```
float4 koncnaBarva = float4(barvaDrugegaSloja.a) * barvaDrugegaSloja +
float4(1.0 - barvaDrugegaSloja.a) * barvaPrvegaSloja
```

Izsek kode 5.16: Pomen oznake `Blend`

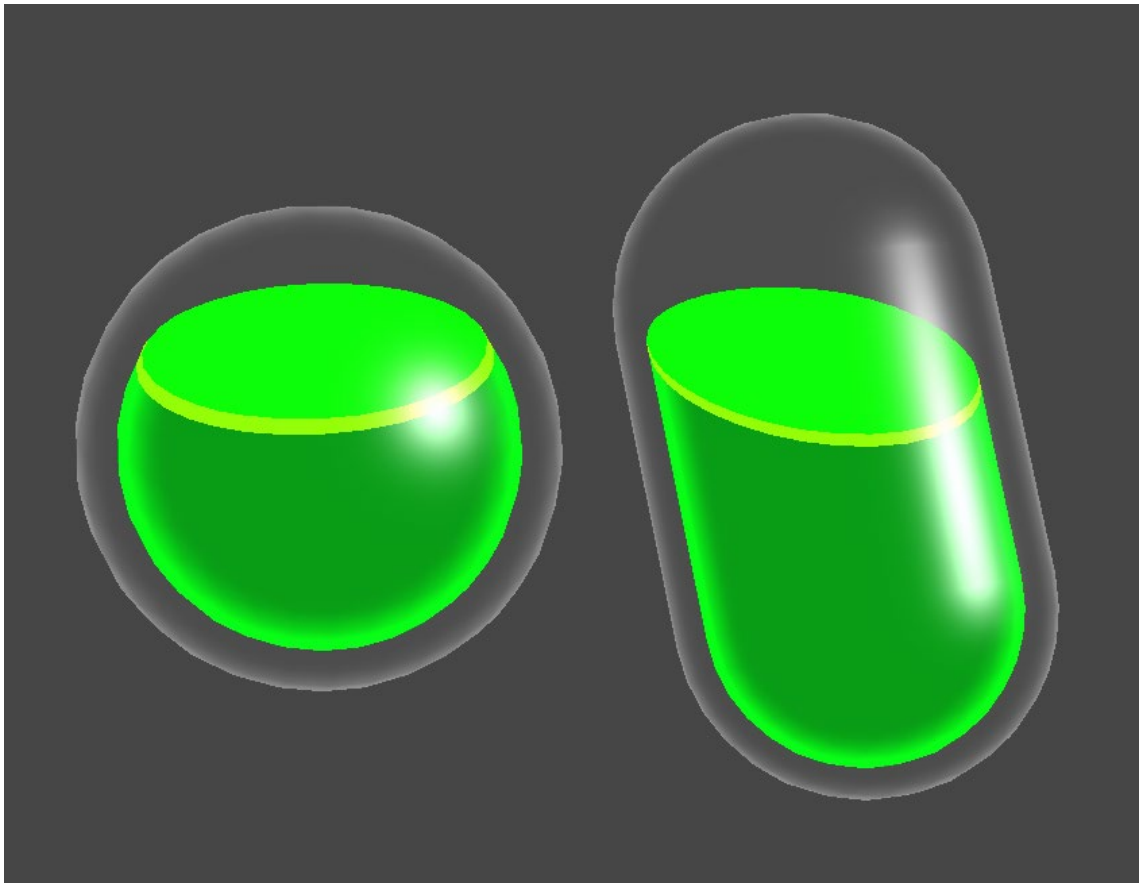
Na ta način dosežemo prozornost na podlagi komponente `a`, ki pripada barvi drugega sloja.

Preostala koda združi nekaj konceptov, ki smo jih videli v prejšnjih primerih. Steklenico izbočimo na podoben način, kot smo to storili pri risani grafiki, nato pa implementiramo še zrcalni odboj in Fresnelov učinek (Izsek kode 5.17).

```
struct Attributes
{
    float4 pozicijaOS : POSITION;
    float3 normalaOS : NORMAL;
};

struct Varyings
{
    float4 pozicijaCS : SV_POSITION;
    float3 normalaWS : TEXCOORD1;
    float3 smerPogleda : TEXCOORD2;
};

float4 _BarvaSteklenice, _BarvaFresnelaSteklenice;
float _DebelinaSteklenice, _MocFresnelaSteklenice,
_IntenzivnostFresnelaSteklenice;
float _MocZrcalnegaOdbojaSteklenice, _SijajSteklenice;

Varyings vert (Attributes IN)
{
    Varyings OUT;
    IN.pozicijaOS.xyz += _DebelinaSteklenice * IN.normalaOS;
    OUT.pozicijaCS = TransformObjectToHClip(IN.pozicijaOS);

    OUT.smerPogleda = normalize(_WorldSpaceCameraPos.xyz -
TransformObjectToWorld(IN.pozicijaOS.xyz));
    OUT.normalaWS = TransformObjectToWorldNormal (IN.normalaOS);


    return OUT;
}

float4 frag (Varyings IN) : SV_Target
{

    IN.normalaWS = normalize(IN.normalaWS);

    float mocZrcalnegaOdboja = exp2((1 - _SijajSteklenice) * 10.0 + 1.0);
    float4 barvaZrcalnegaOdboja = float4(_MocZrcalnegaOdbojaSteklenice,
_MocZrcalnegaOdbojaSteklenice, _MocZrcalnegaOdbojaSteklenice,
_MocZrcalnegaOdbojaSteklenice);


    Light svetloba = GetMainLight();

    float3 polovicniVektor = normalize (svetloba.direction + IN.smerPogleda);
    float4 zrcalniOdboj = pow (max (0,dot (polovicniVektor, IN.normalaWS)),
mocZrcalnegaOdboja) * barvaZrcalnegaOdboja;

    float fresnel = 1.0 - saturate(dot(IN.normalaWS, IN.smerPogleda));
    float4 fresnelBarva = _BarvaFresnelaSteklenice * pow (fresnel,
_MocFresnelaSteklenice) * _IntenzivnostFresnelaSteklenice;

    float4 koncnaBarva = fresnelBarva + _BarvaSteklenice + zrcalniOdboj;

    return koncnaBarva;
}
```

Izsek kode 5.17: Implementacija steklenice

Rezultat tekočine je razviden na spodnji sliki (Slika 5.3).

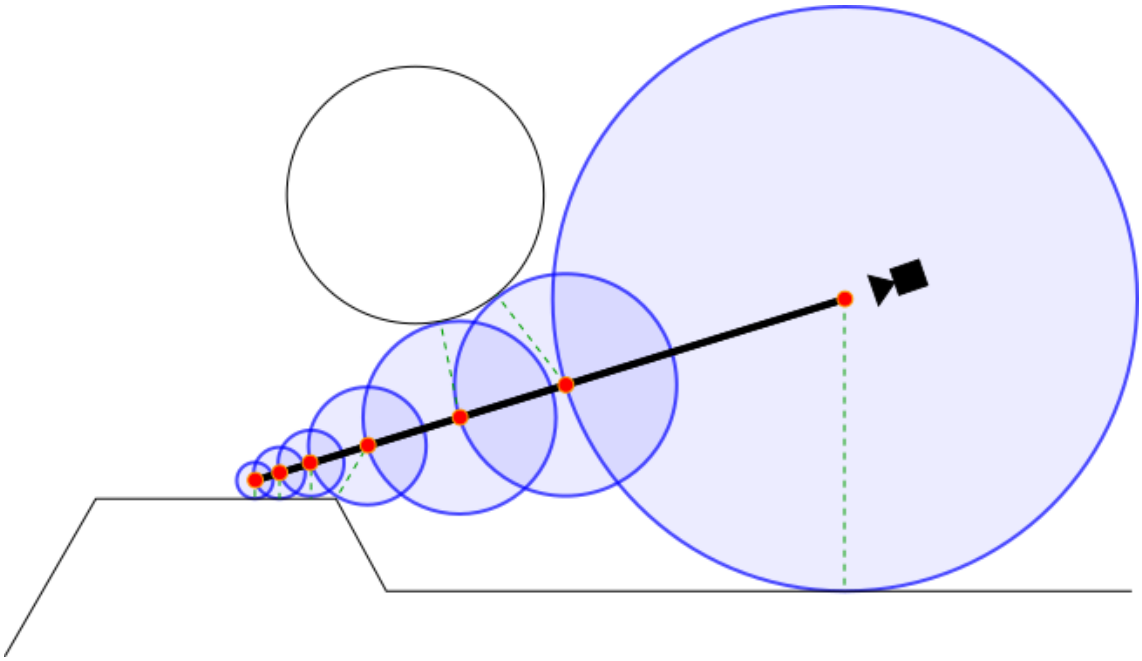


Slika 5.3: Rezultat tekočine

Trenutno najbolj natančne metode za upodabljanje tekočine, ki so realizirane s senčilniki za izračunavanje, temeljijo na metodah SPH (Smoothed Particle Hydrodynamics) in enačbah Navier-Strokes za tekočinsko dinamiko (Markus, Jens, Barbara, Andreas, & Matthias, 2014). Te metode so matematično prezahtevne za to raziskovalno nalogo.

5.3 Volumetrične metode

Objekte lahko upodobimo tudi tako, da uporabimo tridimenzionalno skalarno polje. Volumetrično metanje žarka (angl. volume ray casting ali raymarching) je alternativna metoda, ki temelji na izračunu oddaljenosti od najbližjih površin v svetu. Vsak žarek ima izhodišče v pogledu kamere in napreduje po največjih možnih korakih, dokler ne zadane površine v neki točki (Slika 5.4).



Slika 5.4: Volumetrično metanje žarka

Da lahko žarek napreduje po največjih možnih korakih, potrebujemo skalarno polje, ki se imenuje predznačeno polje razdalj (angl. signed distance field). To polje predstavlja funkcijo, ki kot argument prejme točko in vrne najkrajšo razdaljo od te točke do katerekoli površine v svetu. V primeru, da je točka znotraj objekta, vrne negativno število. Ta metoda je praktična za uporabo, saj lahko na podlagi polja razdalj izračunamo tudi trke med objekti, kar je pomembno predvsem v aplikacijah, ki tečejo v realnem času.

