UNIVERSIDAD DE SEVILLA

Escuela Superior de Ingenieros

Departamento de Ingeniería Electrónica

Proyecto Fin de Carrera

Sistema de reconocimiento postural y ergonometría de bajo coste utilizando componentes comerciales

Autor: Javier Sánchez Delgado
Tutor: Hipólito Guzmán Miranda

Sistema de reconocimiento postural y ergonometría de bajo coste utilizando componentes comerciales

Autor: Javier Sánchez Delgado

Tutor: Hipólito Guzmán Miranda

Ing. Telecomunicaciones, Plan 98

Índice

1. Introducción **4**

2. Antecedentes **6**

3. Ergonometría en la oficina **12**

4. Solución propuesta **16**

4.1. Tipos de cámaras de profundidad 16

4.1.1. Cámaras de luz estructurada 16

4.1.2. Cámaras de tiempo de vuelo 18

4.2. Cámaras de profundidad comerciales 20

4.2.1. Kinect 1.0 y 2.0 20

4.2.2. PrimeSense Carmine 25

4.2.3. Asus Xtion y Xtion PRO Live 26

4.2.4. Tabla comparativa 27

4.3. Librerías y Drivers 29

4.3.1. Libfreecnet y OpenKinect 29

4.3.2. OpenNI, NITE y SensorKinect 30

4.3.3. Kinect SDK 32

4.4. Processing 33

5. Aplicación para la ergonometría en tiempo real **37**

5.1. Objetivo 37

5.2. Librerías, entorno y lenguaje de programación. 38

5.3. El esqueleto humano según Kinect 39

5.4. Representación gráfica y ángulos. 41

5.5. Limitaciones de software y hardware 48

5.6. Desarrollo de la aplicación 50

5.6.1. Clase principal 52

5.6.2. Clase AngleMeasure 54

5.6.3. Clases Extremity y Timer 56

5.7. Ejemplos de ejecución 57

6. Resultados experimentales **62**

7. Conclusiones y trabajos futuros **65**
7.1. Conclusiones 65
7.2. Trabajos futuros 68

Referencias **72**

A. Anexo I: Código **75**
A.1. Clase principal : aplicacion_ergonomia.pde 75
A.2. Clases AngleMeasure y Extremity: AngleMeasure.pde 80
A.3. Clase Timer: Timer.pde 86

1. Introducción

A lo largo del día a día realizamos todo tipo de actividades, desde que nos despertamos hasta que nos vamos a la cama el ajetreo de una persona puede ser continuo. Desayunar, preparar la comida, hacer la compra, ir a la oficina, trabajar, coger el coche o transporte público, usar el ordenador, etc. son todas actividades de lo más comunes y en las que nuestro cuerpo de una manera u otra se ve siempre sometido a un estrés. Dicho estrés puede ser mayor o menor según la actividad, siendo nosotros conscientes de ello en todo momento, o al menos eso pensamos. En ocasiones actuamos de maneras poco beneficiosas para nuestro cuerpo sin darnos cuenta, sobre todo a la hora de seguir unos hábitos o mantener unas posturas correctas, posturas que si no son las adecuadas pueden resultar en problemas de todo tipo a la larga, sobre todo de espalda o de movilidad en las articulaciones.

Con la modernización de la sociedad y la cada vez más abundante cantidad de trabajos de oficina así como de la aparición de una serie de hábitos sedentarios, estas nombradas lesiones y malestares son cada vez más comunes incluso entre los más jóvenes. ¿La razón principal? Es bastante simple: no solemos estar acostumbrados a mantener unas formas correctas a la hora de realizar diversas tareas, ya sea por mero desconocimiento o porque dichas formas no puedan resultarnos las más cómodas para llevarlas a cabo. Normalmente se suele hacer hincapié en corregir posturas incorrectas, pero el hábito hace la fuerza, y lo más común es acabar por no corregirlas o hacerlo tan solo de manera momentánea, volviendo a ellas al poco tiempo por mero descuido. Aunque la herramienta más eficaz para evitar estas molestias sería sin duda alguna la constancia y educación desde una temprana edad, no siempre es posible, y por ello hay todo tipo de ejercicios para intentar que estas malas posturas no se conviertan en graves problemas, como estiramientos y masajes. Pero, ¿no sería mejor prevenir que curar?

Imaginemos por un momento que pudiésemos corregir posturas incorrectas en el acto, de una manera totalmente no intrusiva, y sin que hubiese nadie más pendiente de nosotros, esto es posible gracias hoy en día a la tecnología. Podríamos colocar sensores en nuestra ropa o realizar diferentes monotorizaciones de nuestro cuerpo mediante cables conectados a diversos aparatos que midiesen si la tensión de nuestros músculos en diversas zonas del cuerpo es la adecuada, pero podría resultarnos incómodo y sobre todo costoso. En el mercado actualmente disponemos de una serie de dispositivos conocidos como “cámaras de profundi-

dad”, con los cuales, mediante un proyector de infrarrojos y una cámara o sensor, podemos medir la distancia que habría desde la propia cámara hasta un objeto situado frente a ella. Este tipo de cámaras tiene una aplicación bastante directa y simple para el tema que estamos tratando, por lo que pongámonos en situación. Supongamos que situamos una de estas cámaras en una oficina, la cual estaría enfocada hacía un puesto de trabajo colocado y orientado de una forma concreta, de manera que si un trabajador estuviese en él podríamos observarle, y por tanto llevar a cabo un seguimiento de diversos puntos del cuerpo como las caderas, hombros, piernas, brazos, etc. Sería fácil calcular el ángulo que forman estos puntos de nuestro cuerpo entre ellos si pudiésemos tener constancia de sus posiciones en todo momento, y esto exactamente es una de las herramientas que nos proporcionan las ya mencionadas “cámaras de profundidad” (*Depth cameras*).

Realizando un seguimiento exhaustivo en tiempo real de puntos del cuerpo como los ya mencionados, podremos corregir sobre la marcha todo tipo de posturas o realizar diversas interpretaciones de estas para, mediante software, llevar acabo unas acciones u otras. Así, podremos trabajar con el reconocimiento postural de dos diversas maneras según consideraremos a lo largo de este trabajo, bien sea reconociendo la ejecución de un gesto o posición del cuerpo (poner los brazos en cruz, realizar un paso de baile, etc), y a lo que nos referiremos a partir de ahora como “reconocimiento postural” en el más estricto sentido de la expresión, o bien observando la relación que tienen diferentes puntos del cuerpo entre sí, refiriéndonos a esto último como “ergonometría”, y que será sobre lo que más trabajemos. En el “reconocimiento postural” se realizaran una serie de acciones via software en el caso de que se realice un postura correctamente, mientras que en el caso de la “ergonometría”, dispararemos una serie de procesos en caso de obtener valores que se desvíen de los objetivos al realizar el seguimiento de puntos concretos del cuerpo. Puede que a priori suenen similares, pero como veremos más adelante al estudiarlas en profundidad, aunque relacionadas, son dos maneras diferentes de enfocar el problema, que incluso podrían complementarse entre sí para dar una solución todavía más exacta en algunas ocasiones.

Las posibilidades que nos brindan los dispositivos conocidos como “cámaras de profundidad” serán las que estudiaremos en este trabajo, para, con un reducido coste, poder elaborar sistemas con los que corregir de manera inmediata, y no invasiva, malas posturas para nuestro cuerpo, a fin de suponer una mejora no sólo para nuestra salud, sino también en el ámbito laboral al poder aplicar estos sistemas para reducir el número de bajas derivadas de estos malos hábitos.

2. Antecedentes

A lo largo de la historia de la computación se han ido desarrollando todo tipo de métodos y herramientas para facilitar la interacción entre el usuario y el equipo dedicado a esta, desde el simple ratón y teclado que todos bien conocemos y que resultan ser la forma más común y directa para ello, hasta las ya más que asentadas pantallas táctiles. Sin embargo, estos no son los únicos métodos de los que disponemos para interactuar con una computadora. Las ya anteriormente mencionadas cámaras de profundidad son otro ejemplo más de los diversos dispositivos que, a lo largo de los años, han ido siendo desarrollados, para que mediante una serie de gestos de lo más intuitivos posibles, podamos controlar los diversos equipos informáticos sin necesidad de complejas interfaces o dispositivos físicos. Bien cierto es que estos dispositivos todavía no han conseguido asentarse entre el público general, no solo por su poca eficiencia en algunos casos hasta la fecha, sino también debido a su elevado coste y a que normalmente se limitaba su uso tan solo al ámbito investigativo.

Entre todos estos diversos dispositivos de reconocimiento gesticular, podemos encontrar no solo cámaras, sino también guantes, los cuales no hicieron más que iniciar esta rama en el estudio de la interacción hombre-máquina. Así, en 1977 apareció el primero de estos curiosos dispositivos de entrada. Creado por *Electronic Visualization Laboratory* (EVL), un laboratorio de investigación de diferentes titulaciones de la Universidad de Chicago, el "Sayre Glove", supuso la primera aventura en este terreno del reconocimiento de gestos en la computación, el primer "Wired Glove", o también llamado "Dataglove", pero no el único, pues posteriores avances en 1982 dieron lugar a una serie de patentes que acabarían convirtiéndose en el guante "Data Glove" (no confundir con el nombre que genéricamente se le da a este tipo de guantes). Pero, ¿qué hace exactamente un *Wired Glove*?

(a) Data Glove

(b) Power Glove

Figura 1: Ejemplos de Wired Gloves. Imágenes de Scientific American, Octubre de 1987 y Nintendo.

Los Wired Glove, o Guantes Cableados en castellano, grosso modo, se tratan de guantes repletos de una serie de sensores de diversos tipos, como bien pueden ser magnéticos o inerciales, encargados de medir la posición y rotaciones del guante en todo momento, e incluso el movimiento de nuestros dedos dependiendo de la tecnología empleada. De esta manera, realizando un seguimiento de nuestras manos y los gestos que realizamos con ellas, esta información es a posteriori enviada a un software que se encargará de reconocer dichos gestos y actuar según corresponda. Quizás uno de los ejemplos más famosos para el público general sea el visto en las películas cuando alguien accede a una interfaz futurista de un ordenador o a un juego de realidad virtual, como se da en la película *Minority Report* (2002).

Pero este ejemplo no es el único, ya que entre en los aficionados a los videojuegos destacará siempre el inolvidable Power Glove de Nintendo Entertaiment System (comúnmente conocida como NES en E.E.U.U. y Europa, y Family Computer o Famicom en Japón), una versión bastante simplificada del Data Glove original en la que se vieron involucrados sus mismos creadores, Thomas G. Zimmerman y Jaron Lanier, responsables de las diversas patentes tecnológicas del año 82 que dieron lugar a este. Lanzado en 1989 en E.E.U.U. y fabricado por Mattel (en Japón, PAX CORPORATION), supuso un completo fracaso comercial, pues a su alto precio de venta se le sumaba la baja sensibilidad del periférico, convirtiéndose a día de hoy en una pieza difícil de ver y ansiada por coleccionistas, cuya fama reside principalmente en una escena de la película *El campeón del videojuego (The Wizard)* del mismo año 1989 producida por la propia Nintendo a modo publicitario, y cuya aparición será recordada por la siguiente frase dicha por uno de los antagonistas de la historia: *"I love the Power Glove. It's so bad"*.

"I love the Power Glove. It's so bad"

Figura 2: Fotograma de la película The Wizard, 1989, de Universal Pictures.

Sin embargo, el fracaso del *Power Glove* a nivel comercial entre el público general no supuso el abandono del desarrollo de este tipo de periféricos, aunque sí dejando de lado durante años el sector del entretenimiento y centrándose en el ámbito científico y de desarrollo. Así, siguieron apareciendo durante los sucesivos años guantes fabricados por *Virtual Technologies Inc.*, que posteriormente pasó a formar parte de *Immersion Corporation* en 2000 y la cual a posteriori

abandonó el desarrollo de este tipo de productos en 2009 dando origen a *CyberGlove Systems LLC*, como el *CyberGlove* (1990), *CyberTouch*, *CyberGrasp* o el guante *CyberForce* (con efecto sobre el brazo entero), que supusieron continuas mejoras con respecto al original de 1990. Pero *CyberGlove Systems LLC* no fue la única que se atrevió a continuar con este tipo de productos, pasado un tiempo con respecto al Power Glove, en 2002, apareció el P5 Glove de *Essential Reality*, de nuevo un guante más enfocado en sistemas de realidad virtual y juego que en la investigación, y que, pese a llegar a contar con parte del SDK (*Software Development Kit o Kit de Desarrollo de Software* en Castellano*)* liberado por su propio fabricante, nunca contó con buenas críticas, tratándose de nuevo de un fracaso comercial. Pese a ello todavía es posible adquirirlo en diversas tiendas online como amazon.com.

Dejando a un lado los guantes cableados, podemos encontrar algunas alternativas basadas en el uso de cámaras para el reconocimiento de gestos, sin necesidad de que el usuario deba utilizar algún tipo de periférico adicional, tales como las cámaras individuales (*Single Cameras*), cámaras estéreo (*Stereo Cameras*) o cámaras de profundidad (*Depth cameras* o *Depth-aware cameras*). En el caso de las primeras, las cámaras individuales, disponemos de simples cámaras 2D utilizadas para el reconocimiento de gestos en situaciones en las que otra alternativa no sería viable. En primera instancia podría no parecer la mejor manera de abordar el cometido del reconocimiento de gestos, sin embargo, varias compañías se encuentran trabajando en ello, desarrollando potentes herramientas de software que realicen el trabajo que en otros casos haría la propia cámara, obteniendo buenos resultados a la hora de reconocer gestos con las manos por ejemplo, de manera que están siendo implantadas en portátiles como la gama *Yoga Ultrabooks* de Lenovo, en teléfonos móviles como el *Samsung Galaxy S4*, o en *Smart TV*s como las del fabricante Hisense.

Figura 3: Smart TV de Hisense. Imagen de hardwaresphere.com y Hisense

Las cámaras estéreo, sin embargo, no se tratan de un simple dispositivo, sino de una técnica basada en el uso de dos cámaras 2D normales para construir una imagen 3D, con un principio similar al de los ojos humanos, constituyendo un enfoque más en la llamada “computer vision”. Para poder usar esta técnica, ade-

más de contar con un potente software que se encargue de construir un entorno 3D a raíz de dos imágenes planas, necesitamos contar con una serie de requisitos, tales como conocer la distancia de separación entre las dos cámaras utilizadas o la distancia focal de cada una de ellas. Una vez conocidos estos requisitos de las cámaras, así como tras realizar un proceso de calibración necesario, podremos mediante triangulación construir el espacio 3D observado, calculando para ello la correspondencia de cada punto en ambas imágenes generadas en todo momento. Además de las cámaras, necesitaremos no solo el software ya mencionado para construir la imagen 3D, sino que también un software específico para reconocimiento de gestos. Tampoco podemos olvidar el mayor inconveniente de este tipo de cámaras, y es que la reconstrucción de un espacio 3D a partir de un par de imágenes bidimensionales supone un alto coste computacional, siendo necesario equipos con suficiente potencia preparados para ello.

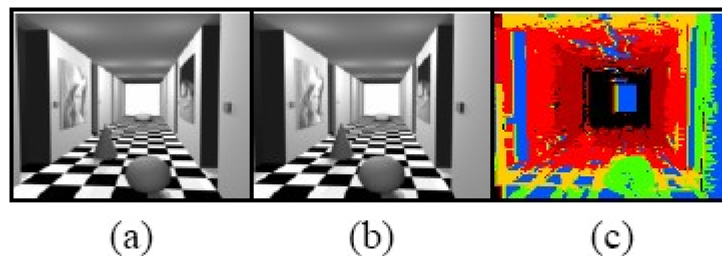


Figura 4: Ejemplo de como se crea una imagen 3D con cámaras estéreo. Imagen de Lat. Am. appl. res. v.37 n.1 Bahía Blanca ene. 2007, Real-time disparity map extraction in a dual head stereo vision system.

Pero el reconocimiento de gestos no es la única utilidad que tiene esta técnica, ya que las cámaras estéreo se emplean con diversos usos, como la de seguimiento de personas, visión para robots, mapeado de entornos en 3D, o juego inmersivo. Uno de sus usos más comunes suelen ser los ya mencionados de visión para robots y mapeado 3D, de manera que es frecuente encontrar este tipo de cámaras en astromóviles como el BH2 Lunar Rover.

Finalmente tenemos las *Depth-aware Cameras* o *Cámaras de profundidad*, que serán las que estudiaremos y utilizaremos en este trabajo. Las cámaras de profundidad se tratan de unas cámaras especiales basadas en el cálculo del "tiempo de vuelo" (*time-of-flight cameras*) o de "luz estructurada" (*structured light cameras*) para calcular un entorno tridimensional con el que trabajar. El principio de funcionamiento de las cámaras de tiempo de vuelo es bastante sencillo,

y será explicado con algo más de profundidad en un capítulo posterior, pero básicamente se trata de medir el tiempo en el que un haz de luz (normalmente infrarroja), tarda en llegar al sensor de la cámara desde que es proyectado y tras reflejarse sobre una superficie. De esta manera, y puesto que conocemos la velocidad de la luz, podemos calcular la distancia a la que se encuentra un objeto con respecto a nuestra cámara, construyendo un mapa tridimensional de un área con mayor facilidad que si se usasen otros métodos, como el de las cámaras estéreo explicadas anteriormente. En el caso de cámaras de luz estructuradas, la cosa difiere, ya que en este caso se sigue proyectando luz infrarroja, a modo de patrón como una serie de píxeles o rejilla, de manera que la forma en que estos píxeles se ven afectados al incidir sobre una superficie, su desviación con respecto a la que debería su posición, sirve para calcular éstas y la profundidad a la que se encuentran. Estas cámaras de profundidad, gracias a su facilidad para calcular la distancia a la que se encuentran obstáculos, nos permiten realizar un seguimiento exhaustivo de ciertos puntos del cuerpo humano, brindándonos la posibilidad de identificar usuarios y seguirlos en todo momento, como ya se comentó en la introducción, además de poder identificar a cortas distancias ciertas partes del cuerpo como las manos, haciéndolas perfectas para interfaces interactivas y de juego.

Figura 5: Imagen infrarroja tomada con el Kinect 1.0, cámara de luz estructurada. Obsérvense los diversos puntos que componen la resolución del sistema, así como las sombras.

En última instancia, pero no por ello relacionado, podemos mencionar brevemente los periféricos de control para el reconocimiento de gestos. Periféricos que suelen basarse en el uso de MEMS (*Microelectromechanical Systems* o *Sistemas Microelectromecánicos* en Castellano) tales como giróscopos o acelerómetros para reconocer movimientos, y entre los que destaca sin duda el famoso *Wiimote* de la consola *Wii* de Nintendo (2005), el cual aún a día de hoy sigue siendo

usado en proyectos y mods (del Inglés 'modification' , modificación) de diversa índole junto con microcontroladores tales como *Arduino*.

Cómo vemos, la mayoría de tecnologías que abordan este problema se basan en cámaras, no ya sólo por lo baratas o caras que puedan resultar, sino por su gran precisión y mejores prestaciones, otorgando una herramienta de lo más útil tanto a desarrolladores como aficionados que resulta de lo más jugosa para aplicaciones de todo tipo, como la que abordamos en este trabajo.

3. Ergonometría en la oficina

Anteriormente, en la introducción, estuvimos hablando del concepto de ergonometría, pero ¿qué es la ergonometría exactamente? Aunque suene similar a ergonomía, y haya quien lo considere lo mismo, no debemos caer en ese error, pues se refieren a conceptos que difieren entre ellos. Mientras que ergonomía puede definirse como:

"Ergonomía *(o factores humanos) es la disciplina científica relacionada con la comprensión de las interacciones entre los seres humanos y los elementos de un sistema, y la profesión que aplica teoría, principios, datos y métodos de diseño para optimizar el bienestar humano y todo el desempeño del sistema*".[1]

la definición de ergonometría no es exactamente igual, la cual podemos enunciar como:

"Ciencia *que se encarga de estudiar las características de un individuo que se han de tener en cuenta a la hora de diseñar aparatos, para que exista una reciprocidad efectiva entre los individuos y las cosas".*[2]

El matiz más importante que diferencia ambas definiciones está en el estudio o no de la persona. Mientras que en la ergonomía observamos las interacciones humano-objeto, en la ergonometría tan solo hacemos uso de las características del ser humano para diseñar máquinas en pos de su beneficio, tratándose de algo más estrictamente relacionado con las medidas, el diseño y la elaboración de nuevas herramientas. De esta manera, puesto que nuestro objetivo es trabajar con una serie de dispositivos para crear herramientas que nos ayuden entre otras cosas a por ejemplo corregir posturas, estaremos trabajando en lo que sería la ergonometría, pero, para ello necesitaremos unos conocimientos básicos de "ergonomía". ¿Qué posturas son correctas? ¿Cómo corregirlas? ¿Qué rango de valores es aceptable? Estas son preguntas que necesitaremos poder contestar para que, a la hora de crear nuestra aplicación ergonométrica, esta sea lo más precisa y fiel posible.

