Interfaz de comunicación entre Blender y Kinect

Memoria final de “Practicas Externas en CITSEM”

30-1-2015

Autor: Ignacio Gómez-Martinho González

Tutora: Martina Eckert

Grupo de trabajo: Tecnologías para realidad aumentada

Resumen

En esta memoria se describe el desarrollo de un complemento software que permita a Blender, el programa de animación de juegos y vídeo, implementar un control por movimientos utilizando la cámara Kinect. El objetivo principal es abrir las puertas al desarrollo de aplicaciones adaptadas a personas con movilidad reducida, de modo que puedan manejarse con gestos o movimientos leves. Se ha estudiado el protocolo OSC (*Open Sound Control*) como medio de transmisión de datos de Kinect, y se ha programado el receptor adaptado a su estructura. Además del sistema de control, se ha añadido una herramienta de grabación que permite almacenar animaciones en la memoria de Blender. A modo de ejemplo, se han desarrollado varias aplicaciones sencillas en las que se ha implementado satisfactoriamente el control por movimientos.

Abstract

This paper describes the development of a software complement for Blender, the animation program for games and video, allowing it to implement motion control based on the Kinect camera. The main target is to allow the development of applications adapted for disabled people, so they can be controlled with gestures or slight movements. OSC protocol (Open Sound Control) has been studied as a way of transmission for Kinect data; the receiver has been programmed according to the OSC structure. Besides from the control system, a recording tool has been added, allowing to store animations in Blender’s memory. As an example, a few simple applications have been developed, where motion control has been satisfactorily implemented.

1. Introducción

El CITSEM lleva ya varios años investigando las posibilidades de la tecnología de realidad aumentada. Ésta consiste en la integración de la información digital y el mundo físico, a menudo sustituyendo las interfaces habituales en informática (ratones, teclados, monitores) por otras basadas en los movimientos o la visión humana, a las que llamamos interfaces naturales. Éstas, además, se pueden adaptar para facilitar la interacción entre los usuarios y los sistemas informáticos corrientes. Puede ser muy útil para personas, por ejemplo, con dificultades motrices, que con el uso de una interfaz natural podrían manejar un ordenador mediantes leves gestos.

En estos meses se ha trabajado con el Kinect, una cámara desarrollada por Microsoft capaz de reconocer los movimientos y la posición espacial del usuario. Se comercializó inicialmente como accesorio para la consola Xbox 360, pero desde entonces Microsoft ha puesto su entorno de desarrollo a disposición de los usuarios. De entre todas las posibilidades del Kinect, este trabajo se centra en la creación de una interfaz que permita su conexión con Blender.

Blender es un software libre de modelado y animación 3D. Permite crear, desde cero, vídeos con calidad fotorrealista, pero también la programación de aplicaciones interactivas de cualquier tipo, siendo los videojuegos su uso más popular. Su funcionamiento se basa en una visión tridimensional del entorno, o Ventana 3D, por la que el usuario puede desplazarse libremente y hacer todas las modificaciones que desee a su contenido: objetos, luces, sistemas de partículas, etc. Construido este entorno, se puede optar por realizar una animación en una escala de tiempo, o bien por programar una "lógica de juego" que dicte el funcionamiento de una aplicación. Las aplicaciones de Blender son ejecutadas por su motor de juegos, el Blender Game Engine.

En realidad, las aplicaciones de Blender no tienen por qué tener un objetivo lúdico y se les puede dar cualquier uso imaginable. El objetivo de estas prácticas es proporcionar una base que permita incorporar el uso de Kinect a aplicaciones desarrolladas por terceros. No es un terreno inexplorado, pero hasta el inicio de estas prácticas las soluciones funcionales para lograr esta conectividad eran de pago. Ahora se dispone de una alternativa de software libre desarrollada íntegramente en el CITSEM, así como de un conocimiento práctico de su funcionamiento interno.

El punto de partida ha sido el software libre KinectOSC[1], que accede al entorno de desarrollo del Kinect y obtiene los datos de posición, en cada instante de tiempo, de veinte articulaciones del cuerpo del usuario. Utilizando el protocolo OSC[2] (*Open Sound Control*), un sustituto del MIDI, los envía al puerto del ordenador especificado. El software desarrollado es un complemento software o *addon* para Blender que actúa como receptor de esos datos y los traslada a movimiento real en el entorno 3D de la aplicación que se esté desarrollando.

A lo largo de este documento se expondrá el estado inicial de las aplicaciones con Blender y Kinect, para después detallar el desarrollo del *addon*. Se describirán los ejemplos prácticos realizados y se hará un breve comentario sobre las posibles líneas de trabajo futuras.

2. Estado del Arte

En el momento de la realización de este trabajo existen ya varias herramientas de conexión entre Blender y Kinect, centradas en unas u otras características. En este apartado se citarán las más significativas.

La primera distinción que se puede hacer entre los distintos softwares disponibles es el entorno de desarrollo en el que se basan, es decir, la librería de programación que utilizan para acceder al Kinect. Está el kit de desarrollo oficial[3] (SDK) distribuido por Microsoft, pero antes de que éste fuera liberado se crearon varias alternativas de código abierto, como el SDK de OpenNI (*Open Natural Interaction*). Ésta es una organización sin ánimo de lucro creada por varias empresas dedicadas a las interfaces naturales, y tiene kits de desarrollo tanto para Kinect como para otros dispositivos de captura de movimiento.

En OpenNI se basa el programa que ha servido de modelo principal a este proyecto: NI-Mate[4], desarrollado por Delicode. La aplicación principal es un emisor que extrae los datos del Kinect y los retransmite utilizando el protocolo OSC o MIDI, a elegir. Hay disponibles varios *addons* que instalan el receptor en los distintos programas con los que es compatible NI-Mate, Blender entre ellos. Este *addon* permite implementar el movimiento en Blender, tanto en la Ventana 3D como en la ejecución de un juego. En ambos casos el *addon* recibe en cada instante de tiempo los datos del usuario, y crea una serie de objetos en el entorno 3D que representan la posición de sus articulaciones.

Las razones para realizar este trabajo, existiendo ya el NI-Mate, son principalmente dos: en primer lugar, se trata de un programa de pago que exige la compra de una licencia. Se pretende desarrollar una aplicación de software libre. Y en segundo lugar, el apoyo al entorno OpenNI decayó a principios de 2014, por lo que se considera más conveniente disponer de un sistema basado en el SDK de Microsoft.

De entre las soluciones gratuitas basadas en el SDK, una de las más prominentes parece ser el *Blender Loop Station*, o Bloop[5]. Consiste en un *addon* para Blender que permite controlar el movimiento de un modelo de personaje utilizando las manos del usuario, o “marionetista”. Su mayor limitación, según los intereses de esta investigación, es que está orientado a la animación de personajes y no al control de una aplicación en el Game Engine.

3. Informe de desarrollo

A continuación se detallarán el desarrollo y las características finales del *addon* programado para Blender. Siguiendo el orden cronológico del proceso, se explicarán los datos y herramientas iniciales, las necesidades del programa y su implementación secuenciada.

El trabajo está centrado en incorporar a Blender los datos de movimiento obtenidos por Kinect. Existen múltiples herramientas software con las que leer estos datos de la cámara, basados en diferentes protocolos de comunicación y entornos. Para este trabajo se ha considerado este aspecto de la comunicación suficientemente cubierto de momento, y se centrará exclusivamente en el lado receptor. Esto, por supuesto, requiere un conocimiento del lenguaje del emisor que supone la primera fase de la investigación y que se tratará en el siguiente subapartado:

3.1. El emisor: KinectOSC.

De entre las diversas opciones de acceso a Kinect, se ha optado por el software libre KinectOSC. Éste se basa en el kit de desarrollo oficial distribuido por Microsoft en lenguaje C#, cuya estructura no es de interés para este informe. Los datos recogidos se envían mediante protocolo OSC (*Open Sound Control*). Éste envía paquetes de longitud variable a un puerto seleccionable por el usuario. Es un protocolo sin conexión, por lo que los mensajes se envían tanto si hay una aplicación escuchando en ese puerto como si no.

El análisis de los paquetes recibidos en un receptor de prueba, codificado en Java, permitió determinar la estructura de los mensajes de KinectOSC. La sintaxis básica de OSC incluye ocho bytes especificando el tipo de mensaje: en este caso, un *bundle*. Y en este caso le siguen ocho bytes de marcado temporal, y a continuación un número entero de cuatro bytes que indica la longitud del campo de datos propiamente dicho. Es en éste campo en el que viajan los datos de posición. KinectOSC puede escribirlos de varias formas diferentes, según las opciones que se muestran en la Figura 1.

