STARU IZSEKOŠANAS ALGORITMA PIELIETOJUMS 3D AINAS RENDERĒŠANAI
RAY TRACING ALGORITHM FOR 3D SCENE RENDERING

Author: **Valdis TĀRAUDS**, e-mail: valdis.tarauds@gmail.com
Scientific supervisors: **doc, Mg.sc.ing. Ivars MEIRĀNS,** e-mail: ivars.meirans@rta.lv
Rēzeknes Tehnoloģiju Akadēmija
Atbrīvošanas aleja 115, Rēzekne, Latvija

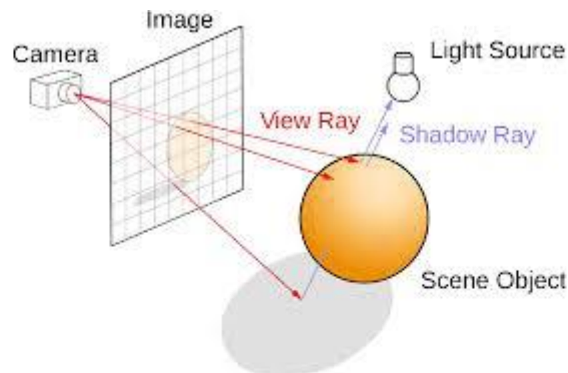
Abstract. *In this work, authors implemented simple gpu ray tracing algorithm, that can render image with realistic reflections. To achieve result author use Unity as main development tool.*

Keywords: *Ray, Ray Tracing, GPU, Compute Shader.*

Ievads

Ray tracing ir algoritms priekš trīs dimensiju grafikas renderēšanas ar ļoti sarežģītu gaismas mijiedarbību. Ar šī algoritma palīdzību var iegūt attēlu, kas pilns ar spoguļiem, caurspīdīgām virsmām un ēnām.

Ray tracing algoritma ideja ir ļoti vienkārša. Kā redzams 1. attēlā, nepieciešama kamera no kuras centra tiks izšauti stari, un attēls (*Image*) uz kura tiks renderēts 3D skats. Katrs stars, kas iziet no kameras centra, iet cauri katram attēla pikselim, un turpina savu kustību taisnā virzienā, kamēr tas trāpa kādam no objektiem scēnā. Kad stars trāpa kādam no objektiem, tad tiek iegūts trāpījuma punkts un krāsa šajā punktā. Bet ja veidot atstarojošas virsmas, tad stars no trāpījuma punkta turpina kustību atstarošanas virzienā. Un pikselis iegūst krāsu apvienojot visu trāpījuma punktu rezultātus.



1. attēls. *Ray trace* algoritma ideja.

Šis algoritms tiek izmantots filmu un seriālu vizuāli reālistisku, vizuālo efektu izveidei.

Šī pētījuma **mērķis** ir implementēt staru izsekošanas algoritmu, lai iegūt attēlu ar spoguļa virsmām.

Materiāli un metodes
Tehnoloģijas

Unity3D – spēļu izstrādes dzinis.

Visual Studio 2019 – integrēta izstrādes vide no *Microsof*.

C# (C-Sharp) – universāla, daudz paradigmu programmēšanas valoda.

HLSL (High-Level Shading Language) – programmēšanas valoda priekš video karšu instrukciju rakstīšanas.

https://doi.org/10.17770/het2020.24.6758

Izstrādes gaita

Izveidojot jaunu *Unity3D* projektu, pirmais solis boja izveidot klasi, kas būs sākuma punkts. Kurā notiek visu nepieciešamo datu inicializācija un metožu izsaukums. Metodē *InitRenderTexture,* kas redzama 2 .attēlā, tiek izveidota virsma uz kuras tiks renderēta 3D scēna.

```
using UnityEngine;
0 references
public class RayTracingMaster : MonoBehaviour
{
    public ComputeShader RayTracingShader;
    private Camera _camera;

    public RenderTexture _target;

    0 references
    private void Awake()[...]

    0 references
    private void OnRenderImage(RenderTexture source, RenderTexture destination)[...]

    1 reference
    private void Render(RenderTexture destination)[...]

    1 reference
    private void InitRenderTexture()
    {
        if (_target == null || _target.width != Screen.width || _target.height != Screen.height)
        {
            // Release render texture if we already have one
            if (_target != null)
                _target.Release();
            // Get a render target for Ray Tracing
            _target = new RenderTexture(Screen.width, Screen.height, 0,
                RenderTextureFormat.ARGBFloat, RenderTextureReadWrite.Linear);
            _target.enableRandomWrite = true;
            _target.Create();
        }
    }

    1 reference
    private void SetShaderParameters()[...]
}
```

