#include <stdio.h>

#include <chrono>

#include <winsock2.h>

#include "chai3d.h"

#include <GLFW/glfw3.h>

#pragma comment(lib, "ws2\_32.lib")

//////////////////////////////////////////////////

using namespace chai3d;

using namespace std;

// Device Param.

#define DOF 3

double pos\_curr[DOF] = { 0, 0, 0 };

double pos\_prev[DOF] = { 0, 0, 0 };

double pos\_recv[DOF] = { 0, 0, 0 };

double Fd[DOF] = { 0, 0, 0 };

double dFe[DOF] = { 0, 0, 0 };

double Xd[DOF];

double DXd[DOF];

double XdP[DOF];

bool nProjection = true;

double FOR\_Mag;

double ProjFOR[3], UnitV[3];// EBA\_FOR;

double InnerProduct;

// EBA Parameters //

double uBETA[3], BETA[3], Bmax[3], Bmin[3];

double c1 = 50;

double c2 = 100;

double Th, Tc;

// Flag Parameters //

bool EnableEBA = true;

// UDP Param.

#define BUFSIZE 2048

#define SERVER\_IP "192.168.4.3" // "127.0.0.1" // "111.68.110.189" "203.237.55.14"

#define UDP\_PORT 4000

#define SENDDATA\_NUM 3

#define READDATA\_NUM 3

string Server = SERVER\_IP;

string Server2;

int retval;

WSADATA wsa;

int TAP;

double avg\_delay;

SOCKET sock;

// µ¥ÀÌÅÍ Åë½Å¿¡ »ç¿ëÇÒ º¯¼ö

SOCKADDR\_IN serveraddr;

SOCKADDR\_IN clientaddr;

int addrlen;

char sbuf[BUFSIZE + 1], rbuf[BUFSIZE + 1];

float SendData[SENDDATA\_NUM], ReadData[READDATA\_NUM];

bool isFirstRecv = false;

int timeUDPR;

int timeUDPS;

double delay = 0;

double delaym = 0;

int countd = 0;

struct Timer

{

chrono::time\_point<std::chrono::steady\_clock>start, end;

chrono::duration<float>duration;

Timer()

{

start = std::chrono::high\_resolution\_clock::now();

}

~Timer()

{

end = std::chrono::high\_resolution\_clock::now();

duration = end - start;

float ms = duration.count() \* 1000.0f;

timeUDPR = ms;

//cout << "Timer took" << ms << "ms" << endl;

}

};

struct Timer2

{

chrono::time\_point<std::chrono::steady\_clock>start, end;

chrono::duration<float>duration;

Timer2()

{

start = std::chrono::high\_resolution\_clock::now();

}

~Timer2()

{

end = std::chrono::high\_resolution\_clock::now();

duration = end - start;

float ms = duration.count() \* 1000.0f;

timeUDPS = ms;

//cout << "Timer took" << ms << "ms" << endl;

}

};

// Delay Param.

#define DELAY 1

#define DELAYDATA\_NUM 3

double ClientData[DELAYDATA\_NUM];

double DelayDataBuf[DELAYDATA\_NUM][DELAY];

double DelayData[DELAYDATA\_NUM];

// GUI - GENERAL SETTINGS

cStereoMode stereoMode = C\_STEREO\_DISABLED;

// fullscreen mode

bool fullscreen = false;

// mirrored display

bool mirroredDisplay = false;

////////////////////////////////////////////////////////////////////////////////

// GUI - DECLARED VARIABLES

////////////////////////////////////////////////////////////////////////////////

// a world that contains all objects of the virtual environment

cWorld\* world;

// a camera to render the world in the window display

cCamera\* camera;

// a light source to illuminate the objects in the world

cDirectionalLight\* light;

// a haptic device handler

cHapticDeviceHandler\* handler;

// a pointer to the current haptic device

cGenericHapticDevicePtr hapticDevice;

// a label to display the haptic device model

cLabel\* labelHapticDeviceModel;

// a label to display the position [m] of the haptic device

cLabel\* labelHapticDevicePosition;

// a global variable to store the position [m] of the haptic device

cVector3d hapticDevicePosition;

// a global variable to store the velocity [m/s] of the haptic device

cVector3d hapticDeviceVelocity;