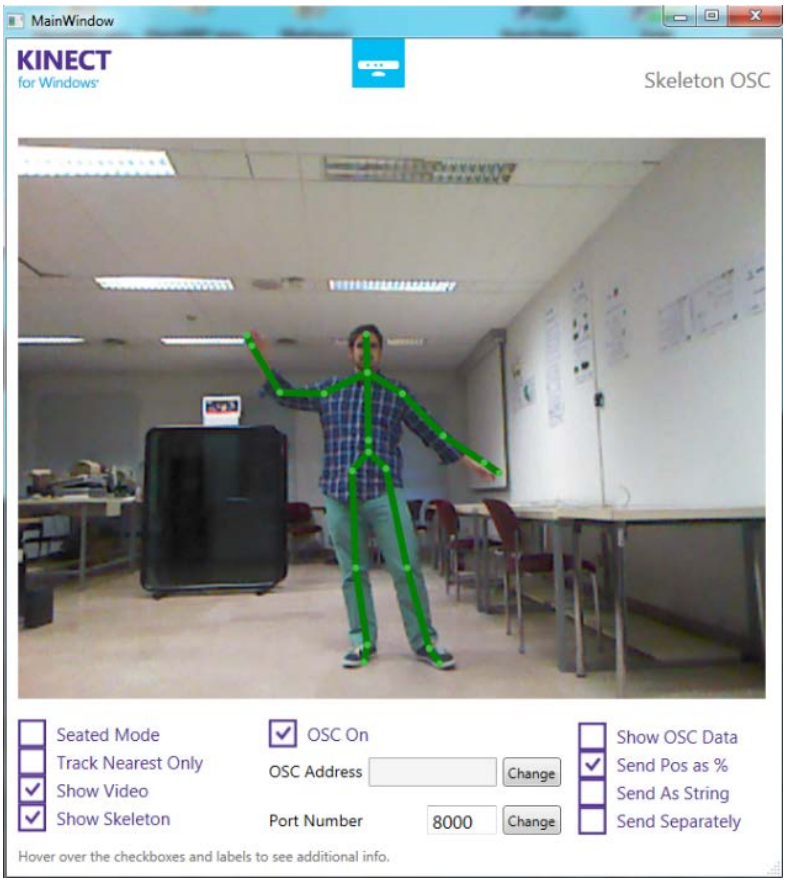


***Fig. 1**. Interfaz de KinectOSC.*

La primera opción que ofrece KinectOSC es enviar las coordenadas como una cadena de texto (*String*) o como números decimales (*Float*). Dentro de esta opción, se pueden enviar las tres coordenadas XYZ seguidas o enviar un paquete OSC con varios campos de datos, uno por cada coordenada. En los tres casos, las coordenadas están precedidas por una cadena de texto que indica la articulación a la que pertenecen. En la Figura 2 se puede ver el contenido de los paquetes, con los *floats* representados en código hexadecimal. Se observa que, después de la etiqueta, aparecen uno o más caracteres que indican el tipo de datos que vienen a continuación: "s" para una cadena de texto, "f" para un *float* y "fff" para tres *floats* seguidos.

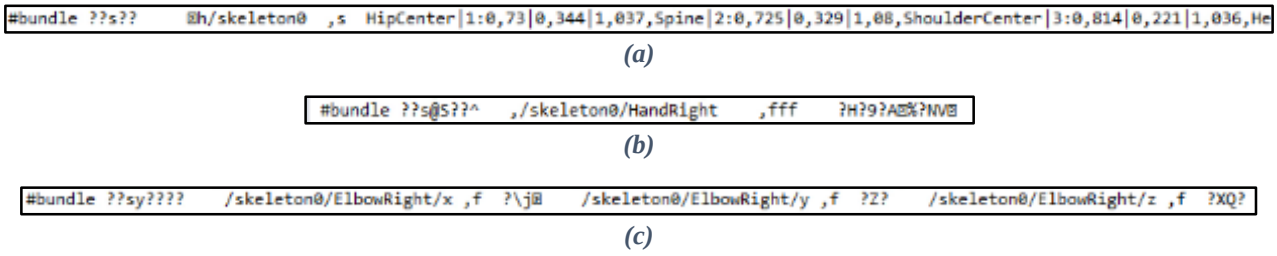
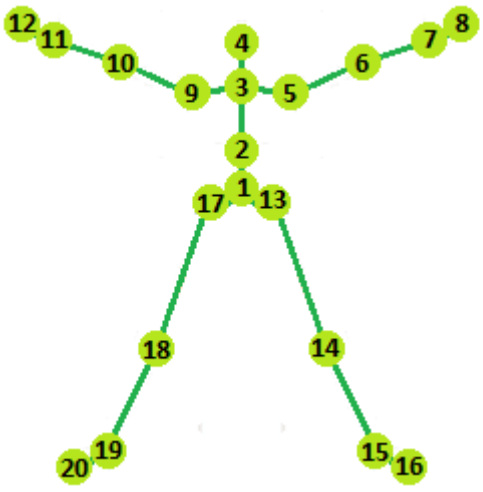


Fig. 2. *Mensajes enviados por KinectOSC.* ***a.-*** *En modo* string. ***b.-*** *En modo* float. ***c.-*** *En modo* float, *por separado.*

El modo *string* se ha juzgado poco eficiente para la transmisión, ya que mezcla texto con datos numéricos en una sola cadena y la posición de los datos es variable en función del número de decimales. Por lo tanto, el receptor codificado en Python, cuyo código se puede consultar en el anexo A, descarta este tipo de mensajes. Para los otros dos modos, la librería PythonOSC[6] permite leer cadenas de texto o números *float* de forma automática, calculando internamente las posiciones de los bytes en el paquete. No se ha conseguido importar la librería al entorno de Blender, por lo que se ha optado por copiar manualmente las funciones necesarias en el código del receptor. Éste determina si los tres datos se están enviando seguidos ("fff") o separados ("f") para alternar de forma adecuada las lecturas de s*trings* y de *floats*, obteniendo así el nombre de la articulación y los números XYZ.



1	HipCenter
2	Spine
3	ShoulderCenter
4	Head
5	ShoulderLeft
6	ElbowLeft
7	WristLeft
8	HandLeft
9	ShoulderRight
10	ElbowRight
11	WristRight
12	HandRight
13	HipLeft
14	KneeLeft
15	AnkleLeft
16	FootLeft
17	HipRight
18	KneeRight
19	AnkleRight
20	FootRight

Fig. 3. *Nombres y orden de retransmisión de las articulaciones en KinectOSC.*

De entre las tres coordenadas que se envían, Z es la única que expresa un valor en metros: el de la distancia desde el Kinect hasta la articulación, y se obtiene usando el sensor de profundidad. X e Y expresan, respectivamente, la columna y la fila de un píxel de la señal de vídeo del Kinect. Esta señal tiene 640 píxeles de ancho por 480 de alto, y la opción por defecto en KinectOSC es expresar el valor normalizado de las coordenadas, entre 0 y 1, aunque también se puede enviar el valor absoluto. Este concepto queda representado con más claridad en la Figura 4.

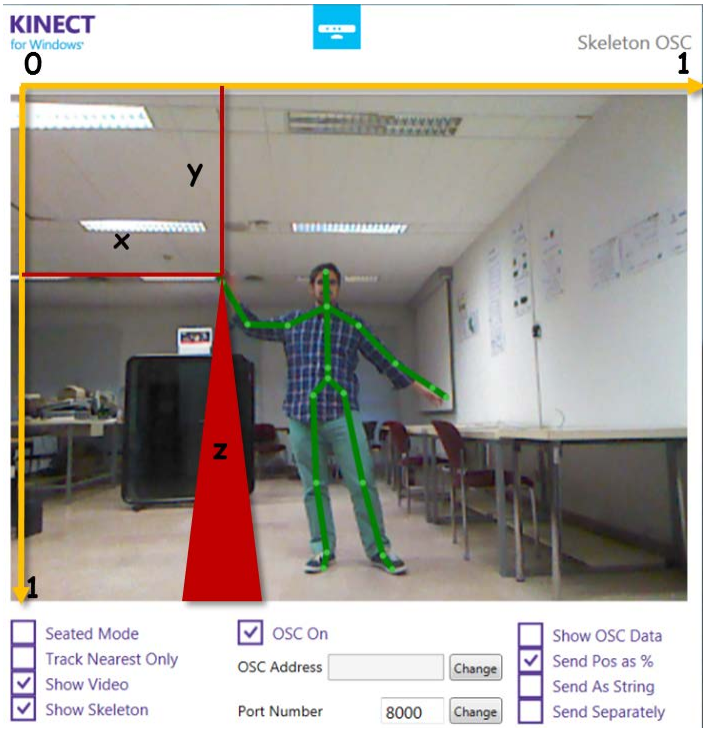


Fig. 4. *Coordenadas de la articulación* HandRight.

Este sistema se corresponde, entonces, con la visión cónica de la cámara del Kinect, y tiene la consecuencia indeseable de que los valores de X e Y experimentarán variaciones más grandes cuanto más cerca esté el usuario de la cámara. El entorno virtual de Blender es un espacio 3D isométrico, por lo que habrá que operar los valores recibidos para obtener las coordenadas XYZ reales.

3.2. El receptor: el *addon* en Blender.

Una de las muchas herramientas de edición que incorpora Blender es una sencilla interfaz que permite escribir y ejecutar *scripts* en lenguaje Python[7]. Se pueden ejecutar directamente en la Ventana 3D o bien incorporarlos a la lógica de juego para que se ejecuten en el momento que se desee dentro de la aplicación. Mediante Python se puede acceder a prácticamente toda la estructura interna de Blender[8], por lo que no solo se puede dar órdenes a los objetos virtuales del juego sino añadir al propio software todas las nuevas funciones que se desee (dentro de sus límites computacionales, por supuesto).

En este trabajo se pretende crear una herramienta que pueda ser utilizada en múltiples aplicaciones, por lo que un *addon* instalable[9] que se incorpore a la interfaz de Blender tiene mucho más sentido que un *script* sumamente complejo que haya que importar a cada proyecto.

