ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: Solución de juegos “First Person Shooter” (FPS) orientado a personas con movilidad reducida

AUTOR: Gerardo Cilleruelo Beltrán

TITULACIÓN: Grado de Ingeniería de Sonido e Imagen

TUTOR: Martina Eckert

DEPARTAMENTO: Teoría de la señal

VºBº

Miembros del Tribunal Calificador:

PRESIDENTE: Álvaro Alonso González

SECRETARIO: Enrique Rendon Angulo

Fecha de lectura: 24/07/2019

Calificación:

El Secretario,

Agradecimientos

En primer lugar, quiero agradecer a mis padres y hermanos todo el acompañamiento que he tenido el privilegio de disfrutar durante toda mi carrera. En unas ocasiones ha servido como distracción y descanso mental y en otras, de grandísimo apoyo, el cual ha sido crucial para darle riqueza y profundidad a mi formación.

Quiero agradecer a todas aquellas grandes personas, más o menos cercanas, que han tenido la generosidad de compartir sus conocimientos y vivencias en la universidad. Gracias a ellos la soledad no ha tenido cabida durante estos años.

Gracias a Paloma, por saber encontrar destellos en mis penas y participar en mis alegrías.

Por último, no me quiero olvidar de Martina, mi tutora para este proyecto. Su seguimiento y preocupación ha sido ejemplar, le otorgo gran parte del trabajo hecho aquí.

A todos ellos, ¡¡muchísimas gracias!!

Resumen

Hoy en día el gran avance de las tecnologías está consiguiendo una revolución palpable en la vida de las personas. Uno de los campos en el que se aprecia más esta revolución es en la detección de movimiento, en combinación con los videojuegos 3D. Existen muchas aplicaciones a este respecto y entre ellas está la posibilidad de mejorar el día a día de aquellos que tengan algún tipo de movilidad reducida.

Este proyecto pretende conseguir mejorar un juego 3D llamado "*WestGun*", desarrollado previamente en la asignatura SAI (Síntesis y Animación de Imágenes), e implementarlo para que pueda ser jugado, no solo con los controles convencionales de los que dispone un PC, sino también con los propios movimientos del cuerpo humano. La solución técnica de esto se centra principalmente en la posibilidad de que personas con movilidad reducida puedan usar el videojuego de cara a complementarlo para una rehabilitación.

Para llevar a cabo esta solución ha sido necesaria la utilización de un dispositivo detector de movimiento capaz de captar las posiciones de hasta 25 articulaciones del cuerpo humano. El uso y procesado de dichas posiciones sirven para realizar una traducción a tiempo real de los movimientos del jugador al modelo 3D del juego.

Gracias a un *software* adicional llamado *middleware* es posible la recepción y procesado de la información que provee el dispositivo; a su vez el *middleware* lo envía al juego, cuyo desarrollo se ha hecho en el motor de videojuegos *Unity 3D*.

Mediante unos algoritmos y, usando la información de las posiciones de las articulaciones, es posible controlar hasta cuatro acciones dentro del juego con el propio cuerpo. En todas las soluciones existe en mayor o menor medida un efecto de amplificación de movimiento especialmente pensados para esas personas con movilidad reducida. La amplificación consigue simular y amplificar aquellos movimientos que sean más difíciles de ejecutar por el usuario.

De manera complementaria, se ha hecho una integración del juego con la web para que, de manera *online,* un terapeuta pueda establecer una serie de parámetros de configuración útiles para el jugador y recibir resultados de las partidas con el fin de ver el progreso. La comunicación del videojuego con la web se hace a través del *middleware*.

Abstract

Nowadays the great advance of new technologies is achieving a remarkable revolution in people's life. One of the scopes in which this revolution is mostly appreciated is the motion detection, in combination with 3D video games. There are many applications regarding this scope, and one of them is the possibility of improving the daily life of those who have some type of reduced mobility.

This project aims to improve a 3D game called "WestGun" that was previously developed in the subject SAI (Synthesis and Animation of Images), and to implement, in addition to the conventional controls, also a control with corporal movements. The technical solution is mainly focused on the possibility that people with reduced mobility can use the video game in order to complement their conventional rehabilitation.

To carry out this solution a motion detector device was used, which is able to capture the positions of 25 joints of the human body. The use and processing of these positions enable a translation in real time of the player's movements to the game 3D model.

The reception and processing of the information provided by the device is possible through a middleware which sends it to the game developed in the Unity 3D video game engine.

Through algorithms and, using the information of the joint positions, it is possible to control up to four actions within the game with the body. The solution also contains there is, in a greater or lesser extent, an effect of motion amplification specially designed for people with reduced mobility. The amplification manages to simulate and amplify those movements that are more difficult for the user to perform.

In a complementary way, an integration of the game with a therapeutic web platform has been made such that a therapist can establish the difficulty of the game. In order to see the progress of the player, the therapist can get the results that are sent from the game to the web page after the player finishes playing. The communication between the video game and the web is done by the middleware.

Índice de contenidos

Agradecimientos 1
Resumen 3
Abstract 4
Lista de acrónimos 7
1 Introducción 9
1.1 Objetivos 10
1.2 Especificaciones y/o restricciones del diseño 11
2 Marco tecnológico 13
2.1 Campo de investigación actual 13
2.2 Marco tecnológico para el desarrollo de la solución 14
2.2.1 Microsoft Kinect V2 15
2.2.2 Middleware 18
2.2.3 Receptor de Unity 3D 21
2.2.4 Juego "*WestGun*" 24
3 Solución propuesta 27
3.1 Ampliación de WestGun 27
3.2 Ampliación del Middleware 30
3.3 Ampliación del receptor de Unity 3D (KinectAsset) 31
3.4 Solución para el movimiento del brazo 33
3.4.1 Control de la posición del cursor del PC 33
3.4.2 Diagrama de flujo general de la solución 34
3.4.3 Posición del jugador y calibración para obtención de parámetros 34
3.4.4 Envío, recepción y generación de nuevos parámetros de altura 37
3.4.5 Transformación del espacio 2D 39
3.4.6 Control del movimiento del cursor/brazo 3D 42
3.5 Solución disparo, recarga y agachado 44
3.5.1 Solución del disparo del arma 45
3.5.2 Solución de la recarga del arma 47
3.5.3 Solución del agachado del jugador 51
3.6 Ampliación del modo de juego 55

3.6.1 Integración con la web (parámetros de configuración online) 55
3.6.2 Menú de configuración (parámetros de configuración local) 57
3.6.3 Fichero de configuración "*PlayerSettings.json*" 58
4 Resultados obtenidos 61
5 Conclusiones 65
6 Futuras líneas de trabajo 67
Referencias 69
ANEXO I: Documento final del proyecto *WestGun* 71
ANEXO II: Instalación del juego *WestGun-Therapy* 87
ANEXO III: K2UM 2.0 – Daniel Iglesias, abril 2019 89

Lista de acrónimos

CITSEM: Centro de Investigaciones en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad.

GAMMA: Grupo de Aplicaciones Multimedia y Acústica.

UDP: Protocolo del nivel de transporte basado en el intercambio de datagramas (*User Data Protocol*).

FPS: Juego en primera persona (*First Person Shooter*).

PFG: Proyecto de Fin de Grado.

IR: Infrarrojos (*InfraRed*).

SDK: Kit de Desarrollo de aplicaciones (*Software Development Kit*).

USB: Conexión de recepción y envío de información (*Universal Serial Bus*).

C#: Lenguaje de programación *C Sharp*.

VR: Realidad Virtual (*Virtual Reality*)

GUI: Interfaz gráfica de usuario (*Graphic User Interface*)

SAI: Síntesis y Animación de Imágenes

K2UM: *Kinect to Unity Middleware*

1 Introducción

Bajo el avance importante que se está llevando a cabo en las tecnologías con respecto a la captura de movimientos, el proyecto "*Blexer*" [1] (Blender Exergames), desarrollado en el grupo de investigación GAMMA [2] (Grupo de Aplicaciones Multimedia y Acústica) ubicado en el CITSEM [3] (Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad), ha querido hacer uso de esta tecnología con fines sociales.

Principalmente el proyecto pretende fusionar la creación de videojuegos con captura de movimientos y así poder jugarlos con el propio cuerpo. Uno de los objetivos consiste en adaptar estos juegos a personas con movilidad reducida y desarrollar soluciones para que su limitación funcional no sea un impedimento a la hora de jugar. El fin social consiste en implementar en cada juego ejercicios orientados a su rehabilitación de manera que estos puedan ser configurables y organizados por niveles o dificultades.

Hasta ahora se ha conseguido un avance importante en muchos de los objetivos; la fusión anteriormente mencionada ya ha sido llevada a cabo y se han desarrollado y probado varios juegos. Incluso, para más desarrollo, estos avances se han conseguido ubicar en un entorno que permite el control y evaluación de los pacientes/usuarios a través de una plataforma web. El entorno engloba un conjunto de herramientas y soluciones técnicas a las que se le ha llamado "*Blexer*", de tal manera que consigue una combinación de comunicaciones entre la web, los juegos y un *middleware* que transmite los datos de movimiento del jugador. Todo ello se explica con más profundidad en capítulos posteriores.

Aún existe la necesidad de crear más entornos virtuales con nuevas y amplias mejoras de dinámicas de juego y se intenta, con el ingenio y creatividad, implementar nuevos entretenimientos y ejercicios de gran utilidad que puedan realmente llegar a mejorar la calidad de muchas personas. Es aquí donde encaja este PFG, que pretende abrir una nueva puerta a los juegos FPS (*First Person Shooter*).

Los juegos FPS son un género de videojuegos y subgénero de los videojuegos de disparos en los que el jugador observa el mundo desde la perspectiva del personaje protagonista [4]. Habiéndose desarrollado con anterioridad un juego de estas características se quiere conseguir una implementación del mismo que consiga orientarlos a personas con movilidad reducida. Esto permite descubrir nuevos ejercicios y dinámicas que puedan ser usados por este tipo de usuarios y ver hasta qué punto pueden ser de ayuda.

1.1 Objetivos

Como objetivos principales de este PFG están:

1. Adaptar la tecnología Kinect [5], encargada esta de la detección de movimiento, con un juego FPS anteriormente mencionado llamado "*WestGun*" [6]. El juego ha sido desarrollado en la asignatura SAI, cuyos creadores han sido Andrés Salom Velásquez, Nicolás Guillén Echegaray y el autor del presente proyecto.
2. Implementarlo para hacerlo funcionar con personas con diferentes grados de discapacidad.
3. Integrarlo todo con la web para que un terapeuta pueda, de manera *online*, establecer una configuración previa de dificultad y así monitorizar los resultados del jugador.

De los objetivos principales presentados anteriormente se pueden sacar otros más específicos. Del primer objetivo se quiere conseguir:

a) La funcionalidad principal de *WestGun* está en el uso y control de un brazo 3D, cuyo objetivo es la eliminación de todos los enemigos que aparezcan durante la partida a través del uso de un arma. Es por ello que se pretenden sustituir con dicha tecnología los controles de recarga, disparo y agachado desde el motor de videojuegos *Unity 3D* y movimiento del brazo 3D desde el *software middleware*. Concretamente se quiere sustituir el uso del ratón y teclado para la ejecución de dichos controles.

Del segundo objetivo:

a) Mediante algoritmos internos, se quiere conseguir un efecto de amplificación para que incluso aquellas personas con movilidad reducida sean aptas para jugarlo.
b) Elegir movimientos corporales útiles y fáciles de ejecutar en el que se pueda jugar sentado, de pie y la posibilidad de usar con ambos brazos.
c) Añadir calibraciones que permitan al usuario elegir a gusto los movimientos corporales.

Y del último objetivo:

a) Conseguir comunicación en dos direcciones entre la web y *WestGun* a través del *middleware*, manejando y procesando los parámetros de configuración establecidos por el terapeuta.
b) Implementación de varios niveles de dificultad en *WestGun* para poder ver el progreso del jugador.

1.2 Especificaciones y/o restricciones del diseño

Las especificaciones se centran sobre todo en el uso de las distintas herramientas y tecnologías necesarias para el desarrollo del presente proyecto. Por otro lado, se ha visto que varias de estas aportan una cierta limitación para un resultado final sin errores, convirtiéndose en puntos mejorables para futuros desarrollos.

- Con Microsoft Kinect V2 se captura el movimiento. El dispositivo puede llegar a tener una precisión insuficiente a la hora de captar las coordenadas del cuerpo.
- Para la creación del juego se usa el motor de juegos *Unity 3D.*
- Para la comunicación entre la Kinect y el juego es necesario el uso de un software ya desarrollado llamado *middleware*. Su uso implica la modificación/ampliación del código fuente debido a nuevas necesidades que se encuentran en la solución de este PFG.
- El lenguaje de programación para este proyecto es C# [7] (C Sharp) que provee de muchas posibilidades, además de ser el lenguaje más usado con Unity.
- Es importante tener en cuenta la limitación de movimiento para este juego. Por ello se deben crear algoritmos de amplificación para el usuario y los distintos controles.
- Se debe reducir al máximo la complejidad de las calibraciones de estos controles para disminuir la explicación y ayuda a la hora de jugar.
- Uso de ficheros en formato *json* para recoger datos de configuración establecidas previamente por un terapeuta.

2 Marco tecnológico

En este apartado se quiere dejar reflejado, por un lado, la situación actual con respecto a las tecnologías de detección de movimiento en combinación con los videojuegos y sus aplicaciones y, por otro, las herramientas de las que se ha servido este PFG para implementar la solución.

2.1 Campo de investigación actual

En pleno comienzo del siglo XXI ya se cuenta con una amplia investigación en formas mejores de rehabilitación física para pacientes con movilidad reducida. A día de hoy ya existen varios *softwares* y entornos que hacen uso de juegos para llevar a cabo dichos fines, incluso, para ello, se utilizan tecnologías de realidad virtual- que está en pleno auge- o aplicaciones móviles. El fin principal es mejorar la calidad de vida de muchísimas personas y el objeto la de hacer que en su proceso de rehabilitación haya un buen entretenimiento y motivación. Por tanto, merece la pena mencionar algunas de estas mismas aplicaciones.

Nairobia Research es una empresa dedicada a "mejorar pacientes con problemas neurológicos con afectaciones en las áreas motoras, cognitiva o del lenguaje" [8]. Sus métodos incluyen el uso de videojuego amenos con ejercicios variados para la Neuro-Rehabilitación basados en la cinemática. Hacen uso de sensores y tecnologías de detección de movimiento tales como la *Kinect*, mencionada anteriormente. Consiguen hacer una buena combinación de las matemáticas y la neurología que se ve ampliada gracias a los videojuegos.

Figura 1.- Muestra de un ejercicio y persona jugando [8]

Otra tecnología interesante de mencionar es la tecnología *vinCi.* Esta ha sido desarrollada por el centro biomecánico Imana [9] para dedicarse a la "aplicación de la terapia psicomotriz, que incorpora ejercicios para la prevención, el mantenimiento, el entretenimiento, la estimulación y también la rehabilitación" [10]. Usa para ello también tecnología de captura de movimiento en tiempo real y principalmente tecnología VR para incorporarlos en videojuegos. Los resultados obtenidos de los mismos son almacenados y transmitidos a través de la web para el alcance del terapeuta; de esta manera se puede ver el progreso del paciente.

Figura 2.- Muestra de uso de la tecnología *vinCi* [10]

2.2 Marco tecnológico para el desarrollo de la solución

Dentro de este ámbito de investigación el proyecto "*Blexer*" también se quiere hacer hueco, de tal manera que durante algunos años ha conseguido aglutinar un entorno que usa las principales tecnologías para la implementación de este tipo de terapias. Se podría hacer una clasificación por módulos para el mismo; la figura 3 contiene su esquema, aunque habría que mencionar ciertos avances ya que lo que se muestra es una versión anterior del mismo.

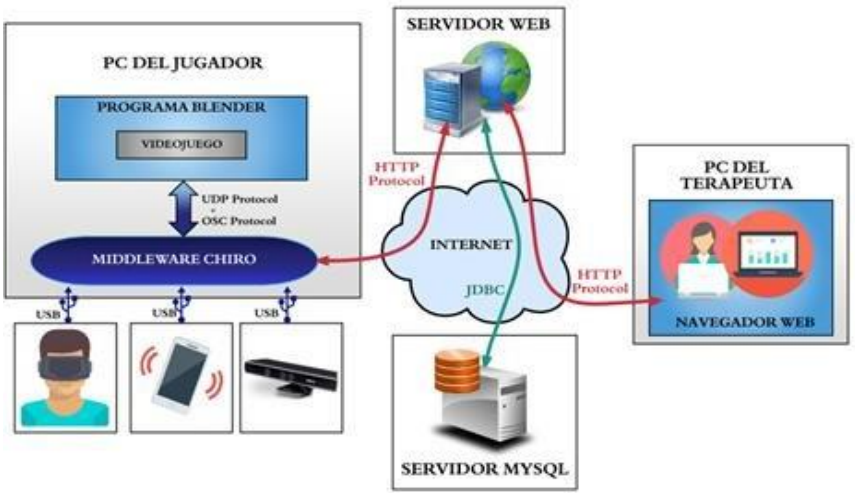


Figura 3.- Esquema del proyecto "*Blexer*" para las primeras fases [13]

Los módulos principales serían, por un lado, el PC del jugador que contiene los videojuegos y el software "*middleware*" que hace las tareas de comunicación entre el juego y el exterior (la web y Kinect). Los dispositivos de detección de movimiento o aquellos que puedan interactuar- añadiendo funcionalidades y modos de jugar- con el videojuego. Por último, el PC del terapeuta, parte fundamental del proyecto ya que sin él no habría asesoramiento en la rehabilitación. En todo ello existe una comunicación en tiempo real entre el jugador y el terapeuta y se debe conseguir que los resultados de los ejercicios puedan ser almacenados para uso posterior. En la solución se apuesta por el volcado de una página web en un servidor que tiene comunicación directa con otro servidor de bases de datos a través de la API de JDBC para persistencia de datos.

Para la solución propuesta han sido requeridas algunas de estas tecnologías, por tanto, a continuación, se desglosa una explicación detallada de las mismas.

2.2.1 Microsoft Kinect V2

Antes se ha hablado de la tecnología Kinect como la encargada de la detección de movimiento para este trabajo. Esta tecnología incluye un hardware llamado Microsoft Kinect V2 desarrollado por Microsoft y creado por Alex Kipman [11]. La V2 es la segunda versión lanzado en el año 2013 del hardware, al que se le adjunta un controlador de código abierto para la recepción de información que pudiese recoger el dispositivo. A su vez, el controlador es capaz de interactuar con los videojuegos compatibles a él (por ejemplo, para los de la Xbox One). Es importante saber que, dado que el controlador es de código abierto, el propio *software* es susceptible de poderse modificar y, por tanto, de crear otro personalizado.

Con objeto de conocer más este hardware a continuación se exponen algunas de sus principales características [12]:

1. Resolución: 1920x1080 para la cámara a color.
2. "*Framerate*": 30 fps.
3. Formato de resolución: 16:9.
4. Sus ángulos de visión son 70 grados en el eje horizontal y 60 grados en el eje vertical.
5. Distancia mínima de uso: 1.37 m.
6. Tiene un emisor de IR Activo.
7. La latencia de transmisión: 20 ms.
8. Puede realizar la detección simultánea de 6 personas.
9. Detecta 25 puntos del cuerpo simultáneamente.

Todas estas características son más que suficientes para lograr el objetivo de este trabajo. El juicio principal a tener en cuenta sobre la aptitud de la Kinect para el proyecto es de la cantidad de tipo de coordenadas que provee. A continuación, se especifican cuáles son:

- Coordenadas de color (*Color Coordinates*): estas vienen en forma de pareja "x" e "y" e indican el ancho y largo de la imagen de una cámara a color incorporada en el dispositivo. Su resolución es de 1920x1080 píxeles en el que el origen de coordenadas se sitúa en la esquina superior izquierda de dicha imagen. La clase del *middleware* guarda esta información en variables de tipo *ColorSpacePoint*.

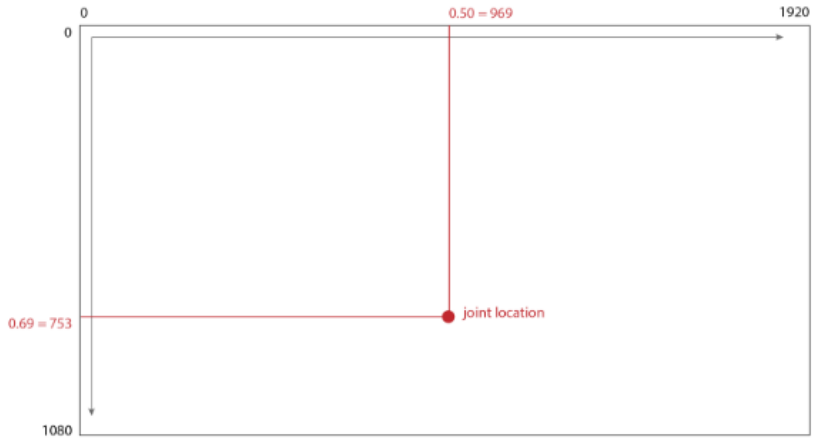


Figura 4.- Sistema de coordenadas de color [28]

- Coordenadas de fondo (*Depth Coordinates*): igual que en las coordenadas de color, pero esta vez provienen de la imagen que se forma con el sensor del dispositivo, el cual tiene una resolución de 512x424 píxeles. La clase del *middleware* guarda esta información en variables de tipo *DepthSpacePoint*.

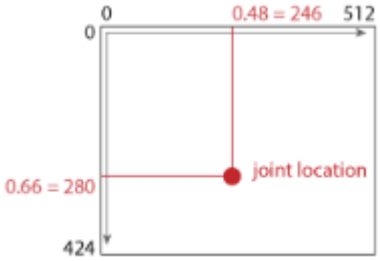


Figura 5.- Sistema de coordenadas de fondo [28]

- Coordenadas de posición (*Position Coordinates*): estas aportan las coordenadas 3D "x", "y" y "z" e indican la desviación horizontal, la desviación vertical y la profundidad respectivamente; todo ello en base a una referencia situada justo en el ojo de la cámara del sensor (figura 6). Todas las coordenadas vienen en metros. La clase del *middleware* guarda esta información en variables de tipo *CameraSpacePoint*.