// a font for rendering text

cFontPtr font;

// a label to display the rate [Hz] at which the simulation is running

cLabel\* labelRates;

cLabel\* labelMax;

cLabel\* labelMin;

cLabel\* labelXd;

cLabel\* labelXd\_Prev;

cLabel\* labelX\_Recv;

cLabel\* labelFd;

cLabel\* labeldFe;

cLabel\* labelIP;

cLabel\* labelDelay;

cLabel\* labelc1;

cLabel\* labelc2;

cLabel\* labeldelay2;

cLabel\* labeldelay3;

cLabel\*labelProjFOR;

// a small sphere (cursor) representing the haptic device

cShapeTorus\* cursor;

// a line representing the velocity of the haptic device

cShapeLine\* velocity;

// a scope to monitor position values of haptic device

cScope\* scope;

cScope\* scope1;

cLevel\* levelVelocity;

// a flag for using force field (ON/OFF)

bool useForceField = true;

// a flag to indicate if the haptic simulation currently running

bool simulationRunning = false;

// a flag to indicate if the haptic simulation has terminated

bool simulationFinished = true;

// a frequency counter to measure the simulation graphic rate

cFrequencyCounter freqCounterGraphics;

// a frequency counter to measure the simulation haptic rate

cFrequencyCounter freqCounterHaptics;

// haptic thread

cThread\* hapticsThread;

// a handle to window display context

GLFWwindow\* window = NULL;

// current width of window

int width = 0;

// current height of window

int height = 0;

// swap interval for the display context (vertical synchronization)

int swapInterval = 1;

////////////////////////////////////////////////////////////////////////////////

// GUI - DECLARED FUNCTIONS

////////////////////////////////////////////////////////////////////////////////

// callback when the window display is resized

void windowSizeCallback(GLFWwindow\* a\_window, int a\_width, int a\_height);

// callback when an error GLFW occurs

void errorCallback(int error, const char\* a\_description);

// callback when a key is pressed

void keyCallback(GLFWwindow\* a\_window, int a\_key, int a\_scancode, int a\_action, int a\_mods);

// this function renders the scene

void updateGraphics(void);

// this function contains the main haptics simulation loop

void updateHaptics(void);

// this function closes the application

void close(void);

void InitUDP(void);

void modifiedEBA(void);

void CloseUDP();

void UDPSendData(float sData[SENDDATA\_NUM]);

void UDPReadData();

void err\_quit(const char\* msg);

void err\_display(const char\* msg);

/////////////////////////////////////////////////////////////

int main(int argc, char\* argv[])

{

InitUDP();

////////////////////////////////////////////////////////////////////////////////

// INITIALIZATION

////////////////////////////////////////////////////////////////////////////////

cout << endl;

cout << "-----------------------------------" << endl;

cout << "Dr. Riazuddin Teleoperation (SERVER)" << endl;

cout << "-----------------------------------" << endl << endl << endl;

cout << "Keyboard Options:" << endl << endl;

cout << "[1] - Enable/Disable potential field" << endl;

cout << "[f] - toggle fullscreen" << endl;

cout << "[M] - Mirrored Display" << endl;

cout << "[q] - Exit application" << endl;

cout << " " << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

////////////////////////////////////////////////////////////////////////////////

// OPEN GL - WINDOW DISPLAY

////////////////////////////////////////////////////////////////////////////////

// initialize GLFW library

if (!glfwInit())

{

cout << "failed initialization" << endl;

cSleepMs(1000);

return 1;

}

// set error callback

glfwSetErrorCallback(errorCallback);

// compute desired size of window

const GLFWvidmode\* mode = glfwGetVideoMode(glfwGetPrimaryMonitor());

int w = 0.8 \* mode->height;

int h = 0.5 \* mode->height;

int x = 0.5 \* (mode->width - w);

int y = 0.5 \* (mode->height - h);

// set OpenGL version

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 2);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 1);

// set active stereo mode

if (stereoMode == C\_STEREO\_ACTIVE)

{

glfwWindowHint(GLFW\_STEREO, GL\_TRUE);

}

else

{

glfwWindowHint(GLFW\_STEREO, GL\_FALSE);