5.3.1 Volumetrično metanje žarka

To metodo bomo implementirali v Unity, da bomo bolje razumeli, kako deluje. Kasneje bomo uporabili odprtokodni dodatek za Unity, ki je veliko bolj izpopolnjen, saj implementiranje robustnega volumetričnega metanja žarka ni v sklopu te raziskovalne naloge.

Upodabljanje objekta s to metodo bo potekalo znotraj volumna, ki bo v našem primeru kocka.

Najprej ustvarimo nov senčilnik in mu v strukturo `Varyings` dodajmo dva tridimenzionalna vektorja, ki bosta predstavljala izhodišče žarka in točko na površini. Uporabili bomo tudi UV koordinate (Izsek kode 5.18).

```
struct Varyings
{
    float4 pozicijeCS  : SV_POSITION;
    float2 uv          : TEXCOORD0;
    float3 izhodisceZarka       : TEXCOORD1;
    float3 povrsinskaTocka      : TEXCOORD2;
};
```

Izsek kode 5.18: Struktura `Varyings` pri volumetričnem metanju žarkov

Izhodišče žarka bomo dobili tako, da pretvorimo pozicijo kamere v realnem svetu, ki je podana v globalni spremenljivki, v objektni prostor preko množenja z matriko (Izsek kode 5.19).

Točka na površini objekta pa je enaka poziciji oglišča v objektnem prostoru.

```
Varyings vert(Attributes IN)
{
    Varyings OUT;
    OUT.pozicijeCS = TransformObjectToHClip(IN.pozicijeOS.xyz);
    OUT.uv = IN.uv;
    OUT.izhodisceZarka = TransformWorldToObject(_WorldSpaceCameraPos);
    OUT.povrsinskaTocka = IN.pozicijeOS;
    return OUT;
}
```

Izsek kode 5.19: Senčilnik oglišč pri volumetričnem metanju žarkov

Izhodišča našega objekta želimo imeti v sredini našega volumna, zato bomo centrirali UV koordinate vsake ploskve. Smer našega žarka izračunamo tako, da odštejemo vektorja in pridobljeni vektor normaliziramo (Izsek kode 5.20).

```
float4 frag(Varyings IN) : SV_Target
{
    float2 uv = IN.uv - 0.5;
    float3 izhodisceZarka = IN.izhodisceZarka;
    float3 smerZarka = normalize(IN.povrsinskaTocka - izhodisceZarka);
}
```

Izsek kode 5.20: Senčilnik fragmentov pri volumetričnem metanju žarkov

Sedaj bomo napisali funkcijo za volumetrično metanje žarka, ki bo prejela izhodišče in smer ter vrnila razdaljo do površine, ki jo žarek zadane. Za delovanje te funkcije bomo definirali nekaj lastnosti (Izsek kode 5.21).

```
_NajvecjeSteviloKorakov ("Največje število korakov", float) = 100
_NajvecjaRazdalja ("Največja razdalja", float) = 100
_OddaljenostOdPovrsine ("Oddaljenost od površine", float) = 0.001
```

Izsek kode 5.21: Lastnosti pri volumetričnem metanju žarkov

Lastnosti bomo uporabili za prenehanje zanke v funkciji (Izsek kode 5.22).

```
float VolumetricnoMetanjeZarka(float3 izhodisceZarka, float3 smerZarka){
    float razdaljaOdIzhodisca = 0;
    float razdaljaDoPovrsine;
    for(int i = 0; i < _NajvecjeSteviloKorakov; i++){
        float3 pozicija = izhodisceZarka + razdaljaOdIzhodisca * smerZarka;
        razdaljaDoPovrsine = PoljeRazdalj(pozicija);
        razdaljaOdIzhodisca += razdaljaDoPovrsine;
        if(razdaljaDoPovrsine < _OddaljenostOdPovrsine || razdaljaOdIzhodisca
> _NajvecjaRazdalja) break;
    }
    return razdaljaOdIzhodisca;
}
```

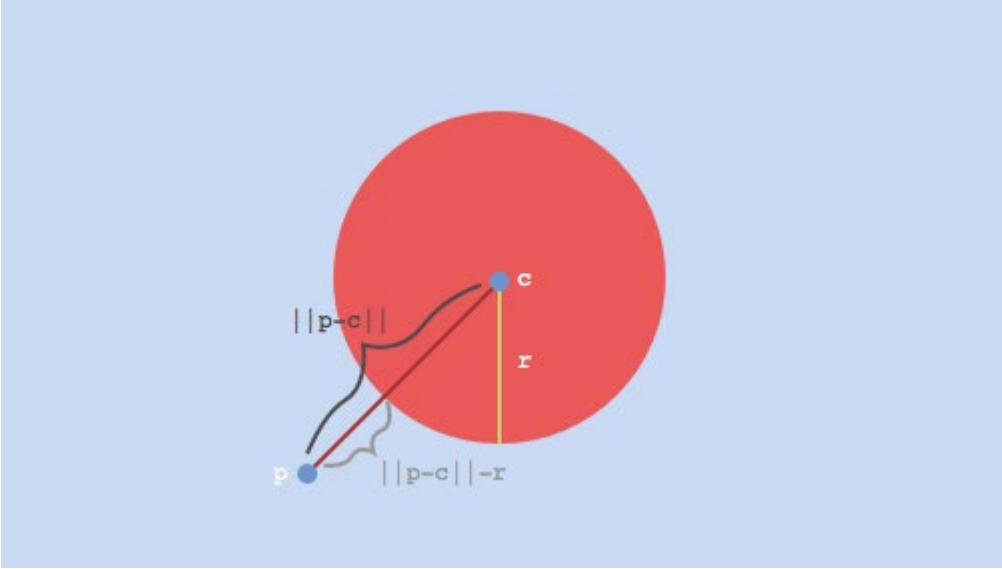
Izsek kode 5.22: Funkcija za volumetrično metanje žarkov

Napisati moramo še funkcijo, ki predstavlja naše polje razdalj. Za ta osnovni primer bomo upodobili kroglo v volumnu kocke (Izsek kode 5.23).

```
float PoljeRazdalj(float3 pozicija){
    float razdalja = length(pozicija) - 0.5;
    return razdalja;
}
```

Izsek kode 5.23: Funkcija za polje razdalj

To polje razdalj lahko izpeljemo iz spodnje slike (Slika 5.5).



Slika 5.5: Izpeljava polja razdalj

Funkcijo za volumetrično metanje žarka nato uporabimo v senčilniku fragmentov. Zapisali bomo še funkcijo za pridobitev normal, ki jih bi lahko uporabili pri implementaciji osvetljevanja (Izsek kode 5.24).

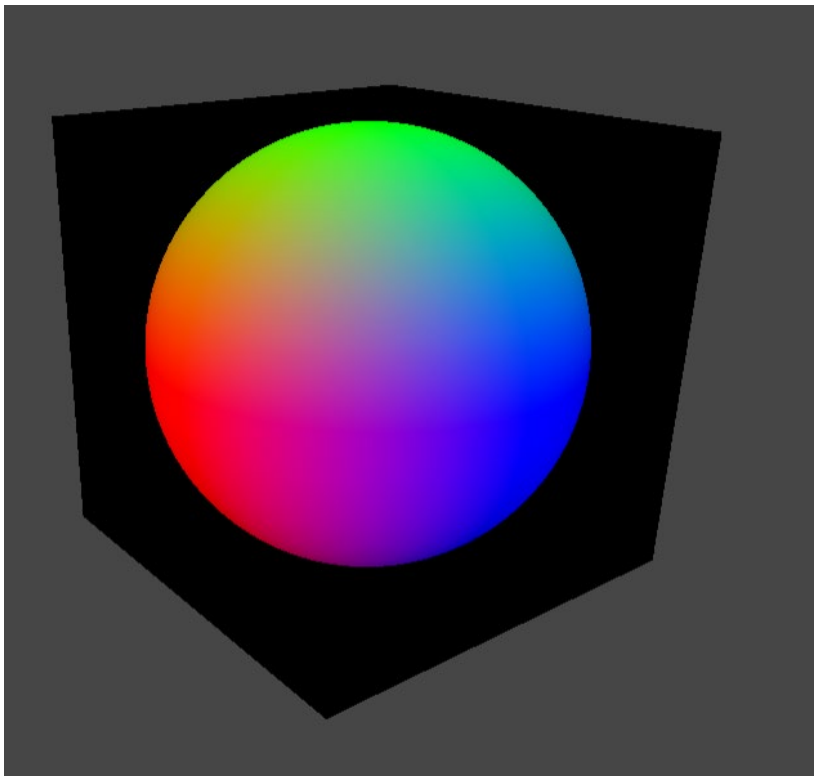
```
float3 PridobiNormale(float3 p){
    float2 e = float2(1e-2,0);
    float3 n = PoljeRazdalj(p) - float3(PoljeRazdalj(p-e.xyy),PoljeRazdalj(p-
e.yxy),PoljeRazdalj(p-e.yyx));
    return normalize(n);
}
```

Izsek kode 5.24: Funkcija za pridobitev normal

Ta funkcija temelji na izračunu odvodov. S senčilnikom fragmentov nato še s črno barvo obarvamo ozadje volumna (Izsek kode 5.25).

```
float4 frag(Varyings IN) : SV_Target
{
    float2 uv = IN.uv - 0.5;
    float3 izhodisceZarka = IN.izhodisceZarka;
    float3 smerZarka = normalize(IN.povrsinskaTocka - izhodisceZarka);

    float razdalja = VolumetricnoMetanjeZarka(izhodisceZarka,smerZarka);


    float4 barva = 0;

    if(razdalja < _NajvecjaRazdalja){
        float3 p = izhodisceZarka + smerZarka * razdalja;
        float3 n = PridobiNormale(p);
        barva.rgb = n;
    }

    return barva;
}
```

