Computer Graphics 1

Tutorial

Assignment 4

Summer Semester 2024

Ludwig-Maximilians-Universität München

Contact

If you have any questions:

cg1ss24@medien.ifi.lmu.de

Organization

- **Theoretical part:**
  - Prepares you for the exam
  - Solve the tasks at home
  - We present the solutions during the tutorial sessions
- **Practical part:**
  - We will indicate which parts you need to do at home
  - Else this will be done & explained during the tutorial session
  - Ask questions!

Revision: Understanding the Up Vector and Look Direction Vector

1. **Look Direction Vector (l-vector)**: represents the direction in which the camera is looking in world space. In the context of computer graphics, this vector helps define the orientation of the camera in the 3D space
2. **Up Vector (u-vector)**: defines what direction should be considered 'up' relative to the camera. It is crucial in defining the camera’s roll around its viewing axis (which is the direction it looks towards). The up-vector helps to keep the camera oriented properly by preventing unwanted rotations.

Revision: Camera Translation

- Task 2a asks for the translation matrix that transforms the position of the camera to the origin. This is a basic transformation in computer graphics, moving the camera to the origin, making it simpler to perform further transformations.
- The 1s along the diagonal keep the x, y, and z components of any point unchanged, apart from the translation effect.

$$T_{\text{view}} = \begin{bmatrix} 1 & 0 & 0 & -p1 \\ 0 & 1 & 0 & -p2 \\ 0 & 0 & 1 & -p3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 2 b): Camera Rotation

- This task is typical in view transformations to align the camera's orientation with the axes used for projecting the scene onto a 2D viewport.

$$\begin{bmatrix} x_{l \times u} & x_u & x_{-l} & 0 \\ y_{l \times u} & y_u & y_{-l} & 0 \\ z_{l \times u} & z_u & z_{-l} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 2 b): Camera Rotation

- Calculate the Orthonormal Basis: The goal is to create an orthonormal basis (right, up, and forward vectors) for the camera based on the target orientation
  - Forward vector: This should be the negative of the look direction vector, normalized
  - Up vector: The up vector is already given as (0, 1, 0)
  - Right vector: The right vector can be computed as the cross product of the up vector and the forward vector
- Construct the Rotation Matrix: Using the orthonormal basis vectors as the columns

Revision: Orthographic Projection

- Orthographic projection is a method where the 3D points are projected straight onto the projection plane along lines parallel to the projection direction (usually perpendicular to the projection plane)
- ignores perspective, meaning objects retain their sizes no matter how far they are from the camera
- The projection does not introduce any foreshortening(no distortion) or convergence(parallel lines stay parallel)
- useful in technical and engineering drawings where accurate measurements are required

Revision: Perspective Projection

- In contrast, perspective projection simulates the way the human eye perceives the world, where objects appear smaller as they are further away, creating a sense of depth
- involves projecting points onto a plane along lines that converge at a single point (the eye or camera position). The result is that parallel lines appear to converge in the distance

Revision: Perspective Projection

- The orthographic projection matrix T_{ortho} transforms the coordinates from the view frustum defined by [l,rl, rl,r] for left and right, [b,tb, tb,t] for bottom and top, and [f,nf, nf,n] for far and near planes to a normalized cube $[-1,1\text{-}1, 1-1,1]^3$
- transforming a view frustum to a normalized cube is an essential step for simplifying and standardizing the rendering process
- Graphics hardware is optimized to operate within this normalized space !
- The matrix can be derived by first translating the frustum to be centered at the origin and then scaling it to the normalized cube dimensions
- foundational for rendering scenes accurately based on the camera’s perspective and the scene's geometry

Revision: Transformation Matrix $T_{persp \rightarrow ortho}$

- In perspective projection, objects farther away from the camera appear smaller due to the projection lines converging at a single point. To transform into an orthographic view, where depth is flattened and the scaling due to distance is removed, we need to perform a transformation that "normalizes" the depth based on the perspective projection characteristics
- **Field of View (θ):** that dictates how wide the camera can see, influences the scale of objects in view; the wider the angle, the larger the projected space
- **Aspect Ratio (λ):** ratio of width to height of the viewing frustum at a given depth
- adjusts for the perspective frustum's widening as it moves away from the viewpoint
- normalizes coordinates in such a way that a point in the perspective frustum is scaled down as it moves away from the camera, making it fit within a standard orthographic frustum

Revision: Combined Transformation Using T_{ortho} and $T_{\text{persp}\rightarrow\text{ortho}}$

- To compute the combined transformation for perspective to a normalized orthographic cube, simply multiply the derived matrices: $T_{\text{combined}} = T_{\text{ortho}} \cdot T_{\text{persp}\rightarrow\text{ortho}}$
- Tortho is the matrix derived in part b) for transforming an orthographic view frustum into a normalized cube
- T_{combined} transforms a point from a perspective frustum directly into a normalized orthographic cube, handling both the perspective distortion and the frustum normalization in one step
- critical for 3D rendering pipelines where a perspective view needs to be accurately mapped to a 2D screen space for rendering

Task 1: Phong Illumination Model

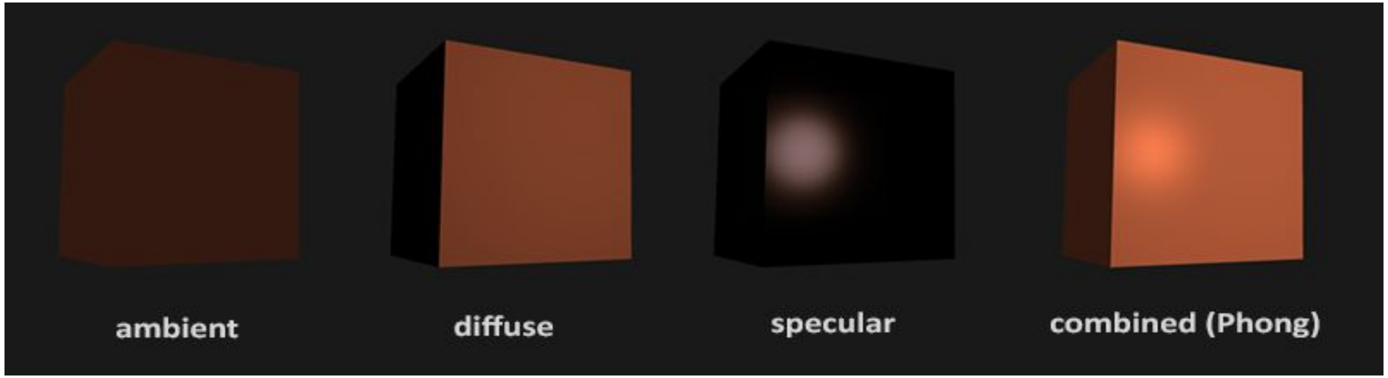
- **Explain the three components of the Phong illumination model and how they contribute to the final color of a pixel in a ray-traced image**
  - The Phong illumination model is composed of three components: ambient, diffuse, and specular lighting

Task 1: Phong Illumination Model

- **Ambient Lighting:**
  - This represents the constant amount of light that exists in a scene, even in the absence of light sources. It's the base level of lighting that all objects receive. Ambient lighting is typically represented by a constant color.
- **Diffuse Lighting:**
  - This depends on the angle between the light source and the surface normal (the direction that is "up" for a surface). The more directly the light hits the surface (i.e., the smaller the angle between the light direction and the surface normal), the brighter the diffuse lighting will be. Diffuse lighting is given by the Lambertian reflectance model, which states that the intensity is proportional to the cosine of the angle between the light direction and the normal.

Task 1: Phong Illumination Model

- **Specular Lighting:**
  - bright highlight that appears on shiny objects when the light source reflects directly into the viewer's eyes
  - intensity depends on the angle between the viewer and the reflection direction
  - modeled using the Phong reflection model, which raises the cosine of the angle between the viewer and the reflection to a high power, producing a sharp highlight



Task 1: Phong Illumination Model

- **Given a point of intersection, a light source position, and a surface normal, calculate the direction vectors required for the diffuse and specular lighting calculations. These are the light direction(diffuse) and the reflection vector(specular).**
  - Let's denote the point of intersection as P, the light source position as L, and the surface normal as N

Task 1: Phong Illumination Model

- The direction vector for the diffuse lighting calculation is simply the direction from P to L. This is calculated as follows:

  $$D = L - P$$

- Normalize D to get the unit direction vector
- The unit direction vector is required to calculate the cosine of the angle between the light direction and the normal for the Lambertian reflectance model

Task 1: Phong Illumination Model

- The direction vector for the specular lighting calculation is the reflection of D about the normal N. The reflection R of a vector V about a normal N is given by:

  $$R = V - 2(V \bullet N)N$$

- Where V • N is the dot product of V and N, and V is the incoming light direction, which is -D in this case. So the reflection R is:

  $$R = -D - 2((-D) \bullet N)N$$

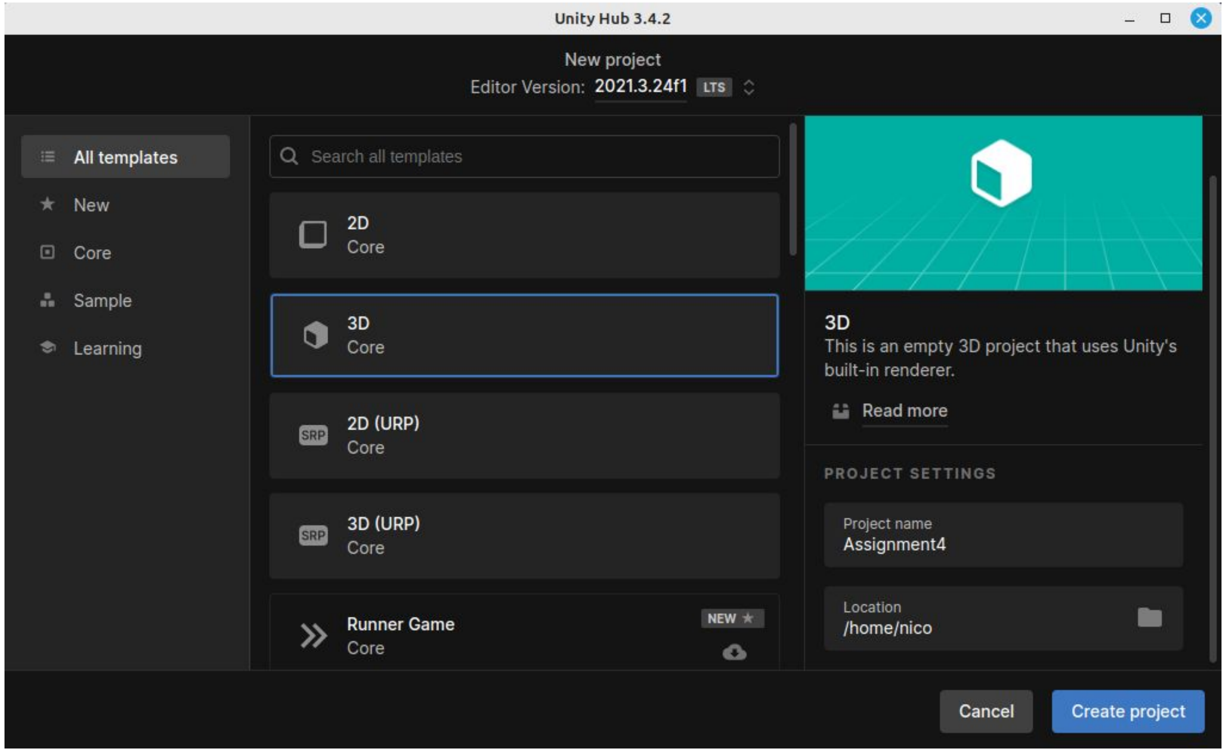
Practical Task: Ray Tracer

- Following is the link to the paper that serves as foundation for our ray tracing implementation:

https://www.cs.drexel.edu/~david/Classes/Papers/p343-whitted.pdf

Practical Part: Ray Tracer

Create a new 3D project



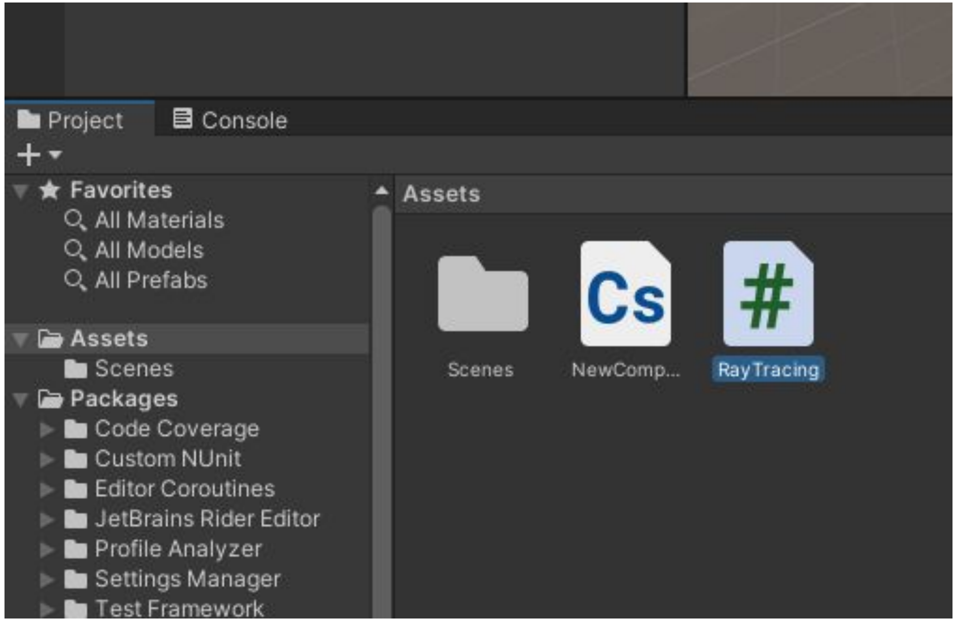
This is an empty 3D project that uses Unity's built-in renderer.

