E.T.S. de Ingeniería Industrial, Informática y de Telecomunicación

Aplicación en Realidad Virtual para diseño de ropa y objetos enganchados al cuerpo

Trabajo Fin de Grado

Autor : Iñaki Pérez del Notario López

Tutor : Oscar Ardaiz Villanueva

Pamplona, 30/5/2020

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Índice

1. Resumen del Proyecto

Introducción

Soy un estudiante del grado de ingeniería informática que está acabando la carrera. Como este año he estado libre de asignaturas, decidí probar lo de las becas de colaboración. Hable con Oscar, elaboramos la solicitud y al final aparecí en la lista de seleccionados.

El tema de la beca era elaborar una aplicación de realidad virtual utilizando Unity para diseño en un entorno virtual utilizando captación de movimiento.

Cuando empecé no tenía ninguna experiencia con Unity ni realidad virtual, así que el desarrollo del proyecto me ha permitido aprender cómo es un motor de desarrollo en 3D y he podido comprobar de primera mano el potencial que tiene la realidad virtual.

Resumen

Este proyecto consiste en el desarrollo de una aplicación de diseño de ropa y objetos enganchados al cuerpo en VR.

La aplicación está formada por un escenario real y otro virtual:

- En el escenario real se utiliza un sistema de captación de movimiento para capturar la posición y los movimientos de una persona y reflejarlos en un modelo humano en el entorno virtual.
- En el entorno virtual hay un diseñador que lleva puestas unas gafas VR y, utilizando unos mandos, puede dibujar planos en el espacio que luego se unirán para formas mallas. Una vez formadas las mallas. Para que las mallas se comporten como una tela, se les pueden asignar físicas de forma que interactúen con el modelo humano virtual reaccionando a sus movimientos y también se pueden enganchar a distintas partes del cuerpo.

Material Utilizado

Hardware Utilizado : PC, KinectV1, Oculus Rift S, Oculus Quest.

Software Utilizado : Unity, NiMate.

Palabras Clave

realidad virtual, captura de movimiento, diseño, ropa

Indice

2. Estado del arte

En este apartado voy a comentar proyectos de los últimos años que tienen relación con mi aplicación.

SecondSkin: sketch-based construction of layered 3D models [1]

Este proyecto simplifica la forma de diseñar la ropa, en lugar de que el diseñador dibuje la ropa como un modelo en 3D, la aplicación permite diseñar ropa haciendo que dibuje la ropa como si fuese un boceto.

En la primera imagen se ve en azul el objeto que el diseñador está dibujando sobre la pantalla, un plano en 2D. Cuando el objeto se ha dibujado, la aplicación interpreta lo que ha dibujado el diseñador y lo adapta al modelo humado dibujando la pieza de ropa en 3D.

Garment modeling with a depth camera [2]

En este proyecto, en lugar de diseñar la ropa directamente con el ordenador, se viste a una persona real y se le saca fotos. Con las imágenes, se utilizan algoritmos de visión para extraer los componentes de la vestimenta y con esa información se dibujan automáticamente en 3D la partes que la componen.

En la primera imagen se puede ver un ejemplo de una imagen a la que se le identifican los componentes de la vestimenta y en la segunda imagen se puede ver como se dibuja como modelos en 3D

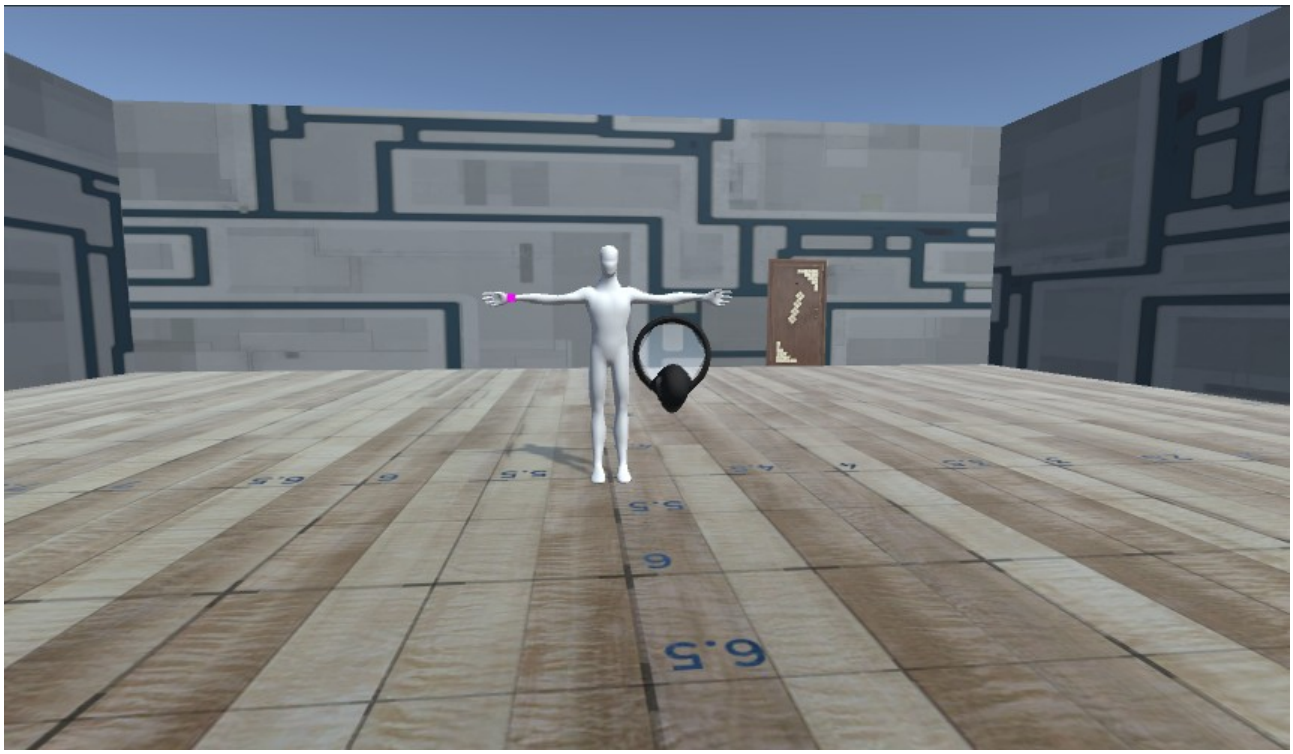

Collar
(Falbala Type)
Body
(Round Neckline,
Basic Type,
Short Length)
Sleeve
(Long Length,
Lantern Cuff,
Hubble-Bubble
Armhole,
Fitted)
(Female Type)
(Small Size)
Skirt
(Horn Outline,
Long Length,
Exist Pleats)

Indice

3. Funcionamiento de la demo

Aquí voy a explicar el funcionamiento de la aplicación para el Usuario.

Cuando se inicia la aplicación, el usuario virtual aparece en una habitación cuadrada en la que puede ver un modelo humano en el centro de la habitación y puede ver los mandos de Oculus que está sujetando, en este caso como no dispongo de las Oculus durante la cuarentena, simulo que hay un mando siguiendo al usuario.



A partir de ahí, el usuario puede interactuar de múltiples formas utilizando teclado y ratón, en ausencia de los mandos.

"w" : Moverse hacia delante.

"a" : Moverse hacia la izquierda.

"s" : Moverse hacia atras.

"d" : Moverse hacia la derecha.

"Espacio" : Saltar.

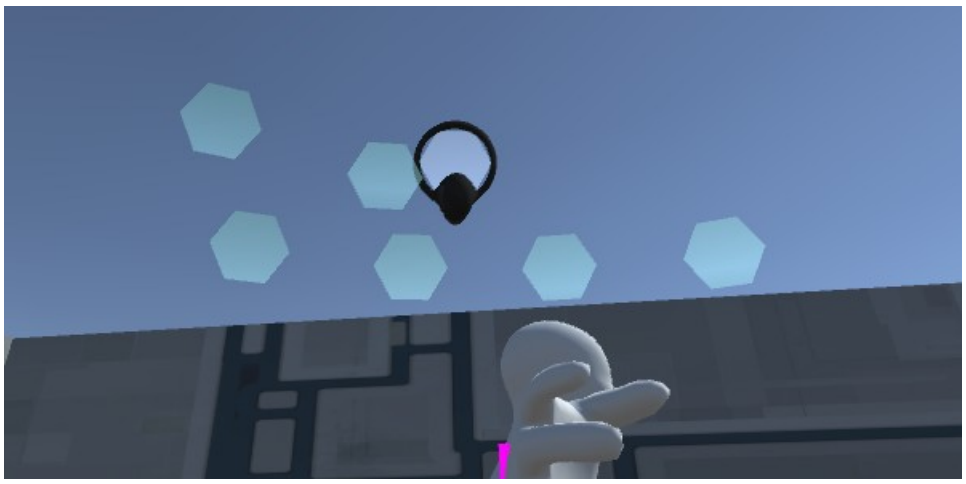
"Shift" : Aumentar velocidad de desplazamiento (mantener pulsado).

"1-9" : Seleccionar un ejemplo predibujado.

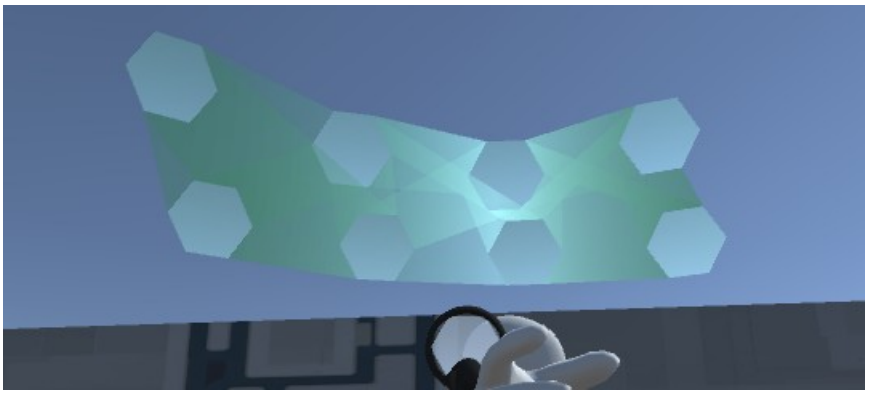
"0" : Seleccionar el modo de dibujar manualmente (Activado por defecto).

"Click izquierdo raton" : Dibujar planos según la posición del mando de Oculus.

Indice



"j" : Unir los planos en el dibujo activo.

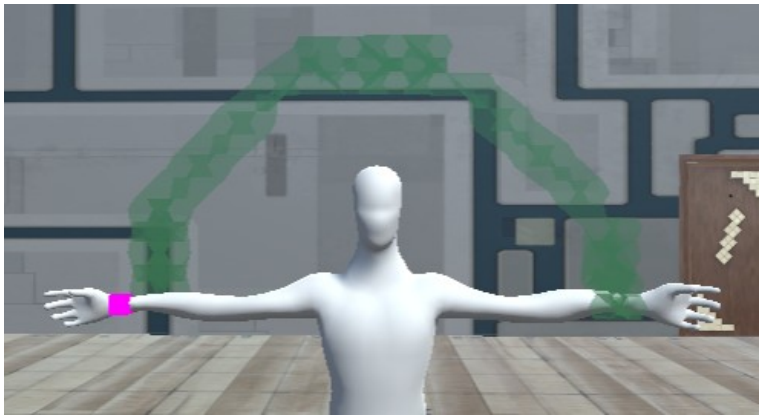
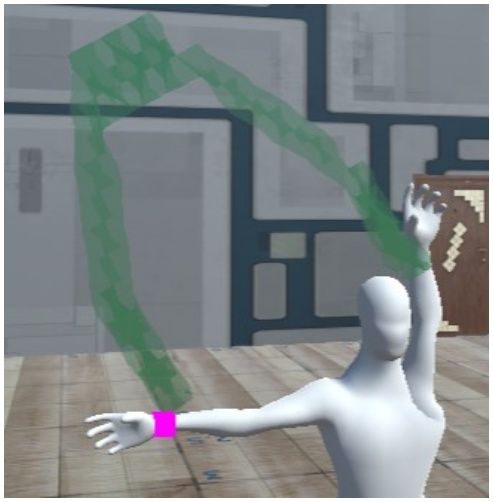


"k" : Pegar y generar cloth en el dibujo activo.

