using System;

using System.Collections.Generic;

using System.Runtime.InteropServices;

using UnityEngine;

using static DLLImportTest;

using System.Collections;

public class DLLImportTest : MonoBehaviour

{

public enum DHD\_STATUS\_ENUM

{

DHD\_STATUS\_POWER,

DHD\_STATUS\_CONNECTED,

DHD\_STATUS\_STARTED,

DHD\_STATUS\_RESET,

DHD\_STATUS\_IDLE,

DHD\_STATUS\_FORCE,

DHD\_STATUS\_BRAKE,

DHD\_STATUS\_TORQUE,

DHD\_STATUS\_WRIST\_DETECTED,

DHD\_STATUS\_ERROR,

DHD\_STATUS\_GRAVITY,

DHD\_STATUS\_TIMEGUARD,

DHD\_STATUS\_WRIST\_INIT,

DHD\_STATUS\_REDUNDANCY,

DHD\_STATUS\_FORCE\_OFF\_CAUSE,

DHD\_STATUS\_LOCKS,

DHD\_STATUS\_AXIS\_CHECKED

}

private bool forcesOn = false;

public double forceTest = 1.0;

public enum deviceStatus { DELTA\_OPEN, DELTA\_CLOSED };

public Transform targetTransform;

public deviceStatus DeviceStatus = deviceStatus.DELTA\_CLOSED;

public GameObject EndEffector;

public GameObject TargetSphere;

public float distanceThreshold = 1.8f;

//variables for the regular attractive force with applying resistive force

private float areaSize = 5.0f; // The size of the area in all three dimensions (5x5x5)

public float timeInsideSphereForForce = 5.0f; // Time user is inside the sphere before resistive force is applied

public float constantForceDuration = 5.0f; // time for constant resistive force

public float maxRandomForceDurationLimit = 10.0f; // Maximum duration for the random force

private float timeInsideTargetSphere = 0.0f;

private float timeSinceRandomForceStart = 0.0f;

private bool applyingRandomForce = false;

private bool applyingRegularForce = true;

private bool isRandomForceOn = false;

public float repelForceTest = 1.0f;

private float repellingForceMultiplier = 1.0f;

public float maxRepellingforceDistance = 2.0f; // The distance where repelling force is at its maximum

private bool repellingForceOn = false;

public Vector3 DhdPosition = Vector3.zero;

IntPtr defaultId = new IntPtr(1);

const int DHD\_MAX\_STATUS = 17;

int[] DHD\_STATUS\_RESULT = new int[DHD\_MAX\_STATUS];

public List<DHD\_STATUS\_ENUM> dhdStatus = new List<DHD\_STATUS\_ENUM>();

[DllImport("dhd64.dll")]

extern static int dhdOpen();

[DllImport("dhd64.dll")]

extern static int dhdStop(IntPtr id);

[DllImport("dhd64.dll")]

extern static int dhdClose(IntPtr id);

[DllImport("dhd64.dll")]

extern static int dhdGetStatus(int[] dhdStatus, IntPtr id);

[DllImport("dhd64.dll")]

extern static int dhdSetBrakes(int val, IntPtr id);

[DllImport("dhd64.dll")]

extern static IntPtr dhdErrorGetLastStr();

[DllImport("dhd64.dll")]

extern static void dhdSleep(double sec);

[DllImport("dhd64.dll")]

extern static int dhdGetPosition(ref double px, ref double py, ref double pz, IntPtr id);

[DllImport("dhd64.dll")]

extern static int dhdEnableForce(UIntPtr val, IntPtr id);

[DllImport("dhd64.dll")]

extern static int dhdSetStandardGravity(double g, IntPtr id);

[DllImport("dhd64.dll")]

extern static int dhdSetForce(double fx, double fy, double fz, IntPtr id);

// Start is called before the first frame update

void Start()

{

Debug.Log("DLLImportTest Start method called");

