

```

/*****
*
*   OpenNI 1.x Alpha
*   Copyright (C) 2011 PrimeSense Ltd.
*
*   This file is part of OpenNI.
*
*   OpenNI is free software: you can redistribute it and/or modify
*   it under the terms of the GNU Lesser General Public License as published
*   by the Free Software Foundation, either version 3 of the License, or
*   (at your option) any later version.
*
*   OpenNI is distributed in the hope that it will be useful,
*   but WITHOUT ANY WARRANTY; without even the implied warranty of
*   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
*   GNU Lesser General Public License for more details.
*
*   You should have received a copy of the GNU Lesser General Public License
*   along with OpenNI. If not, see <http://www.gnu.org/licenses/>.
*
*****/
//-----
// Includes
//-----
#include "SceneDrawer.h"

#ifdef USE_GLES
#if (XN_PLATFORM == XN_PLATFORM_MACOSX)
    #include <GLUT/glut.h>
#else
    #include <GL/glut.h>
#endif
#else
    #include "opengles.h"
#endif

extern xn::UserGenerator g_UserGenerator;
extern xn::DepthGenerator g_DepthGenerator;

extern XnBool g_bDrawBackground;
extern XnBool g_bDrawPixels;
extern XnBool g_bDrawSkeleton;
extern XnBool g_bPrintID;
extern XnBool g_bPrintState;
// B1_UD5 CODE -----
extern XnPoint3D leftHand2D;
extern XnPoint3D rightHand2D;
extern XnBool g_bPrintHandLabels;
extern XnPoint3D target2D;
extern int targetHalfSize;
extern int hits;
extern XnBool targetAvailable;
// END B1_UD5 CODE -----

#include <map>
std::map<XnUInt32, std::pair<XnCalibrationStatus, XnPoseDetectionStatus> > m_Errors;
void XN_CALLBACK_TYPE MyCalibrationInProgress(xn::SkeletonCapability& capability, XnUserID id,
XnCalibrationStatus calibrationError, void* pCookie)
{
    m_Errors[id].first = calibrationError;
}
void XN_CALLBACK_TYPE MyPoseInProgress(xn::PoseDetectionCapability& capability, const XnChar* strPose,
XnUserID id, XnPoseDetectionStatus poseError, void* pCookie)
{
    m_Errors[id].second = poseError;
}

#define MAX_DEPTH 10000
float g_pDepthHist[MAX_DEPTH];
unsigned int getClosestPowerOfTwo(unsigned int n)
{
    unsigned int m = 2;

```

```

    while(m < n) m<=<1;

    return m;
}

GLuint initTexture(void** buf, int& width, int& height)
{
    GLuint texID = 0;
    glGenTextures(1,&texID);

    width = getClosestPowerOfTwo(width);
    height = getClosestPowerOfTwo(height);
    *buf = new unsigned char[width*height*4];
    glBindTexture(GL_TEXTURE_2D,texID);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    return texID;
}

GLfloat texcoords[8];
void DrawRectangle(float topLeftX, float topLeftY, float bottomRightX, float bottomRightY)
{
    GLfloat verts[8] = {    topLeftX, topLeftY,
                           topLeftX, bottomRightY,
                           bottomRightX, bottomRightY,
                           bottomRightX, topLeftY
    };
    glVertexPointer(2, GL_FLOAT, 0, verts);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

    //TODO: Maybe glFinish needed here instead - if there's some bad graphics crap
    glFlush();
}

void DrawTexture(float topLeftX, float topLeftY, float bottomRightX, float bottomRightY)
{
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    glTexCoordPointer(2, GL_FLOAT, 0, texcoords);

    DrawRectangle(topLeftX, topLeftY, bottomRightX, bottomRightY);

    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
}

XnFloat Colors[][3] =
{
    {0,1,1},
    {0,0,1},
    {0,1,0},
    {1,1,0},
    {1,0,0},
    {1,.5,0},
    {.5,1,0},
    {0,.5,1},
    {.5,0,1},
    {1,1,.5},
    {1,1,1}
};

XnUInt32 nColors = 10;
#ifdef USE_GLES
void glPrintString(void *font, char *str)
{
    int i,l = strlen(str);

    for(i=0; i<l; i++)
    {
        glutBitmapCharacter(font,*str++);
    }
}
#endif
void DrawLimb(XnUserID player, XnSkeletonJoint eJoint1, XnSkeletonJoint eJoint2)
{