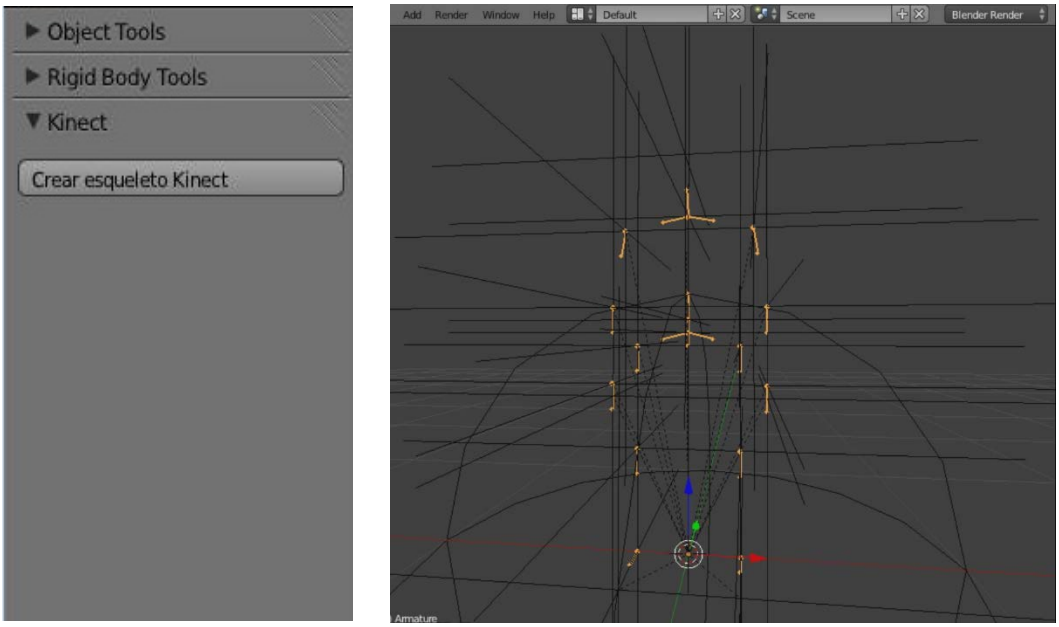
El funcionamiento del receptor y su incorporación al entorno 3D será diferente en la Ventana 3D y en la ejecución del juego, y sus diferencias se detallarán más adelante, pero ambos se basan en lo que se ha denominado “Esqueleto Kinect”: un objeto que se introduce en el entorno que representa esquemáticamente el esqueleto del usuario y que reproduzca sus movimientos. La creación de este objeto es la primera función que debe realizar el *addon*, y lo hace introduciendo en la barra lateral de herramientas de la interfaz el botón “Crear esqueleto Kinect”.

Los elementos básicos del esqueleto Kinect son veinte objetos *empty* que representan a las articulaciones. Los *empties* en Kinect son objetos que se pueden colocar en el entorno 3D y a los que se les puede dotar de las mismas funciones y lógica que a objetos “físicos” como cubos, cilindros o esferas. La diferencia es que los *empties* no aparecen en la imagen final producida por Blender, simplemente señalan un punto en el espacio. Se puede elegir entre varias representaciones gráficas, pero aquí se ha optado por tres ejes que se cortan. A cada

empty se le pone el nombre de una de las articulaciones de KinectOSC, y serán estos objetos los que se moverán en cada instante de tiempo con la recepción de XYZ.

Para reflejar más fielmente el movimiento de una persona que esté usando el Kinect, se han introducido los "huesos" que conectan las articulaciones entre sí. Los esqueletos son uno de los objetos básicos que Blender permite crear, y casi siempre son la base del movimiento de los personajes virtuales. Los huesos se representan simplemente como segmentos con un punto de origen, una orientación y una longitud, la cual es irrelevante para esta aplicación. Blender permite asignar a los huesos órdenes o "restricciones" de muchos tipos. El a*ddon* asignará automáticamente a cada uno una restricción de posición, que les obligará a seguir el movimiento de su articulación correspondiente, y una restricción de apuntamiento hacia la siguiente articulación del esqueleto. Por ejemplo, el hueso "Antebrazo.L" (la L lo señala como el izquierdo) copiará la localización de "ElbowLeft" (codo izquierdo) y apuntará hacia "WristLeft" (muñeca izquierda).

Tener veinte objetos *empty* y un esqueleto completo puede dificultar mucho el manejo del conjunto. Por eso se ha incluido otro objeto *empty* del que dependen todos los demás, esta vez en forma de esfera. Así, todos los desplazamientos, rotaciones y reescalados que se le apliquen a éste se les aplicarán también al esqueleto y las articulaciones. Este objeto servirá también como origen de coordenadas para las articulaciones, por lo que representa la posición del Kinect respecto al usuario. Rotándolo y desplazándolo se podría corregir la posición del Kinect en la realidad: si está en una superficie más alta o más baja, su inclinación…



(a) ***(b)***

Fig. 5. a.- *Barra de herramientas de Blender con el* addon *instalado.* ***b.-*** *"Esqueleto Kinect".*

Como ya se ha explicado, el esqueleto debe recibir las coordenadas XYZ de cada articulación en cada instante de tiempo (la tasa de imágenes del Kinect es de 30 imágenes por segundo, y habiendo veinte articulaciones, se reciben 600 paquetes por segundo). A continuación se detallarán los mecanismos de la recepción en los dos entornos posibles de Blender: la Ventana 3D y la ejecución de un juego.

3.2.1. Recepción en Blender Game Engine.

Blender Game Engine es el motor de juegos de Blender, y el entorno principal en el que se utilizará el *addon*. Su funcionamiento se basa en asociar datos a los objetos del entorno 3D: variables, acciones, condiciones y operadores lógicos. Un ejemplo sencillo de lógica de juego en un personaje sería una condición de teclado (tecla W), una condición de colisión (detecta si el personaje está tocando, por ejemplo, el suelo), un operador AND cuya entrada sean las dos condiciones, y una acción de movimiento a la salida del operador. El resultado es que si se pulsa la tecla indicada y además el personaje está tocando el suelo, se moverá hacia delante.

Una de las acciones que se pueden realizar con esta lógica es llamar a funciones de Python que estén incluidas en el proyecto. Teniendo instalado el *addon* se puede acceder a él. Por defecto, el esqueleto Kinect incorpora:

- Una variable de tipo entero, "puertoOSC". El receptor la leerá para para saber en qué puerto debe esperar la llegada de datos.
- Una variable de tipo booleano, "Activo", de valor por defecto *False*. Lleva asociada una condición que hace que cuando su valor cambie a *True* (el momento del cambio es decisión del usuario) llame a una función que inicia la conexión OSC, creando y abriendo un *socket* en el puerto especificado. Esta variable también está presente en el código del *addon* para que se ponga a *False* en caso de error. Si esto ocurre, o si el cambio está programado por el usuario, la misma función se encargará de cerrar la conexión.
- Otra variable booleana, "recibiendo". Cuando Blender llama a la función de escucha del *socket*, ésta hace esperar a todo el programa hasta que recibe algo. Pero si el usuario sale del campo de visión del Kinect, KinectOSC no envía nada, lo que hace que Blender "se cuelgue". Para evitar esto se ha definido un *timeout* (tiempo máximo de espera) de 500 ms. Cualquier valor de este tiempo lo suficientemente grande para evitar la pérdida de paquetes provoca una ralentización considerable en la ejecución de la aplicación cuando no llega ningún dato, y por eso se ha implementado la variable "recibiendo". Ésta toma el valor *True* cuando empieza la recepción, y cambia a *False* cuando una escucha alcanza su tiempo de espera sin recibir nada. Que "recibiendo" esté en *True* es condición necesaria para que el receptor empiece a escuchar, por lo que al no producirse escucha el programa continúa ejecutándose a velocidad normal. El usuario que programa la aplicación debe decidir cómo manejar estos casos para devolver la variable a *True*, reiniciando así la escucha. Por ejemplo, en las pruebas realizadas, el *timeout* llamaba a una pantalla de pausa en la que se daba tiempo al usuario para colocarse en el campo de visión del Kinect antes de reanudar la recepción.
- Una condición de tipo "Always" que, en cada fotograma del juego (aquí se ha optado por configurar el juego a 30 fps para coincidir con la velocidad del Kinect), realiza una llamada a la función de recepción del *addon*. Ésta inicializa el *socket* si no lo ha hecho ya, recibe los valores de KinectOSC para cada articulación, los opera para obtener las coordenadas XYZ, y según su valor reubica los objetos *empty*.

La obtención de las coordenadas, como se explicó en el apartado anterior, no es inmediata. En la figura 6 se representa esquemáticamente la relación entre los valores entregados por KinectOSC (llamados p_x, p_y y p_z) y las coordenadas reales XYZ. A y B son los ángulos de captación vertical y horizontal de Kinect[10], respectivamente. Sus valores son 43,5º el vertical y 57,5º el horizontal. Con estos ángulos se puede obtener la anchura W y la altura H del cuadro en el que se enmarca la articulación, y que es el que aparece en la pantalla de KinectOSC. Se puede observar que el valor p_z coincide con el valor real de Z en metros.

