Namespace Simulator.Sensors

{

Using System;

Using System.Collections;

Using System.Collections.Concurrent;

Using System.Collections.Generic;

Using System.Threading.Tasks;

Using Analysis;

Using Bridge;

Using Bridge.Data;

Using Plugins;

Using Postprocessing;

Using UI;

Using UnityEngine;

Using UnityEngine.Experimental.Rendering;

Using UnityEngine.Rendering;

Using UnityEngine.Rendering.HighDefinition;

Using Utilities;

Using LensDistortion = Utilities.LensDistortion;

[RequireComponent(typeof(Camera))]

Public abstract class CameraSensorBase: SensorBase

{

Private static readonly int ScreenSizeProperty = Shader.PropertyToID(“\_ScreenSize”);

[SensorParameter]

[Range(1, 1920)]

Public int Width = 1920;

[SensorParameter]

[Range(1, 1080)]

Public int Height = 1080;

[SensorParameter]

[Range(1, 100)]

Public int Frequency = 15;

[SensorParameter]

[Range(0, 100)]

Public int JpegQuality = 75;

[SensorParameter]

[Range(1.0f, 90.0f)]

Public float FieldOfView = 60.0f;

[SensorParameter]

[Range(0.01f, 1000.0f)]

Public float MinDistance = 0.1f;

[SensorParameter]

[Range(0.01f, 2000.0f)]

Public float MaxDistance = 2000.0f;

[SensorParameter]

Public bool Distorted = false;

[SensorParameter]

Public List<float> DistortionParameters;

[SensorParameter]

Public bool Fisheye = false;

[SensorParameter]

Public float Xi = 0.0f;

[SensorParameter]

Public string CameraInfoTopic;

Public List<PostProcessData> Postprocessing;

Public List<PostProcessData> LatePostprocessing;

BridgeInstance Bridge;

Publisher<ImageData> Publish;

Publisher<CameraInfoData> CameraInfoPublish;

Uint Sequence;

CameraInfoData CameraInfoData;

Const int MaxJpegSize = 4 \* 1024 \* 1024; // 4MB

Private float NextCaptureTime;

Private float PreviousCaptureTime = -1f;

Protected Camera sensorCamera;

Protected HDAdditionalCameraData hdAdditionalCameraData;

Protected Camera SensorCamera

{

Get

{

If (sensorCamera == null)

sensorCamera = GetComponent<Camera>();

return sensorCamera;

}

}

Protected HDAdditionalCameraData HDAdditionalCameraData

{

Get

{

If (hdAdditionalCameraData == null)

hdAdditionalCameraData = GetComponent<HDAdditionalCameraData>();

return hdAdditionalCameraData;

}

}

Protected SensorRenderTarget FinalRenderTarget => Distorted ? DistortedHandle : renderTarget;

Protected int ByteBufferSize => Width \* Height \* 4;

Protected RenderTextureReadWrite CameraTargetTextureReadWriteType = RenderTextureReadWrite.sRGB;

Public override SensorDistributionType DistributionType => SensorDistributionType.ClientOnly;

Public override float PerformanceLoad { get; } = 1.0f;

Protected SensorRenderTarget renderTarget;

Private LensDistortion LensDistortion;

Private SensorRenderTarget DistortedHandle;

Float FrustumWidth, FrustumHeight;

[SensorParameter]

Public int CubemapSize = 1024;

Protected readonly int faceMask = 1 << (int) CubemapFace.PositiveX | 1 << (int) CubemapFace.NegativeX |

1 << (int) CubemapFace.PositiveY | 1 << (int) CubemapFace.NegativeY |

1 << (int) CubemapFace.PositiveZ;

Private ConcurrentBag<byte[]> JpegOutput = new ConcurrentBag<byte[]>();

Private Queue<Task> Tasks = new Queue<Task>();

Private GpuReadbackPool<GpuReadbackData<byte>, byte> ReadbackPool;

Private int CurrentByteBufferSize;

#region FPSCalculation

[SensorParameter]