}

// create display context

window = glfwCreateWindow(w, h, "Dr. Riazuddin Teleoperation (SERVER)", NULL, NULL);

if (!window)

{

cout << "failed to create window" << endl;

cSleepMs(10);

glfwTerminate();

return 1;

}

// get width and height of window

glfwGetWindowSize(window, &width, &height);

// set position of window

glfwSetWindowPos(window, x, y);

// set key callback

glfwSetKeyCallback(window, keyCallback);

// set resize callback

glfwSetWindowSizeCallback(window, windowSizeCallback);

// set current display context

glfwMakeContextCurrent(window);

// sets the swap interval for the current display context

glfwSwapInterval(swapInterval);

#ifdef GLEW\_VERSION

// initialize GLEW library

if (glewInit() != GLEW\_OK)

{

cout << "failed to initialize GLEW library" << endl;

glfwTerminate();

return 1;

}

#endif

////////////////////////////////////////////////////////////////////////////////

// WORLD - CAMERA - LIGHTING

////////////////////////////////////////////////////////////////////////////////

// create a new world.

world = new cWorld();

// set the background color of the environment

world->m\_backgroundColor.setBlack();

// create a camera and insert it into the virtual world

camera = new cCamera(world);

world->addChild(camera);

// position and orient the camera

camera->set(cVector3d(0.5, 0.0, 0.0), // camera position (eye)

cVector3d(0.0, 0.0, 0.0), // look at position (target)

cVector3d(0.0, 0.0, 1.0)); // direction of the (up) vector

// set the near and far clipping planes of the camera

camera->setClippingPlanes(0.02, 10.0);

// set stereo mode

camera->setStereoMode(stereoMode);

// set stereo eye separation and focal length (applies only if stereo is enabled)

camera->setStereoEyeSeparation(0.005);

camera->setStereoFocalLength(0.5);

// set vertical mirrored display mode

camera->setMirrorVertical(mirroredDisplay);

// create a directional light source

light = new cDirectionalLight(world);

// insert light source inside world

world->addChild(light);

// enable light source

light->setEnabled(true);

// define direction of light beam

light->setDir(-1.0, 0.0, 0.0);

// create a sphere (cursor) to represent the haptic device

cursor = new cShapeTorus(0.02, 0.03);

// insert cursor inside world

world->addChild(cursor);

// create small line to illustrate the velocity of the haptic device

velocity = new cShapeLine(cVector3d(0, 0, 0), cVector3d(0, 0, 0));

// insert line inside world

world->addChild(velocity);

////////////////////////////////////////////////////////////////////////////////

// HAPTIC DEVICE

////////////////////////////////////////////////////////////////////////////////

// create a haptic device handler

handler = new cHapticDeviceHandler();

// get a handle to the first haptic device

handler->getDevice(hapticDevice, 0);

// open a connection with the haptic device

hapticDevice->open();

// retrieve information about the current haptic device

cHapticDeviceInfo info = hapticDevice->getSpecifications();

// if the device has a gripper, enable the gripper to behave like a user switch

hapticDevice->setEnableGripperUserSwitch(true);

////////////////////////////////////////////////////////////////////////////////

// WIDGETS

////////////////////////////////////////////////////////////////////////////////

// create a font

font = NEW\_CFONTCALIBRI28();

// create a label to display the haptic device model

labelHapticDeviceModel = new cLabel(font);

camera->m\_frontLayer->addChild(labelHapticDeviceModel);

labelHapticDeviceModel->setText(info.m\_modelName);

labelHapticDeviceModel->m\_fontColor.setBlue();

// create a label to display the position of haptic device

labelHapticDevicePosition = new cLabel(font);

camera->m\_frontLayer->addChild(labelHapticDevicePosition);

// create a label to display the haptic and graphic rate of the simulation

labelRates = new cLabel(font);

camera->m\_frontLayer->addChild(labelRates);

// create a scope to plot haptic device position data

scope = new cScope();

camera->m\_frontLayer->addChild(scope);

scope->setLocalPos(100, 60);

scope->setRange(-0.1, 0.1);

//scope->createEffectVibration();

//scope->createEffectViscosity();

scope->setSignalEnabled(true, true, true, true);

scope->setTransparencyLevel(0.6);