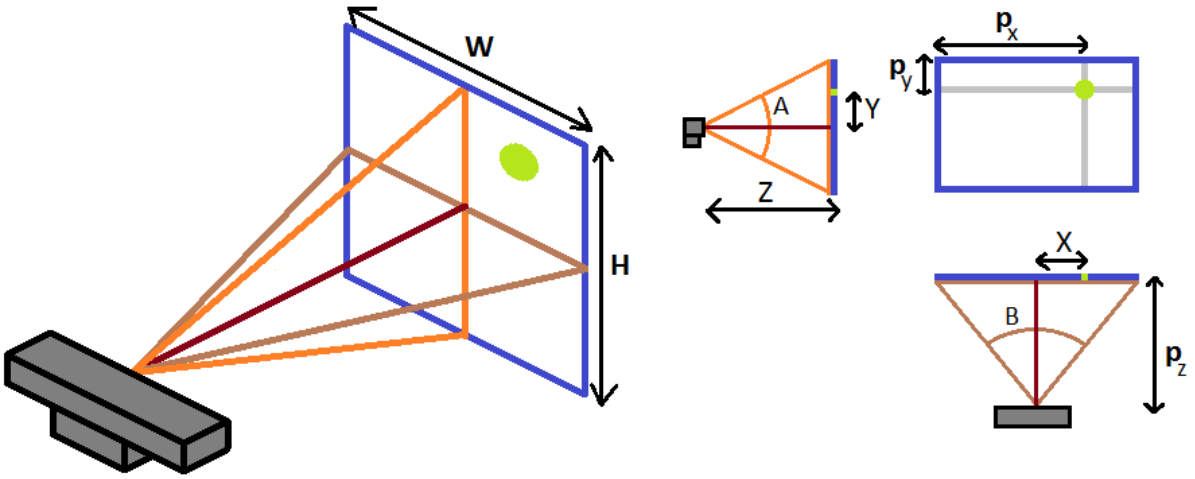


Fig. 6. *Esquema de captación de una articulación por Kinect.*

Sabiendo que, por defecto, los valores de p_x y p_y oscilan entre 0 y 1, y que se considera el origen de X e Y en el punto medio del cuadro (p_x y p_y iguales a 0,5), entonces

$$X = (p_x - 0{,}5) * W. \quad (1)$$

Y, respecto al valor de W, se puede obtener que

$$\tan\left(\frac{57{,}5}{2}\right) = \frac{\frac{W}{2}}{Z}. \quad (2)$$

Despejando W en (2) y sustituyéndola en (1), se obtiene que

$$X = (p_x - 0{,}5) * 2 * Z * \tan(\frac{57{,}5}{2}). \quad (3)$$

La misma fórmula se cumple para Y, sustituyendo el valor del ángulo B por el de A, y teniendo en cuenta que Y se considera positivo para valores de p_y que no lleguen a 0,5:

$$Y = \left(0{,}5 - p_y\right) * 2 * Z * \tan(\frac{43{,}5}{2}). \quad (4)$$

Conviene tener en cuenta que la coordenada X tal y como está obtenida en (3) corresponde a la señal de vídeo del KinectOSC, que está espejada. Para obtener la coordenada real basta con realizar un cambio de signo, o bien con escalar horizontalmente el objeto *empty* principal del conjunto por un factor de -1. En este *addon* se ha optado por realizar el cambio de signo de forma inmediata.

3.2.2. Recepción en la Ventana 3D.

La otra opción para recibir datos que cubre este *addon* es hacerlo fuera de la lógica de juego, en la ventana de edición de Blender. El objetivo es poder almacenar una secuencia de movimientos en la memoria del programa, como una animación a la que se pueda acceder cuando se necesite. Esta técnica se conoce como *motion capture* y facilita enormemente el proceso de animación de un juego o un vídeo. La base de la animación en Blender (y en casi todos los softwares de este tipo) es el uso de *keyframes*: fotogramas clave de la línea de tiempo del programa en los que se define el estado de un parámetro de un objeto, como su posición. Si dos *keyframes* no son consecutivos, entre uno y otro el programa interpolará el valor correspondiente para cada fotograma.

Así pues, el objetivo lógico es insertar en cada fotograma de la animación un *keyframe* que marque la posición de cada articulación. En el *addon* se ha insertado una función que se ocupa de ello, a la cual se accede desde un botón en la barra de herramientas. El funcionamiento del receptor es análogo al del Game Engine, con las diferencias de que el *socket* tiene un uso puntual y que no hay que preocuparse por la ralentización del juego en caso de no recibir paquetes. El programa crea el s*ocket*, y manualmente va haciendo avanzar la línea de tiempo hasta llegar a su final o hasta que el usuario pulse el botón de "Detener". En cada fotograma, se invoca la función de recepción, se obtienen las coordenadas XYZ a partir de las ecuaciones (3) y (4), se recolocan las articulaciones y se inserta el fotograma clave. Al acabar se dispone de una animación que se puede almacenar y modificar a gusto del usuario. Por ejemplo, eliminando los *keyframes* que no sean necesarios para definir un movimiento, aliviando así la carga computacional de Blender.

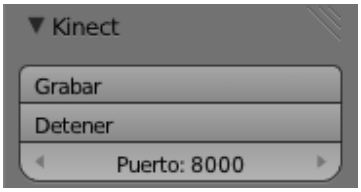


Fig. 7. *Barra de herramientas con el Esqueleto Kinect en la Ventana 3D.*

4. Resultados

A lo largo de todo el desarrollo del *addon* se han ido realizando pruebas para evaluar su funcionamiento. A continuación se detallan las más relevantes:

4.1. Interfaz gráfica del receptor.

Una vez que se hubo establecido la estructura de los mensajes OSC y el código para extraer las coordenadas, se hizo necesario comprobar que los números que el receptor extraía se correspondían con la posición real del usuario. Para ello, implementando el código receptor en Java, se le añadió una interfaz gráfica que mostraba la posición de las articulaciones sobre un lienzo utilizando los valores de p_x, p_y y p_z, de forma análoga a la del KinectOSC. Esta interfaz reveló que los datos recibidos eran correctos.

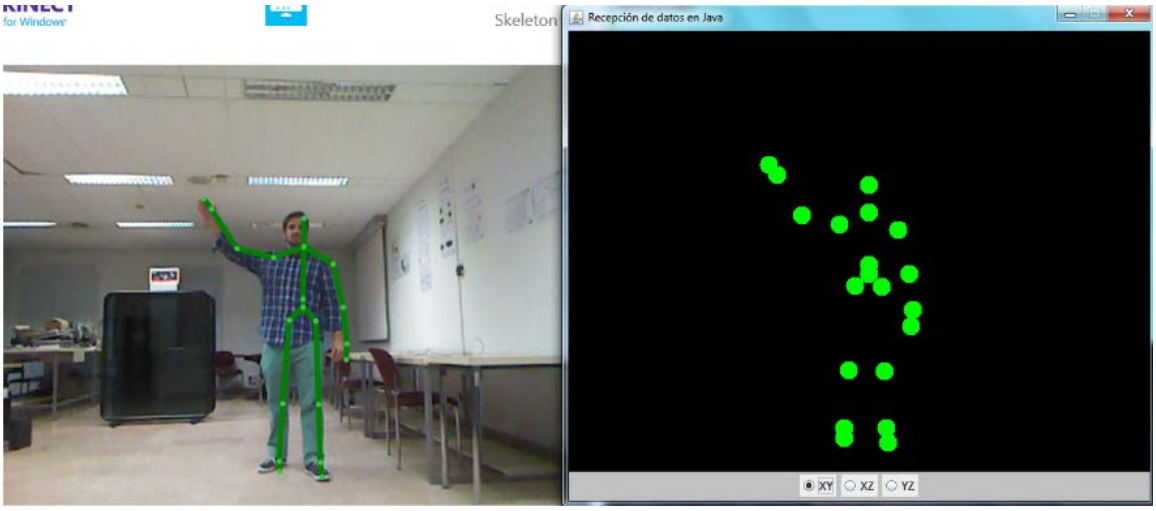


Fig. 8. *Recreación en Java del esqueleto recibido.*

Además de imitar la visión de la cámara del Kinect, que representa los puntos del espacio en un plano XY, se añadió al receptor la opción de dibujar representaciones XZ e YZ, para poder visualizar la profundidad. Fue en esta vista donde se detectó el problema de la visión cónica: no sólo que el esqueleto parecía más pequeño cuanto más lejos estaba del sensor, sino que la posición del usuario también se veía modificada. El mejor ejemplo es el de la Figura 9, donde el usuario está extendiendo los brazos frente a sí y paralelamente al suelo. En el visor de Java, en representación YZ, los brazos no forman la línea horizontal que deberían, porque su posición vertical está marcada por su posición en la señal de vídeo. Esto hace necesaria la conversión de las coordenadas a un sistema isométrico, tal como se detalla en el punto 3.2.1.

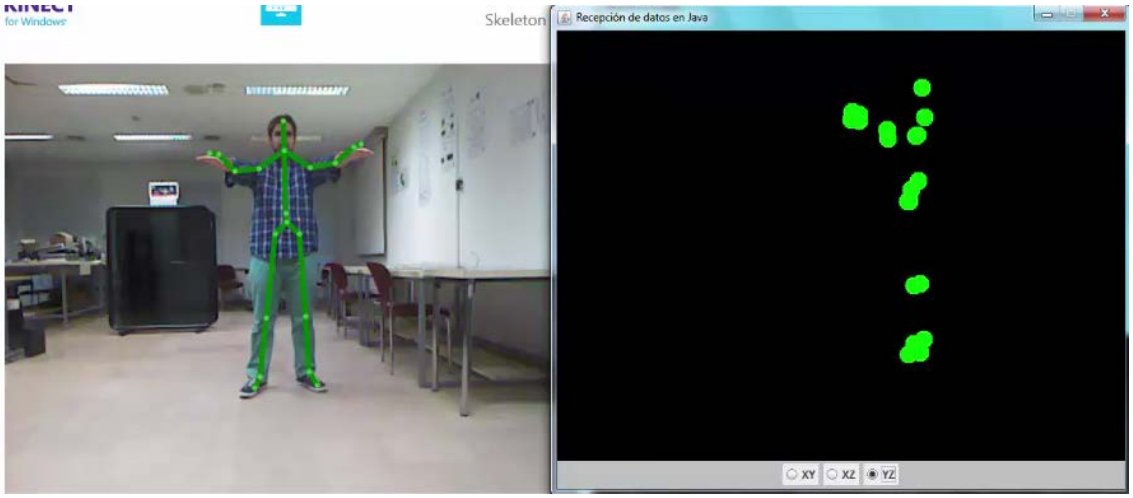


Fig. 9. *Representación del esqueleto en el plano YZ.*

4.2. Pruebas en el receptor de Blender.

Una vez conseguida la recepción e interpretación de los datos en receptores sencillos, tanto en Java como en Python, empezó la tarea de adaptarlo al entorno de Blender. En vez de programar todo el *addon* con todas sus funcionalidades desde el principio, se fueron probando e incorporando sus características una por una. El primer paso fue conseguir imitar en Blender la recepción de los datos.