[1] International Ergonomics Association (IEA), "Definition of Ergonomics"

[2] Definición de ergonometría, http://julasrdiseno2.blogspot.com.es/2012/01/ergonometria.html

Así, puesto que en este trabajo nos situamos en un ámbito laboral sedentario, debemos de conocer qué posturas serán correctas a la hora de trabajar: cómo sentarse correctamente, qué posición deben de tener brazos y piernas al flexionarse, cómo debe de estar nuestro cuello inclinado al mirar a una pantalla por un tiempo prolongado, a qué altura debe estar un asiento para que sea la adecuada, etc. Generalmente podemos encontrar una serie de consejos y pautas bastante conocidos, hasta de sentido común, como sentarse recto, pero además existen estudios e información más concreta sobre el tema, pues, una buena postura no se limita tan solo al aspecto físico que conlleva. Dolores y secuelas debido a malas posturas tienen además un componente psicológico y ambiental, no todo se resume en sentarse con la espalda recta, el pensar demasiado en dichas lesiones puede hasta incitar a ellas, e incluso el no sentirse cómodo en el trabajo en lo referente a interacciones con otros compañeros o con las tareas a realizar puede no sólo ser causa directa de estrés, sino también de las mencionadas dolencias. Y es que mientras que a lo largo de los años se han ido reduciendo las bajas laborales provocadas por diversas causas, las debidas a dolores de espalda no han hecho más que aumentar con el auge del trabajo de oficina y los puestos que requieren pasar varías horas diarias sentado frente a un ordenador, por lo que éste no debería de ser un problema que se tomase a la ligera, siendo más que recomendable tener unos conocimientos mínimos apropiados. Pero no sólo es necesario saber cómo mantener una postura correcta, sino que además, como ya se ha mencionado, éstas lesiones tienen también un componente psicosocial, por lo que fomentar el buen ambiente laboral es también necesario y responsabilidad de la empresa. A día de hoy, los dolores de espalda llegan a constituir la mayor causa de absentismo laboral, suponiendo hasta un coste

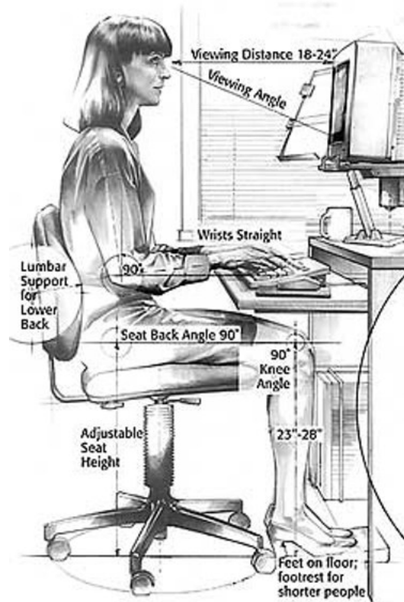


Figura 6: Gráfico referente a unas posturas correctas en un puesto de trabajo de oficina. Imagen de Integrated Safety Management, Berkeley Lab.

de hasta 1,5 % del PIB[3] de un país europeo medio, lo cual demuestra que es necesario actuar con prontitud y certeza[4].

Una postura correcta a la hora de trabajar frente a un ordenador o en una mesa, en lo referente al aspecto físico, debe seguir una serie de pautas básicas, tales como las siguientes:

- Mantener la espalda apoyada sobre el respaldo del asiento, formando unos 90º o más con las piernas.
- Los codos deben estar a la altura de la mesa, apoyados sobre ésta preferiblemente y con un ángulo de 90º o más, de manera que la articulación quede relajada.
- Mantener el cuello recto y los hombros relajados. La pantalla utilizada en el puesto de trabajo deberá colocarse a una altura adecuada para ello, para evitar que tengamos que doblar el cuello continuamente y que así nuestro ángulo de visión abarque toda la pantalla sin molestias.
- Las rodillas deben de estar a unos 90º grados o más, manteniendo las piernas paralelas, sin cruzarlas, y con los pies apoyados en el suelo, o utilizando algún tipo de escalón o reposapiés si fuese necesario. Además debemos de tener espacio suficiente bajo la mesa para mover las piernas con total libertad.

Siguiendo estos consejos, conseguiremos reducir las dolencias y problemas de espalda, e incluso ayudar a evitarlas completamente, por lo que conocerlas es primordial para una buena salud. Estos serán los conocimientos básicos de ergonomía en la oficina que necesitaremos para poder trabajar en ergonometría en el ámbito de oficina.

Como podemos ver entonces, para poder realizar una aplicación ergonométrica, que nos sea útil para adquirir información con la que desarrollar diversas aplicaciones o herramientas a posteriori, necesitamos conocer algo de ergonomía, debido a su estrecha relación. Sin embargo, y para que no haya confusión, haremos un último hincapié en la definición de ergonometría y en el uso que le daremos en este trabajo. En un principio, en la introducción, hablamos de las

[3]Producto Interior Bruto

[4]Información de "El Web de la Espalda", Fundación Kovacs, 2013. http://www.espalda.org/

cámaras de profundidad y de como trabajaríamos con 2 enfoques relacionados, el "reconocimiento postural", en el más estricto sentido de la expresión, y la ergonometría. Aquí, y a lo largo de buena parte de las siguientes secciones, nos referiremos con "ergonometría" a el uso de la información de la relación de los diferentes puntos del cuerpo entre sí cuando usamos cámaras de profundidad, datos que usaremos para realizar diversas acciones con nuestro software, como saltar una alarma si los brazos están en un ángulo incorrecto o si no nos sentamos adecuadamente, para de esta manera avisar al usuario cuando necesite corregir una postura.

4. Solución propuesta

En el presente capítulo se propone el uso de una cámara de profundidad comercial de bajo coste para monitorizar en tiempo real la postura de un trabajador de oficina con el fin de proporcionar una herramienta ergonométrica que nos ayude a reducir las dolencias y lesiones de espalda de dicho trabajador que puedan incluso resultar en baja laboral, suponiendo un coste importante para la empresa. Para dicha tarea disponemos en el mercado de diferentes posibilidades, sin embargo se ha optado por utilizar la cámara Kinect 1.0 de Microsoft para tal propósito, concretamente el modelo de Xbox 360 [5], debido no solo a su reducido coste con respecto a la versión de PC, sino también a su fácil disponibilidad en el mercado de segunda mano y a la gran comunidad de software libre que se movilizó tras su lanzamiento a finales de 2010, dando lugar a multitud de librerías open source que aún están disponibles y con las que se sigue trabajando.

No obstante, aunque en este trabajo utilicemos la cámara Kinect 1.0, los resultados y aplicaciones serán extrapolables a más modelos comerciales, debido a la naturaleza de las librerías que al apoyarse en una serie de drivers genéricos permitirán su uso en varios modelos.

4.1. Tipos de cámaras de profundidad

En el segundo punto de este trabajo mencionamos que había dos tipos diferentes de cámaras de profundidad, de *"luz estructurada"* (*structured light cameras*)" y de *"tiempo de vuelo"* (*time-of-flight cameras*). En el mercado tenemos disponibles en todo momento cualquiera de los 2 tipos, pero debido a sus mecanismos de funcionamiento y precisión, las segundas suelen ser más costosas que las primeras, por lo que, a la hora de realizar este trabajo, como ya se ha dicho, se ha optado por la opción más asequible, una cámara de luz estructurada, cuyo método de funcionamiento veremos a continuación.

4.1.1. Cámaras de luz estructurada

Anteriormente explicamos brevemente como funcionaba la técnica de las cámaras estéreo para construir imágenes tridimensionales de un espacio utilizando dos simples cámaras RGB. En este caso se analizaba punto a punto de la imagen captada por cada una de las dos cámaras utilizadas y mediante triangulación, y dado que conocemos la posición a la que se encuentran cada una de las cámaras y la distancia entre ellas, calculábamos la distancia a la que se encontraba dicho punto, su profundidad en la imagen. Las cámaras de luz estructurada siguen un

[5] Xbox360, videoconsola desarrollada por Microsoft y lanzada en 2005 al mercado

mecanismo similar, pero en este caso tan solo contamos con un único equipo, aunque seguiremos utilizando algoritmos de triangulación para mediante software localizar los diferentes puntos de la imagen. Concretamente veremos el caso de funcionamiento del Kinect.

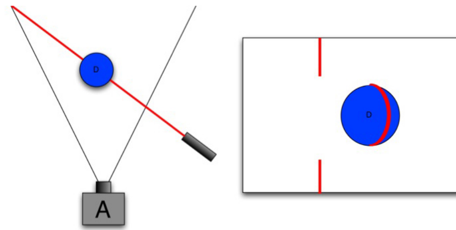


Figura 7: Mecanismo de luz estructurada. El haz de luz incide sobre el objeto y se desvía hacia un lado, este desviamiento será el dato que utilizará la cámara para saber la distancia a la que se encuentra el objeto mediante triangulación comparando con la posición en la que debería de estar si el objeto no estuviese presente. Imagen de gamasutra.com, "The Science Behind Kinects or Kinect 1.0 versus 2.0" de Daniel Lau.

Kinect 1.0 cuenta con 2 cámaras, una cámara RGB y otra de infrarrojos, así como con un proyector también de luz infrarroja. Este proyector se encargará de iluminar un patrón de puntos pseudoaleatorio sobre el entorno en el que nos encontremos, patrón que contiene un número de puntos igual a la resolución de la cámara. El patrón se proyectará sobre lo que encuentre a su paso, provocando que los diferentes puntos infrarrojos se dispersen con respecto a su posición de equilibrio con la que son proyectados, según estén más alejados o cercanos a nuestra cámara, pues, nosotros de antemano sabremos cuál debería de ser la posición correcta a una profundidad dada, desplazándose los puntos hacía un lado si la profundidad es menor, o hacia el otro si es mayor. Ahora bien, se da el caso de que tanto nuestro proyector de infrarrojos como nuestro sensor, o cámara, tienen el mismo campo de visión a efectos prácticos, por lo que nuestro sensor se encargará de observar cada uno de los puntos del patrón con el que iluminamos el área y, comparándolo con la que sabe que es su posición de equilibrio, nuestra cámara mediante unos algoritmos de triangulación podrá calcular la profundidad de cada uno de estos puntos en el espacio. Haciendo esto para cada punto en nuestro patrón proyectado, conseguiremos formar una imagen con profundidad, una imagen tridimensional, que nos describirá el espacio en el que estamos.

Por su coste y su relación calidad-precio, este será el tipo de cámaras con el que trabajemos en este trabajo, aunque no obstante repasaremos resumidamente

el otro tipo disponible a continuación.

4.1.2. Cámaras de tiempo de vuelo

Las cámaras de tiempo de vuelo, tienen un mecanismo más simple de entender y efectivo que el de las anteriores. En este caso, nuestro proyector irá iluminando el espacio con haces de rayos infrarrojos, los cuales se verán reflejados hacia nuestro sensor. De esta manera, conociendo cada cuánto tiempo los haces son proyectados, podemos calcular el tiempo que tardan en rebotar y llegar al sensor de infrarrojos, por lo que dado que la velocidad de un haz es un dato conocido, y el tiempo es algo que medimos, calcular la distancia que ha recorrido nuestro rayo resulta inmediato, y un cálculo bastante simple, proporcionándonos la profundidad a la que se encuentra un punto concreto en el espacio.

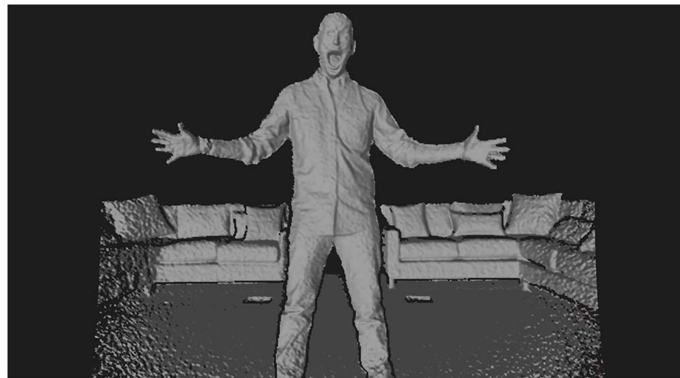


Figura 8: Depth image del Kinect 2.0, cámara de tiempo de vuelo. Se puede ver como las "sombras", o partes negras en la imagen que son consecuencia del fenómeno de oclusión alrededor y tras objetos y usuarios, han reducido notablemente con respecto a la versión original basada en luz estructurada, debido entre otros factores a la proximidad entre proyector y sensor, y un mayor detalle en la imagen. Imagen de wired.com y su video "New Xbox One: Kinect-Game|Life-WIRED Exclusive"

Así, si recorremos todo el espacio poco a poco, recogiendo la información de profundidad de tantos puntos como sea la resolución de nuestro sensor, construiremos de nuevo una imagen tridimensional de lo que observa nuestra cámara. Este tipo de cámaras tiene una ventaja bastante evidente frente a las de luz estructurada. Si por ejemplo intentásemos ver objetos de tamaño reducido como un alfiler o un pelo, con este tipo de cámaras, debido a su mejor precisión y a cómo nuestros rayos rebotan, sería posible. Sin embargo, con una cámara de luz estructurada no, principalmente debido a que en estas nuestro patrón es fijo y

no “cubriría” correctamente la superficie del objeto en cuestión.

Asimismo, en este caso no tenemos el problema de sombreado propio de las cámaras de luz estructurada, puesto que como el rayo de luz IR va recorriendo todo el espacio punto a punto y con una mayor precisión, los bordes de los objetos son perfectamente capturables para su posterior representación.

El Kinect 2.0 usa este tipo de tecnología, lo que le otorga unas mejores prestaciones que a su antecesor, de una manera un tanto peculiar. Kinect 2.0 comienza por separar cada uno de sus píxeles en dos mitades. Una vez está el píxel dividido en dos, se encarga de encender dichas mitades alternativamente, a destiempo, es decir, con un desfase de 180º entre sí. Si la mitad se encuentra encendida, absorberá fotones, si no, los rechazará, y mientras, la otra mitad que se encontrará en el estado opuesto, hará lo contrario. Al mismo tiempo que una mitad del píxel se encuentra encendida, también se encontrará encendido un láser, completamente en fase, que emitirá un pulso, iluminándose y apagándose siempre al mismo tiempo que una de estas dos mitades. Según la distancia recorrida del láser a la cámara, podrían darse diversas situaciones.

Si estamos muy cerca , el láser emitido llegará justamente durante el tiempo que la primera mitad del píxel está encendida. Si ahora nos alejamos un poco, comenzará a darse que, parte de los fotones del láser emitido llega durante el tiempo en que la primera mitad del píxel está encendida, y parte durante el tiempo de la segunda mitad del láser. Si comenzamos a alejarnos más y más, cada vez la primera mitad detectará menos luz, mientras que la segunda mitad detectará más, hasta el punto de que el fotón se absorba por completo durante la ventana de funcionamiento de este. Tras pasar un tiempo, se compararán las cantidades absorbidas por cada mitad, de manera que si la primera mitad ha absorbido más, eso significará que el láser está más cerca, y si la segunda mitad ha absorbido más, entonces está más lejos, pues al llegar en segunda ventana está tardando más. Sin embargo, como parte de la luz se suele absorber cuando se refleja esta sobre una superficie y generalmente lo que detectan nuestras mitades de píxeles no son el láser directamente, sino los rayos reflejados, más que comparar las cantidades absorbidas se compararán los ratios de fotones, pues las pérdidas por reflexión en los fotones afectan a ambas mitades de píxel por igual.

Si todavía nos alejásemos más, se acabará por dar el caso en el que el fotón del láser que llegaba en segunda ventana no sea detectado completamente en esta, y parte se absorba en la nueva ventana que habría entonces, que sería de nuevo de la primera mitad del píxel. Esto supone una ambigüedad, pues como

ya hemos visto, en primera ventana se detectarán solo los objetos más cercanos, por lo que la única forma de acabar con ella reside en aumentar el tiempo de las ventanas de detección asociadas a cada mitad de píxel, de manera que el fotón se acabe absorbiendo por completo en segunda ventana de nuevo. La variación de las ventanas de tiempo asociadas a cada píxel acaban resultando en una mayor complejidad a la hora de distinguir la señal original del láser del ruido ambiental (por ejemplo la luz ambiente), por lo que se realizaran dos mediciones, una primera de baja resolución, y una segunda de alta resolución en la que se usará la primera medición para eliminar cualquier tipo de ambigüedad que pudiese haber.

Como vemos, aunque el principio de funcionamiento se basa en ambos tipos de cámaras en detectar rayos de un proyector de infrarrojos para saber la distancia con respecto a algo, la manera en la que se lleva a cabo es completamente distinta, resultando en dos tipos de cámaras muy diferentes en cuanto a prestaciones.

4.2. Cámaras de profundidad comerciales

Como hemos comentado, en este trabajo utilizaremos el Kinect 1.0, pero no es el único modelo que tenemos disponible en el mercado. A continuación veremos un par de ellas, en concreto las que resultarían más fáciles a la hora de reutilizar el código que elaboramos aquí, así como el propio Kinect.

4.2.1. Kinect 1.0 y 2.0

Se trata de la cámara de profundidad más barata actualmente y la más asequible de las mencionadas en esta sección debido a su relación calidad/precio. Fue lanzada por Microsoft a finales de 2010 como un periférico para su consola Xbox 360, recibiendo en sus primeras fases el nombre de Project Natal. Durante su desarrollo, Microsoft contó con la ayuda de la ya desaparecida compañía israelí PrimeSense[6], esta se encargó de la fabricación de la cámara propiamente dicha, así como de la tecnología de infrarrojos de la cual era propietaria, mientras que Microsoft aportó los resultados de sus diversos años de investigación y algoritmos obtenidos para la detección de usuarios y reconocimiento de gestos.

Sus prestaciones rápidamente captaron la atención de la comunidad online y así, gracias al trabajo de diversos hackers y programadores, se consiguieron elaborar una serie de drivers libres y gratuitos en poco tiempo para permitir su uso en equipos con el fin de desarrollar aplicaciones que fuesen más allá de

[6]Desaparición debida a la compra por parte de Apple de la compañía, la cuál la disolvió como tal y la unió a ella, clausurando sus sitios webs oficiales

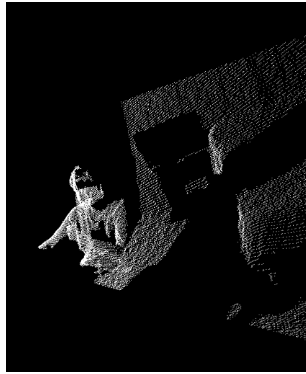
simples juegos como ocurría en la consola. Existen un par de versiones más del Kinect lanzadas a posteriori, sin embargo no todas ellas son todavía compatibles con los drivers y librerías. La primera de estas versiones recibe el nombre de *Kinect for Windows*, siendo nada más y nada menos que una actualización directa del modelo 1.0 (virtualmente 640x480 píxeles a 30fps[7]), contando con un modo cercano para facilitar su uso en entornos más reducidos de trabajo, así como con un SDK [8] oficial de la propia Microsoft, que aunque también sirve para la versión original de Xbox 360, presenta una serie de limitaciones para ella, como el no poder compilar nuestra aplicación para publicarla con objetivos comerciales.

Figura 9: Kinect 1.0 de Xbox 360. De izquierda a derecha, proyector IR, cámara RGB y cámara IR. Imagen extraída de wikipedia.

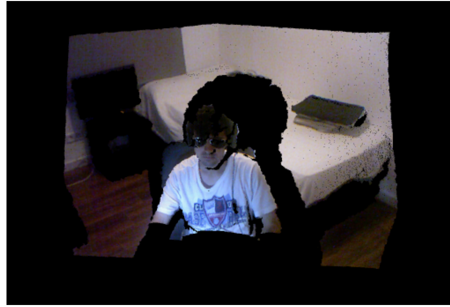
Con la salida de Xbox One[9] se incluyó un nuevo modelo de Kinect, Kinect 2.0, con un funcionamiento diferente y más preciso al del original, pero que sin embargo de momento no cuenta con drivers extraoficiales, aunque este pasado verano (concretamente en Julio), Microsoft lanzó a la venta Kinect 2.0 en su versión para PC, así como una crucial actualización de su SDK oficial para incluir todas las novedades de esta nueva versión de su sensor, haciendo posible a día de hoy trabajar con él sin problemas. Además también se puso a la venta un adaptador para usar en PC la versión de Xbox One. A diferencia del modelo 1.0, el modelo 2.0 no se trata de una cámara de *"luz estructurada"* (*structured light cameras*), si no de *"tiempo de vuelo"* (*time-of-flight cameras*), lo cual le otorga una mayor velocidad de respuesta y precisión, sin contar con que el Kinect 2.0 tiene una resolución efectiva superior (512x424 píxeles a 30fps) al tratarse de una cámara de alta definición.

