```
using System;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using Unity.Collections;
using Newtonsoft.Json;

public class FrameCapture : MonoBehaviour
{
[Header("AR Components")]
public ARCameraManager cameraManager;
public Camera arCamera;

[Header("Capture Settings")]
public string projectName = "NeilCapture";
public bool useRelativeTransforms = true;
public bool useDepthData = false;

private string projectDirectory;
private string imagesDirectory;
private string depthImagesDirectory;
private List<FrameData> frames = new List<FrameData>();
private Matrix4x4 firstFrameMatrixInverse;
private bool isFirstFrame = true;
private int frameId = 0;
private Manifest manifest;

void Start()
{
// Initialize directories
projectDirectory = Path.Combine(Application.persistentDataPath, projectName);
imagesDirectory = Path.Combine(projectDirectory, "images");
depthImagesDirectory = Path.Combine(projectDirectory, "depth_images");
```

```
Directory.CreateDirectory(imagesDirectory);
if (useDepthData)
{
Directory.CreateDirectory(depthImagesDirectory);
}

manifest = new Manifest();

cameraManager = GetComponent<ARCameraManager>();
if (cameraManager == null)
{
Debug.LogError("ARCameraManager component is missing");
return;
}
}

void Update()
{
// For demonstration, capture a frame when the user touches the screen
if (Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Began)
{
CaptureFrame();
}
}

void CaptureFrame()
{
// Capture camera transform
Matrix4x4 localToWorldMatrix = arCamera.transform.localToWorldMatrix;

// Adjust the coordinate system
Matrix4x4 adjustedMatrix = AdjustCoordinateSystem(localToWorldMatrix);

// For the first frame, store the inverse to normalize subsequent frames
if (isFirstFrame)
{
firstFrameMatrixInverse = adjustedMatrix.inverse;
isFirstFrame = false;
}
```

```
// Normalize the matrix relative to the first frame
Matrix4x4 relativeMatrix = useRelativeTransforms ? firstFrameMatrixInverse * adjustedMatrix :
adjustedMatrix;

// Convert the matrix to a list of array
List<List<float>> transformMatrix = MatrixToNestedList(relativeMatrix);



// Get camera intrinsics
XRCameraIntrinsics intrinsics;
if (cameraManager.TryGetIntrinsics(out intrinsics))
{
Debug.Log("Got camera intrinsics success");
}
else
{
Debug.LogWarning("Failed to get camera intrinsics.");
return;
}

// Capture image
Texture2D colorTexture = CaptureCameraImage();
if (colorTexture == null)
{
Debug.LogWarning("Failed to capture camera image.");
return;
}

// Save the image to disk
string imageFileName = $"frame_{frameId:D4}.png";
string imagePath = Path.Combine(imagesDirectory, imageFileName);
byte[] imageBytes = colorTexture.EncodeToPNG();
File.WriteAllBytes(imagePath, imageBytes);

// Capture depth image if enabled
string depthImagePath = null;
if (useDepthData) //false
```

```
{
Texture2D depthTexture = CaptureDepthImage();
if (depthTexture != null)
{
string depthImageFileName = $"frame_{frameId:D4}_depth.png";
depthImagePath = Path.Combine(depthImagesDirectory, depthImageFileName);
byte[] depthImageBytes = depthTexture.EncodeToPNG();
File.WriteAllBytes(depthImagePath, depthImageBytes);
}
}

// Create FrameData and add to frames list
FrameData frameData = new FrameData
{
file_path = $"images/{imageFileName}",
// depth_file_path = depthImagePath != null ?