Figura 6.- Sistema de coordenadas de posición [28]

- Coordenadas de rotación (*Rotation Coordinates*): estas aportan cuatro valores, "x", "y", "z" y "w" los cuales corresponden a las cuatro constantes que tiene un numero llamado "*Quaternion*". Su forma es como la que sigue:

$$q = x + y \times i + z \times j + w \times k \quad (1)$$

Los cuaterniones indican la rotación de las articulaciones del cuerpo en el espacio [20]. La clase del *middleware* guarda esta información en variables de tipo *JointOrientation*.

La *Kinect* recoge las coordenadas de las articulaciones de los cuerpos detectados. En total son 25 puntos identificados por sus nombres tal y como aparecen en la figura 7.

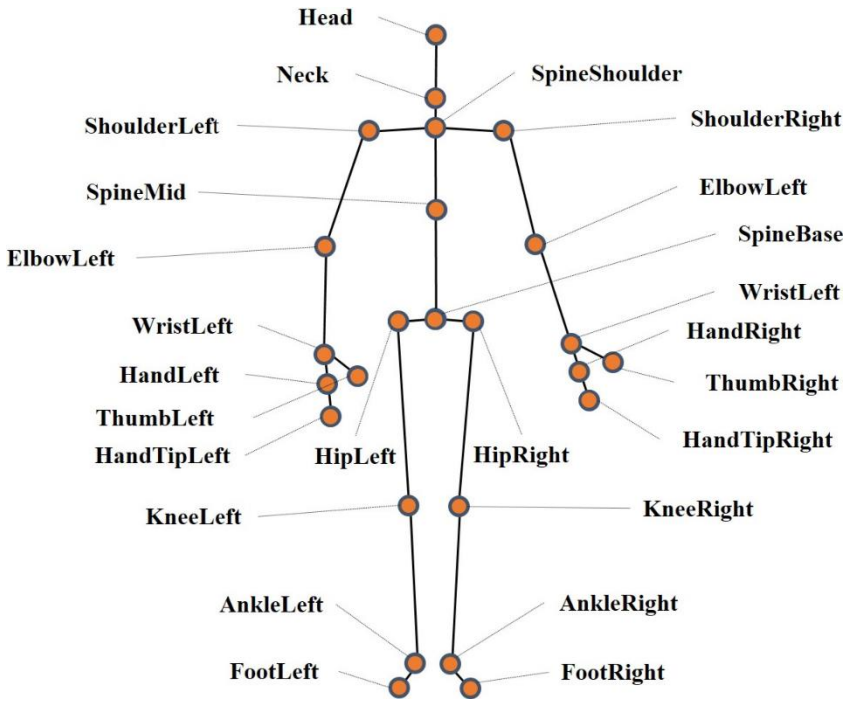


Figura 7.- Articulaciones detectadas por el dispositivo [29]

2.2.2 Middleware

El *middleware* usado en este proyecto, denominado K2UM (*Kinect to Unity Middleware*) es el *software* principal para "la comunicación entre el Hardware Microsoft Kinect V2 y el software de desarrollo Unity 3D" desarrollado por César Luaces Vela [13]. De esta referencia merece la pena extraer el diagrama de bloques del sistema general visualizado en la figura 8.

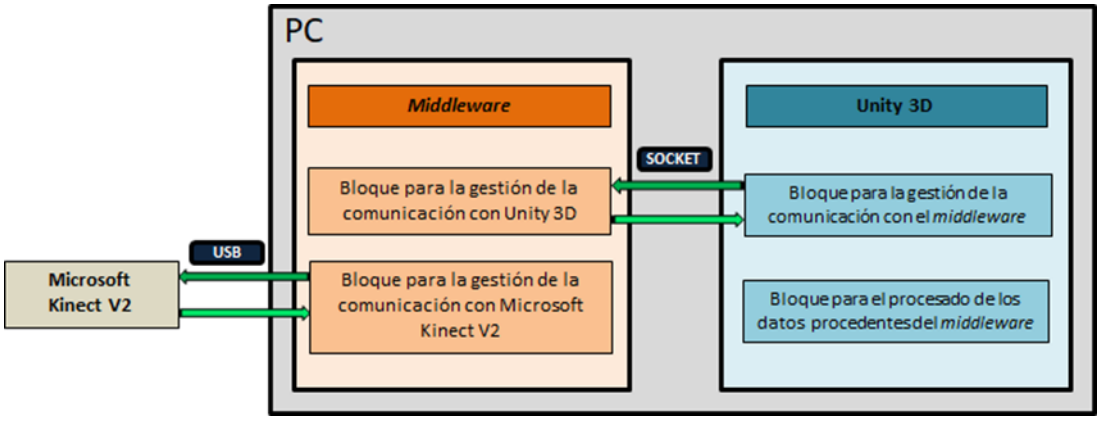


Figura 8.- Diagrama de bloques general del sistema [13]

El *middleware* se ha valido del controlador de la Kinect ya que, como se ha mencionado, parte de su código es abierto y se puede extraer del kit de desarrollo oficial, versión 2.0 de Kinect [14].

La primera versión denominada K2UM 1.0, fue modificada por Daniel Iglesias, para integrar las funcionalidades de comunicación con la web terapéutica, dando lugar a la versión K2UM 2.0 (se adjunta una descripción de sus funcionalidades en el ANEXO III). En esta versión se integraron después las funcionalidades presentes en este trabajo, por tanto, la versión 3.0 (K2UM 3.0) contiene todas funcionalidades de las versiones 1.0 y 2.0.

En la versión K2UM 1.0 reside la gran mayoría del funcionamiento de dicho *middleware* y, como se ha mencionado, gracias a este *software* es posible toda comunicación existente entre la *Kinect V2* y *Unity* 3D. Su implementación ha sido gracias a todas las ventajas que provee el lenguaje de programación C# y un entorno de desarrollo llamado *VisualStudio* [19].

Como resultado final del desarrollo se tiene listo y preparado una aplicación de escritorio en forma de GUI (*Graphic User Interface*) personalizado como la que se muestra en la figura 9.

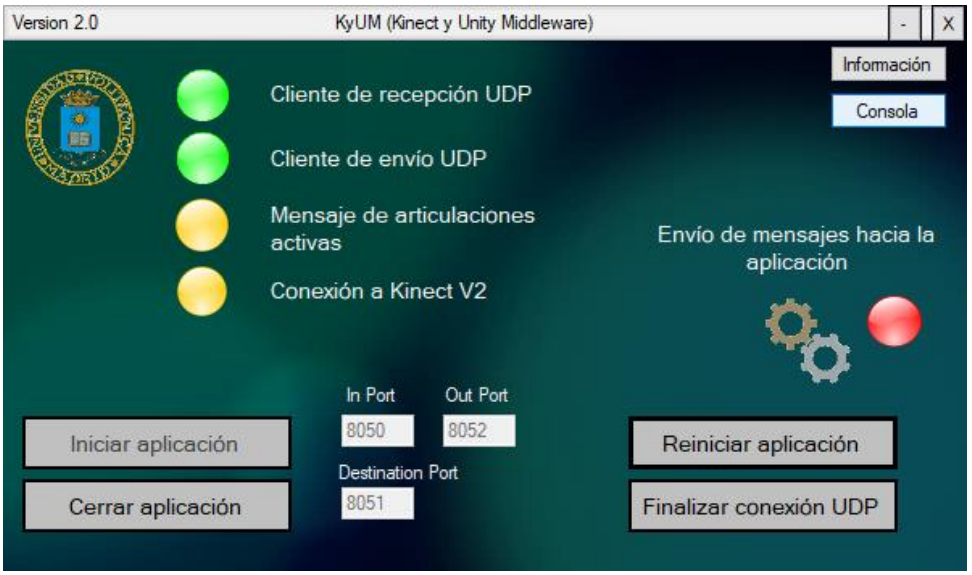


Figura 9.- GUI del *middleware* [13]

Para arrancar la aplicación, como todo programa, existe un fichero ejecutable listo para abrir. Una vez abierta la GUI, (figura 9) solamente se necesita pinchar en el botón "Iniciar aplicación" para que el *middleware* comience a buscar un dispositivo *Kinect* conectado vía USB, si lo estuviera. Para llevar a cabo esto, el código contiene una clase llamada *KinnectMiddleware.cs* que provee de unos métodos que se encargan de ello (todo ello se puede encontrar en la librearía de código abierto). También contiene otros métodos y variables por las que la aplicación comienza a recoger toda la información que el dispositivo manda a través del cable.

Una vez que el *middleware* conecta con la *Kinect*, se mantiene a la espera de que el dispositivo detecte un cuerpo o más y, una vez hecho, comienza a recibir de forma continuada las coordenadas (30 por segundo). Finalmente, estas están listas para ser procesadas por el *middleware*.

2.2.2.1 Procesamiento y envío de coordenadas

Como forma de comunicación entre el *middleware* y *Unity* se ha optado por el uso del protocolo de transporte UDP [21]. Este permite el envío de paquetes a través de puertos dentro del propio ordenador y, al ser un protocolo no orientado a conexión, hace que la transmisión de paquetes sea más rápida.

Las clases *UDPReceive.cs* y *UDPSend.cs* del código del *middleware* son las que se encargan de abrir los canales (hilos/sockets) de comunicación para la recepción y transmisión de paquetes. Por tanto, el procesamiento consiste en la confección de estos colocando de manera ordenada las coordenadas recibidas de la *Kinect* y así prepararlas para su envío.

Los puertos 8052 y 8050 son los puertos usados de salida y entrada respectivamente por los que salen y entran paquetes y, mediante una serie algoritmos y métodos, finalmente se construyen paquetes con la siguiente forma.

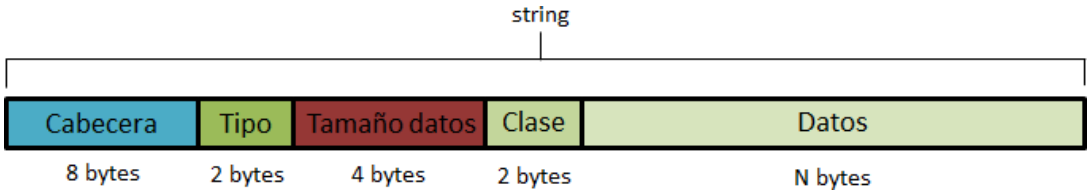


Figura 10.- Formato de paquetes UDP [13]

De la que los paquetes de coordenadas tienen el siguiente aspecto:

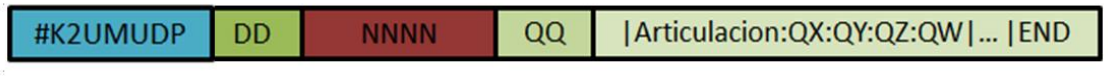


Figura 11.- Formato de paquetes UDP con coordenadas [13]

No solamente están los paquetes UDP que contienen coordenadas; existen otros tipos que llevan mensajes distintos como:

1. Mensajes de error
2. Otros enviados desde *Unity* que se especifican más adelante

Si, desde *Unity* se solicitan las coordenadas, el *middleware* hace comienzo el envío de paquetes; los pasos son:

1. Confección de paquetes
2. Añadir paquetes/*strings* en una cola bloqueante
3. Desencolamiento y envío a través del socket a puerto destino (8051)

2.2.3 Receptor de Unity 3D

Antes de entrar en la explicación del receptor se cree conveniente hablar de *Unity* para ponerse en contexto. Este es un motor multitarea de juegos que ha sido desarrollado por *Unity Technologies* [15] que soporta gráficos en 2D y 3D, funcionalidades de arrastre y adjuntado y uso de C Sharp. Los componentes integrados en Unity, refiriéndose a todo tipo de objetos, personajes, entornos…etc., pueden ser versátiles, aunque es más probable que los desarrolladores quieran ir más lejos de lo que estos componentes son capaces de proveer para implementar características personalizadas. Por ello, la plataforma permite el uso de *scripts* donde el lenguaje de programación C# tiene especial protagonismo. La figura 12 muestra la interfaz gráfica del motor de videojuegos en la que el usuario tiene la posibilidad de colocar las ventanas como quisiera.

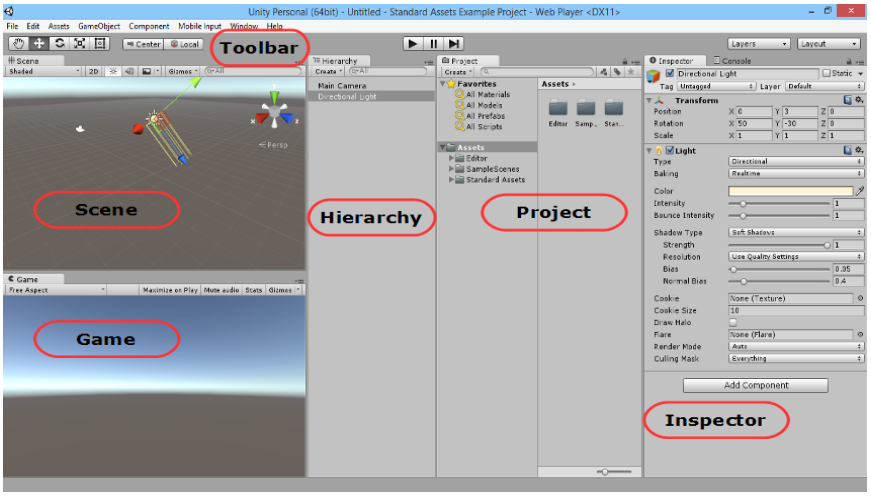


Figura 12.- Interfaz de usuario de Unity 3D

C# es un lenguaje de programación orientado a objetos y componentes desarrollado por Microsoft [16]. Su sintaxis se deriva en parte del lenguaje C y C más más (C++) y usa modelos de objetos similares a *Java*. Este lenguaje permite al usuario activar o desactivar eventos del juego, modificar propiedades de los componentes a tiempo real y responde a cualquier forma de entrada de usuario.

Por otro lado, en *Unity* existe un modelo de componentes llamado "*assets*"; estos son *ítems* que se utilizan en el juego y pueden venir de formas muy diversas, tales

como un modelo 3D, un archivo de audio, una imagen o un código adicional [17]. Una ventaja de ellos es que pueden ser creados fuera de Unity.

Otro tipo de componentes que hay en el motor de videojuegos son los "*prefabs*", que son un tipo de *assets* donde se pueden almacenar objetos (por ejemplo, una figura 3D) prefabricados y de las que se pueden instanciar en cualquier zona de las escenas del juego [18].

El receptor de Unity 3D viene en forma de un *asset* llamado *KinectAsset*, encargado de la recepción de información proveniente del *middleware.* Posibilita la comunicación de este con el juego, por tanto, es un software que debe estar instalado en el mismo. Es una parte fundamental ya que sin el *asset* sería imposible la recepción de coordenadas que el hardware de la tecnología Kinect detecta de los cuerpos. Su contenido viene acompañado de varios *scripts* necesarios para esa recepción y de dos *prefabs*. Esto se enseña en a figura 13.

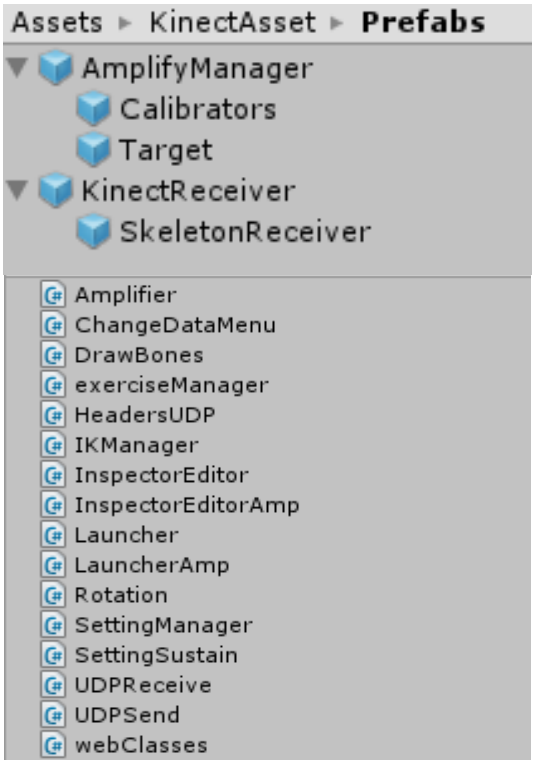


Figura 13.- Contenido del *KinectAsset*

Gran parte del *KinectAsset* y del *middleware* es tenida en cuenta en la solución propuesta, es por ello que hay que dejar claro su funcionamiento.

La función principal del *KinectAsset* es la de gestionar la recepción de datos del *middleware* para su posterior procesamiento y actuación sobre el juego, en este caso, *WestGun*.

Dentro de este *asset* se encuentran todos los *scripts* con los códigos y de estos los principales serían *UDPReceive.cs, UDPSend.cs* y *Rotation.cs*. El primero y segundo tienen

la misma funcionalidad que los del *middleware* y el tercero tiene el cometido de procesamiento y desglose de las coordenadas contenidas en los paquetes UDP.

El *KinectAsset* necesita ubicarse en el proyecto *Unity* del juego y, una vez se arranca el juego, el receptor comienza con la solicitud de recepción de datos al *middleware*; para ello tiene que enviar tres paquetes distintos:

1. Un mensaje de articulaciones activas: en el campo datos se envían los nombres de las articulaciones cuyas coordenadas se desean obtener (figura 14)

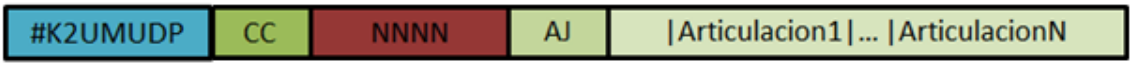


Figura 14.- Formato de un paquete de articulaciones activas [13]

2. Un mensaje con el nombre de usuario
3. Un mensaje de solicitud de datos: este se envía cada vez que llega un paquete de datos (figura 15)

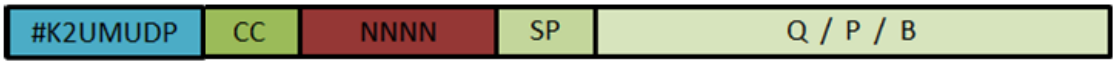


Figura 15.- Formato de un paquete de solicitud de datos [13]

Se distinguen tres opciones que añadir al campo de datos del paquete anterior mostrado:

a) "Q": para solicitar coordenadas de rotación
b) "P": para solicitar coordenadas de posición
c) "B": para solicitar tanto coordenadas de posición como de rotación

Existen más paquetes que transportan otro tipo de mensajes:

1. Mensaje de desconexión de la aplicación
2. Mensaje con resultados del juego

Finalmente, en la figura 16 se muestra la comunicación existente entre las dos aplicaciones. Para este caso se indica la transmisión de datos con coordenadas de posición y de rotación.

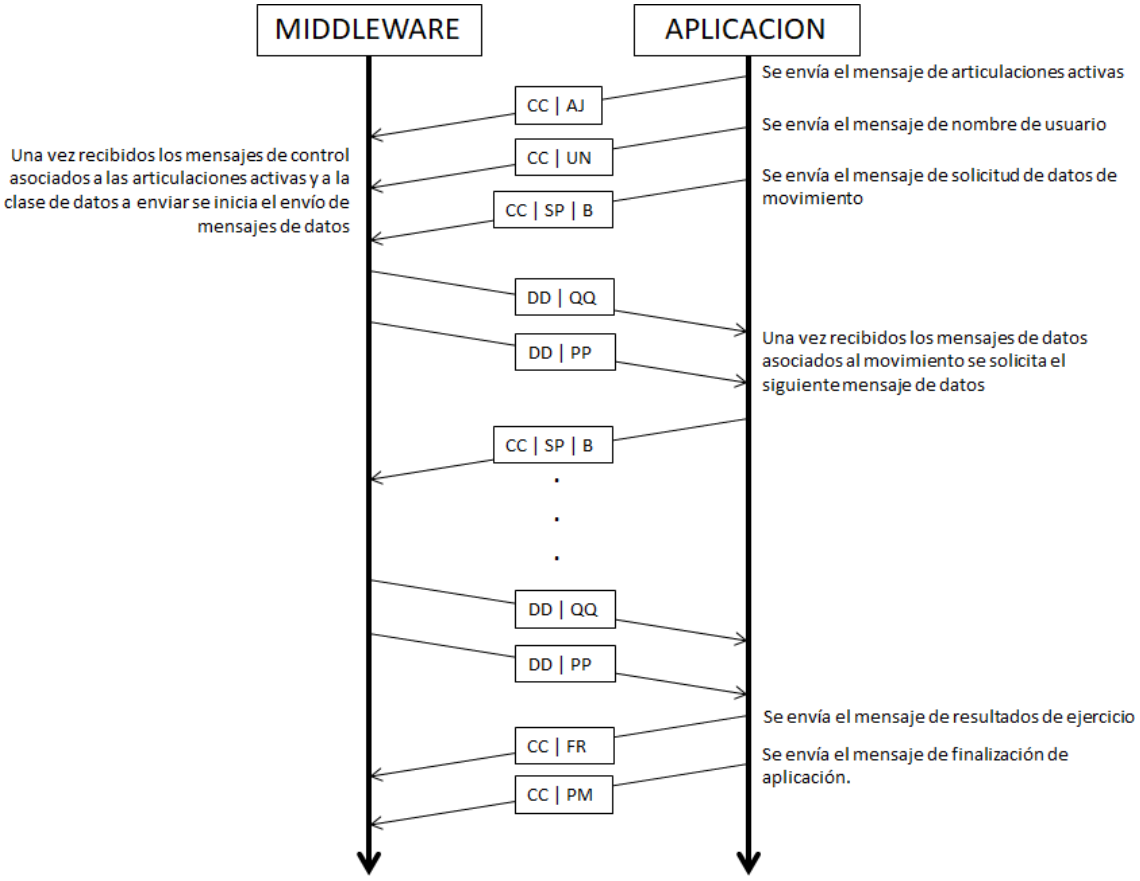


Figura 16.- Comunicación entre *middleware* y receptor de *Unity* [13]

Si se quiere conocer el funcionamiento completo del *middleware* y el *KinectAsset*, puede encontrar la descripción completa en el PFG de Cesar Luaces Vela [13].

2.2.4 Juego "*WestGun*"

WestGun [6] es el juego FPS elegido para este proyecto, como se explica anteriormente, del que ha sido necesaria su ampliación. Es por ello, que para este apartado se explica el estado y funcionamiento del juego antes de modificarlo. Todos los cambios hechos se recogen en apartados posteriores. Por tanto, el nombre del juego no modificado se queda en "*WestGun*" y después de todas las modificaciones se pasa a llamar "*WestGun-Therapy*".