El Kinect 1.0 está compuesto en todas sus versiones por 2 cámaras, una de infrarrojos y otra RGB, y un proyector de infrarrojos, así como por un pequeño motor que le permite ajustar su posición y un array de micrófonos para detectar comandos de voz.

[7]Consultar la sección 5.5 sobre la resolución real de Kinect
[8]Software Development Kit
[9]Sucesora de Xbox 360 lanzada en el año 2014



(a) Imagen tridimensional construida utilizando únicamente la información de profundidad



(b) Imagen creada superponiendo la imagen RGB sobre la información de profundidad

Figura 10: Ejemplos de capturas realizadas con Kinect 1.0

Mediante la utilización de ambas de sus cámaras, es posible fácilmente construir imágenes tridimensionales a color, utilizando tanto la información capturada por la cámara RGB, como la información de profundidad obtenida por el sensor de infrarrojos. Pero Kinect 1.0 tiene una serie de limitaciones que debemos de tener en cuenta. La primera de estas y la más importante, se trata de "las sombras de infrarrojos". La aparición de estas sombras es consecuencia directa de las cámaras de luz estructurada, y se debe a diversas causas.

1. **Sombras debidas a reflexiones en superficies reflectantes, tales como espejos o pantallas de televisión**. En este caso aparecerán en nuestra imagen de profundidad como zonas completamente negras. La razón para ello consiste en la reflexión de los rayos de luz infrarrojos en nuestro entorno, de manera que si por ejemplo tuviésemos un espejo, en el cual vemos reflejada una pared que esté por detrás del Kinect, lo que nuestro sensor vería en la imagen sería la pared, y no la superficie del espejo. Si la pared estuviese suficientemente alejada, se vería negra, ya que, cuando más lejos está algo, más oscuro aparecerá en nuestra imagen de profundidad (suponiendo que estamos trabajando monocromáticamente), pero, si colocásemos adecuadamente una serie de espejos, podríamos utilizarlos para realizar un escaneo por completo de un objeto y construir un modelo tridimensional de este con una sola pasada. Nuestro sensor verá que la distancia del rayo es la que habría del proyector al espejo más la del espejo al objeto que está siendo reflejada en este, de ahí que o sea

completamente negro, o pueda resultar visible en función de la distancia entre el espejo y el objeto.

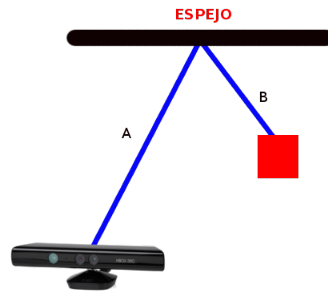


Figura 11: En este caso vemos como el espejo reflejará la imagen del objeto, a una distancia B, y que no tiene por qué ser visible para Kinect, de esta manera, si las distancias A y B son suficientemente grandes, al captar la imagen de profundidad, en la superficie del espejo aparecera todo negro y no veremos el objeto en cuestión.

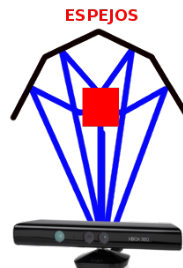


Figura 12: En esta situación nos llegan reflexiones de las diferentes caras del objeto al Kinect, de manera que, situando a una distancia correcta la cámara con respecto a los espejos (que no necesariamente tienen que ser cuatro, puesto que con dos funcionaría), podremos realizar un escaneo 3D completo del objeto de una sola vez.

2. **Sombras aparecidas en los bordes de los objetos.** Al proyectar el patrón de infrarrojos sobre nuestro entorno, los rayos inciden sobre un obstáculo que encuentran a su paso y vuelven al sensor donde se obtiene la información y mediante triangulación sacamos el valor de la profundidad, tal y como se explicó antes. Sin embargo, la precisión del kinect no es lo suficientemente buena, 1 mm, además de que si un rayo infrarrojo se ve reflejado, puesto que no hay ningún otro que tome su lugar en una cámara

estructurada como esta, no podremos capturar lo que haya detrás, dándose un fenómeno de “oclusión”, ya que sensor y proyector no están alineados. Esto ocurre especialmente en los bordes y limites de los objetos, donde no podemos obtener información suficiente.

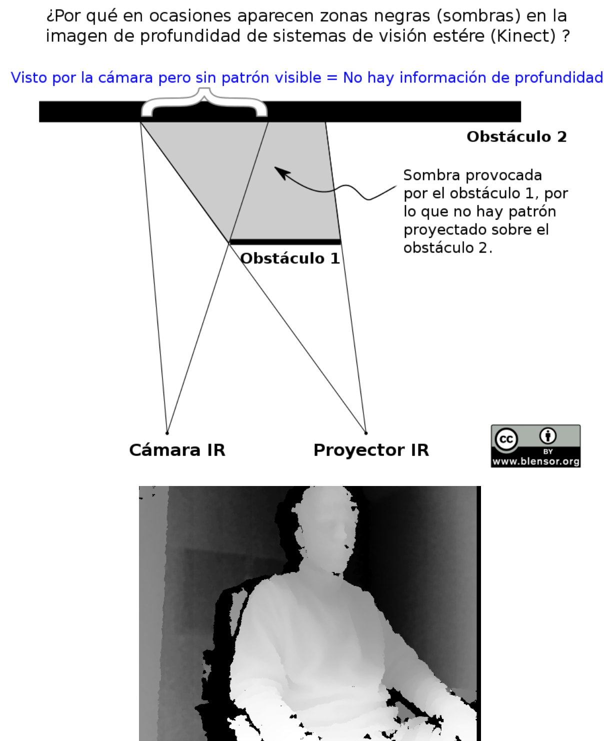


Figura 13: Explicación gráfica del fenómeno de oclusión y ejemplo

3. **Sombras en los objetos cercanos a la cámara**. El Kinect 1.0 tiene un rango de operación que va desde los 1.2 metros hasta los 4 metros aproximadamente, distancia en la que trabaja a una resolución nominal de entorno a 1 mm por píxel. Si tenemos un objeto más cercano que esta distancia o más alejado, no se vería correctamente, ya que los rayos no se proyectarían adecuadamente para que los detectase el sensor, por esto concretamente si colocamos un objeto a medio metro por ejemplo, veremos una sombra negra, pues no llega a estar correctamente dentro del ángulo de visión del sensor. En la anterior imagen (Figura 9 b)en la que superponemos RGB e imagen de profundidad, puede verse esto

perfectamente, basta con observar mi brazo derecho para notarlo.

Estos problemas sin embargo no aparecen en el Kinect 2.0, ya que la tecnología empleada en las cámaras de tiempo de vuelo (que aquí ha sido brevemente explicada) soluciona casi por completo estos problemas propios en las cámaras de luz estructurada, que también estarán presentes en los siguientes modelos que comentemos. Por último, el Kinect cuenta con un conector USB especial, cuya peculiaridad reside en la alimentación que circula por este. Un Kinect 1.0 necesita en torno a una alimentación de 12 V, tensión no disponible en un puerto USB estándar, es por ello necesario utilizar un adaptador a la hora de trabajar en PC, el cual por un lado se encargará de la comunicación entre equipo y cámara y por otro, mediante una fuente incluida en el adaptador, de la alimentación. Esto supone un pequeño sobrecoste a la hora de comenzar a utilizar la cámara, pues no todas las unidades comercializadas lo incluyen de serie en el caso de Xbox 360, debido a la existencia de algunos modelos de la consola que contaban con este puerto especial para suplir la necesidad del adaptador.

4.2.2. PrimeSense Carmine

Durante el desarrollo de la cámara Kinect 1.0, Microsoft contó con la ayuda de otra empresa ajena a ella y especializada en este campo, empresa de origen israelí y nombre PrimeSense, la cual tenía desarrollada una tecnología de infrarrojos y unos algoritmos para calcular la profundidad, que dieron lugar a la cámara de luz estructurada que ya conocemos. Sin embargo, puesto que Microsoft no era propietaria de la empresa en ningún momento, una vez se dio por completo el Kinect, ambas compañías vieron finalizada su relación de negocios. De esta manera, por un lado Microsoft aprendió sobre el hardware necesario para construir una cámara de profundidad, mientras que PrimeSense adquirió conocimientos sobre algoritmos de detección de usuarios y "*skeleton tracking*", o seguimiento de usuarios. Con cada empresa por su lado, Microsoft fue desarrollando algunas mejoras para su cámara y empezó a trabajar en el conocido como *Project Du-*

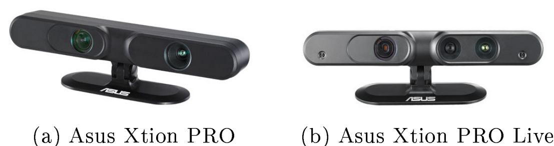
Figura 14: PrimeSense Carmine. Imagen de la web de openni.ru

rango [10], mientras que por otro lado, PrimeSense construyó sus propias cámaras de profundidad de manera independiente, al mismo tiempo que fundó y lideró el consorcio conocido como *OpenNI*[11], en el cual junto con *Willow Garage* y *Asus*, se desarrolló un framework[12] para aplicaciones de interacción natural, es decir, físicamente con el usuario, así como una serie de drivers y módulos para el Kinect de manera no oficial. Pero, ¿para qué conformarse solo con OpenNI? Como ya se ha mencionado, PrimeSense era propietaria de la tecnología de luz estructurada presente en el Kinect, por lo que el lanzamiento de un dispositivo de similares características a este no se hizo esperar.

Hasta la disolución de la compañía, al ser comprada y absorbida por *Apple* el pasado año 2014, PrimeSense puso a la venta dos modelos diferentes, de similares prestaciones al Kinect 1.0, incluso superándolo en algunas especificaciones. Por un lado tenemos el PrimeSense Carmine 1.08, que sería el equivalente más directo a la cámara de Microsoft, mientras que por otro lado tenemos el PrimeSense Carmine 1.09, una cámara de corto alcance, con un rango de operación superior al del Kinect for Windows en su modo de funcionamiento cercano, pudiendo distinguir objetos sin problemas entre los 0.35 y 1.4 metros.

Exteriormente ambas cámaras de la compañía son casi idénticas, siendo su mayor diferencia sus especificaciones técnicas. Ambas se basan en los drivers y módulos desarrollados por y para OpenNI para permitir su utilización a los usuarios, por lo que el usar esta serie de drivers los hace compatibles con casi cualquier sistema operativo gracias su disponibilidad para todo tipo de plataformas, además, nos otorga la posibilidad de utilizar el código realizado en este trabajo, que en principio es para su uso con el Kinect 1.0, en ellas sin ningún tipo de complejidad, pues la manera de programarlas puede ser la misma con el entorno adecuado.

4.2.3. Asus Xtion y Xtion PRO Live



(a) Asus Xtion PRO (b) Asus Xtion PRO Live

Figura 15: Cámaras Asus Xtion. Imagenes extraídas de http://www.asus.com

[10]nombre en clave otorgado por Microsoft a Kinect 2.0 durante su desarrollo
[11]Open Natural Interaction
[12]En el ámbito de la programación una estructura de trabajo o código

Asus, fabricante de ordenadores así como otro tipo de componentes y equipos informáticos, decidió no quedarse atrás y lanzarse al desarrollo de este tipo de dispositivos tan sofisticado como son las cámaras de profundidad. De esta manera, Asus se introdujo en el consorcio de OpenNI, donde contribuyó en el desarrollo del framework y módulos middleware, pero al mismo tiempo construyó, de nuevo como en el caso de Microsoft, con ayuda de PrimeSense, su propia gama de cámaras de profundidad. Como era de esperar, las características de su gama de cámaras son bastantes similares a las de la máquina de Microsoft, pero más aún al modelo Carmine 1.08 de PrimeSense. Profundidad, campo de visión, resolución, drivers, etc. son todas especificaciones casi idénticas a las utilizadas por la compañía israelí como era de esperarse, pero a diferencia de PrimeSense, Asus no desarrolló ningún modelo para poca profundidad, aunque sí uno que prescinde por completo de cámara RGB, el Asus Xtion, que además no es compatible con OpenNI y utiliza un software propio de la compañía, siendo relegado a un segundo plano enfocado puramente para aquellos usuarios que no van a trabajar con él y solo buscan entretenimiento a modo de juegos, por ejemplo.

El Asus Xtion posteriormente fue actualizado al modelo Xtion PRO, el cuál ya sí era funcional con el framework y módulos del consorcio OpenNI, pero que nuevamente seguía prescindiendo de cámara a color, por lo que una actualización del hardware a posteriori en forma de nuevo diseño fue comercializada, el Xtion PRO Live, siendo este finalmente el equivalente por completo al PrimeSense Carmine 1.08 y al Kinect 1.0 de Microsoft.

4.2.4. Tabla comparativa

A modo de resumen para esta sección, podemos observar en más detalle las características de algunos equipos citados anteriormente en la siguiente tabla adjunta.

Además de estos modelos ya mencionados, en el mercado podemos encontrar algunos más de cámaras de profundidad, generalmente de precio más elevado y con unas características superiores, pero que sin embargo aquí no consideraremos pues se sale del objetivo de nuestro trabajo, el cual, recordemos, consiste en montar sistemas de reconocimiento postural y ergonometría a través de dispositivos de bajo coste.

	Kinect 360	Kinect for Windows[a]	Kinect 2.0	Asus Xtion PRO	Asus Xtion PRO Live	PrimeSense Carmine 1.08
Sensores	RGB, IR[b], array de 4 micrófonos	RGB, IR, array de 4 micrófonos	RGB, IR, array de 4 micrófonos	IR	RGB, IR, 2 micrófonos	RGB, IR, 2 micrófonos
Rango de Profundidad	1.2 - 3.5 m	**Por defecto** 0.8-4 m 4-8 m (poco fiable) **Modo cercano** 0.4-3 m 3-8 m (poco fiable)	0.5-4.5 m	0.8-3.5 m	0.8-3.5 m	0.8-3.5 m
Campo de visión	57.5º Horizontal 43.5º Vertical	57.5º Horizontal 43.5º Vertical	70º Horizontal 60º Vertical	58° Horizontal 45° Vertical 70° Diagonal	58° Horizontal 45° Vertical 70° Diagonal	57.5° Horizontal 45° Vertical 69° Diagonal
Resolución	320 × 240 (640 x 480), 16 bits 30fps Depth 640 x 480 (VGA) Color	80 x 60, 320 × 240, 640 x 480, 16 bits 30fps Depth 640 x 480 (VGA) Color	512 x 424, 13 bits 30fps Depth 1920 x 1080 Full HD Color	640 x 480 (VGA) 30 fps Depth	640 x 480 (VGA) 30fps Depth 1280 x 1024 Color	640 x 480 (VGA) Depth 1280 x 1024 Color
Sistemas Operativos	Windows Unix[c] Android	Windows Unix	Windows	Windows Unix Android	Windows Unix Android	Windows Unix
Software	Kinect SDK OpenNI+NITE libfreenect Programación en C/C++/Java	Kinect SDK OpenNI+NITE libfreenect Programación en C/C++/Java	Kinect SDK 2.0	OpenNI+NITE Programación en C/C++/Java	OpenNI+NITE Programación en C/C++/Java	OpenNI+NITE Programación en C/C++/Java
Conexión y Velocidad	USB 2.0 Latencia de 90ms	USB 2.0 Latencia de 90ms	USB 3.0 Latencia mejorada a 60ms	USB 2.0	USB 2.0	USB 2.0
Rastreo	2 usuarios 20 puntos por usuario	2 usuarios 20 puntos por usuario	6 usuarios 25 puntos por cada uno	2 usuarios 20 puntos por usuario	2 usuarios 20 puntos por usuario	2 usuarios 20 puntos por usuario
Otros	Motor orientativo	Motor orientativo 2 Modos de funcionamiento Hasta 4 sensores simultáneos	Capaz de medir el ritmo cardiaco Sin motor Tracking de manos y dedos Face Tracking	Sin motor Sin cámara RGB	Sin motor	Reducido tamaño Sin motor
Precio[d]	60 € nuevo Alta disponibilidad en 2º mano	300€	200€	135€	150€	Descatalogada

Cuadro 1: Tabla comparativa de varios modelos de cámara de profundidad

[a] Tanto Kinect 360 como Kinect for Windows, son referentes al modelo Kinect 1.0, no se deben confundir con el Kinect for Windows basado en el Kinect 2.0 de uso con Xbox One

[b] Sensores de profundidad

[c] Sistemas basados en Linux, tales como Ubuntu, y sistemas MAC

[d] Precios de amazon.es y microsoftstore.com

4.3. Librerías y Drivers

Para poder utilizar el Kinect de Xbox360 en un PC para nuestro trabajo, necesitaremos una serie de drivers, sin embargo, de entre las diversas alternativas deberemos escoger la adecuada para lo que queremos hacer. Las tres opciones mayoritarias son libfreecnet, Kinect SDK y SensorKinect.

4.3.1. Libfreecnet y OpenKinect

Figura 16: Logo OpenKinect. Imagen de la web de OpenKinect

El primero de estos drivers, ***libfreecnet***, se trata del desarrollado por la comunidad de *OpenKinect*, de uso con sus librerías específicas, y que nos da acceso a las siguientes características del Kinect 1.0 :

- Cámaras RGB y de profundidad
- Motores de la base para la orientación de la cámara
- Acelerómetro del Kinect
- LEDs de señalización
- Array de micrófonos

Como vemos, podemos acceder a casi en su totalidad a todas las funciones del Kinect, pues estos drivers se desarrollaron realizando ingeniería inversa a los controladores oficiales de Microsoft. En OpenKinect, puesto que se trata de una comunidad que durante mucho tiempo ha estado en constante actualización, podemos encontrar todo tipo de código de ejemplo y funciones para nuestras aplicaciones que queramos desarrollar, además de encontrarse todo bajo una licencia de código libre Apache 2.0 o GPLv2. También cabe destacar la posibilidad de utilizar múltiples wrappers, "lenguajes de programación", por lo que desarrollar en C, C++,C#, python, ruby, actionscript o Java es posible, dándonos la oportunidad de programar en el lenguaje que nos resulte más cómodo o que sea más potente de acuerdo con lo que queramos hacer. Sin embargo, pese a tener acceso a casi la totalidad de las opciones que nos ofrece el Kinect, este

driver y librerías no resultan los óptimos para el objetivo de nuestro trabajo, así que recurriremos a su competidor más directo, OpenNI y el driver SensorKinect, por razones que explicaremos más adelante.

4.3.2. OpenNI, NITE y SensorKinect

OpenNI, *Open Natural Interaction* (la cuál mencionamos anteriormente), se trata de un consorcio de tres empresas que desarrolló un framework así como algunos módulos para ser utilizados con dispositivos de "interacción natural", dispositivos que podemos definir como aquellos que permiten el control de una aplicación o entorno virtual a una persona de manera completamente intuitiva utilizando únicamente su cuerpo, tales como el Kinect, el Xtion Pro, o los otros modelos de cámaras de profundidad que mencionamos en el apartado anterior. Pero OpenNI no es solamente el nombre que recibió el consorcio, pues se conoce de igual manera al framework y al conjunto global de módulos y drivers desarrollados por y para este, puesto que los drivers en principio solo fueron creados para usarse con las cámaras de PrimeSense y Asus, de manera que, gracias al trabajo del hacker[13] conocido como *avin2*, se adaptaron estos para poder usar el Kinect 1.0 con el framework y módulos disponibles, recibiendo estos drivers el nombre de ***SensorKinect***. Así, como SensorKinect es una especie de 'fork' (ramificación o bifurcación del código original) de los drivers de OpenNI, no soportan en su totalidad todas las opciones del Kinect de Xbox360, dejándose por el camino dos características bastante notables, como son el motor para la orientación o el uso del array de micrófonos no depurado, pero que sin embargo, no nos resulta un inconveniente, pues no los necesitaremos en ningún momento para este trabajo. Pero, ¿por qué preferir OpenNI antes que todo el entorno de la comunidad de OpenKinect? La respuesta es bien simple, y en cierto modo ya hemos ido enunciando algunas de las causas, la cuales repasaremos a continuación.

Figura 17: Logo OpenNI. Imagen extraída de openni.ru

[13]en este contexto hacemos referencia al significado no peyorativo de la palabra, sino al de aquella persona que ya bien sea por afición o estudio, realiza "hacks", programas y aplicaciones curiosas, con dispositivos que en principio no están pensados para ello

- **Portabilidad**. El uso de OpenNI nos da la posibilidad de utilizar nuestro código en diversas máquinas, no sólo el Kinect 1.0 como en el caso del entorno de OpenKinect o Kinect SDK, sino que también en las cámaras "clónicas" de otras compañías, como las ya listadas anteriormente de Asus y PrimeSense. Esto nos otorga la oportunidad de reducir costes, si se diese el caso, pudiendo elegir el sensor más económico para nuestro propósito, lo cual, utilizar dispositivos de bajo coste, es uno de los principales objetivos de este trabajo.

