SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1012

INTERAKTIVNO OSLIKAVANJE OBJEKATA U VR OKRUŽENJU

Matija Kunc

Zagreb, srpanj 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1012

INTERAKTIVNO OSLIKAVANJE OBJEKATA U VR OKRUŽENJU

Matija Kunc

Zagreb, srpanj 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 3. ožujka 2025.

DIPLOMSKI ZADATAK br. 1012

Pristupnik:	**Matija Kunc (0036534412)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Interaktivno oslikavanje objekata u VR okruženju**

Opis zadatka:

Okruženje proširene stvarnosti VR (Virtual Reality) predstavlja nove izazove vezane posebno uz interakciju u takvom okruženju. Proučiti problematiku interakcije s površinom zadanog objekta u VR okruženju uz korištenje tehnologije primjerice HTC Vive sustava. Razviti skripte koje omogućuju interaktivno crtanje po objektima. Omogućiti promjenu boje, debljine linije i drugih parametara vezanih uz površinska svojstva objekta. Posebice obratiti pažnju na kvalitetu iscrtanih linija. Na nizu primjera ostvariti testiranje dobivenih rezultata. Analizirati i ocijeniti postignute rezultate. Diskutirati upotrebljivost rješenja kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 4. srpnja 2025.

Želio bih zahvaliti prof. dr. sc. Željki Mihajlović na mentorstvu, ugodnoj suradnji i vodstvu, kroz godine na preddiplomskom i diplomskom studiju.

Sadržaj

Uvod

Virtualna stvarnost jedno je od novijih, brzo rastućih područja suvremene tehnologije. Omogućuje korisnicima da urone u trodimenzionalne, interaktivne svjetove, daje im mogućnost virtualnog posjećivanja udaljenih mjesta iz udobnosti njihovog doma. Razvojem hardverskih rješenja ove tehnologije, kao virtualnih naočala, rukavica, kontrolera, pratitelja pokreta, i slično, povećava se i mogućnost interakcije stvarnog korisnika s njegovim virtualnim okruženjem. Jedna od tih interakcija je ocrtavanje virtualnih objekata. Ova interakcija nalikuje stvarnom crtanju ili pisanju po fizičkim objektima, prebačena u virtualni svijet, s virtualnim objektima.

Ocrtavanje objekata u VR-u ima široku primjenu – od kreativnih alata i edukacijskih aplikacija do industrijskog dizajna, medicinske simulacije i proširene suradnje na daljinu. Cilj ove tehnologije je da omogući korisnicima da ostave svoj trag u virtualnom svijetu, koji netko drugi kasnije može pronaći.

Implementacija ovakvog sustava ima brojne izazove, za mnogo kojih još nije pronađeno savršeno rješenje. Razlog tomu je što je ovo vrlo novo područje istraživanja i nemamo još dovoljno iskustva s njime. Neki od problema s kojima se susrećemo u ovom području su: praćenje pokreta korisnikove ruke ili alata za crtanje, precizno određivanje točke kontakta s površinom objekta, mapiranje crteža na UV koordinate objekta, te iscrtavanje crteža u stvarnom vremenu bez narušavanja performansi. Posebni izazovi u ovom području su i crtanje po zakrivljenim površinama, crtanje preko UV šavova, teksture koje nisu usklađene s geometrijom tijela.

Ovaj rad bavi se istraživanjem raznih metoda ocrtavanja virtualnih objekata u virtualnoj stvarnosti. Analizirat će se različiti pristupi ocrtavanju, dublje će se ući u probleme koji su gore navedeni i prikazat će se prednosti i nedostaci različitih pristupa.

1. Osnovni pojmovi i korištene tehnologije

1.1. Grafički programski pogon Unity

Unity je grafički programski pogon (eng. *Game engine*) korišten za razvoj 2D i 3D interaktivnih aplikacija i igara. Unity podržava proširenu (AR) i virtualnu stvarnost (VR). Kao glavni programski jezik, Unity koristi C#, u kojem se piše gotovo sva logika vezana uz aplikaciju. Također je podržan i HLSL (*High Level Shader Language* ili *High Level Shading Language)*, jezik u kojem se pišu sjenčari (eng. *Shaders*), o kojima ćemo detaljnije u kasnijem poglavlju. Kako bi Unity podržao VR, nudi razne gotove standarde razvijene upravo za tu upotrebu. U ovom radu korištena biblioteka je *OpenXR* [1], standard koji je razvijen kako bi olakšao razvoj AR/VR aplikacija i koji podržava velik broj AR/VR sustava (u ovom radu korišten HTC Vive).

1.2. Konfiguracija računala

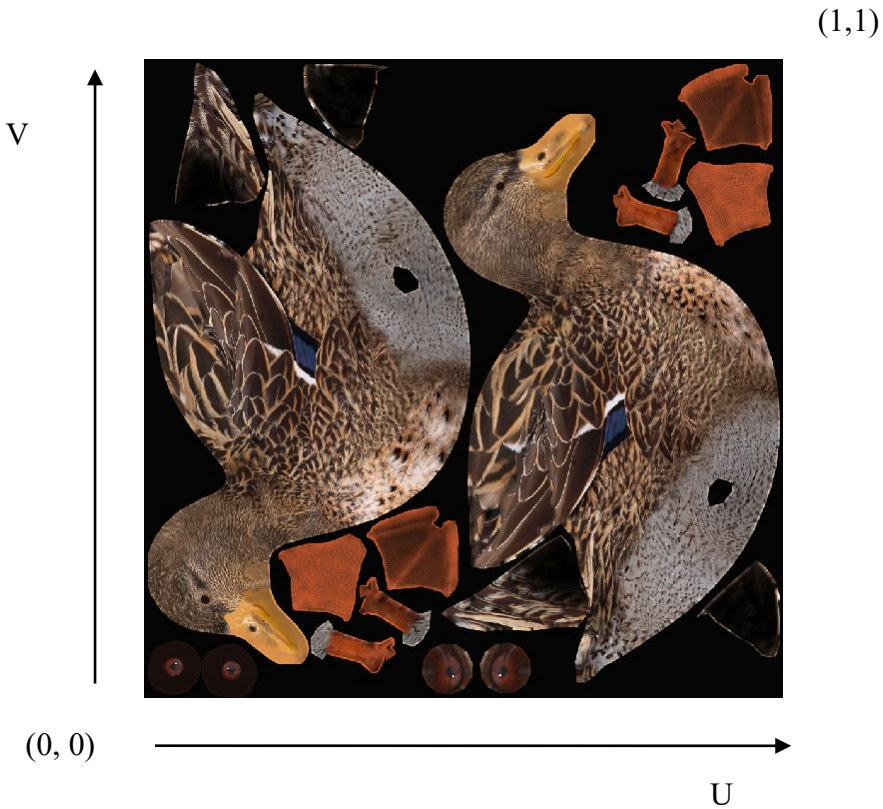
Kao što je navedeno u prethodnom potpoglavlju, korištena tehnologija za virtualnu stvarnost je HTC Vive. Za korištenje HTC Vive sustava bilo je potrebno koristiti *SteamVR*, programsku podršku specifično za tu vrstu opreme. Računalo na kojem je program napravljen i na kojemu se izvodio koristi grafičku karticu *Nvidia GeForce RTX 3060*, i procesor *Intel Core i5-8400*, sa 6 jezgri. Program se izvodio uz zadovoljavajuće performanse, iako je u nekim slučajevima dolazio do manjih problema sa nekim metodama crtanja koja su testirana (kratko kašnjenje, zastajanje i slično).

1.3. Teksture

Virtualne scene su virtualni prostori unutar kojih se korisnici nalaze i gibaju. One sadrže razne objekte s kojima je moguće ili nemoguće interagirati. Objekti unutar scene izgledali bi prazno i jednobojno ako ne bi imali na sebi neki materijal. Materijali pak, unutar Unityja, ali i sličnih programskih pogona (npr. *Unreal Engine*) primaju razne teksture koje objektima daju, boju, dubinu, neravnine i slično. Teksture su zapravo dvodimenzionalni omotači oko objekata, precizno (ili neprecizno) mapirani tako da se određene 2D točke

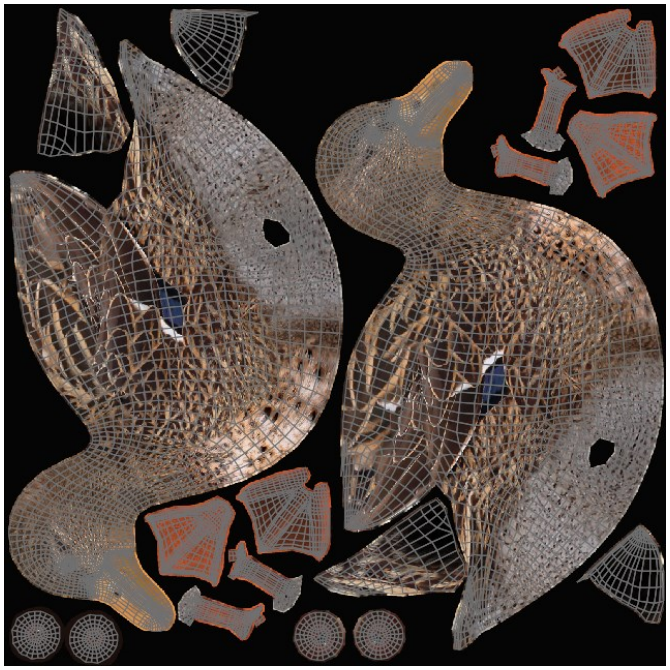
teksture (nadalje zvane UV koordinate) točno preklope s njihovim pripadnim točkama na mreži (eng. *Mesh)* objekta, te objektu daju njegov konačni izgled. Teksture su za ovaj rad bitne jer se u nekim metodama direktno koriste za crtanje po objektima, jer u konačnici boja koja se nanosi na točku objekta u 3D prostoru, će završiti na njenom pripadnom mjestu u teksturi. U računalnoj grafici koristi se više vrsta tekstura, od kojih svaka ima svoju specifičnu svrhu. Primjerice, teksture boje određuju osnovnu boju površine objekta. Teksture normala služe za simulaciju sitnih neravnina na površini bez mijenjanja same geometrije objekta, čime se postiže dojam većeg detalja u osvjetljenju. Tu su i teksture visine (eng. *height maps*), koje se mogu koristiti za stvarno pomicanje površine objekta, primjerice u tehnici *displacement mappinga*.