```

```

    if (!g_UserGenerator.GetSkeletonCap().IsTracking(player))
    {
        printf("not tracked!\n");
        return;
    }

    XnSkeletonJointPosition joint1, joint2;
    g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(player, eJoint1, joint1);
    g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(player, eJoint2, joint2);

    if (joint1.fConfidence < 0.5 || joint2.fConfidence < 0.5)
    {
        return;
    }

    XnPoint3D pt[2];
    pt[0] = joint1.position;
    pt[1] = joint2.position;

    g_DepthGenerator.ConvertRealWorldToProjective(2, pt, pt);
#ifdef USE_GLES
    glVertex3i(pt[0].X, pt[0].Y, 0);
    glVertex3i(pt[1].X, pt[1].Y, 0);
#else
    GLfloat verts[4] = {pt[0].X, pt[0].Y, pt[1].X, pt[1].Y};
    glVertexPointer(2, GL_FLOAT, 0, verts);
    glDrawArrays(GL_LINES, 0, 2);
    glFlush();
#endif
}

const XnChar* GetCalibrationErrorString(XnCalibrationStatus error)
{
    switch (error)
    {
    case XN_CALIBRATION_STATUS_OK:
        return "OK";
    case XN_CALIBRATION_STATUS_NO_USER:
        return "NoUser";
    case XN_CALIBRATION_STATUS_ARM:
        return "Arm";
    case XN_CALIBRATION_STATUS_LEG:
        return "Leg";
    case XN_CALIBRATION_STATUS_HEAD:
        return "Head";
    case XN_CALIBRATION_STATUS_TORSO:
        return "Torso";
    case XN_CALIBRATION_STATUS_TOP_FOV:
        return "Top FOV";
    case XN_CALIBRATION_STATUS_SIDE_FOV:
        return "Side FOV";
    case XN_CALIBRATION_STATUS_POSE:
        return "Pose";
    default:
        return "Unknown";
    }
}

const XnChar* GetPoseErrorString(XnPoseDetectionStatus error)
{
    switch (error)
    {
    case XN_POSE_DETECTION_STATUS_OK:
        return "OK";
    case XN_POSE_DETECTION_STATUS_NO_USER:
        return "NoUser";
    case XN_POSE_DETECTION_STATUS_TOP_FOV:
        return "Top FOV";
    case XN_POSE_DETECTION_STATUS_SIDE_FOV:
        return "Side FOV";
    case XN_POSE_DETECTION_STATUS_ERROR:
        return "General error";
    default:

```

```

        return "Unknown";
    }
}

void DrawDepthMap(const xn::DepthMetaData& dmd, const xn::SceneMetaData& smd)
{
    static bool bInitialized = false;
    static GLuint depthTexID;
    static unsigned char* pDepthTexBuf;
    static int texWidth, texHeight;

    float topLeftX;
    float topLeftY;
    float bottomRightY;
    float bottomRightX;
    float texXpos;
    float texYpos;

    if(!bInitialized)
    {
        texWidth = getClosestPowerOfTwo(dmd.XRes());
        texHeight = getClosestPowerOfTwo(dmd.YRes());

        // printf("Initializing depth texture: width = %d, height = %d\n", texWidth, texHeight);
        depthTexID = initTexture((void*)&pDepthTexBuf, texWidth, texHeight) ;

        // printf("Initialized depth texture: width = %d, height = %d\n", texWidth, texHeight);
        bInitialized = true;

        topLeftX = dmd.XRes();
        topLeftY = 0;
        bottomRightY = dmd.YRes();
        bottomRightX = 0;
        texXpos =(float)dmd.XRes()/texWidth;
        texYpos =(float)dmd.YRes()/texHeight;

        memset(texcoords, 0, 8*sizeof(float));
        texcoords[0] = texXpos, texcoords[1] = texYpos, texcoords[2] = texXpos, texcoords[7] =
texYpos;
    }

    unsigned int nValue = 0;
    unsigned int nHistValue = 0;
    unsigned int nIndex = 0;
    unsigned int nX = 0;
    unsigned int nY = 0;
    unsigned int nNumberOfPoints = 0;
    XnUInt16 g_nXRes = dmd.XRes();
    XnUInt16 g_nYRes = dmd.YRes();

    unsigned char* pDestImage = pDepthTexBuf;

    const XnDepthPixel* pDepth = dmd.Data();
    const XnLabel* pLabels = smd.Data();

    // Calculate the accumulative histogram
    memset(g_pDepthHist, 0, MAX_DEPTH*sizeof(float));
    for (nY=0; nY<g_nYRes; nY++)
    {
        for (nX=0; nX<g_nXRes; nX++)
        {
            nValue = *pDepth;

            if (nValue != 0)
            {
                g_pDepthHist[nValue]++;
                nNumberOfPoints++;
            }

            pDepth++;
        }
    }
}