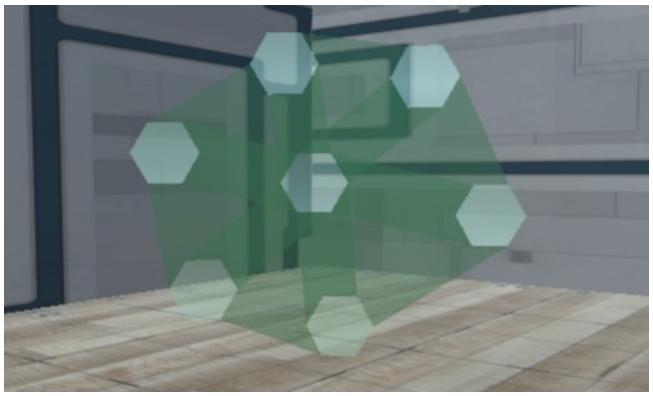
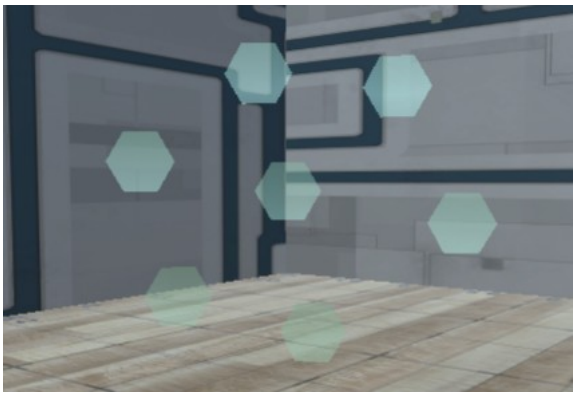
"l" : pegar,generar cloth en el dibujo y enganchar cloth al cuerpo, solo si se ha dibujado manualmente

Indice


"p" : pegar y enganchar objeto solido al cuerpo, solo si se ha dibujado manualmente

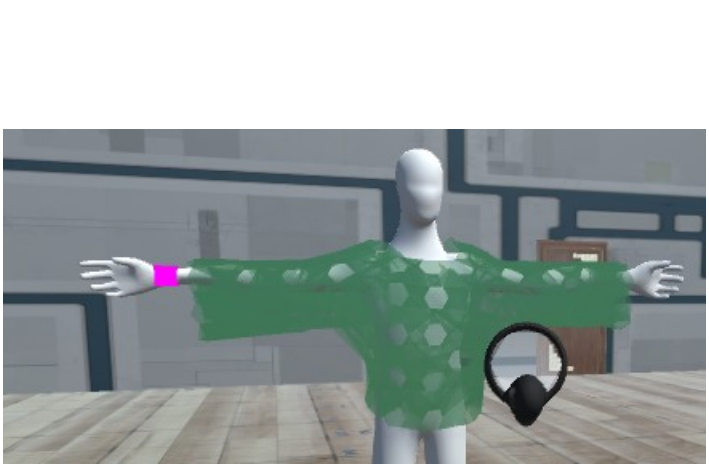



"u" : Deshacer la última unión o tela dibujada, solo si se ha dibujado manualmente.

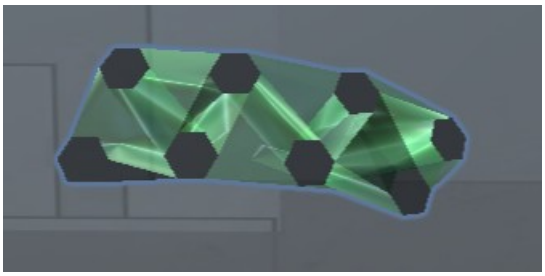
"q" : Activar animación levantar/bajar brazos.

Indice



"e" : Activar animación dar un paso.

"m" : Guardar dibujo.



"c" : Congelar/descongelar modelo humano.

Indice

En el entorno real, la captación de movimiento estará funcionando cuando la aplicación esté en funcionamiento y el sistema de captación de movimiento detecte al usuario.

También existe la posibilidad de que otro usuario se pueda poner otras gafas de realidad virtual y observar la captura de movimiento y ver lo que está pintando el usuario virtual.

Indice

4. Material

Hardware

Kinect

[3]

La Kinect es un accesorio desarrollado por Microsoft para la videoconsola Xbox360 lanzado el 2 de junio de 2009.

Está formada por una cámara RGB, un sensor de profundidad, un micrófono de múltiples matrices y un procesador que se encarga de la captura de movimiento, reconocimiento facial y reconocimiento de voz.

La Kinect se utiliza principalmente para detectar las partes del cuerpo de una persona y formar un esqueleto que luego puede ser utilizado por el software. Es capaz de capturar el esqueleto de varias personas simultaneas, el numero depende de la versión de la Kinect se esté utilizando.

Existen 2 modelos, la KinectV1 y la KinectV2, la cual se lanzo con la videoconsola Xbox One.

Las diferencias importantes entre las 2 son que la KinectV2 tiene mejor cámara, mejor sensor de profundidad, tiene un ángulo de visión mayor y es capaz de detectar los dedos de las manos (la V1 no puede).

En mi proyecto utilizo la Kinect V1 porque es único modelo que hay en el laboratorio, el modelo 2 daría mejores resultados e incluso también se pueden utilizar alternativas de otras marcas.

Indice

Oculus Rift S

[4]

Las Oculus Rift S son unas gafas de realidad virtual desarrolladas por Oculus que se pusieron a la venta el 21 de Mayo de 2019 y son una versión mejorada de las Oculus Rift originales. Las gafas vienen con 2 mandos, los cuales permiten interactuar con las aplicaciones VR y tienen sensores para trackear su ubicación con respecto a las gafas y su orientación.

Estas gafas se tienen que conectar a un ordenador con una tarjeta gráfica potente para poder funcionar.

En el proyecto, estas gafas las utiliza el usuario para poder ver el entorno virtual y moverse por el, y también utilizar los mandos para poder dibujar e interactuar con el mundo virtual.

Oculus Quest

[5]

Las Oculus Quest son unas gafas de realidad virtual desarrolladas por Oculus que se pusieron

a la venta el 21 de Mayo de 2019 (igual que las Rift S). Estas gafas, a diferencia de las Rift S, tienen la capacidad de utilizarse de manera independiente sin estar conectadas a un ordenador, para ello las gafas tienen su propia batería y su propio ordenador interno con el sistema operativo Android. Las gafas también incluyen 2 mandos (los mismos que las Rift S), necesarios para interactuar con la interfaz del sistema operativo y las aplicaciones.

En el proyecto utilizo estas gafas para que un segundo usuario pueda ver la ubicación en el entorno virtual y lo que está dibujando el primer usuario.

Software

Ni mate

[6]

Ni mate es una aplicación desarrollada por Delicode ltd, esta aplicación sirve hacer captura de movimiento a traves de un sensor compatible y luego transmitir la información por mensajes OSC para poder ser utilizada por otras aplicaciones.

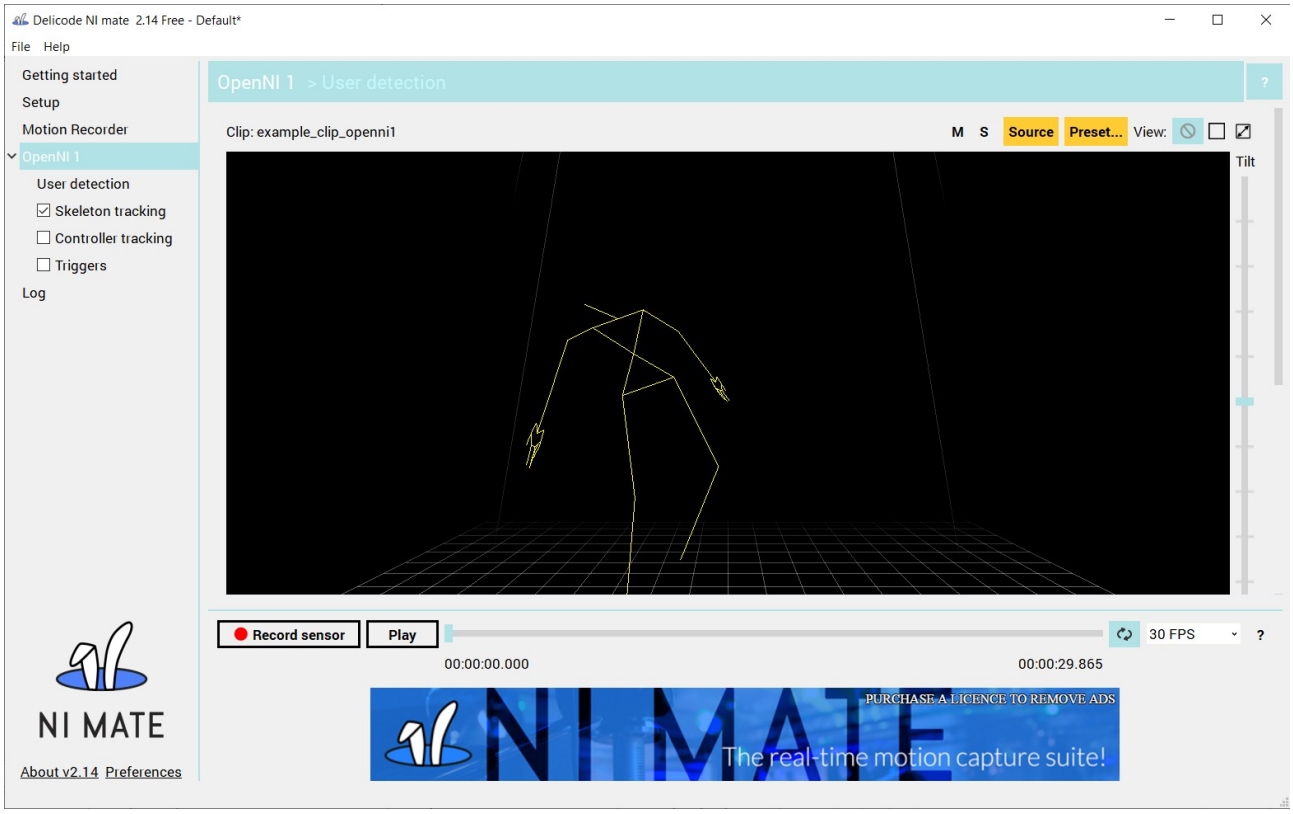
Sensores soportados : KinectV1, KinectV2, Asus Xtion Pro Live, PrimeSense Carmine 1.08, Intel RealSense F200, R200, SR300

Ni mate tiene una versión gratuita y otra de pago, la gratuita solo te permite hacer cosas básicas y la de pago tiene más funcionalidades.

Hay aplicaciones que cuentan con plugins oficiales de Ni mate las cuales reciben los mensajes OSC del programa y los interpretan para ser utilizadas en esa aplicación.

Programas con plugins oficiales : Blender, Cinema 4D y Autodesk Maya.

En mi proyecto utilizo la versión gratuita de esta aplicación para utilizar la Kinect en Unity. Para Unity hace unos años existía un plugin de Ni mate, pero a día de hoy ese plugin está desfasado y no se puede descargar, así que he tenido que interpretar los mensajes por mi cuenta. La versión gratuita está limitada, pero para lo que tengo que hacer es suficiente, el único inconveniente es que hay que introducir los parámetros a mano cada vez que se inicia la aplicación, porque la funcionalidad de guardado no está incluida en la versión gratuita.