[AnalysisMeasurement(MeasurementType.Fps)]

Public float TargetFPS = 0f;

[SensorParameter]

Public float TargetFPSTime = 5f;

Private float LowFPSCalculatedTime = 0f;

Private int TotalFrames;

Private float AveDelta;

[AnalysisMeasurement(MeasurementType.Fps)]

Public float AveFPS = 0f;

[AnalysisMeasurement(MeasurementType.Fps)]

Public float LowestFPS = float.MaxValue;

Private bool LowFPS = false;

#endregion

Protected override void Initialize()

{

SensorCamera.enabled = false;

HDAdditionalCameraData.hasPersistentHistory = true;

ReadbackPool = new GpuReadbackPool<GpuReadbackData<byte>, byte>();

ReadbackPool.Initialize(ByteBufferSize, OnReadbackComplete);

CurrentByteBufferSize = ByteBufferSize;

}

Protected override void Deinitialize()

{

renderTarget?.Release();

DistortedHandle?.Release();

Task.WaitAll(Tasks.ToArray());

ReadbackPool?.Dispose();

}

Public override void OnBridgeSetup(BridgeInstance bridge)

{

Bridge = bridge;

Publish = bridge.AddPublisher<ImageData>(Topic);

If (!string.IsNullOrEmpty(CameraInfoTopic))

{

CameraInfoPublish = bridge.AddPublisher<CameraInfoData>(CameraInfoTopic);

}

}

Protected virtual void Update()

{

SensorCamera.fieldOfView = FieldOfView;

SensorCamera.nearClipPlane = MinDistance;

SensorCamera.farClipPlane = MaxDistance;

If (Distorted)

{

CheckDistortion();

}

While (Tasks.Count > 0 && Tasks.Peek().IsCompleted)

{

Tasks.Dequeue();

}

CheckTexture();

CheckCapture();

If (CameraInfoPublish != null)

{

InitCameraInfoData();

}

ReadbackPool.Process();

}

Void CheckDistortion()

{

If (DistortionParameters.Count == 0)

{

DistortionParameters = new List<float>(new float[4]);

}

Else if (DistortionParameters.Count != 4)

{

Throw new Exception(“Length of DistortionParameters is not 4.”);

}

If (Fisheye && !Distorted)

{

Throw new Exception(“Distorted must be true for fisheye lens.”);

}

If (LensDistortion == null)

{

LensDistortion = new LensDistortion();

LensDistortion.InitDistortion(DistortionParameters, FieldOfView, Xi, Width, Height);

}

Else if (!LensDistortion.IsValid(DistortionParameters, FieldOfView, Xi, Width, Height))

{

LensDistortion.InitDistortion(DistortionParameters, FieldOfView, Xi, Width, Height);

}

}

Void CheckTexture()

{

Var targetWidth = -1;

Var targetHeight = -1;

If (Distorted)

{

// Distorted + fisheye – render to cubemap, then distort to sensor-sized texture

If (Fisheye)

{

If (renderTarget != null && (!renderTarget.IsCube || !renderTarget.IsValid(CubemapSize, CubemapSize)))

{

renderTarget.Release();

renderTarget = null;

}

If (renderTarget == null)

{

renderTarget = SensorRenderTarget.CreateCube(CubemapSize, CubemapSize, faceMask);

SensorCamera.targetTexture = null;

}

}

// Distorted – render to larger texture, then distort to sensor-sized one

Else

{

targetWidth = LensDistortion.ActualWidth;

targetHeight = LensDistortion.ActualHeight;

}

If (DistortedHandle != null && !DistortedHandle.IsValid(Width, Height))

{

DistortedHandle.Release();

DistortedHandle = null;

}

If (DistortedHandle == null)

{

DistortedHandle = SensorRenderTarget.Create2D(Width, Height, GraphicsFormat.R8G8B8A8\_SRGB, true);

}

}

Else

{

// Undistorted – use only sensor-sized texture

If (DistortedHandle != null)

{

DistortedHandle.Release();