Izsek kode 5.25: Dokončan senčilnik fragmentov pri volumetričnem metanju žarkov



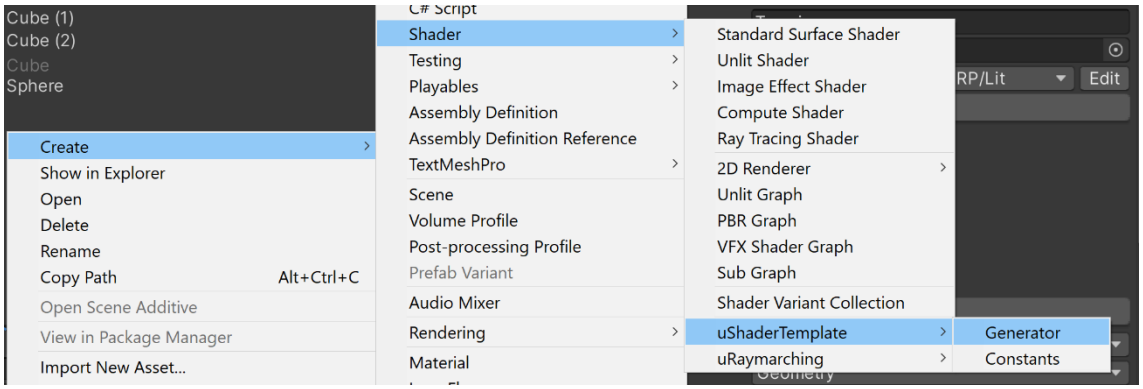
Slika 5.6: Rezultat volumetričnega metanja žarkov

5.3.2 Upodabljanje terena[7]

Od tu naprej bomo uporabili odprtokodno orodje za volumetrično metanje žarka uRaymarching, ki je dostopno na povezavi: (hecomi, 2020). Orodje je zelo robustno in nam omogoča, da se osredotočimo le na polje razdalj in površinske lastnosti objektov, ki jih želimo upodobiti. Trenutno je najnovejša verzija 2.1.1, vendar bi lahko uporabili tudi starejšo različico, saj se trenutne posodobitve nanašajo predvsem na VR tehnologijo.

Orodje uporabljamo tako, da najprej ustvarimo generator senčilnikov, ki bo generiral kodo za volumetrično metanje žarkov (Slika 5.7).

[7] Implementirano po: (Hiruma, [WebGL] レイマーチングでアンチエイリアス（FXAA）してみる, 2018) in (Hiruma, レイマーチングでHeight Map Distance Field, 2018)



Slika 5.7: Ustvarjanje generatorja senčinlikov

V generatorju lahko nastavimo željene lastnosti senčilnika, vendar jih bomo v tem primeru pustili pri miru. Ko želimo, da nam generator ustvari senčilnik, pritisnemo tipki Crtl+R ali pa pritisnemo na gumb `Export`. Omogoča nam tudi takojšno kreacijo primerka senčilnika (Slika 5.8).

Slika 5.8: Gumbi za ustvarjanje in posodabljanje senčilnika

Generator ima tri polja za vnos kode (Slika 5.9). V prvega lahko vnesemo lastnosti, kot bi to storili pri lastnih senčilnikih, v drugega vnesemo definicijo našega polja razdalj, ki se mora imenovati `DistanceFunction(float3 p)` in druge pomožne funkcije kot tudi spremenljivke, v tretjega pa lahko vnesemo dodatne lastnosti o površini objekta, kar nam omogoča funkcija `PostEffect(Raymarchinfo ray, PostEffectOutput o)`.

Slika 5.9: Polja za vnos kode

Za definicijo terena bomo izračunali oddaljenost od ravnine. To bomo storili na podlagi skalarnega produkta med normalo ravnine, ki jo lahko spreminjamo po želji in smerjo žarka. Z w komponento normale bomo določili višino terena.

Kot primer uporabe bomo vzeli volumetrično upodabljanje terena s pomočjo vzorčenja teksture. Pričnemo z definicijo potrebnih lastnosti in spremenljivk (Izsek kode 5.26).

```
_Velikost("Velikost", Range(0, 10)) = 7.64
_Tekstura("Tekstura",2D) = "" {}
_PozicijaRavnine ("Pozicija ravnine", Vector) = (0,0,1,2)
_Barva("Barva", Color) = (1, 1, 1, 0)
```

Izsek kode 5.26: Lastnosti pri upodabljanju terena

Ker je objekt, ki ga upodabljamo, dejansko še vedno kvader, bodo v primeru, ko se kamera nahaja znotraj kvadra, vsi trikotniki zakriti. To težavo odpravimo, če izklopimo zakrivanje trikotnikov z uporabo `CULL OFF`. V odprtokodnem orodju lahko to storimo kar v Unity urejevalniku, saj je implementirana lastnost (Izsek kode 5.27).

```
[Enum(UnityEngine.Rendering.CullMode)] _Cull("Culling", Int) = 2
```

Izsek kode 5.27: Uporaba zakrivanja

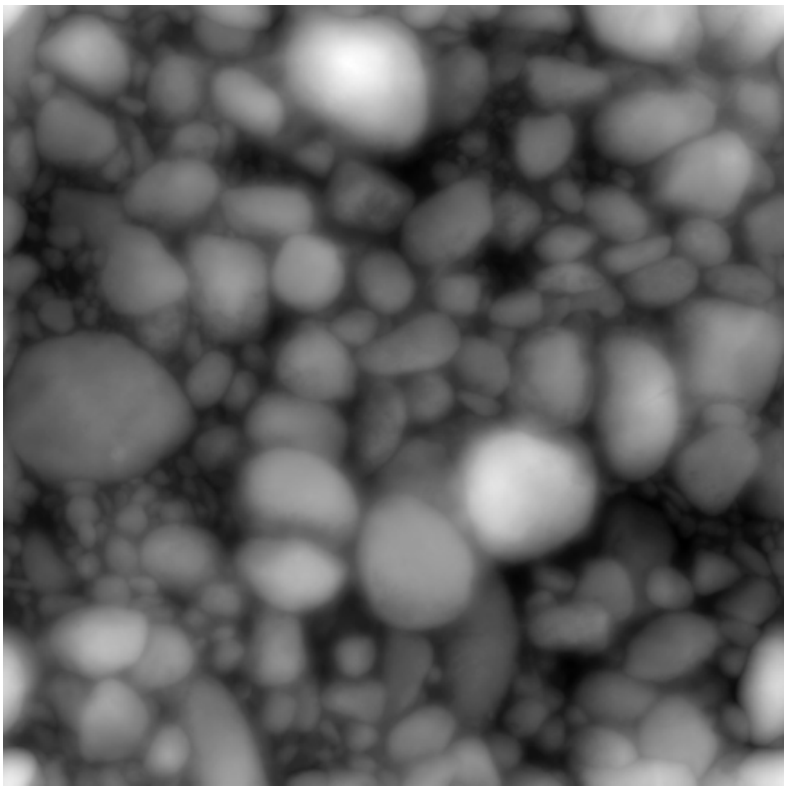
Za pravilno delovanje moramo še obkljukati preverjanje znotraj objekta zaradi same implementacije našega orodja (Slika 5.10).

Slika 5.10: Dodatno preverjanje

Teksturo moramo tokrat vzorčiti z metodo `tex2Dlod`, ker dejansko vzorčenje poteka v senčilniku oglišč. Na podlagi vzorčene teksture lahko definiramo polje razdalj (Izsek kode 5.28). Uporabljena tekstura je razvidna na sliki (Slika 5.11).

```
float _Velikost;
float4 _NormalaRavnine;
sampler2D _Tekstura;
float4 _Barva;

float razdaljaOdRavnine(float3 p, float4 n)
{
    return dot(p, n.xyz) * n.w;
}

float DistanceFunction(float3 pozicija)
{
    float4 ravnina = _NormalaRavnine;

    float razdalja = razdaljaOdRavnine(pozicija, ravnina);
    float4 uv = float4(pozicija.xz,0,0);
    float4 tekstura = tex2Dlod(_Tekstura,uv*0.2 - 1.0 *
floor(uv*0.2/1.0))*_Velikost;

    return razdalja - tekstura.x;
}
```

Izsek kode 5.28: Implementacija polja razdalj z vzorčenjem teksture



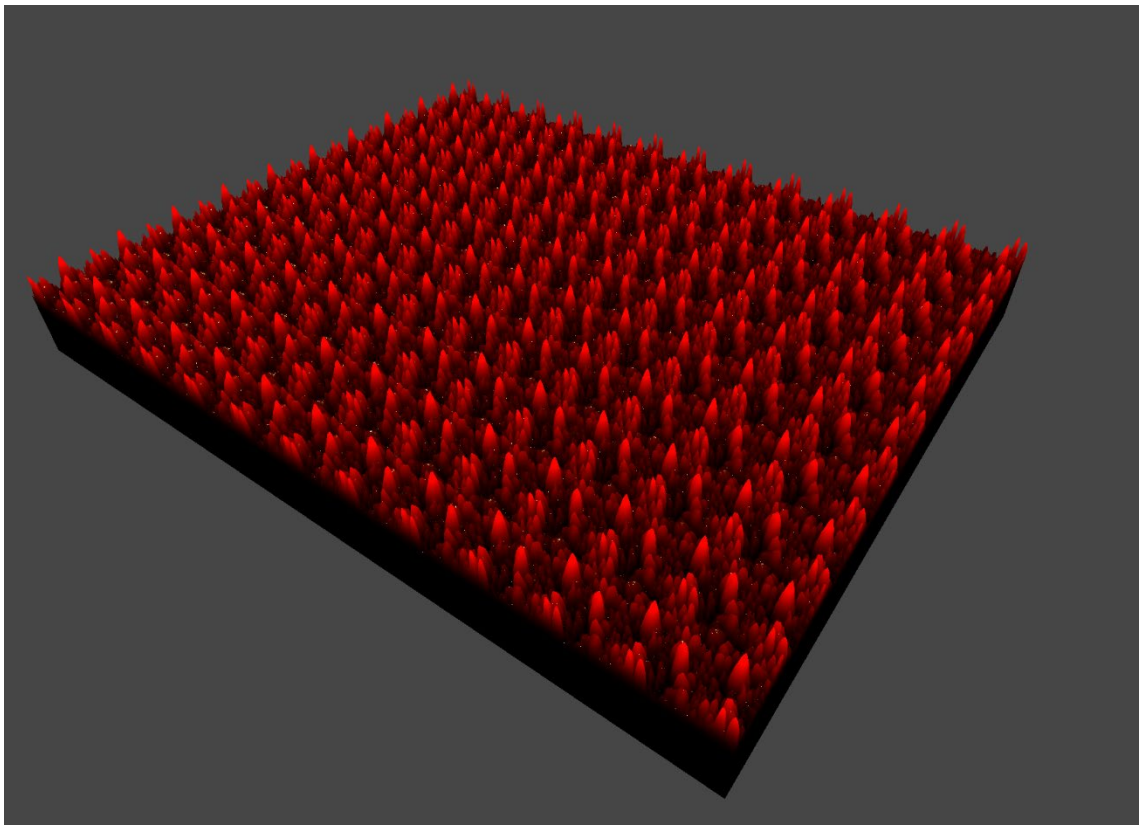
Slika 5.11: Uporabljena tekstura

Za konec še lahko določimo barvo na podlagi višine presečišča. Pri tem moramo uporabiti `RaymarchInfo` in `PostEffectOutput` strukturi, ki sta implementirani v odprtokodnem orodju (Izsek kode 5.29).