En primer lugar, en la figura 17 se muestra el aspecto que tenían el menú principal y menú de opciones. Como se observa contiene los botones más sencillos, el de "*play*" para dar comienzo al juego, "*options*" para entrar en el menú de opciones en el que solo existía un control de volumen; por último, el botón "*quit*" si se desea salir del juego.

Figura 17.- Menú principal y opciones del juego

La mecánica *WestGun* consta del control de un personaje que se mueve en un ambiente desértico que intenta recordar el Viejo Oeste del siglo XIX. De la figura 18 aparece un brazo apuntando con un arma hacia delante, cuyo objetivo es el de eliminar todos los enemigos que vayan surgiendo. Hay tres tipos de enemigos, uno por cada pantalla: en la primera, el reto consiste en disparar unas latas para ir practicando puntería y en la segunda los enemigos tienen forma de alienígenas que disparan y disminuyen la vida del protagonista. En la última pantalla (la tercera) solo aparece un enemigo, cuya forma es la misma que los de la pantalla anterior, pero de tamaño cuatro veces superior y provoca más daño.

Figura 18.- Primera, segunda y tercera pantalla respectivamente

Con respecto a los controles del juego, el usuario es capaz de realizar dos acciones con el teclado; recargar arma (figura 19) y agacharse (figura 20). Para la primera solo es necesario pulsar tecla “r” y la segunda mantener pulsado la tecla del espaciado.

Figura 19.- Simulación de recarga de arma

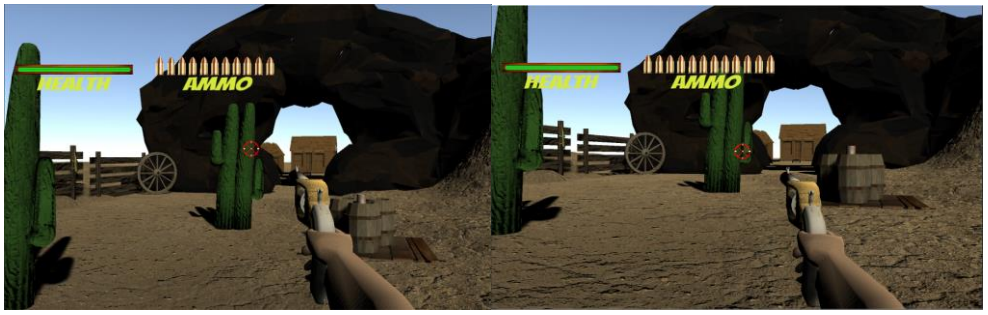


Figura 20.- Simulación de agachado

Otra acción es reservada al ratón, el disparo (figura 21), que se ejecuta pulsando el botón izquierdo del mismo.

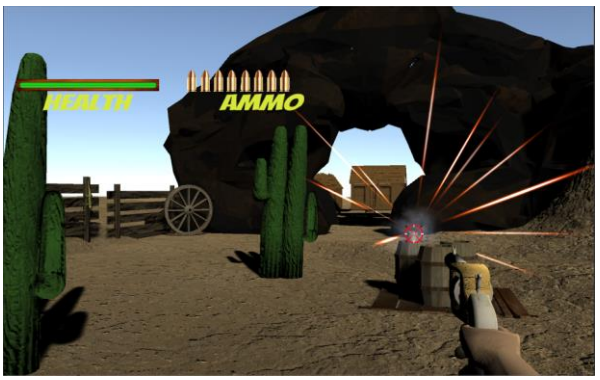


Figura 21.- Disparo de arma

Por último, existe el control del movimiento del brazo, que solo se lleva a cabo moviendo el cursor del ratón.

Como añadidura a la dinámica del juego, se introduce una novedad con respecto al movimiento de la cámara; para esto no se siguen las reglas convencionales de cualquier juego FPS donde la cámara persigue el movimiento del brazo, en sustitución a ello, aquí la imagen permanece estática y solo se mueve/avanza - de manera automática- una vez se hayan eliminado los enemigos de la pantalla. Su propósito es la de crear una sensación de que se dispara a una pantalla desde fuera, como si fuera una mismo el que tiene el arma para disparar. Si a esto último, se añade la posibilidad de poder jugarlo con el cuerpo, la sensación de realidad aumenta.

Finalmente, si se desea conocer más sobre *WestGun* en el primer anexo viene incluido el documento de diseño del juego.

3 Solución propuesta

La solución consta de tres partes, una en la que se explica la ampliación de *WestGun*, *middleware* y *KinectAsset,* otra en la que se describe la solución completa para los controles de agachado, disparo, recarga y brazo de *WestGun-Therapy* y, por último, la explicación de todas las opciones de configuración existentes.

3.1 Ampliación de *WestGun*

Como complementación, se ha visto conveniente acompañar a la descripción de los cambios del juego una comparativa con la versión anterior con deseos de que el lector tenga una visión más clara y global.

El orden de la explicación se procede a desarrollar por escenas:

1. Escena "*Menu*":
    a. *WestGun*: el menú principal tiene las opciones "*PLAY*", "*OPTIONS*" y "QUIT". Dentro del menú de *options* solo existe el control del volumen y la opción de volver atrás (figura 9.2).
    b. *WestGun-Therapy*: los cambios ocurren dentro del menú de opciones, en el que no solo está el control del volumen y retorno, sino también la opción de "*Calibration*" y "*Settings*" (figura 22.2). También se incluye un botón de *Reset,* cuyo funcionamiento se hace en posteriores capítulos. En *calibration* se accede al menú de calibración (figura 22.3) donde se dan las opciones de las distintas calibraciones ya descritas (*arm, reload* y *crouch*). Por último, dentro de la opción de *settings* se encuentra otro menú con la posibilidad de establecer tres parámetros de configuración (brazo con el que jugar, precisión de recarga y modo fácil), cuya explicación se hace más adelante (figura 23).

Figura 22.- Interfaz gráfica principal: menú principal, menú de opciones y menú de calibración respectivamente

Figura 23.- Menú de configuración para establecer tres parámetros

2. Escena "*Scene*":
    a. *WestGun*: esta es la primera pantalla a pasarse en el que se tienen que disparar dos latas. Solo es posible el uso del brazo derecho.
    b. *WestGun-Therapy*: existen hasta cuatro latas (es configurable) y es posible usar también el brazo izquierdo.
3. Escena "*westGun*":
    a. *WestGun*: segunda escena a pasarse donde aparecen 10 enemigos a batir. Uso del brazo derecho.
    b. *WestGun*-Therapy: número de enemigos configurable por niveles. Uso del brazo derecho o izquierdo. Añadir que se han hecho mejoras en los *colliders* de cada enemigo. Un *collider*, el cual es invisible, es un objeto del juego que se asocia a otro para darle características físicas, de manera que pueda colisionar con otros objetos [25]. Los *colliders* de *WestGun* no estaban adaptados a la forma del enemigo- esto es fundamental ya que si hay un fallo de precisión la bala del arma no colisiona con el enemigo y no se le quitaría vida-, sobre todo cuando se agacha; los nuevos *colliders* han sido adaptado de modo que se encojan o estiren según el enemigo se agacha o levanta.
4. Escena "*Final*":
    a. *WestGun*: esta es la última escena a pasarse; en ella aparece un enemigo final que produce más daño que los de la escena anterior. Para matarle se usa el brazo derecho.
    b. *WestGun-Therapy*: aparecen hasta dos enemigos finales (figura 24), según como se configure, y se puede usar el brazo derecho o el izquierdo.

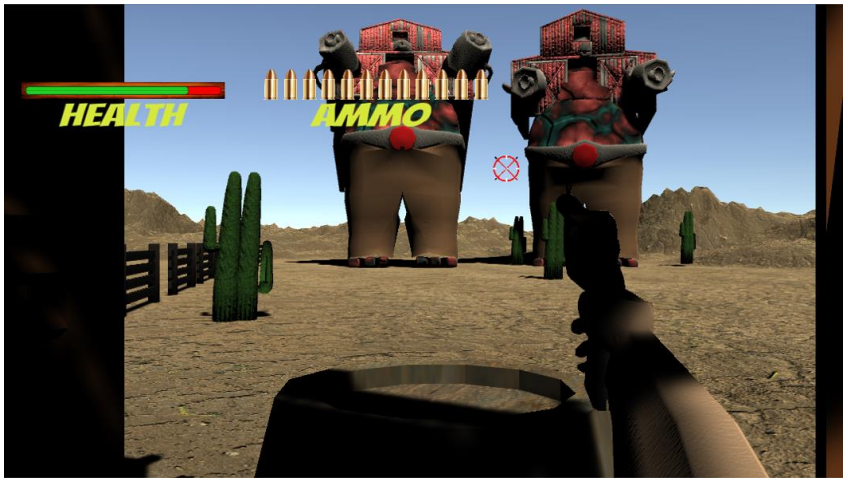


Figura 24.- Escena "*Final*" con dos enemigos

5. Escena "*Calibration*": esta escena solo pertenece a la versión de *WestGun-Therapy* y se ha creado para la solución del movimiento del brazo. A continuación, se añaden dos imágenes (figura 25) que muestran la escena presente, una con el brazo derecho y otra con el izquierdo.

Figura 25.- Escena "*Calibration*" para ambos brazos

Hasta aquí quedan especificados los cambios propios del juego, se sigue con los hechos en el *middleware.*

3.2 Ampliación del *Middleware*

Si se recuerdan los cuatro tipos de coordenadas que se pueden recoger de la *Kinect* (ver punto 2.2.1) resulta que en la versión K2UM 1.0 del *middleware* no existe la posibilidad de procesamiento de las coordenadas de color ni las coordenadas de fondo. En este proyecto se ha requerido el uso de las coordenadas de color para la solución del movimiento del brazo. A pesar de que las coordenadas de fondo no han sido necesarias, se han añadido también por si fueren útiles.

Gracias a un método llamado "*Reader_FrameArrived*" de la clase *KinnectMiddleware.cs* la recepción y construcción de paquetes de datos con coordenadas de color y fondo puede llevarse a cabo. Dichos paquetes se muestran a continuación en las figuras 26 y 27:

Figura 26.- Formato de paquete tipo datos con coordenadas de color

Figura 27.- Formato de paquete tipo datos con coordenadas de fondo

Las coordenadas de rotación están implementadas tanto en el *middleware* como en el receptor de *Unity*, sin embargo, no se ha visto la importancia de su uso y se han omitido para la solución del proyecto.

Por último, bajo el punto de vista de que ahora se manejan cuatro tipos de coordenadas se ha visto muy práctico que hubiera una posibilidad de discriminar coordenadas de una manera más precisa. Para conseguir esto solamente se ha necesitado hacer un pequeño cambio en los paquetes con coordenadas que se envían desde el *middleware*. Se recuerda de que el campo datos para la versión K2UM 1.0 era estructurado de la siguiente forma:

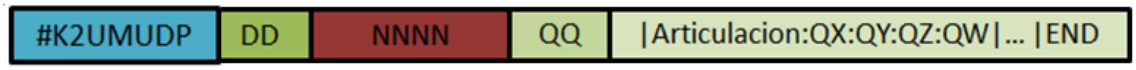


Figura 28.- Formato del campo datos [13]

La figura 28 recoge el ejemplo del formato de un paquete de datos con coordenadas de rotación; como se observa, se añade primero en el campo datos el nombre de la articulación, seguidamente de los valores de las coordenadas, así hasta terminar la lista de todas las articulaciones. En este caso serían cuatro valores

correspondientes a las constantes de un *quaternion.* Ahora, para la versión K2UM 3.0 la estructura cambia a ser:

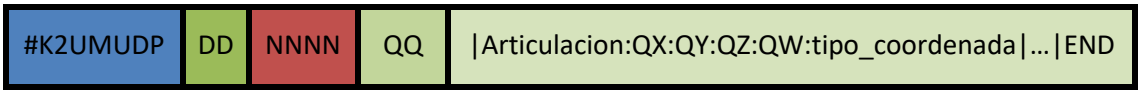


Figura 29.- Ejemplo de formato actualizado del campo datos con coordenadas de color

Como se observa se añade al final de los valores numéricos una cadena específica que indica el tipo de coordenadas que se están recibiendo. Las cadenas son:

1. "position": para posición
2. "colorPoint": para color
3. "depthPoint": para fondo
4. "orientation": para rotación

3.3 Ampliación del receptor de *Unity 3D* (*KinectAsset*)

La ampliación del *KinectAsset* ha sido necesaria debido, principalmente, al manejo de las nuevas coordenadas añadidas al *middleware*. Estas necesitan ser recibidas en el *Asset*, por tanto, debe existir un código adicional que permita dicha recepción.

Otro problema encontrado es que, a pesar de que la versión antigua del *middleware* sí que procesaba las coordenadas de posición, el receptor no tenía las líneas de código necesarias que valoran los campos de un paquete de coordenadas de posición. Este tipo de coordenadas son cruciales para la implementación de las soluciones de recarga, disparo y agachado.

En definitiva, la ampliación del *KinectAsset* pasa por la codificación adicional de la clase *UDPReceive.cs* para conseguir recibir y procesar paquetes con coordenadas de posición, color y fondo. La figura 30 muestra las líneas de código que valoran y detectan el contenido del campo "clase" de los paquetes de datos. Los contenidos posibles son:

1. *"dataFormatP"*: "PP"
2. *"dataFormatS"*: "QQ"
3. *"dataFormatC"*: "CL"
4. *"dataFormatD"*: "DP"

```
if(Encoding.UTF8.GetString(dataFormat).Equals(dataFormatP)){
    //Se convierte de bytes a string
    string text = Encoding.UTF8.GetString(dataInformation);
    //Se encola como mensaje nuevo recibido
    //print(text);
    messageQueue.Enqueue(text);
    //Se modifican las variables de control
    //lastReceivedUDPPacket = text;
    numberReceivedPackets++;
}else if(Encoding.UTF8.GetString(dataFormat).Equals(dataFormatS)){
    //Se convierte de bytes a string
    string text = Encoding.UTF8.GetString(dataInformation);
    //Se encola como mensaje nuevo recibido
    messageQueue.Enqueue(text);
    //Se modifican las variables de control
    //lastReceivedUDPPacket = text;
    numberReceivedPackets++;
}else if(Encoding.UTF8.GetString(dataFormat).Equals(dataFormatC)){
    //Se convierte de bytes a string
    string text = Encoding.UTF8.GetString(dataInformation);
    //Se encola como mensaje nuevo recibido
    messageQueue.Enqueue(text);
    //Se modifican las variables de control
    //lastReceivedUDPPacket = text;
    numberReceivedPackets++;
}else if(Encoding.UTF8.GetString(dataFormat).Equals(dataFormatD)){
    //Se convierte de bytes a string
    string text = Encoding.UTF8.GetString(dataInformation);
    //Se encola como mensaje nuevo recibido
    messageQueue.Enqueue(text);
    //Se modifican las variables de control
    //lastReceivedUDPPacket = text;
    numberReceivedPackets++;
```

Figura 30.- Código adicional para la recepción de paquetes de datos con coordenadas de posición, color y fondo

El uso de las nuevas coordenadas implica que se deben confeccionar otros tipos de paquetes de solicitud de datos acordes a las distintas clases de coordenadas. La figura 31 muestra una comparativa del paquete de solicitud antiguo con el nuevo.

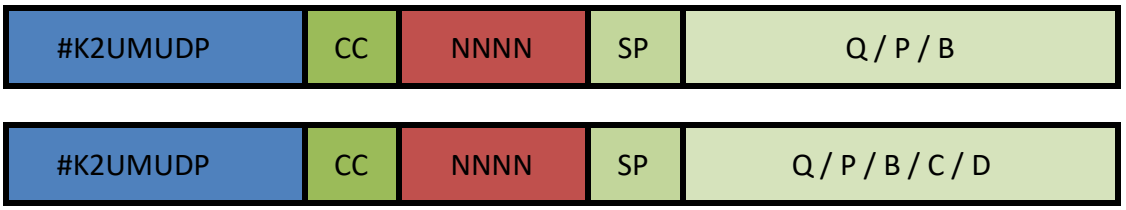


Figura 31.- Formato actualizado de un paquete de solicitud de datos

Donde la opción “C” solicita coordenadas de color y la “D” coordenadas de fondo.

3.4 Solución para el movimiento del brazo

Como se ha mencionado en capítulos anteriores, el brazo del juego es controlado gracias al cursor del ratón, de manera que en la clase *MiraPlayer.cs* se establece que el modelo 3D del brazo apunte hacia dicho cursor en todo momento. Sabiendo esto, la mejor solución encontrada ha sido conseguir que el cursor pueda ser controlado sin necesidad del ratón dentro del código del *middleware*.

Mediante el uso de las coordenadas de color de una mano (depende de la que se elija), captadas por la Kinect, se puede hacer una traducción a tiempo real y trasladar esas coordenadas a las del cursor. Aunque no basta con trasladar directamente las coordenadas captadas; se debe hacer un proceso entre medias para conseguir ajustar las coordenadas a la resolución del ordenador local. Además, hay que tener en cuenta también la amplitud o rango de movimiento que tiene el jugador, el cual, es solventado gracias a una calibración previa.

Como aclaración se eligen concretamente las coordenadas de *HandRight* o *HandLeft* para esta solución.

Para más detalle, lo primero que se quiere explicar, es cómo mover dicho cursor.

3.4.1 Control de la posición del cursor del PC

Gracias a los recursos de que provee el lenguaje de programación C# es posible la utilización de un método/variable que accede al driver del PC y modifica la posición del cursor pasando como parámetros las coordenadas en píxeles. Estas coordenadas deben siempre ajustarse a la resolución local del PC desde donde se hace uso del juego. Por tanto, a la hora de implementar el código también se debe acceder a la información de la resolución mencionada.

El método usado tiene el siguiente aspecto:

```
[DllImport("user32.dll")]
public static extern bool SetCursorPos(int x, int y);
```

Los parámetros siempre deben pasarse como tipos de variables enteras.

En base a esto es importante exponer de forma clara el esquema general del funcionamiento de esta solución.

3.4.2 Diagrama de flujo general de la solución

En la figura 32 se quiere mostrar todo el proceso que existe desde que se prepara el jugador para realizar la calibración hasta que consigue tomar control del cursor con la mano. Todos los pasos vienen enumerados y su explicación se hace con detalle más adelante.

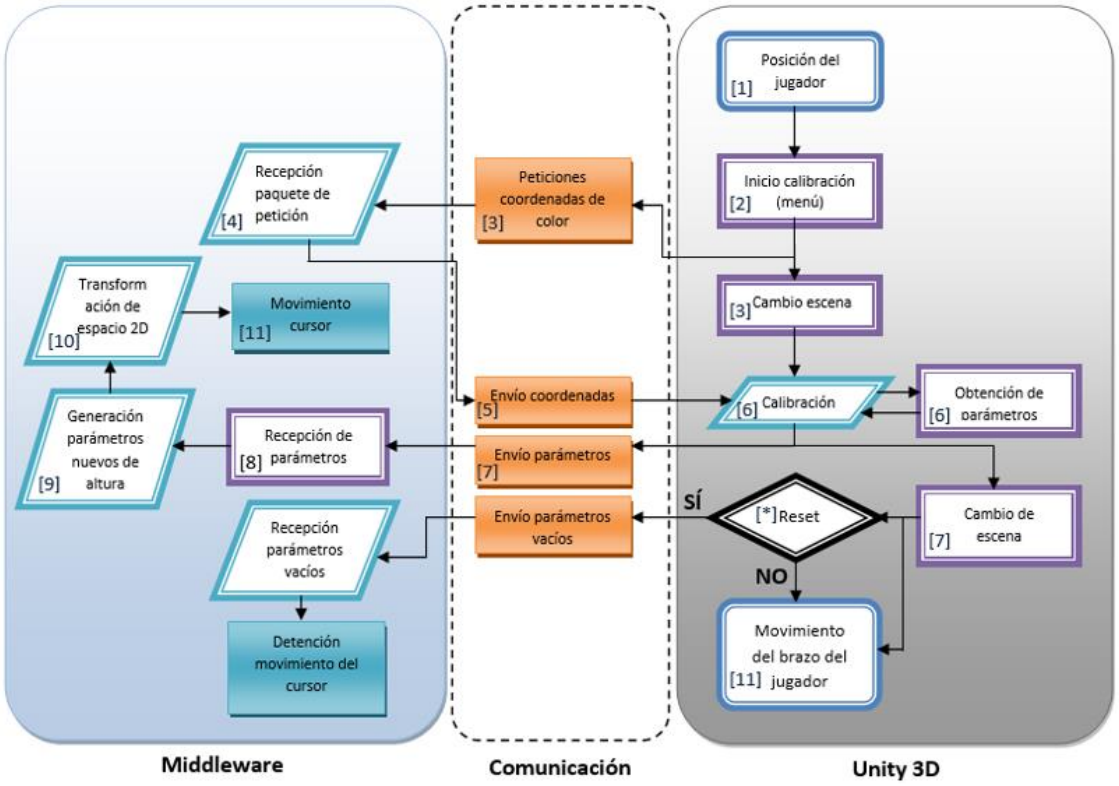


Figura 32.- Diagrama de flujo de la solución para el movimiento del brazo

3.4.3 Posición del jugador y calibración para obtención de parámetros

Para el comienzo de la calibración es necesaria una postura correcta del jugador **(paso 1)**. Este debe posicionarse en frente de la pantalla y la *Kinect* a una distancia mínima de entre dos y tres metros. Se permite estar tanto de pie como sentado con una postura erguida. Para dar comienzo a la calibración se necesita de un asistente que tenga acceso al control de la interfaz del juego, pinchar en el menú de opciones, seguidamente en "calibration" y por último en "arm" **(paso 2)**. Antes de dar a esta última opción, el usuario tiene que estar apuntando con el brazo (izquierdo o derecho, según cuál se haya elegido) al centro de la pantalla del ordenador. Se vuelve a exponer la figura siguiente a modo de recordatorio.

Figura 33.- Interfaz gráfica principal: menú principal, menú de opciones y menú de calibración respectivamente

Una vez pulsado la opción de "*arm*" se cambia a la escena de *Calibration* y, de manera paralela, se envía al *middleware* paquetes de solicitud de datos de coordenadas de color, desde la clase *Rotation.cs* contenido en *KinectAsset* **(paso 3)**. La escena y el paquete de solicitud se ilustran en las figuras 34 y 35.