Project name: Assignment4

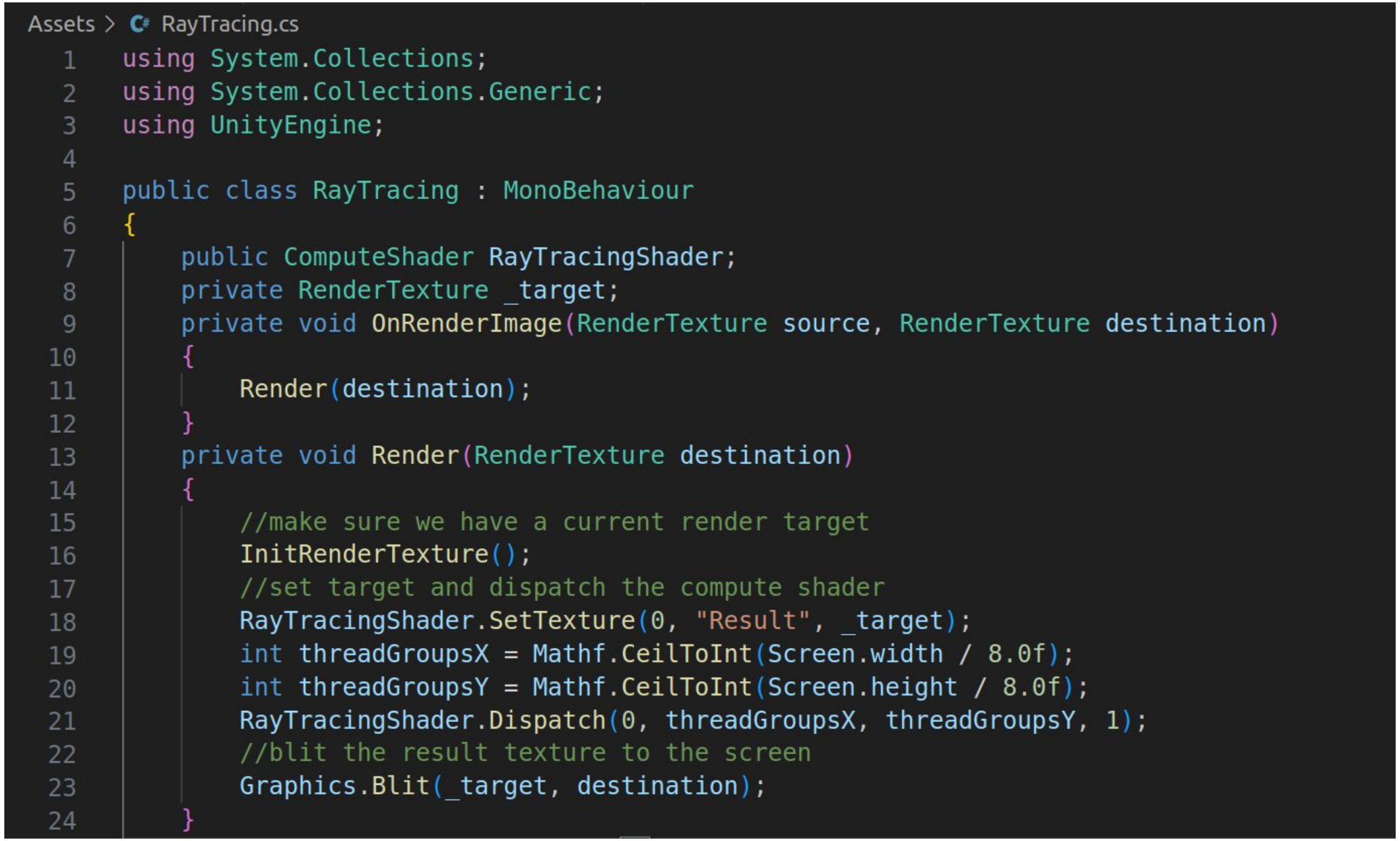
Location: /home/nico

Assets -> Create -> Shader -> Compute Shader

Create a C# script with the name RayTracing.cs and a compute shader



Fill the script with some fundamental code

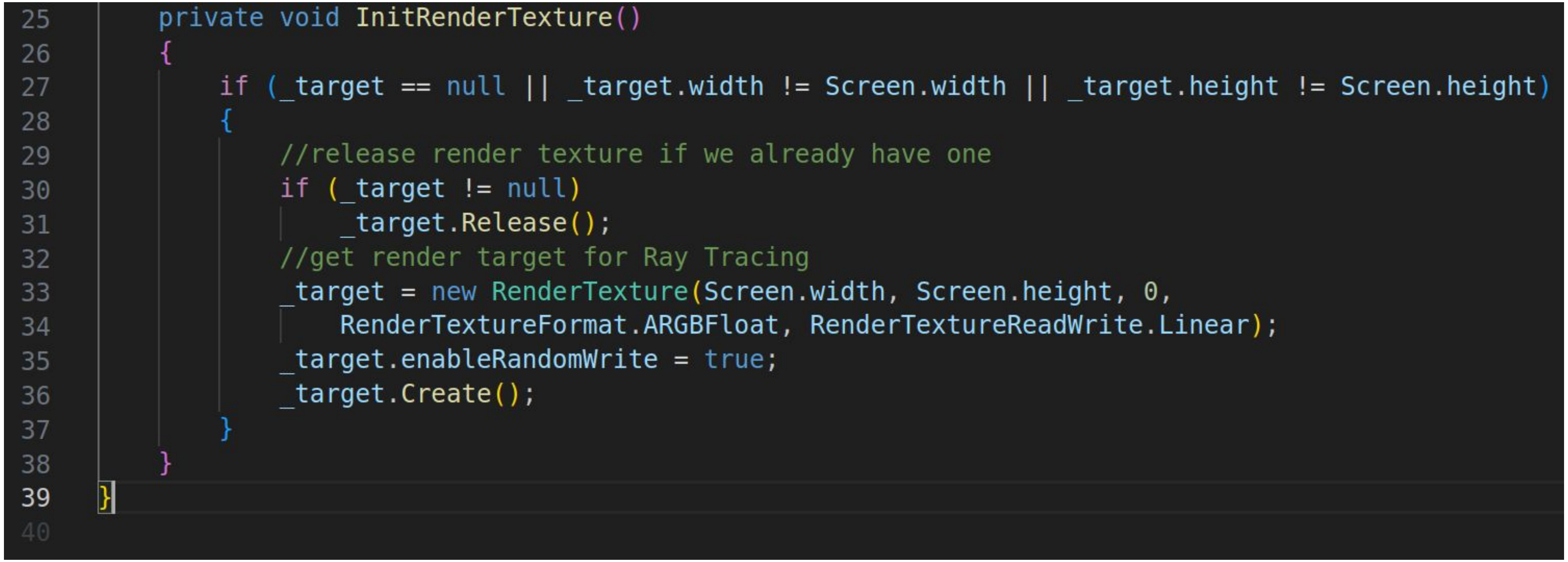


Assets > RayTracing.cs

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

public class RayTracing : MonoBehaviour
{
    public ComputeShader RayTracingShader;
    private RenderTexture _target;
    private void OnRenderImage(RenderTexture source, RenderTexture destination)
    {
        Render(destination);
    }
    private void Render(RenderTexture destination)
    {
        //make sure we have a current render target
        InitRenderTexture();
        //set target and dispatch the compute shader
        RayTracingShader.SetTexture(0, "Result", _target);
        int threadGroupsX = Mathf.CeilToInt(Screen.width / 8.0f);
        int threadGroupsY = Mathf.CeilToInt(Screen.height / 8.0f);
        RayTracingShader.Dispatch(0, threadGroupsX, threadGroupsY, 1);
        //blit the result texture to the screen
        Graphics.Blit(_target, destination);
    }
```

Fill the script with some fundamental code