U ovom radu fokusirat ćemo se na teksture boje, jer su one ključne za simulaciju bojanja po objektu — one izravno određuju kako će površina objekta izgledati u smislu boje.



Slika 1.1: Tekstura patke, bez mreže

Na slici gore vidimo kako bi izgledala tekstura 3D objekta patke, to je kao što smo rekli, 2D prostor, kojeg određuje UV koordinatni sustav. UV koordinatni sustav kreće tipično od donjeg lijevog kuta (npr. *OpenGL)*, ali se nekada uzima kao početna točka i gornji lijevi kut (npr. *DirectX)*, gdje je U koordinata je horizontalna, a V vertikalna. Dobro je napomenuti da se UV sustav uobičajeno normalizira na vrijednosti [0, 1], tako da je (0, 0) ishodište.



Slika 1.2: Tekstura patke s mrežom

Na ovoj slici ponovno se nalazi ista tekstura patke, ali ovaj put je na njoj ocrtana mreža objekta, koja je razmotana u 2D prostoru. Na ovoj slici možemo lijepo vidjeti koji se točno trokut, ili u ovom slučaju kvadrat, mapira na koji dio teksture. Dobro razmotane teksture su vrlo bitne ako se po njima nešto želi omogućiti crtati, a zašto je to tako, ćemo dublje objasniti u daljem poglavlju. Cilj ovog poglavlja je dati kratak uvid u osnove tekstura, kako bi dalje u radu čitatelj lakše pratio što se događa, ako nije imao prethodna znanja o ovoj temi.

1.4. Sjenčari

U prethodnom poglavlju dali smo kratki uvid u teksture, koje su povezane s materijalima u programskim pogonima, kao *Unity*. Sada ćemo se kratko osvrnuti na još jednu cjelinu koja je također usko povezana s materijalima, ali je bitna i generalno u računalnoj grafici, a to su sjenčari (eng. *Shaders)*. Sjenčari su manji programi, specijalizirani da se izvode na grafičkoj kartici koji kontroliraju kako se objekti iscrtavaju. Pisani su u specijaliziranim jezicima kao HLSL, GLSL, ShaderLab i slično, i oni govore grafičkoj kartici kako transformirati vrhove i bojati piksele [2]. Postoje razne vrste sjenčara: sjenčari vrhova (eng. *Vertex shaders*), fragmenata (eng. *Fragment shaders*), geometrije (eng. *Geometry shaders*), itd. Nama su za ovaj rad zanimljivi sjenčari vrhova i fragmenata. Oni su korišteni uz teksture kako bi se ostvarilo oslikavanje objekata. U suštini, način na koji rade sjenčari je sljedeći, dohvatimo bilo koji vrh, recimo nekog objekta, sjenčar vrhova ima zadatak, pretvoriti koordinate tog vrha u prostoru (3D prostoru) i odrediti njegovu poziciju na ekranu (2D prostoru). To se postiže množenjem tog vrha s nekoliko transformacijskih matrica, u slučaju *Unityja*, vrh je na početku u prostoru svog roditeljskog objekta, prvom matricom (*Model Matrix*) ga prebacujemo u prostor svijeta, zatim sljedećom matricom (*View Matrix*) ga prebacujemo iz prostora svijeta u prostor pogleda kamere, i na kraju ga množimo s matricom projekcije (*Projection Matrix*), kojom se vrh konačno transformira u prostor ekrana. Ovaj postupak ponavlja se za sve vrhove koje vidimo i ponavlja se u svakom iscrtavanju, nakon čega dobivamo konačne pozicije svih točaka na ekranu. Nakon toga potrebno je te točke i sve piksele obojiti. Ovdje dolazi na red sjenčar fragmenata koji od sjenčara vrhova dobiva razne informacije, kao UV koordinate, normale, pozicije u svijetu i na ekranu za sve vrhove, i na temelju tih podataka sjenčar fragmenata određuje boje svakog vidljivog piksela. Primjer jednog jednostavnog programa koji sadrži sjenčar vrhova i fragmenata prikazan je na sljedećoj stranici.

```
Shader "Custom/MyShader"
{
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader {
        Pass {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            sampler2D _MainTex;
            float4 _Color;

            struct appdata {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f {
                float2 uv : TEXCOORD0;
                float4 pos : SV_POSITION;
            };

            v2f vert (appdata v) {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv = v.uv;
                return o;
            }

            fixed4 frag (v2f i) : SV_Target {
                fixed4 texColor = tex2D(_MainTex, i.uv);
                return texColor * _Color;
            }
            ENDCG
        }
    }
}
```

Kôd 1.1 – Sjenčar

Program na slici ne radi neki poseban posao, on uzima piksele na glavnoj teksturi objekta i dodaje boju na njih te tako mijenja nijansu objekta u zadanu boju. Ovo je jednostavan primjer sjenčara, ali dobar za pokazati princip na kojem rade sjenčari u *Unityju*. U prvom retku sjenčaru postavljamo ime i skupinu sjenčara u kojoj ćemo ga spremiti (kojih *Unity* već ima mnogo unaprijed definiranih). Sljedeće se postavljaju potrebna svojstva, ovo mogu biti razni tipovi podataka, brojevi, teksture, boje, opsezi (u smislu skala brojeva od npr. 1 do 10) i slično. Svojstva su zapravo ulazne informacije koje dajemo sjenčaru da ih obrađuje. U ovom slučaju imamo samo dvije ulazne varijable ili svojstva, glavnu teksturu i boju. Zatim kreće glavni kod programa, označen sa *SubShader* u kojem se nalazi sva logika, *Pass* označava prolaz sjenčara po vrhu, to je jedan korak u iscrtavanju koji ima svoj sjenčar vrhova i fragmenata. Moguće je imati više prolaza ako želimo ponovno proći po vrhovima i dodati, primjerice, sjene ili simulirati refleksiju svjetlosti. Program se otvara s *CGPROGRAM* kao oznaka za početak bloka koda u CG/HLSL-u kojeg *Unity* koristi.

S *#pragma vertex vert* i *#pragma fragment frag* označavamo funkcije koje ćemo definirati i koristiti u ovom slučaju to su sjenčar vrhova (nazvali smo ga vert) i sjenčar fragmenata (nazvali smo ga *frag*). *_MainTex* i *_Color* su varijable koje deklariramo. *Appdata* i *v2f* su strukture podataka koje koristimo.

```
struct appdata {
float4 vertex : POSITION; -> pozicija vrha u prostoru objekta
float2 uv : TEXCOORD0; -> pozicija vrha u UV teksturi
};

struct v2f {
float2 uv : TEXCOORD0; -> pozicija vrha u UV teksturi
float4 pos : SV_POSITION; -> pozicija na ekranu
};
```

Sada kada imamo sve potrebne strukture i podatke, dolazi na red sjenčar vrhova. On na ulazu dobije strukturu *appdata v* koju pretvara u strukturu *v2f* (*vertex to fragment*) koja ide

sjenčaru fragmenata. Sjenčar vrhova samo proslijedi UV koordinate jer se u ovom slučaju ne mijenjaju, a poziciju vrha iz prostora objekta pretvara u poziciju na ekranu s linijom:

```
o.pos = UnityObjectToClipPos(v.vertex);
```

Ova linija radi upravo onaj umnožaka točke s matricama koji smo prethodno opisali i transformira točku u prostor ne ekranu.

Na kraju dolazi sjenčar fragmenata na red i radi sljedeće:

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 texColor = tex2D(_MainTex, i.uv);
    return texColor * _Color;
}
```

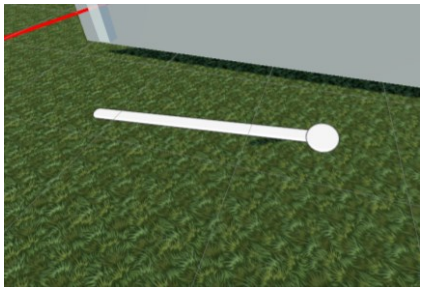
Dobiva na ulaz *v2f* strukturu koju me šalje sjenčar vrhova, dohvaća vrijednost boje piksela na UV koordinatama teksture i vraća tu vrijednost pomnoženu s našom odabranom bojom, tako da konačna boja bude u boja teksture, ali u nijansama neke odabrane boje. Sličan postupak, ali ne u potpunosti jednak, će se koristiti u implementaciji oslikavanja objekta, koju ćemo opisati u sljedećem poglavlju.

2. Implementacija

2.1. Scena i skripte za podešavanje oslikavanja

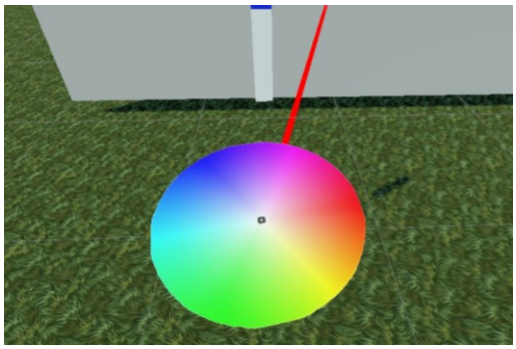
Prva i osnovna stvar koju svaki projekt u *Unityju* mora imati je scena. Scena je prostor u kojemu će se nalaziti svi objekti i unutar kojega će se korisnik kretati i oslikavati objekte. Za potrebe testiranja u scenu su uvezeni razni besplatni materijali dostupni u *Unity Asset Storeu*, tu su dobiveni modeli objekata, skybox, neke teksture terena i slično. Veoma je bitna biblioteka za korištenje opreme za virtualnu stvarnost *OpenXR*, koju smo spomenuli u prethodnom poglavlju. Unutar ove biblioteke dolazi već postavljen objekt koji na sebi sadrži sve potrebne skripte za praćenje pokreta kontrolera i kamere naočala. Ovaj objekt, zvan *Camera Rig*, je bilo potrebno dodati scenu kako bi se omogućilo korištenje opreme, kretanje u prostoru i praćenje pokreta glave i ruku, odnosno kontrolera. Ova biblioteka sadrži još jednu skriptu *XR Grab Interactable* koja omogućava uzimanje markera za crtanje u ruku i njegovo pomicanje s pomoću kontrolera. U sceni je postavljeno još nekoliko stvari, poput skyboxa, ali to nije toliko bitno za sâm projekt pa ćemo taj dio zanemariti.

Sa scenom postavljenom i dodanim objektima za testiranje, sada je potrebno napraviti korisničko sučelje u virtualnoj stvarnosti. Osim samog oslikavanja u radu je bilo potrebno i napraviti mogućnost odabira boje markera i debljine linije koju ostavlja kao trag. Stoga su u svrhu toga dodani UI elementi kojima te parametre možemo kontrolirati. Za promjenu debljine markera korišten je klizač (*UI XR* objekt), kojim ćemo mijenjati debljinu markera ovisno o tome koliko je klizač pomaknut. Kako bi se ostvarila promjena debljine u skripti za crtanje koja će biti prikazana kasnije bilo je potrebno napraviti referencu na ovaj klizač i onda samo dohvatiti trenutačnu vrijednost i koristiti ju u izračunu.



Slika 2.1: Klizač za promjenu debljine markera

Promjena boje markera ostvarena je s pomoću slike kotača boje, skripte i materijala vrha markera.



Slika 2.2: Kotač za odabir boje markera

Skripta detektira pritisak okidača na kontroleru, detektira točku presjeka zrake iz kontrolera sa slikom kotača boje, i dohvaća vrijednost boje pogođenog piksela, nakon čega mijenja boju materijala vrha markera u tu boju. Skripta je prikazana u nastavku:

```
public class ColorPicker : MonoBehaviour
{
    [Header("References")]
    public XRRayInteractor rightRayInteractor;
    public ActionBasedController rightController;
    public MeshRenderer colorWheelRenderer;
    public Marker markerScript;

    void Update()
    {
        if (rightController != null)
        {
             float actionValue =
      rightController.selectAction.action.ReadValue<float>();

            if (actionValue > 0.1f)
            {

                if (rightRayInteractor != null &&
rightRayInteractor.TryGetCurrent3DRaycastHit(out RaycastHit hit))
                {

                    if (hit.collider != null && hit.collider.gameObject ==
colorWheelRenderer.gameObject)
                    {
                        Vector2 textureCoord = hit.textureCoord;

                        Color selectedColor = GetColorAtHit(textureCoord);

                        markerScript.SetMarkerColor(selectedColor);
```

```
                    }
                }
            }
        }
    }

    Color GetColorAtHit(Vector2 uvCoords)
    {
        Texture2D texture = colorWheelRenderer.material.mainTexture as
Texture2D;
        if (texture == null)
        {
            return Color.blue;
        }

        Vector2 pixelCoords = new Vector2(uvCoords.x * texture.width,
uvCoords.y * texture.height);
        int x = Mathf.FloorToInt(pixelCoords.x);
        int y = Mathf.FloorToInt(pixelCoords.y);

        x = Mathf.Clamp(x, 0, texture.width - 1);
        y = Mathf.Clamp(y, 0, texture.height - 1);

        Color color = texture.GetPixel(x, y);
        return color;
    }
}
```

Kôd 2.2 – Program za odabir i promjenu boje markera

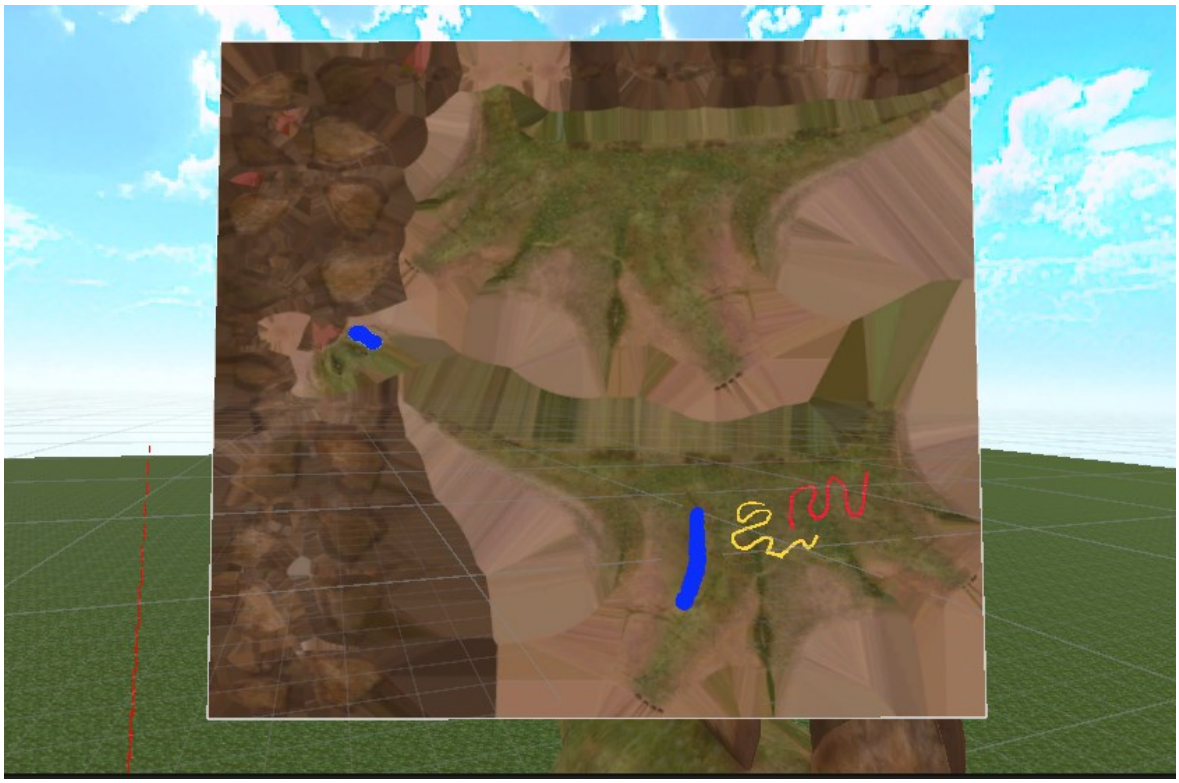
Dodatni UI element koji je napravljen je 2D prikaz teksture pogođenog objekta koji se osvježava u stvarnom vremenu kako bi se moglo vidjeti koje točke mijenjaju boju pri oslikavanju.

```
public class UVPreviewPanel : MonoBehaviour
{

    [Header("UI")]
    public RawImage previewImage;

    [Header("Target")]
    public Renderer targetRenderer;
    public string textureProperty = "_DrawTex";
    public Material uiPreviewMaterial;

    void Update()
    {
        if (targetRenderer == null || uiPreviewMaterial == null) return;

        previewImage.material = uiPreviewMaterial;

        var baseTex = targetRenderer.material.GetTexture("_MainTex");
        var overlayTex = targetRenderer.material.GetTexture("_DrawTex");

        if (baseTex != null)
            uiPreviewMaterial.SetTexture("_BaseTex", baseTex);

        if (overlayTex != null)
            uiPreviewMaterial.SetTexture("_OverlayTex", overlayTex);
```