DhdOpen();

if (dhdEnableForce(new UIntPtr(1), defaultId) >= 0)

{

Debug.Log("Forces set to on");

forcesOn = true;

}

else

{

Debug.LogError("ERROR SETTING FORCES TO ON");

}

UpdateDHDStatus();

}

// Update is called once per frame

void Update()

{

Debug.Log("DLLImportTest Update method called");

if (Input.GetKeyDown(KeyCode.C))

{

DhdClose();

UpdateDHDStatus();

}

if (Input.GetKeyDown(KeyCode.O))

{

DhdOpen();

}

if (Input.GetKeyDown(KeyCode.A))

{

GetDHDPosition();

}

if (Input.GetKeyDown(KeyCode.B))

{

SetDHDBrake(false);

}

if (Input.GetKeyDown(KeyCode.V))

{

SetDHDBrake(true);

}

if (Input.GetKeyDown(KeyCode.S))

{

UpdateDHDStatus();

}

if (Input.GetKeyDown(KeyCode.E))

{

GetLastDHDError();

}

if (Input.GetKeyDown(KeyCode.G))

{

SetDHDGravity(0.0);

}

if (Input.GetKeyDown(KeyCode.F))

{

//ApplyForceTest(true, new Vector3((float)forceTest, (float)forceTest, (float)forceTest));

}

if (Input.GetKeyDown(KeyCode.R))

{

//ApplyForceTest(false, new Vector3((float)0.0f, (float)0.0f, (float)0.0f));

}

}

private void FixedUpdate()

{

if (DeviceStatus == deviceStatus.DELTA\_OPEN)

{

if (GetDHDPosition() >= 0) //check to see if information from haptic device successful

{

//EndEffector is unity object, this matches the position of haptic device (DhdPosition)

EndEffector.transform.position = DhdPosition;

}

// Method to scale the EndEffctors movment in unity

double px1 = 0, py1 = 0, pz1 = 0;

if (dhdGetPosition(ref px1, ref py1, ref pz1, defaultId) >= 0)

{

// Scale the haptic device's position

Vector3 scaledHapticPosition = new Vector3((float)(px1 \* 100), (float)(pz1 \* 100), (float)(py1 \* 100));

// Update the position of the EndEffector

EndEffector.transform.position = scaledHapticPosition;

}

// Applying attractive force towards TargetSphere

if (forcesOn)

{

//Applying attractive forces and turning off the forces when inside the sphere

/\*double px = 0, py = 0, pz = 0;

if (dhdGetPosition(ref px, ref py, ref pz, defaultId) >= 0)

{

Vector3 heading = TargetSphere.transform.position - EndEffector.transform.position;

float distance = heading.magnitude;

Vector3 direction = heading / distance;

Vector3 force = new Vector3((float)direction.x \* (float)forceTest, (float)direction.y \* (float)forceTest, (float)direction.z \* (float)forceTest);

if (distance < distanceThreshold)

{

ApplyForceToHapticDevice(Vector3.zero);

}

else

{

ApplyForceToHapticDevice(force);

}

}\*/

//Applying a repelling force to target sphere

double px = 0, py = 0, pz = 0;

if (dhdGetPosition(ref px, ref py, ref pz, defaultId) >= 0)

{

Vector3 heading = TargetSphere.transform.position - EndEffector.transform.position;

float distance = heading.magnitude;

Vector3 direction = heading.normalized;

// Calculate the repelling force based on the distance

float repellingForce = CalculateRepellingForce(distance);

// Apply the force in the opposite direction of the heading

Vector3 repellingForceVector = -direction \* repellingForce \* repellingForceMultiplier;

// Apply the repelling force

ApplyForceToHapticDevice(repellingForceVector);

}

//applying forces to the forceSpheres

// Check if targetSphere is within range of forceSphere

foreach (GameObject sphere in sphereList)

{

if (IsTargetInRangeOfForceSphere(sphere))

