using System;

using System.Collections.Generic;

using UnityEngine;

namespace UnityStandardAssets.Water

{

[ExecuteInEditMode] // Make water live-update even when not in play mode

public class Water : MonoBehaviour

{

public enum WaterMode

{

Simple = 0,

Reflective = 1,

Refractive = 2,

};

public WaterMode waterMode = WaterMode.Refractive;

public bool disablePixelLights = true;

public int textureSize = 256;

public float clipPlaneOffset = 0.07f;

public LayerMask reflectLayers = -1;

public LayerMask refractLayers = -1;

private Dictionary<Camera, Camera> m\_ReflectionCameras = new Dictionary<Camera, Camera>(); // Camera -> Camera table

private Dictionary<Camera, Camera> m\_RefractionCameras = new Dictionary<Camera, Camera>(); // Camera -> Camera table

private RenderTexture m\_ReflectionTexture;

private RenderTexture m\_RefractionTexture;

private WaterMode m\_HardwareWaterSupport = WaterMode.Refractive;

private int m\_OldReflectionTextureSize;

private int m\_OldRefractionTextureSize;

private static bool s\_InsideWater;

// This is called when it's known that the object will be rendered by some

// camera. We render reflections / refractions and do other updates here.

// Because the script executes in edit mode, reflections for the scene view

// camera will just work!

public void OnWillRenderObject()

{

if (!enabled || !GetComponent<Renderer>() || !GetComponent<Renderer>().sharedMaterial ||

!GetComponent<Renderer>().enabled)

{

return;

}

Camera cam = Camera.current;

if (!cam)

{

return;

}

// Safeguard from recursive water reflections.

if (s\_InsideWater)

{

return;

}

s\_InsideWater = true;

// Actual water rendering mode depends on both the current setting AND

// the hardware support. There's no point in rendering refraction textures

// if they won't be visible in the end.

m\_HardwareWaterSupport = FindHardwareWaterSupport();

WaterMode mode = GetWaterMode();

Camera reflectionCamera, refractionCamera;

CreateWaterObjects(cam, out reflectionCamera, out refractionCamera);

// find out the reflection plane: position and normal in world space

Vector3 pos = transform.position;

Vector3 normal = transform.up;

// Optionally disable pixel lights for reflection/refraction

int oldPixelLightCount = QualitySettings.pixelLightCount;

if (disablePixelLights)

{

QualitySettings.pixelLightCount = 0;

}

UpdateCameraModes(cam, reflectionCamera);

UpdateCameraModes(cam, refractionCamera);

// Render reflection if needed

if (mode >= WaterMode.Reflective)

{

// Reflect camera around reflection plane

float d = -Vector3.Dot(normal, pos) - clipPlaneOffset;

Vector4 reflectionPlane = new Vector4(normal.x, normal.y, normal.z, d);

Matrix4x4 reflection = Matrix4x4.zero;

CalculateReflectionMatrix(ref reflection, reflectionPlane);

Vector3 oldpos = cam.transform.position;

Vector3 newpos = reflection.MultiplyPoint(oldpos);

reflectionCamera.worldToCameraMatrix = cam.worldToCameraMatrix \* reflection;

// Setup oblique projection matrix so that near plane is our reflection

// plane. This way we clip everything below/above it for free.

Vector4 clipPlane = CameraSpacePlane(reflectionCamera, pos, normal, 1.0f);

reflectionCamera.projectionMatrix = cam.CalculateObliqueMatrix(clipPlane);

reflectionCamera.cullingMask = ~(1 << 4) & reflectLayers.value; // never render water layer

reflectionCamera.targetTexture = m\_ReflectionTexture;

GL.invertCulling = true;

reflectionCamera.transform.position = newpos;

Vector3 euler = cam.transform.eulerAngles;

reflectionCamera.transform.eulerAngles = new Vector3(-euler.x, euler.y, euler.z);

reflectionCamera.Render();

reflectionCamera.transform.position = oldpos;

GL.invertCulling = false;

GetComponent<Renderer>().sharedMaterial.SetTexture("\_ReflectionTex", m\_ReflectionTexture);

}

// Render refraction

if (mode >= WaterMode.Refractive)

{

refractionCamera.worldToCameraMatrix = cam.worldToCameraMatrix;

// Setup oblique projection matrix so that near plane is our reflection

// plane. This way we clip everything below/above it for free.