// create a level to display velocity data

levelVelocity = new cLevel();

camera->m\_frontLayer->addChild(levelVelocity);

levelVelocity->setLocalPos(60, 60);

levelVelocity->setRange(0.0, 200);

levelVelocity->setWidth(40);

levelVelocity->setNumIncrements(50);

levelVelocity->setSingleIncrementDisplay(false);

levelVelocity->setTransparencyLevel(0.7);

//double getRangeMax(delay);

Server2 = Server.substr(0, 14);

//cout << Server2;

if (Server2 == "192.168.4.3") {

//Range = 15;

labelMax = new cLabel(font);

camera->m\_frontLayer->addChild(labelMax);

labelMax->setText("Teleoperation running over WAN");

labelMax->setLocalPos(20, height - 220);

labelMax->m\_fontColor.setWhite();

}

else if (Server2 == "172") {

//Range = 20;

labelMax = new cLabel(font);

camera->m\_frontLayer->addChild(labelMax);

labelMax->setText("Teleoperation running over LAN");

labelMax->setLocalPos(20, height - 220);

labelMax->m\_fontColor.setWhite();

}

else if (Server2 == "127") {

//Range = 20;

labelMax = new cLabel(font);

camera->m\_frontLayer->addChild(labelMax);

labelMax->setText("Teleoperation running over LAN");

labelMax->setLocalPos(20, height - 220);

labelMax->m\_fontColor.setWhite();

}

else {

//Range = 700;

labelMax = new cLabel(font);

camera->m\_frontLayer->addChild(labelMax);

labelMax->setText("Teleoperation running over LAN");

labelMax->setLocalPos(20, height - 220);

}

//position scope ranges

labelMax = new cLabel(font);

camera->m\_frontLayer->addChild(labelMax);

labelMax->setText("-0.1");

labelMax->setLocalPos(110, 60);

labelMin = new cLabel(font);

camera->m\_frontLayer->addChild(labelMin);

labelMin->setText(" 0.1");

labelMin->setLocalPos(110, 220);

labelXd\_Prev = new cLabel(font);

labelXd = new cLabel(font);

labelX\_Recv = new cLabel(font);

labelFd = new cLabel(font);

labeldFe = new cLabel(font);

labelIP = new cLabel(font);

labelProjFOR = new cLabel(font);

labelDelay = new cLabel(font);

labelc1 = new cLabel(font);

labelc2 = new cLabel(font);

camera->m\_frontLayer->addChild(labelXd\_Prev);

camera->m\_frontLayer->addChild(labelXd);

camera->m\_frontLayer->addChild(labelX\_Recv);

camera->m\_frontLayer->addChild(labelFd);

camera->m\_frontLayer->addChild(labeldFe);

camera->m\_frontLayer->addChild(labelIP);

camera->m\_frontLayer->addChild(labelProjFOR);

camera->m\_frontLayer->addChild(labelDelay);

camera->m\_frontLayer->addChild(labelc1);

camera->m\_frontLayer->addChild(labelc2);

labelIP->setText("Server IP: ");

labelIP->setLocalPos(20, height - 60);

labelIP->m\_fontColor.setBlueLight();

labelXd\_Prev->setText("Prev Xd: ");

labelXd\_Prev->setLocalPos(20, height - 80);

labelXd\_Prev->m\_fontColor.setGreenLight();

labelXd->setText("Curr Xd: ");

labelXd->setLocalPos(20, height - 100);

labelXd->m\_fontColor.setOrange();

labelX\_Recv->setText("Recv: ");

labelX\_Recv->setLocalPos(20, height - 120);

labelX\_Recv->m\_fontColor.setGreenDark();

labelFd->setText("Fd: ");

labelFd->setLocalPos(20, height - 140);

labelFd->m\_fontColor.setRedCrimson();

labeldFe->setText("dFe: ");

labeldFe->setLocalPos(20, height - 160);

labeldFe->m\_fontColor.setWhite();

labelProjFOR->setText("ProjFOR: ");

labelProjFOR->setLocalPos(20, height - 180);

labelProjFOR->m\_fontColor.setPink();

labelRates->setText("Frequency: ");

labelRates->setLocalPos(100, height - 160);

labelRates->m\_fontColor.setWhite();

labelDelay->setText("Delay: ");

labelDelay->setLocalPos(width -845 , height - 200);

labelDelay->m\_fontColor.setGray();

labelc1->setText("c1 : ");

labelc1->setLocalPos(width - 845, height - 240);

labelc1->m\_fontColor.setGray();

labelc2->setText("c2 : ");

labelc2->setLocalPos(width - 845, height - 260);

labelc2->m\_fontColor.setGray();

