UNIVERSIDAD DE COSTA RICA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA ELÉCTRICA
ESTRUCTURAS ABSTRACTAS DE DATOS Y ALGORITMOS PARA INGENIERÍA
IE-0217
I CICLO 2014
PROYECTO C++

Body Positioning and Orientation Recognition Library KiNAO

Estudiante:
Willy Gerardo Villalobos Marrero B17170
Daniel Méndez Zeledón A83911
Javier Acosta Villalobos A80056

1. Introducción

Como parte del desarrollo de reconocimiento de patrones, se ha tratado de elaborar una pequeña librería la cual permitiese tomar, a partir de una cámara de Kinect, el esqueleto de una persona, mediante un algoritmo. Luego, con las posiciones X, Y, Z de cada uno de los puntos de los ligamentos del cuerpo, se podría analizar el movimiento de cada parte del cuerpo. Debido a que la cámara del Kinect permite analizar la profundidad, es, de cierta manera, sencillo analizar las posiciones, no solamente las que se muestran en el plano XY. Para este análisis, se utilizará como base la librería OpenNI que permite controlar, de manera eficaz, la cámara del Kinect, así como obtener los parámetros que el Kinect analiza.

Ya con los valores XYZ asignados, se logra obtener los ángulos de los ligamentos del cuerpo en tiempo real a partir de la ley de cosenos, donde con 2 partes del cuerpo y una línea imaginaria, se logra determinar el ángulo respectivo para el ligamento y así obtener su valor.

La razón de nuestro proyecto cabe en el ámbito de las teleoperaciones. Como un nuevo paso hacia este tema, se quiere poder hacer en movimientos en tiempo real de tal forma que puedan ser transferidos a un robot y este haga trabajos en zonas de alto riesgo, por ejemplo en un edificio después de un terremoto, en el espacio, en alguna planta nuclear, donde, a pesar del precio que se paga por cada robot, no importe si se daña, pues no se estará arriesgando vidas humanas en el intento de los distintos trabajos de alto riesgo.

Es por eso que la librería va orientada a esto, usar los ángulos de los ligamentos para ser traspasados a los robots que actualmente se encuentran en disposicion de la Escuela de Ingeniería Eléctrica, los NAO, donde los parámetros se le pueden pasar a través de ángulos en radianes y el robot podrá imitar los movimientos que se estén haciendo en tiempo real frente a la cámara del Kinect.

Figura 1: Imagen de esqueleto virtual generado mediante OpenNi

Figura 2: Fotografía de robot NAO con estudiante

2. Objetivos

2.1. Objetivo General

Crear una librería capaz de reconocer los movimientos de cuerpo entero a travéz de la cámara Kinect, utilizando la librería de OpenNI como base del desarrollo, con el fin de hacer que los robots NAO traten de imitar estos movimientos a partir de los parámetros determinados por la librería.

2.2. Objetivos Específicos

- Crear una librería que controle el movimiento por separado de la cabeza, brazos, manos, piernas y pies, utilizando el lenguaje de programación C++, con el fin de ejercer clases para utilizar el lenguaje orientado a objetos.
- Utilizar la ley de cosenos como base para la obtención de los ángulos de los ligamentos del cuerpo, tomando como referencia los puntos identificados por el algoritmo de identificación de ligamentos que incluye la libreria OpenNI.
- Crear unas funciones para trasladar los ángulos obtenidos a los NAO para tratar de hacer que éstos se muevan en tiempo real con el movimientos que esté analizando el Kinect.

3. Funciones agregadas

Entre las operaciones básicas de álgebra lineal que se consideraron necesarias para la implementación del KiNAO están las operaciones de producto punto y producto vectorial, identidades relacionadas con ángulos entre vectores $(\cos\Theta, \text{sen}\,\Theta)$, módulo de un vector, proyecciones, etc. Dichas funciones fueron agregadas al archivo "joint.cppz son las que se emplean para, inicialmente, generar vectores a partir de los joints, definidos cada uno como un punto con coordenadas (X,Y,Z).

Posterior a la obtención de dichos vectores, se realizan operaciones de producto vectorial para obtener el vector ortogonal, el cual, sumado a la obtención de proyecciones de vectores sobre uns superficie hipotética, nos permite obtener los ángulos de roll, pitch y yaw de la extremidad. Dicha información es posteriormente transformada en valores compatibles con el autómata, en este caso un NAO.

La idea básicamente es generar planos sobre los cuales los brazos se van a mover, de forma que sea posible entonces detectar variaciones en la rotación de las extremidades.

A la hora de trasladar la información de ángulos a un robot, se debe tener en cuenta que, por ejemplo, lo que se puede extender un brazo humano en grados, no es lo mismo que lo que se puede extender un NAO, por lo que se debe tener en consideración ese hecho. En este caso se plantea un condicional para que en caso de que el ángulo de rotación de una extremidad humana supere la máxima extensión del robot, simplemente se sigue restringiendo la información de manera que el automata nunca extienda sus extremidades más allá de lo que puede hacerlo físicamente.

Un próximo paso será plantear una escala de variación de manera que no solo se consideren los valores extremos de extensión del robot, sino que se pueda realizar una calibración con la persona para que los movimientos del humano sean realizados por el robot siguiendo una escala adecuada para que la máxima extensión del humano coincida con la del robot.