Indice

Unity

Unity es un entorno de desarrollo de gráficos en 3D. En su interfaz muestra un espacio en 3D en el que se pueden añadir objetos, los cuales tienen una posición, orientación y tamaño, a esos objetos se les puede añadir propiedades como un modelo 3D, físicas, scripts de C#, ...

Todos los modelos de Unity están formados por triángulos, lo que permite que si se necesita algún modelo distinto a los que vienen por defecto, se puede generar uniendo triángulos.

Unity para crear y editar los scripts, utiliza el editor de texto Visual Studio, el cual permite utilizar las funciones de Unity de manera cómoda y efectiva.

También cuenta con una tienda donde se pueden comprar y descargar cosas como librerías, sonidos. Objetos, texturas y animaciones entre muchos otros.

En mi proyecto utilizo la versión de Unity 2019.2.15.f1 porque fue la primera que instalé. Mi aplicación en sí, es un proyecto de Unity en el que he creado un escenario 3D y he ido añadiendo funcionalidades mediante scripts en C#, algunos creados por mi y otros que están incluidos en librerías, y utilizando herramientas y propiedades que están incluidas por defecto en Unity. He utilizado scripts para crear figuras que no existe en únity como por ejemplo un plano hexagonal. De la AssetStore solo he utilizado Assets gratuitos.

Librerías de Unity

extOSC

[7]

La librería extOSC es una librería que implementa el protocolo de comunicación OSC (Open Sound Control). Este protocolo utiliza un transmisor para enviar mensajes por UDP a una dirección IP, utilizando un string como identificador del mensaje y un receptor que se encarga de recibir los mensajes dirigidas a esa IP con el mismo string identificador. Los mensajes pueden contener arrays con datos de distintos tipos o incluso pueden ir vacíos.

La librería es gratuita y viene con varios ejemplos en los que se prueban distintos tipos de datos y se muestran algunos scripts personalizados, por desgracia, los videotutoriales oficiales no son muy buenos y a no ser que se vean los videos a máxima resolución, es difícil entender algo.

La librería cuenta con varios scripts para recibir datos de tipos básicos, pero para combinaciones de datos más complejos, en mi caso un vector de 6 floats, hay que programarlo mirando como funciona internamente la librería.

En mi proyecto es una librería importante, porque gracias a esta librería he podido utilizar la Kinect en Unity y también he podido comunicar las Oculus Quest con el ordenador a través de una LAN inalámbrica.

Oculus

[8]

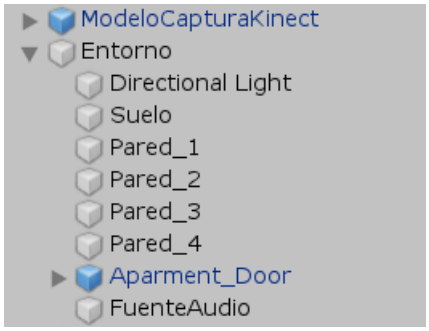
La librería de Oculus el la librería que permite utilizar las gafas de realidad virtual de Oculus y sus mandos en Unity. La librería incluye varios prefabs con un objeto de jugador, objetos de los controles, scripts con parámetros para ajustar ciertos aspectos del jugador y ejemplos sobre las funcionalidades de la librería.

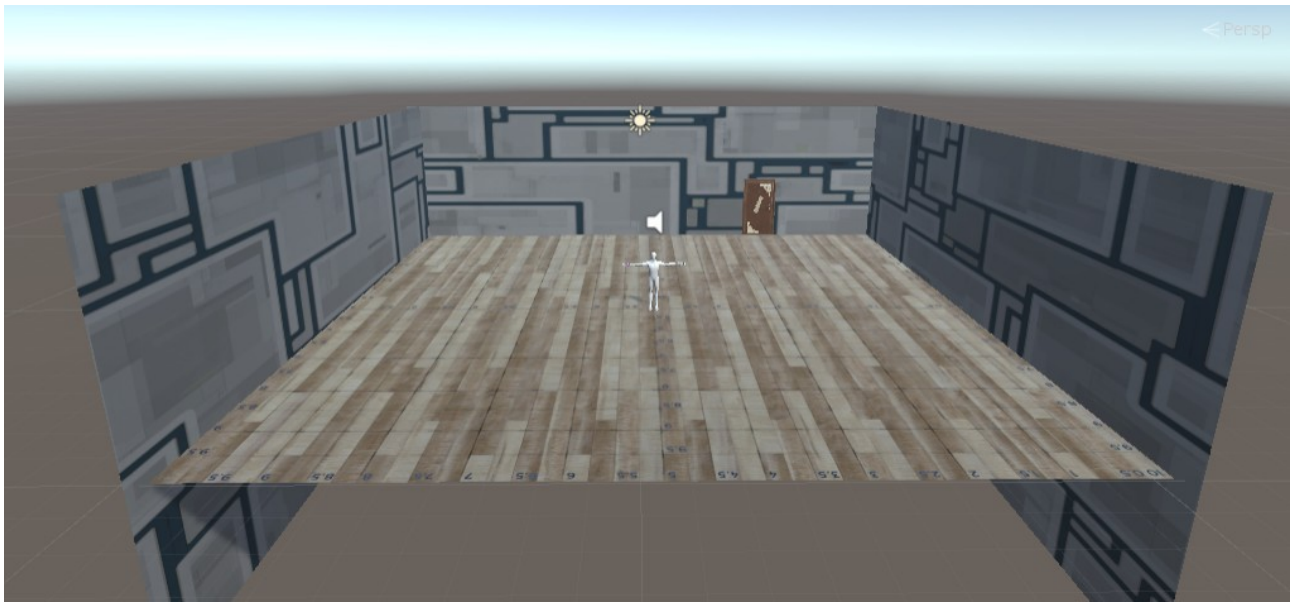
5. Implementación en Unity

Scene

Aquí voy a explicar como tengo hecho a scene principal de mi proyecto.

En la scene principal como entorno físico tengo 5 planos formando un suelo y 4 paredes y una puerta decorativa. Una fuente de luz para poder ver el entorno con claridad, una fuente de Audio para reproducir sonidos mientras se dibuja y un modelo Humano en el centro de la habitación, el cual se utilizará para reflejar la captura de movimiento.



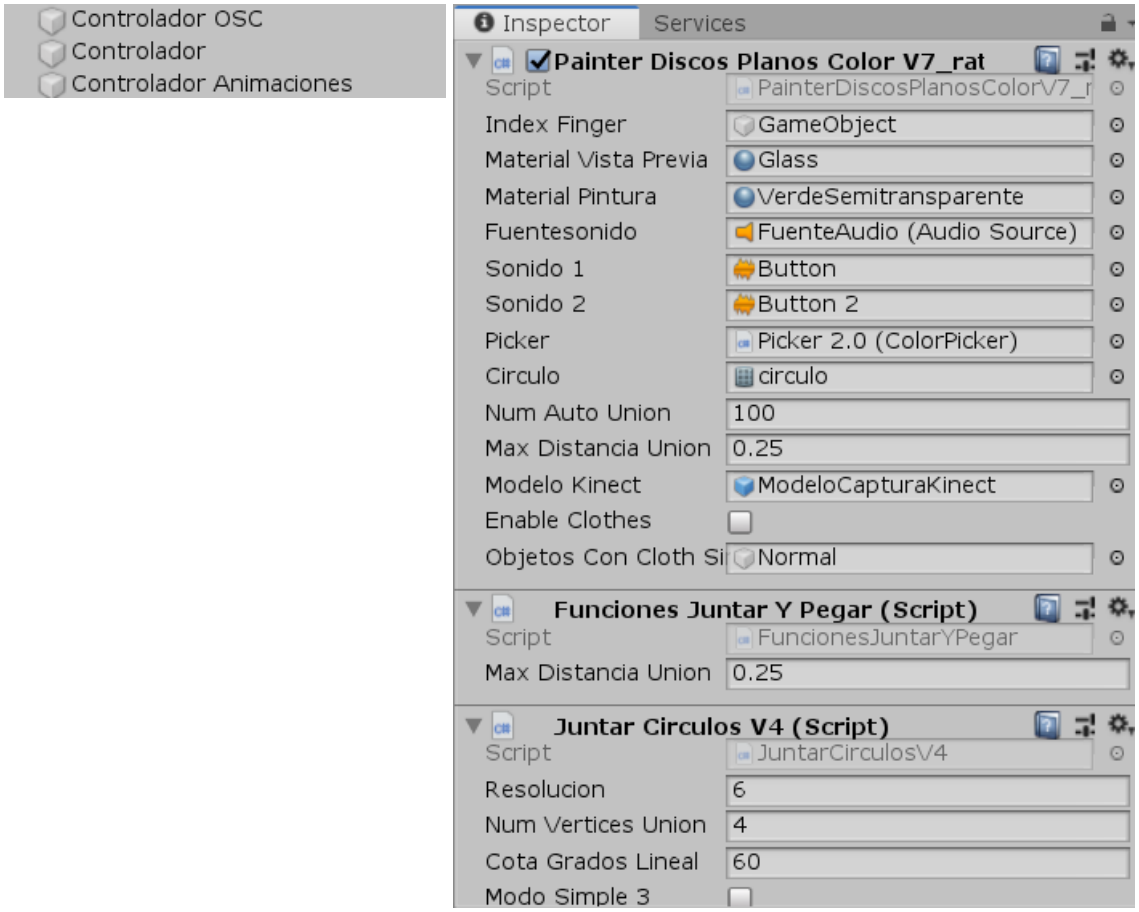


Como objetos jugador he puesto 2, uno llamado "RigidBodyFPSController" para desplazarse por el entorno e interactuar con la aplicación utilizando solo el teclado y el ratón sin depender de las gafas. Y otro llamado "OVRPlayerController" que es para cuando se utilicen las gafas VR. El primero lo he importado de la librería Standard Assets de la AssetStore, el cual es una librería oficial de Unity que tiene diversas utilidades y entre ellas controladores de jugador. El segundo lo he creado juntando los prefabs de los mandos y el prefab del controlador del jugador de la librería de Oculus.

Para la implementación de las funcionalidades, he creado los Objetos "Controlador",

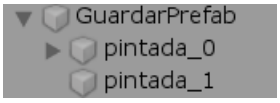
Indice

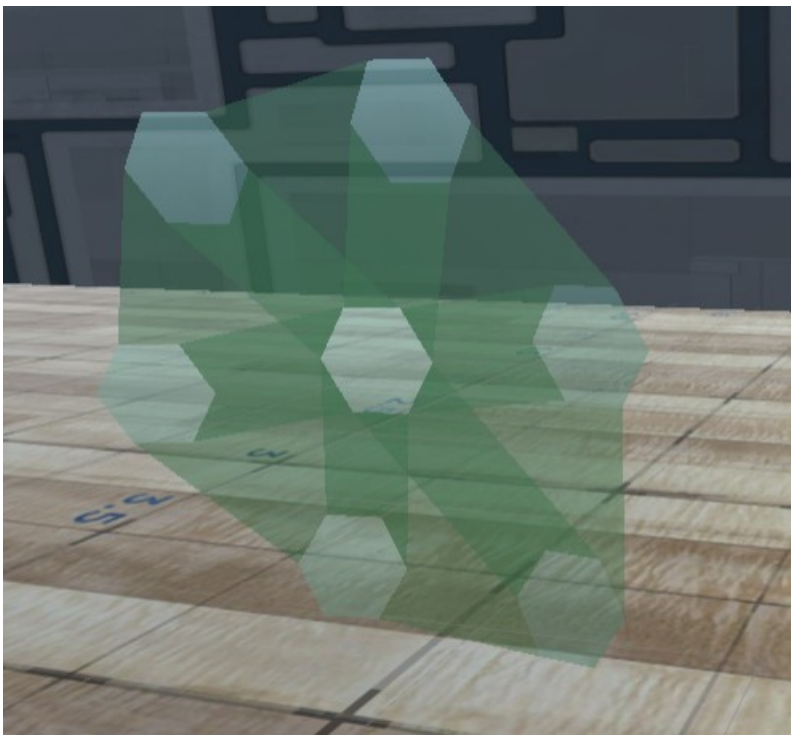
"ControladorOSC" y "ControladorAnimaciones". En "Controlador" es donde están casi todas las funcionalidades de la aplicación en forma de scripts, que se han añadido como componentes del Objeto. El "ControladorOSC", como dice el nombre, es un objeto que tiene los scripts que controlan las direcciones IP y los puertos de la emisión/recepción de mensajes OSC, este controlador viene incluido en la librería extOSC. Y por último el "ControladorAnimaciones" donde almaceno funciones que permiten ejecutar una animación predefinida para cuando no se esté utilizando la kinect.