Vector4 clipPlane = CameraSpacePlane(refractionCamera, pos, normal, -1.0f);

refractionCamera.projectionMatrix = cam.CalculateObliqueMatrix(clipPlane);

refractionCamera.cullingMask = ~(1 << 4) & refractLayers.value; // never render water layer

refractionCamera.targetTexture = m\_RefractionTexture;

refractionCamera.transform.position = cam.transform.position;

refractionCamera.transform.rotation = cam.transform.rotation;

refractionCamera.Render();

GetComponent<Renderer>().sharedMaterial.SetTexture("\_RefractionTex", m\_RefractionTexture);

}

// Restore pixel light count

if (disablePixelLights)

{

QualitySettings.pixelLightCount = oldPixelLightCount;

}

// Setup shader keywords based on water mode

switch (mode)

{

case WaterMode.Simple:

Shader.EnableKeyword("WATER\_SIMPLE");

Shader.DisableKeyword("WATER\_REFLECTIVE");

Shader.DisableKeyword("WATER\_REFRACTIVE");

break;

case WaterMode.Reflective:

Shader.DisableKeyword("WATER\_SIMPLE");

Shader.EnableKeyword("WATER\_REFLECTIVE");

Shader.DisableKeyword("WATER\_REFRACTIVE");

break;

case WaterMode.Refractive:

Shader.DisableKeyword("WATER\_SIMPLE");

Shader.DisableKeyword("WATER\_REFLECTIVE");

Shader.EnableKeyword("WATER\_REFRACTIVE");

break;

}

s\_InsideWater = false;

}

// Cleanup all the objects we possibly have created

void OnDisable()

{

if (m\_ReflectionTexture)

{

DestroyImmediate(m\_ReflectionTexture);

m\_ReflectionTexture = null;

}

if (m\_RefractionTexture)

{

DestroyImmediate(m\_RefractionTexture);

m\_RefractionTexture = null;

}

foreach (var kvp in m\_ReflectionCameras)

{

DestroyImmediate((kvp.Value).gameObject);

}

m\_ReflectionCameras.Clear();

foreach (var kvp in m\_RefractionCameras)

{

DestroyImmediate((kvp.Value).gameObject);

}

m\_RefractionCameras.Clear();

}

// This just sets up some matrices in the material; for really

// old cards to make water texture scroll.

void Update()

{

if (!GetComponent<Renderer>())

{

return;

}

Material mat = GetComponent<Renderer>().sharedMaterial;

if (!mat)

{

return;

}

Vector4 waveSpeed = mat.GetVector("WaveSpeed");

float waveScale = mat.GetFloat("\_WaveScale");

Vector4 waveScale4 = new Vector4(waveScale, waveScale, waveScale \* 0.4f, waveScale \* 0.45f);

// Time since level load, and do intermediate calculations with doubles

double t = Time.timeSinceLevelLoad / 20.0;

Vector4 offsetClamped = new Vector4(

(float)Math.IEEERemainder(waveSpeed.x \* waveScale4.x \* t, 1.0),

(float)Math.IEEERemainder(waveSpeed.y \* waveScale4.y \* t, 1.0),

(float)Math.IEEERemainder(waveSpeed.z \* waveScale4.z \* t, 1.0),

(float)Math.IEEERemainder(waveSpeed.w \* waveScale4.w \* t, 1.0)

);

mat.SetVector("\_WaveOffset", offsetClamped);

mat.SetVector("\_WaveScale4", waveScale4);

}

void UpdateCameraModes(Camera src, Camera dest)

{

if (dest == null)

{

return;

}

// set water camera to clear the same way as current camera

dest.clearFlags = src.clearFlags;

dest.backgroundColor = src.backgroundColor;

if (src.clearFlags == CameraClearFlags.Skybox)

{

Skybox sky = src.GetComponent<Skybox>();

Skybox mysky = dest.GetComponent<Skybox>();

if (!sky || !sky.material)

{

mysky.enabled = false;

}

else

{

mysky.enabled = true;

mysky.material = sky.material;

}

}

// update other values to match current camera.

// even if we are supplying custom camera&projection matrices,

// some of values are used elsewhere (e.g. skybox uses far plane)

dest.farClipPlane = src.farClipPlane;

dest.nearClipPlane = src.nearClipPlane;

dest.orthographic = src.orthographic;

dest.fieldOfView = src.fieldOfView;

dest.aspect = src.aspect;

dest.orthographicSize = src.orthographicSize;

}

// On-demand create any objects we need for water

void CreateWaterObjects(Camera currentCamera, out Camera reflectionCamera, out Camera refractionCamera)

{

WaterMode mode = GetWaterMode();

reflectionCamera = null;

refractionCamera = null;

if (mode >= WaterMode.Reflective)

{

// Reflection render texture

if (!m\_ReflectionTexture || m\_OldReflectionTextureSize != textureSize)

{

if (m\_ReflectionTexture)

{

DestroyImmediate(m\_ReflectionTexture);

}

m\_ReflectionTexture = new RenderTexture(textureSize, textureSize, 16);

m\_ReflectionTexture.name = "\_\_WaterReflection" + GetInstanceID();

m\_ReflectionTexture.isPowerOfTwo = true;

m\_ReflectionTexture.hideFlags = HideFlags.DontSave;

m\_OldReflectionTextureSize = textureSize;