////////////////////////////////////////////////////////////////////////////////

// START SIMULATION

////////////////////////////////////////////////////////////////////////////////

// create a thread which starts the main haptics rendering loop

hapticsThread = new cThread();

hapticsThread->start(updateHaptics, CTHREAD\_PRIORITY\_HAPTICS);

// setup callback when application exits

atexit(close);

////////////////////////////////////////////////////////////////////////////////

// MAIN GRAPHIC LOOP

////////////////////////////////////////////////////////////////////////////////

// call window size callback at initialization

windowSizeCallback(window, width, height);

while (!glfwWindowShouldClose(window))

{

// get width and height of window

glfwGetWindowSize(window, &width, &height);

// render graphics

updateGraphics();

// swap buffers

glfwSwapBuffers(window);

// process events

glfwPollEvents();

// signal frequency counter

freqCounterGraphics.signal(1);

}

// close window

glfwDestroyWindow(window);

// terminate GLFW library

glfwTerminate();

// exit

return (0);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void updateGraphics(void)

{

/////////////////////////////////////////////////////////////////////

// UPDATE WIDGETS

/////////////////////////////////////////////////////////////////////

// update position data

labelHapticDevicePosition->setText(hapticDevicePosition.str(3));

// update haptic and graphic rate data

labelRates->setText(cStr(freqCounterGraphics.getFrequency(), 0) + " Hz / " +

cStr(freqCounterHaptics.getFrequency(), 0) + " Hz");

// update position of label

labelRates->setLocalPos((int)(0.5 \* (width - labelRates->getWidth())), 15);

// update information to scope

scope->setSignalValues(hapticDevicePosition.x(), hapticDevicePosition.y(), hapticDevicePosition.z());

levelVelocity->setValue(TAP);

labelIP->setText("Client IP: " + string(inet\_ntoa(clientaddr.sin\_addr)) + ":" + cStr(ntohs(clientaddr.sin\_port)));

labelXd\_Prev->setText("Prev Xd: " + cStr(pos\_prev[0], 3U) + " " + cStr(pos\_prev[1], 3U) + " " + cStr(pos\_prev[2], 3U));

labelXd->setText("Curr Xd: " + cStr(pos\_curr[0], 3U) + " " + cStr(pos\_curr[1], 3U) + " " + cStr(pos\_curr[2], 3U));

labelX\_Recv->setText("Recv X: " + cStr(pos\_recv[0], 3U) + " " + cStr(pos\_recv[1], 3U) + " " + cStr(pos\_recv[2], 3U));

labelFd->setText("Fd: " + cStr(Fd[0], 3U) + " " + cStr(Fd[1], 3U) + " " + cStr(Fd[2], 3U));

labeldFe->setText("dFe: " + cStr(dFe[0], 3U) + " " + cStr(dFe[1], 3U) + " " + cStr(dFe[2], 3U));

labelProjFOR->setText("Fpr: " + cStr(ProjFOR[0], 3U) + " " + cStr(ProjFOR[1], 3U) + " " + cStr(ProjFOR[2], 3U));

labelDelay->setText("Delay: " + cStr(abs(delay), 3U) + "ms");

labelc1->setText("c1: " + cStr(abs(c1), 3U) );

labelc2->setText("c2: " + cStr(abs(c2), 3U) );

/////////////////////////////////////////////////////////////////////

// RENDER SCENE

/////////////////////////////////////////////////////////////////////

// update shadow maps (if any)

world->updateShadowMaps(false, mirroredDisplay);

// render world

camera->renderView(width, height);

// wait until all OpenGL commands are completed

glFinish();

// check for any OpenGL errors

GLenum err;

err = glGetError();

if (err != GL\_NO\_ERROR) cout << "Error: %s\n" << gluErrorString(err);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void updateHaptics(void)