```
        previewImage.material = uiPreviewMaterial;
    }

    public void SetTarget(Renderer renderer)
    {
        targetRenderer = renderer;
    }
}
```

Kôd 2.3 – Program za osvježavanje prikaza teksture

Kako bi se sama tekstura dohvatila, potrebno je metodom *SetTarget* dohvatiti *Renderer* komponentu objekta kojeg oslikavamo. Ova metoda poziva se u skripti za crtanje kada smo dohvatili objekt po kojem želimo crtati, nakon čega se u prozoru nad lijevim kontrolerom prikazuje tekstura pogođenog objekta.



Slika 2.3 : Panel za prikaz teksture trenutnog objekta (tekstura dinosaura)

2.2. Oslikavanje temeljeno na teksturi

Prva metoda koja je implementirana u radu je oslikavanje temeljeno na teksturi objekta, pa ćemo u ovom poglavlju objasniti ideju iza ove metode, način na koji radi i kako je implementirana u ovom radu.

Prvo ćemo pokazati postupak kojim se oslikavanje postiže, nakon čega ćemo svaki korak malo detaljnije objasniti i pogledati skripte kojima je ovo postignuto. Ideja je sljedeća: korisnik u ruci drži marker, na pritisak okidača na kontroleru iz vrha markera bit će ispucana zraka, koja će se presjeći s nekim objektom. Kada smo detektirali objekt izvršit će se nekoliko provjera prije nego krenemo dalje, neki od njih su provjera nalazi li se markera dovoljno blizu objekta kojega je zraka pogodila, ima li taj objekt *Renderer* objekt na sebi, je li dopušteno crtanje po tom objektu i slično. Kada su sve provjere prošle uspješno dohvatit ćemo točku na objektu u kojoj ga zraka sječe, pretvorit ćemo tu točku iz koordinata objekta (3D prostora) u UV koordinate teksture (2D prostor). Kada smo dohvatili 2D koordinate točke tu ćemo točku obojiti u boju materijala vrha markera. Osim te točke treba pobojiti i točke oko nje u radijusu koji ovisi o debljini linije markera. Ovaj postupak ponavlja se u svakom iscrtavanju okvira (*frameu)*, što znači da ako smo ispunili sve uvjete za crtanje, skripta će pobojati jedan krug na teksturi u svakom okviru. Međutim, ako bi pomicali marker brže nego je moguće skripti bojiti teksturu ostalo bi nam praznog prostora između dviju točaka, i dobili bi isprekidanu liniju. Ovaj problem rješava se linearnom interpolacijom između prethodne i trenutačne točke koje bojimo. Linearnom interpolacijom između tih dviju točaka dobivamo još *n* točaka ovisno o broju koraka koje smo zadali u interpolaciji.

Ovim postupkom dobiveno je oslikavanje po objektima temeljeno na teksturi objekta. Za cijeli ovaj postupak potrebno je nekoliko stvari. Prvo trebamo osigurati da objekt ima teksturu na sebi, odnosno da postoji razmotana mreža objekta koju ćemo koristiti za dohvaćanje 2D koordinate točke presjeka zrake i objekta. Tekstura je ono što ćemo mijenjati pri oslikavanju, no nekada nije dobro mijenjati originalnu teksturu objekta, ako želimo pohraniti promjene, ali kasnije se predomislimo i želimo original nazad, ne bi bilo lako dobiti original nazad iz promijenjene teksture. Kako bi izbjegli ovaj problem, napravit ćemo još jednu teksturu, prozirnu koju ćemo razmotati na isti način kao i originalnu i staviti je na objekt. Kada ćemo htjeti nešto oslikavati, mijenjat ćemo tu prozirnu teksturu i mijenjati njezine piksele u neprozirnu boju materijala markera te ćemo u konačnici dobiti

isti izgleda kao i da smo mijenjali original, ali uz prednost da original sačuvamo. Također, na ovaj način je implementacija brisanja boje jednostavna, samo ćemo obojeni dio pretvoriti u prozirnu boju, i ispod nje će se ponovno vidjeti originalna tekstura. Dobro je primijetiti da bi brisanje bilo puno teže ostvariti da smo mijenjali originalnu teksturu, jer bi u tom slučaju morali pamtiti što smo mijenjali i koje su bi boje originalno na tim koordinatama. U radu je ova tekstura implementirana s pomoću sjenčara i *RenderTexture* objekta. U nastavku je prikazan sjenčar za ovu metodu:

```
Shader "Custom/DrawingShader"
{
    Properties
    {
        _MainTex ("Base Texture", 2D) = "white" {}
        _DrawTex ("Drawn Texture", 2D) = "black" {}
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" "Queue"="Geometry"
"DisableBatching"="True" "LightMode"="ForwardBase" }
        Blend SrcAlpha OneMinusSrcAlpha

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile_instancing
            #pragma multi_compile _ UNITY_SINGLE_PASS_STEREO
            #include "UnityCG.cginc"

            struct appdata_t
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
                UNITY_VERTEX_INPUT_INSTANCE_ID
            };

            struct v2f
            {
```

```
                float2 uv : TEXCOORD0;
                float4 pos : SV_POSITION;
                UNITY_VERTEX_OUTPUT_STEREO
            };

            sampler2D _MainTex;
            sampler2D _DrawTex;
            float4 _BrushColor;

            v2f vert (appdata_t v)
            {
                v2f o;
                UNITY_SETUP_INSTANCE_ID(v);
                UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);

                o.pos = UnityObjectToClipPos(v.vertex);
//mul(UNITY_MATRIX_MVP, v.vertex)
                o.uv = v.uv;
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 baseColor = tex2D(_MainTex, i.uv);
                fixed4 drawColor = tex2D(_DrawTex, i.uv);
                return lerp(baseColor, drawColor,
drawColor.a);
            }

            ENDCG
        }
    }
}
```

Kôd 2.4 – *Shader* za oslikavanje

U *Properties* dijelu vidimo dva parametra, *_MainTex* koji označava originalnu teksturu, *_DrawTex* koji je objekt tipa *RenderTexture* koji će se bojiti. U *SubShaderu* se nalazi nekoliko parametara unutar *Tags* objekta, no oni su neke komande koje koristi Unity

render pipeline i nisu toliko važne pa ćemo ih preskočiti. Vidimo deklaracije naših sjenčara *vert* i *frag*, deklaracije nekih Unity funkcija, koje su informacija Unityju da radimo na sjenčaru koji mora podržavati virtualnu stvarnost i dodatno višestruko instanciranje kako bi objekti koji koriste istu *render* teksturu se razlikovali i bili neovisni jedni o drugima. Bez *#pragma multi_compile _ UNITY_SINGLE_PASS_STEREO* deklaracije Unity ne zna da treba uzeti u obzir da kamera ima dvije perspektive, jednu iz lijevog i jednu iz desnog oka naočala, pa će doći do efekta u kojemu objekt koji nosi materijal s ovim sjenčarom vidimo samo na jedno oko kroz naočale. *#pragma multi_compile_instancing* govori Unityju da ćemo koristiti višestruko instanciranje, kako bi omogućili da isti sjenčar koristimo za različite objekte koji koriste istu teksturu. Unity će dodijeliti ID svakoj kako bi GPU razlikovao vrhove više instanci iste teksture. Strukture koje koristimo ne razlikuju se previše od onih koje smo koristili u pokaznom primjeru u prijašnjem poglavlju. Sjenčar vrhova se ne razlikuje previše u od prijašnjeg izuzev dodatne linije koje su tu zbog toga što ovaj sjenčar podržava VR i višestruko instanciranje.

Sjenčar fragmenata malo je drugačiji. On uzima podatke od sjenčara vrhova, dohvaća boje točke na originalnoj teksturi, dohvaća iste te koordinate na teksturi po kojoj oslikavamo, i vraća interpoliranu boju između originalne teksture, teksture za oslikavanje i prozirnost teksture za oslikavanje *lerp(baseColor, drawColor, drawColor.a)*. Ovim programom dobivamo prozirnu teksturu preko objekta koju ćemo mijenjati. Sve što sada treba napraviti je kreirati novi materijal, dodijeliti mu naš novi *shader* program, i dati referencu na originalnu teksturu objekta na kojem ćemo ga koristiti i teksturu za oslikavanje, kako bi im program mogao pristupiti.

Objektima s ovim materijalom smo omogućili oslikavanje, sada ćemo pogledati skriptu kojom zapravo oslikavamo. Skripta je poveća pa ćemo je gledati po dijelovima kako dolaze na red kod jednog iscrtavanja.

```
void Update()
{
    if (!IsRightTriggerPressed())
    {
        lastDrawUV = null;
        return;
    }
    if (penVisual != null && penVisual.material.color != brushColor)
    {
        SetBrushColor(penVisual.material.color);
```

```
        }
        if(brushSize != brushSizeSlider.value)
        {
            SetBrushSize(brushSizeSlider.value);
        }
        if (Physics.Raycast(penTip.position, penTip.forward, out RaycastHit
hit))
        {
            if (hit.collider.CompareTag("Drawable"))
            {
                float distance = Vector3.Distance(hit.point,
transform.position);
                if (distance <= maxDrawDistance)
                {
                    Renderer rend = hit.collider.GetComponent<Renderer>();
                    if (rend == null) return;

                    Material mat = rend.material;
                    RenderTexture rt = mat.GetTexture("_DrawTex") as
RenderTexture;

                    if (rt == null) return;

                    Texture2D temp = new Texture2D(rt.width, rt.height,
TextureFormat.RGBA32, false);
                    RenderTexture.active = rt;
                    temp.ReadPixels(new Rect(0, 0, rt.width, rt.height), 0,
0);
                    temp.Apply();

                    if (TryGetUV(hit, out Vector2 uv))
                    {

uvPreviewPanel.SetTarget(hit.collider.GetComponent<Renderer>());
                        if (lastDrawUV.HasValue)
                        {
                            Vector2 from = lastDrawUV.Value;
                            Vector2 to = uv;
                            float distanceUV = Vector2.Distance(from, to);
                            float scale = Mathf.Max(brushSize, 0.001f);
                            int steps = Mathf.CeilToInt(distanceUV / scale *
5f);
                            steps = Mathf.Clamp(steps, 1, 50);


                            for (int i = 0; i <= steps; i++)
                            {
                                float t = i / (float)steps;
                                Vector2 lerpedUV = Vector2.Lerp(from, to, t);
                                DrawAtUV(temp, lerpedUV, hit, brushSize,
brushColor);
                            }
                        }
                        else
                        {
                            DrawAtUV(temp, uv, hit, brushSize, brushColor);
                        }

                        temp.Apply();
                        Graphics.Blit(temp, rt);
                        RenderTexture.active = null;
                        Destroy(temp);
                        lastDrawUV = uv;
```

```
                }
            }
        }
    }
    else
    {
        lastDrawUV = null;
    }
}
```

Kôd 2.5 – *Update* funkcija skripte za oslikavanje

Update() funkcija se poziva u svakom iscrtavanju ekrana, što znači između 30 i 60 puta u sekundi, ovisno o postavkama iscrtavanja. Prvo provjeravamo jesmo li pritisnuli okidač, ako nismo izlazimo iz funkcije i funkcija se prekida. Ako smo ga pritisnuli, provjeravamo je li se promijenila boja ili debljina i nju osvježavamo s funkcijama prikazanima niže:

```
public void SetBrushColor(Color newColor)
{
    brushColor = newColor;
}

public void SetBrushSize(float newSize)
{
    brushSize = newSize;
}
```

Kôd 2.6 – Funkcije za osvježavanje boje i debljine markera

Zatim ispucavamo zraku, pozivamo *Raycast* funkciju koja nam vraća informacije o pogođenom objektu. Ako je po objektu dopušteno oslikavanje (ima *tag „Drawable“*), distanca je dovoljno mala između točke pogotka i vrha markera, pogođeni objekt sadrži objekt *Renderer* i njegov materijal ima *Render Texture* objekt, onda krećemo u oslikavanje. Prvo ćemo napraviti privremenu teksturu *temp*, i u nju ćemo pročitati sve vrijednosti piksela s teksture za oslikavanje. Ovo radimo zato što Unity nema mogućnost direktnog mijenjanja *Render Texture* objekta s CPU-a, na kojemu se izvodi naš C# kôd jer se svi podaci o tom objektu nalaze na GPU [1]. Objekt tipa Texture2D je s druge strane moguće mijenjati s C# kodom na CPU. Zašto smo onda koristili *Render Texture* objekt za teksturu za oslikavanje? Pa iz istog razlog što C# kôd na CPU ne može mijenjati piksele s

Render Texture objekta jer se nalazi na GPU, tako ni sjenčar, koji se izvodi na GPU ne može mijenjati podatke s *Texture2D* objekta, koja se nalazi na CPU.