$"depth_images/{Path.GetFileName(depthImagePath)}" : null,
transform_matrix = transformMatrix,
orignal_matrix = MatrixToNestedList(localToWorldMatrix),
timestamp = Time.timeAsDouble,
fl_x = intrinsics.focalLength.x,
fl_y = intrinsics.focalLength.y,
cx = intrinsics.principalPoint.x,
cy = intrinsics.principalPoint.y,
w = colorTexture.width,
h = colorTexture.height
};

frames.Add(frameData);
frameId++;

// Update manifest with first frame's data
if (manifest.w == 0)
{
manifest.w = frameData.w;
manifest.h = frameData.h;
manifest.fl_x = frameData.fl_x;
manifest.fl_y = frameData.fl_y;
manifest.cx = frameData.cx;
```

```
manifest.cy = frameData.cy;
}

Debug.Log($"Captured frame {frameId}");
}

Texture2D CaptureCameraImage()
{
if (cameraManager.TryAcquireLatestCpuImage(out XRCpuImage image))
{
// Choose RGBA format
var conversionParams = new XRCpuImage.ConversionParams
{
inputRect = new RectInt(0, 0, image.width, image.height),
outputDimensions = new Vector2Int(image.width, image.height),
outputFormat = TextureFormat.RGBA32,
transformation = XRCpuImage.Transformation.None
};

// Convert image
int size = image.GetConvertedDataSize(conversionParams);
var buffer = new NativeArray<byte>(size, Allocator.Temp);
image.Convert(conversionParams, buffer);
image.Dispose();

// Create Texture2D
Texture2D texture = new Texture2D(image.width, image.height, TextureFormat.RGBA32,
false);
texture.LoadRawTextureData(buffer);
texture.Apply();
buffer.Dispose();

return texture;
}
else
{
return null;
}
}
```

```
Texture2D CaptureDepthImage()
{
 AROcclusionManager occlusionManager = GetComponent<AROcclusionManager>();
if (occlusionManager != null &&
occlusionManager.TryAcquireEnvironmentDepthCpuImage(out XRCpuImage depthImage))
{
// Choose RFloat format
var conversionParams = new XRCpuImage.ConversionParams
{
inputRect = new RectInt(0, 0, depthImage.width, depthImage.height),
outputDimensions = new Vector2Int(depthImage.width, depthImage.height),
outputFormat = TextureFormat.RFloat,
transformation = XRCpuImage.Transformation.None
};

// Convert image
int size = depthImage.GetConvertedDataSize(conversionParams);
var buffer = new NativeArray<byte>(size, Allocator.Temp);
depthImage.Convert(conversionParams, buffer);
depthImage.Dispose();

// Create Texture2D
Texture2D texture = new Texture2D(depthImage.width, depthImage.height,
TextureFormat.RFloat, false);
texture.LoadRawTextureData(buffer);
texture.Apply();
buffer.Dispose();

return texture;
}
else
{
return null;
}
}

// Matrix4x4 AdjustCoordinateSystem(Matrix4x4 matrix)
// {
```

```
// // Flip Z-axis to convert from left-handed (Unity) to right-handed (NeRF)
// Matrix4x4 flipZ = Matrix4x4.Scale(new Vector3(1, 1, -1));
// return flipZ * matrix;
// }
// Matrix4x4 AdjustCoordinateSystem(Matrix4x4 matrix)
// {
// // Rotate -90 degrees around the X-axis to swap Y and Z axes
// Quaternion rotation = Quaternion.Euler(-90, 0, 0);
// Matrix4x4 rotationMatrix = Matrix4x4.Rotate(rotation);

// // Flip X-axis to convert from left-handed to right-handed coordinate system
// Matrix4x4 flipX = Matrix4x4.Scale(new Vector3(-1, 1, 1));

// // Apply transformations
// return flipX * rotationMatrix * matrix;
// }

// Matrix4x4 AdjustCoordinateSystem(Matrix4x4 matrix)
// {
// // Define the transformation matrix from Unity's coordinate system to NeRFStudio's
coordinate system

// // Rotate -90 degrees around the X-axis to change from Unity's Y-up to NeRFStudio's Z-up
// Quaternion rotationAdjustment = Quaternion.Euler(-90, 0, 0);
// Matrix4x4 rotationMatrix = Matrix4x4.Rotate(rotationAdjustment);

// // Flip the X-axis to convert from left-handed to right-handed coordinate system
// Matrix4x4 flipX = Matrix4x4.Scale(new Vector3(-1, 1, 1));

// // Combine the rotation and flip transformations
// Matrix4x4 transform = flipX * rotationMatrix;

// // Apply the transformation to the original matrix
// Matrix4x4 adjustedMatrix = transform * matrix;

// float scaleFactor = 10.0f; // Adjust as needed
// adjustedMatrix.m03 *= scaleFactor;
// adjustedMatrix.m13 *= scaleFactor;
// adjustedMatrix.m23 *= scaleFactor;
```

```
// return adjustedMatrix;
// }

Matrix4x4 AdjustCoordinateSystem(Matrix4x4 matrix)
{
// Define the flipZ matrix to flip the Z-axis
Matrix4x4 flipZ = Matrix4x4.Scale(new Vector3(1, 1, -1));

// Apply the coordinate system transformation
// Pre-multiply to adjust rotation component
// Post-multiply to adjust translation component
Matrix4x4 adjustedMatrix = flipZ * matrix * flipZ;
float scaleFactor = 10.0f; // Adjust as needed
adjustedMatrix.m03 *= scaleFactor;
adjustedMatrix.m13 *= scaleFactor;
adjustedMatrix.m23 *= scaleFactor;