Desde el principio el desarrollo del *addon* se orientó hacia el funcionamiento de la recepción en el Game Engine. Como ya se ha explicado, una condición "Always" invoca a intervalos regulares a la función de recepción. En esta primera fase, un sencillo *script* mostraba por la ventana de comandos de Blender los datos recibidos. La primera cuestión a resolver fue la frecuencia con la que se debía llamar al receptor. En el código inicial en Java, un bucle infinito realizaba la escucha en el *socket* una vez por iteración. En Blender, la condición "Always" indica el número de fotogramas que se debe esperar antes de realizar la llamada. Esto supone un límite de 60 llamadas por segundo en un juego con velocidad máxima. Ya se ha establecido que el número de recepciones necesarias es de 300 por segundo (30 fps del Kinect por 20 articulaciones), así que hubo que modificar el código: ahora, en cada llamada que se realiza a la función, se efectúan veinte recepciones, una por articulación. Esto limita la velocidad del juego a 30 fps. Además de eso, se ha comprobado que con *timeouts* menores a 3 ms se produce pérdida de paquetes. Puesto que el *timeout* solo va a saltar una vez antes de detener la recepción, se ha optado por asignarle un valor de 500 ms que no deje lugar a dudas sobre la interrupción de la comunicación.

A continuación se procedió a la traducción de los datos recibidos a movimiento de objetos. El primer intento consistió en asignar a un objeto simple (una esfera) un actuador de movimiento, cuya velocidad y dirección se definan por la resta entre las coordenadas en el instante actual (obtenidas tras aplicar las ecuaciones oportunas) y las coordenadas en el instante anterior. Este sistema tiene el peligro de que, si el valor de la velocidad no se actualiza en un fotograma, porque se haya interrumpido la conexión o por alguna otra razón, el objeto sigue con la misma trayectoria que tuviera en el instante anterior. Este error se evidenciaba al insertar una esfera por articulación, ya que hacía que cada una se moviera independientemente y se perdiera la forma del esqueleto humano. En su lugar se optó por modificar la posición local del objeto, es decir, su posición en relación al objeto del que depende, en este caso el que representa al Kinect. Esto permite realizar las correcciones sobre la posición e inclinación del Kinect mencionadas en el punto 3.2, además de suprimir la necesidad de guardar la posición del objeto en el instante anterior.

Con todo esto solucionado, y el *addon* ya programado, se puede comprobar su funcionamiento. En la Figura 10 se muestra el ejemplo más sencillo: los objetos *empty* y los huesos del esqueleto no se ven en la ejecución del Game Engine, así que se les asigna una esfera a cada articulación. Esto ofrece una referencia visual para juzgar si la pose del esqueleto se corresponde con la del usuario. En este caso, funciona pero con una salvedad: a priori, la postura del esqueleto no parece reflejar exactamente la del usuario, sino estar más encorvada. Es necesario estudiar si esto se debe a una mala interpretación de los datos de KinectOSC, a alguna particularidad en la asignación de articulaciones o simplemente a algún hábito postural no aparente en el usuario.

Fig. 10. *Ejecución en Game Engine del ejemplo básico de esqueleto*

4.3. Ejemplo con físicas.

Una vez desarrollado el código del *addon*, se creyó conveniente explorar sus posibles implementaciones en una aplicación real. En una primera aproximación, se intentó dar con un modelo funcional de interacción entre un avatar del usuario y el resto del entorno de Blender: suelo, paredes, objetos, etc.

El primer paso fue crear un modelo de personaje controlado por movimientos. Se optó por un avatar construido con cubos y prismas, uno por cada hueso registrado por Kinect. Hay que tener en cuenta que cada usuario tendrá una complexión distinta y que rara vez coincidirá con la del personaje, que en algunos casos puede llegar a ser mucho más grande o pequeño que una persona real, o tener una anatomía completamente diferente. Por eso el método más adecuado consiste en dar al personaje un esqueleto propio, e imitar la pose del Esqueleto Kinect mediante restricciones de "Copiar Rotación" entre huesos equivalentes.

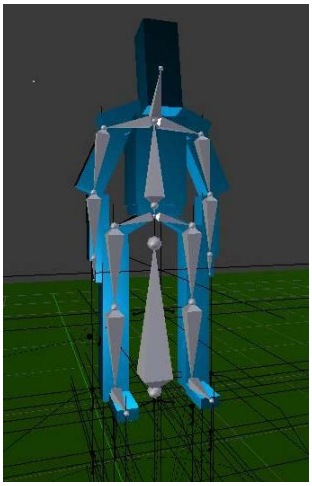


Fig. 11. *Personaje simple.*

Trasladar el movimiento global es una tarea más compleja. La opción más inmediata es dar al personaje una restricción de "Copiar Posición" relativa a alguna articulación del usuario, pero esto presenta una complicación: esta restricción se cumplirá siempre, y no tendrá en cuenta la colisión del personaje con ningún otro objeto del entorno 3D. Esto supone que el personaje atravesará paredes, flotará en el aire o se hundirá en el suelo, según la posición del usuario y del Kinect.

Para conseguir que el avatar interactúe con su entorno, lo más importante es definir su tipo físico. Blender tiene varios posibles, como "Dinámico", "Rígido" o "Personaje", entre otros. Cada uno tiene sus particularidades, y para este ejemplo cualquiera de los tres, que representan a grandes rasgos un objeto sólido. El personaje está compuesto por varias capas, como son el esqueleto y la malla (los cubos, en este caso). Si se definen varias capas como sólidas, colisionarán entre sí y el personaje empezará a moverse descontroladamente. Una de las soluciones más populares es añadir al personaje un "envoltorio" invisible pero sólido que gobierne su interacción con el resto de objetos sólidos del entorno. Éste elemento, por ejemplo un cilindro de alto y ancho similares al personaje, será el que se defina como tipo físico, dejando el esqueleto y la malla intangibles (tipo "Sin colisiones").

Para mover al personaje se ha decidido descartar la restricción de posición y recuperar el uso de un actuador de movimiento, esta vez global para todo el personaje, cuya velocidad esté definida por el desplazamiento de una articulación en concreto. Estos actuadores permiten que el personaje se ve afectado por las físicas de Blender, por lo que no podrá atravesar objetos sólidos y sí le afectará la fuerza de la gravedad. Para elegir la articulación que lo gobierna se barajaron dos opciones: "Spine", el torso, o "FootRight", el pie derecho. "Spine" es la articulación central del esqueleto y la que mejor representa la posición del usuario, pero su uso presenta un inconveniente: si el usuario se agacha, este hueso desciende, pero el personaje (entendido como el cilindro que envuelve la malla) no puede descender más por estar ya apoyado en el suelo. Por lo tanto, el torso del avatar se mantiene fijo mientras la malla adopta la posición de agacharse, dando la impresión de que el personaje esté flotando. El uso del pie derecho evitaría este problema, ya que su posición de reposo ya está al nivel del suelo.

Se produce, en cambio, el efecto contrario: al levantar el pie para caminar, el cilindro y el pie del personaje se mantienen fijos mientras el resto del cuerpo se hunde. Se optó finalmente por utilizar el hueso "Spine", cuya detección por parte de Kinect es mucho más precisa que la de los pies, añadiendo un movimiento secundario a la malla, no al cilindro, para reproducir su movimiento vertical.

Toda esta configuración se puso a prueba en la aplicación de la Figura 12: el personaje aparece sobre una superficie plana con una zanja delante. Además de comprobar que las acciones de saltar y agacharse son reflejadas fielmente, al avanzar el personaje cae a la zanja. Si el usuario camina hacia atrás o adelante, el personaje choca con las paredes y se queda quieto. Si el usuario da un salto real hacia delante, el personaje supera el obstáculo y llega a la segunda zona del suelo.

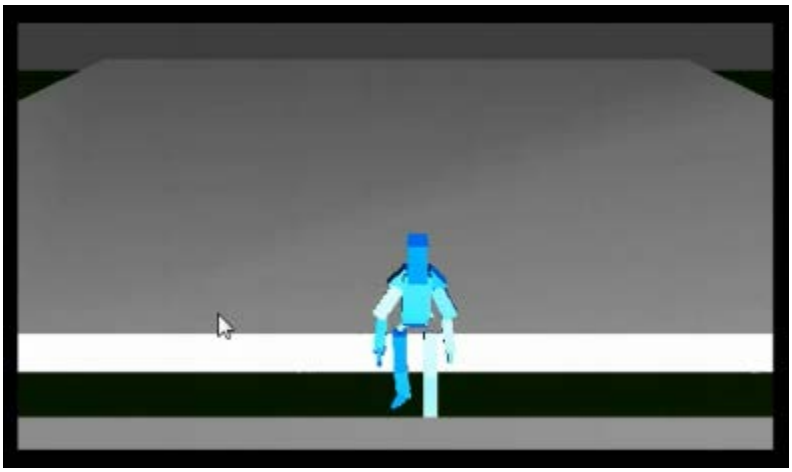


Fig. 12. *Prueba de físicas en Game Engine.*

También se añadió una variante en la cual, en la segunda plataforma, hay una bola que cuelga de una cuerda. El personaje, además, tiene objetos sólidos en las manos con los cuales puede golpear a la bola, probando así las físicas causadas por movimientos del usuario. No se consiguieron los resultados deseados por varios motivos: ralentizaciones de la aplicación, dificultad de acertar a la bola con la mano desde esa vista de cámara, y colisiones de la mano con el cilindro del personaje. El intento se abandonó para explorar una variante en la siguiente aplicación.

4.4. La demo completa.

El final del trabajo se dedicó al desarrollo de una aplicación real que sirviera como demostración de las características del *addon* y permitiera enfrentarlo a problemas típicos. El resultado es un juego corto en el que un personaje debe avanzar por dos escenarios y abrirse paso utilizando diferentes objetos. Combina control por movimientos con animaciones predefinidas, animaciones realizadas usando la función de grabación del *addon*.