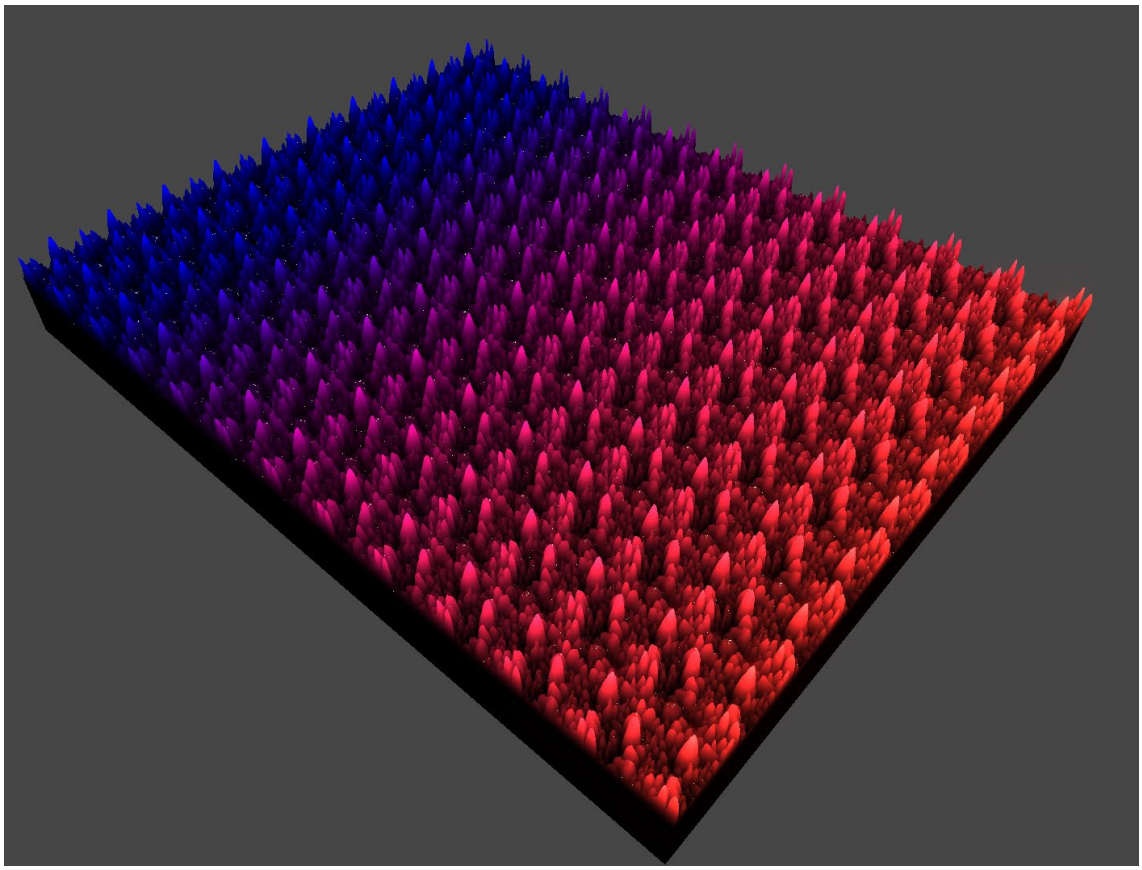
```
inline void PostEffect(RaymarchInfo ray, PostEffectOutput o)
{

    float3 pozicijaOS = ToLocal(ray.endPos);
    o.emission += smoothstep(0, abs(sin(_Time.y)) +1.7, pozicijaOS.y) *
_Barva;

}
```

Izsek kode 5.29: Določanje barve na podlagi višine presečišča

Rezultat lahko vidimo na spodnji sliki (Slika 5.12). Za zanimivejši učinek lahko tudi implementiramo interpolacijo med barvami (Slika 5.13).



Slika 5.12: Prvi prikaz terena



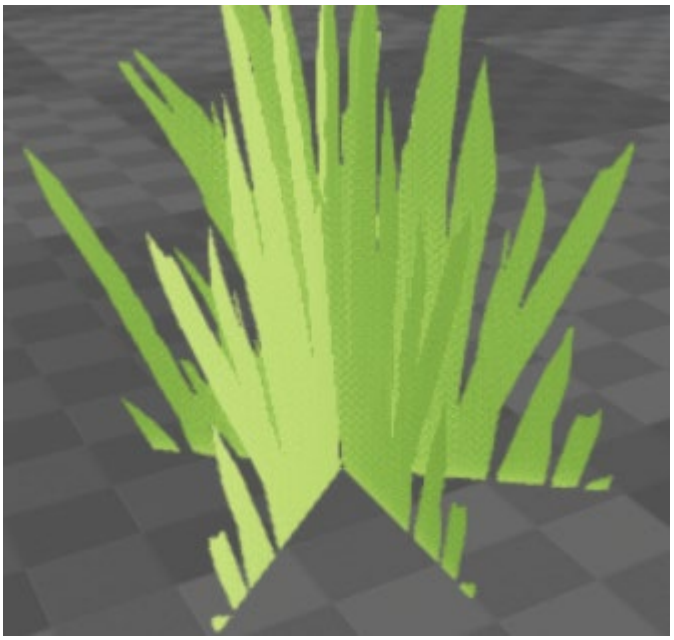
Slika 5.13: Drugi prikaz terena

5.4 Trava[8]

Pri raziskovanju delovanja grafičnega cevovoda sem spoznal, da je izločanje skritih primerkov računsko zahtevna operacija, če imamo zelo veliko primerkov. V računalniški grafiki poznamo senčilnik izračunavanja, katerega namen je pospešiti delovanje zahtevnih operacij.

Za primer upodabljanja velikega števila objektov bom uporabil travo. V veliko igrah je upodabljanje trave to narejeno s trikom, kjer se trava upodablja na dvodimenzionalnih pravokotniki (Slika 5.14). Vendar če gledamo pod kotom, se iluzija izgubi. Dodaten problem je, da je ta trava prikazana s teksturo, zato moramo upoštevati prosojnost, kar pa ni priporočljivo za mobilne ciljne naprave.

[8] Implementirano po: (陈嘉栋, 2017)



Slika 5.14: Običajna implementacija trave

Ta problem pa lahko odpravimo tako, da povečamo število primerkov in travo upodabljamo s preprostimi trikotniki, ki so ves čas usmerjeni proti kameri.

S tem projektom bomo praktično prikazali delovanje številnih delov grafičnega cevovoda, o katerih smo govorili v prvem poglavju ter uporabili večino metod v raziskovalni nalogi.

Pri upodabljanju nam bo pomagala metoda ustvarjanja primerkov z grafičnim procesorjem (v Unity GPU instancing, v pogonu iger Unreal pa mesh auto-instancing). To metodo uporabimo, ko želimo upodobiti veliko število objektov kar se da hitro. Običajno se zakrivanje trikotnikov zgodi avtomatsko, vendar pa moramo pri uporabi te metode zapisati svojo implementacijo. Ker bomo upodabljali zelo veliko količino trave, bomo to storili s senčilnikom izračunavanja.

Uporabiti želimo metodo `Graphics.DrawMeshInstancedIndirect()`, ki za parametre prejme mrežo objekta, indeks podmreže v primeru, da je mreža sestavljena iz več materialov (uporablja več senčilnikov), material, dimenzije območja za upodobitev in medpomnilnik, v katerem je zapisano število primerkov za izris. Ta funkcija se uporablja, kadar želimo večkrat upodobiti nek objekt z uporabo senčilnika.

Sledi nekaj C# kode, katere namen je pripraviti potrebne parametre za izris (izsek kode 5.30). Ker C# ni v ospredju raziskovalne naloge, bo koda bolj komentirana.

```
using System;
using System.Collections.Generic;
using UnityEngine;

public class IzrisovalecTrave : MonoBehaviour
{
    public Material MaterialPrimerkaTrave;

    private List<Vector3> pozicijeBilkTrave;

    private Mesh predpomnjenaMrezaTrave;

    private ComputeBuffer medpomnilnikZaArgumente;
}
```

Izsek kode 5.30: Definicija potrebnih spremenljivk

Material bomo določili naknadno v Unity, zato ga želimo serializirati, medtem ko bomo vse ostale podatke določili v kodi.