```

```

    }

    for (nIndex=1; nIndex<MAX_DEPTH; nIndex++)
    {
        g_pDepthHist[nIndex] += g_pDepthHist[nIndex-1];
    }
    if (nNumberOfPoints)
    {
        for (nIndex=1; nIndex<MAX_DEPTH; nIndex++)
        {
            g_pDepthHist[nIndex] = (unsigned int)(256 * (1.0f - (g_pDepthHist[nIndex] /
nNumberOfPoints)));
        }
    }

    pDepth = dmd.Data();
    if (g_bDrawPixels)
    {
        XnUInt32 nIndex = 0;
        // Prepare the texture map
        for (nY=0; nY<g_nYRes; nY++)
        {
            for (nX=0; nX < g_nXRes; nX++, nIndex++)
            {

                pDestImage[0] = 0;
                pDestImage[1] = 0;
                pDestImage[2] = 0;
                if (g_bDrawBackground || *pLabels != 0)
                {
                    nValue = *pDepth;
                    XnLabel label = *pLabels;
                    XnUInt32 nColorID = label % nColors;
                    if (label == 0)
                    {
                        nColorID = nColors;
                    }

                    if (nValue != 0)
                    {
                        nHistValue = g_pDepthHist[nValue];

                        pDestImage[0] = nHistValue * Colors[nColorID][0];
                        pDestImage[1] = nHistValue * Colors[nColorID][1];
                        pDestImage[2] = nHistValue * Colors[nColorID][2];
                    }
                }

                pDepth++;
                pLabels++;
                pDestImage+=3;
            }

            pDestImage += (texWidth - g_nXRes) *3;
        }
    }
    else
    {
        xnOSMemSet(pDepthTexBuf, 0, 3*2*g_nXRes*g_nYRes);
    }

// B1_UD5 CODE -----
pDestImage = pDepthTexBuf; // restart the pointer
// Draw the orange target
if (targetAvailable)
{
    XnUInt orangeX = target2D.X;
    XnUInt orangeY = target2D.Y;
    for (XnUInt y = 0; y < g_nYRes; ++y)
    {
        for (XnUInt x = 0; x < g_nXRes; ++x)
        {

```

```

        if ( (x>orangeX-targetHalfSize)&&(x<orangeX+targetHalfSize)&&(y>orangeY-
targetHalfSize)&&(y<orangeY+targetHalfSize) )
        {
            pDestImage[0] = 255;
            pDestImage[1] = 128;
            pDestImage[2] = 0;
        }
        pDestImage+=3;
    }
    pDestImage += (texWidth - g_nXRes) *3;
}
// END B1_UD5 CODE -----

glBindTexture(GL_TEXTURE_2D, depthTexID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texWidth, texHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
pDepthTexBuf);

// Display the OpenGL texture map
glColor4f(0.75,0.75,0.75,1);

glEnable(GL_TEXTURE_2D);
DrawTexture(dmd.XRes(),dmd.YRes(),0,0);
glDisable(GL_TEXTURE_2D);

char strLabel[50] = "";
XnUserID aUsers[15];
XnUInt16 nUsers = 15;
g_UserGenerator.GetUsers(aUsers, nUsers);
for (int i = 0; i < nUsers; ++i)
{
#ifdef USE_GLES
    if (g_bPrintID)
    {
        XnPoint3D com;
        g_UserGenerator.GetCoM(aUsers[i], com);
        g_DepthGenerator.ConvertRealWorldToProjective(1, &com, &com);

        xnOSMemSet(strLabel, 0, sizeof(strLabel));
        if (!g_bPrintState)
        {
            // Tracking
            sprintf(strLabel, "%d", aUsers[i]);
        }
        else if (g_UserGenerator.GetSkeletonCap().IsTracking(aUsers[i]))
        {
            // Tracking
            sprintf(strLabel, "%d - Tracking", aUsers[i]);
        }
        else if (g_UserGenerator.GetSkeletonCap().IsCalibrating(aUsers[i]))
        {
            // Calibrating
            sprintf(strLabel, "%d - Calibrating [%s]", aUsers[i],
GetCalibrationErrorString(m_Errors[aUsers[i]].first));
        }
        else
        {
            // Nothing
            sprintf(strLabel, "%d - Looking for pose [%s]", aUsers[i],
GetPoseErrorString(m_Errors[aUsers[i]].second));
        }

        glColor4f(1-Colors[i%nColors][0], 1-Colors[i%nColors][1], 1-Colors[i%nColors]
[2], 1);

        glRasterPos2i(com.X, com.Y);
        glPrintString(GLUT_BITMAP_HELVETICA_18, strLabel);
    }
}
// B1_UD5 CODE -----
    if (true)
    {