- **Comunidad**. OpenNI, al igual que OpenKinect y Kinect SDK, cuenta con una comunidad de trabajo propia, teniendo a disposición no solo las librerías "oficiales", sino también otras como PCL (Point Cloud Library) u OpenCV (Open Source Computer Vision), así como múltiples hacks y códigos de lo más variopintos. Además, OpenNI no se limita a su comunidad propiamente dicha, sino que encontrar código basados en estos drivers es bastante fácil en cualquier web o foro de desarrollo donde se trabaje con el Kinect de Microsoft, pues fue bastante popular en la red debido a la característica que comentamos a continuación.

- **NITE**. ¿Qué es NITE? Y este es uno de los puntos más fuertes de OpenNI, por lo que debe ser explicado correctamente. NITE es un middleware[14] desarrollado por el consorcio OpenNI, involucrando directamente al fabricante del Kinect (y de varias de estas cámaras) y dueño de su tecnología, PrimeSense, lo cual de primeras viene a significar que nadie va a conocer mejor su funcionamiento. NITE se trata así pues, de un middleware especialmente diseñado para proveer capacidad y funciones de seguimiento del esqueleto humano. Es decir, si una cámara de luz estructurada proyecta su patrón de luz infrarroja sobre un usuario, NITE se encarga de determinar un esqueleto a ese usuario mediante el seguimiento de ciertos puntos del cuerpo cuando incide sobre el usuario la luz, consiguiendo así una herramienta sumamente útil, pues no resultaría nada sencillo elaborar un código por nuestra propia cuenta que fuese capaz de identificar a un usuario y además localizar ciertos puntos clave de su cuerpo, independientemente de la estatura y complexión de este. Además, aunque se ha nombrado el cuerpo, NITE puede además ser utilizado para realizar

[14] Software que asiste a un aplicación para interactuar o comunicarse con otras aplicaciones. Se trata de una capa de abstracción de software distribuida y sirve para eliminar carga al programador.

solamente el seguimiento de manos de una manera más precisa que si se realizase de cuerpo entero, incluyendo así mismo varios gestos preconfigurados que podrán ser utilizados para disparar acciones (a esto, es a lo que llamaremos reconocimiento gesticular).

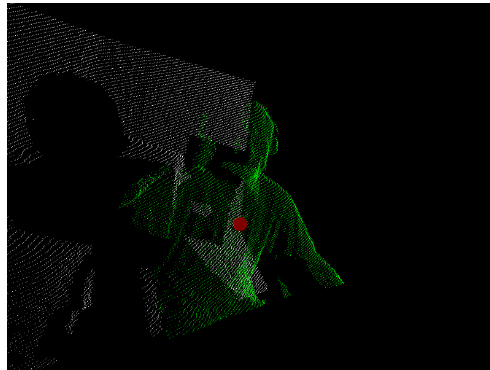


Figura 18: En esta captura podemos observar como mediante el uso de NITE, conseguimos diferenciar a un usuario del resto del entorno, así como localizar su centro de masa (punto rojo de la imagen). Se trata de una representación de nube de puntos, tridimensional, realizada dibujando en pantalla únicamente parte de los puntos proyectados por nuestro proyector con respecto a su posición en el espacio.

Como vemos, NITE es una herramienta extremadamente potente, y todavía más para cubrir nuestras necesidades en este trabajo, por lo que no es de extrañar su popularidad y el que sea la opción por la que nos decantemos en este proyecto.

4.3.3. Kinect SDK

Figura 19: Logo Kinect for Windows. Imagen extraída de microsoft.com

Por último, nos encontramos con la anteriormente mencionada ***Kinect SDK***, proporcionadas por la propia Microsoft. Kinect SDK se desarrolló con motivo del lanzamiento de Kinect for Windows, otorgando tanto drivers capaces de controlar todos los sensores del dispositivo, como una amplia librería con algoritmos sumamente potentes. Sin embargo, pese a estar desarrollada por la propia Microsoft, no

nos supone la mejor solución, pues aunque sea bastante completa, los algoritmos que nos proporciona NITE para OpenNI siguen siendo más fiables y precisos que los del SDK oficial. Además, otra de las grandes pegas del este entorno de desarrollo reside en su propio origen, y es que, puesto que fue desarrollado con motivo del Kinect for Windows, el soporte que se le da al Kinect 1.0, la versión de consola que usamos, es algo inferior, no es completo, es más, la propia Microsoft lo tiene limitado, restringiendo por ejemplo las opciones de compilación disponibles para este modelo en principio. Esto, por muy molesto que resulte, es algo más que obvio desde el punto de vista comercial, pues la diferencia que había entre ambos modelos era de casi el doble de precio en la versión de PC, y si recurrimos a la segunda mano todavía más, pues el modelo para consola se puede encontrar hoy en día por en torno a los 30-50 euros, algo que no suele pasar con el de PC.

No obstante, Kinect SDK cuenta con su propia comunidad, y en la actualidad ha acabado imponiéndose sobre las otras dos alternativas, y es que por el momento es la mejor opción y casi la única viable que se puede encontrar para usar el Kinect 2.0 en un PC, y es que el cambio radical de tecnología con respecto al modelo original, sumada a la desaparición de PrimeSense gracias a Apple, ha frenado bastante en cierto modo a la comunidad open source, que sigue avanzando, pero no a pasos tan agigantados como en el pasado, y que aún en ocasiones sigue trabajando en el modelo original.

4.4. Processing

Figura 20: Logo de Processing, extraída de processing.org

A la hora de desarrollar la aplicación para este trabajo, hemos optado por utilizar, además de los elementos descritos anteriormente, el entorno que nos otorga Processing, pero ¿qué es exactamente Processing? Processing, tal y como se define así misma en su propia web, es tanto un entorno de desarrollo, como un lenguaje de programación open source, como una comunidad online. Fundada en 2001 para promover el software en entornos artísticos, a día de hoy se trata de una herramienta utilizada de manera profesional, no solo con fines como para los que originalmente fue creada, sino también de investigación, estudio y co-

merciales. Processing además está enfocada fuertemente a un aspecto visual y al tratamiento de gráficos, a un contexto visual, por lo que, puesto que estaríamos trabajando con cámaras, nos resultará especialmente útil y simple utilizarlo. De hecho, si observamos la lista de características con las que se define en su propia página web:

- Free to download and open source
- Interactive programs with 2D, 3D or PDF output
- OpenGL integration for accelerated 3D
- For GNU/Linux, Mac OS X, and Windows
- Over 100 libraries extend the core software
- Well documented, with many books available

podemos ver la importancia que se le da por encima de todo a el aspecto gráfico, como al 3D, algo que como ya hemos dicho, nos será muy útil al trabajar con el Kinect.

Processing es un lenguaje de programación de una manera un tanto especial, y que deberemos de explicar. Con Processing nos encontramos con un lenguaje más "simple" de lo habitual, de programación orientada a objetos, pero con una sintaxis y una estructura simplificada, lo cual, puede llegar a limitarnos y suponer una restricción, que aunque pueda resultar molesta, es fácilmente evitable. Esta sintaxis y estructura reducida se trata nada más y nada menos que de una versión simplificada de Java, y fue desarrollada como tal para que la transición futura de nuevos programadores al lenguaje puro como tal, no resultase tan abrupta. De esta manera, Processing es totalmente compatible con Java, y si en algún momento se quisiese pasar de uno a otro no resultaría un mayor problema, pues la propia estructura de Processing basada en *sketches*, deriva nada más y nada menos que de una subclase de la clase de Java PApplet, que implementa la mayoría de las características de Processing.

Puesto que es POO[15], el uso de múltiples clases es posible, pudiendo tener múltiples archivos con estas que podremos reutilizar, como en Java, C++ o cualquier otro lenguaje de POO, en múltiples programas, sin embargo, su funcionamiento no es el mismo. Processing a la hora de traducir a Java, incluirá

[15]Programación Orientada a Objetos

todas nuestras clases como *clases internas (inner classes)*[16] , y una vez ha realizado tal traducción con todas las clases en un único fichero, el que Processing llama como 'sketch', realizará la compilación del código. Esto supone una de las restricciones anteriormente comentadas, y es que, si quisiésemos utilizar variables estáticas o métodos dentro de nuestras clases, tendríamos que programar en Java puro y duro. Como vemos, esto no supondría más que indagar un poco en el uso de las clases clave que Processing utiliza para poder trasladar nuestro código a Java sin mucha complejidad, aunque por otro lado, no será necesario, pues podremos realizar todo lo que nos propongamos sin recurrir a 'static' (palabra clave en java que designa a un atributo o método como propio de una clase y común a todas las instancias que se hagan de esta) al programar en la mayoría de los casos.

Processing es un entorno de desarrollo o más bien un integrated development environment (IDE), comúnmente conocido como sketchbook, donde escribiremos nuestros programas y código (sketches). Este IDE será estrictamente necesario por norma general, pues si queremos programar tanto con la sintaxis simplificada que nos ofrece Processing, como luego poder traducir este a Java puro, tendremos que usar el sketchbook. Por otro lado, nos ofrece además múltiples herramientas y opciones de edición típicas de cualquier IDE, como la ejecución de nuestro código o una consola de errores. Además, desde el IDE de Processing tendremos la oportunidad de importar cualquier tipo de librería de la que dispongamos inmediatamente, tan solo tendremos que colocarla en el directorio adecuado de

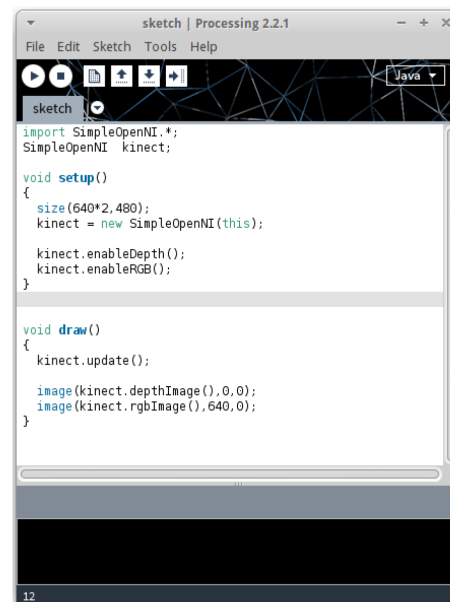


Figura 21: IDE de Processing. Se puede observar que a simple vista es un editor más, contando con las típicas herramientas de uno de estos, aunque algo simplificadas en comparación con otros IDEs como Eclipse o Netbeans.

[16]Clases que se encuentran declaradas en su totalidad dentro del archivo de otras. En el caso de Processing puesto que nuestro programa se define en un documento llamado sketch, todas las clases se incluirán en él.

la aplicación, existiendo multitud de ellas con todo tipo de finalidades, lo que nos lleva al último punto. Por otro lado, si estamos acostumbrados a trabajar con un IDE como Eclipse, Processing permite su uso también con este para que resulto más cómodo y completo a usuarios más avanzados.

Processing es una comunidad online de desarrollo en continuo crecimiento desde su nacimiento, completamente open source, con librerías de todo tipo (para audio, vídeo, gráficos 3D, comunicaciones serie o por red, etc) y ejemplos disponibles en cualquier momento. Además dispone de una serie de foros de debates en los que los usuarios pueden exponer sus dudas o mostrar sus progresos con toda la comunidad, como de su propio repositorio en github o una sección a modo de wiki, así como la posibilidad de consultar toda la información referente a las librerías que nos interesen y a las funciones y clases básicas de Processing. Podremos, por otro lado, encontrar multitud de libros publicados con guías y tutoriales de programación usando el lenguaje, con lo que tenemos todo tipo de información en nuestras manos si la necesitásemos.

Como hemos visto, Processing es una opción muy a tener en cuenta, nos aporta múltiples herramientas que necesitaremos en el trabajo y que además nos lo harán bastante más simple y asequible, convirtiendo a Processing en un recurso imprescindible en muchas ocasiones.

Así, en nuestro trabajo combinaremos todo lo explicado en esta sección para, de una manera simple y eficaz a la vez que económica, conseguir nuestro objetivo, utilizando unos drivers concretos, un entorno de programación simple y eficaz, y unas librerías potentes.

5. Aplicación para la ergonometría en tiempo real

5.1. Objetivo

En este trabajo desarrollaremos una aplicación encargada de realizar el seguimiento de una persona en su puesto laboral en un entorno de oficina. Dicho seguimiento consistirá en comprobar en todo momento que una postura es la correcta, de acuerdo con unas directrices y consejos que previamente fueron introducidos en la sección 3 del trabajo (*Ergonometría en la oficina*), y notificar al usuario cuando esta no lo sea, ya bien mediante una alarma sonora, visual, o de manera silenciosa, creando un informe o log de cuando se produce esta incorrección que podremos consultar al final de cada sesión de seguimiento. Con dicho log podremos servirnos posteriormente para realizar estudios sobre el comportamiento en los puestos de trabajo de cada individuo y llegar a una solución personalizada que no suponga la baja o molestia en el usuario y mejore su salud en la medida de lo posible, además de ayudarnos a mejorar el entorno de trabajo y pudiendo suponer un ahorro para la empresa si se realiza adecuadamente. Por ejemplo, si alguien se pasase demasiado tiempo con el cuello de manera incorrecta podría ser porque la altura de su monitor no fuese la adecuada, o con las rodillas demasiado flexionadas porque la altura de su asiento no fuese tampoco la debida, con esta aplicación avisaremos de estos comportamientos que suelen darse de manera inconsciente.

La aplicación se encargará de una manera más concreta de:

- identificar a un usuario en su puesto de trabajo, el cuál deberá estar colocado de una manera específica para permitir el seguimiento correcto por parte del Kinect.
- Una vez identificado al usuario, deberemos tener en observación constante su "esqueleto" [17], fijándonos en ciertos puntos claves del cuerpo, los cuales, si no se encuentran en unas determinadas condiciones con respecto a otros, supondrán una postura incorrecta.
- En el caso de darse una postura incorrecta por parte del usuario, deberemos informar a éste, o tomar nota (según se desee) y crear un informe con todas las incidencias para su posterior análisis de manera externa de

[17] Cuando hablamos del esqueleto en este contexto nos estaremos refiriendo a aquellos puntos del cuerpo que Kinect es capaz de identificar en todo momento, no del esqueleto humano desde un puto de vista puramente fisiológico.

forma que el usuario pueda solucionar los problemas que pueda tener en su manera de trabajar.

A continuación procederemos a explicar como se realiza cada uno de estos pasos en los siguientes apartados, así como las herramientas que usaremos para ello.

5.2. Librerías, entorno y lenguaje de programación.

De entre todas las librerías y drivers comentados en el punto 4, ya se mencionó por cual nos decantaríamos, así pues, con el fin de utilizar las herramientas más precisas para *skeleton tracking*, recurriremos a **OpenNI+NITE como framework y librerías** en las que programar. Si quisiésemos además codificar con Kinect 1.0, como es nuestro caso para ahorrar en costes, aunque podríamos usar cualquiera de las cámaras de corte similar y de tecnología OpenNI como las de PrimeSense o Asus, necesitaremos utilizar los **drivers de SensorKinect**, especialmente para sistemas operativos basados en UNIX como Ubuntu/GNU/-Linux o MAC OS. En Windows también es posible utilizar los drivers del SDK de Microsoft a día de hoy. Por último, para hacer más simple y potente nuestro código, nos decantaremos por **programar en Java**, concretamente en la versión simplificada de este que nos otorga **Processing**. Dentro de Processing utilizaremos las librerías especificas de OpenNI portadas a Java, conocidas como **SimpleOpenNI**, además de recurrir al sketchbook de Processing como IDE en lugar de utilizar Eclipse, Netbeans u otro entorno de desarrollo, por el simple motivo de que nos aporta todo lo que necesitamos sin complicaciones y de una manera rápida y sencilla, desde un editor de texto donde codificar, hasta una típica consola por la que obtendremos información de las posibles incidencias que se produzcan así como la posibilidad de ejecutar nuestro código directamente y de depurarlo (utilizando para ello el modo experimental que incorpora el IDE de Processing, PDE o Processing Development Environment, desde su versión 2.0b7), es decir, las herramientas básicas de todo entorno.

De las herramientas nombradas anteriormente, usaremos concretamente las siguientes versiones:

- SensorKinect v 5.1.2.1[18]
- OpenNI 2.2 y NITE 2.2[19]

[18]En el caso de Ubuntu y otros sistemas Unix, necesitaremos además las librerías libusb 1.0 o superior

[19]En caso de no disponer de la versión 1.5 o de problemas de incompatibilidad en nuestro

- SimpleOpenNI v1.96
- Processing 2.2.1

Aunque en este caso usemos esta configuración, esto no significa que sea la única, aunque sí la que más se ajusta a lo que queremos con facilidad. Kinect SDK y OpenKinect podrían ser otras posibilidades a tener en cuenta en el futuro.

5.3. El esqueleto humano según Kinect

A la hora de realizar el seguimiento del usuario una vez lo tenemos identificado, nos encargaremos de tener en constante observación la posición de ciertos puntos del cuerpo. Kinect, y OpenNI, junto con NITE más concretamente, nos ayudarán en esta tarea tan compleja.

OpenNI nos permite el seguimiento de una serie de puntos concretos del cuerpo, cuya precisión se verá determinada por un parámetro determinado "confidence", o la seguridad con la que dichos puntos son determinados por la cámara. Dicho parámetro de fidelidad a la hora de medir nos servirá para saber que partes del cuerpo están siendo observadas de manera precisa y cuales no, pues los más de 20 puntos de los que se pueden distinguir con Kinect y OpenNI no resultan siempre útiles si su valor no es el apropiado. Para saber que puntos nos serán de ayuda y cuales no, consideraremos solo aquellos cuyo valor de "confidence" sea igual o mayor que 0,5[20], pues aquellos que no cumplen

Figura 22: Esqueleto humano detectado por Kinect. Imagen no espejada, por lo que el brazo izquierdo se sitúa a la derecha de la imagen y al revés para la extremidad opuesta.

equipo, las versiones 2.2 de OpenNI y NITE servirán perfectamente. Los cambios más significativos introducidos en estas versiones superiores se tratan de nombres de funciones además de su estabilidad, razón por la que es mejor utilizar estas y no anteriores si es posible.

[20]El valor del parámetro siempre oscilará entre 0 y 1, pues se trata de una medida en tanto por 1 que es realizada por el Kinect en base a los puntos que puede detectar correctamente cuando se proyectan los infrarrojos.

con este criterio suelen ser puntos reducidos del cuerpo que no pueden ser detectados correctamente debido a la resolución de la cámara. Estos puntos indetectables son nueve en total: cintura, tobillos, muñecas, punta de los dedos (del índice normalmente) y clavículas. Como vemos, son puntos bastantes reducidos de nuestro cuerpo, y que para el seguimiento que realizaremos pueden ser ignorados por completo, teniendo en cuenta a posteriori las correcciones en nuestro código que fuesen necesarias si se diese el caso.

De esta manera, nos limitaremos a quince puntos útiles, los que se ven en la imagen, y que serán cabeza, cuello, hombros, codos, manos, torso, caderas, rodillas y pies, pudiendo localizarse en la posición esperada, aunque en la imagen 2D no coincida plenamente al tratarse de puntos tridimensionales que están siendo representados en coordenadas proyectivas (es decir, en solo dos dimensiones en vez de tres, teniendo en cuenta una corrección a la hora de hacer la conversión para que sea lo más precisa posible). Todos estos puntos podemos notar que coinciden en buena medida con articulaciones, a excepción de uno en concreto, el torso.

Torso se trata de un punto proporcionado por OpenNI que, aunque no coincide con ninguna articulación pues está situado en medio del cuerpo, nos será útil, pues tiene una característica única que no se da con los demás, y es que, es el único que no puede ser ocultado mientras realizamos el seguimiento de un usuario. Supongamos que colocamos unos de nuestros brazos detrás de nuestra espalda, ¿podríamos localizar dónde se encuentra el codo o la mano? No, se daría el fenómeno de oclusión explicado con anterioridad, y careceríamos de información de profundidad de las articulaciones, siendo imposible de realizar el seguimiento. Pero esto no es posible de realizar con el torso, no habrá manera de ocultar dicho punto al Kinect sin ocultarnos por completo, por lo que si lo hiciésemos estaríamos dejando de ser detectados y acabaríamos con nuestro seguimiento y por tanto la sesión que tendríamos en curso. Torso además se encuentra en el centro de nuestro cuerpo en todo momento aproximadamente, pero no coincide con nuestro centro de masa. Aún así, puede usarse en cualquier momento para calcular la orientación del cuerpo, como saber si alguien está inclinado hacia delante o atrás, si está girado hacía un lado, etc. En resumen, aunque no se trate de una articulación como los demás puntos, nos será de bastante utilidad.