Najprej definirajmo mrežo. V našem primeru bomo izbrali trikotnik, ki velja za najbolj preprosto. Mrežo definiramo z uporabo `SetVertices()` in `SetTriangles()` metodama (Izsek kode 5.31).

```
private void NastaviMrezoTrave()
{
    if (predpomnjenaMrezaTrave) return;

    //če predpomnjena mreža ne obstaja, ustvarimo mrežo
    predpomnjenaMrezaTrave = new Mesh();

    //ena bilka trave (oglišča)
    Vector3[] oglisca = new Vector3[3];
    oglisca[0] = new Vector3(-0.25f, 0);
    oglisca[1] = new Vector3(+0.25f, 0);
    oglisca[2] = new Vector3(-0.0f, 1);


    //bilka trave (Indeksi trikotnika)
    int[] trikotniki = new int[3] { 2, 1, 0, }; //razvrstimo jih v pravilno
zaporedje za izločitev fragmentov za mrežo, ko bomo pisali senčilnik
(determinanta trikotnika)

    predpomnjenaMrezaTrave.SetVertices(oglisca);
    predpomnjenaMrezaTrave.SetTriangles(trikotniki, 0); // trava nima
podmreže, zato je drugi argument 0 (prvi indeks)
}
```

Izsek kode 5.31: Metoda za nastavitev mreže

Sedaj dodajmo metodo za določitev pozicij bilk trave. Najprej moramo vedeti, koliko bilk trave želimo izrisati (Izsek kode 5.32). Ustvarjanje primerkov z GPU je zelo zmogljiva metoda, zato bomo začeli tako, da jih upodobimo milijon. Posodabljanje samih pozicij pa je zelo počasno, zato bomo uporabili število, ki bo služilo kot predpomnilnik. Ker želimo

ohraniti konstantno gostoto trave, ne glede na število bilk, bomo velikost območja določili tako, da delimo število bilk z največjo razdaljo. Za natančnejše urejanje bomo to razmerje tudi korenili, nato bomo to območje naključno zapolnili s pozicijami bilk trave, ki jih na koncu pretvorimo v realni prostor (Izsek kode 5.33).

```
public int SteviloPrimerkov = 1000000;

public float NajvecjaRazdalja = 125f;

public float KoeficientGostote = 125f;

private int predpomnjenoStevilo = -1;
```

Izsek kode 5.32: Zapis dodatnih spremenljivk

```
private void PosodobiPozicije()
{
    if (SteviloPrimerkov == predpomnjenoStevilo) return;

    Debug.Log("Posodablja se pozicija trave!");

    //uporabimo enako seme, da je trava vedno na istem mestu
    UnityEngine.Random.InitState(111);

    //samodejno ohranjaj gostoto
    float velikost = Mathf.Sqrt(SteviloPrimerkov / KoeficientGostote) / 2f;
    transform.localScale = new Vector3(velikost, transform.localScale.y,
velikost); // določi velikost površine trave

    List<Vector3> pozicije = new List<Vector3>(SteviloPrimerkov);
    for (int i = 0; i < SteviloPrimerkov; i++)
    {
        Vector3 pozicija = Vector3.zero;

        pozicija.x = UnityEngine.Random.Range(-1f, 1f) *
transform.lossyScale.x;
        pozicija.z = UnityEngine.Random.Range(-1f, 1f) *
transform.lossyScale.z;

        //pretvorimo v realni prostor
        pozicija += transform.position;

        pozicije.Add(new Vector3(pozicija.x, pozicija.y, pozicija.z));
    }

    //določi vse pozicije
    predpomnjenoStevilo = pozicije.Count;
    pozicijeBilkTrave = pozicije;

}
```

Izsek kode 5.33: Metoda za posodobitev pozicij

Pozicije in mrežo želimo posodobiti takoj na začetku zagona aplikacije, za kar poskrbi Unity dogodek (Izsek kode 5.34).

```
private void Start()
{
    PosodobiPozicije();
    NastaviMrezoTrave();
}
```

Izsek kode 5.34: Dogodek, ki se zažene ob zagonu aplikacije

Za lažje urejanje pa omogočimo še posodabljanje med delovanjem (Izsek kode 3.35).

```
private void Update()
{
    PosodobiPozicije();
    NastaviMrezoTrave();
}
```

Izsek kode 5.35: Dogodek, ki se izvaja med delovanjem aplikacije

Sedaj nam preostane še določitev argumentov za naš medpomnilnik, ki potrebuje pet celih števil na v naprej določenih odmikih. Navezujejo se na mrežo, ki jo želimo upodobiti ter na njeno polje indeksov (Poglavje 2.3.1). Ti argumenti so: število indeksov v polju na primerek, število primerkov, mesto začetnega indeksa v polju, odmik dodan indeksom v polju in mesto začetnega primerka. Za konec izračunamo še ekstremne vrednosti za določitev območja trave (Izseka kode 5.36 in 5.37).

```
private float minX, minZ, maxX, maxZ;
```

Izsek kode 5.36: Določitev spremenljivk za območje trave

```
void PosodobiMedpomnilnik()
{
    //če ni potrebe po posodobitvi potem prej končamo
    if (predpomnjenoStevilo == pozicijeBilkTrave.Count &&
        medpomnilnikZaArgumente != null)
    {
        return;
    }
    minX = float.MaxValue;
    minZ = float.MaxValue;
    maxX = float.MinValue;
    maxZ = float.MinValue;

    // določimo meje na podlagi ekstremnih vrednosti
    for (int i = 0; i < pozicijeBilkTrave.Count; i++)
    {
        Vector3 target = pozicijeBilkTrave[i];
        minX = Mathf.Min(target.x, minX);
        minZ = Mathf.Min(target.z, minZ);
        maxX = Mathf.Max(target.x, maxX);
        maxZ = Mathf.Max(target.z, maxZ);
    }

    medpomnilnikZaArgumente?.Release(); // sprostimo trenutni medpomnilnik v
primeru da ga že imamo
    uint[] args = new uint[5] { 0, 0, 0, 0, 0 }; // podatkovni tip so
nenegativna cela števila
    medpomnilnikZaArgumente = new ComputeBuffer(1, args.Length *
sizeof(uint), ComputeBufferType.IndirectArguments);
    // drugi argument je 5 * velikost podatkovnega tipa in določa odmike
začetkov posameznih parametrov trave

    args[0] = predpomnjenaMrezaTrave.GetIndexCount(0);
    args[1] = (uint)pozicijeBilkTrave.Count;
    args[2] = predpomnjenaMrezaTrave.GetIndexStart(0);
    args[3] = predpomnjenaMrezaTrave.GetBaseVertex(0); // prvo oglišče, ki se
nahaja na odmiku
    args[4] = 0; // prvi primerek je na začetku

    medpomnilnikZaArgumente.SetData(args);
    // posodobi predpomnilnik
    predpomnjenoStevilo = pozicijeBilkTrave.Count;
}
```

Izsek kode 5.37: Metoda za posodobitev medpomnilnika

Tako smo pripravili vse parametre potrebne za klic željene metode. Sedaj lahko upodobimo travo z metodo `DrawMeshInstancedIndirect` (Izsek kode 5.38).

```
void LateUpdate()
{
    // posodobimo medpomnilnik
    PosodobiMedpomnilnik();

    // izrišemo vse z enim klicom DrawMeshInstancedIndirect()
    Bounds ombocje = new Bounds();
    ombocje.SetMinMax(new Vector3(minX, 0, minZ), new Vector3(maxX, 0,
maxZ));//če stožec kamere ne seka to območje, DrawMeshInstancedIndirect() ne
bo upodobil trave
    Graphics.DrawMeshInstancedIndirect(predpomnjenaMrezaTrave, 0,
MaterialPrimerkaTrave, ombocje, medpomnilnikZaArgumente);
}
```

Izsek kode 5.38: Uporaba metode `DrawMeshInstancedIndirect`

Sedaj bomo implementirali še izločanje skritih primerkov. Uporabili bomo izločanje z metodo AABB (Axis-Aligned Bounding Box) ter uporabo celic (Ulf & Tomas, 1999). V računalniški grafiki pogosto izločimo veliko primerkov tako, da jih najprej porazdelimo po celicah, ki so določene velikosti. Nato najprej opravimo izločanje primerkov na nivoju celic. Na ta način lahko izločimo veliko število primerkov.

Ker upodabljamo veliko trave, bomo potrebovali senčilnik izračunavanja, ki bo razločeval med vidnimi in skritimi primerki trave. Posredovali mu bomo dva medpomnilnika. V prvem se bodo nahajale vse pozicije trave, v drugega pa bo senčilnik dodal le primerke, ki so vidni.

Za implementacijo izločanja bomo potrebovali naslednje spremenljivke (Izsek kode 5.39).

```
public ComputeShader sencilnikZaIzlocanje;

private int steviloCelicX = -1;
private int steviloCelicZ = -1;

private float velikostCeliceX = 10;
private float velikostCeliceZ = 10;

private ComputeBuffer medpomnilnikVsehPozicijTrave;
private ComputeBuffer medpomnilnikIndeksovVsehVidnihPrimerov; //potrebujemo
le indekse

private List<Vector3>[] seznamPozicijCelic;
private List<int> IDVidnihCelic = new List<int>();
private Plane[] rezalneRavnineKamere = new Plane[6];
```

Izsek kode 5.39: Definicija preostalih potrebnih spremenljivk

V metodi za posodabljanje medpomnilnikov bomo pripravili nova medpomnilnika (Izsek kode 5.40).

```
// sprostimo oba medpomnilnika
medpomnilnikVsehPozicijTrave?.Release();
medpomnilnikVsehPozicijTrave = new ComputeBuffer(pozicijeBilkTrave.Count,
sizeof(float)*3); //pozicije so tipa float3

medpomnilnikIndeksovVsehVidnihPrimerov?.Release();
medpomnilnikIndeksovVsehVidnihPrimerov = new
ComputeBuffer(pozicijeBilkTrave.Count, sizeof(uint),
ComputeBufferType.Append); //indeksi so pozitivna cela števila
```

Izsek kode 5.40: priprava medpomnilnikov

Nato razporedimo pozicije po celicah. Senčilniku za izločanje moramo posredovati sploščeno polje poziciji v celicah, zato seznam pozicij pretvorimo v polje (Izsek kode 5.41).

```
//izračunamo število celic na podlagi mej in velikosti celice
steviloCelicX = Mathf.CeilToInt((maxX - minX) / velikostCeliceX);
steviloCelicZ = Mathf.CeilToInt((maxZ - minZ) / velikostCeliceZ);

//inicializiramo seznam pozicij celic
seznamPozicijVCelicah = new List<Vector3>[steviloCelicX * steviloCelicZ];
for (int i = 0; i < seznamPozicijVCelicah.Length; i++)
{
    seznamPozicijVCelicah[i] = new List<Vector3>();
}

//vsaki poziciji določimo pripadajočo celico
for (int i = 0; i < pozicijeBilkTrave.Count; i++)
{
    Vector3 pozicija = pozicijeBilkTrave[i];

    //najdemo indeks celice
    int xID = Mathf.Min(steviloCelicX-
1,Mathf.FloorToInt(Mathf.InverseLerp(minX, maxX, pozicija.x) *
steviloCelicX));
    int zID = Mathf.Min(steviloCelicZ-
1,Mathf.FloorToInt(Mathf.InverseLerp(minZ, maxZ, pozicija.z) *
steviloCelicZ));

    seznamPozicijVCelicah[xID + zID * steviloCelicX].Add(pozicija);
}

//inicializiramo sploščeno 1D polje za medpomnilnik
int odmik = 0;
Vector3[] pozicijeBilkTraveVCelicah = new Vector3[pozicijeBilkTrave.Count];
for (int i = 0; i < seznamPozicijVCelicah.Length; i++)
{
    for (int j = 0; j < seznamPozicijVCelicah[i].Count; j++)
    {
        pozicijeBilkTraveVCelicah[odmik] = seznamPozicijVCelicah[i][j];
        odmik++;
    }
}

medpomnilnikVsehPozicijTrave.SetData(pozicijeBilkTraveVCelicah);

// posredujemo polja senčilniku
MaterialPrimerkaTrave.SetBuffer("_MedpomnilnikVsehPozicijTrave",
medpomnilnikVsehPozicijTrave);
MaterialPrimerkaTrave.SetBuffer("_MedpomnilnikIndeksovVsehVidnihPrimerov",
medpomnilnikIndeksovVsehVidnihPrimerov); // trenutno prazno
```