DistortedHandle = null;

}

targetWidth = Width;

targetHeight = Height;

}

If (targetWidth > 0)

{

If (renderTarget != null && !renderTarget.IsValid(targetWidth, targetHeight))

{

renderTarget.Release();

renderTarget = null;

}

If (renderTarget == null)

{

renderTarget = SensorRenderTarget.Create2D(targetWidth, targetHeight, GraphicsFormat.R8G8B8A8\_SRGB, !Distorted);

SensorCamera.targetTexture = renderTarget;

}

}

If (CurrentByteBufferSize != ByteBufferSize)

{

ReadbackPool.Resize(ByteBufferSize);

CurrentByteBufferSize = ByteBufferSize;

}

}

Protected void RenderCamera()

{

Var cmd = CommandBufferPool.Get();

Var hd = HDCamera.GetOrCreate(SensorCamera);

If (renderTarget.IsCube && !HDAdditionalCameraData.hasCustomRender)

{

// HDRP renders cubemap as multiple separate images, each with different exposure.

// Locking exposure will force it to use the same value for all faces, removing inconsistencies.

Hd.LockExposure();

SensorCamera.stereoSeparation = 0f;

SensorCamera.RenderToCubemap(renderTarget, faceMask, Camera.MonoOrStereoscopicEye.Left);

Hd.UnlockExposure();

}

Else

{

SensorCamera.Render();

}

If (Distorted)

{

If (Fisheye)

{

LensDistortion.UnifiedProjectionDistort(cmd, renderTarget, DistortedHandle);

}

Else

{

LensDistortion.PlumbBobDistort(cmd, renderTarget, DistortedHandle);

}

Cmd.SetGlobalVector(ScreenSizeProperty, new Vector4(Width, Height, 1.0f / Width, 1.0f / Height));

Var ctx = new PostProcessPassContext(cmd, hd, DistortedHandle);

SimulatorManager.Instance.Sensors.PostProcessSystem.RenderLateForSensor(ctx, this);

}

FinalRenderTarget.BlitTo2D(cmd, hd);

HDRPUtilities.ExecuteAndClearCommandBuffer(cmd);

CommandBufferPool.Release(cmd);

}

Void CheckCapture()

{

If (Time.time >= NextCaptureTime)

{

CalculateFPS();

RenderCamera();

ReadbackPool.StartReadback(FinalRenderTarget.UiTexture, 0, TextureFormat.RGBA32);

TotalFrames++;

PreviousCaptureTime = Time.time;

If (NextCaptureTime < Time.time – Time.deltaTime)

{

NextCaptureTime = Time.time + 1.0f / Frequency;

}

Else

{

NextCaptureTime += 1.0f / Frequency;

}

}

}

Private void OnReadbackComplete(GpuReadbackData<byte> data)

{

If (Bridge is {Status: Status.Connected})

{

Var imageData = new ImageData()

{

Name = Name,

Frame = Frame,

Width = Width,

Height = Height,

Sequence = Sequence,

};

If (!JpegOutput.TryTake(out imageData.Bytes))

imageData.Bytes = new byte[MaxJpegSize];

Tasks.Enqueue(Task.Run(() =>

{

imageData.Length = JpegEncoder.Encode(data.gpuData, Width, Height, 4, JpegQuality, imageData.Bytes);

if (imageData.Length > 0)

{

Var time = data.captureTime;

imageData.Time = time;

Publish(imageData);

If (CameraInfoData != null)

{

CameraInfoData.Name = Name;

CameraInfoData.Frame = Frame;

CameraInfoData.Time = time;

CameraInfoData.Sequence = Sequence;

CameraInfoPublish?.Invoke(CameraInfoData);

}

}

Else

Debug.Log(“Compressed image is empty, length = 0”);

JpegOutput.Add(imageData.Bytes);

}));

Sequence++;

}

}

Public virtual bool Save(string path, int quality, int compression)

{

CheckTexture();

RenderCamera();