}

// Camera for reflection

m\_ReflectionCameras.TryGetValue(currentCamera, out reflectionCamera);

if (!reflectionCamera) // catch both not-in-dictionary and in-dictionary-but-deleted-GO

{

GameObject go = new GameObject("Water Refl Camera id" + GetInstanceID() + " for " + currentCamera.GetInstanceID(), typeof(Camera), typeof(Skybox));

reflectionCamera = go.GetComponent<Camera>();

reflectionCamera.enabled = false;

reflectionCamera.transform.position = transform.position;

reflectionCamera.transform.rotation = transform.rotation;

reflectionCamera.gameObject.AddComponent<FlareLayer>();

go.hideFlags = HideFlags.HideAndDontSave;

m\_ReflectionCameras[currentCamera] = reflectionCamera;

}

}

if (mode >= WaterMode.Refractive)

{

// Refraction render texture

if (!m\_RefractionTexture || m\_OldRefractionTextureSize != textureSize)

{

if (m\_RefractionTexture)

{

DestroyImmediate(m\_RefractionTexture);

}

m\_RefractionTexture = new RenderTexture(textureSize, textureSize, 16);

m\_RefractionTexture.name = "\_\_WaterRefraction" + GetInstanceID();

m\_RefractionTexture.isPowerOfTwo = true;

m\_RefractionTexture.hideFlags = HideFlags.DontSave;

m\_OldRefractionTextureSize = textureSize;

}

// Camera for refraction

m\_RefractionCameras.TryGetValue(currentCamera, out refractionCamera);

if (!refractionCamera) // catch both not-in-dictionary and in-dictionary-but-deleted-GO

{

GameObject go =

new GameObject("Water Refr Camera id" + GetInstanceID() + " for " + currentCamera.GetInstanceID(),

typeof(Camera), typeof(Skybox));

refractionCamera = go.GetComponent<Camera>();

refractionCamera.enabled = false;

refractionCamera.transform.position = transform.position;

refractionCamera.transform.rotation = transform.rotation;

refractionCamera.gameObject.AddComponent<FlareLayer>();

go.hideFlags = HideFlags.HideAndDontSave;

m\_RefractionCameras[currentCamera] = refractionCamera;

}

}

}

WaterMode GetWaterMode()

{

if (m\_HardwareWaterSupport < waterMode)

{

return m\_HardwareWaterSupport;

}

return waterMode;

}

WaterMode FindHardwareWaterSupport()

{

if (!SystemInfo.supportsRenderTextures || !GetComponent<Renderer>())

{

return WaterMode.Simple;

}

Material mat = GetComponent<Renderer>().sharedMaterial;

if (!mat)

{

return WaterMode.Simple;

}

string mode = mat.GetTag("WATERMODE", false);

if (mode == "Refractive")

{

return WaterMode.Refractive;

}

if (mode == "Reflective")

{

return WaterMode.Reflective;

}

return WaterMode.Simple;

}

// Given position/normal of the plane, calculates plane in camera space.

Vector4 CameraSpacePlane(Camera cam, Vector3 pos, Vector3 normal, float sideSign)

{

Vector3 offsetPos = pos + normal \* clipPlaneOffset;

Matrix4x4 m = cam.worldToCameraMatrix;

Vector3 cpos = m.MultiplyPoint(offsetPos);

Vector3 cnormal = m.MultiplyVector(normal).normalized \* sideSign;

return new Vector4(cnormal.x, cnormal.y, cnormal.z, -Vector3.Dot(cpos, cnormal));

}

// Calculates reflection matrix around the given plane

static void CalculateReflectionMatrix(ref Matrix4x4 reflectionMat, Vector4 plane)

{

reflectionMat.m00 = (1F - 2F \* plane[0] \* plane[0]);

reflectionMat.m01 = (- 2F \* plane[0] \* plane[1]);

reflectionMat.m02 = (- 2F \* plane[0] \* plane[2]);

reflectionMat.m03 = (- 2F \* plane[3] \* plane[0]);

reflectionMat.m10 = (- 2F \* plane[1] \* plane[0]);

reflectionMat.m11 = (1F - 2F \* plane[1] \* plane[1]);

reflectionMat.m12 = (- 2F \* plane[1] \* plane[2]);

reflectionMat.m13 = (- 2F \* plane[3] \* plane[1]);

reflectionMat.m20 = (- 2F \* plane[2] \* plane[0]);

reflectionMat.m21 = (- 2F \* plane[2] \* plane[1]);

reflectionMat.m22 = (1F - 2F \* plane[2] \* plane[2]);

reflectionMat.m23 = (- 2F \* plane[3] \* plane[2]);

reflectionMat.m30 = 0F;

reflectionMat.m31 = 0F;

reflectionMat.m32 = 0F;

reflectionMat.m33 = 1F;

}

}

}