using System;

using UnityEngine;

namespace UnityStandardAssets.ImageEffects

{

[ExecuteInEditMode]

[RequireComponent (typeof(Camera))]

[AddComponentMenu("Image Effects/Rendering/Screen Space Ambient Occlusion")]

public class ScreenSpaceAmbientOcclusion : MonoBehaviour

{

public enum SSAOSamples

{

Low = 0,

Medium = 1,

High = 2,

}

public float m\_Radius = 0.4f;

public SSAOSamples m\_SampleCount = SSAOSamples.Medium;

public float m\_OcclusionIntensity = 1.5f;

public int m\_Blur = 2;

public int m\_Downsampling = 2;

public float m\_OcclusionAttenuation = 1.0f;

public float m\_MinZ = 0.01f;

public Shader m\_SSAOShader;

private Material m\_SSAOMaterial;

public Texture2D m\_RandomTexture;

private bool m\_Supported;

private static Material CreateMaterial (Shader shader)

{

if (!shader)

return null;

Material m = new Material (shader);

m.hideFlags = HideFlags.HideAndDontSave;

return m;

}

private static void DestroyMaterial (Material mat)

{

if (mat)

{

DestroyImmediate (mat);

mat = null;

}

}

void OnDisable()

{

DestroyMaterial (m\_SSAOMaterial);

}

void Start()

{

if (!SystemInfo.supportsImageEffects || !SystemInfo.SupportsRenderTextureFormat (RenderTextureFormat.Depth))

{

m\_Supported = false;

enabled = false;

return;

}

CreateMaterials ();

if (!m\_SSAOMaterial || m\_SSAOMaterial.passCount != 5)

{

m\_Supported = false;

enabled = false;

return;

}

//CreateRandomTable (26, 0.2f);

m\_Supported = true;

}

void OnEnable () {

GetComponent<Camera>().depthTextureMode |= DepthTextureMode.DepthNormals;

}

private void CreateMaterials ()

{

if (!m\_SSAOMaterial && m\_SSAOShader.isSupported)

{

m\_SSAOMaterial = CreateMaterial (m\_SSAOShader);

m\_SSAOMaterial.SetTexture ("\_RandomTexture", m\_RandomTexture);

}

}

[ImageEffectOpaque]

void OnRenderImage (RenderTexture source, RenderTexture destination)

{

if (!m\_Supported || !m\_SSAOShader.isSupported) {

enabled = false;

return;

}

CreateMaterials ();

m\_Downsampling = Mathf.Clamp (m\_Downsampling, 1, 6);

m\_Radius = Mathf.Clamp (m\_Radius, 0.05f, 1.0f);

m\_MinZ = Mathf.Clamp (m\_MinZ, 0.00001f, 0.5f);

m\_OcclusionIntensity = Mathf.Clamp (m\_OcclusionIntensity, 0.5f, 4.0f);

m\_OcclusionAttenuation = Mathf.Clamp (m\_OcclusionAttenuation, 0.2f, 2.0f);

m\_Blur = Mathf.Clamp (m\_Blur, 0, 4);

// Render SSAO term into a smaller texture

RenderTexture rtAO = RenderTexture.GetTemporary (source.width / m\_Downsampling, source.height / m\_Downsampling, 0);

float fovY = GetComponent<Camera>().fieldOfView;

float far = GetComponent<Camera>().farClipPlane;

float y = Mathf.Tan (fovY \* Mathf.Deg2Rad \* 0.5f) \* far;

float x = y \* GetComponent<Camera>().aspect;

m\_SSAOMaterial.SetVector ("\_FarCorner", new Vector3(x,y,far));

int noiseWidth, noiseHeight;

if (m\_RandomTexture) {

noiseWidth = m\_RandomTexture.width;

noiseHeight = m\_RandomTexture.height;

} else {

noiseWidth = 1; noiseHeight = 1;

}

m\_SSAOMaterial.SetVector ("\_NoiseScale", new Vector3 ((float)rtAO.width / noiseWidth, (float)rtAO.height / noiseHeight, 0.0f));

m\_SSAOMaterial.SetVector ("\_Params", new Vector4(

m\_Radius,

m\_MinZ,

1.0f / m\_OcclusionAttenuation,

m\_OcclusionIntensity));

bool doBlur = m\_Blur > 0;

Graphics.Blit (doBlur ? null : source, rtAO, m\_SSAOMaterial, (int)m\_SampleCount);

if (doBlur)

{

// Blur SSAO horizontally

RenderTexture rtBlurX = RenderTexture.GetTemporary (source.width, source.height, 0);

m\_SSAOMaterial.SetVector ("\_TexelOffsetScale",

new Vector4 ((float)m\_Blur / source.width, 0,0,0));

m\_SSAOMaterial.SetTexture ("\_SSAO", rtAO);

Graphics.Blit (null, rtBlurX, m\_SSAOMaterial, 3);

RenderTexture.ReleaseTemporary (rtAO); // original rtAO not needed anymore

// Blur SSAO vertically

RenderTexture rtBlurY = RenderTexture.GetTemporary (source.width, source.height, 0);

m\_SSAOMaterial.SetVector ("\_TexelOffsetScale",

new Vector4 (0, (float)m\_Blur/source.height, 0,0));

m\_SSAOMaterial.SetTexture ("\_SSAO", rtBlurX);

Graphics.Blit (source, rtBlurY, m\_SSAOMaterial, 3);

RenderTexture.ReleaseTemporary (rtBlurX); // blurX RT not needed anymore

rtAO = rtBlurY; // AO is the blurred one now

}

// Modulate scene rendering with SSAO

m\_SSAOMaterial.SetTexture ("\_SSAO", rtAO);

Graphics.Blit (source, destination, m\_SSAOMaterial, 4);

RenderTexture.ReleaseTemporary (rtAO);

}

/\*

private void CreateRandomTable (int count, float minLength)

{

Random.seed = 1337;

Vector3[] samples = new Vector3[count];

// initial samples

for (int i = 0; i < count; ++i)

samples[i] = Random.onUnitSphere;

// energy minimization: push samples away from others

int iterations = 100;

while (iterations-- > 0) {

for (int i = 0; i < count; ++i) {

Vector3 vec = samples[i];

Vector3 res = Vector3.zero;

// minimize with other samples

for (int j = 0; j < count; ++j) {

Vector3 force = vec - samples[j];

float fac = Vector3.Dot (force, force);

if (fac > 0.00001f)

res += force \* (1.0f / fac);

}

samples[i] = (samples[i] + res \* 0.5f).normalized;

}

}

// now scale samples between minLength and 1.0

for (int i = 0; i < count; ++i) {

samples[i] = samples[i] \* Random.Range (minLength, 1.0f);

}

string table = string.Format ("#define SAMPLE\_COUNT {0}\n", count);

table += "const float3 RAND\_SAMPLES[SAMPLE\_COUNT] = {\n";

for (int i = 0; i < count; ++i) {

Vector3 v = samples[i];

table += string.Format("\tfloat3({0},{1},{2}),\n", v.x, v.y, v.z);

}

table += "};\n";

Debug.Log (table);

}

\*/

}

}