Izsek kode 5.41: Razporeditev pozicij v celice

Nazadnje še opravimo izločanje po celicah. Ponovno bomo napisali dogodek, v katerem se upodablja trava. Najprej pridobimo kamero in ji določimo oddaljeno rezalno ravnino na podlagi največje dovoljene razdalje. Nato izračunamo območja celic na podlagi linearne interpolacije med največjo in najmanjšo pozicijo trave. Metoda `TestPlanesAABB` vrne vrednost `TRUE` v primeru, ko je območje znotraj vidnega polja ali ko seka njegove rezalne ravnine. Na podlagi tega lahko določimo celice, nad katerimi

bomo izvedli izločanje na nivoju primerkov. Senčilniku za izločanje moramo posredovati informacije o transformacijski matriki, s katero bo preverjal, če se pozicija nahaja v rezalnem prostoru.

Senčilnik izvršimo za vsako vidno celico, razen, če imata celici, ki sta vidni, sosednji indeks. To pomeni, da imajo tudi pozicije trave v obeh celicah zaporedne indekse. Na ta način dosežemo manj klicev. Delo za vsak klic senčilnika porazdelimo med 64 skupin niti. Po končanem izločanju posredujemo število primerkov, ki je vidnih, metodi za upodobitev. Dejanske pozicije vidnih primerkov pa bomo uporabili v senčilniku trave (Izsek kode 5.42).

```
void LateUpdate()
{
    // posodobimo medpomnilnik
    PosodobiMedpomnilnik();

    IDVidnihCelic.Clear();//izpraznimo seznam celic za izločanje
    Camera kamera = Camera.main;


    float prvotnaOddaljenaRavnina = kamera.farClipPlane;
    kamera.farClipPlane = NajvecjaRazdalja;//upoštevamo največjo razdaljo
    GeometryUtility.CalculateFrustumPlanes(kamera, rezalneRavnineKamere);//
izračunamo rezalne ravnine, ki upoštevajo razdaljo
    kamera.farClipPlane = prvotnaOddaljenaRavnina;//povrnitev prvotne rezalne
ravnine

    //Izločanje skritih primerkov po celicah
    for (int i = 0; i < seznamPozicijVCelicah.Length; i++)
    {
        //ustvarimo območje celice
        Vector3 sredisceCeliceWS = new Vector3 (i % steviloCelicX + 0.5f, 0,
i / steviloCelicX + 0.5f);
        sredisceCeliceWS.x = Mathf.Lerp(minX, maxX, sredisceCeliceWS.x /
steviloCelicX);
        sredisceCeliceWS.z = Mathf.Lerp(minZ, maxZ, sredisceCeliceWS.z /
steviloCelicZ);
        Vector3 velikostWS = new Vector3(Mathf.Abs(maxX - minX) /
steviloCelicX,0,Mathf.Abs(maxX - minX) / steviloCelicX);
        Bounds obmocjeCelice = new Bounds(sredisceCeliceWS, velikostWS);


        // preverimo če je celica v vidnem polju
        if (GeometryUtility.TestPlanesAABB(rezalneRavnineKamere,
obmocjeCelice))
        {
            IDVidnihCelic.Add(i);
        }
    }


    Matrix4x4 v = kamera.worldToCameraMatrix; // matrika iz realnega prostora
v prostor pogleda
    Matrix4x4 p = kamera.projectionMatrix; // matrika iz prostora pogleda v
rezalni prostor
    Matrix4x4 vp = p * v; // matrika iz realnega prostora v rezalni prostor

    // ponovno inicializiramo medpomnilnik za štetje vidnih primerkov
    medpomnilnikIndeksovVsehVidnihPrimerov.SetCounterValue(0);

    // posredujemo matrike senčilniku za izločanje
    sencilnikZaIzlocanje.SetMatrix("_WStoCSMatrika", vp);
    sencilnikZaIzlocanje.SetFloat("_NajvecjaRazdalja", NajvecjaRazdalja);

    //izrvšimo senčilnik za izločanje za vsako vidno celico
    for (int i = 0; i < IDVidnihCelic.Count; i++)
    {
        int IDVidneCelice = IDVidnihCelic[i];
        int odmik = 0;
        for (int j = 0; j < IDVidneCelice; j++)
```

```
        {
            odmik += seznamPozicijVCelicah[j].Count;
        }
        //izločanje se začne pri odmiku
        sencilnikZaIzlocanje.SetInt("_Odmik", odmik);
        int dolzinaSeznama = seznamPozicijVCelicah[IDVidneCelice].Count;


        //združimo n klicev v 1 klic pod pogojem, da sta indeksa v polju
sodednja
        while (i < IDVidnihCelic.Count - 1 &&
               IDVidnihCelic[i + 1] == IDVidnihCelic[i] + 1)
        {
            //če sta sosednja jih združimo v en klic
            dolzinaSeznama += seznamPozicijVCelicah[IDVidnihCelic[i +
1]].Count;
            i++;
        }

        // izvršimo senčilnik za izločanje
        sencilnikZaIzlocanje.Dispatch(0, Mathf.CeilToInt(dolzinaSeznama /
64f), 1, 1); //porazdelimo delo med 64 skupin niti

    }

    // kopiramo število vidnih primerkov v medpomnilnik za argumente
    // na ta način določimo količino, ki jo bomo upodobili z
DrawMeshInstancedIndirect
    ComputeBuffer.CopyCount(medpomnilnikIndeksovVsehVidnihPrimerov,
medpomnilnikZaArgumente, 4);

    // upodobimo vse primerke z enim klicom z metode
DrawMeshInstancedIndirect()
    Bounds obmocje = new Bounds();
    obmocje.SetMinMax(new Vector3(minX, 0, minZ), new Vector3(maxX, 0,
maxZ));//če stožec pogleda ne seka to območje, DrawMeshInstancedIndirect() ne
bo upodobil trave
    Graphics.DrawMeshInstancedIndirect(predpomnjenaMrezaTrave, 0,
MaterialPrimerkaTrave, obmocje, medpomnilnikZaArgumente);
}
```

Izsek kode 5.42: Uporaba senčilnika za izločanje

Senčilniki za izračunavanje so dovolj obširno področje, da bi lahko imel raziskovalno nalogo le o njih. Za namen te naloge bo dovolj, da le prikažemo njihovo uporabnost brez, da se poglabljamo v podrobnosti paralelnih algoritmov in arhitektur grafičnih procesorjev.

Senčilniki so uporabljeni za številne zahtevne operacije, kot so še tudi fizikalne simulacije in zaznavanje trkov.

Podobno kot z običajnimi senčilniki tudi tu s predprocesorkim ukazom določimo funkcijo, ki predstavlja senčilnik. Pred funkcijo se nahaja atribut, ki opisuje število uporabljenih niti v posamezni skupini. Običajno se uporabi 64 niti, ki so porazdeljene po

načinu $64 \times 1 \times 1$ ali pa $8 \times 8 \times 1$. To število ima vpliv na kompatibilnost v različnih senčilnih modelih na grafičnih procesorjih. Kot parameter senčilnik prejme ID niti, ki izvršuje ukaze. Ker smo delo porazdelili med 64 niti, lahko to število uporabimo za pridobitev pozicij trave, ki jih nato pretvorimo v rezalni prostor. Izločanje bi sicer lahko izvršili tudi v prostoru normaliziranih koordinat naprave, vendar na ta način ne potrebujemo izvršiti operacije deljenja. V primeru, da se kamera premika, moramo za izločevanje določiti nek prag odstopanja, saj bi sicer lahko zasledili trenutke, ko se trava upodablja. Ta prag je pomemben predvsem v y in x smeri. V našem primeru dopuščamo 50 % odstopanje v obeh smereh. Nato uporabimo sistem neenačb, ki smo jih spoznali v prvem poglavju. Za lažje preverjanje bomo izračunali absolutno vrednost pozicije. Vidne primerke nato dodamo v posebno implementacijo medpomnilnika, ki se imenuje `AppendStructuredBuffer`, in omogoča dodajanje spremenljive količine primerkov (Izsek kode 5.43).

```
#pragma kernel CSMain

float4x4 _WStoCSMatrika;
float _NajvecjaRazdalja;
uint _Odmik;
StructuredBuffer<float3> _MedpomnilnikVsehPozicijTrave;
AppendStructuredBuffer<uint> _MedpomnilnikIndeksovVsehVidnihPrimerov;


[numthreads(64,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{

   // pretvorba iz realnega prostora v rezalni prostor
   float4 pozicijaCS =
abs(mul(_WStoCSMatrika,float4(_MedpomnilnikVsehPozicijTrave[id.x +
_Odmik],1.0)));

    if (pozicijaCS.z <= pozicijaCS.w && pozicijaCS.y <= pozicijaCS.w*1.5 &&
pozicijaCS.x <= pozicijaCS.w*1.5 && pozicijaCS.w <= _NajvecjaRazdalja)
      _MedpomnilnikIndeksovVsehVidnihPrimerov.Append(id.x + _Odmik);
}
```