{

repellingForceOn = true; // Turn on the forces if targetSphere is within range

}

else

{

repellingForceOn = false; // Turn off the forces if targetSphere is not within range

}

repellingForceOn = true;

if (repellingForceOn)

{

// Calculate repelling force based on distance from forceSphere

//float repellingForce = CalculateRepellingForceFromForceSphere();

Vector3 heading = sphere.transform.position - EndEffector.transform.position;

float distance = heading.magnitude;

Vector3 direction = heading.normalized;

// Calculate the repelling force based on the distance

float repellingForce = CalculateRepellingForce(distance);

// Apply the force in the opposite direction of the forceSphere

Vector3 repellingForceVector = -direction \* repellingForce \* repellingForceMultiplier;

// Apply the repelling force to the haptic device

ApplyForceToHapticDevice(repellingForceVector);

}

}

}

}

}

private void ApplyForceToHapticDevice(Vector3 force)

{

dhdSetForce(force.x, force.z, force.y, defaultId);

}

private float CalculateRepellingForce(float distance)

{

float maxDistanceCalc= maxRepellingforceDistance \* distanceThreshold;

float clampedDistance = Mathf.Clamp(distance, distanceThreshold, maxDistanceCalc);

// Calculate the normalized force (0 to 1) based on distance

float normalizedForce = 1.0f - (clampedDistance - distanceThreshold) / (maxDistanceCalc - distanceThreshold);

// Adjust the normalized force to control the strength of the repelling force

float repellingForce = normalizedForce \* (float)repelForceTest;

return repellingForce;

}

private float CalculateRepellingForceFromForceSphere()

{

Vector3 heading = TargetSphere.transform.position - EndEffector.transform.position;

float distance = heading.magnitude;

return CalculateRepellingForce(distance);

}

public List<GameObject> sphereList = new List<GameObject>();

public void AddSphereTooList(GameObject sphere)

{

sphereList.Add(sphere);

}

public void DeleteForceSpheresFromList()

{

foreach (GameObject sphere in sphereList)

{

Destroy(sphere);

}

sphereList.Clear();

}

private bool IsTargetInRangeOfForceSphere(GameObject forceSphere)

{

float distanceToForceSphere = Vector3.Distance(TargetSphere.transform.position, forceSphere.transform.position);

return distanceToForceSphere <= 10.0f;

}

private bool IsEndEffectorNearForceSphere(GameObject forceSphere)

{

float distanceToEndEffector = Vector3.Distance(EndEffector.transform.position, forceSphere.transform.position);

return distanceToEndEffector <= maxRepellingforceDistance; // Set your desired threshold value here

}

private Vector3 GetForceSphereDirection(GameObject forceSphere)

{

return (forceSphere.transform.position - EndEffector.transform.position).normalized;

}

private void ProduceRandomForce()

{

// Generate random values within the specified area

float randomX = UnityEngine.Random.Range(-areaSize / 2.0f, areaSize / 2.0f);

float randomY = UnityEngine.Random.Range(-areaSize / 2.0f, areaSize / 2.0f);

float randomZ = UnityEngine.Random.Range(-areaSize / 2.0f, areaSize / 2.0f);

// Create a random direction vector within the area

Vector3 randomDirection = new Vector3(randomX, randomY, randomZ).normalized;

// Generate a force in random direction

Vector3 randomForce = randomDirection \* (float)forceTest \* 2;

// Apply the random force

ApplyForceToHapticDevice(randomForce);

// Start the coroutine to stop applying the random force after the duration

if (!applyingRandomForce)

{

StartCoroutine(ApplyRandomForceCoroutine());

}

applyingRegularForce = false;

}

IEnumerator ApplyRandomForceCoroutine()

{

applyingRandomForce = true;

//float forceDuration = UnityEngine.Random.Range(minRandomForceDuration, maxRandomForceDuration); //random duration

float forceDuration = constantForceDuration; // constant duration

while (timeSinceRandomForceStart < forceDuration)