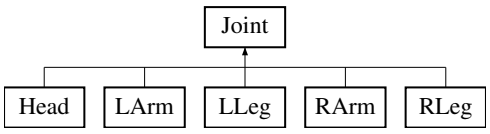
Cabe resaltar que el despliegue de información, empezando por el esqueleto, se realiza tomando como base el generador de esqueletos de OpenNI.

4. Referencia de Clase Joint

En esta sección, se describe la clase base "Joint".

`#include <joint.h>`

Diagrama de herencia para Joint:



Funciones Miembro Públicas

Esta es la declaración de la funcionalidad de la clase base "Joint", de aqui se deriban las demas.

- Joint ()
  - *Constructor de la clase "Joint".*
- virtual ∼Joint (void)
  - *Destructor del Objeto.*
- virtual float length (float[3], float[3])
  - *Genera un valor, que es la longitud entre 2 puntos en el espacio dados.*
- virtual float angle (float, float, float)
- virtual float getAngle (XnVector3D, XnVector3D, XnVector3D)
  - *L1,L2,L3(L3 es el opuesto del angulo a determinar)*
- virtual XnVector3D getNormalVector (XnVector3D, XnVector3D, XnVector3D)
  - *Genera el vector resultante de realizar el producto cruz entre los 2 vectores coplanares.*
- virtual XnVector3D getProyectionVector (XnVector3D, XnVector3D)
  - *Obtiene el vector resultante de proyectar un vector sobre otro.*
- virtual XnFloat getProyection (XnVector3D Vector1, XnVector3D Vector2)
  - *Obtiene la constante de la proyección.*
- virtual XnReferenceAxis generateReference (XnVector3D, XnVector3D, XnVector3D)
  - *Construye un nuevo eje de coordenadas a partir de 3 vectores ortogonales.*

4.1. Descripción Detallada

Aqui se describe la clase base "Joint". En esta clase se obtienen del kinect las posiciones de los joints para generar el esqueleto del cuerpo detectado. Además se implementan las funciones de álgebra lineal necesarias para poder generar vectores a partir de los joints, generar ejes de referencia nuevos y poder así generar los ángulos de rotación pitch, roll y yaw, para su posterior procesamiento y trnasmisión.

4.2. Documentación de Función Miembro

4.2.1. float Joint::angle ([float]RL1, [float]RL2, [float]IML3) [virtual]

Determina el ángulo entre 3 puntos dadas las 3 líneas descritas por estos puntos, en este casp RL1, RL2 y IML3. El ángulo a determinar viene dado por la línea imaginaria que esta opuesta a este ángulo. Dado que el ángulo que se quiere obtener es el de un joint, este método implica crear una línea imaginaria ya que no describe ninguna longitud de alguna parte del cuerpo,IML3 es este segmento contrario al ángulo por determinar.

Para esto se utiliza la ley de cosenos. Utilizando la nomenclatura de esta clase, se describe a continuación la fórmula:

$$\gamma = arccos \frac{RL1^2 + RL2^2 - IML1^2}{2 * RL1 * RL2} \tag{1}$$

4.2.2. XnReferenceAxis Joint::generateReference ([XnVector3D]J1, [XnVector3D]J2, [XnVector3D]J3) [virtual]

Construye un nuevo eje de coordenadas a partir de 3 vectores ortogonales.

Este método genera un nuevo eje de referencia centrado en un Joint. PointNormal1 es un nuevo Joint para calcular una de las normales que será parte del marco de referencia.

Utilizamos el ”struct” XnReferenceAxis definido previamente para almacenar el nuevo marco de referencia

4.2.3. float Joint::getAngle ([XnVector3D]J1, [XnVector3D]J2, [XnVector3D]J3) [virtual]

Aunque se podría, no se crea una función getAngle aquí porque se necesitan pasar los parámetros que usa el kinect para cada una de las posiciones, lo que se quiere es simplemente decir head.getAngle y que se obtenga el de la cabeza. Se podría hacer, pero inicializando todos los joints y que se estén actualizando en tiempo real en la clase Joint, para luego simplemente llamarlos desde las respectivas subclases igualando el valor actual del Joint en x,y,z a mi variable dentro de la subclase, luego como el objeto es de tipo Head, entonces el automaticamente sabe que las variables que debe pasar son las correspondientes a los puntos que describen el ángulo del cuello. Por ahora queda como mejora. Determina directamente el angulo que hay en el segundo vector Se toman los tres vectores y se determina la distancia entre ellos, donde Imag3 es la distancia opuesta el J2

4.2.4. XnVector3D Joint::getNormalVector ([XnVector3D]J1, [XnVector3D]J2, [XnVector3D]J3) [virtual]

Genera el vector resultante de realizar el producto cruz entre los 2 vectores coplanares.

J2 es el punto de unión de los vectores J2->J1 y J2->J3, a los cuales se les sacará el vector normal. Generamos 2 vectores a partir de los cuales calcularemos el producto cruz, Vector1 X Vector2 = VectorNormal.

Para Vector1 = J2->J1.

Para Vector2 = J2->J3.

Generamos el vector ortogonal VectorNormal

4.2.5. XnFloat Joint::getProyection ([XnVector3D]Vector1, [XnVector3D]Vector2) [virtual]

Obtiene la constante de la proyección.