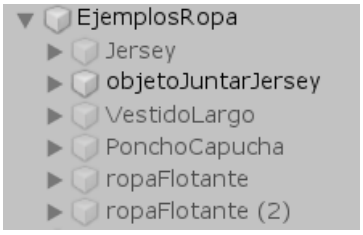
El funcionamiento de los scripts los explicaré más adelante.

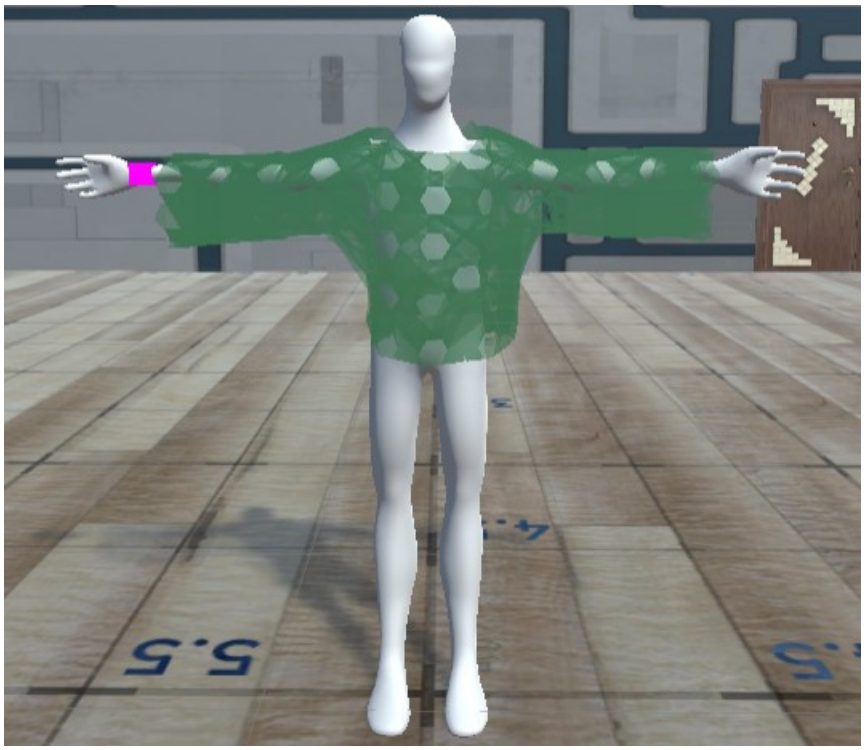
Entre los objetos que quedan, "GuardarPrefab" es un objeto sobre el que se guardan los dibujos que el usuario va haciendo cuando la aplicación está en funcionamiento, de forma que forma que luego se pueda almacenar todo lo que el usuario haya realizado en un prefab de manera más sencilla.





En el objeto "EjemplosRopa" se almacenan algunas prendas de ropa de ejemplo, esos objetos están desactivados para que no sean visibles hasta que se activen cuando la aplicación esté en funcionamiento.





Indice

Scripts

Aquí voy a comentar los scripts que he programado y cual función tienen.

PainterDiscosPlanosColor

El Script "PainterDiscosPlanosColor" es el script más extenso de todos los que he programado, su código implementa la funcionalidad de pintar. Este script ha pasado por muchos cambios a lo largo del desarrollo, la base de este script es de un proyecto que me ha compartido Oscar en el que de dibujaban cubos con los mandos de las oculus.

Parámetros del Script :

Index Finger : El objeto donde se va a colocar Vista Previa para pintar

Material Vista Previa : El material de la Vista previa

Material Pintura : El material con el que se va a pintar

Fuentesonido : La fuente de audio donde se van a emitir los sonidos

Sonido 1 : El sonido que se reproduce al colocar planos

Sonido 2 : El sonido que se reproduce al juntar los planos

Circulo : El Mesh del plano que se va a utilizar para la vista previa y para pintar

Num Auto Union : El número máximo de planos que pueden pintar para unir, una vez llegado al máximo se unen automáticamente.

Max Distancia Union : Distancia máxima a la que se pueden unir 2 planos.

Modelo Kinect : Objeto del modelo humano que se usa para sacar las colisiones.

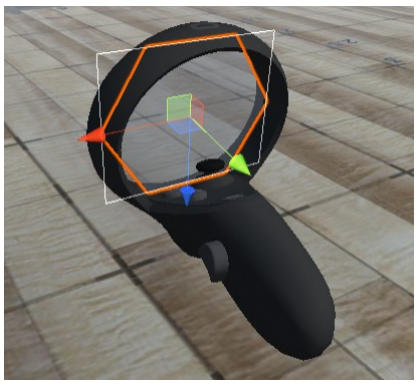
Objetos Con Cloth Sin Mesh : Aquí va un objeto con Cloths ya creados los cuales se usarán en las telas creadas.

distanciaEnganche : Esta variable indica la distancia máxima a la que se pueden enganchar los objetos dibujados al cuerpo

Para adaptarlo a mi proyecto lo primero que hice fue probarlo con mi versión de Unity, entender como funciona el código y corregir algunos errores que surgieron por el aumento de versión.

El funcionamiento de este script consiste en 2 funciones start() y update() y otra función llamada trazar() que es la que dibuja los planos.

La función start() se ejecuta una vez cuando se inicia el programa y se utiliza para inicializar las variables del script y crear el objeto vista previa. El objeto vista previa es un objeto que se coloca sobre el objeto del parámetro "indexFinger" como hijo de ese objeto, está pensado para que ese objeto sea el mando derecho de las oculus, de forma que cuando se mueve el mando la vista previa también se mueve con el, y cuando se pulsa un botón, se dibuja un plano en la posición donde está la vista previa en ese momento.



La función update() se ejecuta una vez cada fotograma que transcurre durante la ejecución del proyecto. La estructura de esta función son varios ifs separados en los que se comprueba si se ha pulsado uno de los botones indicados de forma que si se ha pulsado un determinado botón se ejecuta una funcionalidad.

```
void update()
{
    /*si está pulsado click iquierdo del ratón y no se ha alcanzado
    el límite de planos*/
    if (Input.GetMouseButton(0) && numCirculos < numAutoUnion)
    {
        //Pintamos el plano si la posición del mando está lo suficientemente alejada del plano anterior
        //texto.text = "Posicion: " + position;
        //valorPos.text = "LP: " + lp;

        position = vistaPrevia.transform.position;
        rotation = vistaPrevia.transform.rotation;

        if (!lp.Equals(Vector3.zero))
        {
            distance = (position - lp).magnitude;
            if (distance > scale * 20 * 2)
            {
                trazar();
            }
        }
        else
        {
            trazar();
        }
    }
```

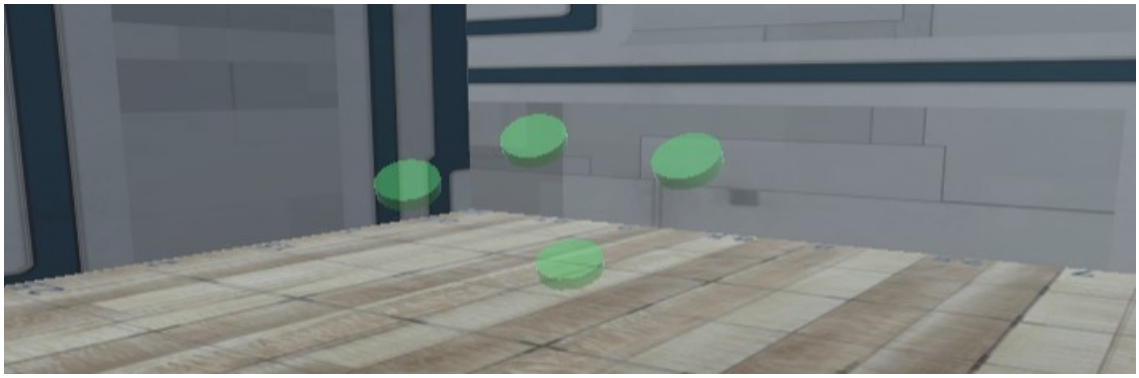
En este caso si el botón izquierdo del ratón está pulsado, calcula la distancia con respecto al anterior plano dibujado y si esa distancia es mayor que una escala, entonces llama a la función trazar para dibujar un plano en la posición y rotación del objeto vista previa.

La función trazar() es una función que crea un objeto plano y lo coloca sobre el espacio con la misma posición y rotación que la vista previa tiene en ese momento, también reproduce un sonido si se ha colocado en los parámetros de entrada la fuente de sonido y el sonido1.

Indice

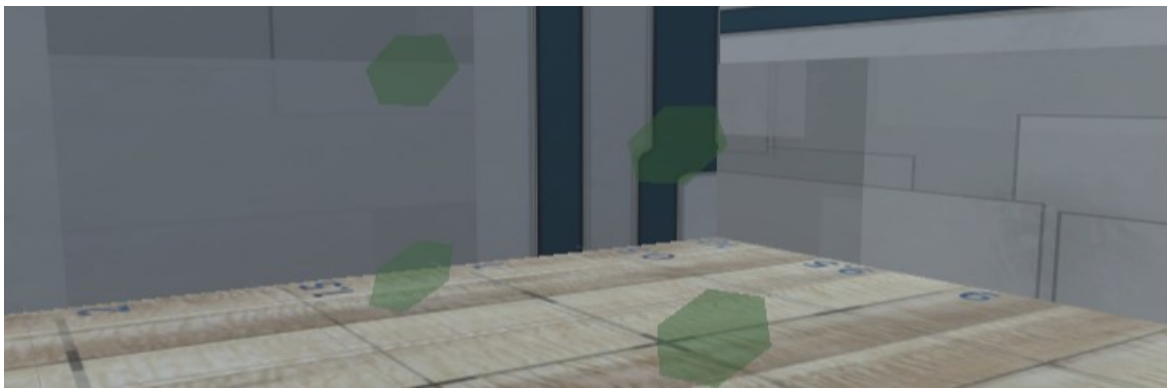
```
/*****************************************************************************************************
 *  Función trazar
 *  Función que permite ir creando el trazo del dibujo en base a la instanciacion de los planos en función
 *  de la posición del mando.
 *****************************************************************************************************/
private void trazar()
{
    // Creamos un nuevo plano
    GameObject plano = new GameObject();
    plano.AddComponent<MeshFilter>().mesh = circulo;
    plano.AddComponent<MeshRenderer>();
    Destroy(plano.GetComponent<CapsuleCollider>());
    plano.transform.localScale = new Vector3(scale, 1, scale);
    plano.transform.SetParent(currentObject.transform);
    plano.transform.position = position;
    plano.transform.rotation = rotation;
    plano.GetComponent<Renderer>().material = materialPintura;
    plano.tag = "Pintura";
    plano.transform.name = "circulo_" + numCirculos;

    //esta linea es para anadir un script que permite almacenar bien los objetos con la funcionalidad de guardado
    plano.AddComponent<TheTide.utils.SerializeMesh>();

    objectList.Add(plano);
    lastPlane = plano;
    lp = new Vector3(position.x, position.y, position.z);
    numCirculos++;

    //reproduce el sonido 1
    if (Fuentesonido != null && sonido1 != null)
    {
        Fuentesonido.PlayOneShot(sonido1, 1);
    }
}
```

En el script original se utilizaban cubos, en nuestro caso al principio pusimos cilindros con poca altura en lugar de cubos.