{

// simulation in now running

simulationRunning = true;

simulationFinished = false;

// main haptic simulation loop

while (simulationRunning)

{

/////////////////////////////////////////////////////////////////////

// READ HAPTIC DEVICE

/////////////////////////////////////////////////////////////////////

// read position

cVector3d position;

hapticDevice->getPosition(position);

//cout <<"position " <<position<<endl;

// read linear velocity

cVector3d linearVelocity;

hapticDevice->getLinearVelocity(linearVelocity);

// read user-switch status (button 0)

bool button0, button1, button2, button3;

button0 = false;

button1 = false;

button2 = false;

button3 = false;

hapticDevice->getUserSwitch(0, button0);

hapticDevice->getUserSwitch(1, button1);

hapticDevice->getUserSwitch(2, button2);

hapticDevice->getUserSwitch(3, button3);

/////////////////////////////////////////////////////////////////////

// UPDATE 3D CURSOR MODEL

/////////////////////////////////////////////////////////////////////

// update arrow

velocity->m\_pointA = position;

velocity->m\_pointB = cAdd(position, linearVelocity);

// update position and orientation of cursor

cursor->setLocalPos(position);

// adjust the color of the cursor according to the status of

// the user-switch (ON = TRUE / OFF = FALSE)

if (button0)

{

cursor->m\_material->setGreenMediumAquamarine();

for (int G = 0; G >= 0; G++)

{

TAP = 0;

c2 = 50;

break;

}

}

else if (button1)

{

cursor->m\_material->setYellowGold();

for (int G = 0; G >= 0; G++)

{

TAP = TAP - 1;

//cout << TAP;

if (TAP == -1 && -2 && -3 && -4 && -5 && -6 && -7 && -8 && -9)

{

TAP = 0;

}

break;

}

}

else if (button2)

{

cursor->m\_material->setOrangeCoral();

//for (int C = 0; C >= 0; C++)

//{

// c2 = c2 + 5;

// //cout << TAP;

// break;

//}

}

else if (button3)

{

cursor->m\_material->setPurpleLavender();

for (int C = 0; C >= 0; C++)

{

TAP = TAP + 1;

//cout << TAP;

break;

}

}

else

{

cursor->m\_material->setBlueRoyal();

}

// update global variable for graphic display update

hapticDevicePosition = position;

hapticDeviceVelocity = linearVelocity;

/////////////////////////////////////////////////////////////////////

// COMPUTE AND APPLY FORCES

/////////////////////////////////////////////////////////////////////

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

{

pos\_curr[i] = hapticDevicePosition(i);

}

// µ¥ÀÌÅÍ ¹Þ±â

UDPReadData();

Fd[0] = -ReadData[0];

Fd[1] = -ReadData[1];

Fd[2] = -ReadData[2];

SendData[0] = pos\_curr[0];

SendData[1] = pos\_curr[1];

SendData[2] = pos\_curr[2];

UDPSendData(SendData);

cVector3d force(0, 0, 0);

double gripperForce = 0.0;

// apply force field

if (useForceField)

{

// compute linear force

double Kp = c2; //25; // [N/m]

cVector3d forceField = -Kp \* position;

force.add(forceField);

for (int i = 0; i < 3; i++)

{

//Fd[i] = Kp \* (pos\_recv[i] - pos\_curr[i]);

//force(i) = Fd[i];

pos\_prev[i] = pos\_curr[i];

//cout << "Fd " << Fd[i] << endl;

//cSleepMs(TAP);

XdP[i] = Xd[i];

//cout << "XdP is " << XdP[i] << endl;

Xd[i] = pos\_curr[i];

//cout << "Xd " << Xd[i] << endl;

//DXd[2] = Xd[2] - XdP[2]+1;

DXd[i] = Xd[i] - XdP[i];

// cout << "DXd is " << DXd[i] << endl;

}

//apply EBA

if (EnableEBA)

{

for (int i = 0; i < 3; i++)

{

if (button2)

{

for (int h = 1; h > 0; h++)

{

c2 += .1;

break;

}

}

if (DXd[i]!=0)

{

//cout << "XdP is" << XdP << endl;

//cout << "EBA DXd is " << DXd[i] << endl;