#K2UMUDP	DD	NNNN	SP	C

Figura 34.- Formato paquete de solicitud de datos de coordenadas de color

Figura 35.- Escena de *Calibration*

El siguiente paso pasa por la recepción del paquete en el *software* del *middleware* **(paso 4)** el cual, reconoce la "C" del campo datos y comienza a enviar las coordenadas de color **(paso 5)** a *Unity*.

Ya en *Unity,* dentro de la escena de calibración **(paso 6)**, se da comienzo a la recepción de las coordenadas de color cuyo objetivo es la de obtener los parámetros. Aclarar que estos parámetros van aparte de los que se pasan a la función *SetCursorPos(x, y)*, anteriormente descrita. Los parámetros que se obtienen de esta calibración son cruciales para conseguir que el juego establezca un rango de movimiento del brazo y pueda detectar en qué zonas el brazo del jugador puede moverse. Estos parámetros

responden a las preguntas de: ¿cuánto desplazamiento necesito hacer del brazo para mover el cursor a lo ancho y alto de la pantalla? ¿Puedo mover el brazo por otras zonas de la habitación o lugar donde esté probándolo? Para aclarar esto se debe comenzar a explicar lo que debe hacer el jugador durante la calibración.

Se recuerda que el usuario ya está apuntando al centro de la pantalla y, durante la calibración, debe agitar el brazo de izquierda a derecha, de manera que es el propio usuario quién va a establecer ese rango de movimiento. Los criterios de movimiento son a gusto del jugador, aunque se recomienda oscilar el brazo poniendo como referencia los bordes laterales de la pantalla. Importante recalcar que no se debe bajar el brazo hasta que la calibración haya terminado; su duración es de quince segundos, tiempo suficiente para elegir bien ese rango de movimiento y, una vez consumido ese tiempo, el juego cambia de escena automáticamente al menú de *calibration* (paso 7).

Para más detalle de los explicado hasta ahora, se tiene en la figura 36.1 una persona dibujada en frente de la pantalla desde donde se está ejecutando el juego. Esta va agitando el brazo de lado a lado formando un trazado y un desplazamiento máximo (D). La *Kinect* capta las coordenadas de *HandRight* o *HandLeft* en píxeles de su imagen a color y, en una clase contenida en *Unity* llamada *Calibration.cs,* se ejecuta un algoritmo que valora las coordenadas máximas y mínimas. Estas se convierten en los parámetros deseados para esta calibración, como muestra la figura 36.2 (*Xmax, Xmin, Y'max y Y'min*)

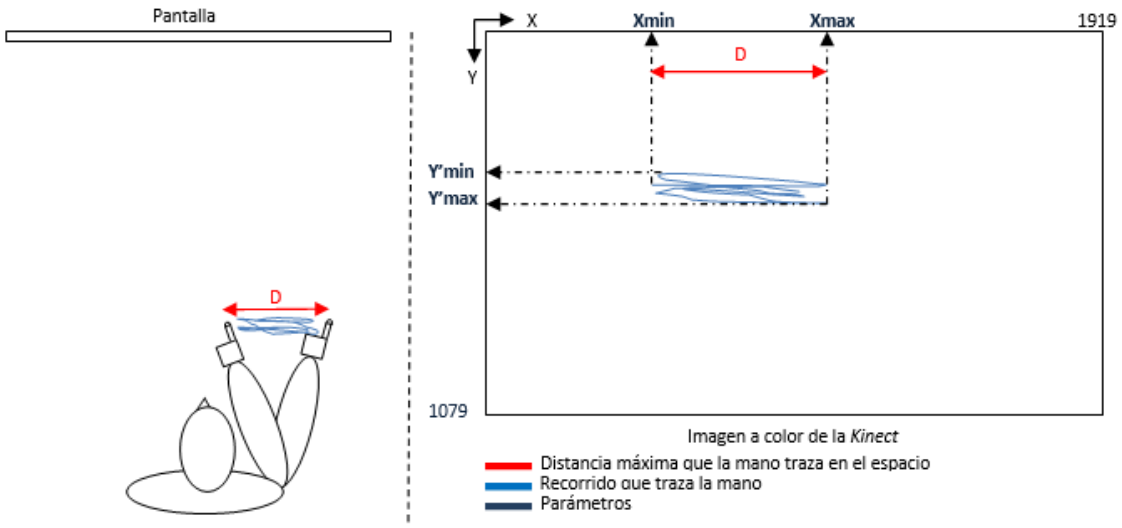


Figura 36.- Agitación del brazo y captación del trazado de la mano

La figura 37 enseña el algoritmo utilizado para la detección y recogida de los máximos y mínimos a tiempo real.

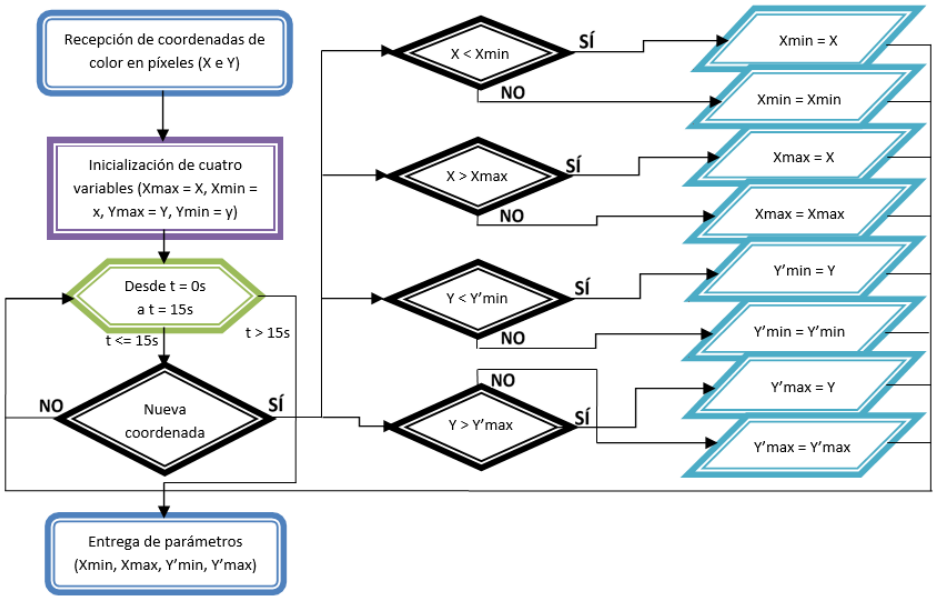


Figura 37.- Algoritmo para la obtención de parámetros

Por último, como se observa, no se contempla la obtención del rango de movimiento de la mano a lo alto. Para esto, se vale más delante de los parámetros *Y'min* y *Y'max* para calcularlo.

3.4.4 Envío, recepción y generación de nuevos parámetros de altura

Como enseña el esquema general (figura 32) de la solución, los parámetros se envían en un paquete UDP al *middleware* desde *Unity* **(paso 7)**. Su formato se presenta en la figura 38.

#K2UMUDP	DD	NNNN	XY	Xmax|Xmin|Y'max|Y'min|ShotMode|Boolean

Figura 38.- Formato de paquete con parámetros

El *middleware* recibe el paquete y extrae los valores del campo datos **(paso 8)**:

1. Parámetros: *Xmax, Xmin, Y'max* e *Y'min*
2. *ShotMode:* indica si vienen valores de la mano izquierda o la mano derecha. Su contenido es o *shot_left* o *shot_right*
3. *Boolean:* es un valor que contiene *True* o *False* e indica si el juego está en la escena de calibración actualmente

Todos estos valores son almacenados en una variable de tipo *Dictionary* que funciona con un sistema de claves y valores; por tanto, para acceder a esos valores solo habría que indicar el nombre de la variable y la clave.

Una vez se almacena, lo primero que se debe hacer es calcular- como se ha indicado anteriormente- el rango de movimiento a lo alto del jugador. Para ello se deben generar unos parámetros nuevos (*Ymax* e *Ymin*) cuya relación de su diferencia con la diferencia de *Xmax* y *Xmin* sea igual a la relación de aspecto de la imagen a color de la *Kinect* **(paso 9)**, es decir, el objetivo es conseguir:

$$(\mathrm{X_{max}} - \mathrm{X_{min}})/(\mathrm{Y_{max}} - \mathrm{Y_{min}}) = \mathrm{D}/\mathrm{A} = 1920/1080 = 16/9 \qquad (2)$$

Con el apoyo de la figura 39, se procede a explicar los pasos de la obtención de *Ymax* e *Ymin*:

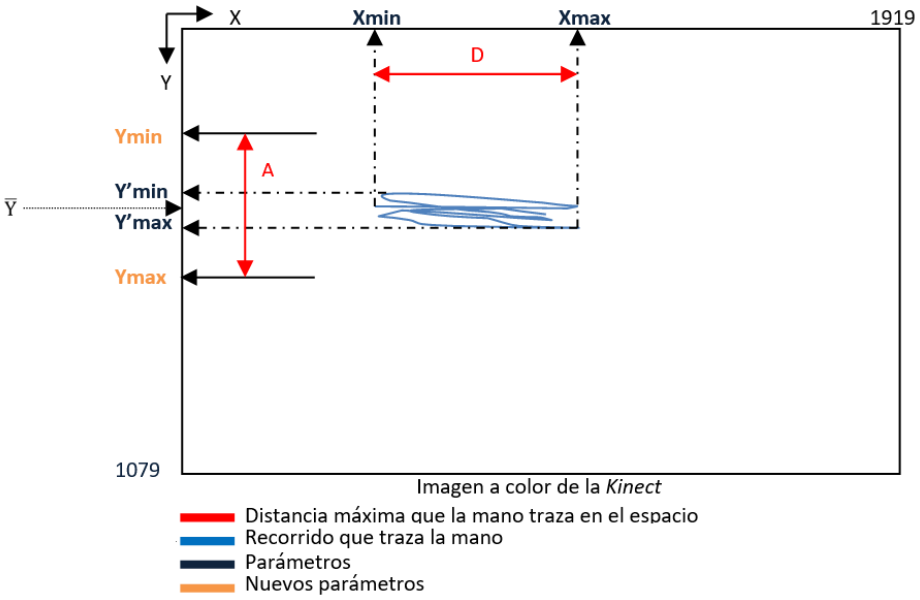


Figura 39.- Esquema ilustrativo para la obtención de nuevos parámetros

Las fórmulas generales son:

$$\mathrm{Y_{min}} = \bar{Y} - \frac{A}{2} \qquad (3)$$

$$\mathrm{Y_{max}} = \bar{Y} + \frac{A}{2} \qquad (4)$$

De lo que se conoce:

$$\bar{Y} = \frac{Y'_{max} + Y'_{min}}{2} \qquad (5)$$

Y de la fórmula 2 se saca:

$$A = \frac{D \times 1080}{1920} = \frac{(X_{max} - X_{min}) \times 1080}{1920} \quad (6)$$

Finalmente se obtiene:

$$\boxed{Y_{min} = \frac{Y'_{max} + Y'_{min}}{2} - \frac{(X_{max} - X_{min}) \times 1080}{2 \times 1920}} \quad (7)$$

$$\boxed{Y_{max} = \frac{Y'_{max} + Y'_{min}}{2} + \frac{(X_{max} - X_{min}) \times 1080}{2 \times 1920}} \quad (8)$$

La ventaja de generar de manera automática el rango de movimiento a lo alto, está en minimizar el esfuerzo que el jugador tuviera que hacer en la calibración del brazo, dado que también sería factible si el jugador agitara el brazo de arriba abajo para que no hiciera falta el cálculo de los nuevos parámetros *Ymax* e *Ymin*. Se ha querido mirar más por el jugador y restar esfuerzo.

3.4.5 Transformación del espacio 2D **(paso 10)**

Una vez se tienen las delimitaciones por las que se dibuja un rectángulo con la misma relación de aspecto que la imagen a color del detector de movimiento, el objetivo ahora pasa a ser el de extender dichas delimitaciones a los bordes de la pantalla local desde donde se está jugando. Llevar a cabo esto pasa por tres fases:

1. Escalado [22] a resolución del PC
2. Traslación [23]
3. Escalado 2D de ajuste final

Es importante mencionar que todas las coordenadas de la mano pasan por esta transformación; los parámetros son valores que nos ayudan a delimitar el movimiento y a establecer un rango como ya se ha explicado en anteriores apartados.

Se comienza con el escalado a resolución de la pantalla local. La figura 40 aclara el proceso.

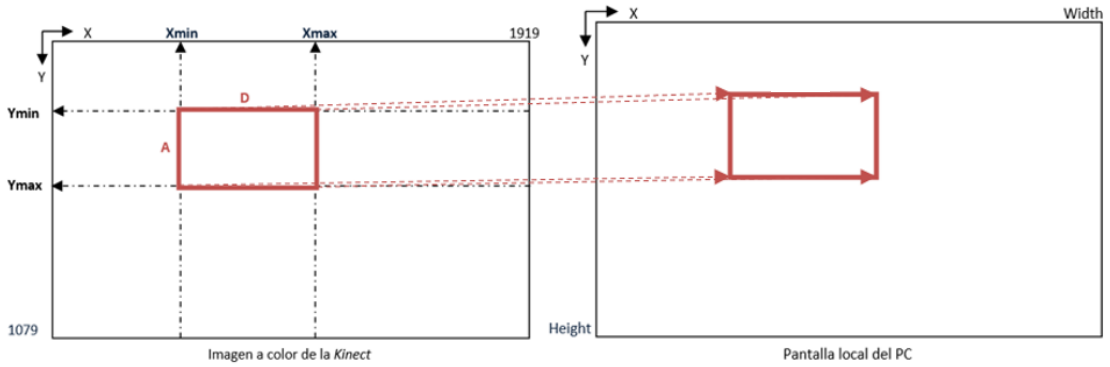


Figura 40.- Escalado a resolución local

Trasladar a resolución local significa que se está tomando en cuenta que el cursor a mover está funcionando en otra pantalla y no en la imagen a color de la *Kinect*. Es por ello, que hay que adaptar las coordenadas a dicha resolución.

Como se ve en la figura 40 solamente es necesaria la multiplicación de las coordenadas por valores que consigan pasar el espacio 2D de la imagen a color al espacio 2D de la pantalla local. Para ello existen dos pasos a realizar:

1) Pasar la coordenada que proporciona el dispositivo a un valor entre 0 y 1, incluyendo los parámetros:

$$X' = \frac{X}{1920} \tag{9}$$

$$Y' = \frac{Y}{1080} \tag{10}$$

2) Multiplicar el resultado de las operaciones anteriores (fórmulas 9 y 10) por el ancho y alto en píxeles de la pantalla local:

$$X'' = X' \times WIDTH \tag{11}$$

$$Y'' = Y' \times HEIGHT \tag{12}$$

Gracias a una función proporcionada por la API de C# se puede acceder al ancho y alto en píxeles del PC local.

Después viene la traslación, paso previo al ajuste final, que consiste en llevar el espacio rectangular al origen de coordenadas. Basta con una operación aritmética de sustracción, en el que se coge la coordenada y se le resta el valor de *Ymin* o *Xmin* dependiendo de si es una coordenada *X* o una coordenada *Y*:

$$X''' = X'' - Xmin \tag{13}$$

$$Y''' = Y'' - Ymin \tag{14}$$

A continuación, en la figura 40 se ilustra este efecto.

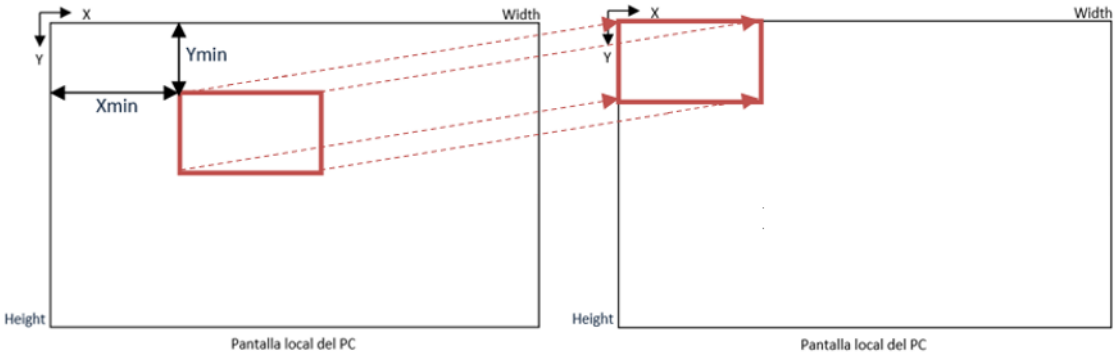


Figura 41.- Traslación del rectángulo al origen de coordenadas

Por último, se termina con el escalado de ajuste. Este se lleva acabo multiplicando las coordenadas resultantes de las fórmulas 13 y 14 por una constante a través de la cual se consigue finalmente ajustar el rango de movimiento a los bordes de la pantalla local. Para las coordenadas *X* hay una constante y, para las *Y*, hay otra; su cálculo se hace partiendo de la idea de que:

$$X'''_{max} \times C1 = WIDTH \quad (15)$$

$$Y'''_{max} \times C2 = HEIGHT \quad (16)$$

En consecuencia, las constantes son:

$$C1 = \frac{WIDTH}{X'''_{max}} \quad (17)$$

$$C2 = \frac{HEIGHT}{Y'''_{max}} \quad (18)$$

De modo que todas las coordenadas se multiplican por las constantes halladas:

$$X_F = X''' \times \frac{WIDTH}{X'''_{max}} \quad (19)$$

$$Y_F = Y''' \times \frac{HEIGHT}{Y'''_{max}} \quad (20)$$

La figura 42 muestra el resultado del escalado:

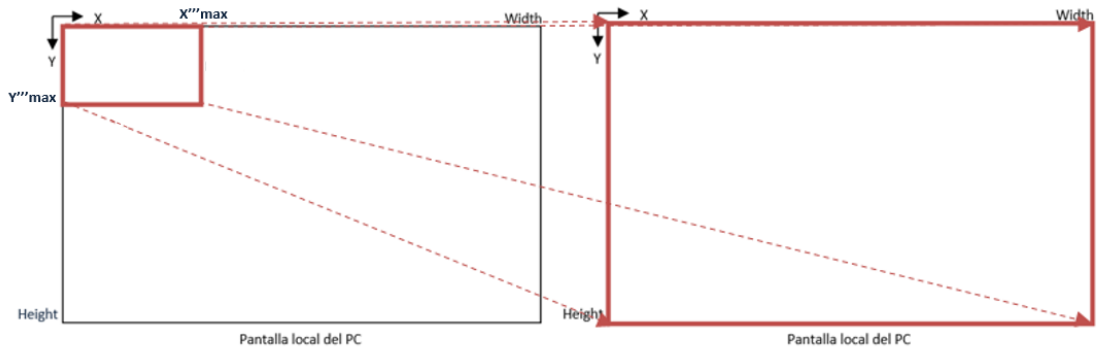


Figura 42.- Escalado de ajuste

Las coordenadas X_F y Y_F son las que se pasan por parámetros en la función:

- `SetCursorPos(`X_F`,` Y_F`)`

Tras esto último, ya se podría mover el cursor y, en consecuencia, el brazo 3D del juego **(paso 11)**; asegurándose de poder moverlo por toda la pantalla.

De forma superficial, se le han añadido unos algoritmos de control para solventar problemas como:

1) Cuando el jugador consigue controlar el brazo después de haber hecho la calibración puede erradicar el uso del ratón de manera que si un asistente quisiera tomar control del cursor no podría. Se debe proteger la posibilidad de usar el ratón cuando el jugador tenga el brazo relajado y no esté jugando.

Por ello, seguidamente se explica cómo se ha solventado

3.4.6 Control del movimiento del cursor/brazo 3D

Dentro del código del *middleware* y, concretamente en la clase de *KinectMiddleware.cs* aparece todo el código de la solución del brazo. En esta clase también aparece la sección de código donde se confeccionan los paquetes de datos con coordenadas; es precisamente en esta sección mencionada, donde se elige que se contenga la función para mover el cursor.

En primer lugar, se han establecido unos límites de coordenadas, dentro de los cuales se puede mover el brazo 3D. Si el jugador desea relajar el brazo sería necesario pararlo para que se pueda manejar con el ratón. Para entenderlo mejor se muestra la siguiente figura.

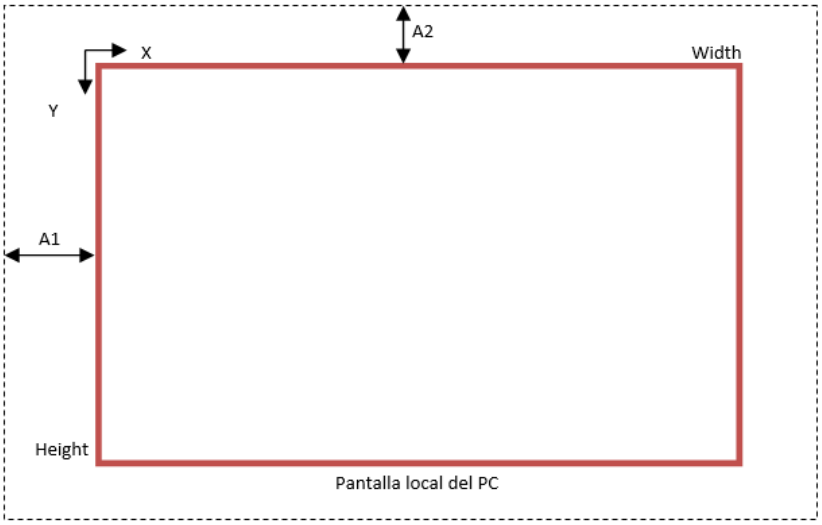


Figura 43.- Delimitaciones de movimiento

Cuando la posición de la mano sobrepasa la línea discontinua ya se deja de mover el cursor. Esta delimitación cumple:

$$A1 = \frac{WIDTH}{6} \tag{21}$$

$$A2 = \frac{HEIGHT}{6} \tag{22}$$

Por otro lado, se añade también el control de si existen parámetros o no. Esta condición es importante ya que no es de interés mover el cursor con coordenadas de valor nulo.

Otra funcionalidad que se le añade al juego es el botón de "*Reset*"; este, si se le pulsa, manda un paquete (figura 44) con parámetros vacíos al *middleware*:

#K2UMUDP	DD	NNNN	XY	0|0|0|0|ShotMode|Boolean

Figura 44.- Formato de paquete con parámetros vacíos

De este modo se fuerza a que no existan parámetros para volver a ejecutar la calibración. Este control es útil si el usuario quiere ajustar de nuevo su posición del brazo.