Izsek kode 5.43: Zapis senčilnika za izločanje

Sedaj potrebujemo še material oziroma senčilnik. Zapišemo do zdaj že znano kodo (Izsek kode 5.44).

```
struct Attributes
{
     float4 pozicijaOS   : POSITION;
     half4 barva         : COLOR;
};

struct Varyings
{
    float4 positionCS  : SV_POSITION;
    half3 barva        : COLOR;
};


float _SirinaTrave;
float _VisinaTrave;
float _MocVetraEna;
float _FrekvencaVetraEna;
float2 _PonavljanjeVetraEna;
float2 _ZavijanjeVetraEna;
float _MocVetraDva;
float _FrekvencaVetraDva;
float2 _PonavljanjeVetraDva;
float2 _ZavijanjeVetraDva;
half3 _GlavnaBarva;
float4 _BarvaTeksture_ST;
half3 _GroundColor;
half _NakljucnostNormale;
StructuredBuffer<float3> _MedpomnilnikVsehPozicijTrave;
StructuredBuffer<uint> _MedpomnilnikIndeksovVsehVidnihPrimerov;
```

Izsek kode 5.44: Definicija spremenljivk in lastnosti

Ker upodabljamo zelo preprosto mrežo, bomo v tem primeru osvetljevanje opravili kar v senčilniku oglišč. Sama osvetljava trave v našem primeru ne bo osrednji del tega poglavja, zato bomo uporabili zelo preprost osvetljevalni model Half Lambert. Zasnovan je tako, da preprečuje, da bi zadnji del objekta izgubil obliko in izgledal preveč ravno (Izsek kode 5.45).

```
half3 Osvetlitev(Light svetloba, half3 normala, half3 smerPogleda, half
yPozicijaOS)
{
    half3 polovicniVektor = normalize(svetloba.direction + smerPogleda);

    //neposredna razprsenost
    half neposrednaRazprsenost = dot(normala, svetloba.direction) * 0.5 +
0.5; //half lambert, da ustvarimo iluzijo razprševanja podploskve (Sub
surface scattering)

    //neposredna zrcalnost
    float neposrednaZrcalnost =
pow(saturate(dot(normala,polovicniVektor)),4);


    neposrednaZrcalnost *= 0.1 * yPozicijaOS;//samo uporabimo na zgornji
površini, da simuliramo ambientno okluzijo

    half3 razsvetljava  = svetloba.color * (svetloba.shadowAttenuation *
svetloba.distanceAttenuation); // slabljenje svetlobe
    half3 rezultat = (neposrednaRazprsenost + neposrednaZrcalnost) *
razsvetljava;
    return rezultat;
}
```

Izsek kode 5.45: Zapis funkcije za osvetlitev

Večino dela bomo opravili v senčilniku oglišč. Pozicije trave tokrat preberemo iz medpomnilnika vidnih primerkov, zato potrebujemo indeks našega primerka, ki ga lahko pridobimo z uporabo semantične oznake `SV_InstanceID`. Za zanimivejši učinek bomo višino posamezne bilke trave izračunali psevdonaključno z uporabo sinusne funkcije in x komponente pozicije. Lahko bi zapisali tudi bolj napredno funkcijo za naključnost, kot v Poglavju 3.3.3. Posamezna bilka trave bo vedno obrnjena proti pogledu kamere, zato moramo pridobiti vektorje pogleda. Na podlagi teh vektorjev in lastnosti definiramo pozicijo trave. Pri tem moramo še upoštevati oddaljenost kamere od pogleda, saj se lahko zgodi, da bo posamezna bilka v danem trenutku manjša kot ena zaslonska pika, kar lahko vodi v neželeno utripanje. Tega problema se znebimo tako, da povečamo širino trave na podlagi oddaljenosti od pogleda (Izsek kode 5.46).

```
Varyings vert(Attributes IN, uint IDPrimerka : SV_InstanceID)
{
    Varyings OUT;

    float3 pozicijaTraveWS =
_MedpomnilnikVsehPozicijTrave[_MedpomnilnikIndeksovVsehVidnihPrimerov[IDPrime
rka]];

    float visinaTrave = lerp(2,5,(sin(pozicijaTraveWS.x*23.4643 +
pozicijaTraveWS.z) * 0.45 + 0.55)) * _VisinaTrave;



    float3 vektorKamereDesnoWS = UNITY_MATRIX_V[0].xyz;
    float3 vektorKamereGorWS = UNITY_MATRIX_V[1].xyz;
    float3 vektorKamereNaprejWS = -UNITY_MATRIX_V[2].xyz;

    float3 pozicijaOS = IN.pozicijaOS.x * vektorKamereDesnoWS * _SirinaTrave
* (sin(pozicijaTraveWS.x*95.4643 + pozicijaTraveWS.z) * 0.45 + 0.55);


    pozicijaOS += IN.pozicijaOS.y * vektorKamereGorWS;
    pozicijaOS.y *= visinaTrave;

    //povečamo širino trave na podlagi oddaljenosti pogleda
    float3 pogledWS = _WorldSpaceCameraPos - pozicijaTraveWS;
    float dolzinaVektorjaPogleda = length(pogledWS);
    pozicijaOS += vektorKamereDesnoWS * IN.pozicijaOS.x * max(0,
dolzinaVektorjaPogleda * 0.0225);


    //pretvorba iz objektnega v realni prostor
    float3 pozicijaWS = pozicijaOS + pozicijaTraveWS;
```

Izsek kode 5.46: Določitev pozicije, velikosti in širine trave

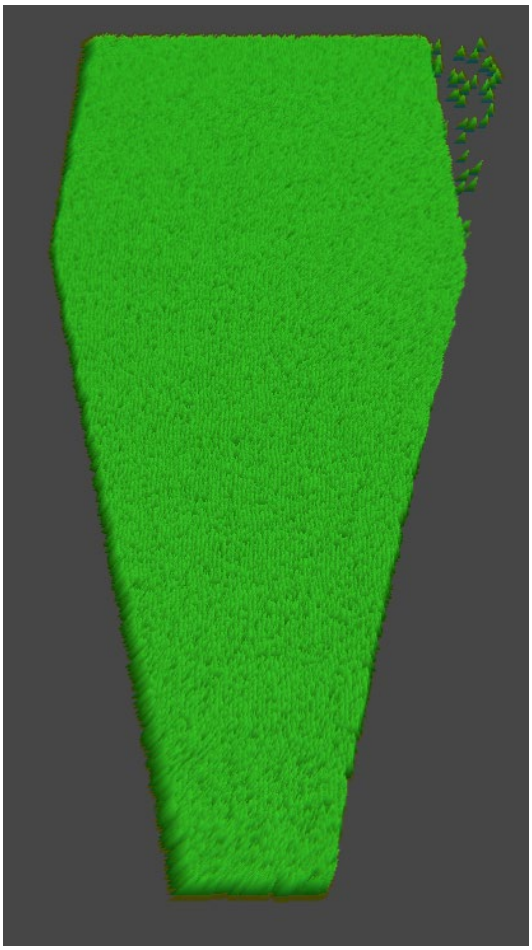
Implementirali bomo tudi premikanje v vetru, da je naša trava manj statična. To dosežemo z uporabo funkcije sinus. Sledi še definicija normale, ki bo v našem primeru vedno usmerjena navzgor, saj na ta način dobimo boljše osvetljevanje. Dodali bomo tudi malo naključnosti, da osvetljevanje ni preveč enotno, in upoštevali, da bo trava ves čas obrnjena proti pogledu kamere (Izsek kode 5.47).

```
    //veter
    float veter = 0;
    veter += (sin(_Time.y * _FrekvencaVetraEna + pozicijaTraveWS.x *
_PonavljanjeVetraEna.x + pozicijaTraveWS.z *
_PonavljanjeVetraEna.y)*_ZavijanjeVetraEna.x+_ZavijanjeVetraEna.y) *
_MocVetraEna;
    veter += (sin(_Time.y * _FrekvencaVetraDva + pozicijaTraveWS.x *
_PonavljanjeVetraDva.x + pozicijaTraveWS.z *
_PonavljanjeVetraDva.y)*_ZavijanjeVetraDva.x+_ZavijanjeVetraDva.y) *
_MocVetraDva;

    veter *= IN.pozicijaOS.y;

    pozicijaWS.xyz += vektorKamereDesnoWS * veter;

    OUT.positionCS = TransformWorldToHClip(pozicijaWS);


    Light svetloba;

    svetloba = GetMainLight();

    half3 nakljucnostNormale = (_NakljucnostNormale* sin(pozicijaTraveWS.x *
82.32523 + pozicijaTraveWS.z) + veter * -0.25) * vektorKamereDesnoWS


    half3 normala = normalize(half3(0,1,0) + nakljucnostNormale -
vektorKamereNaprejWS*0.5);

    half3 smerPogleda = pogledWS / dolzinaVektorjaPogleda;


    half3 rezultatRazsvetljave = lerp(_BarvaNaDnu,_GlavnaBarva,
IN.pozicijaOS.y);


    rezultatRazsvetljave += Osvetlitev(svetloba, normala, smerPogleda,
pozicijaOS.y);


    OUT.barva = rezultatRazsvetljave;

    return OUT;
}

half4 frag(Varyings IN) : SV_Target
{
    return half4(IN.barva,1);
}
```

Izsek kode 5.47: Implementacija vetra in osvetljevanje

Na zadnje pridemo do željenega rezultata, ki ga lahko spreminjamo po želji (Slika 5.15). Trava bo vedno prikazana le v pogledu (Slika 5.16).

Slika 5.15: Prikaz v igri



Slika 5.16: Prikaz v urejevalniku

6 Rezultati

Vsi senčilniki razen zadnjega, ki so predstavljeni v tej raziskovalni nalogi, so dostopni na naslednji spletni strani https://alenci.github.io/RaziskovalnaNaloga/ (priporočljiva je novejša verzija brskalnika FireFox). Na njej lahko uporabnik spreminja nekatere lastnosti prikazanih senčilnikov. Zadnji primer, ni prikazan, saj trenutni standard za nizkonivojsko 3D grafiko WebGL 2.0 ne podpira senčilnikov izračunavanja. Trenutno je v razvoju novi standard WebGPU, ki pa za čas pisanja te raziskovalne naloge še ni dostopen vsem razvijalcem. V namizno verzijo aplikacije, ki je dostopna na povezavi https://nabladev.itch.io/razi, pa je vključena tudi trava.