//cout << "DXd222 " << endl;// Control law

// Calculation of Beta

uBETA[i] = BETA[i] = (Fd[i] - dFe[i]) / DXd[i];

// Calculation of Gamma\_max & Gamma\_min

Bmax[i] = c2 - dFe[i] / DXd[i] + sqrt(dFe[i] \* dFe[i] / DXd[i] / DXd[i] + c2 \* c2); //Gamma\_max

Bmin[i] = c2 - dFe[i] / DXd[i] - sqrt(dFe[i] \* dFe[i] / DXd[i] / DXd[i] + c2 \* c2); //Gamma\_min

if (Bmax[i] > c1)

Bmax[i] = c1; // min(c1, Gamma\_max)

// Bounding Law

if (BETA[i] < Bmin[i])

BETA[i] = Bmin[i];

if (BETA[i] > Bmax[i])

BETA[i] = Bmax[i];

}

else

{

//cout << "beta\_else is running" << endl;

uBETA[i] = BETA[i];

BETA[i] = BETA[i];

}

dFe[i] = dFe[i] + BETA[i] \* DXd[i];

if (nProjection)

{

////////////// Projection method //////////////

FOR\_Mag = sqrt(Fd[0] \* Fd[0] + Fd[1] \* Fd[1] + Fd[2] \* Fd[2]);

if (fabs(FOR\_Mag) > 0.0) // fabs(Fmag) > 0.0 is used instead of FeMag != 0.0 because 0.0 looks ZERO but it can be 0.000000000000000000000000000000XXX

//if(FOR\_Mag != 0.0)

{

UnitV[i] = Fd[i] / FOR\_Mag; // Unit Vector

}

else

{

UnitV[i] = 0.0;

}

// Inner Product

InnerProduct = dFe[0] \* UnitV[0] + dFe[1] \* UnitV[1] + dFe[2] \* UnitV[2]; // Inner Product

ProjFOR[i] = InnerProduct \* UnitV[i];

force(i) = -ProjFOR[i];

}

}

}

else

{

for (int i = 0; i < 3; i++)

{

cout << "else is running" << endl;

dFe[i] = Fd[i];

force(i) = dFe[i];

}

}

}

// send computed force

hapticDevice->setForce(force);

// update frequency counter

freqCounterHaptics.signal(1);

}

// exit haptics thread

simulationFinished = true;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void windowSizeCallback(GLFWwindow\* a\_window, int a\_width, int a\_height)

{

// update window size

width = a\_width;

height = a\_height;

// update position of label

labelHapticDevicePosition->setLocalPos(20, width - 60, 0);

// update position of label

labelHapticDeviceModel->setLocalPos(20, height - 40, 0);

// update position of scope

scope->setSize(width - 200, 180);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void errorCallback(int a\_error, const char\* a\_description)

{

cout << "Error: " << a\_description << endl;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void keyCallback(GLFWwindow\* a\_window, int a\_key, int a\_scancode, int a\_action, int a\_mods)

{

// filter calls that only include a key press

if ((a\_action != GLFW\_PRESS) && (a\_action != GLFW\_REPEAT))

{

return;

}

// option - exit

if ((a\_key == GLFW\_KEY\_ESCAPE) || (a\_key == GLFW\_KEY\_Q))

{

glfwSetWindowShouldClose(a\_window, GLFW\_TRUE);

}

// option - enable/disable force field

if (a\_key == GLFW\_KEY\_1)

{

useForceField = !useForceField;

if (useForceField)

cout << "> Enable force field \r";

else

cout << "> Disable force field \r";

}

if (a\_key == GLFW\_KEY\_M)

{

mirroredDisplay = !mirroredDisplay;

camera->setMirrorHorizontal(mirroredDisplay);

}

if (a\_key == GLFW\_KEY\_F)

{

// toggle state variable

fullscreen = !fullscreen;

// get handle to monitor

GLFWmonitor\* monitor = glfwGetPrimaryMonitor();

// get information about monitor

const GLFWvidmode\* mode = glfwGetVideoMode(monitor);

// set fullscreen or window mode

if (fullscreen)

{

glfwSetWindowMonitor(window, monitor, 0, 0, mode->width, mode->height, mode->refreshRate);

glfwSwapInterval(swapInterval);