Por último, está el control de la elección de coordenadas de mano izquierda o de mano derecha. Gracias al valor del *ShotMode* es posible recoger unas coordenadas u otras.

A continuación, se explican las demás soluciones y, para ello, hay que tener en cuenta que en todas se usan coordenadas de posición.

3.5 Solución disparo, recarga y agachado

Una de las limitaciones que se han encontrado en la recepción de coordenadas es que, si se traen al mismo tiempo a *Unity* más de un tipo de las mismas (por ejemplo, coordenadas de color más coordenadas de posición), se provoca un retardo. Este provoca que no se consiga recibir en tiempo real todas las coordenadas una detrás de otra. Es así debido al tipo de almacenamiento y procesamiento que hay en el envío de datos/coordenadas. Los datos se van encolando en una variable especial en el *middleware*, tal como se ha explicado en el capítulo de procesamiento y envío de coordenadas. Si a esta cola le añades dos tipos de datos, en cada uno de ellos se multiplicaría por dos el tiempo que se tarda para su extracción en recepción; en definitiva, si se envían dos tipos, se tarda el doble en procesar un tipo de dato. Si se envían tres, tardaría el triple, y así sucesivamente.

Como se ha mencionado, las soluciones de recarga, agachado y disparo requieren de la recepción y procesamiento de coordenadas de posición durante el juego. Es por ello, que se ha optado por procesar las coordenadas de color- las requeridas para la solución del movimiento del brazo- en el *middleware*, evitando así, tener que recibir dos tipos de coordenadas (posición y color) en *Unity* mientras se esté jugando evitando así cualquier tipo de retardo.

Antes de proseguir, se cree conveniente dejar indicado con qué tipo de coordenadas se asocian cada una de las escenas:

1. Escena del *Menu*: recepción de coordenadas de posición para la calibración de la solución de la recarga (*Reload*) y agachado (*Crouch*)
2. Escenas 1, 2 y 3 del juego (*Scene, WestGun* y *Final*): recepción de coordenadas de posición para la ejecución de la recarga, agachado y disparo
3. Escena de la calibración del brazo (*Calibration*): recepción de coordenadas de color para la obtención de parámetros

Otro tema a explicar es que estas tres soluciones se valen de la existencia previa de un control para la recarga, agachado y disparo en el código del juego. Todas estas acciones están implementadas en una clase llamada *RayCaster.cs*, de la que merece la pena extraer sus códigos (figura 45).

```
//recargar en cualquier momento
if (Input.GetButtonDown ("Recargar") || FindObjectOfType<CalibrationManager> ().reload == true)
{
    StartCoroutine (Reload ());
    return;
}
if (Input.GetKey (KeyCode.Space) || FindObjectOfType<CalibrationManager> ().crouch == true) {
    animatorCamera.SetBool ("Crouching", true);
    isCrouching = true;
    return;
}else
{
    animatorCamera.SetBool ("Crouching", false);
    isCrouching = false;
}
if (Input.GetMouseButtonDown(0) || FindObjectOfType<CalibrationManager>().shoot) {
    //Debug.Log ("El daño es: "+TheDamage);
    currentAmmo --;
    updateAmmo(currentAmmo);
    animator.SetBool("animDisparar", true);
    StartCoroutine(Disparar());
    FindObjectOfType<CalibrationManager> ().shoot = false;
```

Figura 45.- Código donde se controlan las acciones de recarga, agachado y disparo

Como se observa, para cada acción existen dos condiciones: una en la que se debe apretar una tecla o botón (ver ANEXO I en la sección de controles) y otra en la que una variable de tipo *boolean* debe estar a "*true*". Todas las variables de este tipo se extraen de la clase *CalibrationManager.cs*, desde donde se controlan. Por tanto, cabe mencionar que todas estas últimas soluciones están implementadas en dicha clase.

Si se cumple una u otra condición, entonces se procede con la recarga, el agachado o el disparo. A continuación, da comienzo el desglose de cada solución.

3.5.1 Solución del disparo del arma

El disparo se presenta con algunas dificultades que han tenido que ser estudiadas para conseguir la solución más idónea. Una de las posibilidades para disparar hubiera sido la de simplemente doblar la muñeca, aunque esta solución presenta dos problemas: no hay suficiente precisión en la captura de las coordenadas de la mano y no todos los usuarios serían capaces de doblar dicha articulación.

La solución que mejor se ha encontrado es la de disparar de manera automática, sin necesidad de mover alguna articulación de más. También tiene ciertas ventajas terapéuticas porque se practica mantener el brazo lo más estático posible. Lo único que se tiene que hacer es apuntar un blanco y el arma se dispara de manera continua cada cierto tiempo. Este genera lo que se llama frecuencia de disparo. La frecuencia de disparo es configurable, aunque esto se especifica más adelante.

Para llevar a cabo esto, debe existir en el código un control de posición de la mano, de manera que, si esta no supera un rango (también configurable) del espacio 2D

dibujado por las coordenadas *X* e *Y* durante un tiempo mínimo, el arma dispara. En definitiva, para disparar se tiene que procurar apuntar y mantener el brazo quieto.

La figura 46 enseña el espacio 2D dentro del cual la mano debe moverse para poder disparar. Los dos rangos que aparecen tienen la misma medida y deben ser lo suficientemente pequeños como para que el usuario tenga que tener el brazo lo más quieto posible.

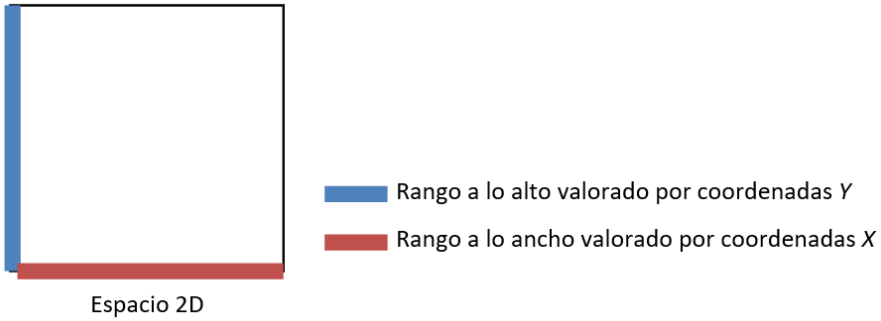


Figura 46.- Espacio máximo 2D para que el arma dispare

El algoritmo de esta solución está contenido en la clase *CalibrationManager.cs* donde se debe estar continuamente calculando máximos y mínimos de las coordenadas *X* e *Y* de la mano que vaya a disparar (algoritmo de la figura 47).

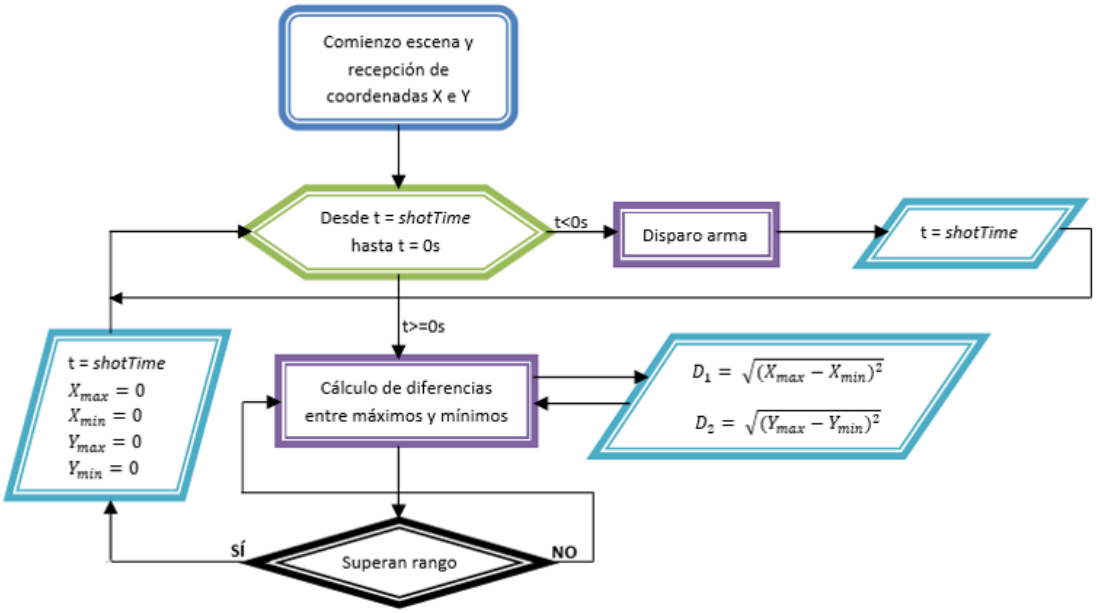


Figura 47.- Algoritmo para la solución del disparo

Se pone en marcha, en primer lugar, una cuenta atrás; esta coincide con el tiempo para establecer la frecuencia de disparo. Durante esa cuenta se valora si las diferencias de esos máximos y mínimos son menores al rango del espacio que se ha establecido. En paralelo una variable de tipo *boolean* estaría en estado "*true*"

continuamente y, por tanto, la cuenta atrás seguiría en marcha. Por el contrario, en cuanto esas diferencias superen el rango de espacio la variable pasaría a estado "*false*" y se resetearían todos los valores de coordenadas, no dejando que se prosiga con la cuenta atrás, estableciéndola a su valor inicial. La idea es conseguir que la variable tuviera valor "*true*" hasta el final de la cuenta; en ese momento otra variable de tipo *boolean* (*shoot*), se pone a "*true*" y se dispara el arma.

Como se observa en la figura 47 el tiempo "*shotTime"* es el nombre de la variable que contiene la duración entre cada disparo. No se especifica su valor ya que es configurable como se ha explicado anteriormente.

3.5.2 Solución de la recarga del arma

Para la recarga del arma se ha optado usar un movimiento de doblar el brazo. Para detectar este movimiento se calcula, a tiempo real, los ángulos de flexión de dos articulaciones. Esta detección se efectúa en el brazo configurado inicialmente para este movimiento, pudiéndose elegir cualquiera de los dos. En adelante se explica este procedimiento para el caso de sólo uno de los brazos, ya que es el mismo para ambos, aplicando las coordenadas correspondientes.

Se comienza con la calibración. Esta comienza en la escena del menú donde ya se estarían recibiendo las coordenadas de posición; después se accede al menú de "*Calibration*" y dentro de aquí se debe pinchar en "*Reload*" (ver figura 22.3) pero antes de esto el jugador debe ponerse en una posición correcta. El usuario debe elegir a qué posición del brazo quiere llegar para recargar, pero esto se aclara más adelante. Al pulsar en "*Reload*" el juego llama a una función interna (*positionsToReload())* contenida en la clase *CalibrationManager.cs* que registra las posiciones espaciales puntuales (*X, Y* y *Z*) de las articulaciones elegidas. Dichas articulaciones son las siguientes (figura 48):

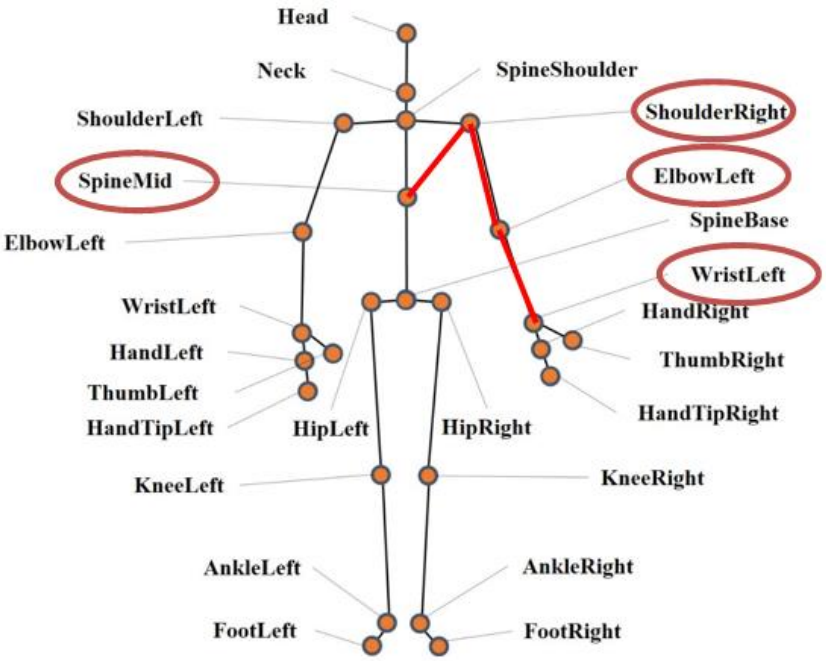


Figura 48.- Articulaciones registradas para la recarga

Una vez se registran las posiciones espaciales de estas articulaciones la función que ha sido llamada ejecuta otra (*calculateAngles()*) donde está el proceso de cálculo de los ángulos de flexión. De la figura 48 se extraen las líneas dibujadas por las articulaciones para aclarar qué ángulos se quieren calcular (figura 49).

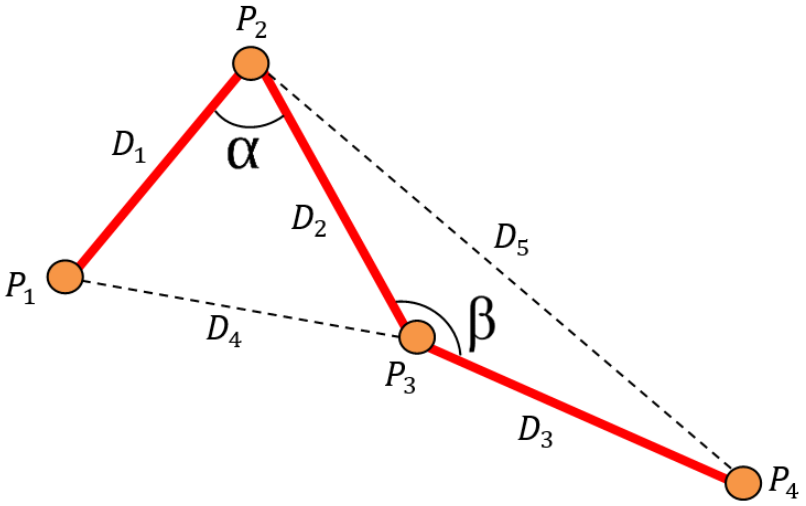


Figura 49.- Ángulos y distancias a calcular entre articulaciones

De la figura anterior, se pretende calcular los ángulos *Alpha* y *Beta*. Para proceder a ello la mejor solución es recurrir al "teorema del coseno" [24].

Para su cálculo, en primer lugar, se deben tomar en cuenta los triángulos de los que forman parte los ángulos (figura 44). Para llevar a cabo el teorema es fundamental el cálculo de distancias; por un lado, están las distancias D1, D2 y D4 para el triángulo 1

y, por otro, las distancias D2, D3 y D5 para el triángulo 2. Con la siguiente fórmula se puede calcular la distancia entre dos puntos en el espacio tridimensional:

$$d(p_1, p_2) = \sqrt{(x_{p_1} - x_{p_2})^2 + (y_{p_1} - y_{p_2})^2 + (z_{p_1} - z_{p_2})^2} \quad (23)$$

En base a la fórmula 23 las distancias a calcular son la siguientes:

$$D_1 = d(p_1, p_2) \quad (24)$$

$$D_2 = d(p_2, p_3) \quad (25)$$

$$D_3 = d(p_3, p_4) \quad (26)$$

$$D_4 = d(p_1, p_3) \quad (27)$$

$$D_5 = d(p_2, p_4) \quad (28)$$

Y, teniendo los valores de las distancias, se consiguen los valores de los ángulos:

$$\boxed{\alpha = \cos^{-1}\left(\frac{{D_1}^2 + {D_2}^2 - {D_4}^2}{2 \times {D_1}^2 \times {D_2}^2}\right)} \quad (29)$$

$$\boxed{\beta = \cos^{-1}\left(\frac{{D_2}^2 + {D_3}^2 - {D_5}^2}{2 \times {D_2}^2 \times {D_3}^2}\right)} \quad (30)$$

Estas últimas fórmulas se obtienen gracias al teorema; que dice que el lado opuesto al ángulo de un triángulo se calcula:

$${D_4}^2 = {D_1}^2 + {D_2}^2 - 2 \times D_1 \times D_2 \times \cos\alpha \quad (31)$$

$${D_5}^2 = {D_2}^2 + {D_3}^2 - 2 \times D_2 \times D_3 \times \cos\beta \quad (32)$$

Como último paso, se ha considerado que es mejor manejar los valores de los ángulos en grados, ya que las fórmulas 29 y 30 dan los valores en radianes. Con las fórmulas 33 y 34 se hace la conversión a grados:

$$\alpha^\circ = \frac{180 \times \alpha}{\pi} \quad (33)$$

$$\beta^\circ = \frac{180 \times \beta}{\pi} \quad (34)$$

Este procedimiento de cálculo de ángulos *alpha* y *beta* se da tanto en el registro puntual que se lleva a cabo en la calibración, como a tiempo real durante el juego. El objetivo al fin, es la de comparar los ángulos a tiempo real con dichos ángulos registrados.

De la comparación surge una condición y, cuando esta salte, el personaje del juego recarga arma, poniendo una variable de tipo *boolean* a "*true*" (*reload*).

Por tanto, los pasos serían:

1. Posición del jugador
2. Calibración y registro de ángulos puntuales *Alpha* y *Beta* del jugador
3. Comienzo del juego y cálculo de los ángulos a tiempo real
4. Comparación continua entre ángulos fijos registrados y ángulos dinámicos a tiempo real
5. Recarga de arma si la condición salta en tiempo de juego

Del primer paso queda pendiente aclarar la posición del brazo que debe tomar el jugador. Este simplemente tiene que colocar el brazo como le gustaría, es decir, la posición que elija sería la posición a la que debe llegar durante la partida para poder recargar. Por tanto, una vez el jugador consiga la colocación del brazo, se pulsa el botón "*Reload*" como se ha descrito anteriormente y se procede al registro (paso 2).

En el paso 3 se comienza a jugar y a calcular de manera continua los ángulos dinámicos a tiempo real, de igual modo que se calcula con los ángulos registrados en la calibración.

El paso 4 empieza nada más empezar a calcular ángulos dinámicos.

Del último paso queda aclarar la condición que hace que se pueda recargar el arma. Como se ha dicho, los ángulos dinámicos están siendo continuamente comparados con los registrados en la calibración. La condición consiste en detectar el momento en que estos ángulos coinciden con un rango de valores establecidos a partir de esos ángulos registrados. Para hacerlo más entendible se expone el siguiente esquema (figura 50).

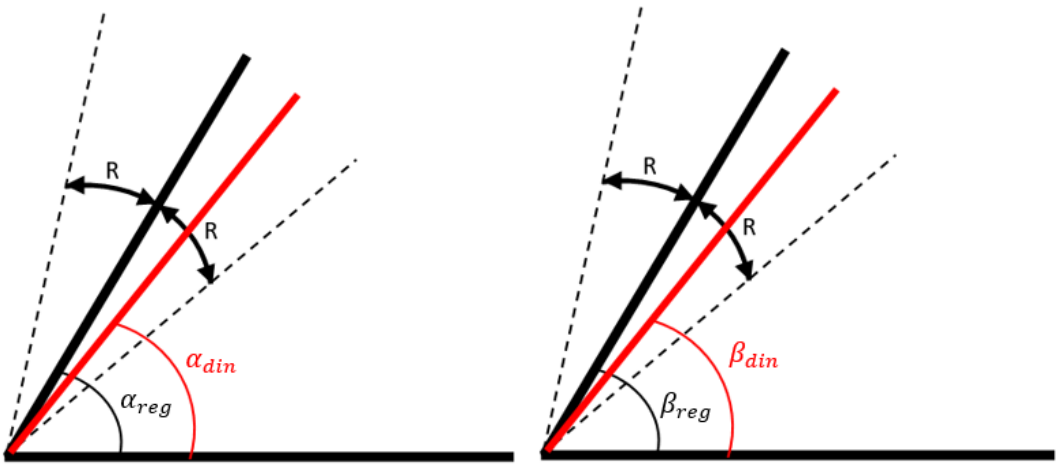


Figura 50.- Condición para la recarga del arma: negro ángulo registrado y rojo ángulo dinámico

Como se observa e la figura 50 si los dos ángulos *Alpha* y *Beta* dinámicos (rojo) coinciden dentro del umbral marcado por los rangos "R", que envuelven los ángulos registrados (negro), entonces se cumple la condición y se recarga el arma.

El rango R es un valor configurable cuya explicación tiene lugar más adelante. El establecimiento de este rango tiene su justificación en que, al ser dos ángulos a comparar, se necesita de un margen de error para que la condición no sea demasiado precisa; la precisión es importante, ya que si es muy elevada sería muy difícil encontrar la posición del brazo que se recoge en la calibración.

Por último, mencionar que todo el cálculo de ángulos y su comparación tiene lugar en la clase *CalibrationManager.cs*.

3.5.3 Solución del agachado del jugador

Esta solución sigue una filosofía parecida a la solución anterior, en el que se hace un registro inicial de las posiciones de varias articulaciones. Este registro también tiene lugar en el menú de *Calibration* donde habría que pinchar en la opción de "*Crouch*" (ver figura 22.3).

Por tanto, la solución consiste en los siguientes pasos:

1. Posición del jugador
2. Calibración y registro de ángulos puntuales *Gamma* y *Sigma* del jugador
3. Comienzo del juego y cálculo de los ángulos a tiempo real
4. Comparación continua entre ángulos fijos registrados y ángulos dinámicos a tiempo real
5. Agachado del personaje si la condición salta en tiempo de juego

El primer paso es referido a la colocación física del jugador antes del registro de coordenadas; este debe posicionarse de pie o sentado e intentando que las piernas estén lo más simétricas posibles entre sí. Por otro lado, al mismo tiempo debe colocar el cuerpo superior de manera erguida y seguidamente, inclinarse hasta conseguir la posición deseada. Esta última sería la posición a gusto del jugador, a la que debe llegar durante la partida para poder agachar el personaje.