Uno de los principales problemas que supone el control de un personaje por captura de movimientos es la movilidad: el usuario solo puede moverse en la pequeña zona en la que el Kinect puede detectarle, mientras que casi cualquier videojuego obliga a moverse por una zona mucho más extensa. Surge así la necesidad de desarrollar un sistema de control que permita mover al personaje libremente. En esta aplicación se ha optado por utilizar el pie derecho como una suerte de ratón: se establece una posición de reposo en relación al cuerpo, si el pie se adelanta, el personaje avanza, si se mueve hacia un lado, el personaje gira hacia ese lado. Se consideraron otras opciones, como implementar el mismo sistema pero con las manos, o bien dotar al personaje de un medio de transporte y hacer que el usuario imite con las manos los controles reales de ese medio (por ejemplo, el manillar de una bicicleta o las riendas de un caballo).

Esto conduce a otro problema: si el personaje imita los movimientos del usuario, cuando el usuario se limite a mover un pie para mover al personaje, parecerá que el personaje se desliza sobre el suelo en esa pose. Afortunadamente, la lógica de juego de Blender permite activar y desactivar las restricciones de los huesos en

el momento oportuno, lo que significa que se puede alternar entre imitar al usuario y ejecutar una animación predefinida. Utilizando este mecanismo, el personaje de la demo ejecuta en todo momento sus animaciones, andar y reposar, excepto por su brazo derecho, que empuña una espada. Los huesos del hombro, brazo y antebrazo derechos tienen siempre puesta la restricción de imitar al usuario, pero se podría anular temporalmente si tuvieran que ejecutar una animación, como recoger algo del suelo.

Fig. 13. *Pantalla de juego de la demo.*

El uso de los objetos conduce a otro aspecto que se quería investigar: la detección de acciones, que Blender reconozca un determinado movimiento como una orden. El personaje empieza la demo con una espada en la mano, y el movimiento del brazo del usuario gobierna el brazo de la espada. Al no tener un botón tradicional de "Ataque", se hace necesario distinguir entre el movimiento normal del brazo y un movimiento rápido capaz de cortar. Un *script* del proyecto registra en todo momento la posición actual de la mano derecha y su posición en el instante anterior, y si la diferencia supera un margen determinado, se considera que el personaje está atacando con la espada. También se pueden activar acciones utilizando la posición: ese mismo *script* compara la posición de la mano con la del hombro derecho, y si ambos se acercan (como haciendo el gesto de envainar la espada a la espalda), se abre el menú de selección de objetos.

El corte de la espada se utiliza para eliminar unos troncos que bloquean parte del camino. Éstos están programados para desaparecer si colisionan con la espada y se cumple la condición de velocidad que se ha mencionado. La condición de colisión requiere que la espada sea sólida, lo cual, como se ha visto, puede dar problemas. Para evitar que un movimiento del brazo haga colisionar la espada con el cilindro que representa la posición del personaje, provocando el mismo efecto que la colisión del esqueleto con la malla, se ha optado por reducir el radio del cilindro hasta el punto en el que el riesgo de que entren en contacto sea mínimo. Aun así, la colisión sigue siendo una posibilidad, y además un cilindro pequeño puede provocar errores y hacer que el personaje atraviese superficies sólidas. Queda pendiente la investigación de otras soluciones posibles para casos como éste.

Antes se ha mencionado la integración de un menú en el juego. Uno de los aspectos que se debe diseñar en una aplicación con Kinect es la navegación por un menú sin el uso del ratón y el teclado. La solución más inmediata es programar un puntero controlado por la mano del usuario, que haga las veces de ratón, pero realizar una selección es menos intuitivo. Como sustituto al *click* de ratón, la solución más extendida en las aplicaciones de Kinect es la siguiente: cuando el puntero pasa sobre una de las opciones, aparece algún tipo de medidor temporal, como una barra de progreso o un anillo que se va cerrando. Si el usuario deja el puntero sobre la

opción el tiempo suficiente para que la barra llegue a su fin, se considera que ha elegido esa opción. Si lo aparta, la barra desaparece y el contador vuelve a su posición inicial.

Otro reto que se asumió en esta demo fue el cambio de escenas, es decir, pasar de un nivel a otro del juego o de una pantalla a otra de una aplicación. El código del *addon* accede a las articulaciones que tiene que mover en base a su nombre. En Blender no puede haber dos objetos con el mismo nombre, así que solo puede haber un Esqueleto Kinect en todo el proyecto. En esta demo se ha optado por enlazarlo entre una y otra, de forma que es el mismo objeto aunque esté presente en dos escenas diferentes. Como el objeto se destruye al final de una escena y se vuelve a crear al empezar la siguiente, se ha optado por crear una escena invisible que se superponga a ambas sin destruirse nunca, que guarde la configuración del juego entre una escena y otra para no tener que repetir todo el proceso. Quizá fuera más sencillo tener una única copia del Esqueleto Kinect en esta escena superpuesta y que no se destruya en el cambio de escena. En este proyecto no se ha elegido esta opción porque, para cuando se añadió el cambio de escenas, la demo ya estaba construida teniendo en cuenta que el esqueleto estaba en la misma escena de juego, y rehacerlo hubiera supuesto demasiado trabajo.

Por último, es necesario señalar que esta demo reveló una carencia de la herramienta de grabación del *addon*, que se utilizó para crear las animaciones predefinidas del personaje (correr, descansar, coger un objeto). El código utiliza un bucle que inserta una clave por cada fotograma de la línea de tiempo, y que no termina hasta que se alcanza su final. Este bucle bloquea el programa y le impide realizar ninguna otra acción que no sea recibir y registrar datos de posición, por lo que el botón de “Detener” es inútil. Esto puede volverse especialmente problemático si se ha configurado una línea de tiempo larga y es necesario interrumpir la grabación por cualquier motivo, ya que no quedará más remedio que esperar a que el programa alcance el final preestablecido.

5. Conclusiones

El resultado final del *addon* cumple satisfactoriamente los objetivos iniciales marcados. Ofrece las herramientas necesarias para que cualquier usuario de Blender incorpore el control por Kinect en su aplicación, o bien para realizar animaciones mediante captura de movimientos. Probablemente su mayor inconveniente es la complejidad de manejo de la recepción de datos, especialmente en el entorno del Game Engine. En cuanto a la herramienta de grabación, sería muy útil encontrar la forma de ejecutarla sin bloquear el resto del programa, de forma que los movimientos que se realicen se puedan reproducir en la Ventana 3D en tiempo real.

Por lo demás, la demo realizada demuestra que este *addon* expande notoriamente las capacidades de Blender, permitiendo incorporar el control por movimientos a cualquier aplicación con relativa sencillez, ya que la demo fue desarrollada desde cero en un plazo menor a dos semanas.

6. Trabajo futuro

Además de refinar los aspectos que se han mencionado en las conclusiones, hay muchas líneas posibles de trabajo a seguir a partir de ahora. La más inmediata es, ahora que el *addon* está prácticamente terminado, utilizarlo en una aplicación real, a poder ser orientada a personas con movilidad reducida, como era el objetivo inicial de la investigación.

Otro objetivo posible es sustituir el KinectOSC por un software emisor de desarrollo propio, más ajustado a las necesidades de Blender. La característica más útil sería que no interrumpiera la emisión de datos al perder de vista al usuario, sino que enviara coordenadas de valor nulo, por ejemplo. Se podrían añadir campos al paquete que permitieran la identificación de más de un usuario simultáneo, o investigar las posibilidades de la detección de dedos.

Además, esta interfaz podría servir como base para establecer comunicaciones entre Blender y otros dispositivos. El mando Wiimote de Nintendo es una posibilidad, pero especialmente interesante es la conexión con dispositivos móviles. Una aplicación en el sistema operativo Android podría permitir el uso de móviles y tabletas como controlador para Blender, enviando los datos recogidos por la pantalla táctil y los acelerómetros. Incluso, si se encontrara la forma de transmitir señal de vídeo en tiempo real, como ya hacen algunas aplicaciones, se podría utilizar el móvil como base para construir un dispositivo de realidad virtual en Blender similar al Project Cardboard de Google[11].

7. Referencias

[1]. HAYTER, Andre. KinectOSC [en línea]. Actualizada: 22 octubre 2013. [Fecha de consulta: 30 de enero de 2015]. Disponible en: https://github.com/ahsquared/KinectOSC

[2]. WRIGHT, Matt. The Open Sound Control 1.0 Specification [en línea]. Actualizada: 26 marzo 2002. [Fecha de consulta: 30 de enero de 2015]. Disponible en: http://opensoundcontrol.org/spec-1_0

[3]. MICROSOFT. Kinect for Windows SDK [en línea]. Actualizada: 22 octubre 2014. [Fecha de consulta: 30 de enero de 2015]. Disponible en: http://www.microsoft.com/en-us/kinectforwindows/develop/downloads-docs.aspx

[4]. DELICODE LTD. NI mate and Blender [en línea]. Actualizada: 15 marzo 2013. [Fecha de consulta: 30 de enero de 2015]. Disponible en: http://www.ni-mate.com/use/blender/

[5]. BIERMANN, Florian, STEENBERGEN, Nikolaas, WALTHER-FRANKS, Benjamin. Bloop [en línea]. Actualizada: 11 de octubre 2012. [Fecha de consulta: 30 de enero de 2015]. Disponible en: http://dm.tzi.de/bloop/

[6]. PYTHON Software Foundation. Pyhton-OSC [en línea]. Actualizada: 26 octubre 2014. [Fecha de consulta: 30 de enero de 2015]. Disponible en: https://pypi.python.org/pypi/python-osc