}

else

{

int w = 0.8 \* mode->height;

int h = 0.5 \* mode->height;

int x = 0.5 \* (mode->width - w);

int y = 0.5 \* (mode->height - h);

glfwSetWindowMonitor(window, NULL, x, y, w, h, mode->refreshRate);

glfwSwapInterval(swapInterval);

}

}

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void close(void)

{

CloseUDP();

// stop the simulation

simulationRunning = false;

// wait for graphics and haptics loops to terminate

while (!simulationFinished)

{

cSleepMs(100);

}

// close haptic device

hapticDevice->close();

// delete resources

delete hapticsThread;

delete world;

delete handler;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void InitUDP()

{

// À©¼Ó ÃÊ±âÈ­

if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)

{

MessageBox(NULL, "Socket Open Error!!", "UDP", MB\_OK);

}

// socket()

sock = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sock == INVALID\_SOCKET)

{

err\_quit("socket()");

}

// bind()

ZeroMemory(&serveraddr, sizeof(serveraddr));

serveraddr.sin\_family = AF\_INET;

serveraddr.sin\_port = htons(UDP\_PORT);

serveraddr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP);

retval = ::bind(sock, (SOCKADDR\*)&serveraddr, sizeof(serveraddr));

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void CloseUDP()

{

// closesocket()

closesocket(sock);

// À©¼Ó Á¾·á

WSACleanup();

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void UDPSendData(float sData[SENDDATA\_NUM])

{

// sprintf(sbuf, "%f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f", sData[0], sData[1], sData[2], sData[3], sData[4], sData[5], sData[6], sData[7], sData[8], sData[9], sData[10], sData[11], sData[12], sData[13], sData[14], sData[15], sData[16], sData[17], sData[18], sData[19]);

Timer time;

sprintf(sbuf, "%f %f %f", sData[0], sData[1], sData[2]);

retval = sendto(sock, sbuf, strlen(sbuf), 0, (SOCKADDR\*)&clientaddr, sizeof(clientaddr));

if (retval == SOCKET\_ERROR)

{

err\_display("sendto()");

exit(-1);

}

cSleepMs(TAP);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void UDPReadData()

{

Timer2 time;

addrlen = sizeof(clientaddr);

retval = recvfrom(sock, rbuf, BUFSIZE, 0, (SOCKADDR\*)&clientaddr, &addrlen);

if (retval == SOCKET\_ERROR)

{

err\_display("recvfrom()");

exit(-1);

}

delay = timeUDPS - timeUDPR;

countd = countd++;

// sscanf(rbuf, "%f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f", &ReadData[0], &ReadData[1], &ReadData[2], &ReadData[3], &ReadData[4], &ReadData[5], &ReadData[6], &ReadData[7], &ReadData[8], &ReadData[9], &ReadData[10], &ReadData[11], &ReadData[12], &ReadData[13], &ReadData[14], &ReadData[15], &ReadData[16], &ReadData[17], &ReadData[18], &ReadData[19]);

sscanf(rbuf, "%f %f %f", &ReadData[0], &ReadData[1], &ReadData[2]);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// ¼ÒÄÏ ÇÔ¼ö ¿À·ù Ãâ·Â ÈÄ Á¾·á

void err\_quit(const char\* msg)

{

LPVOID lpMsgBuf;

FormatMessage(

FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |

FORMAT\_MESSAGE\_FROM\_SYSTEM,

NULL, WSAGetLastError(),

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT),

(LPTSTR)&lpMsgBuf, 0, NULL);

MessageBox(NULL, (LPCTSTR)lpMsgBuf, msg, MB\_ICONERROR);

LocalFree(lpMsgBuf);

exit(-1);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// ¼ÒÄÏ ÇÔ¼ö ¿À·ù Ãâ·Â

void err\_display(const char\* msg)

{

LPVOID lpMsgBuf;

FormatMessage(

FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |

FORMAT\_MESSAGE\_FROM\_SYSTEM,

NULL, WSAGetLastError(),

MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT),

(LPTSTR)&lpMsgBuf, 0, NULL);

printf("[%s] %s", msg, (LPCTSTR)lpMsgBuf);

LocalFree(lpMsgBuf);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////