Calculamos la proyección del Vector1 sobre el Vector2. Generamos el vector en el cual se va a guardar la proyección resultante

Calculamos el producto punto entre Vector1 y Vector2

Calculamos la norma del Vector2.

Ahora calculamos la constante que multiplica al vector sobre el cual estamos proyectando para generar el nuevo vector proyección

4.2.6. XnVector3D Joint::getProyectionVector ([XnVector3D]Vector1, [XnVector3D]Vector2) `[virtual]`

Se obtiene el vector resultante de proyectar un vector sobre otro.

Se calcula la proyección del Vector1 sobre el Vector2. Se genera el vector en el cual se va a guardar la proyección resultante

Se calcula el producto punto entre Vector1 y Vector2

Se calcula la norma del Vector2

4.2.7. float Joint::length ([float]p1[3], [float]p2[3]) `[virtual]`

A partir de 2 puntos en el espacio, con coordenadas X,Y,Z, se obtiene la longitud de éstos, a partir de la ecuación para la obtención de la magnitud de un vector, de la siguiente forma:

$$l = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{2}$$

La documentación para esta clase fue generada apartir de los archivos:

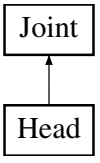
- joint.h
- joint.cpp

4.3. Referencia de Clase Head

Aqui se describe la clase derivada "Head".

`#include <joint.h>`

Diagrama de herencia para Head:



Funciones Miembro Públicas

- Head ()
  - *Constructor de la clase Head "Head" y su funcionalidad.*
- virtual ~Head (void)
  - *Destructor de la clase Head.*

- virtual float getHeadPitch (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir de la información brindada por kinect.*
- virtual float setHeadPitch (float, float &, float &)
  - *Determina el ángulo que es necesario que se mueva el NAO.*

4.4. Descripción Detallada

En esta sección, se describe la clase derivada "Head". Los joints relacionados con la cabeza sirven como referencia para la generación de los joints del torso, y por ende, los de las extremidades. Se aprovecha la capacidad de percepción de profundidad del Kinect para obtener datos de inclinación y rotación de la cabeza.

4.5. Documentación de Función Miembro

4.5.1. float Head::getHeadPitch ([XnUserID]user, [xn::UserGenerator]guser) `[virtual]`

Primero se obtienen los puntos que van a determinar el ángulo de la cabeza, se toman los puntos de la frente, cuello y torso.
Luego, como el Kinect permite obtener la profundidad, entonces se puede determinar, llamando a las funciones creadas en Joint, determinar el ángulo.

La documentación para esta clase fue generada apartir de los archivos:

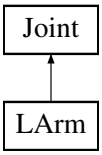
- joint.h
- head.cpp

5. Referencia de Clase LArm

Esta es la clase que describe el comportamiento para el brazo izquierdo.

`#include <joint.h>`

Diagrama de herencia para LArm:



Funciones Miembro Públicas

- virtual float getElbowRoll (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir del kinect del codo.*
- virtual float setElbowRoll (float, float &, float &)
  - *Genera el ángulo al que se necesita mover el NAO.*

- virtual float getShoulderRoll (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir del kinect del hombro.*
- virtual float setShoulderRoll (float, float &, float &)
  - *Genera el ángulo al que es necesario mover el NAO.*
- virtual float getElbowYaw (XnUserID, xn::UserGenerator)
- virtual float setElbowYaw (float, float &, float &)

5.1. Descripción Detallada

Esta es la clase que describe el comportamiento para el brazo izquierdo.

5.2. Documentación de Función Miembro

5.2.1. float LArm::getElbowRoll ([XnUserID]user, [xn::UserGenerator]guser) [virtual]

Primero se obtienen los puntos que van a determinar el ángulo de codo, se toman los puntos de la muñeca, codo y hombro. Luego, utilizando las funciones descritas en la función base Joint, se procede a determinar el ángulo para el movimiento del codo.

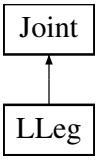
5.2.2. float LArm::getElbowYaw ([XnUserID]user, [xn::UserGenerator]guser) [virtual]

Marco de referencia en un momento anterior con punto central sobre el codo.
Se obtiene el vector elToHand (codo -> mano).
Se obtiene la proyección del vector elToHand sobre el eje Z de la referencia en el codo.
Se obtiene el punto imaginario.
Se obtiene la proyección sobre el plano XY, para ello obtenemos se forma entre la proyección y el vector elToHand.
Se obtiene primer punto que es la proyección de la mano sobre el plano XY.
Se obtiene el segundo punto sobre nuestro eje Y.
Se obtiene el ángulo entre la proyección entre el plano XY y el eje Y.
Se determina el signo del ángulo para saber si rotamos el codo en sentido horario o antihorario Se obtiene la proyección del vector proyVectorPlane sobre el eje X de la referencia en el codo.
La documentación para esta clase fue generada apartir de los archivos:

- joint.h
- larm.cpp

6. Referencia de Clase LLeg

Inheritance diagram for LLeg:



Funciones Miembro Públicas

- virtual float getHipRoll (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina utilizando kinect para la cadera.*
- virtual float setHipRoll (float, float &, float &)
  - *Genera el ángulo necesario para mover el NAO.*
- virtual float getKneePitch (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir del kinect.*
- virtual float setKneePitch (float, float &, float &)
  - *Genera el ángulo necesario para mover el NAO.*
- virtual float getAnklePitch (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir del kinect.*
- virtual float setAnklePitch (float, float &, float &)
  - *Genera un ángulo para mover el NAO.*

6.1. Descripción Detallada

Aqui se describe la clase derivada "LLeg". Es el caso similar a los brazos o la pierna derecha. Se utilizan como joints de referencia para realizar los cálculos los ubicados en la cadera, rodilla y tobillo. La idea es procesar la información del kinect para generar los ángulos, asignarlos y posteriormente desplegarlos o transmitirlos.