Por otro lado, cabe comentar una peculiaridad con respecto a las manos y pies. Como ya hemos dicho, la fiabilidad con las que son medidas tanto muñecas como tobillos es despreciable por ser puntos muy pequeños, sin embargo, el

lugar en el que se miden tanto pies como manos no está mucho más alejado de estos, y con solo unos centímetros de diferencia, tendremos una medida lo suficientemente buena como para no echar en falta los anteriores puntos a la hora de comprobar una postura. Este es un dato que deberemos de tener en cuenta, especialmente en el caso de las manos, pues por ejemplo, al escribir con teclado, nuestras muñecas no tienen porque ser "despreciables" a la hora de ver si una postura es correcta o si nos estamos dañando la articulación al forzarla, pero que con esta versión del Kinect no podremos considerar.

5.4. Representación gráfica y ángulos.

En el desarrollo de la aplicación nos encargaremos de comprobar si una postura será correcta o no de acuerdo con la posición relativa de varios puntos del cuerpo entre sí. En un principio, dijimos que distinguiríamos entre "reconocimiento postural" y "ergonometría", pues el enfoque que le daremos sobre todo a la hora de programar no es el mismo. Supongamos que colocásemos los brazos abiertos, como si estuviésemos siendo crucificados: esta postura, si programásemos ahora una acción que se disparase al identificarla, sería a lo que nos referimos con "reconocimiento postural". Sin embargo, consideremos que queremos ver el ángulo con el que está "abierto" un brazo tomando como punto de referencia el codo, el punto que sería el vértice del ángulo, o que queremos saber la orientación de nuestro cuerpo(boca arriba, boca abajo, hacia la derecha, etc) o de algunas de sus partes (por ejemplo moviendo la cabeza hacia los lados e identificando dicho lado). En este caso, no estaríamos observando una postura en su totalidad, sino la posición u orientación de ciertas partes del cuerpo, y de acuerdo a esto, dispararíamos una acción si quisiéramos, siendo esto lo que llamaremos como "ergonometría". De una manera más sencilla, podría decirse que el "reconocimiento postural" es lo que incita a pensar intuitivamente, además de poder implicar varías condiciones de "ergonometría" al mismo tiempo que resultan en una condición mucho más restrictiva para disparar una acción. Aunque parezcan similares, realmente no lo son, pues pueden suponer dos enfoques diferentes al programar.

Ahora bien, nosotros nos encargaremos de comprobar si la postura de alguien es correcta en todo momento, pero, ¿tendríamos que observar si tenemos una postura o gesto adecuado continuamente?¿valdría con que se cumpliesen diversas condiciones por separado? En un principio podríamos pensar que la postura debe de ser perfecta siempre, sin embargo, el reconocimiento total de

esta no es lo que necesitamos. ¿Qué pasaría si alguien se sentase con la espalda recta pero la manera en la que coloca la cabeza no es la adecuada?¿Sería su postura completamente incorrecta? No. Aquí tratamos de buscar qué es lo que cada usuario hace mal y corregirlo, como ya hemos explicado, por lo que nos centraremos en realizar un estudio de la "ergonometría", en asegurarnos de que diversas partes del cuerpo de cada usuario mantienen unas formas correctas e informar de ello en caso contrario.

Una vez que tenemos claro que no vamos buscando un gesto corporal en su totalidad, sino el que ciertas partes mantengan una postura relativa adecuada, para realizar el estudio de la "ergonometría" nos deberemos de fijar en los **ángulos y orientaciones de las articulaciones** adecuadas. Cómo calcular dichos ángulos, qué valores deben de tener y cuándo y dónde se pueden calcular es lo que veremos a continuación.

Kinect es una cámara de profundidad, y como tal, captura información del entorno en 3D. Si queremos conseguir información sobre un punto del espacio, los datos que tendremos a nuestra disposición será tanto su orientación (en el caso de ser uno de los puntos concretos del esqueleto que suelen coincidir con articulaciones), como sus coordenadas en el espacio. Con estas coordenadas, podríamos crear un vector en el espacio que definiese su posición, y una vez lo tuviésemos, podríamos realizar cualquier operación utilizando las herramientas que nos proporciona el álgebra vectorial, centrándonos principalmente en el producto escalar.

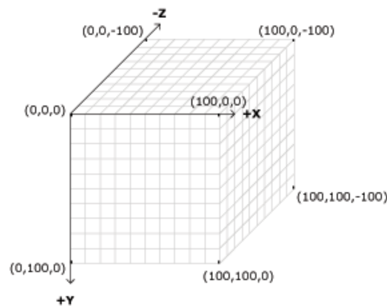


Figura 23: Sistema de coordenadas cartesiano en Processing, extraído de la web de processing.org

En Processing el sistema de coordenadas se encuentra definido como en la imagen, coincidiendo nuestro origen de coordenadas con la esquina superior izquierda de la pantalla en el caso de ser una imagen 2D, o en el centro de esta si estamos en una imagen 3D. En un principio pensaríamos que estamos en un sistema tridimensional al obtener información de una cámara de profundidad, sin embargo, esto no es exactamente así. Processing nos desplazará la cámara al centro de

la pantalla[21], pudiendo rotarla y desplazarla al trabajar con un sistema tridimensional, tal y como especifica la documentación oficial, sin embargo el Kinect no funciona de esta manera. Cuando nosotros capturamos una imagen con el Kinect, este se encarga de otorgarnos la profundidad de los 640x480 puntos de su resolución. Este valor de profundidad, lógicamente, podría interpretarse como el valor de la coordenada Z en el espacio, pero de acuerdo con el eje de coordenadas de Processing, "hacia adentro" de la pantalla irían valores negativos de Z y los valores que nosotros obtenemos asociados a cada píxel de nuestra resolución es positivo, ya que es una distancia real. La manera en la que se diferencian valores positivos y negativos de la coordenada Z consiste en su "tamaño" por pantalla, es decir, cuanto "mayor" sea el valor de z más "cerca" estará el objeto de nosotros, más grande parecerá en pantalla, y cuanto más pequeño, más lejos. Esto quiere decir que, aunque el sistema de coordenadas esté centrado en la imagen, el punto (0,0,0) no estará "pegado" a la superficie de la pantalla, sino "dentro" de esta. No hay que confundir esto con el efecto de acercar y alejar la cámara en un entorno tridimensional, que no es que haga al objeto menor o mayor, sino que nos situará en una distinta posición con respecto a este

¿Cómo podremos realizar los cálculos correctamente? Todo sistema de coordenadas es utilizado a la hora de realizar una representación, pero nosotros no necesitamos representar los vectores, aunque sí los puntos en el que se encuentran nuestras "articulaciones", pero sin embargo, lo haremos de una manera bidimensional. El porqué de una representación bidimensional reside en la interacción con el usuario final. Un entorno 3D, aunque simple, puede resultar confuso al usuario, sobretodo en lo referente al Kinect, puesto que no calcula un espacio tridimensional completo, sino solo aquel que puede observar, ya que tan solo tenemos una única cámara y adquirir información de todos los ángulos de visión resulta imposible, dándose lugar fenómenos como el de oclusión y el sombreado, pues la representación más fiel que podríamos hacer sería usando una nube de puntos.

[21] Cuando nos referimos a cámara en este contexto, se trata del punto desde dónde observaremos la representación gráfica que tenemos por pantalla, no al Kinect propiamente dicho.

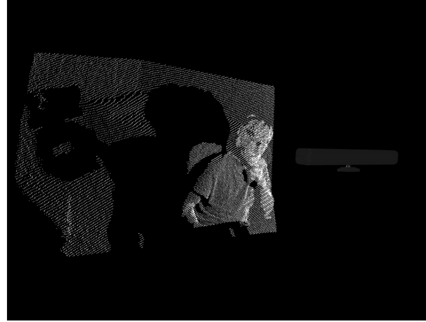


Figura 24: La representación 3D más simple que podremos realizar con Kinect se trata de una nube de puntos. Podemos ver un Kinect “virtual” en la imagen el cual hemos centrado en el origen de coordenadas en la cámara de Processing tras desplazarnos. Aquí también podemos observar el efecto de la coordenada Z, donde el usuario al estar más cercano a la cámara aparece a mayor tamaño que por ejemplo la pared de fondo.

Recurriremos entonces a una representación no sólo lo más intuitiva, sino también más simple sin necesidad de traslaciones de la cámara ni rotaciones para su correcta interpretación, la imagen de profundidad. En el caso de que queramos representar un punto que capturemos en el espacio, tal como una articulación, será necesario llevar a cabo una conversión, pasando de coordenadas reales a coordenadas proyectivas o bidimensionales. Una imagen de profundidad se encarga de mostrarnos el espacio capturado por la cámara pero de una manera completamente bidimensional, utilizando para la tercera dimensión un gradiente en el color utilizado, por ejemplo el gris. Claro que, necesitamos realizar cálculos en el espacio, por lo que debemos de tener en cuenta que la imagen bidimensional que se mostrará por pantalla tendrá tan sólo un cometido meramente informativo de cara al usuario, pero toda la matemática tras la aplicación se realizará en un espacio vectorial de base tres. Ahora bien, una vez tenemos en cuenta este matiz, ¿cómo calcularemos los ángulos de cada extremidad? ¿A qué nos referimos con extremidad? ¿Qué proceso seguiremos?

Extremidad	Puntos que la componen
Brazo derecho	Hombro, codo y mano derecha
Brazo Izquierdo	Hombro, codo y mano izquierda
Cuello	Cabeza, cuello y torso
Pierna Derecha	Cadera, rodilla y pie derecho
Pierna Izquierda	Cadera, rodilla y pie izquierdo

Cuadro 2: Extremidades definidas en la aplicación.

En esta aplicación definiremos como extremidad al conjunto de tres puntos seguibles del cuerpo que, generalmente, tienen similitud con una extremidad real. Definiremos los brazos como el conjunto de hombros, codos y mano, tres puntos identificables con el seguimiento de esqueleto de Kinect, pero también definiremos como extremidad al conjunto de hombro, cadera y rodilla, para poder comprobar la "rectitud" de la espalda. La manera por la que definiremos a una extremidad como tal reside en el cálculo de los ángulos de éstas en todo momento. En el caso del brazo, como ya se ha comentado, tomaremos el codo como el vértice del ángulo que forma nuestro brazo, teniendo hombro y mano como un punto a cada lado de este que nos permita definir dos vectores. Dichos vectores, uno del codo al hombro y otro del codo a la mano, serán multiplicados escalarmente para encontrar el ángulo de apertura.

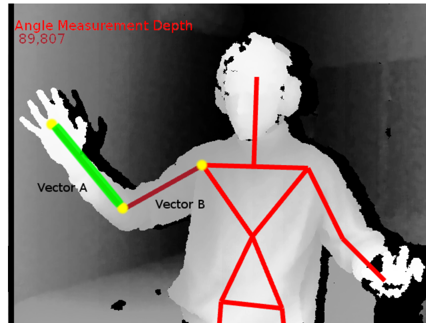


Figura 25: Imagen de como definiremos los vectores a multiplicar para calcular el ángulo de apertura de las extremidades.

Además, pese que hablamos de calcular el producto escalar, realmente esto no es lo que necesitamos, sino parte de la definición geométrica de este, y puesto que estamos en un espacio euclídeo tal como es el tridimensional y nuestros vectores también lo son, entonces:

$$\overline{A} \cdot \overline{B} = \|\overline{A}\| \|\overline{B}\| \cos(\alpha)$$

donde α representa el ángulo que forman los vectores A y B. De esta manera seguiremos los siguientes pasos a la hora de calcular:

1. Obtenemos la posición de los tres puntos (codo, hombro y mano en la imagen) y las almacenamos en un vector para cada uno de ellos

2. Definimos los vectores A y B mediante resta de vectores, restando a los

vectores de los puntos extremos de la “extremidad” el vector del punto intermedio. Esto lo haremos porque de otra manera no tendríamos los vectores desde el punto intermedio a los extremos, sino que se tendrían los vectores con respecto al origen de coordenadas (en este caso tridimensional). Tampoco es posible trasladar el origen de coordenadas al punto intermedio y luego obtener la información de los extremos, pues las librerías de OpenNI no tienen en cuenta este tipo de traslaciones a la hora de aportar información y nos indicarán los puntos con respecto al origen original.

3. Normalizamos los vectores A y B.

4. Multiplicamos los vectores A y B y calculamos el arcocoseno del producto.

5. Pasamos el resultado de gradianes a grados para que sea más fácil trabajar con él y que el usuario lo entienda mejor al mostrárselo por pantalla.

En el caso de tener más de una extremidad a observar, haremos lo propio para todas ellas, teniéndolas almacenadas todas en un array que iremos recorriendo para hacer los cálculos.

A la hora de calcular los ángulos además es necesario tener en cuenta un par de factores. Observando la anterior imagen podemos ver como la línea recta que define el vector A no parece estar centrada en nuestro brazo completamente, uniendo tan sólo codo con mano. En extremidades como los brazos y piernas se dará este suceso, la razón es simple y se ha explicado de pasada con anterioridad. Para que dicha línea coincidiese plenamente tendríamos que unir no codo y mano, sino que codo y muñeca, sin embargo esto no es posible. Muñecas y tobillos son puntos detectables por el Kinect, pero el valor de fidelidad (o confidence como se conoce generalmente) con la que estos son detectados por NITE, resulta prácticamente nulo en casi todo momento, por lo que deberemos de recurrir al punto más cercano, mano o pie, introduciendo un pequeño error en el cálculo. Aquí sin embargo, no entraremos a calcular una expresión matemática para corregirlo, pues dicho error no resulta significativo en la gran mayoría de los casos, por lo que a la hora de disparar una acción o dar un aviso, tan solo añadiremos un margen de error en base a diversas medidas que se han ido realizando al probar la aplicación.

Otro aspecto importante que nos podemos preguntar es que ángulo estamos midiendo exactamente. En principio hemos dicho que calcularemos el ángulo que forman dos vectores en el espacio, por lo que tan solo debe de haber una

solución posible. Concretamente si nos centramos en el caso del cuello pueden darse varios casos que nos puedan incitar a duda. Supongamos que tenemos el cuello hacia delante, si la "extremidad" a medir la hemos definido utilizando los puntos torso, cuello y cabeza, el ángulo que mediremos en este caso deberá de ser generalmente obtuso (a menos que doblemos tanto la cabeza que la peguemos al pecho), teniendo que, para que el cuello esté adecuadamente, tendremos que obtener una medida en torno a unos 180º. Si ahora, teniendo el cuello recto, lo inclinamos hacia un lado, el ángulo que obtendríamos sería el de esa inclinación, pero por mucho que lo intentemos, de esta manera no podremos tener el cuello realmente recto, y dicho ángulo obtenido será debido tanto porque la cabeza no esté correctamente erguida como por la inclinación lateral. A efectos prácticos la postura seguirá siendo incorrecta de una manera u otra, por lo que tan solo es un matiz a tener en cuenta.

Algo parecido pasará con la espalda, aunque en este caso será necesario no solo comprobar que la espalda está recta, sino que además no está inclinada hacía un lado, pues sí que puede darse el caso en el que al cumplirse una condición no se cumpla la otra, por lo que, para asegurarnos, haremos una doble comprobación. Miraremos el ángulo como en el caso de las demás extremidades y además, comprobaremos el ángulo que hay entre una vertical que sale del punto de la cadera y el vector superior.

Una vez tenemos definido como calcularemos los ángulos y cuales serán, debemos fijar unos valores nominales y las desviaciones permitidas, así como de que partes del cuerpo realizaremos seguimientos. Estas últimas serán cuello, brazos, piernas y espalda, seis en total al mismo tiempo. Los valores permitidos y sus desviaciones, calculadas experimentalmente en la mayoría de los casos, en grados serán:

- Cuello: 155º - 190º $\pm$5º
- Brazos: $\geq$ 85º
- Piernas: $\geq$ 85º
- Espalda : 85º - 115º . La espalda debe de mantenerse lo más recta posible, por lo que teniendo en cuenta la precisión de Kinect se ha llegado experimentalmente a que dentro de este rango será válido. Además el ángulo que deberá formar con la vertical no puede ser mayor de 15º.

Valores nominales obtenidos a raíz de lo estudiado en el apartado 3, **Ergonometría en la oficina**, que usaremos a la hora de hacer las comprobaciones en

nuestro programa.

5.5. Limitaciones de software y hardware

En el desarrollo de la aplicación, ya bien sea por la propia elección de librerías y drivers que hemos hecho, como por las características del Kinect, nos encontraremos con una serie de limitaciones, ya bien adheridas a las librerías por su mero planteamiento inicial, como al hardware debido a su diseño.

Comenzando por lo más esencial, en lo referente a drivers, SensorKinect no consigue explotar todas las capacidades del Kinect 1.0 dejando algunas de sus características secundarias con un uso bastante pobre o completamente nulo. Kinect 1.0 incluye un motor para ajustar levemente el ángulo de la cámara y su campo de visión, así como un array de micrófonos en toda la parte inferior, sin embargo, esto no es así en el resto de cámaras de diseño OpenNI. La cámara Asus Xtion o la PrimeSense Carmine, por ejemplo, carecen de cualquier tipo de motor, de manera que al ser SensorKinect un fork de los drivers de OpenNI para poder ser usados con la cámara Kinect 1.0, no es posible implementar dicha característica. El limitado uso de los micrófonos sin embargo, se debe a una pobre implementación, aunque tampoco necesitaremos hacer uso de ellos en ningún momento. Pero esto no es solo un problema con SensorKinect, y es que al no haber dichos elementos en las cámaras de OpenNI o no ser usados normalmente, en las librerías no solemos encontrar funciones o clases para permitir su uso, ya que en ningún momento se reconocen esas características, por lo que no había necesidad de implementarlas a nivel de librería si por driver no estaban activas.

A nivel de Software además contamos con las limitaciones de Processing. Como ya comentamos, Processing a la hora de traducir el código a Java puro y compilar, convierte a todas las clases de ficheros adicionales en clases internas del sketch principal, de manera, que el estado de funciones y variables esté limitado, no pudiendo tener de tipo static a menos que programemos en Java puro. Otro punto débil de Processing es que tan solo podremos utilizar librerías que hayan sido portadas para su Java simplificado, por lo que no nos valdría cualquiera que ya estuviese en Java puro a priori.

A nivel de Hardware nos encontraremos con un par de pegas importantes. La primera de ellas se trata tanto de la resolución como del tipo de cámara que es el Kinect. Kinect es una "cámara virtual", por lo que la resolución de 640x480 con la que trabajaremos realmente es menor, y se trata de una de 320x240 píxeles, menor que la cámara RGB si queremos trabajar a unos fps adecuados.

Sin embargo, a la hora de codificar la tomaremos como si fuese de 640x480, haciendo uso de la resolución "virtual" y no de la "real". La razón de ello tiene que ver con como funciona el Kinect y el proyector de infrarrojos que incorpora.

El Kinect crea su imagen de profundidad combinando el patrón de infrarrojos proyectado con lo que capta con su cámara IR. Ahora bien, el patrón se supone que se trata de 640x480, lo que nos da la resolución de la cámara de profundidad, pero la cámara de IR no puede calcular el valor de profundidad de todos los puntos del patrón (pese a que su resolución es de 1280x1024 píxeles), sino el valor de aquellos puntos que coincidan con sus propios píxeles. Además, como los puntos del patrón están distanciados entre sí un cierto espacio que varía con respecto a la distancia sobre la que se encuentren proyectados, no todos coincidirán con píxeles, sino solo una parte de ellos que no siempre es la misma, de manera que, para conseguir los 640x480 píxeles de resolución que tiene la cámara de profundidad teóricamente, se realizaran interpolaciones con los puntos que sí hayan sido debidamente detectados para construir la imagen que usaremos en nuestros cálculos. Este fenómeno no nos otorga un imagen 100 % real, pero sí suficientemente buena, por lo que es una limitación que podremos obviar, ya que al querer detectar el "esqueleto" humano, Kinect no se fija en un único punto, sino en un conjunto de éstos en el cuerpo de un usuario y determina su posición. Si intentásemos trabajar con reconocimiento de gestos o detectar objetos muy finos sí que resultaría imposible debido a la baja resolución, como en el caso de los puntos del esqueleto que Kinect no detecta con suficiente precisión, tales como las muñecas, tobillos o dedos. Esta limitación es una de las que se ha conseguido mejorar notablemente con el paso de una cámara de profundidad de luz estructurada a tiempo de vuelo en el Kinect 2.0 .

A priori estas serán las limitaciones más importantes con las que nos toparemos en la aplicación y que tendremos que tener en cuenta a la hora de programar.

5.6. Desarrollo de la aplicación

Al realizar la aplicación seguiremos unos procedimiento determinado, por lo que empezaremos planteándonos la estructura completa que queremos que esta tenga. Ya tenemos claro a modo general como queremos organiza la aplicación, así como los pasos que debemos de seguir para calcular los ángulos con la información extraída de la cámara, como hemos visto en la sección 5.4, así que lo primero será elaborar un diagrama de flujo completo del programa.