```
                    Texture2D temp = new Texture2D(rt.width, rt.height,
TextureFormat.RGBA32, false);
                    RenderTexture.active = rt;
                    temp.ReadPixels(new Rect(0, 0, rt.width, rt.height), 0,
0);
                    temp.Apply();
```

Kôd 2.7 – Isječak kôda za stvaranje privremene teksture

Sada dohvaćamo UV koordinate pogođene točke s funkcijom *TryGetUV*, koja prima pogođeni objekt i vraća nam UV koordinate pogođene točke. Funkcija izgleda ovako:

```
    bool TryGetUV(RaycastHit hit, out Vector2 uv)
    {
        uv = Vector2.zero;

        if (!TryGetMeshData(hit, out Vector3[] vertices, out Vector2[] uvs,
out int[] triangles)) return false;

        int i0 = triangles[hit.triangleIndex * 3 + 0];
        int i1 = triangles[hit.triangleIndex * 3 + 1];
        int i2 = triangles[hit.triangleIndex * 3 + 2];

        Vector2 uv0 = uvs[i0];
        Vector2 uv1 = uvs[i1];
        Vector2 uv2 = uvs[i2];

        Vector3 bary = hit.barycentricCoordinate;
        uv = uv0 * bary.x + uv1 * bary.y + uv2 * bary.z;

        return true;
    }
```

Kôd 2.8 – *TryGetUV* funkcija

Vidimo da funkcija prvo poziva *TryGetMeshData* funkciju, koja nam vraća vrhove listu vrhova u 3D prostoru, listu vrhova u 2D prostoru i listu trokut mreže objekta. Ta funkcija prikazana je u nastavku:

```
    bool TryGetMeshData(RaycastHit hit, out Vector3[] vertices, out Vector2[]
uvs, out int[] triangles)
    {
        Mesh mesh = null;
        vertices = null;
        uvs = null;
        triangles = null;
```

```
        if (hit.collider is MeshCollider mc && mc.sharedMesh != null)
        {
            mesh = mc.sharedMesh;
        }
        else
        {
            SkinnedMeshRenderer smr =
hit.collider.GetComponent<SkinnedMeshRenderer>();
            if (smr != null)
            {
                if (bakedMesh == null) bakedMesh = new Mesh();
                smr.BakeMesh(bakedMesh);
                mesh = bakedMesh;
            }
        }

        if (mesh != null)
        {
            vertices = mesh.vertices;
            uvs = mesh.uv;
            triangles = mesh.triangles;
            return true;
        }
        return false;
}
```

Kôd 2.9 – *TryGetMeshData* funkcija

TryGetMeshData dohvaća *MeshRenderer* objekt s našeg pogođenog objekta i iz njega čita podatke o vrhovima i vraća ih na izlaz. U ovoj funkciji je bilo potrebno pripaziti da objekt može imati obične *Mesh Renderer* koji se nalazi na statičnim objektima, dok *Skinned Mesh Renderer* imaju objekti koji se mogu animirati. U statičnim objektima teksture su statično pohranjene, dok se kod animiranih objekata geometrija deformira animacijom, stoga se ne pohranjuje statično geometrija objekta već ju je potrebno „uslikati“ u danom *frameu*, što radi *smr.BakeMesh(bakedMesh)* funkcija.

Vratimo se sada natrag na *TryGetUV* funkciju, ona je sada dobila potrebne informacije o vrhovima i trokutima te na temelju njih izračunamo s pomoću baricentričnih koordinata točno koji piksel teksture je pogođen zrakom.

Sa dobivenim koordinatama ulazimo u sljedeći dio programa:

```
            uvPreviewPanel.SetTarget(hit.collider.GetComponent<Renderer>());
            if (lastDrawUV.HasValue)
            {
                Vector2 from = lastDrawUV.Value;
                Vector2 to = uv;
                float distanceUV = Vector2.Distance(from, to);
                float scale = Mathf.Max(brushSize, 0.001f);
                int steps = Mathf.CeilToInt(distanceUV / scale * 5f);
```

```
            steps = Mathf.Clamp(steps, 1, 50);

            for (int i = 0; i <= steps; i++)
            {
                float t = i / (float)steps;
                Vector2 lerpedUV = Vector2.Lerp(from, to, t);
                DrawAtUV(temp, lerpedUV, hit, brushSize, brushColor);
            }
        }
        else
        {
        DrawAtUV(temp, uv, hit, brushSize, brushColor);
        }
```

Kôd 2.10 – Isječak kôda za oslikavanje

Postavljamo *uvPreviewPanel* za naš prikaz teksture na 2D plohi koji smo spomenuli u prethodnom poglavlju. Provjerimo jesmo li u prethodnom *frameu* nešto iscrtali, ako nismo samo ćemo pozvati *DrawAtUV* funkciju koja će nacrtati krug na dobivenoj točki. Ako jesmo nešto iscrtali u prethodnom *frameu* onda moramo provesti interpolaciju između prethodne i trenutačne točke, pa ćemo pozvati *DrawAtUV* funkciju za 1 do 50 točaka između, ovisno o debljini markera. Ako je debljina mala željet ćemo više točaka pa će broj koraka narasti, a ako je debljina dovoljno velika onda nam nije potrebno toliko točaka pa će se broj koraka smanjiti. Za svaku točku koju želimo pobojati, pozivamo *DrawAtUV* koja će pobojiti tu točku i točke oko nje u obliku kruga.

DrawAtUV funkcija je nešto veća pa ćemo samo prikazati najbitniji dio:

//dohvaćanje indeks trokuta, izračun duljina stranica, dohvaćanje UV koordinata svakog vrha svakog trokuta

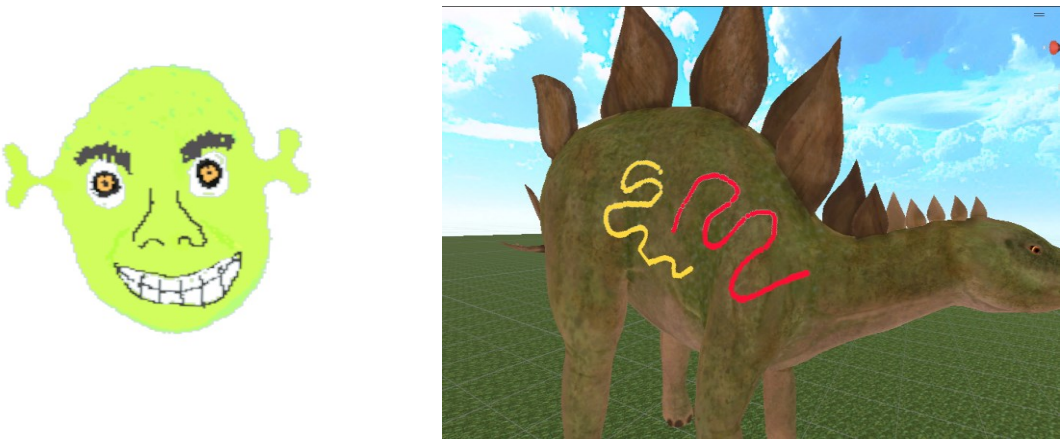
```
float uvEdge0 = Vector2.Distance(uv0, uv1);
float uvEdge1 = Vector2.Distance(uv1, uv2);
float uvEdge2 = Vector2.Distance(uv2, uv0);
float avgUVEdgeLength = (uvEdge0 + uvEdge1 + uvEdge2) / 3f;

float worldToUVScale = avgUVEdgeLength / avgWorldEdgeLength;
float uvRadius = worldRadius * worldToUVScale;

int pixelRadius = Mathf.RoundToInt(uvRadius * texWidth);
int centerX = Mathf.RoundToInt(uv.x * texWidth);
int centerY = Mathf.RoundToInt(uv.y * texHeight);

for (int y = -pixelRadius; y <= pixelRadius; y++)
{
    for (int x = -pixelRadius; x <= pixelRadius; x++)
    {
```