Var readback = AsyncGPUReadback.Request(FinalRenderTarget.UiTexture, 0, TextureFormat.RGBA32);

Readback.WaitForCompletion();

If (readback.hasError)

{

Debug.Log(“Failed to read GPU texture”);

Return false;

}

Debug.Assert(readback.done);

Var data = readback.GetData<byte>();

Var bytes = new byte[16 \* 1024 \* 1024];

Int length;

Var ext = System.IO.Path.GetExtension(path).ToLower();

If (ext == “.png”)

{

Length = PngEncoder.Encode(data, Width, Height, 4, compression, bytes);

}

Else if (ext == “.jpeg” || ext == “.jpg”)

{

Length = JpegEncoder.Encode(data, Width, Height, 4, quality, bytes);

}

Else

{

Return false;

}

If (length > 0)

{

Try

{

Using (var file = System.IO.File.Create(path))

{

File.Write(bytes, 0, length);

}

Return true;

}

Catch

{

}

}

Return false;

}

Public override void OnVisualize(Visualizer visualizer)

{

Debug.Assert(visualizer != null);

Visualizer.UpdateRenderTexture(FinalRenderTarget.UiTexture, SensorCamera.aspect);

}

Public override void OnVisualizeToggle(bool state)

{

//

}

Private void CalculateFPS()

{

If (LowFPS)

Return;

If (PreviousCaptureTime < 0f)

Return;

Var delta = (Time.time – PreviousCaptureTime);

Var fps = 1.0f / delta;

LowestFPS = Mathf.Min(fps, LowestFPS);

AveDelta = (AveDelta \* (TotalFrames – 1) + delta) / (TotalFrames);

AveFPS = 1f / AveDelta;

If (fps < TargetFPS)

{

LowFPSCalculatedTime += delta;

If (LowFPSCalculatedTime >= TargetFPSTime)

{

LowFPSEvent(GetComponentInParent<IAgentController>().GTID, delta, fps);

LowFPSCalculatedTime = 0f;

LowFPS = true;

}

}

Else

{

LowFPSCalculatedTime = 0f;

}

}

Private void LowFPSEvent(uint id, float ms, float fps)

{

Hashtable data = new Hashtable

{

{ “Id”, id },

{ “Type”, “LowFPS” },

{ “Time”, SimulatorManager.Instance.GetSessionElapsedTimeSpan().ToString() },

{ “MS”, ms },

{ “FPS”, fps },

{ “Average FPS”, AveFPS },

{ “Status”, AnalysisManager.AnalysisStatusType.Failed },

};

SimulatorManager.Instance.AnalysisManager.AddEvent(data);

}

Private void InitCameraInfoData()

{

If (CameraInfoData != null)

{

Return;

}

Var vFOV = SensorCamera.fieldOfView \* Mathf.Deg2Rad;

Var hFOV = 2 \* Mathf.Atan(Mathf.Tan(SensorCamera.fieldOfView \* Mathf.Deg2Rad / 2) \* SensorCamera.aspect);

Double fx = (double)(SensorCamera.pixelWidth / (2.0f \* Mathf.Tan(0.5f \* hFOV)));

Double fy = (double)(SensorCamera.pixelHeight / (2.0f \* Mathf.Tan(0.5f \* vFOV)));

Double cx = SensorCamera.pixelWidth / 2.0f;

Double cy = SensorCamera.pixelHeight / 2.0f;

CameraInfoData = new CameraInfoData();

CameraInfoData.Width = SensorCamera.pixelWidth;

CameraInfoData.Height = SensorCamera.pixelHeight;

CameraInfoData.FocalLengthX = fx;

CameraInfoData.FocalLengthY = fx;

CameraInfoData.PrincipalPointX = cx;

CameraInfoData.PrincipalPointY = cy;

If (Distorted)

{

CameraInfoData.DistortionParameters = DistortionParameters.ToArray();

}

Else

{

CameraInfoData.DistortionParameters = new float[4] { 0.0f, 0.0f, 0.0f, 0.0f };

}

}

}

}