```

```

        if (!g_UserGenerator.GetSkeletonCap().IsTracking(aUsers[i]))
        {
            printf("not tracked!\n");
            return;
        }

        XnSkeletonJointPosition lefthand, righthand;

        // Access hands in OpenNI Skeleton
        g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(aUsers
[i], XN_SKELETON_LEFT_HAND, lefthand);
        g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(aUsers
[i], XN_SKELETON_RIGHT_HAND, righthand);

        // Read left hand position and label if confidence>0.5
        if (lefthand.fConfidence > 0.5)
        {
            XnPoint3D lefthand3D;
            // Get the 3D position of the left hand
            lefthand3D = lefthand.position;
            // Get the pixel for the left hand (Update extern leftHand2D)
            g_DepthGenerator.ConvertRealWorldToProjective(1, &lefthand3D, &leftHand2D);

            // Print
            if (g_bPrintHandLabels)
            {
                sprintf(strLabel, "LeftHand (%d, %d)", (int)(leftHand2D.X), (int)
(leftHand2D.Y));

                glRasterPos2i(leftHand2D.X, leftHand2D.Y);
                glPrintString(GLUT_BITMAP_HELVETICA_18, strLabel);
            }
        }

        // Read right hand position and label if confidence>0.5
        if (righthand.fConfidence > 0.5)
        {
            XnPoint3D righthand3D, righthand2D;
            // Get the 3D position of the right hand
            righthand3D = righthand.position;
            // Get the pixel for the right hand (Update extern rightHand2D)
            g_DepthGenerator.ConvertRealWorldToProjective(1, &righthand3D, &rightHand2D);

            // Print
            if (g_bPrintHandLabels)
            {
                sprintf(strLabel, "RightHand (%d, %d)", (int)(rightHand2D.X), (int)
(rightHand2D.Y));

                glRasterPos2i(rightHand2D.X, rightHand2D.Y);
                glPrintString(GLUT_BITMAP_HELVETICA_18, strLabel);
            }
        }
    }

    // Print number of hits at the top left corner of the screen
    sprintf(strLabel, "Collected: %d", hits);
    glRasterPos2i(30, 30);
    glPrintString(GLUT_BITMAP_HELVETICA_18, strLabel);

// END B1_UD5 CODE -----
#endif

        if (g_bDrawSkeleton && g_UserGenerator.GetSkeletonCap().IsTracking(aUsers[i]))
        {
#ifdef USE_GLES
            glBegin(GL_LINES);

            glEnd()

            glColor4f(1-Colors[aUsers[i]%nColors][0], 1-Colors[aUsers[i]%nColors][1], 1-
Colors[aUsers[i]%nColors][2], 1);
            DrawLimb(aUsers[i], XN_SKELETON_HEAD, XN_SKELETON_NECK);

            DrawLimb(aUsers[i], XN_SKELETON_NECK, XN_SKELETON_LEFT_SHOULDER);
            DrawLimb(aUsers[i], XN_SKELETON_LEFT_SHOULDER, XN_SKELETON_LEFT_ELBOW);
            DrawLimb(aUsers[i], XN_SKELETON_LEFT_ELBOW, XN_SKELETON_LEFT_HAND);

```

```
DrawLimb(aUsers[i], XN_SKELETON_NECK, XN_SKELETON_RIGHT_SHOULDER);
DrawLimb(aUsers[i], XN_SKELETON_RIGHT_SHOULDER, XN_SKELETON_RIGHT_ELBOW);
DrawLimb(aUsers[i], XN_SKELETON_RIGHT_ELBOW, XN_SKELETON_RIGHT_HAND);
```

```
DrawLimb(aUsers[i], XN_SKELETON_LEFT_SHOULDER, XN_SKELETON_TORSO);
DrawLimb(aUsers[i], XN_SKELETON_RIGHT_SHOULDER, XN_SKELETON_TORSO);
```

```
DrawLimb(aUsers[i], XN_SKELETON_TORSO, XN_SKELETON_LEFT_HIP);
DrawLimb(aUsers[i], XN_SKELETON_LEFT_HIP, XN_SKELETON_LEFT_KNEE);
DrawLimb(aUsers[i], XN_SKELETON_LEFT_KNEE, XN_SKELETON_LEFT_FOOT);
```

```
DrawLimb(aUsers[i], XN_SKELETON_TORSO, XN_SKELETON_RIGHT_HIP);
DrawLimb(aUsers[i], XN_SKELETON_RIGHT_HIP, XN_SKELETON_RIGHT_KNEE);
DrawLimb(aUsers[i], XN_SKELETON_RIGHT_KNEE, XN_SKELETON_RIGHT_FOOT);
```

```
DrawLimb(aUsers[i], XN_SKELETON_LEFT_HIP, XN_SKELETON_RIGHT_HIP);
```

```
glEnd();
```

```
#ifndef USE_GLES
```

```
#endif
```

```
}
```

```
}
```

```
}
```