{

// Wait for the next fixed update

yield return new WaitForFixedUpdate();

timeSinceRandomForceStart += Time.fixedDeltaTime;

}

// After the random force duration, stop applying the force

ApplyForceToHapticDevice(Vector3.zero);

applyingRandomForce = false;

applyingRegularForce = true;

timeSinceRandomForceStart = 0.0f;

}

// Method to check if the EndEffector is inside the TargetSphereArea

private bool IsInsideTargetSphereArea()

{

Vector3 heading = TargetSphere.transform.position - EndEffector.transform.position;

float distance = heading.magnitude;

return distance < distanceThreshold;

}

public int DhdOpen()

{

// open the first available device

if (dhdOpen() < 0)

{

DeviceStatus = deviceStatus.DELTA\_CLOSED;

IntPtr intPtr = dhdErrorGetLastStr();

string myErrorString = Marshal.PtrToStringAnsi(intPtr);

Debug.LogError(String.Format("error: cannot open device {0}\n", myErrorString));

dhdSleep(2.0);

return -1;

}

else

{

Debug.Log(String.Format("Device Succesfully Opened"));

DeviceStatus = deviceStatus.DELTA\_OPEN;

UpdateDHDStatus();

return 0;

}

}

private void OnDestroy()

{

DhdClose();

}

public int DhdClose()

{

if (dhdClose(defaultId) < 0)

{

IntPtr intPtr = dhdErrorGetLastStr();

string myErrorString = Marshal.PtrToStringAnsi(intPtr);

Debug.LogError(String.Format("error: Failed to stop device {0}\n", myErrorString));

return -1;

}

else

{

DeviceStatus = deviceStatus.DELTA\_CLOSED;

Debug.Log(String.Format("Device Closed!"));

return 0;

}

}

private int UpdateDHDStatus()

{

if (dhdGetStatus(DHD\_STATUS\_RESULT, defaultId) < 0)

{

dhdStatus.Clear();

return -1;

}

else

{

//Debug.Log(String.Format("Succesfully Got Status"));

dhdStatus.Clear();

for (int i = 0; i < DHD\_MAX\_STATUS; i++)

{

int currentResult = DHD\_STATUS\_RESULT[i];

if (currentResult > 0)

{

dhdStatus.Add((DHD\_STATUS\_ENUM)i);

}

}

return 0;

}

}

public int GetDHDPosition()

{

double px = 0;

double py = 0;

double pz = 0;

if (dhdGetPosition(ref px, ref py, ref pz, defaultId) < 0)

{

Debug.LogError("ERROR GETTING POSITION");

return -1;

}

else

{

//Debug.Log(String.Format("{0:0.000},{1:0.000},{2:0.000}", px, py, pz));

DhdPosition = new Vector3((float)px, (float)pz, (float)py);

return 0;

}

}

public int SetDHDBrake(bool brakeOn)

{

int success = -1;

if (brakeOn)

{

dhdEnableForce(new UIntPtr(1), defaultId);

if (dhdSetBrakes(1, defaultId) < 0)

{

Debug.LogError("ERROR TURNING ON BRAKE");

success = -1;

}

else

{

success = 0;

}

}

else

{

dhdEnableForce(new UIntPtr(0), defaultId);

if (dhdSetBrakes(0, defaultId) < 0)

{

Debug.LogError("ERROR TURNING OFF BRAKE");

success = -1;

}

else

{

success = 0;

}

}

UpdateDHDStatus();

return success;

}

public void GetLastDHDError()

{

IntPtr intPtr = dhdErrorGetLastStr();

string myErrorString = Marshal.PtrToStringAnsi(intPtr);

Debug.LogError(String.Format("Last Error: {0}\n", myErrorString));

}

public void SetDHDGravity(double g)

{

dhdSetStandardGravity(g, defaultId);

}

}