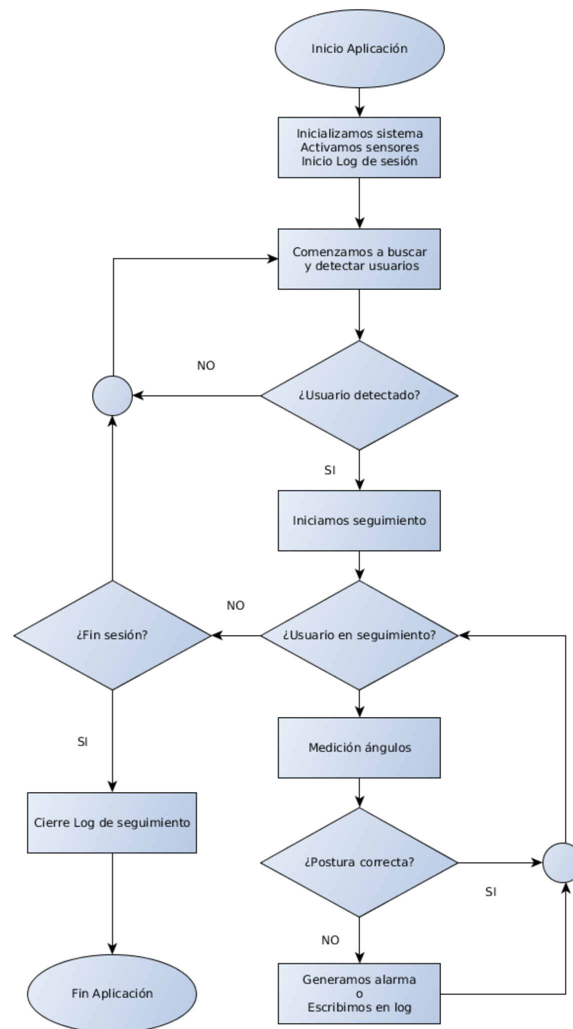


Figura 26: Diagrama de Flujo general de la aplicación

Observando el diagrama llegamos a que lo mejor será la utilización de diversas clases, para así facilitar la reutilización de código, así como una clase principal en dónde se encontrará el bucle principal del programa, además de porque estaremos trabajando en Java[22]. De esta manera acabaremos utilizando las siguientes clases:

- Una ***clase principal*** donde estará el bucle principal de la aplicación en continua ejecución, y que se encargará de hacer las llamadas a Kinect para comprobar la disponibilidad de usuarios, y en caso afirmativo, pasaría a hacer uso de las otras clases para llevar a cabo los cálculos y tareas necesarias en la aplicación.

- Una clase de nombre ***AngleMeasure***, encargada de gestionar todas las extremidades, así como el que hacer con ellas, desde calcular los ángulos, comprobar que los valores son los adecuados o representar su información por pantalla. AngleMeasure se servirá a su vez de dos clases más, Timer y Extremity.

- La clase ***Extremity***, creada para cada una de las extremidades definidas en el apartado 5.4 , será la que realice las funciones de más bajo nivel cuando AngleMeasure se lo pida. Esta última, AngleMeasure, será la que creará cada una de las instancias de Extremity y las guardará en una lista, de manera que a la hora de llevar a cabo cada una de las funciones anteriormente descritas, se servirá de funciones de Extremity realmente, llamando a cada una de ellas al recorrer la lista donde las almacena, y en donde estarán solo aquellas extremidades que tengamos en seguimiento.

- ***Timer*** será una clase que utilizaremos a modo de contador para comprobar cuando una extremidad está más de cierto tiempo en una posición incorrecta. Por defecto hemos definido el tiempo de margen antes de dar el aviso en 5 segundos, aunque podría cambiarse en cualquier momento fácilmente. Timer lleva una instancia asociada a cada instancia de Extremity que sea creada por AngleMeasure, aunque la que realmente se encargará de hacer uso de ella será esta última, pues AngleMeasure será la que lleve a cabo las comprobaciones, a fin de así llevar un archivo de log único asociado a todas las extremidades.

[22]En Processing más concretamente, como ya hemos comentador anteriormente.

- La clase ***PrintWriter*** es una clase específica de Processing que utilizaremos para crear el archivo de log en el que recoger las incidencias que se vayan produciendo cuando haya algún sujeto en observación.

Como se puede ver la cantidad de clases que utilizaremos no es muy numerosa, por lo que no habrá una gran complejidad, además con esta estructura podremos reutilizar el código en varias ocasiones, haciéndolo más legible, intuitivo y organizado. Todas estas clases las tendremos tan solo en 3 ficheros, a excepción de PrintWriter que al ser propia de Processing se importará automáticamente. A continuación podemos ver un pequeño diagrama de como están relacionadas estas clases entre ellas.

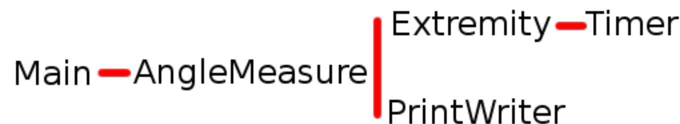


Figura 27: Diagrama simple que relaciona las clases y como hacen uso de ellas unas y otras.

Una vez tenemos correctamente identificada cada parte de la aplicación y qué debe de hacer cada una de ellas, pasaremos a la programación y a ver cada clase por separado.

5.6.1. Clase principal

Como ya se ha mencionado, será la que contendrá el bucle que estará en continua ejecución hasta que paremos el programa o lo cerremos manualmente. Además de esto, al ser el archivo principal de Processing en el que definiremos la función ***setup()***, será donde estableceremos el tamaño de pantalla (640x480 para que coincida con la resolución de Kinect que utilizaremos), iniciaremos el dispositivo e instanciaremos la clase AngleMeasure y le diremos que instancie a su vez las clases Extremity necesarias. Esta función setup() se ejecutará una única vez al iniciarse la aplicación siempre.

El bucle principal estará definido en la función ***draw()***, la cuál se ejecutará con cada refresco de pantalla, y en la que, según hemos definido en el diagrama de flujo, seguiremos los siguientes pasos:

- Actualizar la imagen que recibimos de Kinect.

- Realizar un borrado de la pantalla y volver a representar la nueva imagen obtenida de Kinect.
- Mostrar los datos necesarios referentes a los ángulos y las extremidades en un pequeño display encima de la imagen.
- Pediremos a Kinect la lista de usuarios que ha detectado. En caso de haber algún usuario comprobaremos si ya se está realizando un seguimiento de su esqueleto.
- En caso afirmativo calcularemos los ángulos de sus extremidades y comprobaremos si son los adecuados. Así mismo representaremos el esqueleto por pantalla, con un color específico para cada extremidad que habrá sido elegido aleatoriamente al instanciar su clase Extremity correspondiente.
- En caso negativo volveremos al primer punto y esperaremos a que se dé el caso positivo.

En la clase main() también tendremos la función drawSkeleton con la que dibujaremos el esqueleto que observa Kinect así como una serie de funciones de evento, una función para realizar capturas de pantalla al pulsar una tecla de ratón y tres funciones de evento que disparará Kinect en diversos casos:

- ***onNewUser*** - Se disparará al detectar Kinect a un nuevo usuario dentro de su campo de visión, en ella mostraremos por consola un par de notificaciones y además iniciaremos el proceso de *skeleton tracking* para dicho usuario, utilizando para ello la función ***startTrackingSkeleton(userId)*** de SimpleOpenNI.
- ***onLostUser*** - Se disparará al desaparecer un usuario o dejar de observarlo involuntariamente. No le daremos ninguna función específica, ya que no nos es necesario.
- ***onVisibleUser*** - Igual que el caso anterior, carecerá de uso en esta ocasión. Se disparará al ver un usuario, sin necesidad de llegar a identificarlo para hacer el seguimiento.

No necesitaremos ningún otro tipo de función especial en la clase principal, por lo que con las anteriormente citadas será más que suficiente. Para más detalle sobre el código este puede observarse en el anexo. Sin embargo, sí que utilizaremos más funciones concretas de SimpleOpenNI y Processing, no sólo en esta clase, sino a lo largo de toda la aplicación, entre las cuales tendremos:

- ***setMirror(boolean estate)*** - para espejar la imagen y hacerla más intuitiva para el usuario final.
- ***enableDepth()*** y ***enableUser()*** - para habilitar el uso de la cámara de profundidad de Kinect y la detección de usuario respectivamente.
- ***drawLimb(int userId, int a, int b)*** - representa en coordenadas proyectivas, tras hacer la conversión de 3D a 2D, los distintos vectores que constituyen el esqueleto detectado por Kinect. userId se trata del identificador de usuario asociado al usuario en cuestión del que queremos representar el vector, y los enteros a y b se utilizan para indicar los puntos inicial y final del segmento a representar, por ejemplo mano y codo. El entero asociado a userId es estrictamente necesario que sea el correcto para representar el vector en la posición adecuada, pues las posiciones de a y b se tomarán del usuario especificado.
- ***update()*** - actualiza la imagen obtenida por Kinect.
- ***depthImage()*** - obtiene la información referente a la imagen de profundidad de Kinect para poder representarla mediante la función image de Processing.
- ***getUsers(IntVector userList)*** - nos otorga la lista de usuarios detectados por Kinect y los almacena en una lista de enteros especial de Processing, IntVector, la cual ya lleva asociada que todos sus elementos serán enteros y no de ningún otro tipo.
- ***isTrackingSkeleton(int userId)*** - comprueba si el skeleton tracking para el usuario especificado por userId se está llevando a cabo.

Si volvemos de nuevo al diagrama de flujo podremos ver fácilmente en qué momento necesitaremos del uso de estas funciones.

5.6.2. Clase AngleMeasure

Como ya se ha comentado esta clase será la encargada de gestionar las extremidades específicas en seguimiento y que hacer con ellas. La clase será instanciada en la función setup() de la clase principal y accederemos a ella cada vez que queramos trabajar con las extremidades. AngleMeasure principalmente se trata de una lista con varias instancias de Extremity, la cual tendrá además las siguientes funciones:

- ***addExtremity(int topJoint,int midJoint,int lowJoint,int lowLimit, int highLimit)*** - instancia cada una de las extremidades definidas por los tres puntos o joints especificados, les asigna los ángulos límite de apertura de la extremidad y las agrega a la lista.

- ***getAngle(int userId)*** - calcula el ángulo de apertura de cada extremidad de su lista. Recorrerá la lista y llamará a a la función de Extremity del mismo nombre, para cada una de las extremidades, que seguirá el proceso ya comentado para obtener el valor que buscamos.

- ***displayAngle()*** - muestra por pantalla la información de los ángulos de apertura de todas las extremidades de la lista. Al diferencia de getAngle, no hará uso de una función similar en Extremity.

- ***drawExtremity(int userId)*** - dibuja las extremidades en seguimiento en un color específico para distinguirlas fácilmente. El color se elegirá aleatoriamente al instanciar el objeto Extremity. Al igual que getAngle hará uso de una función más específica en Extremity, limitándose aquí tan solo a recorrer la lista y a hacer las llamadas a la función para cada objeto.

- ***checkAngle()*** - comprobará que el ángulo obtenido para cada extremidad está dentro del rango que se especificó en cada uno de sus constructores originalmente. De no ser así iniciará el contador de cada extremidad y si es necesario, al pasar el tiempo límite de considerarse a una extremidad con una mala postura o posición incorrecta, dejará constancia de ello en el fichero de log.

AngleMeasure no utilizará más funciones propias, siendo éstas todas las necesarias. Además de crear en su constructor una lista para las extremidades también instanciará un objeto PrintWriter con el que creará el fichero de log para cada sesión. Como AngleMeasure es instanciada en el setup() de la clase principal, el log que creará será único para cada ejecución del programa, de manera que si se diese el caso de perder un usuario o de cambiarlo por otro, todas las incidencias que ocurriesen con éstos quedarían registradas en un único fichero, por lo que será necesario reiniciar la aplicación cada vez que se quiera cambiar a la persona a observar en cuestión. Mejorar este funcionamiento quedaría como un trabajo futuro en caso de interés, en este caso se ha considerado que al ser una aplicación con interés puramente académico se podría prescindir de dicha funcionalidad.

El log por su parte, seguirá una estructura bastante simple y fácil de interpretar para su posterior procesado. En él, especificaremos lo siguiente: el momento en que se dió lugar una incidencia en cuestión (indicando la fecha y hora exacta del día), la extremidad en cuestión de la incidencia (aunque no distinguiremos entre derecha o izquierda, pese a poder implementar dicha distinción facilmente, ya que experimentalmente se ha podido observar que por lo general cuando alguien se coloca indebidamente no lo hace tan sólo con la mitad de su cuerpo), y el valor del ángulo de apertura que ha hecho que se disparase la alarma y se acabase registrando en el log la incidencia.

5.6.3. Clases Extremity y Timer

Como se ha podido ver en el apartado de AngleMeasure, las funciones son prácticamente las mismas en nombre, aunque funcionalmente serán las que realicen realmente las operaciones que se buscan. Así, tendremos las funciones drawExtremity y getAngle con el funcionamiento ya descrito. Pero además, cada instancia de Extremity tiene una instancia de la clase Timer asociada como hemos podido ver, así como todos los datos referente a la extremidad en cuestión, desde el color que se asigna aleatoriamente para representarla, hasta las "joints" que la componen o los vectores que la definen.

Por su parte la clase Timer será bastante simple. Se encargará de extraer el tiempo actual, mediante la función millis() de Processing, de almacenarla y de ir comparando el tiempo que obtuvo al ponerse en marcha por primera vez con el de ese momento, de manera que activará una bandera si la diferencia transcurrida coincide con el tiempo que hemos definido como límite. Hará uso de tres funciones propias bastante auto-descriptivas, de nombre ***StartTimer()***, ***UpdateTimer()*** y ***ResetTimer()***.

Haciendo un repaso a todas estas clases explicadas anteriormente, vemos como podremos cumplir con todos los requisitos que queríamos en nuestra aplicación, por lo que solo hace falta pasar esta descripción en lenguaje natural a código como último paso. De nuevo, el código específico para cada una de estas clases lo podremos encontrar en el anexo, a continuación podremos observar una serie de ejemplos de ejecución de la aplicación en diferentes situaciones.

5.7. Ejemplos de ejecución

A continuación podemos observar algunas capturas de la aplicación en plena ejecución. Podremos observar como hemos mantenido una interfaz simple sin ningún tipo de complejidad, lo único que necesita el usuario es situarse delante de la cámara y nada más. Cada "extremidad", de las ya descritas, tiene asociado un color aleatorio, el cual no sólo servirá para representarla y distinguirla de las demás, sino que también será el color en el que por pantalla se mostrará el ángulo de apertura asociado a ella.

En este primer caso podemos ver a un usuario situado delante de la cámara sentado, sin ningún tipo de mesa delante u obstáculo. Debido a la proximidad no se muestra su cuerpo completo (los pies no se ven en su totalidad), y por tanto la observación de las extremidades inferiores no será la correcta.

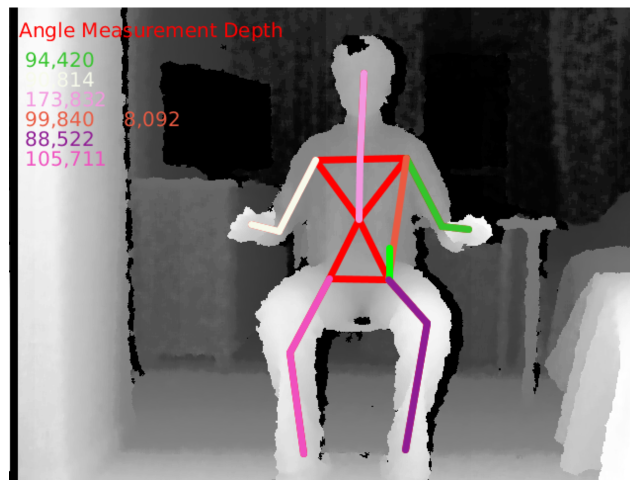


Figura 28: Ejemplo de ejecución 1. Usuario sentado frente a Kinect.

En este otro caso situamos la cámara desde una posición inferior a la anterior, teniendo que orientarla hacia arriba, y desde una posición superior, teniendo que orientarla hacia abajo, por lo que nuevamente habrá puntos en el cuerpo que no se detectarán correctamente mientras que otros sí según la posición. Especialmente en el caso del cuello, dónde desde un ángulo inferior el punto será más distinguible que desde cualquier otra posición. Esto es debido a que si colocásemos el Kinect en una posición elevada la barbilla, y la cabeza en general, producirían oclusión del punto del cuello.

Figura 29: Ejemplo de ejecución 2. Usuario sentado, Kinect en una posición baja.

Figura 30: Ejemplo de ejecución 3. Usuario sentado, Kinect en una posición alta.

Ahora observemos al usuario completamente de pie y observando su cuerpo entero. Obviamente habrá condiciones que se incumplirán, pero puede verse como la detección de los puntos del esqueleto es completa y más precisa que estando sentado.

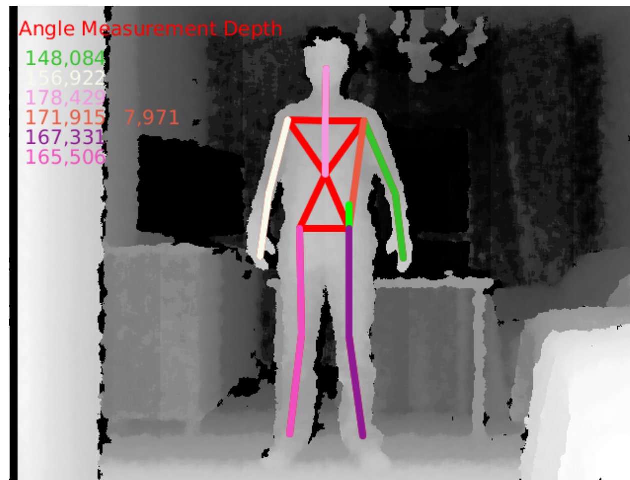


Figura 31: Ejemplo de ejecución 4. Usuario de pie frente a Kinect.

Finalmente coloquemos al usuario frente a Kinect y con un puesto de trabajo delante. Claramente con la aparición de obstáculos el Kinect 1.0 verá dificultada su tarea, pero aún así es capaz de localizar al usuario y realizar el seguimiento con una fiabilidad aceptable en la mayoría del tiempo.

Figura 32: Ejemplo de ejecución 5. Usuario sentado frente a Kinect con obstáculos entre él y la cámara.

Se llegaron a probar varias mesas, llegando a la conclusión de que cuanto más despejada por debajo y más fina resultase la tabla de ésta, más precisa sería la medida, pues de esta manera los puntos del esqueleto tendrían una mayor visibilidad. Por supuesto, es esencial que la superficie de la mesa no sea de cristal, o en caso de serlo que se encuentre totalmente tapada, para así evitar reflexiones indeseadas en los infrarrojos del proyector y que se pierda información de profundidad. Aquí podemos ver también un par de capturas en una mesa algo más equipada con un monitor, teclado y posa lápices, y cómo la detección sigue funcionando pese a estos obstáculos adicionales, que sin embargo están tan presentes en situaciones laborales.

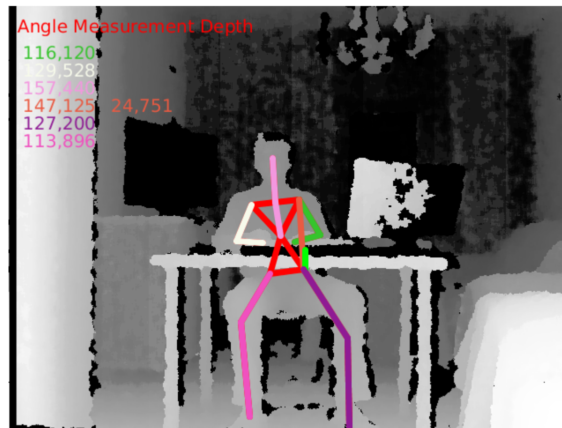


Figura 33: Ejemplo de ejecución 6. Usuario sentado en una mesa de trabajo con monitor y más objetos

En estos últimos dos ejemplos (figuras 33 y 34), además hemos podido observar como la detección del usuario cuando está sentado es algo más ineficaz, por lo que para poder realizar el seguimiento, es necesario que el usuario comience inicialmente de pie, y una vez localizado puede sentarse sin problema. Eso sí, hay que tener en cuenta siempre la posibilidad de perder el seguimiento, pues al movernos en la mesa podremos estar ocultando puntos inconscientemente.

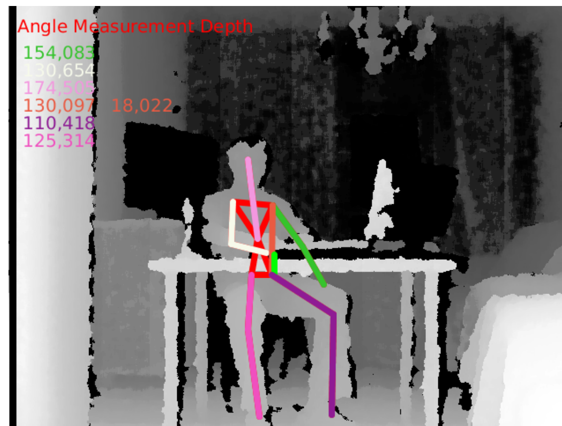


Figura 34: Ejemplo de ejecución 7. Usuario sentado en una mesa de trabajo con monitor y más objetos. En este caso ha variado ligeramente el ángulo de observación de la cámara y la posición del usuario.