6.2. Documentación de Función Miembro

6.2.1. float LLeg::getHipRoll ([XnUserID]user, [xn::UserGenerator]guser) `[virtual]`

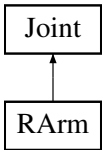
Primero se obtienen los puntos que van a determinar el ángulo de la cadera, se toman los puntos de la cadera derecha, cadera izquierda y rodilla.
Luego, utilizando las funciones descritas en la función base Joint, se procede a determinar el ángulo para el movimiento de la pierna respecto a la cadera.

La documentación para esta clase fue generada apartir de los archivos:

- joint.h
- lleg.cpp

7. RArm Class Reference RArm

Diagrama de herencia para RArm:



Public Member Functions

- virtual float getElbowRoll (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir del kinect del codo.*
- virtual float setElbowRoll (float, float &, float &)
  - *Genera el ángulo al que se necesita mover el NAO.*
- virtual float getShoulderRoll (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir del kinect del hombro.*
- virtual float setShoulderRoll (float, float &, float &)
  - *Genera el ángulo necesario al que se debe mover el NAO.*
- virtual float getElbowYaw (XnUserID, xn::UserGenerator)
- virtual float setElbowYaw (float, float &, float &)

7.1. Descripción Detallada

Aquí se describe la clase derivada "RArm".

7.2. Documentación de Función Miembro

7.2.1. float RArm::getElbowRoll ([XnUserID]user, [xn::UserGenerator]guser) `[virtual]`

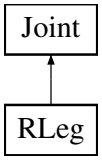
Primero se obtienen los puntos que van a determinar el ángulo de codo, se toman los puntos de la muñeca, codo y hombro. Luego, utilizando las funciones descritas en la función base Joint, se procede a determinar el ángulo para el movimiento del codo.

La documentación para esta clase fue generada apartir de los archivos:

- joint.h
- rarm.cpp

8. Referencia de Clase RLeg

Inheritance diagram for RLeg:



Funciones Miembro Públicas

- virtual float getHipRoll (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir del kinect de la cadera.*
- virtual float setHipRoll (float, float &, float &)
  - *Genera el ángulo necesario para mover el NAO.*
- virtual float getKneePitch (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir del kinect.*
- virtual float setKneePitch (float, float &, float &)
  - *Genera el ángulo necesario para mover el NAO.*
- virtual float getAnklePitch (XnUserID, xn::UserGenerator)
  - *Obtiene el ángulo que se determina a partir del kinect.*
- virtual float setAnklePitch (float, float &, float &)
  - *Genera ángulo para mover NAO.*

8.1. Descripción Detallada

Aqui se describe la clase derivada "LLeg", la cual obtiene la información necesaria para generar y establecer los ángulos de rotación de la pierna izquierda. Implica información de cadera, rodilla y tobillo.

8.2. Documentación de Función Miembro

8.2.1. float RLeg::getHipRoll ([XnUserID]user, [xn::UserGenerator]guser) `[virtual]`

Primero se obtienen los puntos que van a determinar el ángulo de la cadera, se toman los puntos de la cadera izqueirda, la cadera derecha y la rodilla.
Luego, como el Kinect permite obtener la profundidad, entonces se puede determinar, llamando a las funciones creadas en Joint, determinar el ángulo.

La documentación para esta clase fue generada apartir de los archivos:

- joint.h
- rleg.cpp

9. Referencia de Estructura XnReferenceAxis

Atributos Públicos

- XnVector3D NewX
  - *Es un vector de coordenadas del nuevo eje en X.*
- XnVector3D NewY
  - *Es un vector de coordenadas del nuevo eje en Y.*
- XnVector3D NewZ
  - *Es un vector de coordenadas del nuevo eje en Z.*

La documentación para esta estructura fue generada apartir de los siguientes archivos:

- definition.h

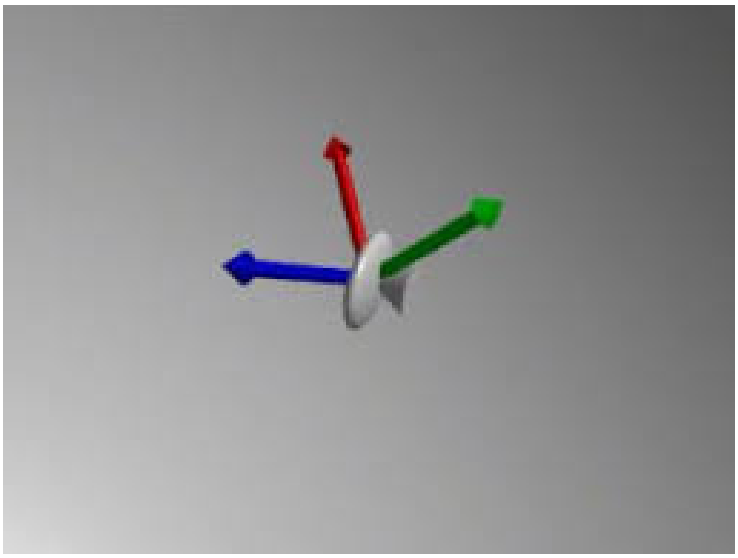


Figura 3: Eje de coordenadas rotado y trasladado en el espacio

10. Conclusiones y Recomendaciones

Se ha analizado las adiciones realizadas al proyecto openni para lograr extraer la información de los ángulos de rotación de brazs y piernas para su posterior conversión a una escala compatible con el NAO. Para ello se agregaron funciones de álgebra lineal compatibles con los tipos de datos de openni, y además se agregó una nueva estructura XnReferenceAxis, el cual almacena 3 vectores correspondientes a un nuevo eje de coordenadas.

Ha sido posible concluir que a pesar de las ventajas del Kinect mediante Openni, pueden llegar a existir ciertas limitaciones en cuanto a documentación (particularmente por el cierre del proyecto openni).

Además, existe una limitación a la hora de que la persona que está siendo detectada se coloco de perfil frente al kinect, pues ya el dispositivo no lee directamente la información del otro perfil de la persona, generando datos incompletos o erroneos, o cuando se utiliza ropa holgada, la cual puede provocar que no se detecten correctamente las rotaciones o torque de forma adecuada. Otro detalle, el cual está más directamente relacionado con las funciones agregadas, es la generaciónd de datos erroneos en ciertas posiciones, como por ejemplo los brazos completamente extendidos, en donde algunos de los parámetros que se emplean en los cálculos se indefinen.

Dado lo anterior se recomienda mejorar los algoritmos de detección de rotación para hacerlos más robustos ante estos detalles, así como ampliar la capacidad de recepción de información, ya sea empleando un segundo dispositivo kinect o Asus Xtion. Además se recomiendo migrar esta implementación a un sistema más robusto y complejo, como lo es el Motion Capture, para generar datos de forma más confiable y precisa. Es bueno intentar buscar alternativas a openni para lograr procesar la información obtenida con el kinect.

11. Referencias

1. Press, W., Teukolsky, W., Vetterling, W., Flannery, B. *Numerical Recipes: The Art of Scietific Computing*, 3ra. Ed. Cambridge University Press, 2007
2. Cormen, T., Leiserson, C., Rivest, R., Stein, C. *Introduction To Algorithms*, 3ra. Ed. MIT Press, 2009
3. Sedgewick, R., Wayne, K. *Algorithms*, 4ta. Ed. Pearson Education, 2011
4. Eckel, B. *Thinking in C++, volume I*, 2da. Ed. Prentice Hall, 2000
5. Referencias web:
   - Librería alternativa para álgebra linealhttp://seldon.sourceforge.net/contents.php
   - Cálculo vectorial en c++http://stackoverflow.com/questions/13984154/different-methods-for-finding-ang
   - Documentación del robot NAOhttps://community.aldebaran.com/doc/1-14/index.html
   - Instalación openni http://igorbarbosa.com/articles/how-to-install-kin-in-linux-mint-12-ubuntu/
   - http://blog.jorgeivanmeza.com/2011/12/instalacion-openni-sensor-kinect-y-nite-en-gnulinux-ubun

12. Anexos: Clases implementadas

12.1. class_head

Aqui se describe la clase derivada "Head", la cual hereda sus propiedades de joint.h.

```
///Declaracion de la clase derivada "Head" y su funcionalidad

#include "joint.h"

Head::Head()
```

```
{
}
Head::~Head()
{
}

///Obtiene el angulo real dado por el kinect del movimiento frontal de la cabeza
float Head::getHeadPitch(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation head, neck, torso;

        Head finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_HEAD,head);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_NECK,neck);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_TORSO,torso);


        return (finalAngle.getAngle(head.position.position,
        neck.position.position, torso.position.position));

}

///Establece un limite para el angulo de inclinacion de la cabeza
float Head::setHeadPitch(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (29.5+(angle-MIN)*(-29.5)/(MAX-MIN));
}
```