```
    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 render target for Ray Tracing
            _target = new RenderTexture(Screen.width, Screen.height, 0,
                RenderTextureFormat.ARGBFloat, RenderTextureReadWrite.Linear);
            _target.enableRandomWrite = true;
            _target.Create();
        }
    }
}
```

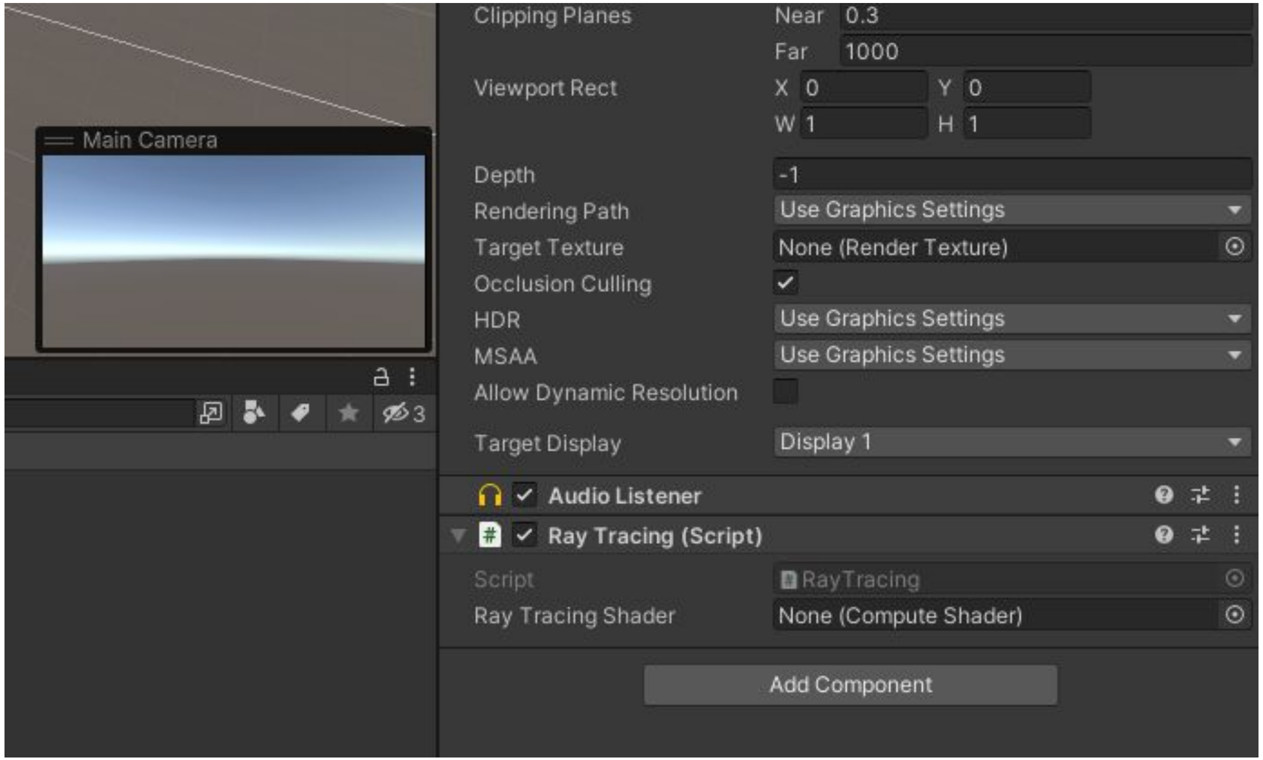
Code Explanation

- The OnRenderImage function is automatically called by Unity whenever the camera has finished rendering
- To render, we first create a render target of appropriate dimensions
- Then we tell the compute shader about it
- The 0 is the index of the compute shader’s kernel function – we have only one
- Next, we dispatch the shader: Effectively we are telling the GPU to get busy with a number of thread groups executing our shader code

Code Explanation

- Each thread group consists of a number of threads which is set in the shader itself
- The size and number of thread groups can be specified in up to three dimensions, this makes it easy to apply compute shaders to problems of either dimensionality
- In our case, we want to spawn one thread per pixel of the render target
- The default thread group size as defined in the Unity compute shader template is [numthreads(8,8,1)]
- we will stick to that and spawn one thread group per 8×8 pixels
- Then, we can finally write our result to the screen using Graphics.Blit

Open the camera in the inspector and add the new script



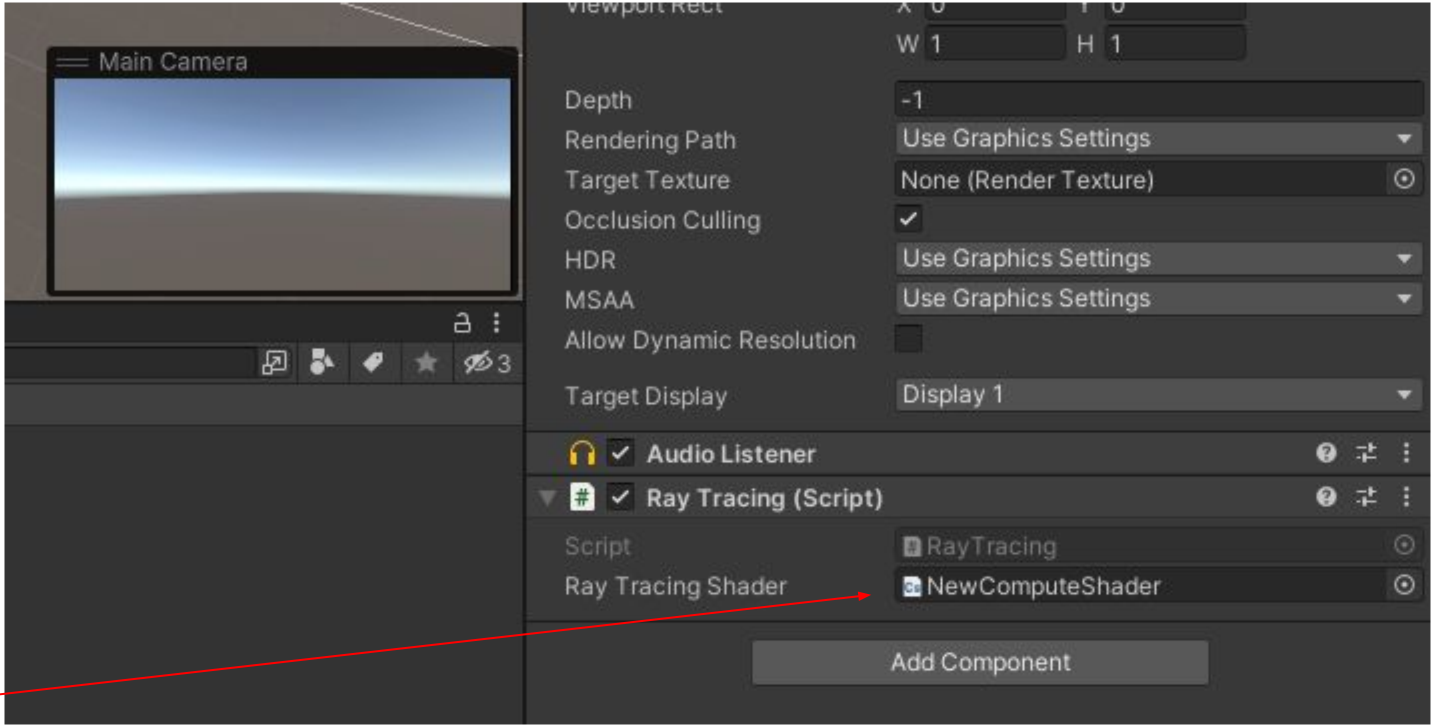
Clipping Planes	Near	0.3
	Far	1000
Viewport Rect	X 0	Y 0
	W 1	H 1
Depth	-1	
Rendering Path	Use Graphics Settings	
Target Texture	None (Render Texture)	
Occlusion Culling	✓	
HDR	Use Graphics Settings	
MSAA	Use Graphics Settings	
Allow Dynamic Resolution		
Target Display	Display 1	

✓ Audio Listener

✓ Ray Tracing (Script)

Script	RayTracing
Ray Tracing Shader	None (Compute Shader)

Open the camera in the inspector and add the new script



Viewport Rect X 0 Y 0
W 1 H 1

Depth: -1
Rendering Path: Use Graphics Settings
Target Texture: None (Render Texture)
Occlusion Culling: ✓
HDR: Use Graphics Settings
MSAA: Use Graphics Settings
Allow Dynamic Resolution:
Target Display: Display 1

Audio Listener

Ray Tracing (Script)

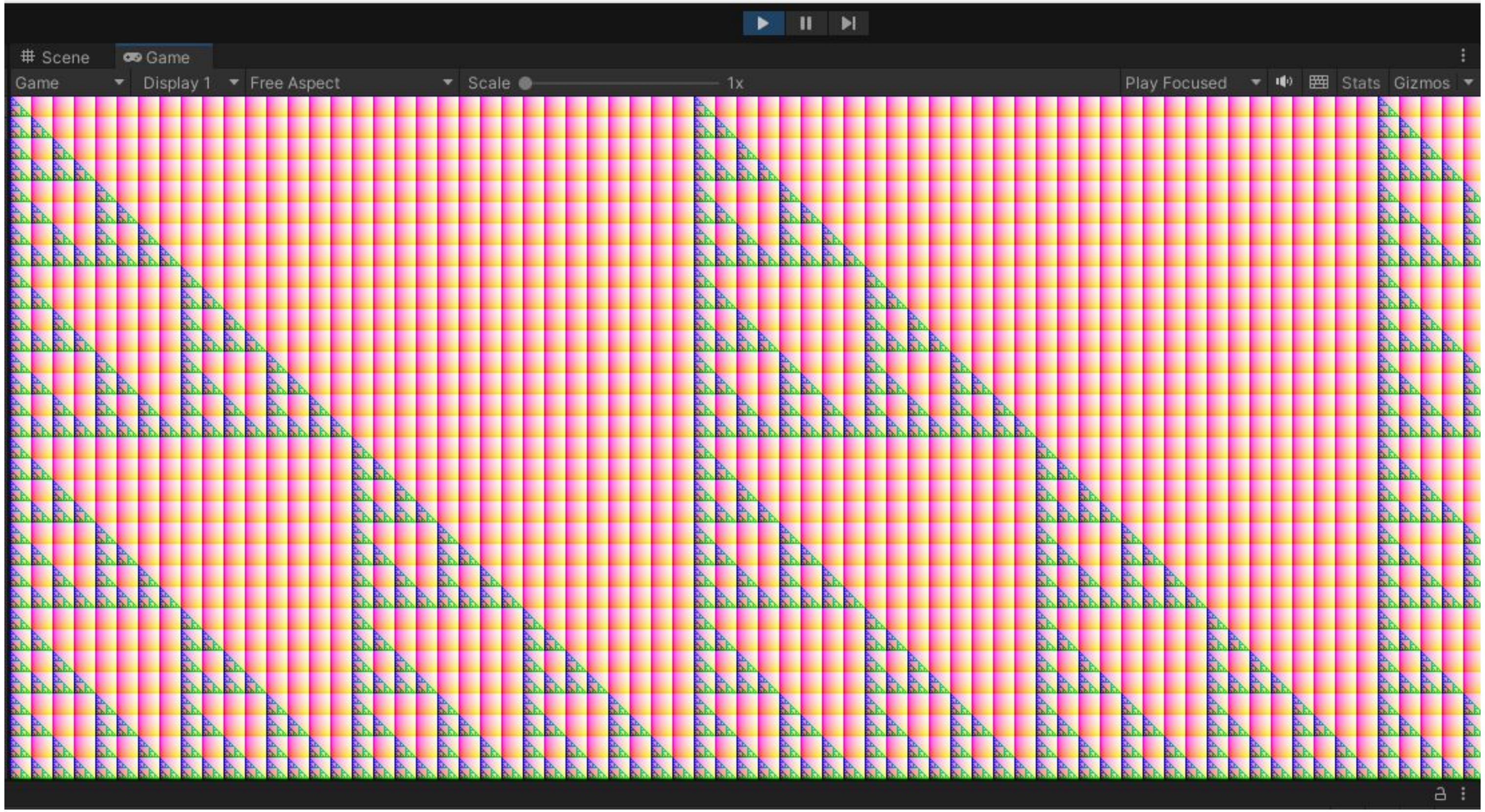
Script: RayTracing
Ray Tracing Shader: NewComputeShader

Add Component

drop the computeshader

in this field

All you should see is a triangle fractal

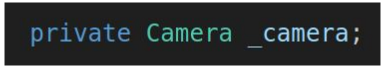


Camera

Now that we can display things on screen, let’s generate some camera rays.

Since Unity gives us a fully working camera, we will just use the calculated matrices to do this.

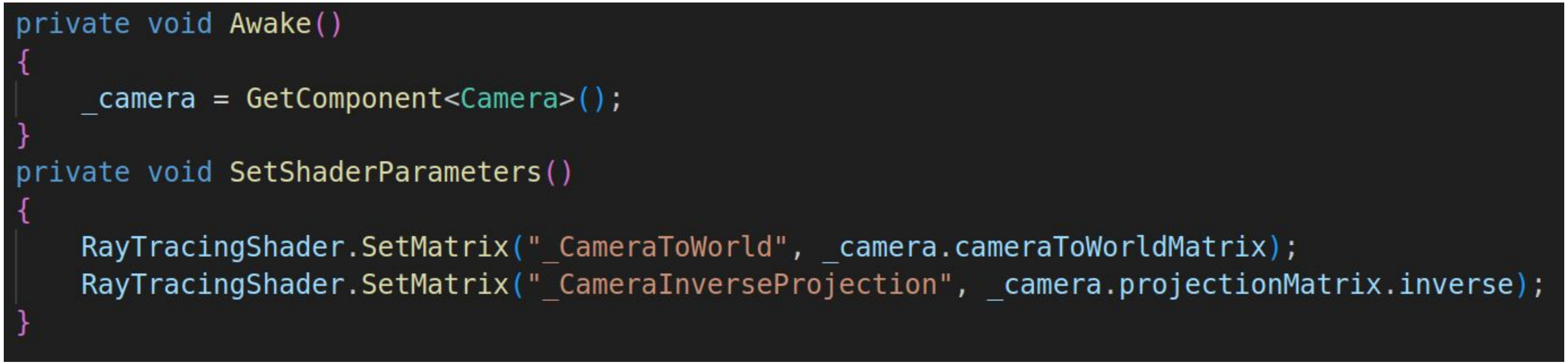
Start by setting the matrices on the shader. Add a variable for the camera to RayTracingMaster.cs:



```
private Camera _camera;
```

Camera

Make sure to call SetShaderParameters() from OnRenderImage before rendering

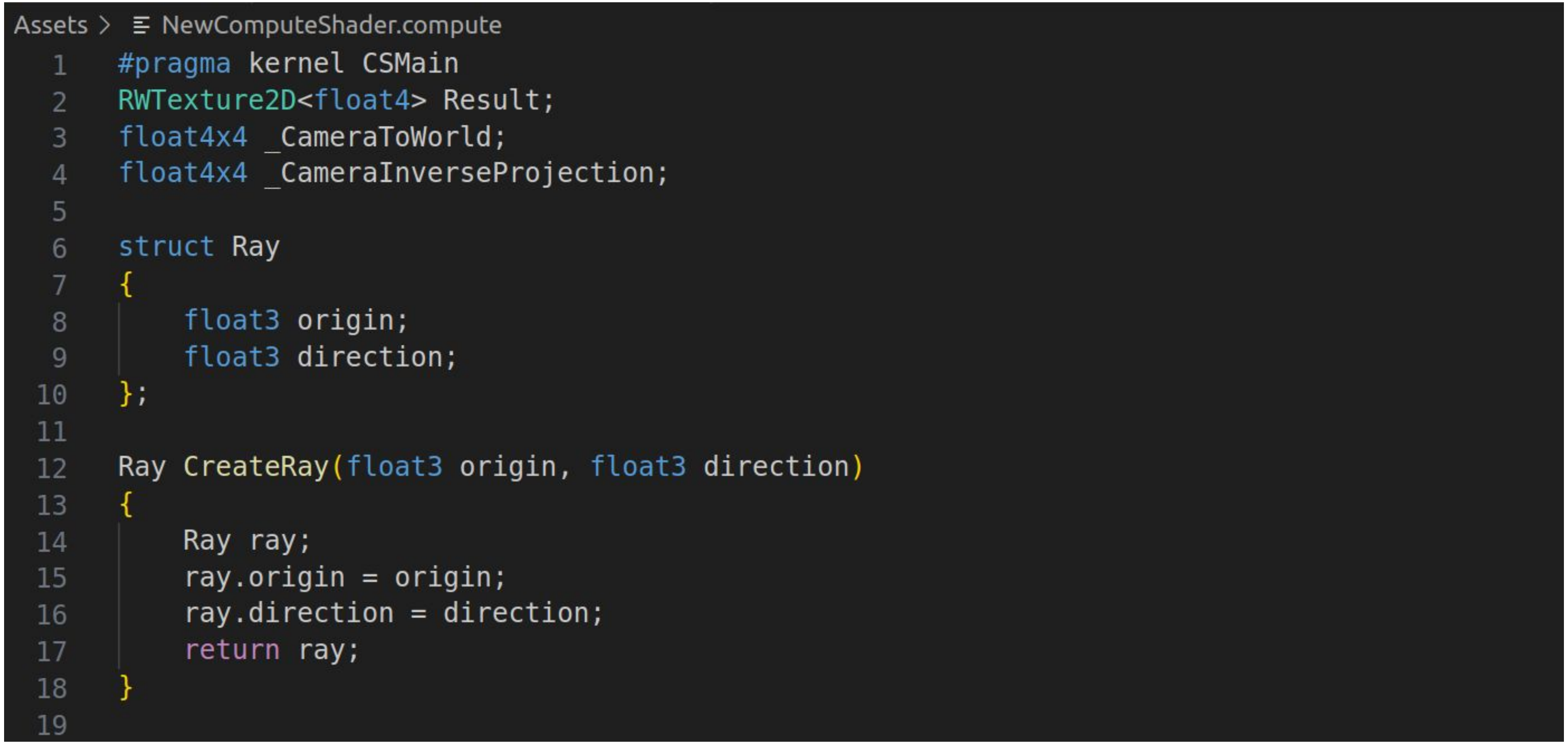