```
                int px = centerX + x;
                int py = centerY + y;

                if (px >= 0 && px < texWidth && py >= 0 && py < texHeight)
                {
                    float dist = Mathf.Sqrt(x * x + y * y);
                    if (dist <= pixelRadius)
                    {
                        tex.SetPixel(px, py, color);
                    }
                }
            }
        }
```

Kôd 2.11 – Isječak kôda funkcije *DrawAtUV*

U ovoj funkciji izračunavamo duljine stranica trokuta u teksturi i prostoru, da bi dobili aproksimaciju prosječne duljine stranice u 3D svijetu i 2D prostoru teksture. To koristimo da bi umanjili, odnosno uvećali radijus kruga koji ćemo obojiti. Ovaj postupak radimo kako bi zadržali konzistentnu debljinu markera na svim objektima. Naime, ako je tekstura razvučena preko manjeg broja većih trokuta, obojeni krug bi bio znatno veći nego na objektu koji ima više manjih trokuta razvučenih po teksturi.

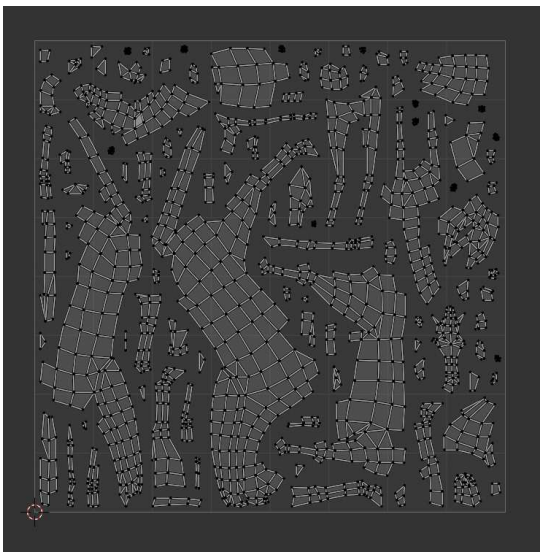
Sada izračunamo središte kruga i krećemo u dvostruku *for* petlju kojom prolazimo po svim x i y koordinatama na teksturi, koje se nalaze unutar kvadrata duljine stranice jednake radijusu kruga koji bojimo i određujemo je li ta točka upada unutar kruga. Ako upada znači da ju bojimo, ako ne upada u krug je preskočimo. Također radimo i provjeru jesmo li izašli iz prostora teksture, ako bojimo preko ruba teksture da ne dobijemo grešku.



Slika 2.4 – Rezultati crtanja na ravnoj plohi (lijevo) i na 3D objektu (desno)

Ovim postupkom ostvarili smo oslikavanje objekta temeljeno na njegovoj teksturi. No, ova metoda nije savršena i ima neke nedostatke. Jedan od tih nedostataka smo već uočili i ublažili. Radi se o nedostatku da veličina nacrtanog kruga, i njegova deformacija, ovisi o izobličenosti razvučenih trokuta na teksturi. Maloprije smo rekli da smo prilagođavali veličinu radijusa kruga ovisno o omjeru duljine stranice u 3D prostoru i 2D prostoru. Rekli smo da je razlog to što neki objekti mogu imati razmotane teksture s velikim trokutima pa će onda i krug koji crtamo biti veći. Također, ako su ti trokuti jako izobličeni, izduženi ili spljošteni i krug će isto biti izdužen ili spljošten i neće biti pravilan. Ovo je jedna od velikih mana ovog pristupa, a to je da jako ovisimo o teksturi objekta po kojem crtamo. U radu je ovaj problem djelomično riješen tako da detektiramo ako je došlo do deformacije provjerom jesu li vertikala i horizontala našeg kvadrata jednako duge, ako nisu onda ih treba ujednačiti. Međutim, rezultati nisu bili zadovoljavajući, stoga predloženo rješenje ovdje neću uključivati.

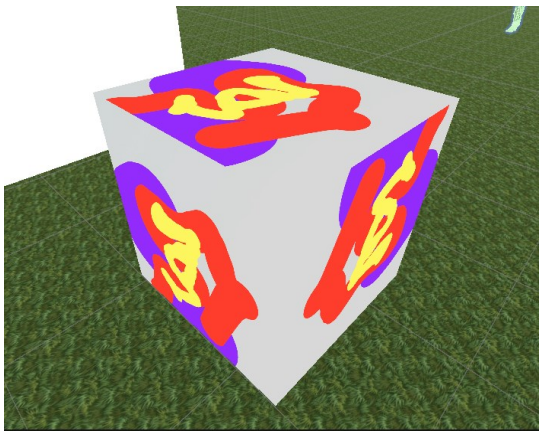
Još jedan problem javlja se na teksturama s otocima. Otoci u teksturi su razdvojene grupe trokuta na teksturi. Ovdje se javlja problem da pri crtanju po objektu prelazimo preko šava koji je granica između dvije skupine otoka. Iako su ti trokuti susjedni u 3D prostoru, oni ne moraju biti susjedni u 2D prostoru teksture, stoga dolazi do crtanja na u potpunosti drugom dijelu objekta, jer će dio kruga skripta detektirati na trokutu koji se nalazi pored ciljanoga u teksturi dok u 3D prostoru, koji vidimo, oni mogu biti jako udaljeni.



Slika 2.5 : Otoci u razmotanoj teksturi

U radu je ovaj problem također djelomično riješen skriptom koja ne dopušta prelazak piksela kruga na susjedne trokute. Ideja je bila paziti da kad prelazimo šav otoka ne crtamo na susjedni trokut. No, ovo je uvelo nove probleme. Linije su sada bile odrezane rubovima trokuta i više nisu bile neprekinute. Stoga je probano još jedno rješenje koje bi se pobrinulo za ovaj problem. Bilo je potrebno izgraditi popis svih susjednih trokuta u 3D prostoru i pratiti njihovu susjednost u prostoru teksture i onda omogućiti da krug prelazi preko šava ako je taj trokut susjedan u oba prostora. No ovo je dosta zahtjevno za izvesti, pogotovo ako imamo velik broj trokuta, a i rezultati nisu bili dobri, stoga ovu ideju samo navodim za buduće istraživače.

Postoji i problem ponavljajućih tekstura na objektima. Naime, neki objekti imaju istu teksturu razvučenu na više njihovih lica. Najbolji primjer je Unityjeva kocka, koja ima istu teksturu razvučenu na svako njezino lice. Rezultat crtanja po jednom licu je da je svako lice pobojano.



Slika 2.6: Ponavljajuće teksture na kocki

Jedno poboljšanje, koje je uspjelo, je poboljšanje izgleda samog nacrtanog kruga. U kôdu u kojem smo iscrtavali krug na temelju provjere upada li unutar radijusa kruga s Pitagorinim poučkom u ovoj liniji:

```
float dist = Mathf.Sqrt(x * x + y * y);
if (dist <= pixelRadius)
```

Iako je krug izgledao dosta dobro, mogao se poboljšati na način da umjesto da mi radimo

gornju provjeru, uzmemo novu teksturu, na kojoj je običan krug, i onda je „zalijepimo“ preko našeg kvadrata. Zatim uzorkujemo svaku točku s kvadrata kojeg bojimo i njegovu „kopiju“ na teksturi kruga i samo ju pobojamo ako je na teksturi kruga ona pobojana. Na ovaj način krug koji smo dobili je izgledao čišće jer se ne bi moglo dogoditi da neko zaokruživanje brojeva rezultira da neki piksel ipak ne upadne u radijus kruga. Slika za usporedbu prikazana je u nastavku:

Slika 2.7: Usporedba kruga crtanog "ručno" (lijevo) i crtanog s teksture (desno)

Ovime smo završili s metodom oslikavanja temeljeno na teksturi objekta. Kasnije ćemo je se ponovno dotaknuti kada ćemo dati kratki prikaz raznih metoda i njihovih prednosti i nedostataka.

2.3. Oslikavanje temeljeno na osima svijeta

U prethodnom poglavlju objasnili smo oslikavanje objekata temeljeno na njihovim teksturama. Sada ćemo pogledati malo drugačiji pristup, onaj baziran na osima svijeta. Glavni problemi prethodnog pristupa bili su nepravilne i izobličene razmotane mreže trokuta na teksturi. Ideja ovog pristupa je ukloniti ovisnost o toj razmotanoj mreži u potpunosti, što je i postignuto, ali kako ćemo ubrzo vidjeti, ovaj pristup uvodi neke nove probleme kao kompromis kod rješavanja starih.

Prvi dio postupka crtanja će ostati isti, što znači da još uvijek koristimo ispucavanje zrake iz vrha markera i tražimo 3D koordinate točke presjeka. Te koordinate ćemo opet pretvoriti u 2D koordinate koje ćemo postavljati na teksturu, ali ovaj put našu teksturu za crtanje nećemo mapirati na način na koji je mapirana originalna tekstura. Pa pogledajmo skriptu pa ćemo objasniti postupak detaljnije:

```
public class WorldSpacePainter : MonoBehaviour
{
    public Transform penTip;
    public float maxDrawDistance = 0.2f;
    public float brushWorldRadius = 0.01f;
    public Color brushColor = new Color(1, 0, 0, 0.9f);

    public int textureResolution = 1024;
    public float textureScale = 1f;

    private RenderTexture drawTexture;
    private Texture2D tempTexture;
    private Material targetMaterial;

    void Start()
    {
        drawTexture = new RenderTexture(textureResolution, textureResolution,
0, RenderTextureFormat.ARGB32);
        drawTexture.wrapMode = TextureWrapMode.Repeat;
        drawTexture.filterMode = FilterMode.Bilinear;
        drawTexture.Create();

        tempTexture = new Texture2D(textureResolution, textureResolution,
TextureFormat.RGBA32, false);
        ClearTexture(tempTexture, Color.clear);
        Graphics.Blit(tempTexture, drawTexture);

        targetMaterial = GetComponent<Renderer>().material;
        targetMaterial.SetTexture("_DrawTex", drawTexture);
        targetMaterial.SetFloat("_WorldToDrawTexScale", textureScale);
    }
```

Kôd 2.12 – *Start* funkcija skripte za oslikavanje u prostoru svijeta

Krenimo od *Start()* metode, u ovoj metodi prvo ćemo stvoriti novu *Render* teksturu (sjetimo se, podaci o ovoj teksturi nalaze se na GPU), zatim ćemo napraviti i 2D teksturu (CPU strana), 2D teksturu ćemo resetirati i takvu je poslati na GPU u našu *Render* teksturu. Ovo radimo zato da kada pokrenemo scenu počistimo teksture ako je nešto na njima bilo prethodno mijenjano. Zatim dohvaćamo materijal našeg objekta i njemu proslijedimo *Render* teksturu koju smo stvorili.