12.2. class_joint

```
///Esta es la declaracion de la funcionalidad de la clase base "Joint".
///De aqui se derivan las demas.

#include "joint.h"

Joint::Joint()
{
}
Joint::~Joint()
{
}

float Joint::getAngle(XnVector3D J1, XnVector3D J2, XnVector3D J3)
///J2 es la articulacion a la que se va a sacar el angulo
{
        float Real1, Real2, Imag3;

        ///Se toman los tres vectores y se determina la distancia entre ellos, donde Imag3
```

```
        ///es la distancia opuesta del J2
        Real1 = sqrt(pow((J1.X-J2.X),2)+pow((J1.Y-J2.Y),2)+pow((J1.Z-J2.Z),2));
        Real2 = sqrt(pow((J3.X-J2.X),2)+pow((J3.Y-J2.Y),2)+pow((J3.Z-J2.Z),2));
        Imag3 = sqrt(pow((J3.X-J1.X),2)+pow((J3.Y-J1.Y),2)+pow((J3.Z-J1.Z),2));

        return ((180/PI)*acos((pow(Real1,2)+pow(Real2,2)-pow(Imag3,2))/(2*Real1*Real2)));

        }

float Joint::length(float p1[3], float p2[3]){

    return (sqrt(pow((p2[0]-p1[0]),2)+pow((p2[1]-p1[1]),2)+pow((p2[2]-p1[2]),2)));
    ///Determina el vector que va de P1 a P2
}

float Joint::angle(float RL1, float RL2, float IML3){

    return ((180/PI)*acos((pow(RL1,2)+pow(RL2,2)-pow(IML3,2))/(2*RL1*RL2)));
    ///IML3 es el segmento contrario al angulo por determinar, el cual no existe, es imaginario

}

XnVector3D Joint::getNormalVector(XnVector3D J1, XnVector3D J2, XnVector3D J3)
///J2 es el punto de union de los vectores J2->J1 y J2->J3,
///a los cuales se les sacara el vector normal
{
        ///Generamos 2 vectores a partir de los cuales calcularemos
        ///el producto cruz, Vector1 X Vector2 = VectorNormal
        XnVector3D Vector1, Vector2, NormalVector;
        ///Para Vector1 = J2->J1
        Vector1.X=J1.X-J2.X;
        Vector1.Y=J1.Y-J2.Y;
        Vector1.Z=J1.Z-J2.Z;
        ///Para Vector2 = J2->J3
        Vector2.X=J3.X-J2.X;
        Vector2.Y=J3.Y-J2.Y;
        Vector2.Z=J3.Z-J2.Z;
        ///Generamos el vector ortogonal VectorNormal
        NormalVector.X=(Vector1.Y*Vector2.Z)-(Vector2.Y*Vector1.Z);
        NormalVector.Y=(Vector2.X*Vector1.Z)-(Vector1.X*Vector2.Z);
        NormalVector.Z=(Vector1.X*Vector2.Y)-(Vector2.X*Vector1.Y);

        return (NormalVector);

        }

XnVector3D Joint::getProyectionVector(XnVector3D Vector1, XnVector3D Vector2)
///Calculamos la proyeccion del Vector1 sobre el Vector2
{
        ///Generamos el vector en el cual se va a guardar
        ///la proyeccion resultante

        XnVector3D ProyectionVector;
```

```
        ///Calculamos el producto  punto entre Vector1 y Vector2
        XnFloat PointProduct = (Vector1.X*Vector2.X)+(Vector1.Y*Vector2.Y)+(Vector1.Z*Vector2.Z);
        ///Calculamos la norma del Vector2
        XnFloat NormVector2 = sqrt(pow(Vector2.X,2)+pow(Vector2.Y,2)+pow(Vector2.Z,2));

        XnFloat Proyection=PointProduct/(pow(NormVector2,2));

        ProyectionVector.X=Proyection*Vector2.X;
        ProyectionVector.Y=Proyection*Vector2.Y;
        ProyectionVector.Z=Proyection*Vector2.Z;

        return (ProyectionVector);

        }

XnFloat Joint::getProyection(XnVector3D Vector1, XnVector3D Vector2)
///Calculamos la proyeccion del Vector1 sobre el Vector2
{
        ///Generamos el vector en el cual se va a guardar la proyeccion resultante
        XnVector3D ProyectionVector;

        ///Calculamos el producto  punto entre Vector1 y Vector2
        XnFloat PointProduct = (Vector1.X*Vector2.X)+(Vector1.Y*Vector2.Y)+(Vector1.Z*Vector2.Z);
        ///Calculamos la norma del Vector2
        XnFloat NormVector2 = sqrt(pow(Vector2.X,2)+pow(Vector2.Y,2)+pow(Vector2.Z,2));
        ///Ahora calculamos la  constante que multiplica al vector
        ///sobre el cual estamos proyectando para generar el nuevo vector proyeccion
        XnFloat Proyection=PointProduct/(pow(NormVector2,2));

        return (Proyection);

        }

XnReferenceAxis Joint::generateReference(XnVector3D J1,XnVector3D J2,XnVector3D J3)
///Este metodo genera un nuevo eje de referencia centrado en un Joint
{
///PointNormal1 es un nuevo Joint para calcular una de las normales
///que sera parte del marco de referencia
        XnVector3D PointNormal1;
///Utilizamos el struct XnReferenceAxis definido previamente para
///almacenar el nuevo marco de referencia
        XnReferenceAxis NewAxis;

        NewAxis.NewX=Joint::getNormalVector(J1,J2,J3);

        PointNormal1.X=J2.X+NewAxis.NewX.X;
        PointNormal1.Y=J2.Y+NewAxis.NewX.Y;
        PointNormal1.Z=J2.Z+NewAxis.NewX.Z;

        NewAxis.NewY=Joint::getNormalVector(J1,J2,PointNormal1);

        NewAxis.NewZ.X=J1.X-J2.X;
        NewAxis.NewZ.Y=J1.Y-J2.Y;
        NewAxis.NewZ.Z=J1.Z-J2.Z;
```

```
        return (NewAxis);
}
```