Para el paso 2, cuando el jugador esté preparado con la posición deseada, el asistente que estuviera manejando el ratón debe pinchar en la opción de "*Crouch*" como se menciona anteriormente, para que, de manera interna, el juego llame a una función (*positionsToCrouch()*) contenida en la clase *CalibrationManager.cs*. La operación que lleva a cabo dicha función es la de registrar en varias variables las posiciones puntuales de las articulaciones elegidas para la solución. Una vez se almacenan dichas posiciones, al final de la función, se llama a otra (*calculateAngles()*) que procede con el algoritmo de cálculo de ángulos *Gamma* y *Sigma*.

Antes de proseguir con el paso 3, se procede a explicar cuáles son las articulaciones elegidas y las operaciones que se han llevado a cabo. En la figura siguiente se muestran los puntos elegidos.

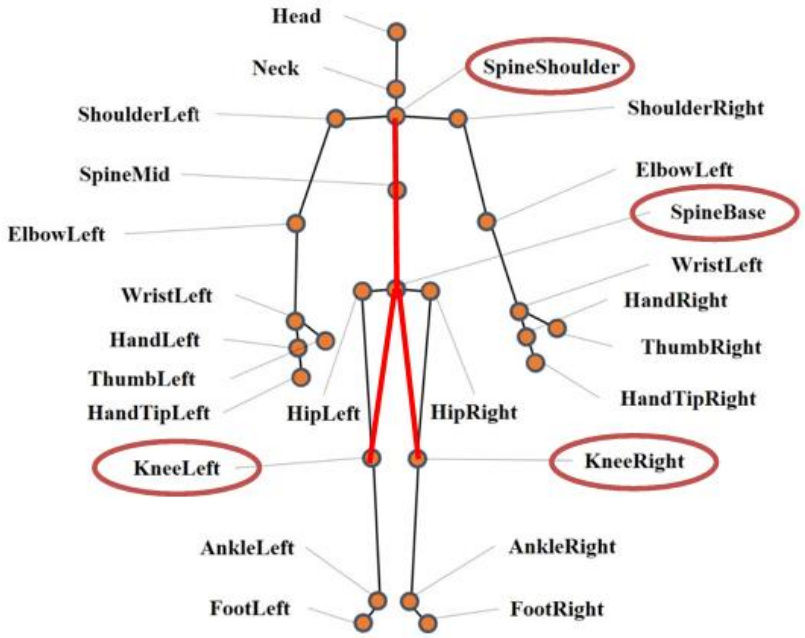


Figura 51.- Articulaciones registradas para el agachado

Las líneas que unen los puntos elegidos (en rojo), según como aparece en la figura anterior, se extraen a continuación (figura 52).

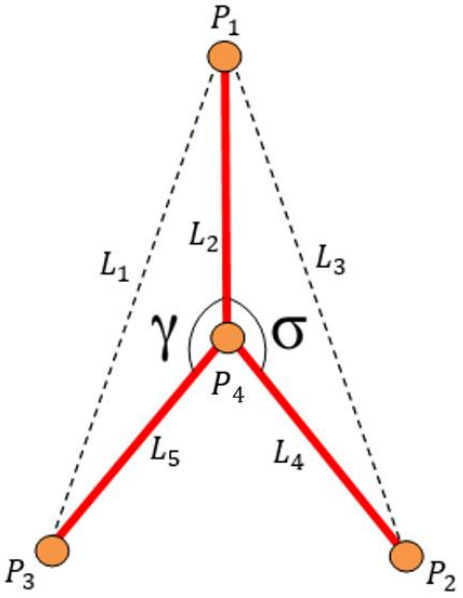


Figura 52.- Ángulos y distancias a calcular entre articulaciones

El punto 1 (P1), de la figura anterior, corresponde con la articulación *SpineShoulder*, el punto 4 (P4) con *SpineBase*, el 2 (P2) con *KneeRight* y el 3 (P3) con *KneeLeft*. La unión de todos ellos forma dos triángulos; uno de ellos se confecciona con los lados L1, L2 y L5 y el otro triángulo con los lados L2, L3 y L4. Los ángulos a calcular serían *Gamma (γ)* y *Sigma (σ)*. Los pasos a seguir son los mismos que se han descrito para la solución anterior. Primero se hayan los lados de los triángulos:

$$L_1 = d(p_1, p_3) \quad (35)$$

$$L_2 = d(p_1, p_4) \quad (36)$$

$$L_3 = d(p_1, p_2) \quad (37)$$

$$L_4 = d(p_4, p_2) \quad (38)$$

$$L_5 = d(p_4, p_3) \quad (39)$$

Y, a partir del cálculo de los lados, las fórmulas finales para calcular los ángulos son las siguientes (40 y 41):

$$\boxed{\gamma = \cos^{-1}\left(\frac{{L_2}^2 + {L_4}^2 - {L_3}^2}{2 \times {D_1}^2 \times {D_2}^2}\right)} \quad (40)$$

$$\boxed{\sigma = \cos^{-1}\left(\frac{{L_2}^2 + {L_5}^2 - {L_1}^2}{2 \times {L_2}^2 \times {L_5}^2}\right)} \quad (41)$$

Mencionar que, de igual modo que en la solución anterior, se pasan los valores a grados.

Una vez se tengan los ángulos calculados y almacenados ya se puede seguir con el paso 3. Este paso se ejecuta habiendo pasado por todas las calibraciones- en el caso de que se deseara usar todas, ya que no es obligatorio- en el que el asistente vuelve al menú principal y pincha en la opción "*PLAY*". En ese momento aparece la primera escena del juego donde comienzan a llegar coordenadas de posición actualizadas del jugador y, de manera análoga a la solución de la recarga, se van calculando constantemente los ángulos *Gamma* (γ) y *Sigma* (σ). Aquí también hay distinción entre ángulos fijos registrados y ángulos dinámicos a tiempo real.

El paso cuatro se refiere a la comparación continua de los ángulos dinámicos con los registrados en la calibración. De esta comparación surge la condición que debe saltar para poner la variable de tipo *boolean (crouch)* a "*true*", tal como se recoge en la figura 40.

Hasta aquí la solución tiene una similitud cercana a la de la recarga del arma. Lo único en que difieren es en la condición que se debe cumplir para ejecutar la acción. Por tanto, como último paso, se procede a la explicación de dicha condición para el agachado.

La condición salta en el momento en que los ángulos dinámicos (*Gamma* (γ_{din}) y *Sigma* (σ_{din})) coinciden o son menores que los registrados (*Gamma* (γ_{reg}) y *Sigma* (σ_{reg})) en la calibración. Por tanto, en tiempo de juego el jugador debe inclinar el tronco hasta alcanzar la inclinación que elige previamente en la calibración. La imagen siguiente (figura 53) ilustra de manera más clara la condición.

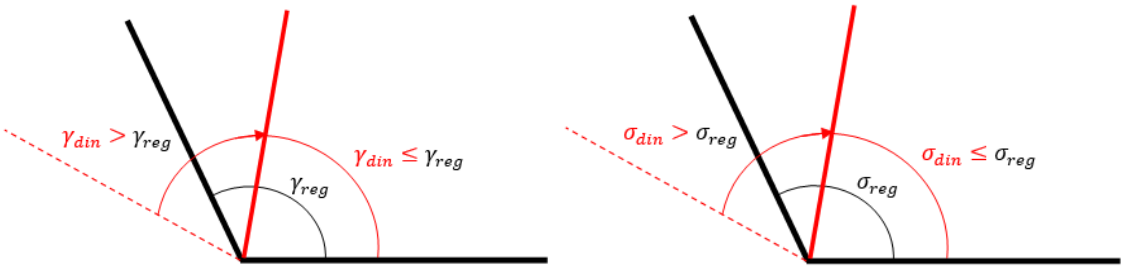


Figura 43.- Condición para el agachado: negro ángulo registrado y rojo ángulo dinámico

En la figura 53 se ven representadas las dos condiciones que se deben cumplir:

$$\gamma_{din} \leq \gamma_{reg} \quad (42)$$

$$\sigma_{din} \leq \sigma_{reg} \quad (43)$$

Se tienen que cumplir las dos para que el personaje del juego se agache, por eso se debe insistir en procurar jugar con las piernas posicionadas de manera simétrica.

3.6 Ampliación del modo de juego

En este capítulo se procede a explicar todo lo relativo a las posibilidades de configuración que tiene el juego *WestGun-Therapy*.

3.6.1 Integración con la web (parámetros de configuración online)

Uno de los objetivos que se han expuesto en el presente documento ha sido la necesidad de integrar el juego con la página web desde donde un terapeuta establece unos parámetros de configuración. No es materia de explicación la implementación y funcionamiento de dicha integración, ya que esta ha sido llevada a cabo por Daniel Iglesias (ANEXO III). Lo que se expone aquí es la interacción que hay entre el juego y la recepción de parámetros desde la web, descripción de dichos parámetros y envío de resultados desde el juego a la página.

Se recuerda que la comunicación existente entre el juego y la web se lleva a cabo a través del *middleware*. Este, al ser arrancado, trae los parámetros personales configurados por el terapeuta de la base de datos y los almacena (también existe la opción de no traerlos, si se deseara jugar con valores por defecto) para que, al arrancar el videojuego se envíe un paquete de petición de parámetros de configuración al *middleware* y este responda con otro paquete que los contenga en el campo Datos. Si se elige jugar sin parámetros, el paquete último descrito iría sin datos.

Lo siguiente es describir los parámetros posibles a configurar desde la plataforma web, la cual deja establecer un máximo de cuatro parámetros. Para el juego de *WestGun-Therapy,* se han definido estos parámetros como tal:

1. "*Level*": este parámetro establece la dificultad del juego, de modo que, el valor "1" es el nivel más fácil y el "5" el más difícil. El criterio de dificultad se basa en la cantidad de enemigos que debe haber en cada escena. Por tanto, cuando dentro del juego, en una clase llamada *gestorJuego.cs,* se recoge el valor de este parámetro, se establece el número de enemigos en unas variables llamadas "*numEnemies1*", "*numEnemies2*" y "*numEnemies3*", dependiendo del nivel elegido. En la tabla 1 se describe la relación entre el nivel y el número de enemigos:

Tabla 1.- Relación de número enemigos con el nivel elegido

Nivel	Escena 1 (nº latas)	Escena 2 (nº enemigos)	Escena 3 (nº enemigos)
1	1	2	1
2	2	4	1
3	2	6	1
4	3	8	2
5	4	10	2

2. "*shot-range*": recordando la figura 46, este valor representa el rango a lo alto y ancho del espacio 2D por el que se vale el arma para disparar. Cuando el juego recibe este parámetro, su valor se almacena en una variable llamada "*shotRange*". Debe ser introducido su valor en metros.
3. "shot-time": este valor establece la frecuencia de disparo descrito en el capítulo "Solución del disparo del arma". Cuando el juego recibe este parámetro, su valor se almacena en una variable llamada "*shotTime*". El valor se introduce en segundos.
4. "*reload*": este parámetro establece el brazo que se quiere usar para recargar el arma. Para el brazo izquierdo se debe introducir "0" y el derecho un "1".

Para el último parámetro lo ideal sería escribir "*reload_left*" o "*reload_right*", debido a que estas cadenas de caracteres son las que realmente se valoran para distinguir entre un brazo u otro dentro del juego; pero la interfaz de la página solo permite introducir dígitos. En consecuencia, en el código del juego, concretamente en la clase *gestorJuego.cs,* se hace una traducción de los valores "0" y "1" a "*reload_left*" y "*reload_right*" respectivamente, guardándolos en una variable llamada "*reloadMode*".

Mencionar que los resultados posibles a obtener por el terapeuta de cada partida son:

1. Intentos realizados
2. Tiempo requerido para pasarse el juego

Estos resultados son susceptibles de ser mejorados de cara al futuro; para este PFG no han sido tomados en cuenta en el apartado de resultados.

Los últimos cambios a describir que quedan son los provocados por los parámetros de configuración local

3.6.2 Menú de configuración (parámetros de configuración local)

Como se ha explicado anteriormente, se ha añadido a la nueva versión del juego un menú de configuración (figura 23). Este contiene parámetros que no podrían elegirse desde la web, sino, solamente de manera local. Se ha considerado que pueden ser útiles para el usuario si quisiera usarlos. En total, se pueden establecer tres parámetros:

1. "*Degrees to reload*": en este campo se pueden establecer los grados de precisión de los ángulos de flexión de las articulaciones elegidas para recargar. El valor se corresponde con el valor "R" que aparece en la figura 50. Si, una vez introducido el valor, se pulsa el botón "*Load*", de manera interna se llama a una función contenida en la clase *gestorJuego.cs* que recoge dicho valor y lo almacena en una variable llamada "*reloadDegrees*".
2. "*Shot Mode*": si se deseara usar este parámetro, el campo debe contener la cadena "*shot_right*" o "*shot_left*". El primero establece el brazo derecho para disparar y, el segundo, el izquierdo. Si, una vez introducido el valor, se pulsa el botón "*Load*", de manera interna se llama a una función contenida en la clase *gestorJuego.cs* que recoge dicho valor y lo almacena en una variable llamada "*shotMode*". Se debe recordar que el contenido de esta variable es la que se manda con el paquete de parámetros (figura 33) al *middleware*, ya que es en este dónde se toma control del cursor. Por tanto, una vez almacenado el contenido en la variable se manda al *middleware* el paquete mencionado. En definitiva, existen dos puntos donde se valora el presente parámetro; uno en el *middleware* para usar coordenadas de *HandRight* o *HandLeft* y otro en *Unity* para establecer el brazo 3D izquierdo o el brazo 3D derecho.
3. "*Easy Mode*": este parámetro activa lo que se llama el "*Easy Mode*". Activar este modo implica que se juega en modo fácil, de manera que, se podría usar este modo para cada nivel de dificultad. Si, una vez activado, se pulsa el botón "*Load*", de manera interna se llama a una función contenida en la clase *gestorJuego.cs* que valora si está activado o no y lo almacena en una variable de tipo *boolean* llamada "*easyMode*". Si está activado, la variable se pone a "*true*", de lo contrario se pondría a "*false*". Si se elige el *easy mode,* repercutiría en los enemigos haciendo que el margen de frecuencia de sus disparos sea más lento. También consigue que cada vez que se eliminen más enemigos, más lentitud existiría en dichos disparos. En general, la frecuencia se establece de manera aleatoria, no debe sorprender que en algún momento esa frecuencia aumente. Si no se elige el modo fácil, la frecuencia de disparo es siempre la misma y de manera continuada, sea cual sea el número de enemigos existente.

Cuando se configura un número de enemigos menor a la cantidad posible, el resto quedan desactivados, pero no destruidos. Esto quiere decir que, durante el juego, de manera interna, aunque estén desactivados, también disparan. Este disparo no repercute en la vida y no provoca nada más que un margen de tiempo para poder eliminar a los que sí están activados. Este efecto hace que se pueda jugar en modo fácil. Cuantos menos enemigos desactivados, menos margen de tiempo libre existe, es por ello, que el nivel cinco, al tener todos los enemigos activados, el efecto no se produce. Por tanto, el nivel cinco solo es posible jugarlo en modo difícil.

Se ha hecho mención hasta aquí de todos los parámetros de configuración existentes en el juego. Como funcionalidad especial, todos ellos pueden ser guardados para futuras partidas. Como último apartado, a continuación, se explica cómo se consigue guardar esos parámetros y cómo repercute en *WestGun-Therapy*.

3.6.3 Fichero de configuración "*PlayerSettings.json*"

Cómo se ha mencionado, existe la posibilidad de guardar para futuras partidas todos los parámetros de configuración existentes que hay en el juego; aunque también se guardan otros valores. Antes de comenzar con la lista de valores, cabe mencionar que el fichero contenedor de todos estos está en formato "*json*" [27]. Dicho formato de texto hace más fácil el acceso desde el código a dichos valores de los parámetros guardados en el fichero, ya que el formato provee de un sistema de organización basado en claves-valores.

Dicho esto, la lista de claves-valores es la que sigue:

1. "*reloadDegrees*":valor
2. "*numEnemies1*":valor
3. "*numEnemies2*":valor
4. "*numEnemies3*":valor
5. "*shotRange*":valor
6. "*shotTime*":valor
7. "reloadMode":valor (*reload_left* o *reload_right*)
8. "shotMode":valor (*shot_left* o *shot_right*)
9. "easyMode":valor (*false* o *true*)
10. "*alpha*":valor
11. "*betta*":valor
12. "*gamma*":valor
13. "*sigma*":valor
14. "*Xmax*":valor
15. "*Xmin*":valor
16. "*Ymax*":valor
17. "*Ymin*":valor

El hecho de poder guardar las ocho últimas claves permite que, una vez hechas todas las calibraciones del juego, no sea necesario volverlas a hacer en posteriores partidas.

Hay dos posibles formas de guardar los parámetros contenidos en *PlayerSettings.json*. Una es que se guarde de forma automática, cuyos casos son los siguientes:

1. Cuando se juega con la web y el juego recoge los parámetros
2. Al terminar cada una de las calibraciones

La otra posibilidad es guardar de forma manual, si se deseara. Solamente se tiene que pulsar la tecla “S” del teclado del PC.

Al igual que existe la posibilidad de guardar, también es posible cargar los parámetros al juego. De forma automática se hace cuando se arranca el juego y manual, pulsando la tecla “L”.

En el caso de que al cargar la configuración no se haya establecido ningún parámetro, existen unos valores por defecto que vienen previamente en el fichero que se debe proporcionar a la hora de instalar el juego (ANEXO II):

1. "*reloadDegrees*":15
2. “*numEnemies1*”:1
3. “*numEnemies2*”:2
4. “*numEnemies3*”:2
5. “*shotRange*”:0.02
6. “*shotTime*”:1
7. “reloadMode”:*reload_right*
8. “shotMode”:*shot_right*)
9. “easyMode”: *true*
10. “*alpha*”:valor
11. “*betta*”:valor
12. “*gamma*”:valor
13. “*sigma*”:valor
14. “*Xmax*”:valor
15. “*Xmin*”:valor
16. “*Ymax*”:valor
17. “*Ymin*”:valor

4 Resultados obtenidos

En este apartado se han querido dejar reflejados una serie de resultados obtenidos a partir de probar el juego con varias personas de entre 20 y 26 años de edad. Estos resultados han sido confeccionados sin tener en cuenta los resultados que se mandan a la web.

A pesar de no haber tenido la oportunidad de probarlo con personas con algún tipo de movilidad reducida, estos resultados permiten tener una idea muy clara del funcionamiento correcto de la solución.

Durante el desarrollo técnico del proyecto ha habido una continua prueba y error a medida que se iba avanzando con cada solución, aunque solamente se usaba una de las escenas. Estos resultados han dejado entrever el funcionamiento global del juego y, sobre todo, ha permitido establecer hasta qué nivel un jugador es capaz de llegar.

También, con respecto a toda la configuración posible del juego, se ha podido tener una idea clara de los parámetros óptimos para cada nivel.

En la tabla 2 se han registrado todos los resultados obtenidos. Se ha hecho una clasificación por jugadores y, por cada uno de ellos, una clasificación por niveles. Los datos a registrar son los siguientes:

1. Periodo de tiro: es el tiempo que establece la frecuencia de disparo, es decir, la duración entre disparos
2. Margen de recarga: el ángulo que establece el rango de precisión para la detección de la recarga (ver figura 50)
3. Margen de disparo: el valor que establece la precisión del disparo (ver figura 46)
4. Brazo para disparo: brazo que se usa para disparar y mover el brazo
5. Brazo para recarga: brazo que se usa para la recarga del arma
6. Modo fácil: establece si se juega en modo fácil o no
7. Se lo pasa: establece si el jugador se ha pasado el juego o no

Tabla 2.- Resultados registrados de varios jugadores

Jugador	Nivel	Periodo de tiro (s)	Margen de recarga (grados)	Margen de disparo (cm)	Brazo para el disparo	Brazo para la recarga	Modo fácil	Se lo pasa
1	1							
	2	1	20	2	D	I	si	si
		0.3	10	2	D	I	no	si
	3	1	15	2	D	I	si	si
		0.3	15	2	D	I	no	si
	4							
	5							
2	1	1	15	2	D	I	si	si
	2	1	15	2	D	I	si	si
		1	15	2	D	I	no	si
	3	0.3	15	2	D	I	no	no
	4							
	5							
3	1							
	2	1	15	2	D	I	si	si
	3							
	4							
	5							
4	1	0.3	15	2	D	I	si	si
		1	15	2	D	I	no	si
	2							
	3	1	15	2	D	I	si	si
		1	15	2	D	I	no	si
	4							
	5							
5	1							
	2							
	3	1	15	2	D	I	si	si
	4							
	5							
6	1							
	2	0.3	15	2	D	I	si	si
	3	1	15	2	D	I	si	si
	4	0.3	15	2	D	I	si	si
	5							

Los resultados ponen de manifiesto la eficacia de poder finalmente jugar el juego con el cuerpo. Se observa los niveles conseguidos en relación con los parámetros de configuración. Esto es importante de ver para poder hacer una serie de recomendaciones antes de disponerse a jugar. Las lecturas de que se pueden hacer de la tabla son las siguientes:

1. El nivel 1 y 2 pueden ser niveles asequibles para las personas con movilidad reducida. Se ha observado que han sido relativamente fáciles de pasar por los usuarios. En modo fácil se hace mucho menos complicado.
2. Se ha comprobado que jugar con dos brazos (uno para el disparo y otro para la recarga) se hace más ameno y dinámico el juego. El hecho de poder mantener el brazo quieto para disparar mientras se recarga con el otro, permite dar más rapidez a la eliminación de los enemigos.
3. Es más asequible cuando se juega con un periodo de disparo más pequeño. El valor “0.3” de dicho periodo hace que se consiga pasar la última escena con dos enemigos, sin embargo, si se le pone un valor de un segundo reduce esa posibilidad a cero.
4. El valor que establece la precisión de la recarga más idónea es 15 grados. Existe una buena relación entre la eficacia de la detección de los ángulos del brazo y la precisión con que el jugador debe posicionar el brazo.
5. El nivel máximo al que se ha llegado es el 4 en modo fácil. Aunque, según lo observado, se cree posible llegar a pasarse el nivel cinco con un periodo de disparo de 0.3 segundos.

5 Conclusiones

Recordando los objetivos explicados en el apartado 1.1 ha sido posible la adaptación completa de la tecnología *Kinect* al juego de *WestGun* de forma satisfactoria. Cada uno de los controles se pueden ejecutar de manera precisa sin fallos destacables. Es cierto que, por limitaciones del propio juego, a veces puede resultar confuso ejecutar los controles con el cuerpo. Esto ocurre, por ejemplo, en la ejecución del agachado; cuando uno quiere agacharse durante el juego, sea jugando con el cuerpo o con el PC, necesita esperar un retardo de un segundo hasta que se vea que el personaje del juego se agacha.