```
private void Awake()
{
    _camera = GetComponent<Camera>();
}
private void SetShaderParameters()
{
    RayTracingShader.SetMatrix("_CameraToWorld", _camera.cameraToWorldMatrix);
    RayTracingShader.SetMatrix("_CameraInverseProjection", _camera.projectionMatrix.inverse);
}
```

Basic Structure of Compute Shader

- we define the matrices, a Ray structure and a function for construction.
- in HLSL, a function or variable declaration needs to appear before it is being used !!!
- For each screen pixel’s center, we calculate the origin and direction of the ray
- Then we output the ray as colour

Compute Shader


Assets > NewComputeShader.compute

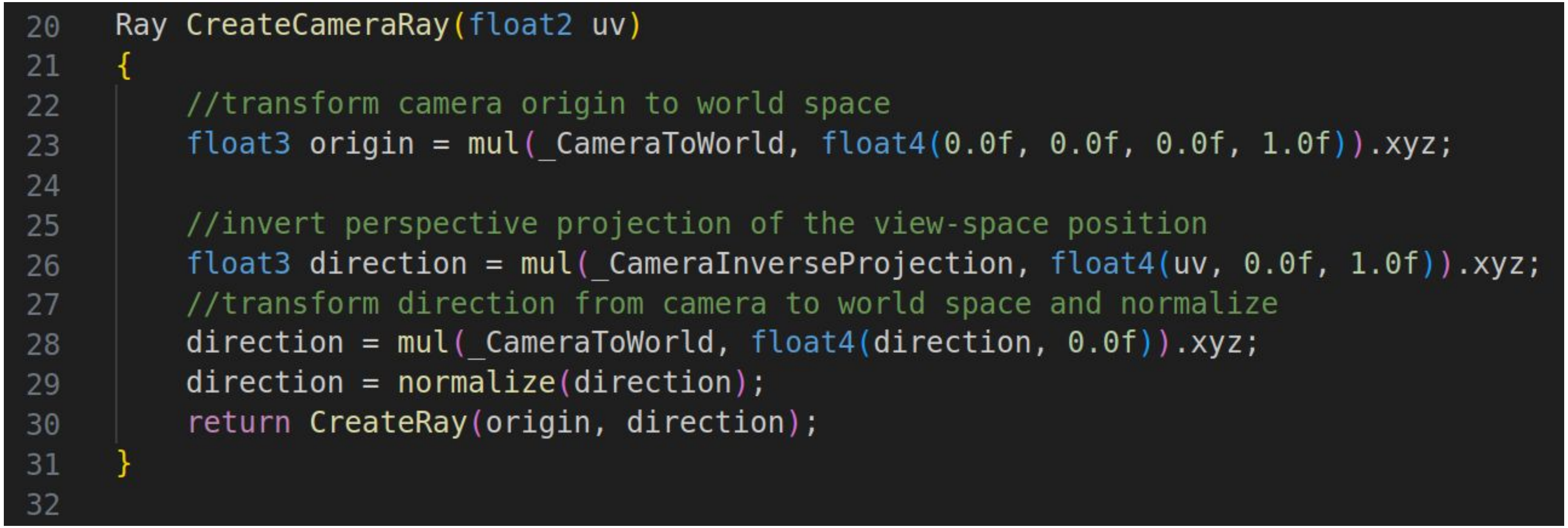
```
#pragma kernel CSMain
RWTexture2D<float4> Result;
float4x4 _CameraToWorld;
float4x4 _CameraInverseProjection;

struct Ray
{
    float3 origin;
    float3 direction;
};

Ray CreateRay(float3 origin, float3 direction)
{
    Ray ray;
    ray.origin = origin;
    ray.direction = direction;
    return ray;
}

```

Compute Shader

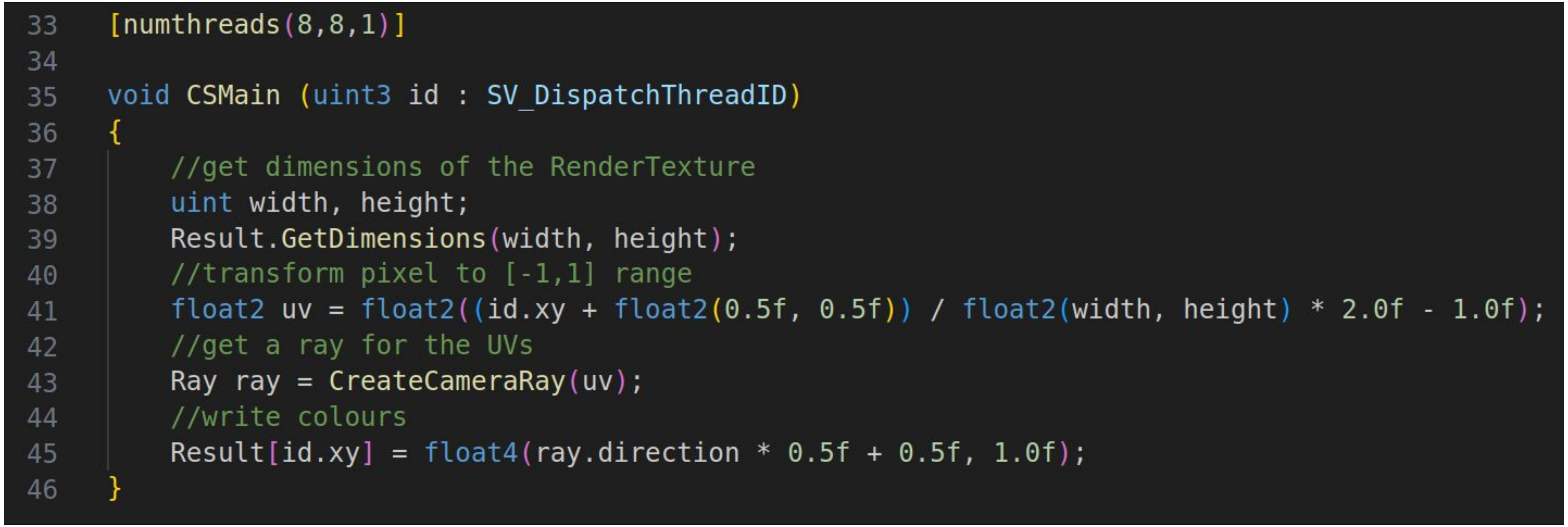


```
Ray CreateCameraRay(float2 uv)
{
    //transform camera origin to world space
    float3 origin = mul(_CameraToWorld, float4(0.0f, 0.0f, 0.0f, 1.0f)).xyz;

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

```

Compute Shader



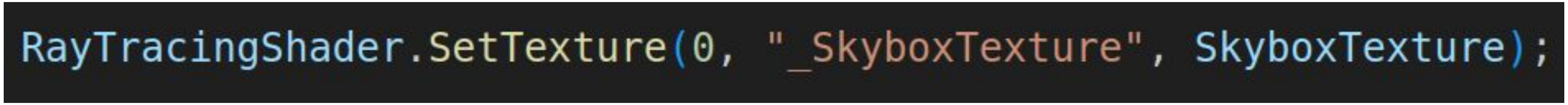
```
[numthreads(8,8,1)]

void CSMain (uint3 id : SV_DispatchThreadID)
{
    //get 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 colours
    Result[id.xy] = float4(ray.direction * 0.5f + 0.5f, 1.0f);
}
```

Compute Shader

Now we replace the colours with a skybox:

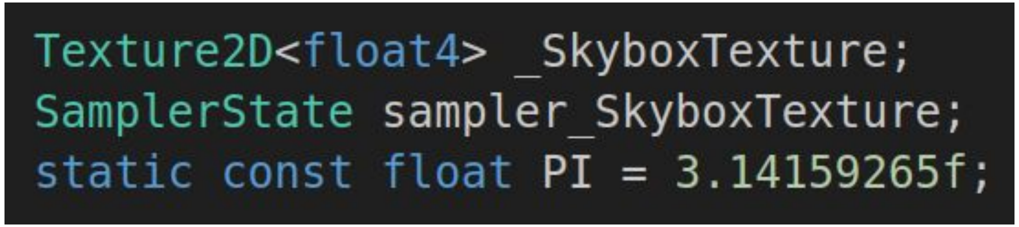
- We use https://polyhaven.com/a/sunset_fairway, but you can change it to your liking
- Download and drop it into Unity in the Assets folder
- Now add a public Texture SkyboxTexture to the script, assign your texture in the inspector
- then set it on the shader by adding this line to the SetShaderParameters function:



```
RayTracingShader.SetTexture(0, "_SkyboxTexture", SkyboxTexture);
```

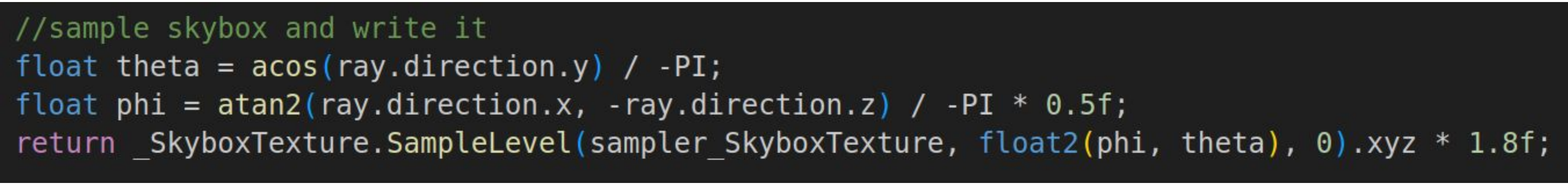
Camera

- in the shader, define the texture and a corresponding sampler, and a pi constant