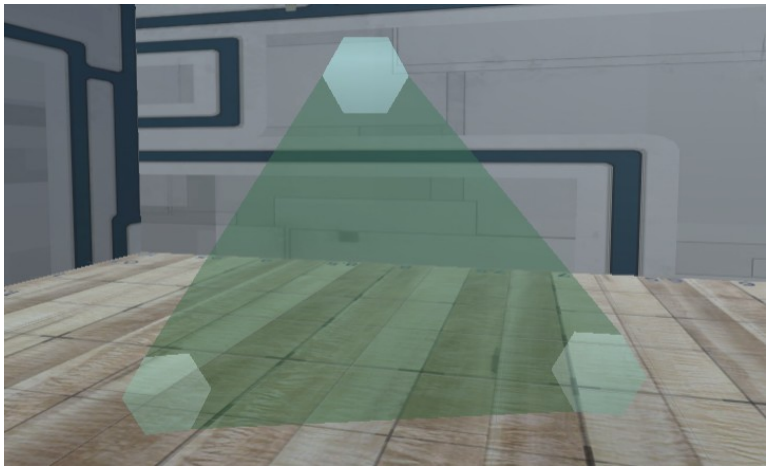


Mas adelante decidimos crear nuestra propia figura, con el script "crear_plano_circulo" creamos un plano circular y lo guardamos como un mesh en la carpeta "Meshes" del Proyecto.

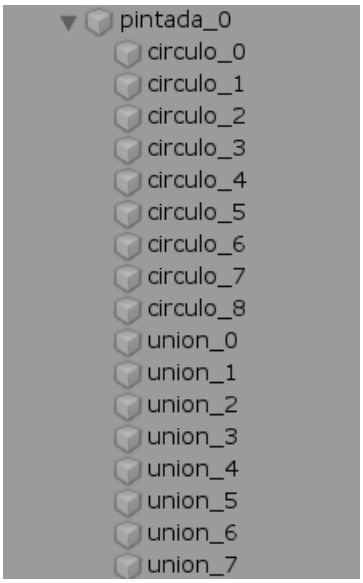
Indice



Después de decidir el objeto con el que se va a pintar, nos interesa que cuando el usuario dibuje varios objetos en el espacio, estos se puedan unir en grupos de 3 utilizando triángulos, para ello he creado un script a parte llamado "JuntarCirculos" donde implemento funciones que realizan la unión de 3 planos, para luego llamar a esas funciones en el script de pintar.



Ya tenemos la funcionalidad de pintar y unir 3 planos, ahora falta implementar una función que dados unos planos en el espacio los una formando grupos de 3 en 3. Esta función está implementada en el script "FuncionesUnirJuntarYPegar", con esa función el resultado queda así.



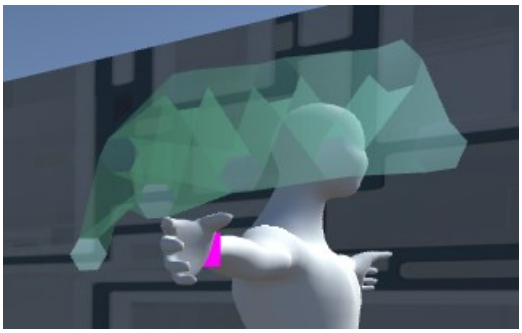
Indice

En la función update() el código para juntar planos queda así

```
/*si se pulsa la tecla j del teclado tras haber dibujado 3 o más planos
 los junta*/
if (Input.GetKeyDown("j") && numCirculos > 2)
{
    GetComponent<FuncionesJuntarYPegar>().JuntarPlanos3(objectList, currentObject, materialPintura);
    numCirculos = 0;

    if (Fuentesonido != null && sonido2 != null)
    {
        Fuentesonido.PlayOneShot(sonido2, 1);
    }

    //si esta activado el modo simple, borra los planos de referencia
    if (GetComponent<JuntarCirculosV4>().esSimple())
    {
        foreach (GameObject element in objectList)
        {
            element.SetActive(false);
        }
    }

    objectList = new List<GameObject>();
}
```

Después de implementar esto, estuvimos mirando una propiedad de Unity llamada Cloth, una propiedad de físicas para hacer que un plano se comporte como una tela. Para que la propiedad funcione bien con lo que dibuja este script, hay que tratar el mesh para que todos los planos y uniones estén unidos y funcionando como una unidad. Para esto he creado dos funciones en el script "FuncionesUnirJuntarYPegar".

Una vez implementadas estas 2 funciones y ajustados los parámetros de la propiedad cloth, el objeto ya se comporta como una tela.




En la función update() el código para generar telas queda así.

```
//si se pulsa la tecla k o l despues de haber juntado algo, se le añade la propiedad cloth
//si se ha pulsado la letra l, el cloth se enganchará a las partes del cuerpo más cercanas
if ((Input.GetKeyDown("l") || Input.GetKeyDown("k")) && lastObject!=null && juntado)
{
    juntado = false;
    GetComponent<FuncionesJuntarYPegar>().juntarMeshHijos(lastObject, materialPintura);
    GetComponent<FuncionesJuntarYPegar>().pegarMeshObjeto(lastObject);

    GameObject ObjetoFinal = lastObject;

    //si hay algun cloth precreado se asigna el mesh del objeto
    if (objetosConClothSinMesh.transform.childCount > 0)
    {
        ObjetoFinal = objetosConClothSinMesh.transform.GetChild(0).gameObject;
        ObjetoFinal.transform.SetParent(GameObject.Find("GuardarPrefab").transform);
        ObjetoFinal.transform.position = lastObject.transform.position;
        ObjetoFinal.transform.rotation = ObjetoFinal.transform.rotation;

        ObjetoFinal.AddComponent<MeshFilter>().mesh = lastObject.GetComponent<MeshFilter>().mesh;
        ObjetoFinal.GetComponent<SkinnedMeshRenderer>().material = materialPintura;
        ObjetoFinal.GetComponent<SkinnedMeshRenderer>().sharedMesh = lastObject.GetComponent<MeshFilter>().mesh;

        lastObject.SetActive(false);

        ObjetoFinal.transform.name = "pintada_" + (objectStack.Count - 1) + "_cloth";
    }
    else
    {
        ObjetoFinal.AddComponent<Cloth>();
        ObjetoFinal.GetComponent<Cloth>().friction = 1;
        ObjetoFinal.GetComponent<Cloth>().friction = 0.6f;
        ObjetoFinal.GetComponent<Cloth>().capsuleColliders = capsulasColisionMalla;
    }
```

Después de crear la tela, si se ha creado pulsado la tecla "l" del teclado en lugar de la "k", la tela se enganchará, si es posible, a 1 o 2 partes del modelo, para ello las funciones "engancharCloth" y "buscar2padres" del script "FuncionesJuntarYPegar" se encargan las partes del cloth cercanas a una parte del cuerpo y asignándole al objeto el script "Emparentar2" como un componente del objeto, para que se muevan las partes enganchadas a la vez que el modelo en caso de que se enganche con 2 padres.

```
    //si se ha pulsado la  tecla l se engancha el cloth
    if (Input.GetKeyDown("l"))
    {
        GetComponent<FuncionesJuntarYPegar>().engancharCloth(ObjetoFinal, modeloKinect, diatanciaEnganche);
        (GameObject padre1, GameObject padre2) = GetComponent<FuncionesJuntarYPegar>().
            buscar2padres(modeloKinect, ObjetoFinal, diatanciaEnganche);
        if (padre1 != null && padre2 != null)
        {
            ObjetoFinal.AddComponent<Emparentar.Emparentar2>();
            ObjetoFinal.GetComponent<Emparentar.Emparentar2>().objeto = ObjetoFinal;
            ObjetoFinal.GetComponent<Emparentar.Emparentar2>().padre1 = padre1;
            ObjetoFinal.GetComponent<Emparentar.Emparentar2>().padre2 = padre2;
        }
        else if (padre1 != null && padre2 == null)
        {
            ObjetoFinal.transform.SetParent(padre1.transform);
        }
    }
}
```

Indice

También se pueden enganchar los objetos sin la tela, cuando se pulsa la tecla "p".

```
//si se pulsa la tecla p, se juntan los objetos y se enganchan a los padres cercanos
if (Input.GetKeyDown("p") && lastObject != null && juntado)
{
    juntado = false;
    GetComponent<FuncionesJuntarYPegar>().juntarMeshHijos(lastObject, materialPintura);
    //GetComponent<FuncionesJuntarYPegar>().pegarMeshObjeto(lastObject);
    (GameObject padre1, GameObject padre2) = GetComponent<FuncionesJuntarYPegar>().
        buscar2padres(modeloKinect, lastObject, diatanciaEnganche);
    if (padre1 != null && padre2 != null)
    {
        lastObject.AddComponent<Emparentar.Emparentar2>();
        lastObject.GetComponent<Emparentar.Emparentar2>().objeto = lastObject;
        lastObject.GetComponent<Emparentar.Emparentar2>().padre1 = padre1;
        lastObject.GetComponent<Emparentar.Emparentar2>().padre2 = padre2;
    }
    else if (padre1 != null && padre2 == null)
    {
        lastObject.transform.SetParent(padre1.transform);
    }

}
```

Una vez que se ha creado una unión o tela, esta se puede deshacer pulsando el botón "u" para corregir algún fallo de diseño. Para implementarlo, hay que tener en cuenta como se han generado los cloths para deshacer el proceso en caso de que tenga cloth y en caso de que se haya emparentado con 1 o 2 partes del cuerpo.

Cuando se unen los planos, pero no se ha generado ni emparentado nada, con eliminar las uniones que componen el dibujo es suficiente. Si se ha generado un cloth, el cloth se puede haber generado creando un objeto a parte y ocultando el original o generándolo en el propio objeto original. Para el primer caso, hay que eliminar el objeto generado y para el segundo caso, hay que eliminar el cloth y el mesh del propio objeto. En el caso de que se haya emparentado con 1 o 2 padres, hay que

eliminar el script "Emparentar2" del propio objeto en caso de que lo tenga y reemparentarlo con el objeto "GuardarPrefab", si estaba emparentado con una parte del cuerpo.