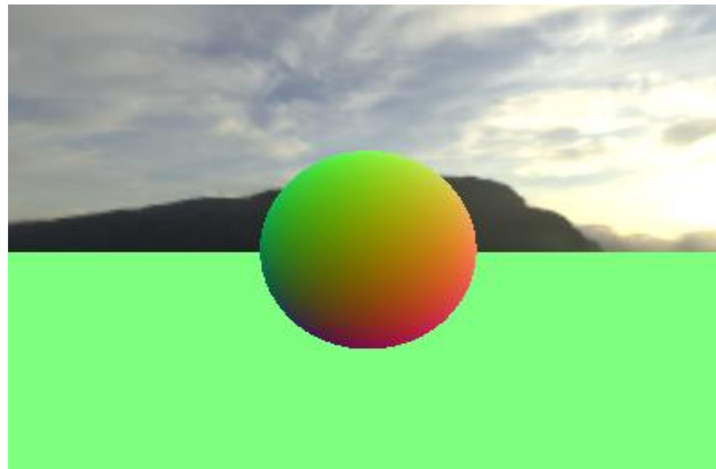
2. attēls. *C#* kods

Kā arī bija nepieciešams izveidot *Compute Shader*, lai visus nepieciešamos aprēķinus veikt izmantojot video karti (*GPU*). *Compute Shader* ir programma, kas izpildās videokartē. Šis *Compute Shader* instrukcijas, tiek rakstītas *HLSL* valodā. 3. attēlā redzamajā kodā, tiek izveidoti stari (*Rays*), no kameras centra caur attēla, virsmas uz kuras tiks renderēta 3D scēna, katru pikseli.

```
#pragma kernel CSMain

RWTexture2D<float4> Result;
float4x4 _CameraToWorld;
float4x4 _CameraInverseProjection;

struct Ray{ ... };
Ray CreateRay(float3 origin, float3 direction){ ... }

Ray CreateCameraRay(float2 uv)
{
    // Transform the camera origin to world space
    float3 origin = mul(_CameraToWorld, float4(0.0f, 0.0f, 0.0f, 1.0f)).xyz;

    // Invert the perspective projection of the view-space position
    float3 direction = mul(_CameraInverseProjection, float4(uv, 0.0f, 1.0f)).xyz;
    // Transform the direction from camera to world space and normalize
    direction = mul(_CameraToWorld, float4(direction, 0.0f)).xyz;
    direction = normalize(direction);
    return CreateRay(origin, direction);
}

[numthreads(8, 8, 1)]
void CSMain(uint3 id : SV_DispatchThreadID)
{
    // Get the dimensions of the RenderTexture
    uint width, height;
    Result.GetDimensions(width, height);
    // Transform pixel to [-1,1] range
    float2 uv = float2((id.xy + float2(0.5f, 0.5f)) / float2(width, height) * 2.0f - 1.0f);
    // Get a ray for the UVs
    Ray ray = CreateCameraRay(uv);
    // Write some colors
    Result[id.xy] = float4(ray.direction * 0.5f + 0.5f, 1.0f);
}
```

3. attēls. *Compute Shader* kods.

Nākamais solis bija izveidot grīdas virsmu un sfēru ar kuriem stari varēs mijiedarboties. Papildinot esošo kodu tika iegūts rezultāts, kas redzams 4. attēlā. Grīda ir bezgalīgi gara un plata.