12.3. class_l_arm

```
#include "joint.h"

LArm::LArm()
{
}
LArm::~LArm()
{
}

///Obtiene el angulo real dado por el kinect del movimiento del codo
float LArm::getElbowRoll(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation shoulder, elbow, hand;

        LArm finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_SHOULDER,shoulder);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_ELBOW,elbow);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_HAND,hand);

        return (finalAngle.getAngle(shoulder.position.position, elbow.position.position, hand.position.

}


float LArm::setElbowRoll(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (-88.5+(angle-MIN)*(-2-88.5)/(MAX-MIN));
}

float LArm::getElbowYaw(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation shoulder, elbow, hand, pastShoulder, pastElbow, pastHand;

        LArm varAngle, refAx;
        XnReferenceAxis elbowRefAx;
        XnVector3D elToHand, proyVectorZ, pImag, proyVectorPlane, pProyOverXY, pOverAxY;
        float changeAngle, finalAngle;

        pastShoulder.position.position = shoulder.position.position;
        pastHand.position.position = hand.position.position;
        pastElbow.position.position = elbow.position.position;
```

```
        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_SHOULDER,shoulder);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_ELBOW,elbow);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_HAND,hand);


        ///Marco de referencia en un momento anterior con punto central sobre el codo
        elbowRefAx = refAx.generateReference(pastHand.position.position, pastElbow.position.position, p

        ///Obtenemos el vector elToHand (codo -> mano)
        elToHand.X = hand.position.position.X-elbow.position.position.X;
        elToHand.Y = hand.position.position.Y-elbow.position.position.Y;
        elToHand.Z = hand.position.position.Z-elbow.position.position.Z;

        ///Obtenemos la proyeccion del vector elToHand sobre el eje Z de la referencia en el codo
        proyVectorZ = getProyectionVector(elToHand, elbowRefAx.NewZ);

        ///Obtenemos el punto imaginario
        pImag.X = proyVectorZ.X + elbow.position.position.X;
        pImag.Y = proyVectorZ.Y + elbow.position.position.Y;
        pImag.Z = proyVectorZ.Z + elbow.position.position.Z;

        ///Obtenemos la proyeccion sobre el plano XY, para ello obtenemos se forma entre la proyeccion
        proyVectorPlane.X = hand.position.position.X - pImag.X;
        proyVectorPlane.Y = hand.position.position.Y - pImag.Y;
        proyVectorPlane.Z = hand.position.position.Z - pImag.Z;

        ///Obtenemos primer punto que es la proyeccion de la mano sobre el plano XY
        pProyOverXY.X = elbow.position.position.X + proyVectorPlane.X;
        pProyOverXY.Y = elbow.position.position.Y + proyVectorPlane.Y;
        pProyOverXY.Z = elbow.position.position.Z + proyVectorPlane.Z;

        ///Obtenemos el segundo punto sobre nuestro eje Y
        pOverAxY.X = elbow.position.position.X + elbowRefAx.NewY.X;
        pOverAxY.Y = elbow.position.position.Y + elbowRefAx.NewY.Y;
        pOverAxY.Z = elbow.position.position.Z + elbowRefAx.NewY.Z;

        ///Obtenemos el angulo entre la proyeccion entre el plano XY y el eje Y
        changeAngle = varAngle.getAngle(pProyOverXY, elbow.position.position, pOverAxY);

        ///Determinamos el signo del angulo para saber si rotamos el codo en sentido horario o antihora
        ///Obtenemos la proyeccion del vector proyVectorPlane sobre el eje X de la referencia en el cod
        if(getProyection(proyVectorPlane,elbowRefAx.NewX) < 0){
          finalAngle = finalAngle+changeAngle;
        }
        else{
          finalAngle = finalAngle-changeAngle;
        }
        return(finalAngle);
}

float LArm::setElbowYaw(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
```

```
        if (angle<MIN) MIN=angle;
        return (119.5+(angle-MIN)*(-119.5-119.5)/(MAX-MIN));
}

float LArm::getShoulderRoll(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation elbow, shoulder, waist;

        LArm finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_ELBOW,elbow);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_SHOULDER,shoulder);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_WAIST,waist);

        return (finalAngle.getAngle(elbow.position.position, shoulder.position.position, waist.position

}

float LArm::setShoulderRoll(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (-18+(angle-MIN)*(76+18)/(MAX-MIN));
}
```

12.4. class_l_leg

```
#include "joint.h"

LLeg::LLeg()
{
}
LLeg::~LLeg()
{
}

///Obtiene el angulo real dado por el kinect del movimiento de la cadera
float LLeg::getHipRoll(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation rhip, lhip, knee;

        LLeg finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_HIP,rhip);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_HIP,lhip);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_KNEE,knee);

        return (finalAngle.getAngle(rhip.position.position, lhip.position.position, knee.position.positi
```

```
}


float LLeg::setHipRoll(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (-21.74+(angle-MIN)*(45.29+21.74)/(MAX-MIN));
}

///Aca hacemos lo mismo pero para la rotacion de la rodilla
float LLeg::getKneePitch(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation hip, knee, ankle;

        LLeg finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_HIP,hip);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_KNEE,knee);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_FOOT,ankle);

        return (finalAngle.getAngle(hip.position.position, knee.position.position, ankle.position.positi

}

float LLeg::setKneePitch(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (121.04+(angle-MIN)*(-5.29-121.04)/(MAX-MIN));
}

///Para el tobillo
float LLeg::getAnklePitch(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation knee, ankle, foot;

        LLeg finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_KNEE,knee);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_ANKLE,ankle);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_FOOT,foot);

        return (finalAngle.getAngle(knee.position.position, ankle.position.position, foot.position.posi

}

float LLeg::setAnklePitch(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
```

```
        return (-68.15+(angle-MIN)*(52.86+68.15)/(MAX-MIN));
}
```