[7]. FISICOMOLON. Python para Blender Game Engine [en línea]. Actualizada: 6 mayo 2014. [Fecha de consulta: 30 de enero de 2015]. Disponible en: https://www.youtube.com/playlist?list=PLSRQI0r9SD8yjlrHSp6v8HNk2VMkx4IOQ

[8]. BLENDER Foundation. Blender 2.68 API [en línea]. Actualizada: 19 julio 2013. [Fecha de consulta: 30 de enero de 2015]. Disponible en: http://www.blender.org/api/blender_python_api_2_68_release/contents.html

[9]. MILLET, David, TOMBS, Arthur, MONDAY, Louie, BULLER, Jeremy, M'ULE, Fred. Blender 3D: Noob to Pro [en línea]. Wikibooks. Unit 4: Taking Off with Advanced Tutorials. Python Scripting.[Fecha de consulta: 30 de enero de 2015]. Disponible en: http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Advanced_Tutorials/Python_Scripting/Introduction

[10]. MICROSOFT. Human Interface Guidelines v1.8 [en línea]. Introduction. p. 7 [Fecha de consulta: 30 de enero de 2015]. Disponible en: http://go.microsoft.com/fwlink/?LinkID=247735

[11]. GOOGLE. Google Cardboard [en línea] [Fecha de consulta: 30 de enero de 2015]. Disponible en: https://www.google.com/get/cardboard/