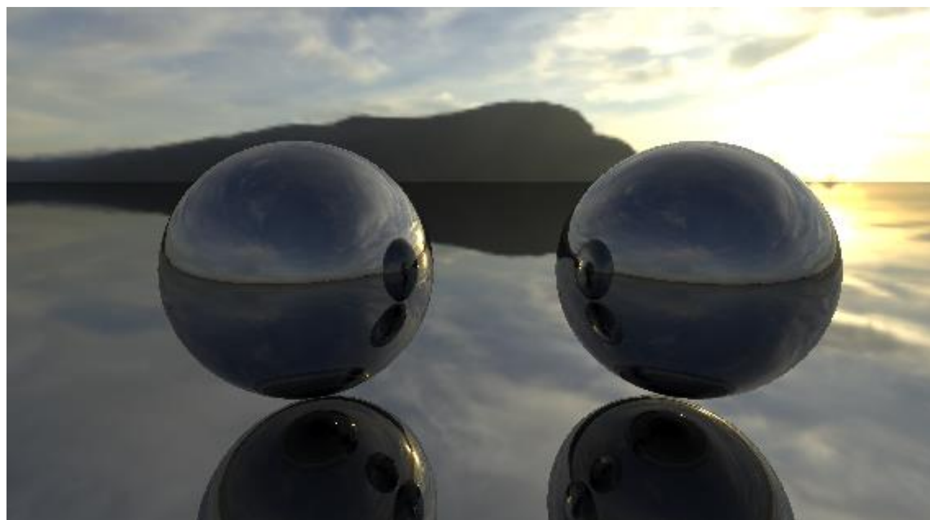


4. attēls. *Ray Tracing* rezultāts

Tagad ar staru palīdzību ir iespējams renderēt 3D objektus uz izveidotās virsmas. Nākošais solis ir izveidot staru atstarošanu, lai iegūt spoguļa virsmu. Ideja ir ļoti vienkārša, katru reizi kad stars trāpa uz virsmas, tas tiek atstarots, saskaņā ar gaismas atstarošanas likumu (gaismas krītošais leņķis = gaismas atstarošanas leņķis), kā arī tiek samazināta stara enerģija. Tas tiek atkārtots kamēr staram ir pietiekoši daudz enerģijas (noteikts atstarošanās skait), vai kamēr stars netrāpa debesīs. Visi atstarošanās rezultāti tiek apvienoti un tādējādi tiek iegūta pikseļa krāsa uz attēla virsmas.

Rezultāts

Kā redzams 5. attēlā. rezultāts ir iespaidīgs. Tika iegūta perfekta spoguļa virsma izmantojot *Ray trace* algoritmu. Attēls zemāk izskatās diezgan reālistisks, kaut arī vēl nav implementētas ēnas un gaismas.



5. attēls. *Ray Tracing* rezultāts ar atstarojošām virsmām

Secinājumi

Šajā darbā tika veikts neliels ieskats *Ray tracing* algoritmā un tika izveidots piemērs, kas spēj renderēt attēlus ar reālistiskām spoguļu virsmām. Izveidotais piemērs var renderēt tikai sfēriskus objektus un bezgalīgu grīdas virsmu. Lai varētu renderēt sarežģītākas formas objektus, nepieciešams veikt papildus pētījumus.

Izveidoto piemēru ir plānots papildināt ar ēnām un caurspīdīgiem objektiem.

Summary

In this work, a small insight into the Ray tracing algorithm was taken and an example was created that is able to render images with realistic mirror surfaces. The created example can only render spherical objects and an infinite floor surface. In order to be able to render more complex shaped objects, it is necessary to perform additional research.

It is planned to add shadows and transparent objects to the created example.

Literatūra

[1] ievads staru izsekošanas algoritmā: vienkārša metode 3D attēlu izveidošanai, [atsauce uz 17.04.2020.]. Pieejams: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/raytracing-algorithm-in-a-nutshell

[2] staru izsekošanas algoritma implementēšana, [atsauce uz 17.04.2020.]. Pieejams: http://blog.three-eyed-games.com/2018/05/03/gpu-ray-tracing-in-unity-part-1/

[3] staru izsekošanas algoritms, [atsauce uz 17.04.2020.]. Pieejams: https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html

[4] "*UC Davis Academics*" lekcijas video par *Ray tracing* [atsauce uz 17.04.2020.]. Pieejams: https://www.youtube.com/watch?v=Ahp6LDQnK4Y