```
Texture2D<float4> _SkyboxTexture;
SamplerState sampler_SkyboxTexture;
static const float PI = 3.14159265f;
```

- add this to the shade function in the shader
- transform cartesian direction vector to spherical coordinates and map it to texture coordinates



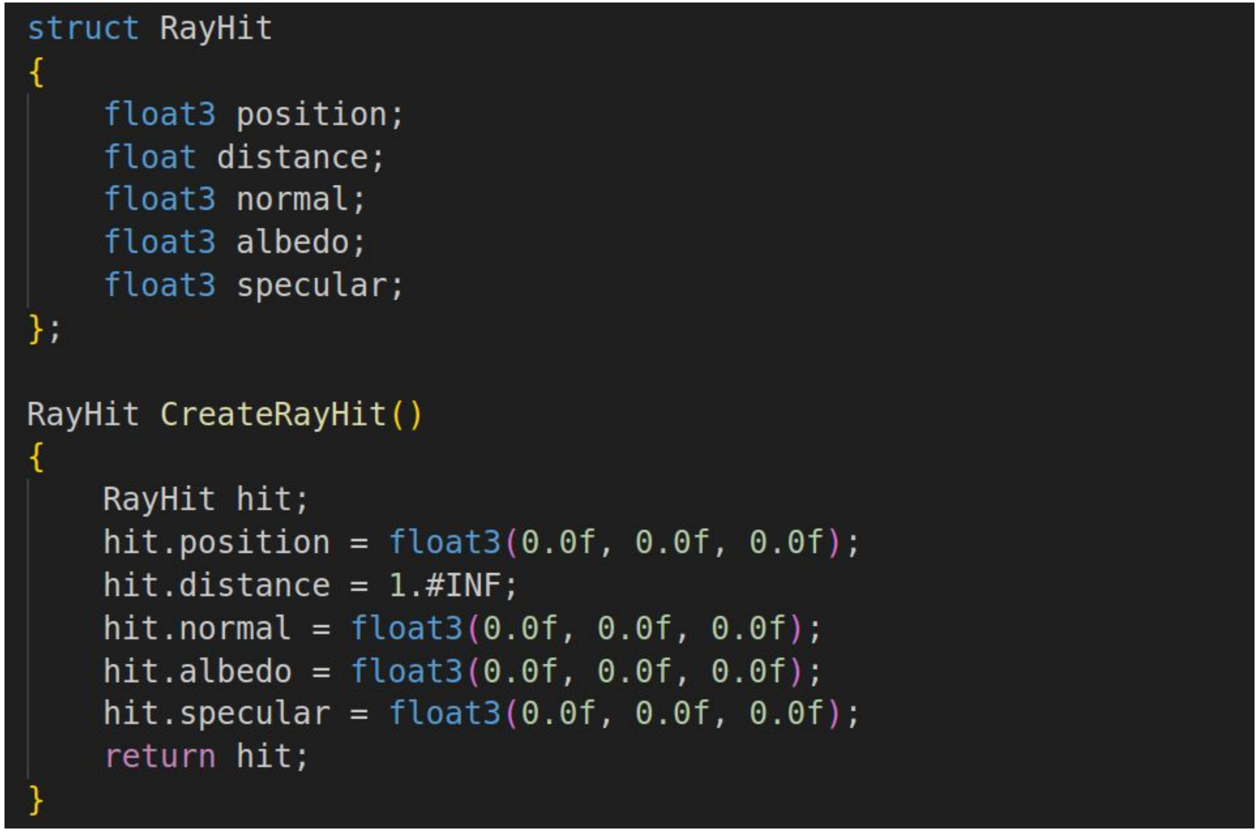
```
//sample skybox and write it
float theta = acos(ray.direction.y) / -PI;
float phi = atan2(ray.direction.x, -ray.direction.z) / -PI * 0.5f;
return _SkyboxTexture.SampleLevel(sampler_SkyboxTexture, float2(phi, theta), 0).xyz * 1.8f;
```

Tracing

- we will calculate the intersection between our ray and our scene geometry
- we store the hit parameters (position, normal and distance along the ray)
- If our ray hits multiple objects, we will pick the closest one
- Let’s define the struct RayHit in the shader:

Tracing

- define the struct RayHit in the shader:



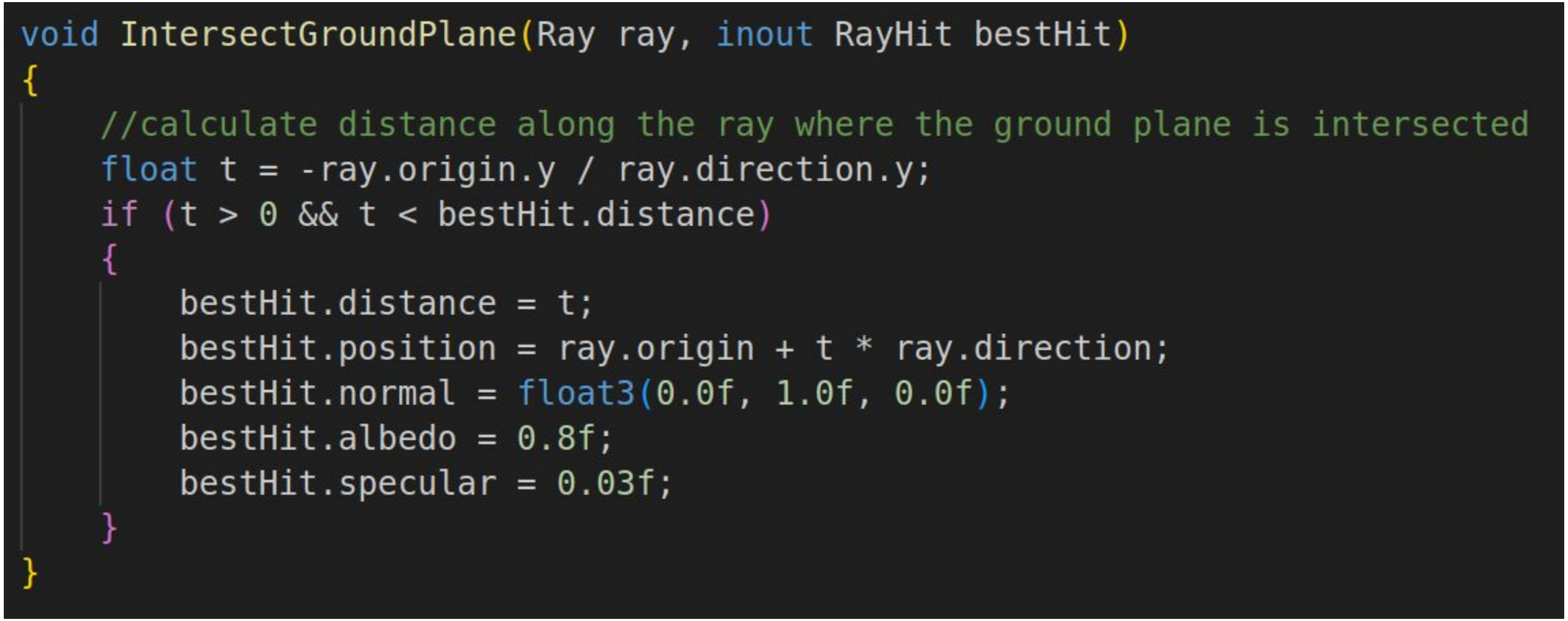
```
struct RayHit
{
    float3 position;
    float distance;
    float3 normal;
    float3 albedo;
    float3 specular;
};

RayHit CreateRayHit()
{
    RayHit hit;
    hit.position = float3(0.0f, 0.0f, 0.0f);
    hit.distance = 1.#INF;
    hit.normal = float3(0.0f, 0.0f, 0.0f);
    hit.albedo = float3(0.0f, 0.0f, 0.0f);
    hit.specular = float3(0.0f, 0.0f, 0.0f);
    return hit;
}
```

Environment

- Intersecting a line with an infinite plane at y=0 is fairly straightforward
- We only accept hits in positive ray direction, and reject any hit not closer than a potential previous hit
- By default, parameters in HLSL are passed by value and not by reference
- => we would only be able to work on a copy and not propagate changes to the calling function
- We pass RayHit bestHit with the inout qualifier to be able to modify the original struct

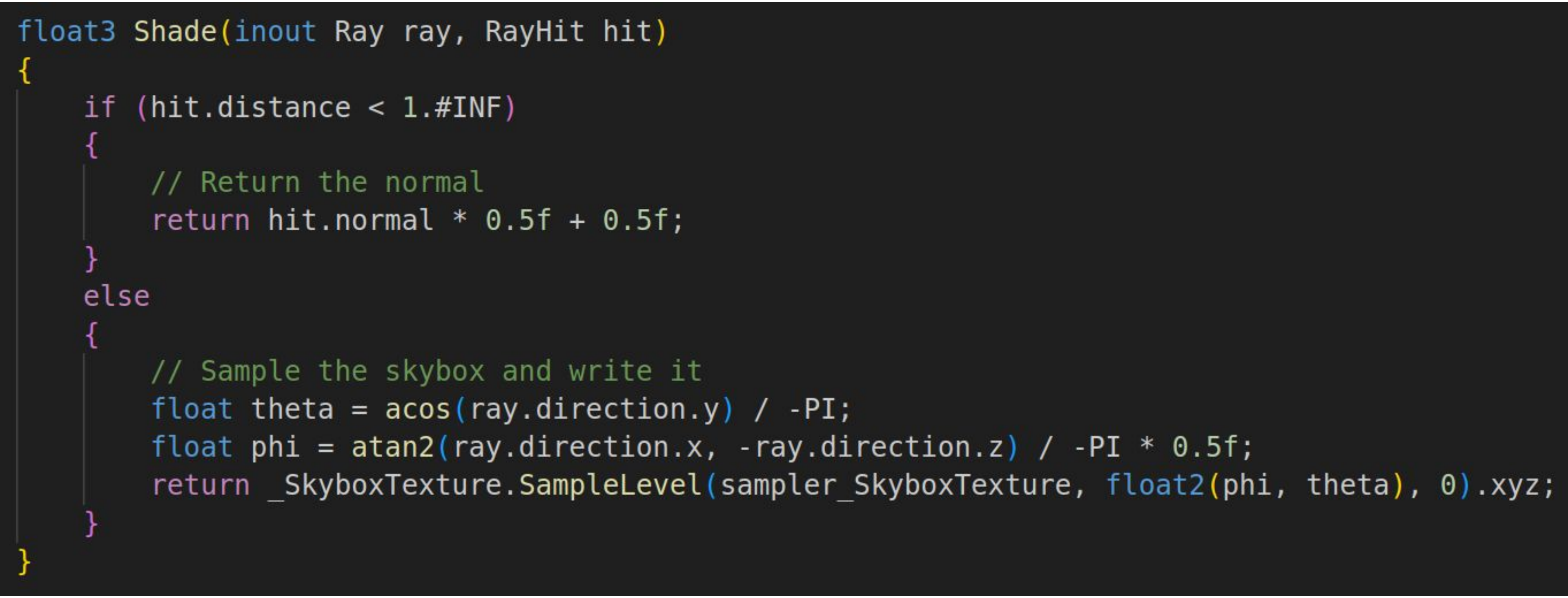
Environment: Shader Code



```
void IntersectGroundPlane(Ray ray, inout RayHit bestHit)
{
    //calculate distance along the ray where the ground plane is intersected
    float t = -ray.origin.y / ray.direction.y;
    if (t > 0 && t < bestHit.distance)
    {
        bestHit.distance = t;
        bestHit.position = ray.origin + t * ray.direction;
        bestHit.normal = float3(0.0f, 1.0f, 0.0f);
        bestHit.albedo = 0.8f;
        bestHit.specular = 0.03f;
    }
}
```

Environment: Shader Code - shade function

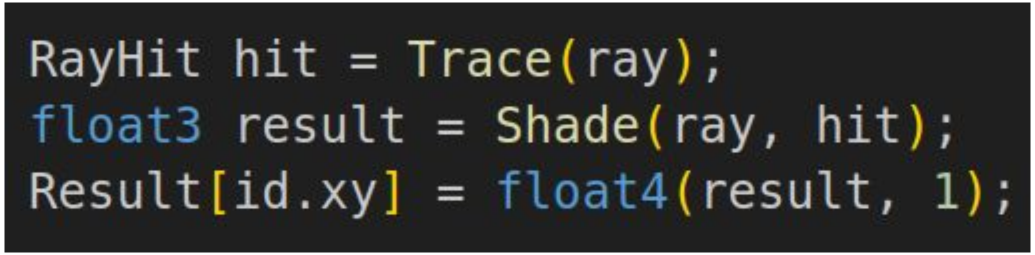
- we pass the Ray with inout – we will modify it later on when we talk about reflection
- we return the normal if geometry was hit, and choose our skybox sampling code otherwise



```
float3 Shade(inout Ray ray, RayHit hit)
{
    if (hit.distance < 1.#INF)
    {
        // Return the normal
        return hit.normal * 0.5f + 0.5f;
    }
    else
    {
        // Sample the skybox and write it
        float theta = acos(ray.direction.y) / -PI;
        float phi = atan2(ray.direction.x, -ray.direction.z) / -PI * 0.5f;
        return _SkyboxTexture.SampleLevel(sampler_SkyboxTexture, float2(phi, theta), 0).xyz;
    }
}
```

Environment: Shader Code - shade function

- we use both functions in CSMain:

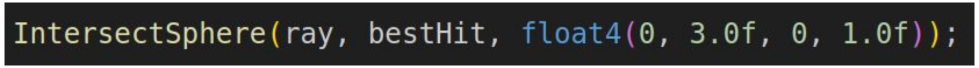