7 Ovrednotenje hipotez

H1 Senčilniki se uporabljajo za veliko več, kot le senčenje objektov.

Pred raziskavo sem imel občutek, da je velika večina senčilnikov napisana le za namene implementacije različnih modelov osvetljevanja. Zdelo se mi je, da so to programi, v katerih prevladujejo razne fizikalne kalkulacije glede senčenja in osvetljevanja. Skozi lastno raziskavo sem ugotovil, da obstaja veliko več primerov uporabe senčilnikov, in da obstajajo tudi senčilniki, ki sploh niso povezani z upodabljanjem objektov.

Skozi raziskavo sem implementiral dovolj senčilnikov, ki niso bili povezani zgolj le za senčenje, tako da lahko to hipotezo potrdim.

H2 Senčilniki so primeren uvod v računalniško grafiko in bi jih lahko obravnavali tudi pri srednješolskem pouku računalništva.

Te hipoteze ne morem potrditi, saj sta računalniška grafika in senčilniki bolj zapleteni področji, kot se mi je zdelo na začetku. Posledica tega je, da je velik del raziskovalne naloge usmerjen v predstavljanje konceptov računalniške grafike, ki so potrebni za vsaj neko osnovno znanje o senčilnikih.

Snov je tudi zelo hitro prešla v linearno algebro, ki presega srednješolsko matematiko. Skoraj vsa prikazana vsebina v tem raziskovalnem delu ni del srednješolskega izobraževanja.

H3 S senčilniki lahko spreminjamo stil računalniške grafike (bolj ali manj realistična) na parameteriziran način.

Med raziskavo sem odkril, da lahko senčilnike uporabimo kot dobro alternativo tradicionalnim metodam, ki temeljijo na zamudnem risanju objektov. Risano grafiko lahko namreč dosežemo tudi v treh dimenzijah. To je bistvena prednost, saj so animacijske tehnike v treh dimenzijah veliko manj časovno potratne, kot če je potrebno ročno narisati sličice. Še več, spreminjamo lahko tudi gladkost prehoda med barvami, določamo obrobe in menjujemo modele osvetljevanja, kar zelo pospeši razvoj projektov.

Ta uporaba senčilnikov se mi zdi zelo pomembna v današnji industriji animiranih filmov in iger, zato lahko to hipotezo potrdim.

H4 Senčilnike lahko uporabimo pri upodabljanju zelo velike količine objektov v realnem času.

Tudi danes je na določenih platformah redkeje videti upodabljanje številnih objektov zaradi same zapletenosti – potrebujemo zmogljivo strojno opremo. Če želimo upodobiti zelo veliko število dinamičnih objektov, nam poznavanje senčilnika za izračunavanje tu zelo koristi, še posebej, če aplikacija poteka v realnem času. To hipotezo lahko torej potrdim.

H5 S senčilniki, s katerimi opišemo volumne, lahko nekatere objekte upodobimo na lažji način, kot če bi jih ročno modelirali.

To hipotezo bi sicer lahko delno potrdil, saj je večina funkcij za definicijo polja razdalj geometrijskih primitivov preprostih, lahko pa tudi vzorčimo teksture. Razlog, da je ne morem potrditi pa je v tem, da je v večini primerov še vedno bolj enostavno modelirati zapletene objekte ročno, saj je te objekte težje opisati s poljem razdalj.

H6 Senčilniki za opisovanje volumnov so primerni za aplikacije, ki tečejo v realnem času.

Na spletu sem našel veliko virov, ki pravijo, da volumetrične metode niso primerne za aplikacije v realnem času.

Tudi v mojih rezultatih je razvidno, da so te metode računsko zahtevnejše in jih procesor izvršuje dlje. Po drugi strani pa služijo kot dobra alternativa v primeru, da so objekti sestavljeni iz preprostih primitivov, zato lahko to hipotezo delno potrdim.

8 Zaključek

V področje računalniške grafike sem se podal, ker mi je to predstavljalo izziv, danes pa vedno bolj ugotavljam, da je poznavanje teh konceptov tudi zelo koristno. Vedno več ljudi, ki se poda v to področje, ni zmožnih popolnoma uresničiti svoje umetniške vizije, ker se ne spoznajo na senčilnike. V dobi, kjer je izstopanje od običajnega najbolj pomembno za uspeh projekta, lahko na podlagi omejene količine virov trdim, da se premalo govori o dejanskem delovanju in implementacijah teh metod. Večinoma so za nepoznavalce nedostopne zaradi sorazmerno velike količine potrebnega predznanja, zato jih večina uporablja uporabniku prijazna orodja. V prihodnosti, ko bodo naši grafični procesorji še zmogljivejši, pa bodo procesirali še več kode, ki služi zgolj enostavnemu uporabniškemu vmesniku in ne v namen upodabljana objektov. Na podlagi tega menim, da se splača spoznati na implementacijo takšne kode, ki uporablja čim manj abstrakcij.

Pri pisanju dela se mi je pogosto zgodilo, da sem naletel na oviro, za katero sem potreboval veliko več matematičnega znanja, ob koncu pa se mi zdi, da bi lahko razumel tudi ostale metode, ki niso opisane v tej raziskovalni nalogi, če bi v njih vložil še nekaj časa, in upam, da ima takšno mnenje tudi bralec. Ta raziskovalna naloga predstavlja enostavnejše izmed vseh različnih metod računalniške grafike. To področje v Sloveniji še ni zelo raziskano, zato upam, da sem z implementacijo teh metod v trenutno najsodobnejšem okolju pripomogel k večjemu dostopu posameznikov in podjetij v to področje.

9 Bibliografija

Art, M. (18. 4 2018). *Quick Game Art Tips - Unity Liquid Shader*. Pridobljeno 2. 4 2021 iz Patreon: https://www.patreon.com/posts/quick-game-art-18245226

Golus, B. (2020). *The Quest for Very Wide Outlines.* Pridobljeno 2. 4 2021 iz Medium: https://bgolus.medium.com/the-quest-for-very-wide-outlines-ba82ed442cd9

hecomi. (2020). *uRaymarching*. Pridobljeno 2. 4 2021 iz GitHub: https://github.com/hecomi/uRaymarching

Hiruma, K. (9. 12 2018). *[WebGL] レイマーチングでアンチエイリアス（FXAA）してみる*. Pridobljeno 2. 4 2021 iz Qiita: https://qiita.com/edo_m18/items/c211fea23b4747a8da3c

Hiruma, K. (8. 10 2018). *レイマーチングでHeight Map Distance Field*. Pridobljeno 2. 4 2021 iz Qiita: https://qiita.com/edo_m18/items/af7fa86541634b59466f

Lengyel, E. (2019). *Foundations of Game Engine Development, Volume 2: Rendering.* Terathon Software LLC.

Markus, I., Jens, O., Barbara, S., Andreas, K., & Matthias, T. (2014). *SPH Fluids in Computer Graphics.* Pridobljeno 2. 4 2021 iz https://people.inf.ethz.ch/~sobarbar/papers/Sol14/2014_EG_SPH_STAR.pdf

Project, E. M. (5. 2 2021). Pridobljeno 2. 4 2021 iz Enjoy Math: https://enjoymath.pomb.org/?p=15

Quilez, I. (2. 6 2021). *fBm.* Pridobljeno 2. 4 2021 iz https://www.iquilezles.org/www/articles/fbm/fbm.htm

Rotation formalisms in three dimensions. (brez datuma). Pridobljeno 2. 4 2021 iz Wikipedia: https://en.wikipedia.org/wiki/Rotation_formalisms_in_three_dimensions

Tomas, M., Haines, E., & Hoffman, N. (2018). *Real-Time Rendering, Fourth Edition.* CRC Press.

Tschmits. (1. 1 2008). *UV mapping*. Pridobljeno 2. 4 2021 iz Wikipedia: https://en.wikipedia.org/wiki/UV_mapping

Ulf, A., & Tomas, M.¨ (1999). *Optimized View Frustum Culling Algorithms.* Pridobljeno 2. 4 2021 iz http://www.cse.chalmers.se/~uffe/vfc.pdf

Viewing frustum. (18. 6 2012). Pridobljeno 2. 4 2021 iz Wikipedia: https://en.wikipedia.org/wiki/Viewing_frustum

Vries, J. d. (brez datuma). *Coordinate Systems.* Pridobljeno 2. 4 2021 iz Learn OpenGL: https://learnopengl.com/Getting-started/Coordinate-Systems

WALKINGFAT. (16. 12 2018). *Sparkle Shader – 闪烁亮片材质.* Pridobljeno 2. 4 2021 iz WalkingFat: http://walkingfat.com/sparkle-shader-闪烁亮片材质/

陈嘉栋. (27. 9 2017). ***利用****GPU实现无尽草地的实时渲染.* Pridobljeno 2. 4 2021 iz Zhihu: https://zhuanlan.zhihu.com/p/29632347

IZJAVA

Mentor Boštjan Resinovič, v skladu z 2. in 17. členom Pravilnika raziskovalne dejavnosti »Mladi za Celje« Mestne občine Celje, zagotavljam, da je v raziskovalni nalogi naslovom Alternativne senčilne metode za računalniško grafiko, katere avtor je Alen Cigler

- besedilo v tiskani in elektronski obliki istovetno,
- pri raziskovanju uporabljeno gradivo navedeno v seznamu uporabljene literature,
- da je za objavo fotografij v nalogi pridobljeno avtorjevo dovoljenje in je hranjeno v šolskem arhivu,
- da sme Osrednja knjižnica Celje objaviti raziskovalno nalogo v polnem besedilu na spletnih portalih z navedbo, da je nastala v okviru projekta Mladi za Celje,
- da je raziskovalno nalogo dovoljeno uporabiti za izobraževalne in raziskovalne namene s povzemanjem misli, idej, konceptov oziroma besedil iz naloge ob upoštevanju avtorstva in korektnem citiranju,
- da smo seznanjeni z razpisni pogoji projekta Mladi za Celje.

Celje, 9.5.2021

ŠOLSKI CENTER CELJE
SREDNJA ŠOLA ZA ENERGIJO, ELEKTROTEHNIKO IN RAČUNALNIŠTVO
CELJE, Pot na Lavo 22
1 šole

Podpis mentorja(-ice)

Podpis odgovorne osebe