Anexos

A. Código del *addon*.

```
import bpy
import math
import socket
import sys

bpy.types.Object.kinect = bpy.props.BoolProperty()
game = False

alfa = math.radians(61.25)
beta = math.radians(68.25)

t1 = math.tan(alfa)
#a = math.pow(t1,2)

t2 = math.tan(beta)
#b = math.pow(t2,2)

bl_info = \
	{
		"name" : "Esqueleto Kinect",
		"version" : (1, 0, 6),
		"location" : "View 3D > Edit Mode > Tool Shelf",
		"description" :
		"Genera un esqueleto Kinect",
		"warning" : "",
		"wiki_url" : "",
		"tracker_url" : "",
		"category" : "Add Mesh",
	}

class PanelKinect(bpy.types.Panel):
	bl_space_type = "VIEW_3D"
	bl_region_type = "TOOLS"
	bl_context = "objectmode"
	bl_category = "Create"
	bl_label = "Kinect"

	def draw(self, context):
		alguno = False
		Columna = self.layout.column(align=True)
		for o in context.scene.objects :
			if o.kinect == True:
				alguno = True
		if alguno == False:
			Columna.operator("mesh.crear_kinect", text="Crear esqueleto Kinect")
			Columna.separator()
		else:
			Columna.operator("anim.grabar_mov", text="Grabar")
			Columna.operator("anim.detener_grabacion", text="Detener")
			Columna.prop(context.scene, "puerto")
			Columna.label(text="Propiedades de juego:")
			Columna.separator()
```

```
                Columna.label(text="- puertoOSC: puerto de escucha.")
                Columna.label(text="  Valor por defecto, 8000.")
                Columna.separator()
                Columna.label(text="- Activo. Controla la recepción.")
                Columna.label(text="  El usuario debe programar que")
                Columna.label(text="  se ponga a True al inicio de la ")
                Columna.label(text="  escena y a False antes de")
                Columna.label(text="  empezar otra.")
                Columna.separator()
                Columna.label(text="- recibiendo. Indica si están ")
                Columna.label(text="  llegando datos al puerto. El ")
                Columna.label(text="  usuario debe programar la ")
                Columna.label(text="  acción del juego en caso de ")
                Columna.label(text="  que cambie a False, por ejemplo ")
                Columna.label(text="  mostrando una pantalla de")
                Columna.label(text="  pausa. Debe devolverse a True ")
                Columna.label(text="  para reanudar.")

class CrearKinect(bpy.types.Operator):
        bl_idname = "mesh.crear_kinect"
        bl_label = "Crear Esqueleto Kinect"

        def invoke(self,context,event):
                huesos =
["HipCenter","Spine","ShoulderCenter","Head","ShoulderLeft","ElbowLeft","WristLeft","HandLeft","ShoulderRight","
ElbowRight","WristRight","HandRight","HipLeft",
“KneeLeft","AnkleLeft","FootLeft","HipRight","KneeRight","AnkleRight","FootRight"]
                posX = [0,0,0,0,0.25,0.3,0.3,0.3,-0.25,-0.3,-0.3,-0.3,0.2,0.2,0.2,0.2,-0.2,-0.2,-0.2,-0.2]
                posY = [0,0,0,-0.1,0,0,0,0,0,0,0,0,0,0,-0.25,0,0,0,-0.25]
                posZ = [0.85,0.9,1.3,1.5,1.25,0.95,0.65,0.6,1.25,0.95,0.65,0.6,0.8,0.4,0,0,0.8,0.4,0,0]
                nombres =
['Torso','Cadera','Cadera.L','Muslo.L','Espinilla.L','Pie.L','Cadera.R','Muslo.R','Espinilla.R','Pie.R','Hombro.L','Brazo.L','A
ntebrazo.L','Mano.L','Hombro.R','Brazo.R','Antebrazo.R','Mano.R','Cabeza']
                fom = [1,1,0,12,13,14,0,16,17,18,2,4,5,6,2,8,9,10,2]
                to = [2,0,12,13,14,15,16,17,18,19,4,5,6,7,8,9,10,11,3]
                #Creamos un nuevo objeto de tipo empty
                bpy.ops.object.add(type='EMPTY')
                ob = bpy.context.object
                ob.name = "Kinect"
                padre = ob.name
                ob.empty_draw_type = "SPHERE"
                ob.kinect = True
                bpy.ops.object.game_property_new(type='INT', name='puertoOSC')
                ob.game.properties['puertoOSC'].value = 8000
                bpy.ops.object.game_property_new(type='BOOL', name='Activo')
                ob.game.properties['Activo'].value = False
                bpy.ops.object.game_property_new(type='BOOL', name='recibiendo')
                ob.game.properties['recibiendo'].value = False
                bpy.ops.logic.sensor_add(type='ALWAYS',name='Siempre',object=padre)
                ob.game.sensors['Siempre'].use_pulse_true_level = True
                ob.game.sensors['Siempre'].frequency = 0
                bpy.ops.logic.sensor_add(type='PROPERTY',name='Socket',object=padre)
                ob.game.sensors['Socket'].evaluation_type = 'PROPCHANGED'
                ob.game.sensors['Socket'].property = 'Activo'
```

```
bpy.ops.logic.controller_add(type='PYTHON',name='Llamar',object=padre)
ob.game.controllers['Llamar'].mode = 'MODULE'
ob.game.controllers['Llamar'].module = "kinectSiete.actualizar"
ob.game.controllers['Llamar'].link(sensor=ob.game.sensors['Siempre'])
bpy.ops.logic.controller_add(type='PYTHON',name='Sock',object=padre)
ob.game.controllers['Sock'].mode = 'MODULE'
ob.game.controllers['Sock'].module = "kinectSiete.socketGE"
ob.game.controllers['Sock'].link(sensor=ob.game.sensors['Socket'])
for i in range(20):
        abc = [posX[i],posY[i],posZ[i]]
        bpy.ops.object.add(type='EMPTY')
        ob = bpy.context.object
        ob.name = huesos[i]
        ob.parent = bpy.context.scene.objects[padre]
        ob.location = abc
objetos = bpy.context.scene.objects
bpy.ops.object.armature_add()
armadura = bpy.context.object
armadura.data.draw_type = 'STICK'
armadura.parent = objetos[padre]
bpy.ops.object.mode_set(mode='EDIT')
armadura.data.edit_bones['Bone'].name = 'Torso'
armadura.data.edit_bones['Torso'].head = (0,0,0.1)
armadura.data.edit_bones['Torso'].tail = (0,0,0)
bone = armadura.data.edit_bones.new(nombres[1])
bone.tail = (0,0,0)
bone.head = (0,0,0.1)
for i in range(17):
        bone = armadura.data.edit_bones.new(nombres[i+2])
        bone.tail = (0,0,0)
        bone.head = (0,0,0.1)
bpy.ops.object.mode_set(mode='POSE')
for i in range(19):
        h = armadura.pose.bones[nombres[i]]
        const = h.constraints.new(type='COPY_LOCATION')
        const.target = objetos[huesos[fom[i]]]
        const = h.constraints.new(type='STRETCH_TO')
        const.target = objetos[huesos[to[i]]]
ob = bpy.context.object
bpy.ops.logic.sensor_add(type='ALWAYS',name='Siempre',object=armadura.name)
bpy.ops.logic.controller_add(type='LOGIC_AND',name='And',object=armadura.name)
bpy.ops.logic.actuator_add(type='ARMATURE',name='Iniciar',object=armadura.name)
ob.game.actuators['Iniciar'].mode = 'RUN'
ob.game.actuators['Iniciar'].link(controller=ob.game.controllers['And'])
ob.game.sensors['Siempre'].link(controller=ob.game.controllers['And'])
bpy.ops.object.mode_set(mode='OBJECT')
bpy.ops.object.add(type='EMPTY')
ob = bpy.context.object
ob.name = "Cubo vacio"
ob.parent = bpy.context.scene.objects['Armature']
ob.parent_type = 'BONE'
ob.parent_bone = "Antebrazo.R"
ob.empty_draw_type = "CUBE"
return {"FINISHED"}
```

```
def socketGrabar():
	puertoS = bpy.context.scene.puerto
	global sock
	abrir = bpy.context.scene.abrir
	dir = "127.0.0.1"
	if (abrir == True)and('sock' not in globals()):
		try :
			sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
			sock.settimeout(0.1)
			print("Socket creado")
		except socket.error :
			print("Error al crear el socket")
			bpy.context.scene.abrir = False
		try :
			sock.bind((dir,puertoS))
			print("Conexión abierta")
		except socket.error :
			print("Error: " + str(sys.exc_info()))
			bpy.context.scene.abrir = False
	elif (abrir == False) and ('sock' in globals()):
		try :
			sock.close()
			print("Conexión terminada")
			del sock
		except socket.error :
			bpy.context.scene.abrir = True
			print("Error: " + str(sys.exc_info()))

def mover(f,nombre,X,Y,Z):
	x = 2*Z*(0.5-X)/t1
	y = 2*Z*(0.5-Y)/t2
	abc = [x,Z,y]
	hueso = bpy.context.scene.objects[nombre]
	hueso.location = abc
	hueso.keyframe_insert(data_path="location",frame=f)

def recibirGrabar(f):
	rec = True
	try :
		i = 0
		while (i < 20) and (rec == True):
			i = i + 1
			[h,x,y,z] = recibir()
			if h != "Ninguno":
				mover(f,h,x,y,z)
				rec = True
			else:
				rec = False
	except :
		print("Iniciando...")

class Grabar(bpy.types.Operator):
	bl_idname = "anim.grabar_mov"
	bl_label = "Grabar animación con Kinect"

	def invoke(self,context,event):
```

```
                fin = context.scene.frame_end
                context.scene.tool_settings.use_keyframe_insert_auto = True
                context.scene.abrir = True
                socketGrabar()
                seguir = True
                f = 0
                bpy.ops.screen.animation_play()
                while (f < fin) and (seguir == True):
                        f = context.scene.frame_current
                        recibirGrabar(f)
                        seguir = context.scene.abrir
                        print("Fotograma = " + str(f))
                        context.scene.frame_current = f + 1
                bpy.ops.screen.animation_cancel()
                context.scene.abrir = False
                socketGrabar()
                context.scene.tool_settings.use_keyframe_insert_auto = False
                return {"FINISHED"}

class Detener(bpy.types.Operator):
        bl_idname = "anim.detener_grabacion"
        bl_label = "Detener la grabación con Kinect"

        def invoke(self,context,event):
                context.scene.abrir = False
                return {"FINISHED"}

def socketGE(controlador):
        dir = "127.0.0.1"
        global sock
        objeto = controlador.owner
        puertoS = objeto['puertoOSC']
        abrir = objeto['Activo']
        if (abrir == True)and('sock' not in globals()):
                try :
                        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                        sock.settimeout(0.5)
                        print("Socket creado")
                except socket.error :
                        print("Error al crear el socket")
                        objeto['Activo'] = False
                try :
                        sock.bind((dir,puertoS))
                        print("Conexión abierta")
                        objeto['recibiendo'] = True
                except socket.error :
                        print("Error: " + str(sys.exc_info()))
                        objeto['Activo'] = False
        elif (abrir == False) and ('sock' in globals()):
                try :
                        sock.close()
                        print("Conexión terminada")
                        del sock
                except socket.error :
                        #objeto['recibiendo'] = True
```

```
                        objeto['Activo'] = True
                        print("Error: " + str(sys.exc_info()))

class OSCTypes():
        """Functions to get OSC types from datagrams and vice versa"""
        class ParseError(Exception):
                """Base exception for when a datagram parsing error occurs."""
        class BuildError(Exception):
                """Base exception for when a datagram building error occurs."""
        def get_string(dgram, start_index):
                """Get a python string from the datagram, starting at pos start_index.
                According to the specifications, a string is:
                "A sequence of non-null ASCII characters followed by a null,
                followed by 0-3 additional null characters to make the total number
                of bits a multiple of 32".
                Args:
                dgram: A datagram packet.
                start_index: An index where the string starts in the datagram.
                Returns:
                A tuple containing the string and the new end index.
                Raises:
                ParseError if the datagram could not be parsed.
                """
                offset = 0
                _STRING_DGRAM_PAD = 4
                try:
                        while dgram[start_index + offset] != 0:
                                offset += 1
                        if offset == 0:
                                raise ParseError(
                                        'OSC string cannot begin with a null byte: %s' % dgram[start_index:])
                        # Align to a byte word.
                        if (offset) % _STRING_DGRAM_PAD == 0:
                                offset += _STRING_DGRAM_PAD
                        else:
                                offset += (-offset % _STRING_DGRAM_PAD)
                        # Python slices do not raise an IndexError past the last index,
                        # do it ourselves.
                        if offset > len(dgram[start_index:]):
                                raise ParseError('Datagram is too short')
                        data_str = dgram[start_index:start_index + offset]
                        return data_str.replace(b'\x00', b'').decode('utf-8'), start_index + offset
                except IndexError as ie:
                        raise ParseError('Could not parse datagram %s' % ie)
                except TypeError as te:
                        raise ParseError('Could not parse datagram %s' % te)
        def get_int(dgram, start_index):
                """Get a 32-bit big-endian two's complement integer from the datagram.
                Args:
                dgram: A datagram packet.
                start_index: An index where the integer starts in the datagram.
                Returns:
                A tuple containing the integer and the new end index.
                Raises:
                ParseError if the datagram could not be parsed.
                """
```

```
                import struct
                _INT_DGRAM_LEN = 4
                try:
                        if len(dgram[start_index:]) < _INT_DGRAM_LEN:
                                raise ParseError('Datagram is too short')
                        return (
                                struct.unpack('>i',
                                                dgram[start_index:start_index +
_INT_DGRAM_LEN])[0],
                                start_index + _INT_DGRAM_LEN)
                except (struct.error, TypeError) as e:
                        raise ParseError('Could not parse datagram %s' % e)
        def get_float(dgram, start_index):
                """Get a 32-bit big-endian IEEE 754 floating point number from the datagram.
                Args:
                dgram: A datagram packet.
                start_index: An index where the float starts in the datagram.
                Returns:
                A tuple containing the float and the new end index.
                Raises:
                ParseError if the datagram could not be parsed.
                """
                import struct
                _FLOAT_DGRAM_LEN = 4
                try:
                        if len(dgram[start_index:]) < _FLOAT_DGRAM_LEN:
                                # Noticed that Reaktor doesn't send the last bunch of \x00 needed to make
                                # the float representation complete in some cases, thus we pad here to
                                # account for that.
                                dgram = dgram + b'\x00' * (_FLOAT_DGRAM_LEN - len(dgram[start_index:]))
                        return (
                                struct.unpack('>f',
                                                dgram[start_index:start_index +
_FLOAT_DGRAM_LEN])[0],
                                start_index + _FLOAT_DGRAM_LEN)
                except (struct.error, TypeError) as e:
                        raise ParseError('Could not parse datagram %s' % e)

def arrancar():
        import bge
        from bge import logic
        global escena
        escena = logic.getCurrentScene()

def recibir() :
        try :
                paquete = sock.recv(1024)
                long0 = len(paquete)
                i = 0
                cond = False
                f = float(0)
                f = [f,f,f]
                while cond == False :
                        (n,off) = OSCTypes.get_int(paquete, 16)
                        (hueso,off) = OSCTypes.get_string(paquete, off)
                        (dato,off) = OSCTypes.get_string(paquete, off)
```

```
                        if dato == ",fff" :
                                (f[0], off) = OSCTypes.get_float(paquete, off)
                                (f[1], off) = OSCTypes.get_float(paquete, off)
                                (f[2], off) = OSCTypes.get_float(paquete, off)
                                cond = True
                                fin = len(hueso)
                        elif dato == ",f" :
                                fin = len(hueso) -2
                                (f[i], off) = OSCTypes.get_float(paquete, off)
                                i += 1
                                if i == 3 :
                                        cond = True
                        elif dato == ",s" :
                                cond = True
                                if hueso != "/noskeleton" :
                                        print("Recibiendo datos tipo String")
                if dato != ",s" :
                        esqueleto = hueso[9]
                        hueso = hueso[11:fin]
                        x = f[0]
                        y = f[1]
                        z = f[2]
        except socket.timeout:
                print("¡Timeout!")
                hueso = "Ninguno"
                x = 0
                y = 0
                z = 0
        return [hueso,x,y,z]

def trasladar(nombre,X,Y,Z):
        x = 2*Z*(0.5-X)/t1
        y = 2*Z*(0.5-Y)/t2
        abc = [x,Z,y]
        hueso = escena.objects[nombre]
        hueso.localPosition = abc

def actualizar(controlador):
        objeto = controlador.owner
        rec = True
        try :
                if objeto['recibiendo'] == True:
                        i = 0
                        while (i < 20) and (rec == True):
                                i = i + 1
                                [h,x,y,z] = recibir()
                                if h != "Ninguno":
                                        trasladar(h,x,y,z)
                                        rec = True
                                else:
                                        rec = False
                                        objeto['recibiendo'] = False
        except :
                arrancar()
                socketGE(controlador)
                print("Iniciando...")
```

```
def register() :
        bpy.utils.register_class(CrearKinect)
        bpy.utils.register_class(Grabar)
        bpy.utils.register_class(Detener)
        bpy.utils.register_class(PanelKinect)
        bpy.types.Scene.puerto = bpy.props.IntProperty \
    (
      name = "Puerto",
      description = "Puerto del socket para la grabación",
      default = 8000
    )
        bpy.types.Scene.abrir = bpy.props.BoolProperty \
    (
      name = "Abierto",
      description = "Indica si se está efectuando una grabación",
      default = False
    )

def unregister() :
        bpy.utils.unregister_class(CrearKinect)
        bpy.utils.unregister_class(PanelKinect)
        bpy.utils.unregister_class(Grabar)
        bpy.utils.unregister_class(Detener)
        del bpy.types.Scene.puerto
        del bpy.types.Scene.abrir

if __name__ == "__main__" :
        register()
```