12.5. class_r_arm

```
#include "joint.h"

RArm::RArm()
{
}
RArm::~RArm()
{
}

///Obtiene el angulo real dado por el kinect del movimiento del codo
float RArm::getElbowRoll(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation shoulder, elbow, hand;

        RArm finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_SHOULDER,shoulder);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_ELBOW,elbow);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_HAND,hand);

        return (finalAngle.getAngle(shoulder.position.position, elbow.position.position, hand.position.

}

float RArm::setElbowRoll(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (-88.5+(angle-MIN)*(-2-88.5)/(MAX-MIN));
}

float RArm::getElbowYaw(XnUserID user, xn::UserGenerator guser)
{
        /*
        XnSkeletonJointTransformation shoulder, elbow, hand;

        LArm finalAngle, refAx;
        XnReferenceAxis elbowRefAx;
        XnVector3D elToHand, proyVectorZ, pImag, proyVectorPlane, pastShoulder, pastElbow, pastHand;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_SHOULDER,shoulder);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_ELBOW,elbow);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_HAND,hand);
```

```
        pastShoulder = shoulder;

        elbowRefAx = refAx.generateReference(pastShoulder, pastElbow, pastHand);

        elToHand.X = hand.position.position.X-elbow.position.position.X;
        elToHand.Y = hand.position.position.Y-elbow.position.position.Y;
        elToHand.Z = hand.position.position.Z-elbow.position.position.Z;

        proyVectorZ = getProyectionVector(elToHand, elbowRefAx.NewZ);
        pImag.X = proyVectorZ.X + elbow.position.position.X;
        pImag.Y = proyVectorZ.Y + elbow.position.position.Y;
        pImag.Z = proyVectorZ.Z + elbow.position.position.Z;

        proyVectorPlane.X = hand.position.position.X - pImag.X;
        proyVectorPlane.Y = hand.position.position.Y - pImag.Y;
        proyVectorPlane.Z = hand.position.position.Z - pImag.Z;



        return (finalAngle.getAngle(proyVectorZ, elbow.position.position, elbowRefAx.NewX));
*/  return(0);
}

float RArm::setElbowYaw(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (119.5+(angle-MIN)*(-119.5-119.5)/(MAX-MIN));
}

float RArm::getShoulderRoll(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation elbow, shoulder, waist;

        RArm finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_ELBOW,elbow);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_SHOULDER,shoulder);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_WAIST,waist);

        return (finalAngle.getAngle(elbow.position.position, shoulder.position.position, waist.position

}

float RArm::setShoulderRoll(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (18+(angle-MIN)*(-76-18)/(MAX-MIN));
}
```

12.6. class_r_leg

```
#include "joint.h"

RLeg::RLeg()
{
}
RLeg::~RLeg()
{
}

///Obtiene el angulo real dado por el kinect del movimiento de la cadera
float RLeg::getHipRoll(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation lhip, rhip, knee;

        RLeg finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_LEFT_HIP,lhip);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_HIP,rhip);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_KNEE,knee);

        return (finalAngle.getAngle(lhip.position.position, rhip.position.position, knee.position.positi

}

float RLeg::setHipRoll(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (21.74+(angle-MIN)*(-45.29-21.74)/(MAX-MIN));
}

float RLeg::getKneePitch(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation hip, knee, ankle;

        RLeg finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_HIP,hip);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_KNEE,knee);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_FOOT,ankle);

        return (finalAngle.getAngle(knee.position.position, hip.position.position, ankle.position.positi

}

float RLeg::setKneePitch(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (121.04+(angle-MIN)*(-5.29-121.04)/(MAX-MIN));
```

```
}

float RLeg::getAnklePitch(XnUserID user, xn::UserGenerator guser)
{

        XnSkeletonJointTransformation knee, ankle, foot;

        RLeg finalAngle;

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_KNEE,knee);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_ANKLE,ankle);

        guser.GetSkeletonCap().GetSkeletonJoint(user,XN_SKEL_RIGHT_FOOT,foot);

        return (finalAngle.getAngle(knee.position.position, ankle.position.position, foot.position.posi

}

float RLeg::setAnklePitch(float angle, float& MIN, float& MAX){
        if (angle>MAX) MAX=angle;
        if (angle<MIN) MIN=angle;
        return (-68.15+(angle-MIN)*(52.86+68.15)/(MAX-MIN));
}
```

12.7. struct_xn_reference_axis

```
typedef struct XnReferenceAxis
{
        XnVector3D NewX; ///Es un vector de coordenadas del nuevo eje en X
        XnVector3D NewY; ///Es un vector de coordenadas del nuevo eje en Y
        XnVector3D NewZ; ///Es un vector de coordenadas del nuevo eje en Z
} XnReferenceAxis;
```