Además de estos ejemplos también se han llevado a cabo varias pruebas para ajustar los valores límites, pues había variaciones importantes en algunos casos entre el valor teórico que debería de obtenerse y el calculado a raíz de los datos otorgados por Kinect 1.0 en todo momento provocado por los márgenes de error en las medidas de Kinect.

6. Resultados experimentales

A la hora de obtener una serie de resultados experimentales se sometió a un seguimiento a 4 personas diferentes, de distintas edades y estaturas, durante un tiempo de unos 4 minutos por usuario, para así conseguir una serie de resultados mínimamente representativos. Durante ese tiempo a los usuarios se les otorgó un libro a modo de entretenimiento, para que lo leyesen sin ningún tipo de preocupación y así, mientras que se encontraban en una situación relajada, observar todas las incorrecciones posibles en su postura. Los usuarios se encontraron en todo momento en la situación descrita al final de la última sección, sentados y en un puesto de trabajo más o menos despejado y con una mesa lo más simple posible.

Tras realizar los seguimientos de cada uno de ellos observamos los diferentes archivos de log, entre los que pudimos observar que las posturas incorrectas más comunes que se daban eran las siguientes:

- Espalda. La usuarios se mantuvieron erguidos la mayoría del tiempo, sin embargo, con el paso del tiempo comenzaron a relajarse, principalmente los hombros, adoptando posturas encorvadas, resultando en la incidencia más común entre la mayoría de los usuarios.

- Cuello. Por lo general los usuarios tendían a doblar el cuello más de lo necesario, echándolo hacia delante, algo que suele resultar en dolores cervicales y molestias si se mantiene la postura regularmente.

- Piernas. Mientras que en los casos anteriores los usuarios eran más conscientes de sus errores e intentaban corregir sus posturas, no era así con las piernas. Inconscientemente solían doblarlas o cruzarlas, resultando en molestias en las articulaciones.

En el caso de los brazos los usuarios no solían tener problemas con ellos, manteniéndolos extendidos en todo momento por comodidad, sin llegar a forzar las articulaciones. A continuación podemos ver un diagrama de barras con las incidencias para cada uno de los usuarios. En el gráfico podemos observar como las malas posturas tanto de espalda como de cuello son comunes a todos ellos por lo general, así como un despunte en uno de los usuarios en lo que se refiere a una mala postura de piernas.

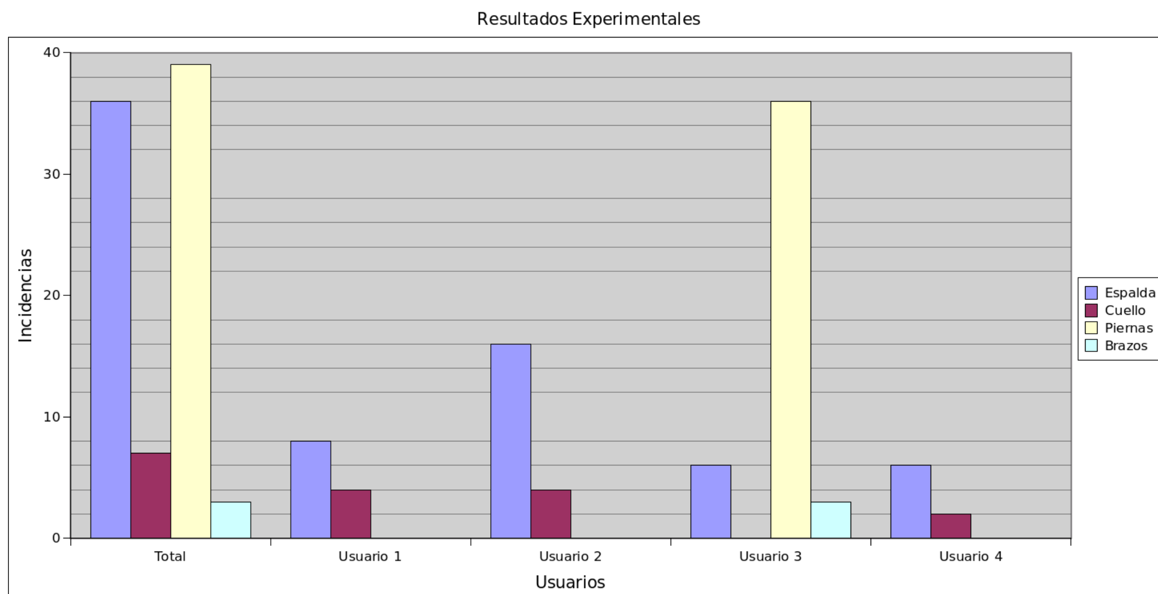


Figura 35: Gráfico de barras con los resultados obtenidos

Estos datos nos sirven para comprobar que lo más frecuente en una mala postura es no colocar la espalda de manera correcta, así como el cuello. En este montaje experimental se le entregó un libro a los usuarios y el experimento tan sólo duró unos minutos, aún así las incidencias que algunos de ellos produjeron en este breve periodo de tiempo dan lugar a entender que en caso de extrapolar a duraciones aún mayores, los resultados no harían más que empeorar. Si además nos basásemos en los datos anteriormente comentados de "El web de la Espalda", tendría sentido que las incidencias más comunes entre usuarios sean las de espalda, pues, tal y como se puede leer en la web, la mayoría de lesiones laborales se debe a malas posturas de ésta. Estos resultados eran de esperar como se ha venido comentando en varias ocasiones, la gente no suele adoptar buenas posturas, lo que acaba por pasarles factura a modo de lesiones y contracturas.

Resulta así mismo curioso que la mayoría de los usuarios mantenga los brazos en posiciones correctas al igual que las incidencias por parte de un usuario en lo referente a posturas de las piernas. Mientras que con los brazos se mantienen extendidos para trabajar, teclear o leer un libro en este caso, casi intuitivamente, sí que podemos ver como el usuario 3 resulta ser una persona inquieta, pues durante toda la sesión no paró de mover de un lado para otro las piernas y de

flexionarlas o colocarlas hacía atrás, forzando de esta manera las articulaciones, lo que al levantarse de la silla le provocase ciertas molestias. Además hemos de prestar atención al caso especial del cuello, el cual, como ya se ha comentado, puede estar afectado por una oclusión involuntaria de la propia cabeza del usuario, causando incertidumbre en la medida y por lo tanto en los datos recogidos.

¿Es importante mantener una buena postura?¿Tendemos a colocarnos indebidamente? Sí. A la vista de los resultados está que si realmente adoptásemos posturas correctas, las incidencias que habríamos detectado serían básicamente despreciables en todas las extremidades definidas en la aplicación, pero nada más lejos de la realidad. Ya sea de manera consciente o no, mantener una postura correcta es una tarea difícil, por lo que lo más normal es no hacerlo, significando una alta probabilidad de dolencias a largo plazo. Dolencias que sólo podemos evitar mejorando nuestros hábitos de manera continuada e insistente. Con esta aplicación hemos creado una herramienta para poder obtener resultados como los aquí expuestos, sin embargo, el qué hacer con dicha información quedará en manos del usuario, pues por el momento sólo sirve para concienciar de la existencia de posturas incorrectas. Esta información podría ser procesada de múltiples maneras, desarrollando sistemas de monitorización más avanzados, correctores de posturas, mobiliario inteligente o simplemente utilizada a la hora de desarrollar estudios más complejos.

7. Conclusiones y trabajos futuros

7.1. Conclusiones

A lo largo de este trabajo hemos podido ver las diferentes formas de interacción e interfaces físicas desarrolladas en la historia para facilitar la comunicación y control de los ordenadores por parte del ser humano. Frente a la omnipotente combinación formada por teclado y ratón, hemos descubierto desde diversos tipos de guantes de lo más estrafalario, hasta sistemas de control basados en cámaras de lo más sofisticado. Cámaras cuyas posibilidades van más allá del entretenimiento en los videojuegos, que son aquellas que nos interesan. Escaneado tridimensional o reconocimiento y seguimiento de usuarios, por ejemplo, son solo algunas de las características que nos ofrecen estos sistemas ignorados por el público generalísta al que en ocasiones se encuentran enfocados, y es que en los últimos años el mundo del videojuego se ha visto plagado por dispositivos de esta índole, desde el WiiMote de Nintendo hasta el Kinect de Microsoft. Este último, el Kinect, con su aparición en el mercado, rápidamente ganó popularidad, no por su contribución como sistema de control en el mundo del videojuego, por el que pasó sin pena ni gloria, sino por todo aquello que podía hacer en manos de un programador. La comunidad creció rápidamente, aparecieron múltiples librerías, y proliferaron "hacks" de todo tipo para hacer uso de las posibilidades que ofrecía el dispositivo en cuestión.

En esta ocasión nos hemos adentrado en el mundo de este tipo de dispositivos, concretamente en el del ya mencionado Kinect y el de las cámaras de profundidad, descubriendo tanto su arquitectura y métodos de funcionamiento, como explotando sus capacidades mediante la programación no solo de la aplicación que aquí se presenta, sino de muchos y curiosos pequeños "hacks" que se fueron realizando durante todo el estudio del trabajo en pos de familiarizarse con el aparato en cuestión. Gracias a ello, se ha ganado conciencia de qué son las cámaras de profundidad, cómo funcionan, la tecnología que emplean y el gran abanico de posibilidades que nos presentan, las cuales hemos intentado explotar aquí. Pero no sólo esto, y es que otro de los objetivos principales de este trabajo era el uso de componentes comerciales de bajo coste, y es por esa misma razón por la que se decidió utilizar el Kinect, pues gran parte de su popularidad se debe a su relación calidad-precio, y más aún hoy en día, dónde es posible adquirir uno de estos sistemas por incluso menos de 50 euros en el mercado de segunda mano.

Bien es cierto que la comunidad es muy amplia y aún sigue activa, sin embargo, cierta información no resulta fácil de encontrar en ocasiones, llegando a veces a ser inexistente. Y puede que sorprenda, pues ya se ha mencionado en diversas ocasiones la gran actividad que había en torno al dispositivo, pero para un novato en el tema puede resultar algo caótico debido a esto mismo, el gran exceso de información con el que nos podemos encontrar, y que no siempre resulta fiable. Windows SDK, OpenKinect, OpenNI, OpenCV, PCL y así como multitud más de librerías, drivers, frameworks y APIs se acaban entrelazando entre ellas ofreciendo todo tipo de posibilidades que en principio no estaban pensadas para el dispositivo, como un reconocimiento de manos mucho más depurado y preciso utilizando OpenCV, por ejemplo. Pero también se llega a dar que aparentemente parezca que hay información de sobra, y luego encontrar el usuario documentaciones oficiales bastante escuetas y resumidas, siendo totalmente necesario recurrir a los foros de consulta, como es el caso de SimpleOpenNI. Pese a ello, esto no significa que resulte menos interesante, sino todo lo contrario, puede acabar siendo una motivación más, invitándonos a experimentar y descubrir por nosotros que puede hacer o no hacer el Kinect 1.0, que además es lo que se ha conseguido aquí en cierta medida.

Y es que hablando de limitaciones, nos hemos encontrado tanto a la hora de buscar información como a nivel de software y hardware. Algunas de las más destacables, sobre todo a nivel de hardware, han sido sin duda alguna el efecto de sombreado, las limitaciones en rango de la cámara, la resolución limitada por la tecnología empleada o las incompatibilidades con algunos sistemas operativos al programar.

También tenemos restricciones curiosas, algunas de lo más variopintas y poco intuitivas, como la que nos encontramos a la hora de medir el ángulo del cuello. Con este ángulo en concreto surgía el siguiente problema durante la práctica: según el lugar dónde se coloque la cámara, la medida a obtener será la adecuada o completamente errónea. Esto tiene que ver con la posición en la que Kinect identifica el punto central, el del cuello, a la hora de medir el ángulo, pues si observamos el entorno con la cámara en una posición más elevada que nuestra propia cabeza, no veremos correctamente el punto del cuello y las medidas serán erróneas, pero si lo hacemos desde un posición por debajo de la cabeza, calcular el ángulo resulta bastante más preciso y no habrá problemas mayores. Este resultado, el cuál a priori no debería de darse, tan solo ha podido ser descubierto a base de prueba y error, pues los cálculos realizados no anticipaban dicha situación, y las propias funciones de librería se supone que deben calcular la

posición del punto correctamente sin darse ningún tipo de oclusión. De esta manera, se llega a la conclusión de la importancia que tiene una correcta posición de la cámara a la hora de realizar un seguimiento correcto.

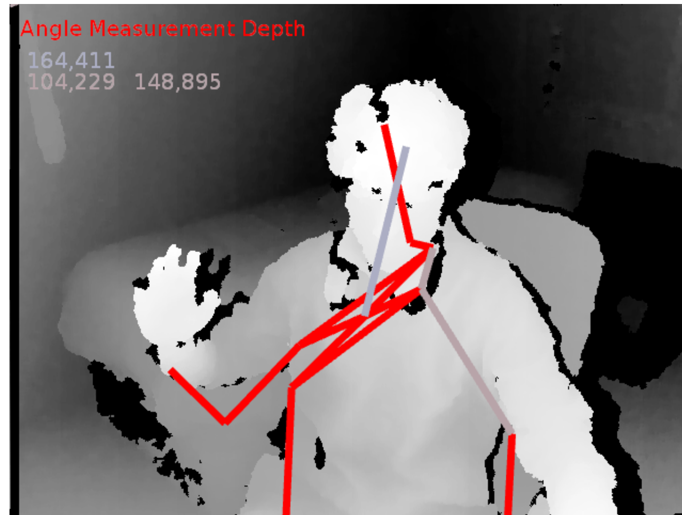


Figura 36: Una incorrecta posición de la cámara, así como el no mostrar correctamente todos los puntos de los que se realiza un seguimiento, puede acabar en resultados totalmente erróneos y sin sentido debido a la sensibilidad del Kinect.

El entorno de trabajo también supone una problema importante. Debemos poder observar al usuario en todo momento, por lo que la manera en la que este tenga organizado su puesto frente al ordenador, así como el tipo de mesa o la localización de la cámara en este entorno, será crucial, ya que resulta primordial evitar ocultar los puntos o "joints" lo máximo posible para que el sistema pueda funcionar correctamente, de no ser así, obtendríamos valores completamente aleatorios y sin sentido (que en ocasiones es más común de lo que parece), pues el esqueleto del usuario no podría localizarse debidamente y su representación por pantalla resultaría totalmente caótica. De esta manera el realizar el total seguimiento del cuerpo, pues si, como se ha comentado anteriormente, necesitamos colocar la cámara por debajo del cuello para que detecte bien dicho punto, esto puede dar lugar a que se oculten otras "joints", con lo que conseguir una precisión total con Kinect 1.0 resultará bastante complejo, por no decir imposible en algunos casos.

Figura 37: Diagramas sobre como debería de ser un entorno de trabajo para poder realizar el seguimiento de la forma más precisa posible. Este entorno fue el que se preparó en las pruebas en todo momento.

Todas estas limitaciones sin duda deberían ser superadas en futuras posibles ampliaciones o continuaciones del proyecto, como las que comentamos a continuación.

7.2. Trabajos futuros

En la realización de este trabajo se partió completamente de cero, hasta conocer el funcionamiento del Kinect y obtener la capacidad de desarrollar aplicaciones que hiciesen uso de sus características, llevando a cabo una serie de tareas de aprendizaje durante todo el proceso, que sin duda en el futuro podrían seguir aumentando. Estas tareas o trabajos futuros suponen una lista de ampliaciones de lo más variadas, algunas de ellas podrían ser las siguientes:

- **Portabilidad a otros dispositivos OpenNI.** En varias ocasiones se ha comentado que la elección de librerías, por ejemplo, se debía a la posibilidad de reutilizar el código aquí elaborado en otras cámaras de diversos fabricantes, sin embargo, puesto que no tenemos acceso a dichos dispositivos, dicha comprobación queda como tarea pendiente a verificar.
- **Utilización de dos cámaras de profundidad.** El efecto de sombreado y oclusión que tenemos continuamente podría ser reducido si en lugar de una única cámara utilizásemos dos o más. Con la posibilidad de capturar el espacio desde diversos ángulos y con diversos campos de visión, podríamos realizar un "escaneado" más fiel y obtener información acerca de esas áreas que aparecen sombreadas cuando utilizamos únicamente un dispositivo. Con la información obtenida de múltiples cámaras reduciríamos este

efecto considerablemente, y como OpenNI permitía el uso de al menos dos dispositivos simultáneos, ésta sería una ampliación interesante de cara al futuro.

- **Estudio y uso del dispositivo Kinect 2.0 .** En las fases tempranas de búsqueda de información de este trabajo, incluso antes de decantarnos finalmente por este, nos encontramos con el nuevo y mejorado modelo de Kinect, y bastante más caro, el cuál estaba a punto de salir a la venta en su versión para PC, con el SDK correspondiente. Kinect 2.0 continúa donde el original se quedó, mejorando todo aquello que supone una limitación en el primer modelo como resolución, localización de puntos del esqueleto, oclusión y sombreado, alcance, etc. así como las características ya existentes de este, gracias al nuevo tipo de tecnología utilizada. Además nos aporta funciones nuevas de manera automática, como el medir el pulso cardíaco o incluso reconocimiento facial de manera sencilla a la hora de programar. Sin embargo, el uso de este dispositivo puede que se aleje del objetivo de bajo coste que buscamos.

- **Comparación con librerías alternativas.** A la hora de elegir unos drivers y librerías optamos por aquella que nos otorgaba la función más útil y precisa para nuestro cometido. Pendiente quedaría de verificar que las alternativas disponibles son realmente menos eficientes o si nos aportan funciones similares o no y hasta que punto OpenKinect y Kinect SDK son útiles para lo que aquí hacemos.

- **Mejora en la medida de los ángulos en la aplicación.** En nuestra aplicación nos vimos forzados a utilizar ciertos puntos no óptimos para el cálculo de los ángulos que estábamos observando, recurriendo a las manos o pies en vez de a muñecas y tobillos. En principio esto se debía a la poca fiabilidad con la que OpenNI era capaz de identificar dichos puntos, de manera que nos resultaban inservibles. Resultaría interesante encontrar una manera de mejorar dicha fiabilidad de muñecas y tobillos, o si no fuese posible, de encontrar un dispositivo o librería que nos lo permitiese. La solución más rápida pasaría por migrar la aplicación a Kinect SDK y utilizar el Kinect 2.0 con sus funciones específicas, pues este nuevo modelo entre sus muchas novedades, incorpora un mejorado seguimiento así como más puntos del esqueleto observables, entre los que se encuentran los mencionados. De esta manera conseguiríamos una medición bastante más

realista y sin necesidad de tener en cuenta dicha desviación si se diese el caso.

- **Seguimiento de varios usuarios.** En nuestra aplicación, debido a su carácter académico, tan solo nos hemos centrado en identificar a un único usuario y observarle, sin embargo, con el Kinect 1.0 tenemos la oportunidad de hacer lo propio con un segundo individuo. Si quisiéramos incluso aumentar ese número, podríamos recurrir al Kinect 2.0, el cual nos da la oportunidad de realizar el seguimiento de hasta seis personas al mismo tiempo.
- **Entorno de trabajo.** Actualmente necesitamos que el espacio quede bien despejado y sin obstáculos necesarios si es posible para realizar el correcto seguimiento del usuario, sin embargo, quedaría pendiente encontrar una manera de poder reducir este espacio a uno más cerrado y diminuto sin alterar el funcionamiento, lo cual parece pasar por utilizar otro modelo de cámara de mayores prestaciones, teniendo como referente más inmediato, de nuevo, el Kinect 2.0.
- **Mejora en el diseño de la interfaz de usuario.** Cambios en el aspecto gráfico para hacerla más atractiva de cara al usuario final, ya que al tratarse de una aplicación de mero carácter académico no se ha hecho demasiado hincapié en ello, pues no resultaba primordial.