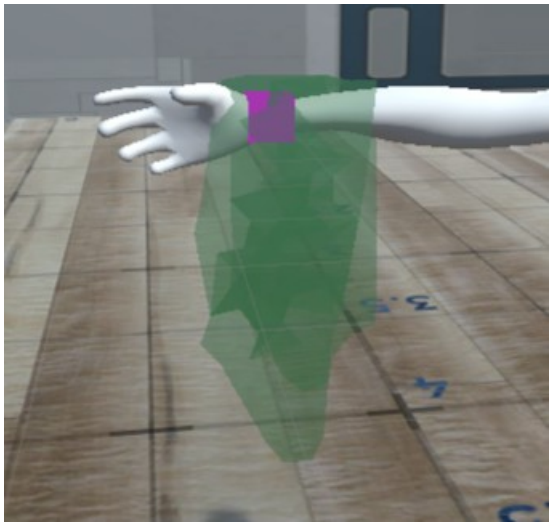
```
/*si se pulsa la tecla u y ya se ha juntado un dibujo
 se deshacen las uniones para poder arreglar el dibujo*/
if (Input.GetKeyDown("u") && lastObject != null)
{
    //destruyo el nuevo objeto creado despues de juntar
    GameObject.Destroy(currentObject);
    objectStack.Pop();

    /*si ya se habia generado un cloth del objeto,
     este se destruye y se reactivan los hijos*/
    if (!juntado)
    {
        GameObject aux = objectStack.Peek();
        aux.SetActive(true);
        foreach (Transform child in aux.transform)
        {
            child.gameObject.SetActive(true);
        }

        /*Se eliminan los componentes del cloth y
         *el mesh de la union */
        if(aux.GetComponent<Cloth>() != null)
        {
            GameObject.Destroy(aux.GetComponent<Cloth>());
            GameObject.Destroy(aux.GetComponent<SkinnedMeshRenderer>());
        }
```

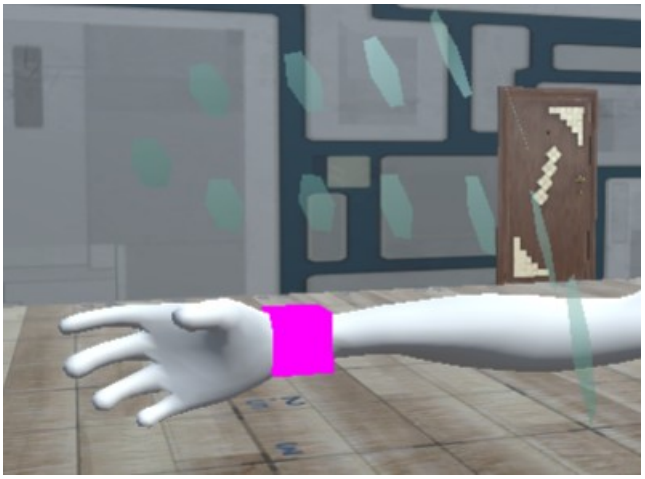
```
        /*Si el objeto estba emparentado, se desemparenta*/
        if (aux.GetComponent<Emparentar.Emparentar2>() != null)
        {
            GameObject.Destroy(aux.GetComponent<Emparentar.Emparentar2>());
        }
        aux.GetComponent<MeshFilter>().mesh = new Mesh();
        aux.transform.SetParent(GameObject.Find("GuardarPrefab").transform);

        /*Si el cloth es uno de los precreados, se elimina su objeto*/
        foreach (Transform child in GameObject.Find("GuardarPrefab").transform)
        {
            if (child.name.Equals(aux.name + "_cloth"))
            {
                GameObject.Destroy(child.gameObject);
                break;
            }
        }
    }
```

Indice

```
    objectList = lastObjectList;
    currentObject = lastObject;

    /*Se eliminan todos los objetos union de los hijos*/
    foreach (Transform child in currentObject.transform)
    {
        if (child.tag == "Union")
        {
            GameObject.Destroy(child.gameObject);
        }
    }

    lastObject = null;
    lastObjectList = null;
    numCirculos = objectList.Count;

}
```

Con este código se pueden deshacer objetos ya dibujados para devolverlos al estado en el que estaban antes de dibujarlos. En las imágenes de abajo se muestra un ejemplo de un trozo de tela en la izquierda y al pulsar el botón u, vuelve a su estado antes de juntar en la derecha.





FuncionesUnirJuntarYPegar

En este script están las funciones que sirven para unir, juntar y pegar planos dibujados dibujados con el script "PainterDiscosPlanosColor" y otras funciones para enganchar objetos.

Parámetros del Script:

MaxDistanciaUnion : Es un valor numérico en el que se indica la distancia máxima a la que se pueden unir 2 planos.

La función "JuntarPlanos3" es una función en la que dados n planos ya dibujados con n>=3, los une de forma que se formen los triángulos más pequeños posible, que queden el menor número de espacios en blanco y evitar que se dibujen triángulos innecesarios. Para abordar el problema, la estrategia que hemos pensado es empezar por el plano con la distancia media a los demás planos más baja, o sea, empezamos a unir por el plano más hacia el centro.

Una vez hemos seleccionado un plano, seleccionamos como segundo plano el plano más cercano a el.

En base a esos 2 buscamos un tercer plano en el que la suma de las distancias entre los 3 planos sea la menor posible y los unimos.

Indice

En el script uno de los parámetros de entrada es 'MaxDistanciaUnion' en el cual se indica la distancia máxima a la que se pueden unir 2 planos, si se seleccionan planos que sobrepasan esa distancia se descartan y no se unen. Una vez se haya alcanzado un plano demasiado alejado entonces se selecciona el siguiente plano con la distancia media más pequeña. Antes de unir 3 planos se comprueba que no exista ya esa unión.

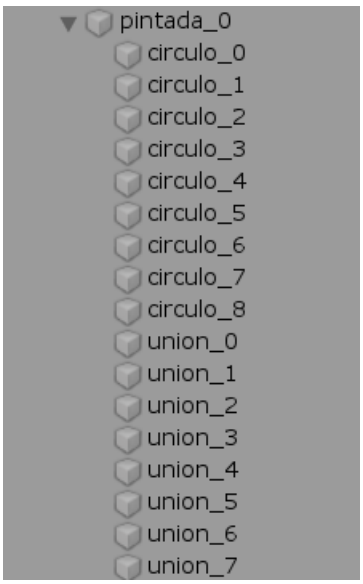
Resumen del código de la función:

```
public List<GameObject> JuntarPlanos3(List<GameObject> jplanos, GameObject currentObject, Material materialPintura)
{
    //por cada plano calculo la distancia media a los otros planos y los ordeno de menor a mayor distancia

    for (/*selecciono el primer planos por orden de distancia media (menor a mayor)*/)
    {
        //obtener_N_planos_mas_cercanos
        for (/*selecciono el segundo plano por orden de distancia (menor a mayor) mientras no sobrepase la distancia máxima*/)
        {
            if (/*compruebo que los 2 primeros planos no son iguales*/)
            {
                for (/*una vez elegido el segundo plano, busco un tercer plano
                    que me forme el triangulo con la suma de lados mas pequenia posible*/)
                {
                    /*en cada iteracion compruebo si la suma de vertices del conjunto de
                    triangulos elegido es menor que otro que ya se haya elegido
                    y que no se hayan seleccionado dos veces el mismo plano*/
                }
                if (/*Si no existe ya una union con los 3 planos seleccionados*/)
                {
                    //unir los 3 planos por ambos lados
                }
            }
        }
        /*Descarto el primer plano seleccionado del conjunto de planos
        porque ya he hecho todas las uniones de ese plano*/
    }
}
```

Los parámetros de entrada son una lista con los objetos de los planos, el objeto donde se van a guardar las uniones y el material para las uniones.

Esta función permite obtener algo como esto.

Indice



Esta función utiliza funciones del script "JuntarCirculos" para hacer las uniones. Y también para implementar esta función también he implementado 2 funciones auxiliares :

- "obtener_N_planos_mas_cercanos" que calcula la distancia de un plano con el resto y lo devuelve en una tupla con los planos ordenados de menor a mayor distancia y la distancia a cada plano.

Indice

```
//dado una lista de planos, obtiener los n planos mas cercanos al objeto de entrada
//el ratioPenalizacion es un valor que penaliza los circulos mas antiguos
//el umbral anula circulos cuya distancia sea muy proxima a la de otro objeto
3 referencias
(GameObject[], float[]) obtener_N_planos_mas_cercanos(int n, List<GameObject> objectList, Vector3 posobjeto)
{
    GameObject[] cercanos = new GameObject[n];
    float[] distancia = new float[n];
    float dist;
    int i = 0;

    foreach (GameObject element in objectList)
    {
        if (element != null && element.activeSelf)
        {
            dist = Vector3.Distance(posobjeto, element.transform.position);

            i = 0;
            while (i < n - 1 && cercanos[i] != null && dist >= distancia[i])
            {
                i++;
            }
            if (cercanos[i] == null)
            {
                cercanos[i] = element;
                distancia[i] = dist;
            }
            else if (dist < distancia[i])
            {
                for (int j = n - 2; j >= i; j--)
                {
                    distancia[j + 1] = distancia[j];
                    cercanos[j + 1] = cercanos[j];
                }
                distancia[i] = dist;
                cercanos[i] = element;
            }
        }
    }
    return (cercanos, distancia);
}
```

- Y "calcularDistanciasMedias" que calcula la distancia media de cada plano al resto y la devuelve en una tupla con los planos ordenados de menor a mayor distancia media.

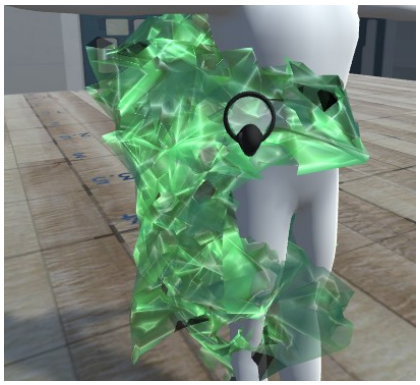
```
/*Calcula las distancias medias de cada plano al resto y lo devuelve en
 2 arrays con los objetos y el valor ordenados de menor a mayor distancia*/
1 referencia
(GameObject[],float[]) calcularDistanciasMedias(List<GameObject> jplanos)
{

    GameObject[] jDist;
    float[] distancia;
    int n = 0;
    float avg;

    //Cuento el numero de planos no nulos que hay en la lista
    foreach (GameObject jDplano in jplanos)
    {
        if (jDplano != null && jDplano.activeSelf)
        {
            n++;
        }
    }
```

```
    GameObject[] jObjDistanciaMedia = new GameObject[n];
    float[] jNumDistanciaMedia = new float[n];

    //por cada plano calculo la distancia media a los otros planos y los ordeno de menor a mayor distancia
    foreach (GameObject jDplano in jplanos)
    {
        if (jDplano != null && jDplano.activeSelf)
        {
            (jDist, distancia) = obtener_N_planos_mas_cercanos(n, jplanos, jDplano.transform.position);
            int i = 0;

            avg = 0;
            while (i < distancia.Length)
            {
                if (jDist[i] != null)
                {
                    avg += distancia[i];
                    i++;
                }
            }

            avg = avg / distancia.Length;
```

```
            i = 0;
            while (jObjDistanciaMedia[i] != null && jNumDistanciaMedia[i] < avg)
            {
                i++;
            }

            if (jObjDistanciaMedia[i] == null)
            {
                jObjDistanciaMedia[i] = jDplano;
                jNumDistanciaMedia[i] = avg;
            }
            else
            {
                for (int j = n - 2; j >= i; j--)
                {
                    jObjDistanciaMedia[j + 1] = jObjDistanciaMedia[j];
                    jNumDistanciaMedia[j + 1] = jNumDistanciaMedia[j];
                }
                jObjDistanciaMedia[i] = jDplano;
                jNumDistanciaMedia[i] = avg;
            }

        }
    }

    return (jObjDistanciaMedia, jNumDistanciaMedia);
}
```

La función "juntarMeshHijos" crea un mesh como combinación de los meshes de todos los hijos de un objeto utilizando la función "CombineMeshes" de Unity.

Aunque los mesh estén unidos y visualmente así se vea, si se les añade la propiedad cloth, estos se destruyen en pedazos.

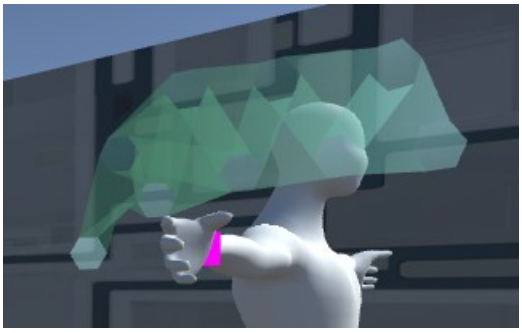


Esto sucede porque la función "CombineMeshes" junta los meshes, pero aunque 2 vértices del mesh estén en la misma posición, siguen siendo 2 vértices distintos, así que he tenido que crear una función que corrija eso llamada "pegarMeshObjeto", la cual busca todas las combinaciones de vértices en las que 2 vértices estén en la misma posición y entonces elimina uno de los vértices y reasigna en los triángulos donde se utilizaba el vértice repetido al que se ha mantenido.