return adjustedMatrix;
}



// Matrix4x4 AdjustCoordinateSystem(Matrix4x4 matrix)
// {
// // Extract rotation and translation from the original matrix
// Vector3 position = matrix.GetColumn(3);
// Quaternion rotation = Quaternion.LookRotation(matrix.GetColumn(2), matrix.GetColumn(1));

// // Define the coordinate transformation
// Quaternion rotationAdjustment = Quaternion.Euler(-90, 0, 0); // Rotate -90 degrees around
X-axis
// Vector3 scaleAdjustment = new Vector3(-1, 1, 1); // Flip X-axis to change handedness

// // Adjust the rotation
// rotation = rotationAdjustment * rotation;
// rotation = new Quaternion(rotation.x * scaleAdjustment.x, rotation.y * scaleAdjustment.y,
rotation.z * scaleAdjustment.z, rotation.w);
```

```
// // Adjust the position
// position = Vector3.Scale(position, scaleAdjustment);
// position = rotationAdjustment * position;

// // Reconstruct the adjusted matrix
// Matrix4x4 adjustedMatrix = Matrix4x4.TRS(position, rotation, Vector3.one);

// return adjustedMatrix;
// }



// Matrix4x4 AdjustCoordinateSystem(Matrix4x4 matrix)
// {
// // Extract rotation and translation from the original matrix
// Vector3 position = matrix.GetColumn(3);
// Quaternion rotation = Quaternion.LookRotation(matrix.GetColumn(2), matrix.GetColumn(1));

// // Define the coordinate transformation
// Quaternion rotationAdjustment = Quaternion.Euler(-90, 0, 0); // Rotate -90 degrees around X-axis
// Vector3 scaleAdjustment = new Vector3(-1, 1, 1); // Flip X-axis

// // Adjust the rotation
// rotation = rotationAdjustment * rotation;
// rotation = new Quaternion(rotation.x * scaleAdjustment.x, rotation.y * scaleAdjustment.y, rotation.z * scaleAdjustment.z, rotation.w);

// // Adjust the position
// position = Vector3.Scale(position, scaleAdjustment);
// position = rotationAdjustment * position;

// // Reconstruct the adjusted matrix
// Matrix4x4 adjustedMatrix = Matrix4x4.TRS(position, rotation, Vector3.one);

// return adjustedMatrix;
// }
```

```
List<List<float>> MatrixToNestedList(Matrix4x4 matrix)
{
List<List<float>> nestedList = new List<List<float>>();

for (int i = 0; i < 4; i++)
{
List<float> row = new List<float>
{
matrix[i, 0],
matrix[i, 1],
matrix[i, 2],
matrix[i, 3]
};
nestedList.Add(row);
}

return nestedList;
}



void OnApplicationQuit()
{
// Create the manifest when the application quits
CreateManifest();
}

void CreateManifest()
{
manifest.frames = frames;

// Serialize to JSON
string manifestJson = JsonConvert.SerializeObject(manifest, Formatting.Indented);
string manifestPath = Path.Combine(projectDirectory, "transforms.json");
File.WriteAllText(manifestPath, manifestJson);

Debug.Log($"Manifest saved to {manifestPath}");
}

// Optional: Visualize camera poses in the editor
```

```
// #if UNITY_EDITOR
// void OnDrawGizmos()
// {
// Gizmos.color = Color.red;
// foreach (var frame in frames)
// {
// Matrix4x4 matrix = ArrayToMatrix(frame.transform_matrix);

// // Extract position and rotation
// Vector3 position = matrix.GetColumn(3);
// Quaternion rotation = QuaternionFromMatrix(matrix);

// // Draw a small cube at the position
// Gizmos.DrawCube(position, Vector3.one * 0.05f);

// // Draw orientation axes
// Gizmos.color = Color.blue;
// Gizmos.DrawRay(position, rotation * Vector3.forward * 0.1f);
// Gizmos.color = Color.green;
// Gizmos.DrawRay(position, rotation * Vector3.up * 0.1f);
// Gizmos.color = Color.red;
// Gizmos.DrawRay(position, rotation * Vector3.right * 0.1f);
// }
// }

// Matrix4x4 ArrayToMatrix(float[] array)
// {
// Matrix4x4 matrix = new Matrix4x4();
// matrix.m00 = array[0]; matrix.m01 = array[1]; matrix.m02 = array[2]; matrix.m03 = array[3];
// matrix.m10 = array[4]; matrix.m11 = array[5]; matrix.m12 = array[6]; matrix.m13 = array[7];
// matrix.m20 = array[8]; matrix.m21 = array[9]; matrix.m22 = array[10]; matrix.m23 = array[11];
// matrix.m30 = array[12]; matrix.m31 = array[13]; matrix.m32 = array[14]; matrix.m33 =
array[15];
// return matrix;
// }

// Quaternion QuaternionFromMatrix(Matrix4x4 m)
// {
// return Quaternion.LookRotation(m.GetColumn(2), m.GetColumn(1));
```

```
// }
// #endif
[System.Serializable]
public class FrameData
{
public string file_path;
// public string depth_file_path;
public List<List<float>> transform_matrix;

public List<List<float>> orignal_matrix;
public double timestamp;
public float fl_x;
public float fl_y;
public float cx;
public float cy;
public int w;
public int h;
}


[System.Serializable]
public class Manifest
{
public int w;
public int h;
public float fl_x;
public float fl_y;
public float cx;
public float cy;
public List<FrameData> frames = new List<FrameData>();
}
}
```