Como vemos, todavía hay multitud de cosas que en las que se podría indagar, tanto a nivel de software como hardware, por lo que los conocimientos aquí presentados para familiarizarnos con este tipo de dispositivos podría servir como base de cara al futuro. Futuro en el que sin lugar a dudas se seguirá trabajando con cámaras de profundidad, pudiendo llegar el momento en el que su uso no se limite tan sólo a videojuegos y a algunos modelos de televisores, sino que se extienda a ámbitos más avanzados de los que hay ahora mismo disponibles en el mundo civil. Ciertamente uno de los campos en los que más se usa este tipo de cámaras hoy en día es en el de la robótica[23], su implantación como interfaz de control cada vez está más presente, por lo que adelantarse a ese momento

[23]Uno de los varios usos propuestos sería el propuesto por Mishra, A.K. y Meruvia-Pastor, O. en su artículo Robot arm manipulation using depth-sensing cameras and inverse kinematics, de 2014, públicado por el IEEE (http://ieeexplore.ieee.org) , o el del artículo Apple detection algorithm for robotic harvesting using a RGB - D camera del International Conference of Agricultural Engineering, AgEng 2014 (http://www.geyseco.es/geystiona/adjs/comunicaciones/304/C02290001.pdf).

podría suponer una ventaja de cara al futuro, la cuál no debemos dejar escapar, y explotar todas las posibilidades de las cámaras de profundidad, que sin lugar a dudas y como hemos podido ver, no son pocas ni mucho menos.

Referencias

[1] Occupational Safety & Health Administration. Computer workstations - good working positions, Diciembre 2014. URL: https://www.osha.gov/SLTC/etools/computerworkstations/positions.html.

[2] avin2. Sensorkinect - primesensor modules for openni, Mayo 2012. Github page. URL: https://github.com/avin2/SensorKinect.

[3] Ross Bencina. P5 glove developments, Agosto 2010. URL: http://www.simulus.org/p5glove/.

[4] Greg Borenstein. *Making Things See*. Make: Books. O'Reilly Media, Inc., primera edición edition, Enero 2012. URL: http://makingthingssee.com/.

[5] Craig Chapple. Next xbox leak reveals kinect 2 specs, Febrero 2013. URL: http://www.develop-online.net/news/next-xbox-leak-reveals-kinect-2-specs/0114096.

[6] Andrew Davison. Kinect: Open source progprogram secrets, Septiembre 2012. URL: http://fivedots.coe.psu.ac.th/~ad/kinect/.

[7] Ben Fry and Casey Reas. Processing, Mayo 2014. URL: https://processing.org.

[8] G.Calin and V.O. Roda. Real-time disparity map extraction in a dual head stereo vision system. *Latin American applied research*, 37(1), 2007. URL: http://www.scielo.org.ar/scielo.php?script=sci_arttext&pid=S0327-07932007000100005.

[9] ASUSTeK Computer Inc. Xtion pro live, Enero 2015. Xtion Depth Cameras. URL: http://www.asus.com/es/Multimedia/Xtion_PRO_LIVE/.

[10] iPiSoft. Depth sensors comparison, Mayo 2013. URL: http://wiki.ipisoft.com/Depth_Sensors_Comparison.

[11] Fundación Kovacs. El web de la espalda - adictos al ordenador, Diciembre 2013. URL: http://www.espalda.org/divulgativa/su_espalda/adictos.asp.

[12] Fundación Kovacs. El web de la espalda - trabajadores, Diciembre 2013. URL: http://www.espalda.org/divulgativa/su_espalda/trabajadores/workers.asp.

[13] Hartmann K. Langmann B. and Loffeld O. Depth camera technology comparison and performance evaluation. In *Proceedings of the 1st International Conference on Pattern Recognition Applications and Methods*, pages 438–444. SciTePress, 2012.

[14] Daniel Lau. The science behind kinects or kinect 1.0 versus 2.0. Blog, Noviembre 2013. URL: http://www.gamasutra.com/blogs/DanielLau/20131127/205820/The_Science_Behind_Kinects_or_Kinect_10_versus_20.php.

[15] Marcal Montserrat. CaracterÃsticas kinect 2. Blog, Enero 2014. URL: http://www.kinectfordevelopers.com/es/2014/01/28/caracteristicas-kinect-2/.

[16] Alejandro Murillo. Diferencias entre kinect xbox 360 y kinect for windows. Blog, Marzo 2012. URL: www.kinectfordevelopers.com/es/2012/03/01/diferencias-entre-kinect-xbox-360-y-kinect-for-windows/.

[17] Neoteo.com. P5 glove: Guantes de realidad virtual, Diciembre 2008. URL: http://www.neoteo.com/p5-glove-guantes-de-realidad-virtual-14368/.

[18] Regents of the University of California. Ergonomic program, Diciembre 2014. Múltiples páginas de la web. URL: http://ergonomics.ucr.edu/.

[19] okreylos. Kinect 2.0. Blog, Mayo 2013. URL: http://doc-ok.org/?p=584.

[20] piedar. libfreenect - drivers and libraries for the xbox kinect device on windows, linux, and os x, Diciembre 2014. Github page. URL: https://github.com/OpenKinect/libfreenect.

[21] PrimeSense. 3d sensor carmine1.08 specifications, Diciembre 2012. URL: http://www.openni.ru/rd1-08-specifications/index.html.

[22] PrimeSense. Nite 2.2.0.11, Octubre 2013. URL: http://www.openni.ru/files/nite/index.html.

[23] PrimeSense. Opopen - the standard framework for 3d sensing, Enero 2014. URL: http://www.openni.ru.

[24] OpenKinect project. Openkinect, Marzo 2012. URL: http://openkinect.org/wiki/Main_Page.

[25] Segaretro. Activator, Febrero 2015. URL: http://segaretro.org/Activator.

[26] SEPRUS Servicio Prevención Riesgos Laborales de la Universidad de Sevilla. Postura sentado confortable, Junio 2011. URL: https://r2h2.us.es/prevencion/index.php?page=i-pildoras-ergonomia.

[27] Structure. Openni 2 sdk binaries & docs, Mayo 2014. URL: http://structure.io/openni.

[28] Wikipedia. Gesture recognition, Diciembre 2014. URL: http://en.wikipedia.org/wiki/Gesture_recognition.

[29] Wikipedia. Openni, Diciembre 2014. URL: http://en.wikipedia.org/wiki/OpenNI.

[30] Wikipedia. Power glove, Diciembre 2014. URL: http://en.wikipedia.org/wiki/Power_Glove.

[31] Wikipedia. Ergonomía, Febrero 2015. URL: http://es.wikipedia.org/wiki/Ergonom%C3%ADa.

[32] Wikipedia. Kinect, Febrero 2015. URL: http://en.wikipedia.org/wiki/Kinect.

[33] Wikipedia. Lidar, Enero 2015. URL: http://en.wikipedia.org/wiki/Lidar.

[34] Wikipedia. Processing (programming language), Enero 2015. URL: http://en.wikipedia.org/wiki/Processing_%28programming_language%29.

[35] Wikipedia. Stereo cameras, Enero 2015. URL: http://en.wikipedia.org/wiki/Stereo_cameras.

[36] Wikipedia. Time-of-flight camera, Enero 2015. URL: http://en.wikipedia.org/wiki/Time-of-flight_camera.

A. Anexo I: Código

A.1. Clase principal : aplicacion_ergonomia.pde

```
import SimpleOpenNI.*;

SimpleOpenNI kinect;

//Class we are gonna use to measure the angles we might be
    interested
AngleMeasure medidas; //medidas contains all the extremities
     we'll use, it creates its own array to do that

float angle = 0.0; //This will keep track of the angle
    formed in the arm for now

//Declare the limit angle values
int AngArmMin = 85;
int AngArmMax = 200;
int AngLegMin = 85;
int AngLegMax = 200;
int AngNeckMin = 155;
int AngNeckMax = 190;
int AngBackMin = 85;
int AngBackMax = 115;

void setup(){

  size(640,480); //Right now we are working in 3D to do all
      the math

  kinect = new SimpleOpenNI(this);
  kinect.setMirror(true); // to make it more intuitive
  kinect.enableDepth();  //We need the depth data and user
      data to work with the skeleton data
  kinect.enableUser();

  medidas = new AngleMeasure(kinect);
  //topJoint, midJoint, lowJoint
  //It's the right/left side of the screen, not the right/
```

```
    left arm, if we mirror or not it will change

  //Joints for the right arm
  medidas.addExtremity(SimpleOpenNI.SKEL_RIGHT_SHOULDER,
      SimpleOpenNI.SKEL_RIGHT_ELBOW,SimpleOpenNI.
      SKEL_RIGHT_HAND,AngArmMin,AngArmMax);
  //Joints for the left arm
  medidas.addExtremity(SimpleOpenNI.SKEL_LEFT_SHOULDER,
      SimpleOpenNI.SKEL_LEFT_ELBOW,SimpleOpenNI.
      SKEL_LEFT_HAND,AngArmMin,AngArmMax);
  //Joints for the neck
  medidas.addExtremity(SimpleOpenNI.SKEL_HEAD,SimpleOpenNI.
      SKEL_NECK,SimpleOpenNI.SKEL_TORSO,AngNeckMin,AngNeckMax
      );
  //Joints for the "right part" of the back
  medidas.addExtremity(SimpleOpenNI.SKEL_RIGHT_SHOULDER,
      SimpleOpenNI.SKEL_RIGHT_HIP,SimpleOpenNI.
      SKEL_RIGHT_KNEE,AngBackMin,AngBackMax);
  //Joints for the right leg
  medidas.addExtremity(SimpleOpenNI.SKEL_RIGHT_HIP,
      SimpleOpenNI.SKEL_RIGHT_KNEE,SimpleOpenNI.
      SKEL_RIGHT_FOOT,AngLegMin,AngLegMax);
  //Joints for the left leg
  medidas.addExtremity(SimpleOpenNI.SKEL_LEFT_HIP,
      SimpleOpenNI.SKEL_LEFT_KNEE,SimpleOpenNI.SKEL_LEFT_FOOT
      ,AngLegMin,AngLegMax);

  //It's 2D but we can still create the font.
  PFont font = createFont("Verdana", 20);
  textFont(font);
}

void draw(){

  kinect.update();
  background(0);
  image(kinect.depthImage(),0,0);

  //Heads-up display
  fill(255,0,0);
```

```
  text("Angle␣Measurement␣Depth", 10, 30);
  medidas.displayAngle(); //It will cycle through a list
      with all the "extremities" we are measuring displaying
      the angles

  //First we start by identifying an user
  IntVector userList = new IntVector();
  kinect.getUsers(userList);
  //In case we detect an user
  if (userList.size() > 0){
    int userId = userList.get(0); // We get the int
        associated with the user

    //If we've identified the user and we are tracking its
        skeleton then we can proceed
    if( kinect.isTrackingSkeleton(userId)){

      stroke(255,0,0);
      drawSkeleton(userId);//We are gonna draw the skeleton
          in red, then the extremities in a random color to
          see what we are measuring
      medidas.drawExtremity(userId);

      //Now we can the function to get the angle of each "
          extremity"
      medidas.getAngle(userId);
      medidas.checkAngle();

    }
  }
}

void drawSkeleton(int userId){
  strokeWeight(7);

  //the OpenNI version draws on projective coordinates, so
      we use our own function to draw in 3D if necessary

  kinect.drawLimb(userId, SimpleOpenNI.SKEL_HEAD,
      SimpleOpenNI.SKEL_NECK);
```

```
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_NECK,
      SimpleOpenNI.SKEL_LEFT_SHOULDER);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER,
      SimpleOpenNI.SKEL_LEFT_ELBOW);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_ELBOW,
      SimpleOpenNI.SKEL_LEFT_HAND);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_NECK,
      SimpleOpenNI.SKEL_RIGHT_SHOULDER);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER,
      SimpleOpenNI.SKEL_RIGHT_ELBOW);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_ELBOW,
      SimpleOpenNI.SKEL_RIGHT_HAND);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER,
      SimpleOpenNI.SKEL_TORSO);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER,
      SimpleOpenNI.SKEL_TORSO);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_TORSO,
      SimpleOpenNI.SKEL_RIGHT_HIP);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_TORSO,
      SimpleOpenNI.SKEL_LEFT_HIP);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_HIP,
      SimpleOpenNI.SKEL_RIGHT_KNEE);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_KNEE,
      SimpleOpenNI.SKEL_RIGHT_FOOT);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_HIP,
      SimpleOpenNI.SKEL_LEFT_KNEE);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_KNEE,
      SimpleOpenNI.SKEL_LEFT_FOOT);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_HIP,
      SimpleOpenNI.SKEL_LEFT_HIP);

}

//Mouse callbacks
void mousePressed(){
  save("App2D.png");
}

//user tracking callbacks!
void onNewUser(SimpleOpenNI curContext,int userId){
```

```
  println("onNewUser␣-␣userId:␣" + userId);
  println("\tstart␣tracking␣skeleton");

  curContext.startTrackingSkeleton(userId);
}

void onLostUser(SimpleOpenNI curContext,int userId){ //
    curContext is actually kinect,here is just a local name

  println("onLostUser␣-␣userId:␣" + userId);
}

void onVisibleUser(SimpleOpenNI curContext,int userId){

  //println("onVisibleUser - userId: " + userId);

}
```

A.2. Clases AngleMeasure y Extremity: AngleMeasure.pde

```
//This file contains the two classes
//AngleMeasure and Extremity
//First should go the Extremity class, so that the
    AngleMeasure class can use it later

//Auxiliary class
class Extremity{

  int topJoint;
  int midJoint;
  int lowJoint;
  PVector topVector;
  PVector midVector;
  PVector lowVector;
  PVector vVector; //For the Vertical line

  color fillColor; //The color will be assigned randomly to
      every extremity
  color strokeColor;

  float lowLimit = 0.0;
  float highLimit = 0.0;
  float angle = 0.0;
  float angle2 = 0.0; //This angle will indicate the
      deviation according to the vertical coming up from the
      hip in the back

  Timer counter; //Timer asociated to the extremity
  int tLimit = 5;

  SimpleOpenNI kinect;

  Extremity(SimpleOpenNI context, int topTemp, int midTemp,
      int lowTemp, float llTemp, float hlTemp){

    kinect = context;
    fillColor = strokeColor = color(random(255),random(255),
        random(255));
```

```
    counter = new Timer(tLimit); //5 seconds limit right now

    topJoint = topTemp;
    midJoint = midTemp;
    lowJoint = lowTemp;

    lowLimit = llTemp;
    highLimit = hlTemp;

    topVector = new PVector();
    midVector = new PVector();
    lowVector = new PVector();
    vVector = new PVector();
  }

  void getAngle(int userId){

    //We calculate each one of the Vectors refering them to
        the midJoint
    kinect.getJointPositionSkeleton(userId, midJoint,
        midVector);
    kinect.getJointPositionSkeleton(userId, topJoint,
        topVector);
    topVector.sub(midVector);
    topVector.normalize();
    kinect.getJointPositionSkeleton(userId,lowJoint,
        lowVector);
    lowVector.sub(midVector);
    lowVector.normalize();

    //Now we multiply and get the angle
    angle = acos(topVector.dot(lowVector));
    angle = degrees(angle);

    //Special case regarding the back of the user
    if (midJoint == SimpleOpenNI.SKEL_RIGHT_HIP || midJoint
        == SimpleOpenNI.SKEL_LEFT_HIP)
     {
       //Create a vertical line and calculate the angle
           formed with the topVector
```

```
      PVector auxVec = new PVector(midVector.x,midVector.y,
          midVector.z);
      vVector = new PVector(midVector.x, midVector.y+100,
          midVector.z); //3D vector
      vVector.sub(auxVec);
      vVector.normalize();
      angle2 = degrees(acos(topVector.dot(vVector)));
    }
}

void drawExtremity(int userId){

  strokeWeight(7);
  stroke(strokeColor);
  //We are working in 2D with the depth camera, so we draw
      in projectives, to do that we use SimpleOpenNI's
      drawLimb function.
  kinect.drawLimb(userId,topJoint,midJoint);
  kinect.drawLimb(userId,midJoint,lowJoint);

  //We have to draw in proyective coordinates
  if (midJoint == SimpleOpenNI.SKEL_RIGHT_HIP || midJoint
      == SimpleOpenNI.SKEL_LEFT_HIP) //Special case for the
      back
   {
      vVector = new PVector(midVector.x, midVector.y+100,
          midVector.z); //3D vector
      PVector auxvVector = new PVector();
      PVector auxmidVector = new PVector();
      kinect.convertRealWorldToProjective(vVector,
          auxvVector);
      kinect.convertRealWorldToProjective(midVector,
          auxmidVector);
      stroke(0,255,0);
      //converted to projectives already
      line(auxvVector.x, auxvVector.y,auxmidVector.x,
          auxmidVector.y);
   }
}
```

```
}

//Main class in this file
class AngleMeasure{

  SimpleOpenNI kinect;
  ArrayList extremities;
  PrintWriter log;

  AngleMeasure(SimpleOpenNI context){

    kinect = context;
    extremities = new ArrayList();
    log = createWriter(year()+","+month()+","+day()+":"+hour
        ()+":"+minute()+":"+second()); //So we create a log
        for each session
  }

  void addExtremity(int topJoint,int midJoint,int lowJoint,
      int lowLimit, int highLimit){

    Extremity part = new Extremity(kinect,topJoint, midJoint
        , lowJoint, lowLimit, highLimit);
    extremities.add(part);
  }

  //We only need it to calculate the angle formed by each "
      extremity", we don't need to return anything
  void getAngle(int userId){

    for(int i = 0; i < extremities.size(); i++){
      Extremity part = (Extremity)extremities.get(i); //
          casting is needed
      part.getAngle(userId);
    }

  }

  //Checks if the value for the angle is within the limits,
      if not it will write in the log the time and the name
```

```
    of the part
void checkAngle(){

  String exType = null;

  for(int i = 0; i < extremities.size(); i++){
    Extremity part = (Extremity)extremities.get(i); //
        casting is needed
    if(part.angle <= part.lowLimit || part.angle >= part.
        highLimit || part.angle2 >= 15){
      //Start the timer for that part if is not already
          running
      println("Antes␣de␣comprobar␣el␣timer");
      if(part.counter.running == false)
          part.counter.StartTimer();
      else if(part.counter.UpdateTimer() == true){
        //Write to log and reset the timer

        println("Pasados␣5␣segundos,␣escribimos␣y␣
            reseteamos");
        part.counter.ResetTimer();

        if( part.midJoint == SimpleOpenNI.SKEL_NECK)
          exType = "Cuello";
        else if (part.midJoint == SimpleOpenNI.
            SKEL_RIGHT_HIP || part.midJoint ==
            SimpleOpenNI.SKEL_LEFT_HIP)
          exType = "Espalda";
        else if (part.midJoint == SimpleOpenNI.
            SKEL_RIGHT_KNEE || part.midJoint ==
            SimpleOpenNI.SKEL_LEFT_KNEE)
          exType = "Piernas";
        else
          exType = "Brazos";

        println(year()+"/"+month()+"/"+day()+"/"+"\t"+
            hour()+":"+minute()+":"+second()+"\t"+exType+"
            \t"+part.angle);
        log.println(year()+"/"+month()+"/"+day()+"\t"+
            hour()+":"+minute()+":"+second()+"\t"+exType+"
```

```
                \t"+part.angle);
            log.flush(); //We make sure to write in the log
                everything
          }
        }
        else
          //If there was an error and it disappeared before
              the Timer went off, we reset the timer and stop
              it
          part.counter.ResetTimer();
    }
  }

  void displayAngle(){ //We don't need anything in the
      Extremity class

    for(int i = 0; i < extremities.size(); i++){
      Extremity part = (Extremity)extremities.get(i); //
          casting is needed
      fill(part.fillColor); //Each extremity has 1 random
          color asociated to it
      text(part.angle,10, 60+i*20); //We display one angle
          measure in each line
      if (part.midJoint == SimpleOpenNI.SKEL_RIGHT_HIP ||
          part.midJoint == SimpleOpenNI.SKEL_LEFT_HIP)
        text(part.angle2,10 + 100, 60+i*20); //We display
            one angle measure in each line
    }
  }

  void drawExtremity(int userId){
     for(int i = 0; i < extremities.size(); i++){
      Extremity part = (Extremity)extremities.get(i); //
          casting is needed
      part.drawExtremity(userId);
     }
  }

}
```

A.3. Clase Timer: Timer.pde

```
//This file contains the class Timer
//Timer will be used to count up to a certain number of
    seconds
//Uses the millis function to get the actual time and
    compares it
//with an initial sample to find the time that has passed

class Timer {

  int limit; //in seconds
  int init; // in milliseconds
  int seconds;

  boolean running; //To know if it's running the counter
  boolean state; //To know if it has reached the limit

  //Constructor
  Timer(int limit){
    this.limit = limit;
    running = false;
    state = false;
    seconds = 0;
  }

 //Start the Timer
 void StartTimer(){
   init = millis();
   running = true;
 }

  //Update the Timer, in seconds
  boolean UpdateTimer(){
    int aux = 0;
    aux = millis();
    seconds = aux - init; //Right now is the value in
        milliseconds, so we convert to seconds
    seconds = seconds/1000;
    if ( seconds == limit)
```

```
      state = true;
    println(seconds);
    return state;
  }

  //Reset the Timer
  //Doesn't restart the process, it just restarts the value
      to 0
  void ResetTimer(){
    init = 0;
    running = false;
    state = false;
  }
}
```