Indice

```
void pegarMeshObjeto(GameObject objeto)
{
    for (/*para todos los vertices del mesh*/)
    {
        while (/*busco otro vertice que se encuentre en la misma posición*/)
        {
            //incrementar contador
        }
        if (/*si se ha encontrado un vértice en la misma posición*/)
        {
            /*elimino el primer vertice y */
            for (/*para todos los triangulos*/)
            {
                /*reasigno los triángulos que hecian referencia al primer vertice al segundo*/
            }
        }
    }
}
```

Una vez implementadas estas 2 funciones, y ajustados los los parámetros de la propiedad cloth, el objeto ya se comporta como una tela.




La función "buscar2padres" es una función que se utiliza en la funcionalidad de enganchar objetos al cuerpo, lo que hace es dado un a malla, busca cuales son las 2 partes del cuerpo a las que se les puede enganchar el objeto. Para ello busca las partes del cuerpo que están lo suficientemente cerca a un plano cualquiera del objeto para que puedan ser enganchadas y de esas, escoge las 2 con menor distancia. En caso de que no se encuentre ninguna parte del cuerpo que tenga esas características o solo encuentra 1, devuelve 1 o 2 objetos nulos.

```
public (GameObject,GameObject) buscar2padres(GameObject modeloKinect, GameObject objeto, float maxDistancia)
{
    GameObject padre1 = null;
    GameObject padre2 = null;

    float dist1 = maxDistancia;
    float dist2 = maxDistancia;

    Vector3[] vertices = objeto.GetComponent<MeshFilter>().mesh.vertices;
    int n = vertices.Length;
    float aux = maxDistancia;
    float aux1;
```

Indice

```
foreach (Transform child in modeloKinect.transform)
{
    aux = maxDistancia;
    for (int i = 0; i < n; i++)
    {
        //minima distancia entre todos los vertices del mesh
        aux1 = Vector3.Distance(objeto.transform.InverseTransformDirection(child.position), vertices[i]);
        aux = Mathf.Min(aux, aux1);
    }

    if (aux < maxDistancia && aux < dist2)
    {
        //Sustituir padre1
        if (aux < dist1)
        {
            dist2 = dist1;
            if (padre1 != null)
            {
                padre2 = padre1;
            }
            dist1 = aux;
            padre1 = child.gameObject;
        }
        //Sustituir padre2
        else
        {
            dist2 = aux;
            padre2 = child.gameObject;
        }
    }
```

La función "engancharCloth" es una función que recibe un cloth y engancha las partes del cloth cercanas al modelo humano de forma que cuando esté el cloth activo, esas partes no se van a ver afectadas por las físicas del cloth.

```
/*Esta función engancha los vertices del mesh
 * si tienen una parte del cuerpo cerca */
1 referencia
public void engancharCloth(GameObject objetocloth, GameObject modeloKinect, float distancia)
{
    Vector3[] vertices;
    Vector3 posicionLocalenganche;

    vertices = objetocloth.GetComponent<MeshFilter>().mesh.vertices;

    ClothSkinningCoefficient[] newConstraints;
    newConstraints = objetocloth.GetComponent<Cloth>().coefficients; ;

    int n = vertices.Length;

    for (int i = 0; i < n; i++)
    {
        foreach (Transform child in modeloKinect.transform)
        {
            posicionLocalenganche = objetocloth.transform.InverseTransformPoint(child.position);
            if (Vector3.Distance(vertices[i], posicionLocalenganche) < distancia)
            {
                newConstraints[i].maxDistance = 0;
                break;
            }
        }
    }

    objetocloth.GetComponent<Cloth>().coefficients = newConstraints;
}
```

CrearPlanoCircular

Este script es el que se encarga de crear el mesh del plano circular que se va a utilizar a la hora de pintar, el script no se ejecuta en el scene principal, se ha ejecutado en un scene a parte para crear el mesh. Una vez creado el mesh, el script ya no se vuelve a utilizar si no es para crear otro círculo distinto.

Parámetros del Script :

resolucion : Es un entero > 2 en el que se indica el número de vértices (sin contar el centro) que va a tener el plano.

Radio : Un float en el que se indica el radio del círculo.

Material : (Opcional) Si se pone un material, mostrará el circulo con ese material en el scene.

ActivarGuardado : booleno que si está a true, guarda el mesh del circulo en la ruta especificada.

Ruta : La ruta en la que se guarda el mesh. (¡¡ Hay que tener cuidado con la ruta, puede sobrescribir carpetas y otros ficheros del proyecto si no se pone con cuidado !!)

Para explicar como funciona el script, antes voy a explicar como funcionan los mesh en Unity. Todos los meshes están formados por combinación de triángulos, para generar esos triángulos se utiliza un array de Vector3, donde en cada posición se almacena la posición de un vértice; Un array de Vector2 donde se almacena las coordenadas de las texturas, las cuales son las mismas coordenadas que los vértices sin el componente y. Y un array de enteros donde cada 3 posiciones del array se almacenan los índices del array de vértices que forman ese triángulo. Cuando se forman los triángulos, estos solo son visibles desde una cara, por lo que si nos interesa que se vea por las 2 caras, hay que cambiar el orden de los triángulos. Ej:

Tengo el array de vértices ((0,0,0),(0,1,2),(2,1,0))

El array de uvs es igual que el de vértices, eliminando el segundo componente de cada vértice ((0,0),(0,2),(2,0))

Si quiero hacer un triángulo que una esos 3, en el array de triángulos almaceno el índice de los vértices que forman ese triángulo (0,1,2)

Si quiero que el triángulo se vea por las 2 caras, entonces añado otras 3 posiciones invirtiendo el orden de los triángulos (0,1,2,1,0,2)

De esta forma ya tendría un triángulo que une los 3 vértices por las 2 caras.

Para la función "crearCirculo", primero inicializo las variables con el tamaño que vaya a utilizar.

En la variable "vertices" la inicializo como un array de Vector3 de tamaño del número de vértices del círculo + 1 que es el centro del círculo, utilizo la misma inicialización para el array uv cambiando el Vector3 por Vector2.

Para la variable triángulos hay que calcular el número de triángulos que va a tener el mesh. En mi caso es "resolución" * 3 (cada triángulo ocupa 3 posiciones) * 2 (dibujo los triángulos por ambas caras)

Indice

```
GameObject crearCirculo(float radio) {

    float rad = 10 * radio;

    Vector3[] vertices = new Vector3[resolucion + 1];
    Vector2[] uv = new Vector2[resolucion + 1];
    int[] triangulos = new int[6 * resolucion];
```

Para crear los vértices y los triángulos primero creo el vértice central del círculo en la posición (0,0,0) y después, para recorrer el bucle y obtener cada vértice, creo una variable "angulo" donde almaceno el angulo del vértice a crear y otra variable "trozo" donde almaceno el incremento de "angulo en cada iteración.

```
    vertices[resolucion] = new Vector3(0,0,0);
    uv[resolucion] = new Vector2(0, 0);

    int angulo = 0;
    int trozo = 360 / resolucion;
```

```
    vertices[0] = obtenerPuntoCirculo(angulo, rad);
    uv[0] = new Vector2(vertices[0].x, vertices[0].z);
    angulo += trozo;

    for(int i=1; i<resolucion; i++) {

        vertices[i] = obtenerPuntoCirculo(angulo, rad);
        uv[i] = new Vector2(vertices[i].x, vertices[i].z);

        //creo triangulos en ambas caras
        triangulos[6 * (i - 1)] = resolucion;
        triangulos[6 * (i - 1) + 1] = i;
        triangulos[6 * (i - 1) + 2] = i - 1;

        triangulos[6 * (i - 1) + 3] = i - 1;
        triangulos[6 * (i - 1) + 4] = i;
        triangulos[6 * (i - 1) + 5] = resolucion;

        angulo += trozo;
    }
```

```
    triangulos[6 * (resolucion - 1)] = resolucion - 1;
    triangulos[6 * (resolucion - 1) + 1] = resolucion;
    triangulos[6 * (resolucion - 1) + 2] = 0;

    triangulos[6 * (resolucion - 1) + 3] = 0;
    triangulos[6 * (resolucion - 1) + 4] = resolucion;
    triangulos[6 * (resolucion - 1) + 5] = resolucion - 1;
```

```
    Mesh mesh = new Mesh();

    mesh.vertices = vertices;
    mesh.uv = uv;
    mesh.triangles = triangulos;

    GameObject plano = new GameObject();

    plano.AddComponent<MeshFilter>().mesh = mesh;
    if(material != null)
    {
        plano.AddComponent<MeshRenderer>().material = material;
    }
    plano.transform.name = "PlanoCircular";

    if (activarGuardado){
        AssetDatabase.CreateAsset(mesh, ruta);
        AssetDatabase.SaveAssets();
    }

    return plano;

}
```

Para obtener la posición de los vértices hago un cálculo con la función "obtenerPuntoCirculo", con la ayuda de la posición del centro, el ángulo y el radio del círculo.

```
/*Dado el angulo y el radio del circulo obtiene un punto del borde del circulo
correspondiente a ese angulo*/
2 referencias
Vector3 obtenerPuntoCirculo(int angulo, float rad) {

    Vector3 punto = new Vector3();

    punto.x = rad * Mathf.Cos(angulo * Mathf.Deg2Rad);
    punto.y = 0;
    punto.z = rad * Mathf.Sin(angulo * Mathf.Deg2Rad);

    return punto;

}
```

Ejemplo de un Circulo creado utilizando este script con resolución = 6.

Indice

JuntarCirculos

Este script se encarga de juntar los planos en grupos de 3, para eso he creado 2 formas de hacerlo, cada una con sus ventajas e inconvenientes.

Resolucion : Número de vértices de los círculos.

NumVerticesUnion : Número de vértices que se van a unir en cada círculo (este número debe ser par)

cotaGradosLineal : Parámetro que sirve para calcular si 3 círculos están colocados en línea en la función "esLineal"

modoSimple3 : Activa la versión simplificada para juntar los círculos, genera muchos menos triángulos, pero funciona peor con la propiedad cloth.

Una forma de juntar los círculos es con la función "obtenerMesSimpleNoLineal_3", este script solo si el parámetro "modoSimple3" está a true. Para juntar los círculos, simplemente crea un triángulo que junte los centros de los 3 círculos por los 2 lados y los devuelve como un mesh.

Los parámetros de entrada de la función son un array con los objetos que contienen los círculos ("circulos") y el objeto para el que se va a crear el mesh resultante de la unión ("medio").

```
Mesh obtenerMeshSimpleNoLineal_3(GameObject[] circulos, GameObject medio) {
    Vector3[] vertices = new Vector3[3];
    Vector2[] uv = new Vector2[3];
    int[] triangulos = new int[6];
    Mesh mesh = new Mesh();

    for (int i = 0; i < 3; i++) {
        vertices[i] = medio.transform.InverseTransformPoint(
            circulos[i].transform.TransformPoint(
                circulos[i].GetComponent<MeshFilter>().mesh.vertices[resolucion]));
        print(vertices[i]);
        uv[i] = medio.transform.InverseTransformPoint(
            circulos[i].transform.TransformPoint(
                circulos[i].GetComponent<MeshFilter>().mesh.uv[resolucion]));
        triangulos[i] = i;
    }
    triangulos[3] = 0;
    triangulos[4] = 2;
    triangulos[5] = 1;

    mesh.vertices = vertices;
    mesh.uv = uv;
    mesh.triangles = triangulos;

    return mesh;
}
```

La función "InverseTransformPoint" es una función de Unity que convierte las coordenadas globales pasadas como parámetro de entrada en locales respecto al objeto con el que se ha llamado a la función.