```
    void Update()
    {
        if (Physics.Raycast(penTip.position, penTip.forward, out RaycastHit
hit, maxDrawDistance))
        {
            Vector2 uv = ProjectWorldToDrawUV(hit.point, hit.normal);

            int pixelRadius = Mathf.RoundToInt(brushWorldRadius * textureScale
* textureResolution);

            RenderTexture.active = drawTexture;
            tempTexture.ReadPixels(new Rect(0, 0, textureResolution,
textureResolution), 0, 0);
            tempTexture.Apply();
            RenderTexture.active = null;

            DrawAtUV(tempTexture, uv, brushColor, pixelRadius);

            tempTexture.Apply();
            Graphics.Blit(tempTexture, drawTexture);
        }
    }
```

Kôd 2.13 – *Update* funkcija skripte za oslikavanje u prostoru svijeta

U *Update()* metodi ispucavamo zraku i pogođeni objekt šaljemo u *ProjectWorldToDrawUV* metodu:

```
    Vector2 ProjectWorldToDrawUV(Vector3 worldPos, Vector3 worldNormal)
    {
        Vector3 absNormal = new Vector3(Mathf.Abs(worldNormal.x),
Mathf.Abs(worldNormal.y), Mathf.Abs(worldNormal.z));
        Vector2 uv;

        if (absNormal.x >= absNormal.y && absNormal.x >= absNormal.z)
            uv = new Vector2(worldPos.y, worldPos.z);
        else if (absNormal.y >= absNormal.x && absNormal.y >= absNormal.z)
            uv = new Vector2(worldPos.z, worldPos.x);
        else
            uv = new Vector2(worldPos.x, worldPos.y);

        uv *= textureScale;
```

```
        uv.x = Mathf.Repeat(uv.x, 1f);
        uv.y = Mathf.Repeat(uv.y, 1f);

        uv *= textureResolution;
        return uv;
    }
```

Kôd 2.14 – *ProjectWorldToDrawUV* funkcija skripte za oslikavanje

U ovoj metodi izračunamo normalu u pogođenoj točki te na temelju nje određujemo ravninu (XY, XZ, YZ) na koju je normala najokomitija. Tu ravninu ćemo uzeti kao dominantnu i nju vratiti na izlaz metode. Sada smo se vratili u *Update()* metodu gdje izračunavamo radijus kruga, pripremamo kopiramo podatke s *Render* teksture na 2D teksturu na CPU i pozivamo metodu *DrawAtUV()*:

```
    void DrawAtUV(Texture2D tex, Vector2 uv, Color color, int radius)
    {
        int cx = Mathf.RoundToInt(uv.x);
        int cy = Mathf.RoundToInt(uv.y);

        for (int y = -radius; y <= radius; y++)
        {
            for (int x = -radius; x <= radius; x++)
            {
                int px = cx + x;
                int py = cy + y;

                if (px >= 0 && px < tex.width && py >= 0 && py < tex.height)
                {
                    float dist = Mathf.Sqrt(x * x + y * y);
                    if (dist <= radius)
                    {
                        Color existing = tex.GetPixel(px, py);
                        Color final = Color.Lerp(existing, color, color.a);
                        tex.SetPixel(px, py, final);
                    }
                }
            }
        }
    }
```

Kôd 2.15 – *DrawAtUV* funkcija skripte za oslikavanje

DrawAtUV() metoda radi isti postupak kao i prije, boja piksele unutar kruga.

Ova skripta imitira triplanarnu projekciju teksture na ravnine svijeta, ali kako bi to uspjela potreban joj je i poseban sjenčar fragmenata, koji je prikazan u nastavku. je ključan u ovom pristupu jer bez njega ne bi bilo moguće prikazati *Render* teksturu u odnosu na

koordinate objekta u sustavu svijeta. Radi jednostavnosti prikazat ćemo samo sjenčar fragmenata umjesto cijelog programa, jer su dijelovi kôda vrlo slični onima iz prethodnog primjera :

```
fixed4 frag(v2f i) : SV_Target
{
	float3 normal = normalize(i.worldNormal);
	float3 absNormal = abs(normal);
	float2 uv;

	if (absNormal.x >= absNormal.y && absNormal.x >= absNormal.z)
		uv = i.worldPos.yz * _WorldToDrawTexScale;
	else if (absNormal.y >= absNormal.x && absNormal.y >= absNormal.z)
		uv = i.worldPos.zx * _WorldToDrawTexScale;
	else
		uv = i.worldPos.xy * _WorldToDrawTexScale;

	uv = frac(uv);

	fixed4 drawColor = tex2D(_DrawTex, uv);
	fixed4 baseColor = tex2D(_MainTex, i.uv);
	return lerp(baseColor, _BrushColor, drawColor.a);
}
```

Kôd 2.16 – Sjenčar fragmenata za oslikavanje u prostoru svijeta

Ovaj sjenčar fragmenata radi istu stvar kao i skripta. Dobiva normalu od sjenčara vrhova, i računa dominantnu ravninu, na kojoj će se prikazati tekstura te „lijepi“ našu teksturu za oslikavanje preko originalne na toj ravnini. U kombinaciji sa skriptom dobili smo oslikavanje predmeta bez korištenja njegove originalne teksture. Drugim riječima, dobili smo iluziju oslikavanje teksture, bez korištenja originalne teksture i UV mape. Analogija ovom postupku bila bi kao da smo uzeli kist i nacrtali nešto na prozirno platno (ovaj dio obavlja skripta) zatim smo uzeli projektor i osvijetlili naš objekt, a platno stavili između

projektora i objekta (ovo radi sjenčar). Na objektu vidimo sjenu onoga što je na platnu nacrtano.

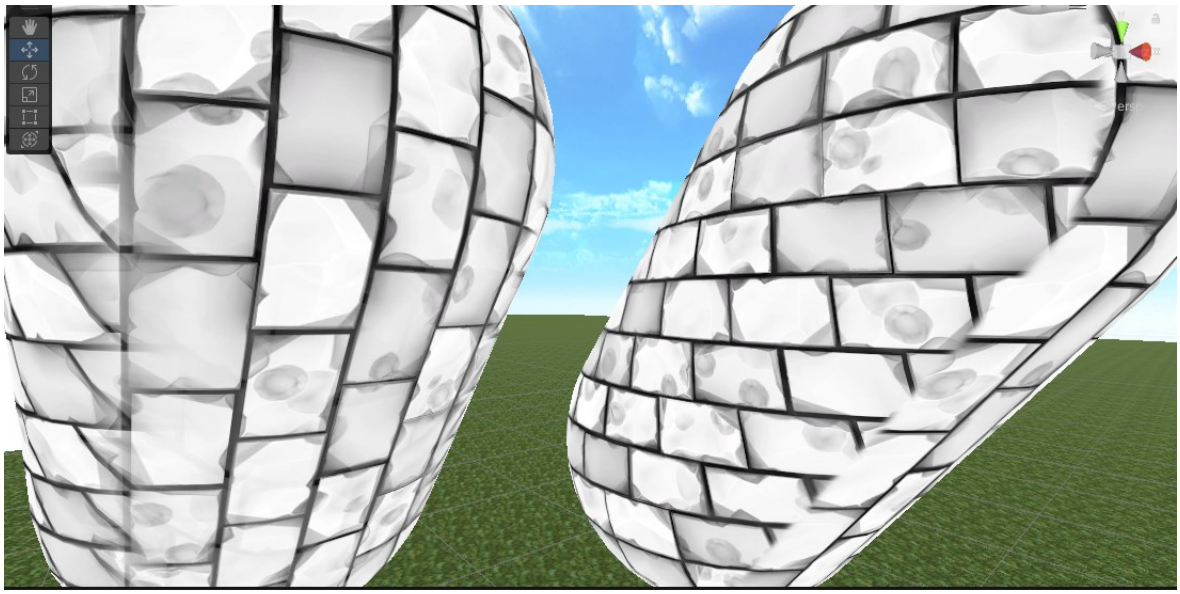
Prednost ovog postupka nad pristupom gdje smo koristili teksturu objekta i njegovu mrežu je ta što nikako ne ovisimo o toj istoj teksturi, što znači da ona može biti nepravilna, loše razmotana, a nama to ne radi probleme.

Nedostataka ovog postupka ima nažalost mnogo. Prvi je taj da na neravnim objektima projekcija izgleda loše, isprekidano na prijelazima iz jedne ravnine u drugu. Stoga ovaj postupak je primjenjiv najbolje na ravnim objektima koji su poravnati s osima svijeta, kao zidovi i podovi u sceni. Kako bi ovaj problem bi manje uočljiv napravljen je još jedan malo poboljšani sjenčar fragmenata, koji stapa linije na tim prijelazima. Kôd za taj sjenčar prikazan je niže:

```
fixed4 frag(v2f i) : SV_Target
{
	float3 absNormal = abs(i.worldNormal);
	float3 blendWeights = pow(absNormal, _BlendSharpness);
	blendWeights /= dot(blendWeights, 1.0);
	float2 uvX = i.worldPos.yz * _TextureScale;
	float2 uvY = i.worldPos.zx * _TextureScale;
	float2 uvZ = i.worldPos.xy * _TextureScale;
	fixed4 colX = tex2D(_DrawTex, uvX);
	fixed4 colY = tex2D(_DrawTex, uvY);
	fixed4 colZ = tex2D(_DrawTex, uvZ);
	fixed4 drawColor = colX * blendWeights.x + colY * blendWeights.y + colZ *
	blendWeights.z;
	colX = tex2D(_MainTex, uvX);
	colY = tex2D(_MainTex, uvY);
	colZ = tex2D(_MainTex, uvZ);
	fixed4 baseColor = colX * blendWeights.x + colY * blendWeights.y + colZ *
	blendWeights.z;
	fixed4 result = lerp(baseColor, drawColor * _BrushColor, drawColor.a);
	result.a = 1.0;
	return result;}
```

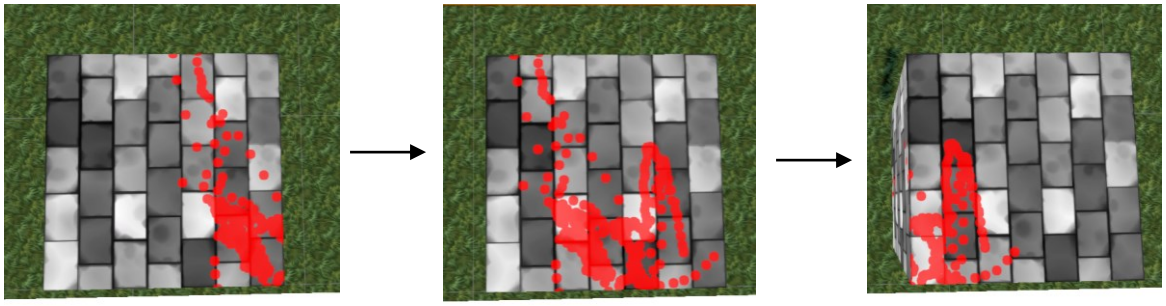
Kôd 2.17 – Sjenčar fragmenata sa stapanjem

U gornjem sjenčaru uzimamo doprinose svake komponente normale (x, y, z) i na temelju njih izračunavamo konačnu boju. Na ovaj način dobili smo mekše prijelaze između ravnina.



Slika 2.8: Usporedba sjenčara sa stapanjem i bez stapanja

Drugi, i možda najveći problem ovog pristupa je što ne možemo pohraniti promjene na sâm objekt, jer nismo koristili njegovu originalnu UV mapu pa ne znamo mapirati točke teksture za crtanje na mrežu objekta. Imamo samo projekciju teksture na os, i ta projekcija je statična, što znači da rotacijom objekta i pomicanjem objekta, oslikano područje ga neće „pratiti“, kao što smo rekli kao da smo obasjali objekt svjetlom, svjetlo je statično, i ako pomaknemo objekt svjetlo će još uvijek obasjavati istu točku u prostoru. Ovaj pristup je osjetljiv na skaliranje i pomicanje objekta jer veza između 3D pozicije i 2D teksture nije 'ugrađena' nego se računa iz koordinata svijeta.



Slika 2.9: Pomicanjem kocke u prostoru ne pomiče se boja nego ostaje statična

Ovime završavamo ovo poglavlje i krećemo u kratak pregled još nekih metoda oslikavanja.

2.4. Ostale metode

Proučili smo dvije bitne metode za oslikavanje predmeta u sceni, sada ćemo spomenuti još neke, koje u ovom radu nisu implementirane, ali se koriste.

2.4.1. Projekcija 3D geometrije

Ova metoda uključuje dodavanje plosnatih geometrijskih objekata, kao kvadrata, na površinu objekta na mjestu pogotka zrake. Zatim se sjenčarima postiže nevidljiv prijelaz između tog kvadrata i objekta po kojem crtamo. Prednosti ovog postupka su što može dati vrlo kvalitetna vizualne rezultate, i kako ni u jednom koraku ne koristimo teksture, nemamo problema s nepravilnim teksturama kao što smo imali u prvom načinu. Međutim, nedostatak ovog pristupa je to što je skupo postavljati geometriju, pogotovo ako je postavimo toliko da počne narušavati performanse. Još jedan problem koji se javlja je *„z-fighting“,* što znači da može doći do problem da pri iscrtavanju, jer se kvadrati nalaze na otprilike istoj udaljenosti od kamere kao i površina objekta, pa program ne zna koji kvadrat treba iscrtati, onaj od objekta ili onaj koji smo dodali. Također, kako ovim pristupom nismo mijenjali teksturu, ne možemo pohraniti promjene.

2.4.2. Oslikavanje u prostoru ekrana

Ovaj pristup koristi crtanje u koordinatama ekrana, koje se onda projiciraju na svijet s pomoću ispucavanja zraka ili *depth-buffera*. Ova metoda podobna je za oslikavanje iz pogleda kamere, ali nije dobra za oslikavanje 3D prostora jer nije dosljedna u 3D prostoru.

2.4.3. Oslikavanje po vrhovima

Metoda uključuje bojanje vrhova, nakon čega interpoliramo boju trokuta na temelju boja u vrhovima. Ova metoda je vrlo brza i efikasna i nema problema s teksturama, ali se rezultati vrlo niske rezolucije ako mreža trokuta nije dovoljno gusta, stoga nije prigodna za bilo kakvo detaljnije oslikavanje.

2.5. Pregled prednosti i nedostataka

Spomenuli smo i objasnili neke postupke korištene u tehnikama oslikavanja predmeta u prostoru, sada ćemo ih sažeti u tablicu s njihovim prednostima i nedostatcima kako bi imali još jedan sažet pregled svake od ovih metoda:

Metoda	**Prednosti**	**Nedostatci**
Oslikavanje temeljeno na teksturi	• Precizno • Ostaje zapisano na teksturi • Neovisno o položaju i rotaciji objekta	• Vrlo ovisno o dobro razmotanoj mreži • Otoci i šavovi mogu proizvesti artefakte
Oslikavanje temeljeno na osima svijeta	• Neovisno o razmotanoj mreži objekta	• Ne ostavlja zapis na teksturi, nema lakog načina za pohranu promjene • Ovisi o položaju i rotaciji objekta • Zakrivljeni objekti nisu pogodni za ovu metodu
Projekcija 3D geometrije	• Daje rezultate visoke kvalitete • Ne ovisi o razmotanoj mreži objekta	• Teško pohraniti promjene • Skupo i može narušiti performanse • *Z-fighting*
Oslikavanje u prostoru ekrana	• Korisno za slikanje iz pogleda kamere	• Nema dosljednost u prostoru
Oslikavanje po vrhovima	• Vrlo brzo • Nema potrebe za teksturom	• Loša kvaliteta ako mreže nisi iznimno guste

Zaključak

Oslikavanje predmeta u 3D virtualnom svijetu novo je područje istraživanja. Vrlo je zanimljivo područje jer bi njegove primjene mogle biti brojne (medicina, videoigre, obrazovanje, …).

U ovom radu prikazane su neke metode oslikavanja predmeta u virtualnoj stvarnosti. Implementirane su dvije metode i detaljno analizirani koraci u njihovom ostvarenju. Za sve spomenute metode prikazane su njihove prednosti i nedostatci. Za metodu oslikavanja teme na teksturi, rekli smo da je jedna od glavnih metoda oslikavanja objekta, no njezina je mana prevelika ovisnost o mreži objekta.

Metoda oslikavanja temeljeno na osima svijeta bila je sljedeći korak, jer je neovisna o mreži objekta i teksturi, no ograničena je na ravne površine i predmete poravnate s osima svijeta. Također, nije pogodna za pohranu promjena.

Projekcija 3D geometrije je metoda koja može dati rezultate visoke kvalitete, ali je skupa i potrebno je pripaziti u implementaciji kako ne bi došlo do *z-fightinga*. Oslikavanje u prostoru ekrana nije previše korisno u oslikavanju 3D prostora, već više za oslikavanje ekrana koji vidimo kroz kameru. A oslikavanje po vrhovima, iako je vrlo brzo i efikasno daje rezultate niske kvalitete, ako mreže nisu dovoljno guste.

Literatura

[1] OpenXR. *Khronos OpenXR Registry.* Poveznica Khronos OpenXR Registry - The Khronos Group Inc.

[2] Unity. Unity Shaders. Poveznica: Unity - Manual: Creating shaders with Shader Graph

[3] Agrawala,M.,Beers,A.C.,Levoy,M.,*3D Painting on Scanned Surfaces*, *Computer Science Department*, *Stanford University*

[4] Klein, V., *Projection-Mapping-Based 3D Painting, Friedrich-Alexander-Universitaet Erlangen-Nuernberg*

Sažetak

Interaktivno oslikavanje objekata u virtualnom okruženju

Interakcija korisnika s objektima u VR u svrhu oslikavanja objekata novo je i zanimljivo područje istraživanja. Područja primjene su široka. Postoje brojne metode kojima se ova interakcija postiže, svake sa svojim prednostima i nedostacima. Ovaj rad daje pregled nekih metoda i detaljna pojašnjenje i korake u njihovom ostvarenju. Za određene metode napravljena je implementacija u programskom pogonu *Unity Engine*, i korišten je VR sustav *HTC Vive.*

Ključne riječi: oslikavanje objekata, virtualna stvarnost, sjenčari, teksture

Summary

Interactive painting of objects in virtual reality

User interaction with objects in VR for the purpose of painting is a new and interesting area of research. The fields of application are broad. There are numerous methods by which this interaction can be achieved, each with its own advantages and disadvantages. This paper provides an overview of some of these methods, along with detailed explanations and implementation steps. For certain methods, implementations were created using the Unity Engine, and the HTC Vive VR system was used.

Keywords: object painting, virtual reality, shaders, textures

Privitak

Kôdu sa implementacijom moguće je pristupiti na Github repozitoriju: https://github.com/Yami0305/diplomski-rad