El efecto de amplificación implementado en la solución para el movimiento del brazo ha resultado ser un éxito. Con este efecto es posible establecer cualquier rango de movimiento en la calibración y poder controlar el cursor por toda la pantalla. Por otro lado, el agachado, recarga y disparo ha resultado cómodo para todos los jugadores que se ven reflejados en la tabla 2. Además, al tener la posibilidad de elegir las posiciones del cuerpo hace que sea aún más fácil.

Por último, se ha observado que, con el gran abanico de posibilidades de configuración del juego, se hace más interesante *WestGun-Therapy*, ya que no hay posibilidad de estancarse en un solo modo de juego. Esto permite descubrir y adaptar cuál sería la forma de jugar más idónea para el jugador, ya que cada uno tiene facultades distintas. Todo ello se ha implementado con éxito y, gracias a las pruebas llevadas a cabo, se ha corroborado su funcionamiento.

No obstante, se han encontrado algunos puntos que podrían ser mejorables de cara al futuro.

6 Futuras líneas de trabajo

Existen mejoras que pueden ser implementadas en un futuro. Entre ellas están las propias del juego *WestGun-Therapy*. Se ha visto que aún se puede hacer el juego más óptimo y real para el jugador:

1. No mover el cursor cuando se apunta: a veces resulta complicado intentar apuntar y procurar que el cursor se quede quieto sobre el objetivo. El temblor de la mano podría pasar factura a la hora de eliminar un enemigo. Es por ello, que podría conseguirse dejar el cursor estático en esos momentos en los que se apunta.
2. Mejorar el agachado: en cuanto a la rapidez de respuesta del personaje del juego para evitar los retardos.
3. Ampliación del juego añadiendo más pantallas y niveles.
4. Establecer unos resultados para la plataforma web más útiles de los que ya están implementados.

Por último, para futuras pruebas con el juego, es necesario poner en práctica el uso del mismo con aquellas personas que pudieran tener algún tipo de movilidad reducida. De esta manera es posible confirmar la ayuda que supondría este proyecto para ellas.

Referencias

[1] M. Eckert, *"The Blexer system – Adaptive full play therapeutic exergames with web-based supervision for people with motor dysfunctionalities"*, EAI Endorsed Transactions on Serious Games, 2017.

[2] UPM «Grupo de Aplicaciones Multimedia y Acústica (GAMMA)». *Fecha de última consulta 02/02/2018* https://www.citsem.upm.es/index.php/es/personal/grupos/personal-gamma

[3] UPM «Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad (CITSEM)». *Fecha de última consulta 02/02/2018* https://www.citsem.upm.es

[4] Wikipedia. 2018. *Videojuego de disparos en primera persona.* 2018. pág. 1. https://es.wikipedia.org/wiki/Videojuego_de_disparos_en_primera_persona

[5] Microsoft «Kinect - Desarrollo de Aplicaciones» *Fecha de última consulta 02/02/2018* https://developer.microsoft.com/es-es/windows/kinect

[6] G. Cilleruelo, A. Salom, N. Guillén. 2018. *GDD_V1 (WEST GUN).* 2018.

[7] Wikipedia. 2018. *C Sharp.* 2018. pág. 1.

[8] Nairobia Research. [En línea] 2015. [Citado el: 10 de julio de 2019.] https://neurobia.com/somos/.

[9] Domínguez, Ana. Umana. [En línea] 10 de julio de 2019. http://www.umana.es/pruebas-biomecanicas/.

[10] vinCi.[En línea] 2006. [Citado el: 10 de julio de 2019.] http://www.umana.es/vinci-realidad-virtual-rehabilitacion/.

[11] Kinect.[En línea] 2008. [Citado el: 10 de julio de 2019.] https://es.wikipedia.org/wiki/Kinect#Otros_usos_propuestos_para_el_Kinect

[12] Computer Hoy «Así es Kinect V2» *Fecha de última consulta 02/02/201,*

https://computerhoy.com/noticias/hardware/asi-es-kinect-20-windows-pc-10937

[13] C. Luaces Vela, *"Diseño e implementación de un entorno virtual de ejercicios físicos, basados en captura de movimiento"*, 2018

[14] Microsoft «Kinect for Windows SDK 2.0» *Fecha de última consulta 02/02/2018*

https://www.microsoft.com/en-us/download/details.aspx?id=44561

[15] *Riccitiello, John (octubre 23, 2014).* "John Riccitiello sets out to identify the engine

of growth for Unity Technologies (interview)". *VentureBeat (Interview). Entrevista con Dean Takahashi. Retrieved January 18, 2015.*

[16] Anders Hejlsberg, *"Anders Hejlsberg: Microsoft Technical Fellow"*. Microsoft Archivado del original el día 27 de abril de 2009. Recuperado en 2003-04-06.

[17] *Unity3d Assets*.[En línea] 2000. [Citado el: 10 de julio de 2019.] https://docs.unity3d.com/es/current/Manual/AssetWorkflow.html

[18] *Unity3d Prefabs*.[En línea] 2000. [Citado el: 10 de julio de 2019.] https://docs.unity3d.com/es/2018.2/Manual/Prefabs.html

[19] Microsoft Visual Studio. [En línea] 2019. [Citado el: 10 de julio de 2019.] https://es.wikipedia.org/wiki/Microsoft_Visual_Studio.

[20] Nacional, Universidad Tecnológica. Introducción a los cuaterniones. [En línea] noviembre de 2008. [Citado el: 10 de julio de 2019.]

http://www.edutecne.utn.edu.ar/cuaterniones/cuaterniones.pdf.

[21] Protocolo de transporte UDP. [En línea] [Citado el: 10 de julio de 2019.] http://umh2266.edu.umh.es/wp-content/uploads/sites/197/2013/04/T2_UDP.pdf.

[22] Geometría. [En línea] [Citado el: 10 de julio de 2019.] https://www.ugr.es/~jgodino/edumat-maestros/manual/4_Geometria.pdf

[23] Transformaciones geométricas. [En línea] 2013. [Citado el: 8 de julio de 2019.] http://catarina.udlap.mx/u_dl_a/tales/documentos/mcc/cruz_m_ia/capitulo3.pdf.

[24] Matesfacil. Teorema del coseno. [En línea] 2018. [Citado el: 8 de julio de 2018.] https://www.matesfacil.com/proco/trigonometria/teorema-coseno.pdf.

[25] *Unity3d Colliders*.[En línea] 2000. [Citado el: 10 de julio de 2019.] https://docs.unity3d.com/es/current/Manual/CollidersOverview.html

[26] Json. [En línea] noviembre de 2018. [Citado el: 7 de julio de 2019.] https://si.ua.es/es/documentacion/mootools/documentos/pdf/json.pdf.

[27] Jamhoury, Isa. Understanding Kinect V2 Joints and Coordinate System. [En línea] 23 de julio de 2018. [Citado el: 4 de julio de 2019.] https://medium.com/@lisajamhoury/understanding-kinect-v2-joints-and-coordinate-system-4f4b90b9df16.

[28] researchgate. [En línea] [Citado el: 10 de julio de 2019.] https://www.researchgate.net/figure/3D-skeleton-joints-tracked-by-the-Kinect-v2-sensor_fig1_282503184

ANEXO I: Documento final del proyecto *WestGun*

GAN ENTERTAINMENT

GAN
ENTERTAINMENT

Documento de diseño para:

West Gun

Miembros del equipo:

- Andrés Salom Velásquez/ [ASV]
- Gerard Cilleruelo Beltrán/ [GCB]
- Nicolás Guillén Echegaray / [NGE]

Versión #2.0
08/06/2018

Índice:

1. 73

1.1 Roles **2**

1.2 Trabajo realizado **3**

2. 75

2.1 75

2.2 75

2.3 75

2.4 75

2.5 76

2.6 77

2.7 78

2.8 78

3. 78

3.1 Interfaz dentro del juego **7**

3.2 Menús y/u otras pantallas auxiliares **7**

3.3 Armas y/o equipables **8**

3.4 Objetos e ítems **8**

3.5 Guión **8**

3.6 Logros y progreso **8**

3.7 Lógica y Scripts **8**

3.8. Técnicas **10**

3.9 Arte **11**

3.10 Componentes **11**

3.11 Fuentes **12**

1. Equipo

1.1 Roles

Los roles son los que siguen:

Para el rol de **líder** el mejor candidato es Andrés Salom Velásquez, que vela por la realización del trabajo, por tanto, se encarga del orden en general y que tengamos una involucración todos en el proyecto, sobretodo para el cumplimiento de los tiempos.

El **diseñador** es Nicolás Guillén Echegaray ya que la idea del juego es suya. Tiene una idea más clara de lo que se va a hacer.

En términos artísticos todos aportan de manera conjunta y se decide en base a reuniones. Aunque alguien debe poner las primeras ideas artísticas en el juego para tener un rumbo fijo. Ese es Andrés Salom Velásquez.

Por último, en cuanto a la **integración y programación** Gerardo Cilleruelo Beltrán es el que asume el rol.

Se manifiesta el compromiso de que todos los miembros del equipo tomen parte en todos los roles, por tanto, aunque uno solo por ejemplo tenga el rol de diseñador, todos se involucran en mayor o menor medida a ello.

1.2 Trabajo realizado

El trabajo realizado durante los meses de desarrollo del proyecto, hasta la fecha de entrega, ha sido el siguiente:

En primer lugar, trabajamos individualmente, nos centramos en desarrollar todo lo posible las capacidades técnicas con las herramientas que se van a usar, principalmente Unity y Blender. Cabe mencionar que los tutoriales del profesor Enrique Rendón, junto con otros aportes encontrados en YouTube, han servido como inicio de cada avance que se ha realizado.

Se dividió los cargos dentro del proyecto democráticamente, buscando siempre un equilibrio entre trabajo obligatorio y placer a la hora de realizar dicho trabajo. Una vez dividido el trabajo cada integrante del equipo trabajó en sus tareas, para posteriormente reunir las ideas y progresos una vez a la semana. De esta manera los problemas más tediosos se han ido resolviendo en grupo de manera más dinámica.

Andrés Salom ha sido el encargado de desarrollar los objetos principale y visualmente más importante del juego, el arma y los enemigos. Gracias a su capacidad artística se han conseguido resultados muy próximos a la idea original del juego. Ha

realizado las animaciones de estos, y en una visión general se ha encargado de juntar todo para obtener un resultado de manera uniforme (objetos, código, materiales, etc.).

Gerard Cilleruelo se ha encargado principal y fundamentalmente de la programación del juego. Ha centrado todo el trabajo en desarrollar sus capacidades en Unity para conseguir adaptar el código al juego. Se le han planteado dificultades en cuanto a funciones concretas, pero ha sido capaz de resolverlos gracias a la ayuda de internet y del profesor Enrique Rendón. Uno de los grandes avances ha sido la posibilidad de añadir un sistema capaz de controlar el audio del juego de manera intuitiva por los desarrolladores, de esta manera se podrá avanzar en cuanto sonido del juego. Su aporte se extiende también a la parte de modelado, con algunos objetos como barriles, cactus y por encima de estos, el escenario principal de la pantalla final del juego, el granero. Entre todas las funciones y características que ha conseguido implementar sería interesante destacar:

Movilidad del personaje principal, disparar a los objetos y enemigos y sus efectos correspondientes.

·Sistema de daños.

·Sistema “Game Over”.

·Implementación de la HUD

·Animación de enemigo y animación del FPS (agachado)¿?

·Implementación de los menús.

Nicolás Guillén ha desarrollado la idea principal del juego, poco a poco ha ido madurando un objetivo y una historia que se amoldara a las posibilidades técnicas de los integrantes del proyecto. En la parte más artística su trabajo en cuanto al modelado ha consistido en objetos secundarios de la escena, tales como: cajas de madera, vallas, troncos, piezas de madera, rueda de carro (partiendo de una base descargada "Wheelbarrow" by ahedov), casas.

Como grupo de trabajo la valoración es positiva. No siempre ha sido posible asistir a las reuniones propuestas por parte de los integrantes, pero se ha podido trabajar de manera fluida, en cada entrega se han cumplido los requisitos exigidos a tiempo.

2. Definición de las Bases

2.1 Resumen

West Gun narra la historia de un vaquero indio llamado “pistola del Oeste” que debe salvar la granja/fábrica de tabaco para pipa de la paz (G/FTPP), que ha sido invadida por unos aliens. El juego abarca lo nunca visto del “spaghetti western” desde el punto de vista de los que siempre perdían. Prepara tu técnica de disparo, porque en este shooter en primera persona no lo vas a tener nada fácil.

2.2 Mecánica del juego

El juego se basa en una simulación de una máquina **arcade**, podríamos definirlo como “jugar a que juegas”. La dinámica y los controles son básicos, adquiriendo las propiedades de clásicos como “Time Crisis” de las máquinas arcade. Se interactúa con el personaje con el ratón, el cual nos permitirá apuntar y disparar hacia los objetivos, la barra espaciadora con la que podremos ocultarnos y cubrirnos de los enemigos, y con la tecla “R” recargamos el arma manualmente. No existe control sobre el movimiento en el entorno del personaje, esto se debe a que el **avance que realiza es automático**, según se consiguen objetivos el jugador va avanzando a las siguientes zonas. Para poder avanzar se debe eliminar enemigos (en primer lugar y a modo de tutorial latas, posteriormente aliens).

2.3 Cámara

El juego hace uso de una cámara:

Cámara fija en primera persona. Se desplaza con el ratón la mirilla del arma que porta el personaje, pero no la cámara, los movimientos de esta están limitados debido a la automatización del avance del personaje.

2.4 Controles

Los controles referidos a los movimientos del jugador son limitados ya que su avance se hace de manera automática (descrito anteriormente). El único control atribuido al avatar es el de cubrirse, ejecutado con la **barra espaciadora** del teclado, en los momentos que aparezcan enemigos. Para apuntar al oponente se hace uso del joystick del **ratón** y disparar con el **botón izquierdo** del mismo. Con la tecla **R** del teclado se tiene la opción de recargar el arma.

Acciones a tener en cuenta: se puede recargar mientras se está de pie o agachado.

·De pie: mientras se recarga no se puede agachar. Al finalizar la recarga vuelve la libertad de movimiento y disparo.

·Agachado: se puede recargar, pero no disparar.

Con la tecla de **escape** el juego entra en "pausa", seguido de la aparición de un submenú con distintas opciones. Las opciones se pueden recorrer con el joystick del ratón y elegir una, pulsando el **botón izquierdo**.

2.5 Mundo del juego

Menú principal: Al ejecutar el juego vemos en primera instancia una máquina arcade, la cual al iniciar el juego se acercará y se situará de manera que solo veamos la pantalla de esta. Aquí tendremos el menú principal donde podremos ver las diferentes opciones del juego. Esta introducción hace referencia al tipo de juego al que nos enfrentamos, un guiño a las máquinas arcade.

Pantallas: Las distintas pantallas del juego van a girar todas en un entorno desértico. El ambiente intenta recordar al Viejo Oeste del siglo XIX, en el que tantas películas se han inspirado. Se cuenta con 3 distintas.

Sonido: Sonido ambiente de viento, efectos de disparos y explosiones, gritos, y música ambiente relacionada con el oeste.

Motivaciones:

2.6 Niveles

Pantalla tutorial: Paisaje desértico, con una cueva al fondo de la pantalla. En ella vemos diferentes objetos acordes con el contexto.

Pantalla de desarrollo: Pueblo árido, se pueden observar algunas casas de temática “western”.

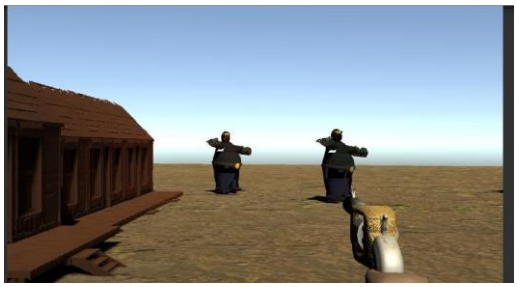
Pantalla final: Granero de grandes dimensiones que sufre una transformación inesperada.

2.7 Personaje principal

Durante el transcurso del juego nuestro personaje no estará visible, solo observaremos en pantalla el arma que esta porta. Pocos datos son conocidos por el jugador del personaje principal, su misión será eliminar los aliens que le atacan y se centrará todo el desarrollo en este punto.

2.8 NPCs y/o Enemigos

Aliens: Los enemigos del juego son criaturas extraterrestres muy territoriales que intentarán matar a nuestro protagonista. Disparan armas cargadas con el ácido de su propio cuerpo. Todos ellos son iguales, a excepción del último enemigo que es el final y más difícil de todos.



Primeras imágenes de los enemigos.

3. Detalles

3.1 Interfaz dentro del juego

·**Barra de vida:** La barra de vida tiene una estética sencilla de color verde en la que cada vez que el jugador recibe daño, esta disminuye su tamaño y deja al descubierto una barra roja. Se la ha añadido un marco con tonos marrones.

·**Munición:** Para representar la munición se opta por poner imágenes de bala. Las balas van desapareciendo de la pantalla a medida que vayamos disparando. Al recargar el arma vuelven las imágenes.

·**Mirilla:** Se representa con un círculo y una cruceta, la mirilla clásica de los juegos shooter.

3.2 Menús y/u otras pantallas auxiliares

Menú principal con las opciones de: *"play"*, *"options"* (hacemos control de volumen del sonido ambiente) y *"quit"* para salir del juego.

Submenú durante la partida del juego con las opciones de: *"resume"* para reanudar la partida, "menu" para volver al principal y *"quit"* para salir del juego.

Juego ganado/perdido: cuando pierdes o ganas una partida aparece un texto en el centro de la pantalla en el que se indica "You win"(ganar) o "You loose"(perder).

3.3 Armas y/o equipables

Revolver equipado de munición infinita (opción de mejora al subir de nivel) con disparo instantáneo y capaz de quitar vida a los objetos (la vida se define con una variable tipo int). Por cada disparo se quita 100 puntos de vida.

3.4 Objetos e ítems

No se encuentran consumibles ni objetos activos durante el desarrollo del juego, la interacción del jugador se limita a eliminar objetivos.

3.5 Guión

El juego comienza en un tutorial guiado, el personaje se encuentra en el desierto haciendo puntería con unas latas. Al fondo del paisaje se ve una cueva, al derribar las latas con éxito el personaje se dirigirá hacia ella. No es posible fracasar en esta primera misión.

En los niveles siguientes el jugador puede fracasar sufriendo daño provocado por los enemigos, la única manera de evitar esto es cubrirse tras los objetos destinados a ello. Por otro lado, si el jugador consigue eliminar a los enemigos de cada pantalla el personaje avanza, hasta que finalmente una vez eliminados todos los enemigos sucede una transición al siguiente nivel.

Por tanto, al superar el tutorial comenzará la acción del juego, la primera pantalla consiste en avanzar hasta el enemigo final. Se considera la última pantalla la batalla contra el enemigo final.

3.6 Logros y progreso

El progreso principal de juego consiste en avanzar en la trama de este. Por ello se comienza en un tutorial y acto seguido empieza la acción. Al eliminar objetivos se avanza hasta un objetivo final, un alien mucho mayor y más poderoso.

NOTA: Una mejora pendiente sería una tabla de puntuaciones en la que se almacene el lugar del jugador más rápido. De esta manera no solo se competiría por el avance en el juego si no que también por el tiempo conseguido.

3.7 Lógica y Scripts

La lógica principal del juego se basa en el avance de distintos estados al matar a los objetivos. Al tratarse de un juego en primera persona, la cámara hace una doble función:

Cámara: La cámara principal (Main Camera) tiene adjunta a sí misma 2 prefabs encargados del movimiento de la mirilla y disparar. Ella misma tiene un script encargado de ir cambiando los booleanos encargados de indicar cuando toca avanzar.

Personaje principal: al personaje principal se le llama player (objeto con una jerarquía menor a la cámara) y, en este, queda volcado un script donde se tiene el mayor peso de la lógica del juego. Este script recoge toda la funcionalidad del disparo en tanto a que, mediante un raycaster, se detectan los puntos de las superficies que se apuntan. Cuando se pulsa el botón izquierdo del ratón y además el raycaster está chocando contra un objeto, se hace una instanciado de un objeto que permite generar un efecto de chispas en el punto de choque del raycaster, además de una llamada a una función pública de otro script donde se resta vida al objeto. Este script también añade una función pública encargada de realizar la recarga del arma. Esta función cambia los respectivos booleanos para el cambio de estados de la animación de dicha recarga.

Munición: la munición está gestionada en el mismo script anteriormente descrito donde es controlado por una variable pública. Cada vez que se dispara se resta munición y se van actualizando las imágenes de la pantalla (balas) según el número contenido en las variables públicas. Esta actualización la hacen dos métodos implementados: updateAmmo() y setAmmo(). Este último es llamado una vez se recarga arma.

Vida: la vida del jugador es gestionada con otro código donde se controla el escalado de la barra verde con un vector cada vez que recibe daño. En este código se decide añadir un método encargado de actualizar el número de muertes de los enemigos para que, una vez que todos mueren, se pase a otra escena.

Rotación: para el movimiento del revólver se ha decidido crear un objeto vacío con una jerarquía mayor al player. A este se le mete un script en el que el código hace que el objeto vacío se dirija a un punto conformado por el origen del rayo del ratón, su vector y su punto medio.

Enemigos: los enemigos tienen sus propias interacciones con el juego, por tanto, tienen su propio script común a todos. Aquí se contiene la función pública para la actualización del daño de los disparos y la función encargada de la destrucción del objeto. Los enemigos modelados son hijos de un objeto vacío el cual contiene dichos scripts comunes, aunque estos tienen añadido otro script que se encarga de realizar el daño al jugador. Por otro lado, se decide añadir un nuevo objeto vacía llamado "EnemyController" que lleva un script encargado de añadir y activar cuantos enemigos se quieran.

Menú: en la función onClick() del botón "*play*" del menú se le añade otra función recogida en un script sencillo que aumenta en 1 el índice del array de escenas. Por tanto, de esta manera se cambia de escena, en este caso al juego. Existe también un cambio de menú cuando se da al botón "*options*" en el que su control se hace mediante booleanos dentro de la función onClick() creado por defecto en Unity. Dentro de options existe un control de volumen- con un "slider"- del sonido ambiente. Este consigue variar el valor del fader de un AudioMixer a través de un nuevo código dentro del script llamado "SettingMenu".