```
RayHit hit = Trace(ray);
float3 result = Shade(ray, hit);
Result[id.xy] = float4(result, 1);
```

Sphere Intersection

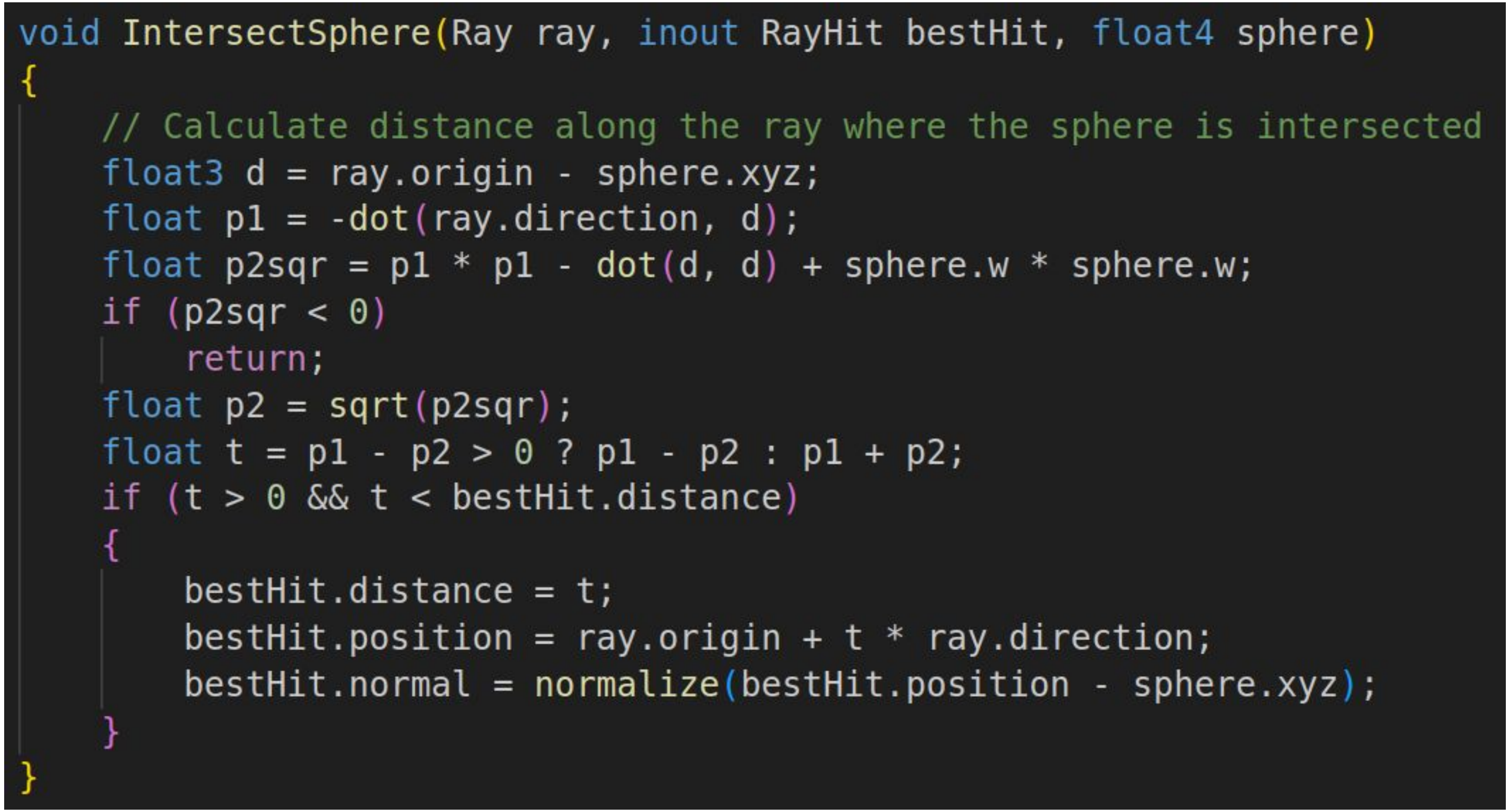
- Now we add a sphere
- If you need help for the line-sphere intersection check Wikipedia
- here there can be two ray hit candidates: the entry point p1 - p2, and the exit point p1 + p2
- We check the entry point first, and only use the exit point if the other one is not valid
- A sphere in our case is defined as a float4 composed of position (xyz) and radius (w)
- Try and implement the IntersectSphere function on your own!

Tip: to add a sphere call the function from Trace



```
IntersectSphere(ray, bestHit, float4(0, 3.0f, 0, 1.0f));
```

Sphere Intersection



```
void IntersectSphere(Ray ray, inout RayHit bestHit, float4 sphere)
{
    // Calculate distance along the ray where the sphere is intersected
    float3 d = ray.origin - sphere.xyz;
    float p1 = -dot(ray.direction, d);
    float p2sqr = p1 * p1 - dot(d, d) + sphere.w * sphere.w;
    if (p2sqr < 0)
        return;
    float p2 = sqrt(p2sqr);
    float t = p1 - p2 > 0 ? p1 - p2 : p1 + p2;
    if (t > 0 && t < bestHit.distance)
    {
        bestHit.distance = t;
        bestHit.position = ray.origin + t * ray.direction;
        bestHit.normal = normalize(bestHit.position - sphere.xyz);
    }
}
```

Anti-Aliasing

- Problem with current approach: We are only testing the center of each pixel => aliasing effects
- To circumvent this we trace not one but multiple rays per pixel
- Each ray gets a random offset inside the pixel's region
- To keep a good frame rate, we do progressive sampling
- This means we trace one ray per pixel each frame and average the result over time if the camera does not move
- Every time the camera moves (or field of view, scene geometry, scene lighting), we to start all over

Add another compute shader

- You can simply create the AddShader.shader file in VSCode
- we will use it for adding up several results
- the first line has to be Shader "Hidden/AddShader"
- In the SubShader add this to enable alpha blending:

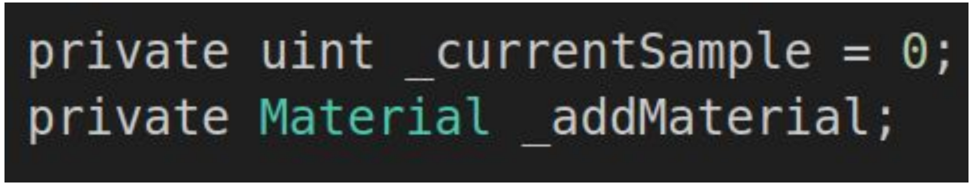
```
Cull Off ZWrite Off ZTest Always
Blend SrcAlpha OneMinusSrcAlpha
```

- replace the default frag function with the following lines:

```
float _Sample;
float4 frag (v2f i) : SV_Target
{
    return float4(tex2D(_MainTex, i.uv).rgb, 1.0f / (_Sample + 1.0f));
}
```

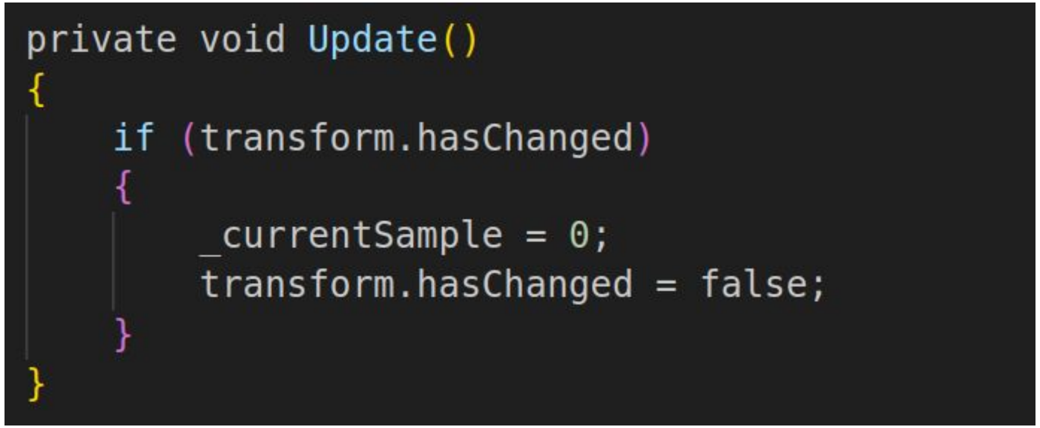
Add another compute shader

- the shader will now just draw the first sample with an opacity of 1, ½ , ⅓, … averaging all samples
- we need to count the samples and make use of the newly created image effect shader:



```
private uint _currentSample = 0;
private Material _addMaterial;
```

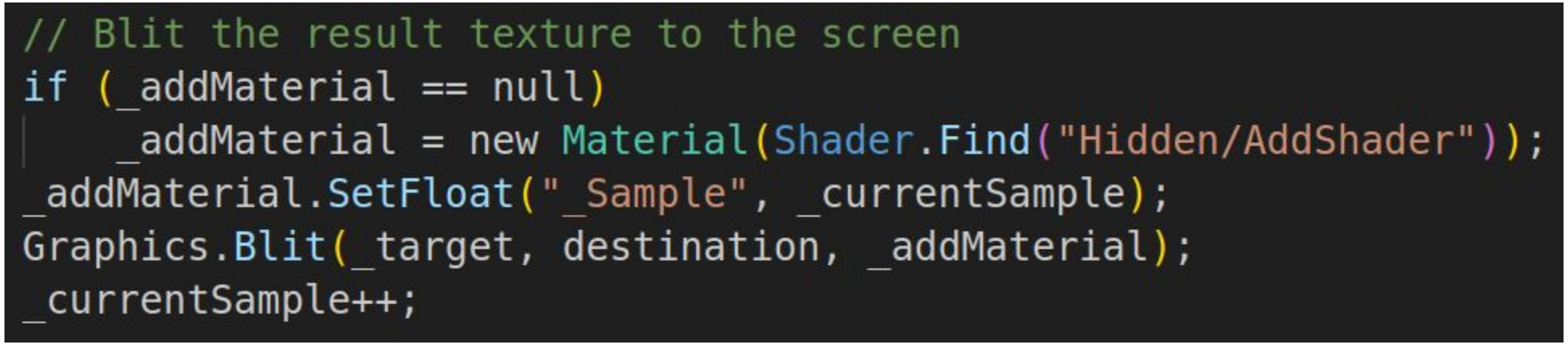
- We reset the samples when the target is rebuilt in InitRenderTexture, and detect camera changes:



```
private void Update()
{
    if (transform.hasChanged)
    {
        _currentSample = 0;
        transform.hasChanged = false;
    }
}
```

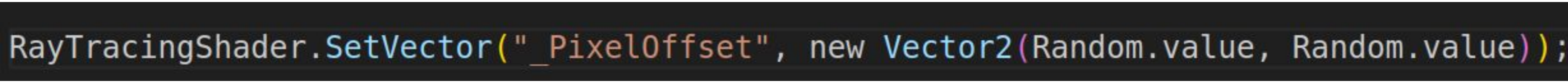
Add another compute shader

- to use our custom shader, we need to initialize a material, tell it about the current sample and use it for blitting to the screen in the Render function:



```
// Blit the result texture to the screen
if (_addMaterial == null)
    _addMaterial = new Material(Shader.Find("Hidden/AddShader"));
_addMaterial.SetFloat("_Sample", _currentSample);
Graphics.Blit(_target, destination, _addMaterial);
_currentSample++;
```

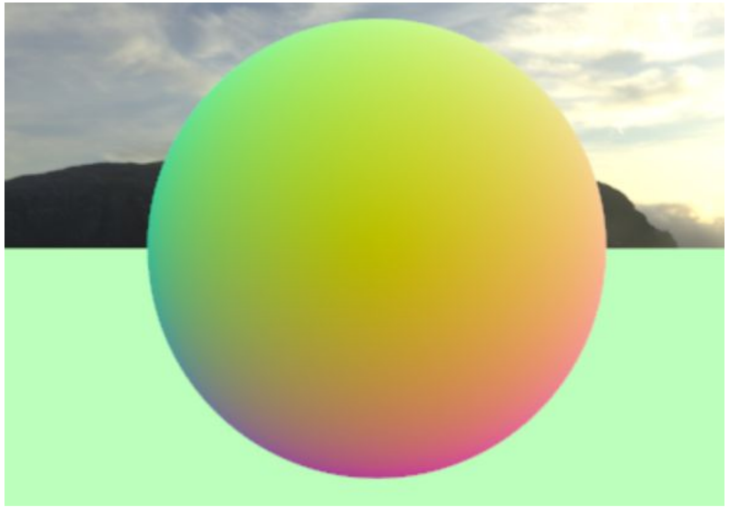
- we do progressive sampling, but still use the pixel center. In the compute shader, define a "float2 _PixelOffset" and use it in CSMain instead of the hard coded float2(0.5f, 0.5f) offset
- In the script create a random offset by adding this line to SetShaderParameters:



```
RayTracingShader.SetVector("_PixelOffset", new Vector2(Random.value, Random.value));
```

Add another compute shader

- if we move the camera, you should see that the image still shows aliasing, but it will quickly vanish if you stand still for a couple of frames

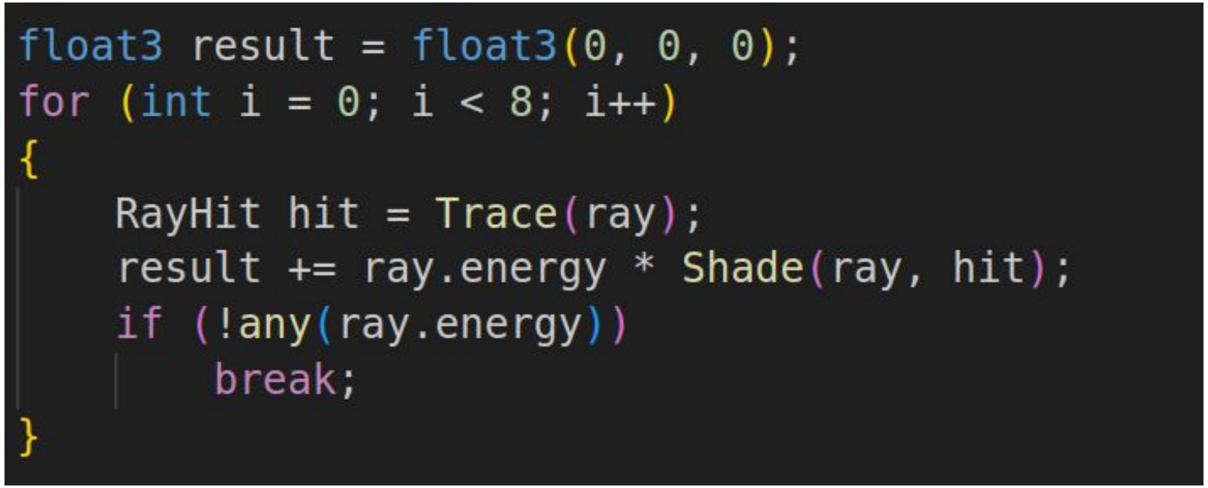


Add Reflection

- whenever we hit the surface, we reflect the ray according to the law of reflection (incident angle = angle of reflection), reduce its energy, and repeat until we either hit the sky, run out of energy or after a fixed amount of maximum bounces
- In the compute shader, add a float3 energy to the ray and initialize it in the CreateRay function as “ray.energy = float3(1.0f, 1.0f, 1.0f)”
- The ray starts with full throughput on all colour channels and diminishes with each reflection
- we execute a maximum number of 8 traces (the original ray plus 7 bounces)
- We add up the results of the Shade function calls, but multiplied with the ray’s energy

Add Reflection

- As an example, imagine a ray that has been reflected once and lost 34 of its energy
- now it travels and hits the sky, so we only transfer 14 of the energy of the sky hit to the pixel
- Adapt the CSMain like this:

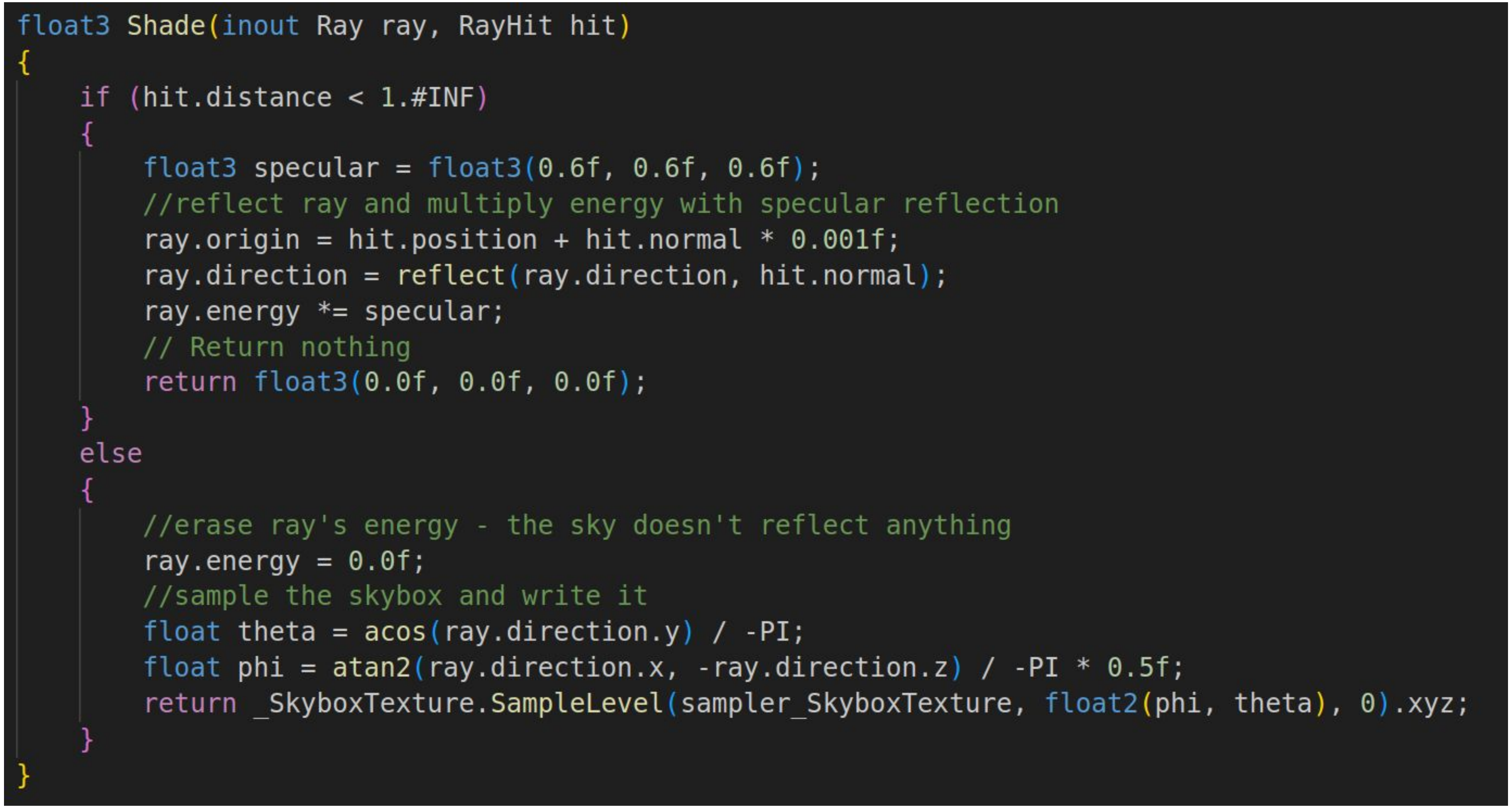


```
float3 result = float3(0, 0, 0);
for (int i = 0; i < 8; i++)
{
    RayHit hit = Trace(ray);
    result += ray.energy * Shade(ray, hit);
    if (!any(ray.energy))
        break;
}
```

Add Reflection

- The Shade function is now also responsible for updating the energy and generating the reflected ray
- To update the energy, we perform an element-wise multiplication with the specular colour of the surface
- gold for example has a specular reflectivity of roughly float3(1.0f, 0.78f, 0.34f) => it will reflect 100% of red light, 78% of green light, but only 34% of blue light, making the reflection appear golden
- ! Do not to go over 1 with any of those values, we would create energy out of nowhere !
- reflectivity is often lower than expected, google Hoffman for realistic values
- HLSL has an inbuilt function to reflect a ray using a given normal
- Due to floating point inaccuracy, a reflected ray can be blocked by the surface it is reflected on
- To prevent self-occlusion offset the position just a bit along the normal direction
- Try and update the Shade function yourself with this new knowledge

Add Reflection



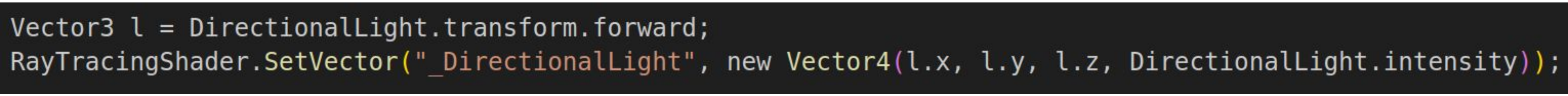
```
float3 Shade(inout Ray ray, RayHit hit)
{
    if (hit.distance < 1.#INF)
    {
        float3 specular = float3(0.6f, 0.6f, 0.6f);
        //reflect ray and multiply energy with specular reflection
        ray.origin = hit.position + hit.normal * 0.001f;
        ray.direction = reflect(ray.direction, hit.normal);
        ray.energy *= specular;
        // Return nothing
        return float3(0.0f, 0.0f, 0.0f);
    }
    else
    {
        //erase ray's energy - the sky doesn't reflect anything
        ray.energy = 0.0f;
        //sample the skybox and write it
        float theta = acos(ray.direction.y) / -PI;
        float phi = atan2(ray.direction.x, -ray.direction.z) / -PI * 0.5f;
        return _SkyboxTexture.SampleLevel(sampler_SkyboxTexture, float2(phi, theta), 0).xyz;
    }
}
```

Directional Light

- for non-metals we need one more thing: diffuse reflection
- metals will only reflect incoming light tinted with their specular color
- non-metals allow light to refract into the surface, scatter and leave it in a random direction tinted with their albedo color
- We assume an ideal Lambertian surface => the probability is proportional to the cosine of the angle between this direction and the surface normal
- You can use this this discussion to learn more about the topic: https://computergraphics.stackexchange.com/questions/1513/how-physically-based-is-the-diffuse-and-specular-distinction

Directional Light

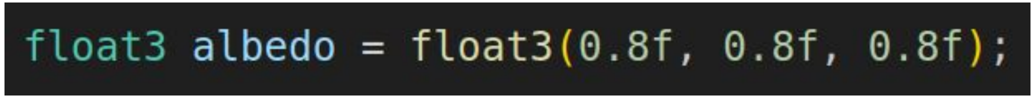
- To get started with diffuse lighting add a “public Light DirectionalLight” to RayTracing.cs and assign the scene’s directional light
- detect the light’s transform changes in the Update function, like we already do it for the camera’s transform changes
- add the following lines to the SetShaderParameters function:



```
Vector3 l = DirectionalLight.transform.forward;
RayTracingShader.SetVector("_DirectionalLight", new Vector4(l.x, l.y, l.z, DirectionalLight.intensity));
```

Directional Light

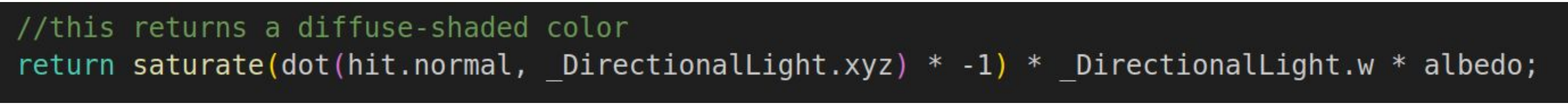
- in the shader define a “float4 _DirectionalLight”
- in the Shade function, define the albedo color right below the specular color



```
float3 albedo = float3(0.8f, 0.8f, 0.8f);
```

- Replace the previously black return with a simple diffuse shading
- Recall the definition of the dot product

Directional Light

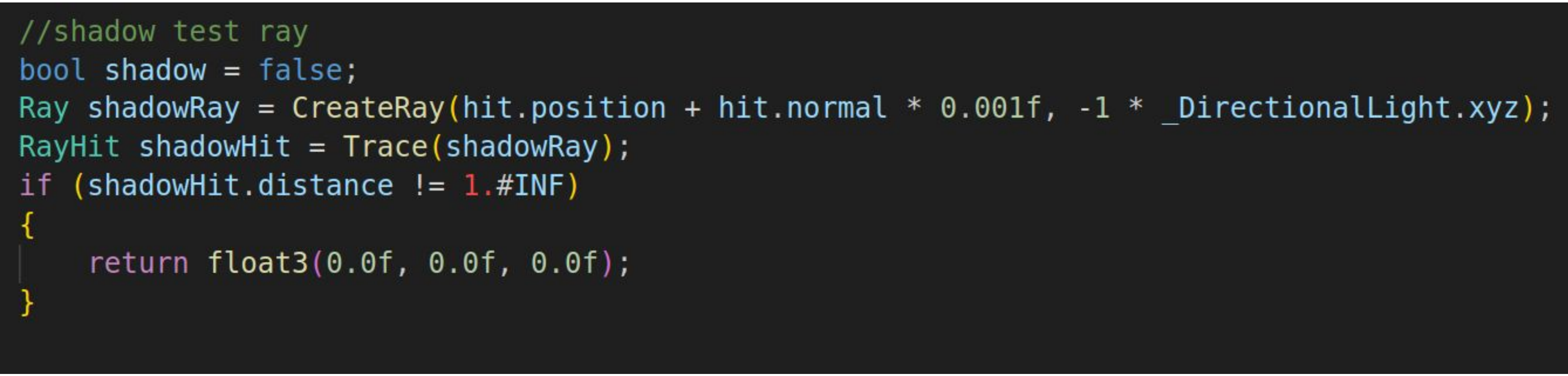