Submenú: se ha creado el script "*PauseMenu*". En las variables públicas del mismo está el crosshair, como variable de tipo transform y el pause menu UI (donde cuelgan los botones) como variable de tipo gameObject. Los botones son los siguientes:

ResumeButton: en on click () se llama a la función PauseMenu.Resume

MenuButton: en on click () se llama a la función PauseMenu.LoadMenu

QuitButton: en on click () se llama a la función PauseMenu.QuitGame

Transiciones de pantallas: existe un script dedicado a estas transiciones a la se ha llamado "*gestor Juego*". Incluye las posibilidades de reiniciar los niveles una vez pierdes o volver al menú. Añadir que se decide usar el script de uso público llamado "*Singleton*" para que el anterior tenga solamente una instancia en el juego.

Mira: para la mira del revólver también se ha creado un código que sustituye la posición del ratón por un canvas.

Gestor de audio: el script básicamente crea unas variables públicas (nombre del sonido, clip, volumen, pitch, loop) para poder hacer una gestión del sonido directamente

desde la interfaz de Unity. Luego se pueden llamar a los clips desde cualquier sitio de cualquier script.

Todo el código creado está pensado para poder ser reutilizado en las siguientes pantallas y con ello obtener una metodología de trabajo más eficiente.

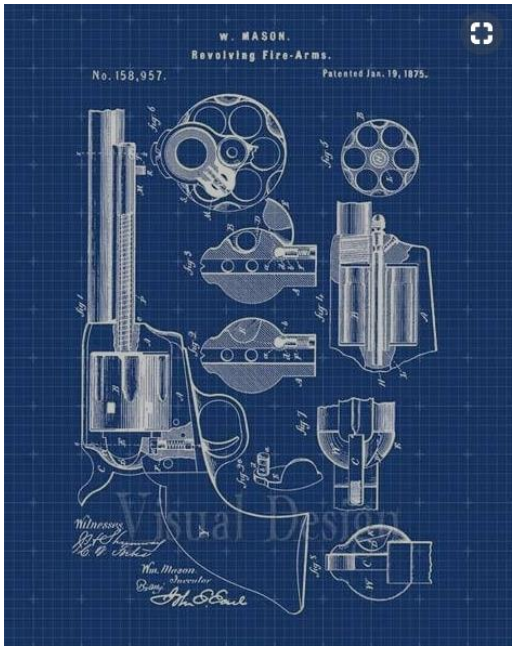
3.8. Técnicas

El procedimiento principal, tanto para el modelado y el código, ha sido la inspiración y el análisis de otros objetos o juegos. De esta manera se consigue ordenar la idea en un grupo de objetos inspiradores y poder sacar la idea con un toque personal.

Por otra parte, una metodología que agiliza mucho el trabajo es la utilización de **texturas PBR, texturas fotorealistas** que ahorran la necesidad de tener que pintar a mano el mapa UV y que funciona a la perfección con Unity, además estas texturas no tienen problema con distintos tipos de luces.

Otros programas usados como herramientas para mejorar la calidad final del juego han sido: **Photoshop y After Effects**, ambos del paquete Adobe.

3.9 Arte



Boceto pantalla prototipo (Andrés Salom). Esquema revolver inspiración

3.10 Componentes

1. *La organización de las escenas es la siguiente:*
    - ***a. Menu:***
        - *i. MainCamera*
        - *ii. Directional Light*
        - *iii. Canvas*
        - *iv. EventSystem*
    - ***b. Scene:***
        - *i. MainCamera*
            1. *Rotation*
                a. *Player*
                    i. *ModeladoPistola*
        - *ii. Canvas*
        - *iii. Enemy1*
        - *iv. Enemy2*
        - *v. Cueva*
        - *vi. Barriles*
        - *vii. Cactus*
        - *viii. Valla y carro (carro recurso creado por otras personas)*
        - *ix. Luz*
        - *x. Partículas para impacto*
2. *Por otra parte, las carpetas del proyecto están organizadas según el tipo para su fácil implementación en las siguientes fases de desarrollo, así su organización sería:*
    a. ***Animation:*** *animaciones hechas en el propio Unity*
    b. ***Materials:*** *materiales generados a partir de las PBRs*
    c. ***Metal Impact:*** *Texturas para las partículas que surgen al impactar en un objeto*
    d. ***PBRs:*** *Texturas fotorealistas (recurso creado por otras personas)*
    e. ***Prefabs:*** *Modelos Blender*
    f. ***Scenes:*** *Escenarios*
    g. ***Scripts:*** *Código para incluir en los objetos*
    h. ***Sounds:*** *Galería de sonido (recurso creado por otras personas)*
    i. ***FBX:*** *carpeta con contenido fbx creado*
    j. Baking pictures.
    k. ***Vídeos:*** *material de videos utilizado en el proyecto*

3.11 Fuentes

Tutoriales: toda la lógica del juego y programación se ha sacado de videos tutoriales de youtube. Los vídeos a destacar son los siguiente:

·Para la animación y recarga de del revólver: https://www.youtube.com/watch?v=kAx5g9V5bcM&t=373s

·Para el menú y submenú: https://www.youtube.com/watch?v=zc8ac_qUXQY

·Para el raycaster del player y destruccion de objetos: https://www.youtube.com/watch?v=mpxim8YbsMk

·Para la munición: https://www.youtube.com/watch?v=v7i7CnJblvA&t=202s

·Para la vida: https://www.youtube.com/watch?v=tDA7Y3lHIcw

·Para animación y daño de los enemigos: lista de reproducción "How to create a video game with Unity by Jayanam".

Objetos: Gran parte de los objetos que aparecen son inspiración de imágenes. Andrés Salom se ha basado en su carpeta de Pinterest hecha a propósito para el juego:

https://www.pinterest.es/andressalom/west-gun/

Los objetos utilizados de otros creadores son:

https://www.blendswap.com/blends/view/69141

http://www.blendswap.com/blends/view/2110

http://www.blendswap.com/blends/view/67729

http://www.blendswap.com/blends/view/69831

http://www.blendswap.com/blends/view/84748

Materiales: Todos los materiales utilizados en este juego son PBRs gratis de las 2 siguientes páginas:

https://3dtextures.me/

https://freepbr.com/

Sonido: para los sonidos ambiente, disparo y destrucción de objetos se saca de páginas web públicas.

Para el sonido del desierto: https://www.soundsnap.com/tags/western

Hay distintos sonidos y se utilizarán más de la biblioteca gratuita de Adobe: https://offers.adobe.com/en/na/audition/offers/audition_dlc.html

Modelado: para el modelado de objetos en blender se hace uso principal de los video tutoriales del canal de youtube “Enrique Rendon” y diversos vídeos del modelado de otros objetos orientados a los típicos que hubiera en el oeste.

Para los cactus, por ejemplo: https://www.youtube.com/watch?v=a0A0_JdfZSE

ANEXO II: Instalación del juego *WestGun-Therapy*

Para usar el juego es necesaria una lista de ficheros y carpetas que van acompañadas del ejecutable con extensión ".exe" necesario para ejecutar el juego. La figura 52 ilustra la lista de archivos necesarios.

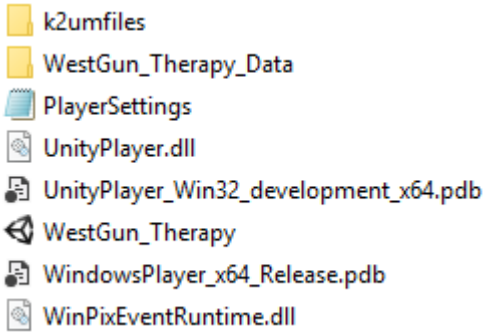
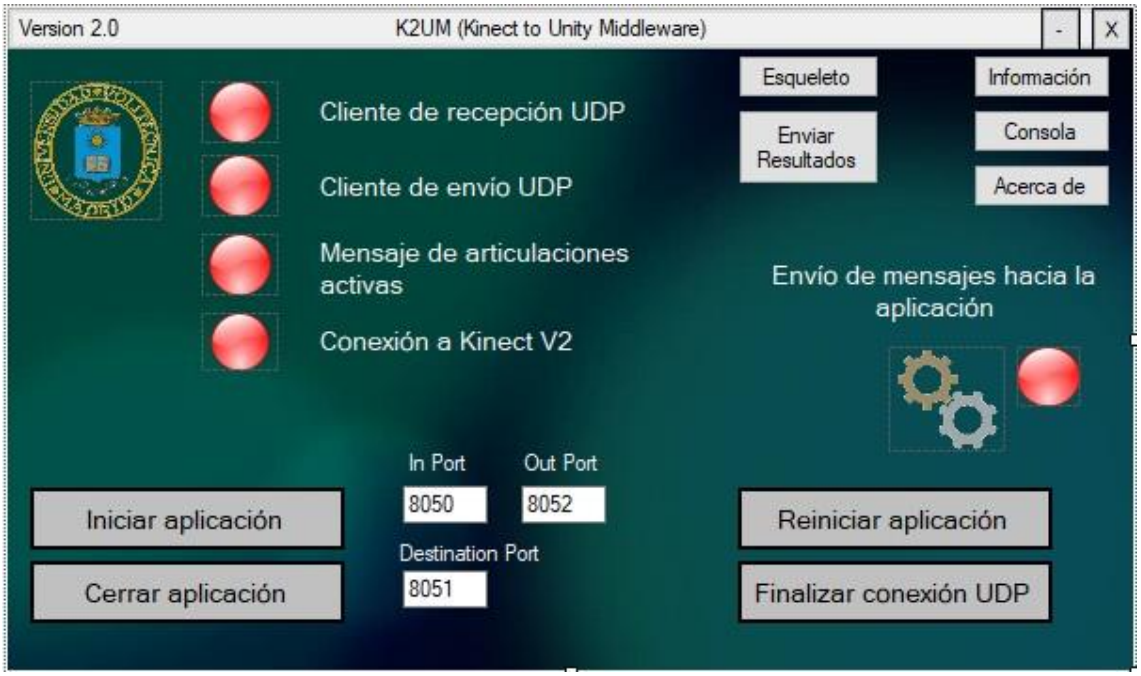


Figura 5.- Archivos necesarios para la instalación del juego

Todos los archivos menos el fichero llamado "*PlayerSettings*" deben estar ubicados en una carpeta localizable por el usuario.

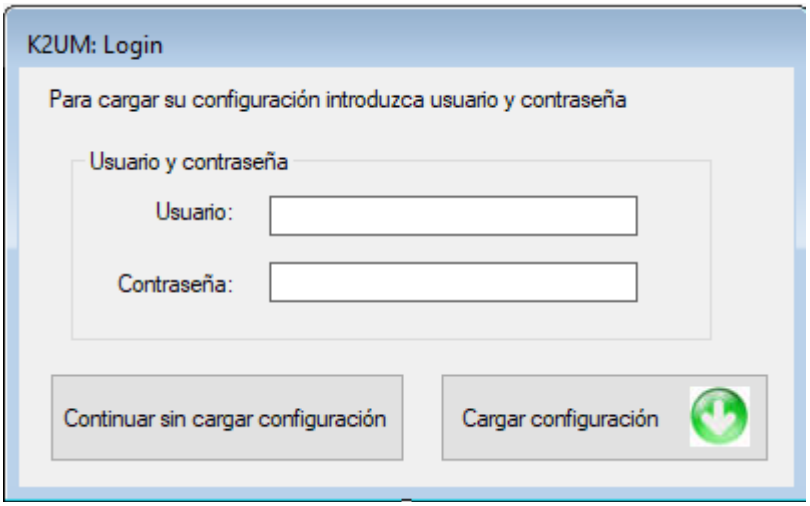
El archivo "*PlayerSettings*" debe estar ubicado en la carpeta "*k2umfiles*" y, dentro de este en otra llamada "users-settings".

ANEXO III: K2UM 2.0 – Daniel Iglesias, abril 2019

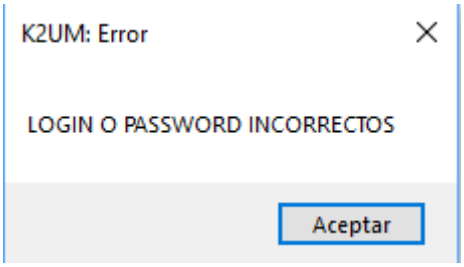


Inicio de la aplicación.

Al arrancar la aplicación se muestra la ventana que pide el usuario y contraseña.



Hay dos opciones: continuar sin descargar ningún fichero de configuración o introducir un nombre de usuario y contraseña. Si este usuario no está registrado en la web o su contraseña no es correcta saltará un aviso.

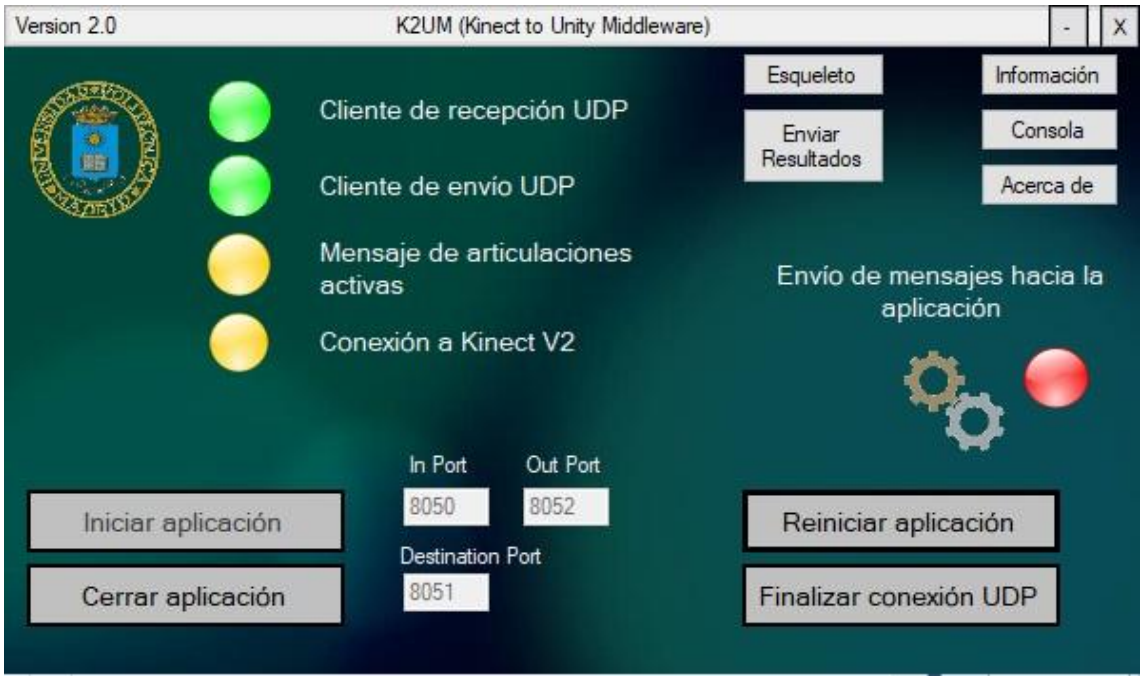


Esta parte del código se desarrolla en la clase **LoginForm.cs** la cual contiene las variables y métodos que se usan para la comunicación con la página web.

Al pulsar el botón *Cargar Configuración* se ejecuta el método **cargarConfiguración()** el cual envía un mensaje de solicitud de datos a la web mediante el protocolo HTTP. Esta parte de del código se ha copiado tal cual del middleware Chiro y se ha adaptado al actual.

La configuración que el middleware recoge de la web se guarda en un fichero de texto de nombre el login del usuario seguido de la extensión .txt. Este fichero se guarda en la carpeta donde esté instalado el programa en la subcarpeta **"\k2umfiles\users-settings**".

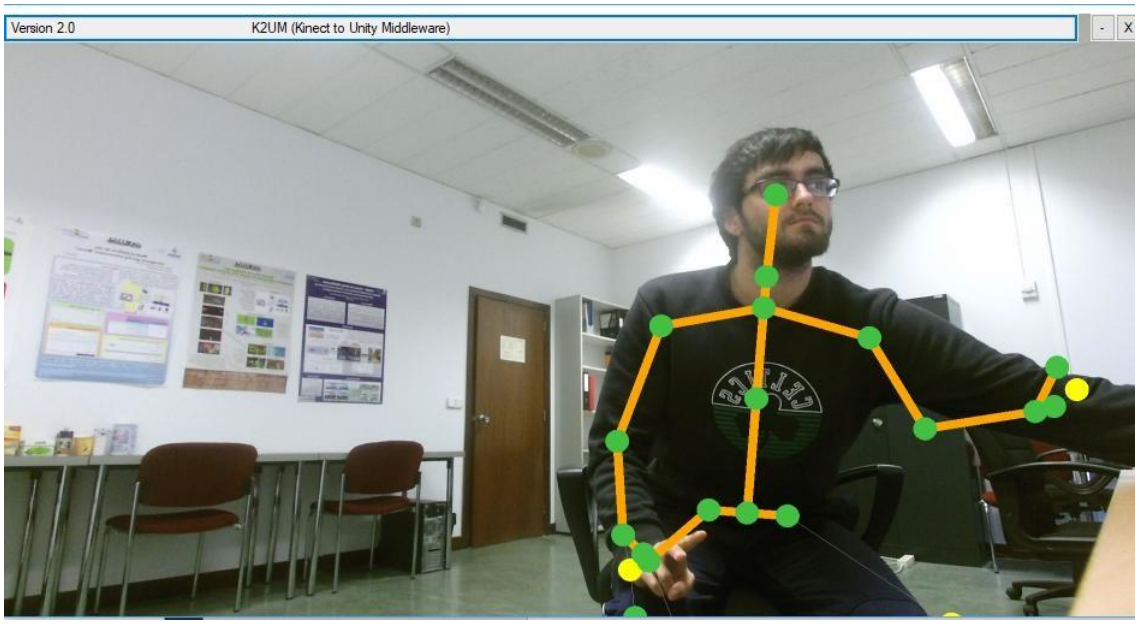
Al pasar el proceso de autenticación se abre la ventana principal gestionada por el formulario diseñado en **Form1.cs**. Al pulsar el botón *Iniciar aplicación* comienza la comunicación entre el middleware y la Kinect.



Ventana Kinect Skeleton

Una vez iniciada la conexión se pueden usar el resto de funciones del middleware como la información de los mensajes enviados o la ventana de depuración para el seguimiento de la aplicación.

Al pulsar el botón *Esqueleto* se abre la ventana independiente que muestra la imagen de la cámara de Kinect y dibuja si detecta algún cuerpo el esqueleto del mismo.

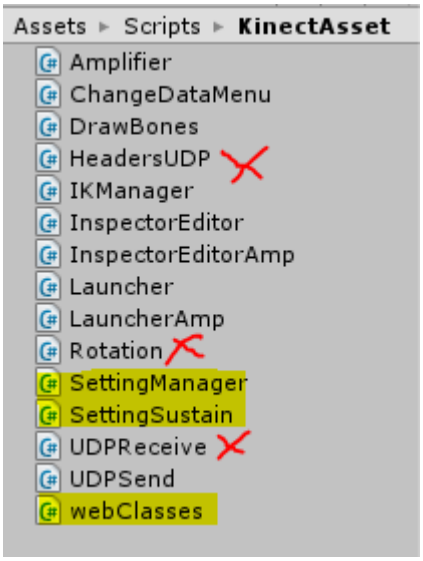


El seguimiento del código utilizado comienza en la clase **Form1.cs.** Al pulsar el botón *Esqueleto* se crea un hilo secundario llamado **skeletonThread** el cual activa la ventana de la cámara. El resto del proceso se realiza en la clase **Form2.cs**.

En ella se toman los datos que manda la Kinect (30 frames por segundo) y se diferencian si son datos de imagen (**ColorFrame)** o datos de los esqueletos (**BodyFrame**). Ambos datos se combinan para formar la imagen resultante. Kinect puede captar hasta 6 esqueletos distintos al mismo tiempo.

Kinect Asset

Una vez iniciada la conexión con Kinect se puede establecer conexión con Unity a través del código incluido en el asset. Al iniciar una escena que contenga el objeto kinectReceiver del asset de Kinect, que puede ser por ejemplo el menú principal, se inicializan todas las variables para la conexión UDP con el middleware y se envía la solicitud de datos de configuración al mismo. Este proceso involucra los scripts **udpSend.cs y settingManager.cs**.



En amarillo, scripts nuevos añadidos. En rojo scripts existentes modificados.

Para que este proceso sea posible se han implementado dos nuevos tipos de mensajes de control, mensaje de solicitud de configuración (*Setting Request* – SR) y mensaje de configuración enviada desde el middleware (*Setting Sent* - SS).

Desde la clase **SettingManager.cs** se solicita el archivo de configuración tras cargar la escena. El middleware recibe la solicitud de configuración (SR), la cual contiene información del juego cuyos datos solicita y tras esto deserializa la información correspondiente al juego solicitado, contenida en el fichero de configuración descargado. La información del juego se serializa de nuevo en formato JSON y se envía a Unity (SS) donde la clase **UDPReceive.cs** se encarga de recibir el mensaje, comprobar qué tipo de mensaje es y colocarlo en la cola de mensajes recibidos. La clase **Rotation.cs**

desencola el mensaje y extrae la cadena de texto en formato JSON. La información del JSON se deserializa en la clase **SettingManager.cs** cuyo contenido es la configuración de todos los ejercicios del juego y los guarda en un diccionario de ejercicios. Este diccionario está definido en la clase **SettingSustain.cs** que se asignará como componente a un gameObject "*settingObject"* que no se destruirá al cambiar de escena y permitirá recoger la configuración para cada ejercicio.

También se ha desarrollado un ejemplo de script para extraer la configuración y al terminar el ejercicio mandar los resultados, como ayuda para los desarrolladores de los juegos. Este script se llama **exerciseManager.cs.**

Cuando el usuario decida puede enviar los resultados de los ejercicios obtenidos hasta el momento usando el botón "Enviar Resultados" del middleware. Los resultados que se van llegando se guardan en un fichero de texto de nombre el login del usuario seguido de la extensión .txt y almacenado en el directorio del middleware dentro de **"\k2umfiles\users-settings**".

El componente **SettingManager.cs** debe estar presente en el objeto *kinectReceiver* en la escena en la que se decida solicitar el archivo de configuración, por ejemplo, el menú del juego.

Todas las clases que estructuran los datos de configuración y resultados que se envían a la web se guardan en el fichero **webClasses.cs**.