```
//this returns a diffuse-shaded color
return saturate(dot(hit.normal, _DirectionalLight.xyz) * -1) * _DirectionalLight.w * albedo;
```

- Since both our vectors (the normal and the light direction) are of unit length, the dot product is exactly what we are looking for: the cosine of the angle
- The ray and the light are pointing in opposite directions, so for head-on lighting the dot product returns -1 instead of 1

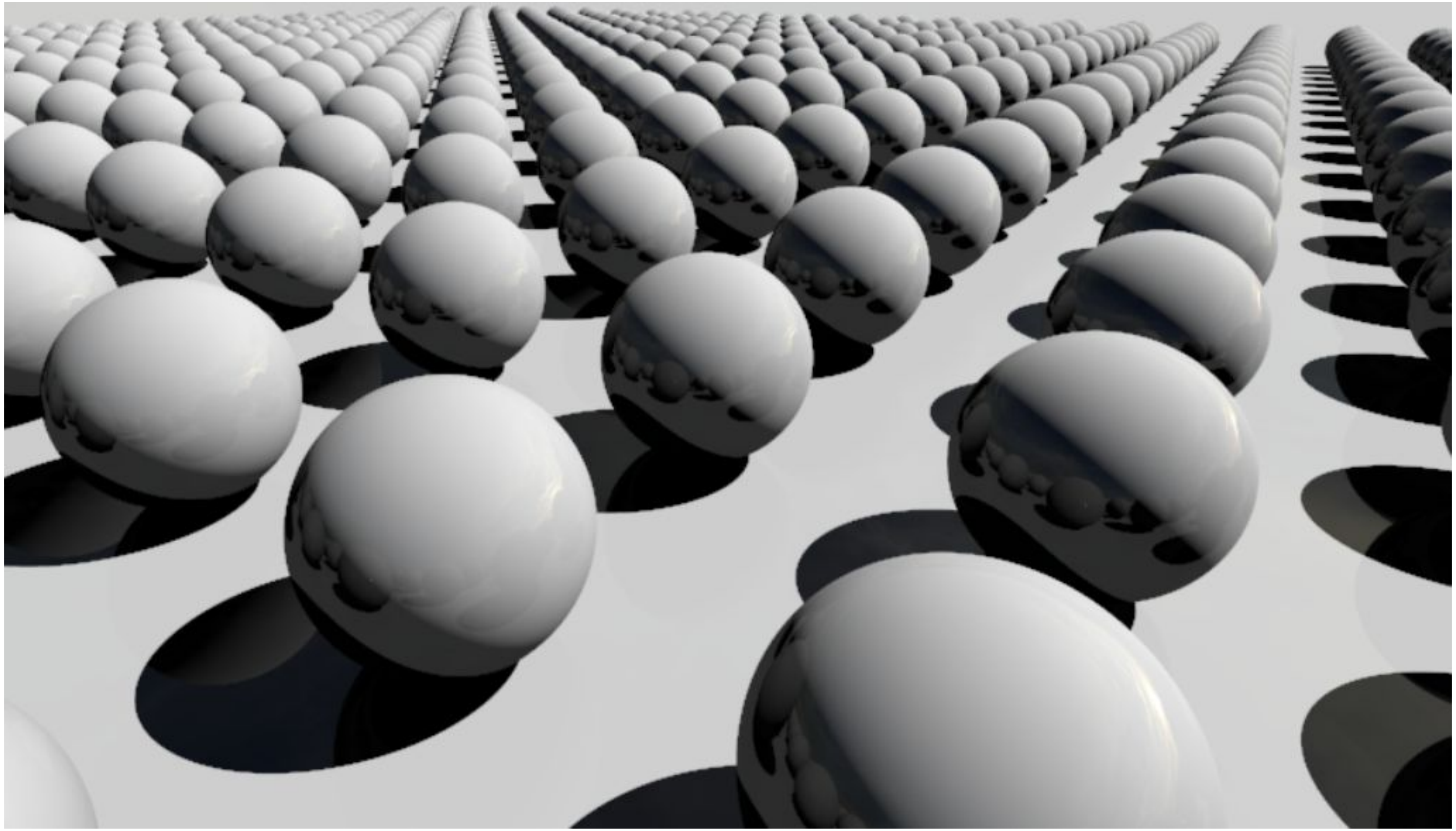
Directional Light

- For directional light to cast shadows, we trace a shadow ray
- starts at the surface position in question (with tiny displacement to avoid self-shadowing), and points in the light source direction
- should anything block the way => we won’t use any diffuse light
- therefore we add these lines before the diffuse light return statement



```
//shadow test ray
bool shadow = false;
Ray shadowRay = CreateRay(hit.position + hit.normal * 0.001f, -1 * _DirectionalLight.xyz);
RayHit shadowHit = Trace(shadowRay);
if (shadowHit.distance != 1.#INF)
{
    return float3(0.0f, 0.0f, 0.0f);
}
```

Directional Light

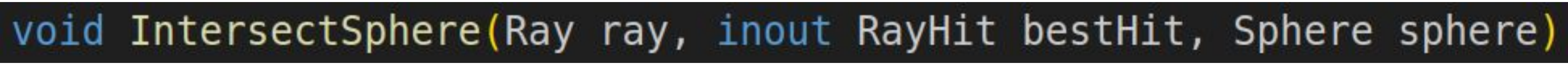


Materials: making the scene more colourful

- instead of hard-coding everything in the shader, we define the scene in C# for more flexibility
- First extend the RayHit structure in the shader
- instead of globally defining the material properties in the Shade function, define them per object and store in the RayHit
- Add float3 albedo and float3 specular to the struct + initialize them to float3(0.0f, 0.0f, 0.0f) in CreateRayHit
- adjust the Shade function to use these values from hit instead of the hard-coded ones
- To establish what a sphere is on the CPU and the GPU, define a struct Sphere both in your shader and in the C# script

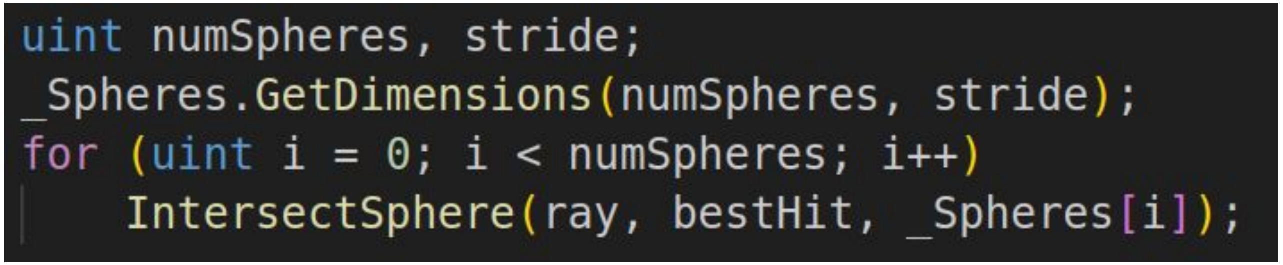
Materials

- In the shader make the IntersectSphere function work with our custom struct instead of the float4:



```
void IntersectSphere(Ray ray, inout RayHit bestHit, Sphere sphere)
```

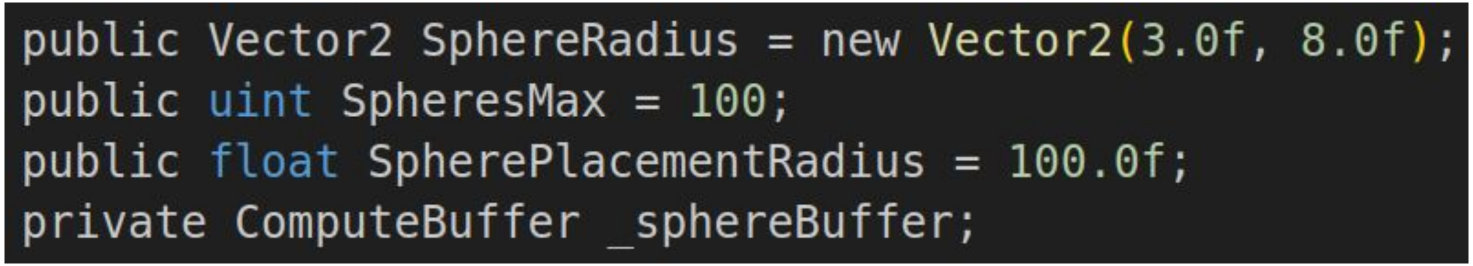
- set bestHit.albedo and bestHit.specular in the IntersectGroundPlane function to adjust its material
- Next, define StructuredBuffer<Sphere> _Spheres, this where the CPU will store all spheres
- Remove all hardcoded spheres from the Trace function and adapt it:



```
uint numSpheres, stride;
_Spheres.GetDimensions(numSpheres, stride);
for (uint i = 0; i < numSpheres; i++)
    IntersectSphere(ray, bestHit, _Spheres[i]);
```

Materials

- In the script add some public parameters to control sphere placement and the compute buffer:



```
public Vector2 SphereRadius = new Vector2(3.0f, 8.0f);
public uint SpheresMax = 100;
public float SpherePlacementRadius = 100.0f;
private ComputeBuffer _sphereBuffer;
```

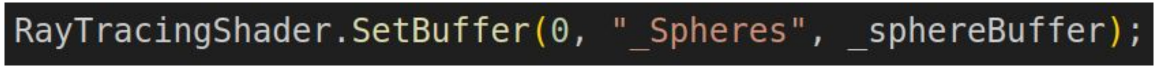
- Set up the scene in OnEnable, and release the buffer in OnDisable
- This way, a random scene will be generated every time we enable the component

Materials

- The SetUpScene function positions spheres in a certain radius, and rejects those that would intersect existing spheres
- Half of the spheres are metallic (black albedo, colored specular), the other half is non-metallic (coloured albedo, 4% specular)
- The number 40 in new ComputeBuffer(spheres.Count, 40) is the stride of the buffer: the byte size of one sphere in memory

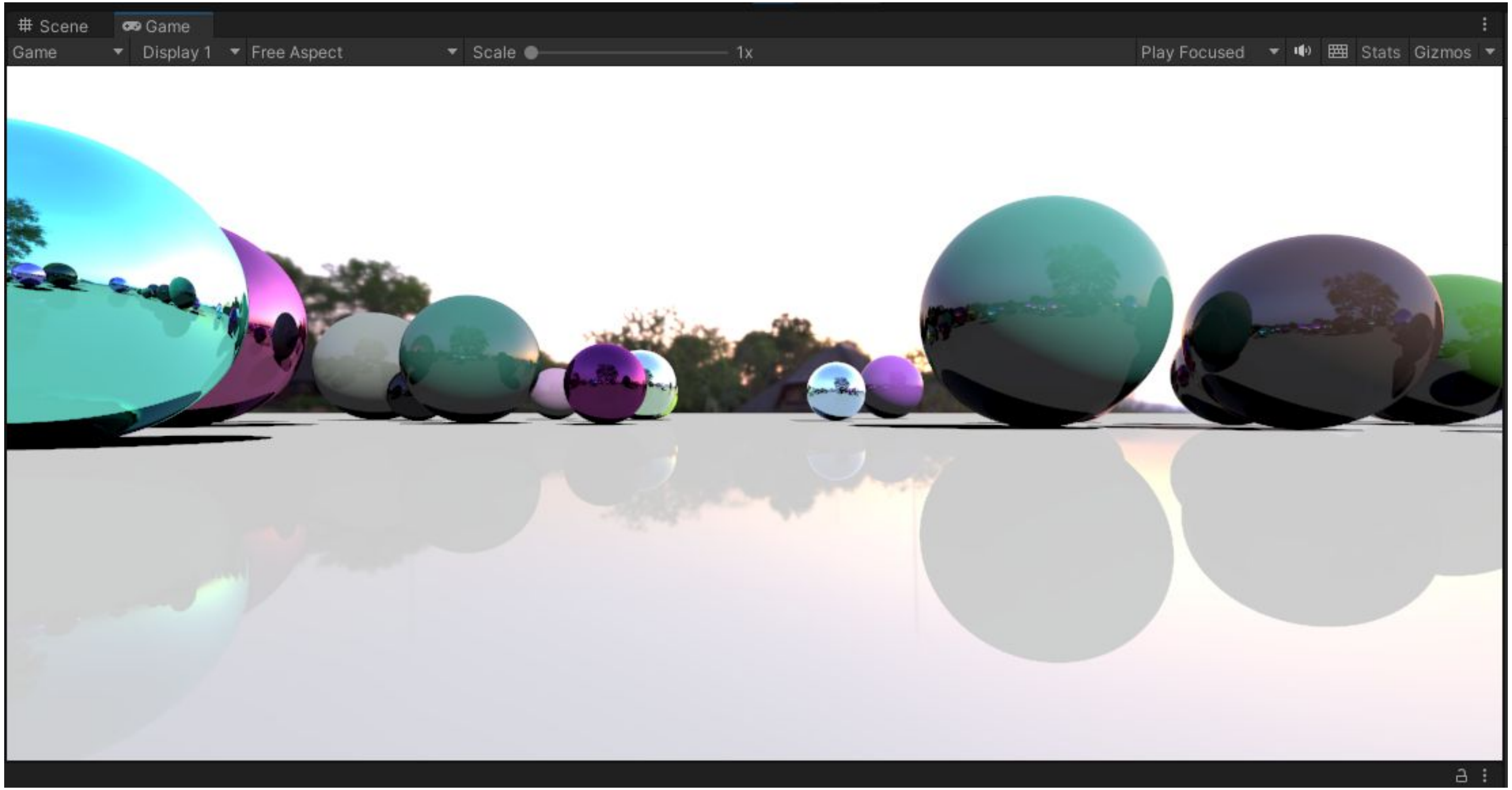
Materials

- To calculate this count the number of floats in the Sphere struct and multiply it by float’s byte size (4 bytes)
- Finally, set the buffer on the shader in the SetShaderParameters:



```
RayTracingShader.SetBuffer(0, "_Spheres", _sphereBuffer);
```

Final Result - Example:


Scene
Game
Game
Display 1
Free Aspect
Scale
1x
Play Focused
Stats
Gizmos