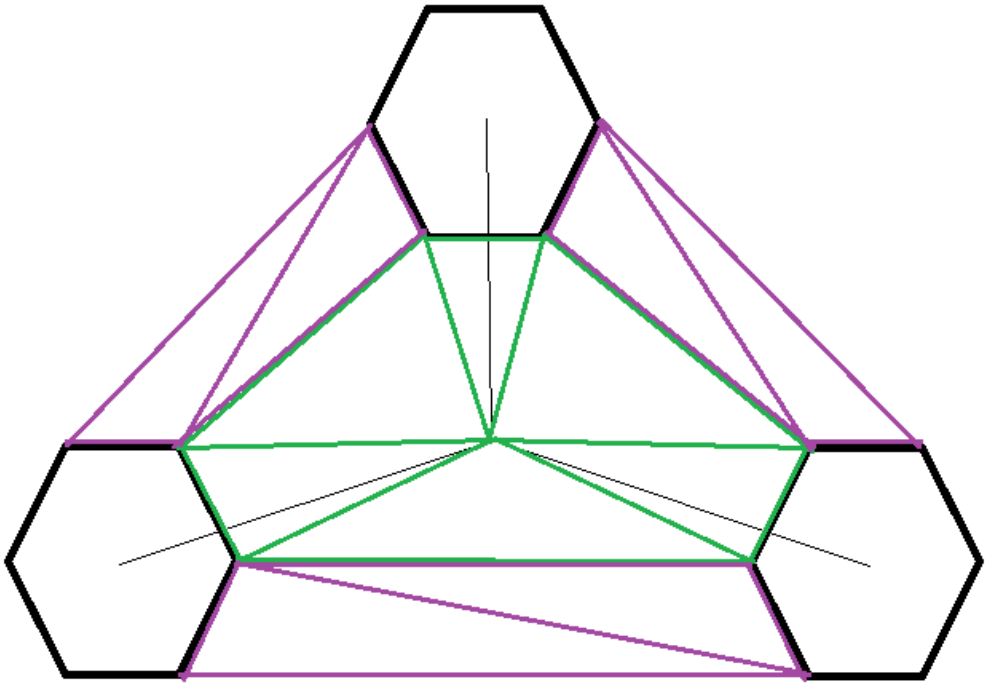
La función "TransformPoint" hace lo contrario, transforma coordenadas locales con respecto a un

objeto en globales.

Ambas funciones las utilizo para transformar las coordenadas de los centros de los círculos en coordenadas para el objeto "medio".

Ejemplo unión Simple :

La otra función que he implementado para unir los círculos es "obtenerMeshNoLineal". Para unir los círculos se pretende unirlos como en el dibujo de abajo. Las líneas negras gruesas negras son los círculos que hay que unir, las líneas verdes son los triángulos que se van a generar siempre que se ejecute esta función, las líneas moradas son líneas que se pueden dibujar o no en función del valor del parámetro "NumVerticesUnion" del script y las líneas negras delgadas muestran en qué aristas cortan la unión entre el centro de los 3 círculos y los centros de cada círculo.



Para hacer esta función he presupuesto que los círculos que se pasan por entrada han sido creados utilizando un mesh creado por la función "CrearPlanoCircular", de forma que vértice de la posición "resolución" es el centro del círculo, el primer vértice es el dibujado en el borde de 0º y los

siguientes aumentan sucesivamente el ángulo hasta llegar al último antes de 360º. Si se pasan por entrada otro tipo de figuras o el parámetro "resolución" está mal configurado, el resultado de la unión es impredecible.

Para la implementación de la función he utilizado 5 funciones auxiliares : "obtenerPosLocal", "obtenerAngulo", "obtenerNverticesCercanos", "verticesMasCercanos", "asignarTriangulo".

La función "obtenerPosLocal" simplemente recibe como entrada 2 objetos y devuelve la posición relativa del objeto2 al objeto1, como si en objeto1 estuviera el origen de coordenadas.

```
//calcula la posicion relativa del objeto2 al objeto1
7 referencias
Vector3 obtenerPosLocal(GameObject objeto1, GameObject objeto2) {
    return objeto1.transform.InverseTransformPoint(objeto2.transform.position);
}
```

La función "obtenerAngulo" devuelve el ángulo donde está el vértice del círculo más cercano al centro de los 3 círculos.

```
/*devuelve el angulo donde esta el punto más proximo al centro de los circulos
dado la posicio local del un circulo al centro de circulos*/
7 referencias
float obtenerAngulo(Vector3 Local) {
    float ang = Mathf.Atan(Local.z / Local.x) * Mathf.Rad2Deg;
    //compensa el angulo cuando el cuadrante de x es negativo
    if (Local.x < 0) {
        ang += 180;
    }

    ang = ang % 360;
    if (ang < 0) {
        ang += 360;
    }

    return ang;
}
```

La función "obtenerNverticesCercanos" devuelve los n vértices más cercanos del círculo al ángulo dado.

```
/*devuelve en coordenadas reales los vertices del circulo mas cercanos al angulo dado alpha
en caso de que un vertice coincida con alpha el segundo vertice sera el mas proximo con menor angulo*/
//Para que funcione es importante que los circulos se hayan creado con el script "crear_plano_circulo.cs"
//n <= resolucion, n par
1 referencia
Vector3[] obtenerNverticesCercanos(GameObject circulo, float alpha, int resolucion, int n = 2) {
    Vector3[] vertices = new Vector3[n];

    Mesh mesh = circulo.GetComponent<MeshFilter>().mesh;

    int angulo = 0;
    int trozo = 360 / resolucion;

    int i = 0;
    float dist_ite_ant = Mathf.Abs(alpha - angulo);
    float dist_ite_sig = Mathf.Abs(alpha - (angulo + trozo));

    //busco entre qué vertices esta el angulo
    while (angulo < 360 && dist_ite_sig < dist_ite_ant && i<resolucion-1) {
        i++;
        angulo += trozo;
        dist_ite_ant = dist_ite_sig;

        dist_ite_sig = Mathf.Abs(alpha - (angulo+trozo));
    }
```

```
    int vi;
    int vj;

    //compruebo si el angulo esta entre el ultimo vertice y el primero
    if (angulo >= alpha && i!=0) {
        vertices[0] = circulo.transform.TransformPoint(mesh.vertices[i - 1]);
        vertices[1] = circulo.transform.TransformPoint(mesh.vertices[i]);

        vi = i - 1;
        vj = i;
    }
    else if (i < resolucion-1){
        vertices[0] = circulo.transform.TransformPoint(mesh.vertices[i]);
        vertices[1] = circulo.transform.TransformPoint(mesh.vertices[i + 1]);

        vi = i;
        vj = i + 1;
    }
    else {
        vertices[0] = circulo.transform.TransformPoint(mesh.vertices[i]);
        vertices[1] = circulo.transform.TransformPoint(mesh.vertices[0]);

        vi = i;
        vj = 0;
    }
```

```
    //obtengo el resto de vertices
    for(int ii = 2; ii < n; ii = ii + 2) {

        vi--;
        vj++;

        if (vi < 0) {
            vi = resolucion - 1;
        }
        if (vj > resolucion - 1) {
            vj = 0;
        }

        vertices[ii] = circulo.transform.TransformPoint(mesh.vertices[vi]);
        vertices[ii + 1] = circulo.transform.TransformPoint(mesh.vertices[vj]);
    }

    return vertices;
}
```

La función "verticesMasCercanos" dados 4 vértices en grupos de 2, devuelve los índices donde están los vértices más cercanos entre grupos distintos.

```
//devuelve la posicion de los 2 vertices mas cercanos entre los 2 grupos de vertices
//Los vectores de entrada deben ser ambos de tamanio 2
4 referencias
int[] verticesMasCercanos(Vector3[,] vertices) {
    int[] cercanos = new int[2];

    float dist;
    float dist_new;

    cercanos[0] = 1;
    cercanos[1] = 0;
    dist = Vector3.Distance(vertices[0,1], vertices[1,0]);
    dist_new = Vector3.Distance(vertices[0,0], vertices[1,1]);

    if (dist_new < dist) {
        cercanos[0] = 0;
        cercanos[1] = 1;
        dist = dist_new;
    }

    return cercanos;
}
```

La función "asignarTriangulos" dados el vector de triángulos, el indice actual y los indices de los vértices del triangulo, crea 1 nuevo triángulo por ambas caras, el objetivo de esta función es simplemente reducir el número de líneas de código de la función "obtenerMeshNoLineal".

```
void asignarTriangulo(int[] triangulos, int index,int v1,int v2, int v3)
{
    triangulos[index] = v1;
    triangulos[index + 1] = v2;
    triangulos[index + 2] = v3;

    triangulos[index + 3] = v2;
    triangulos[index + 4] = v1;
    triangulos[index + 5] = v3;

}
```

Una vez explicadas las funciones auxiliares, voy a explicar la función "obtenerMeshNoLineal".

Antes de crear los triángulos, primero inicializo los arrays calculando el número de espacios que voy a utilizar.

Como número de vértices, utilizo los vértices de todos los círculos y uno nuevo, el centro de los círculos.

Como número de uvs igual que de vértices.

Como número de triángulos he hecho el siguiente cálculo : 3 (3 posiciones/triangulo) * 2(2 caras) * circulos.Length (numero de círculos) * (2 (de media por cada círculo hay 2 triángulos verdes, los cuales se dibujan siempre) + numVerticesUnión (de media por cada círculos hay "numVerticesUnion" triángulos morados))

Tras inicializar las variables, inicializo el vértice del centro de círculos, es cual está en el origen del objeto "medio".

```
Mesh obtenerMeshNoLineal(GameObject[] circulos, GameObject medio) {
    float alpha;
    int tamV = numVerticesUnion * circulos.Length + 1;
    int tamT = 3 * 2 * circulos.Length * (2 + numVerticesUnion);

    int[] cercanos;
    Vector3[,] ver = new Vector3[2,2];
    Vector3[] verticesCirculo;
    Vector3[] vertices = new Vector3[tamV];
    Vector2[] uv = new Vector2[tamV];
    int[] triangulos = new int[tamT];

    //inicializo el vertice del centro de circulos
    vertices[tamV - 1] = new Vector3(0, 0, 0);
```

Llamo a las funciones creadas anteriormente para obtener los vértices que hay que unir y almacenarlos en el array de vértices.

Creo las variables "indexV" e "indexT", las cuales utilizo para recorrer los arrays de vértices y triángulos.

Los vértices que devuelve la función "obtenerNverticesCercanos" están ordenados de forma que los primeros 2 son los vértices más cercanos al medio, por lo que formo un triángulo con ellos.

```
/*en cada iteracion se obtiene la posicion de los vertices del circulo
mas cercanos al punto medio y se crean los triangulos*/
for (int i = 0; i < circulos.Length; i++) {
    alpha = obtenerAngulo(obtenerPosLocal(circulos[i], medio));

    verticesCirculo = obtenerNverticesCercanos(circulos[i], alpha, resolucion, numVerticesUnion);

    int indexV = numVerticesUnion * i;
    int indexT = 3 * i * (2 + 2 + numVerticesUnion * 2);

    /*Se convierten las coordenadas locales de los vertices del circulo a
    posiciones locales del medio*/
    for(int ii = 0; ii < numVerticesUnion; ii++) {
        vertices[indexV + ii] = medio.transform.InverseTransformPoint(verticesCirculo[ii]);
        uv[indexV + ii] = new Vector2(vertices[indexV + ii].x, vertices[indexV + ii].z);
    }

    //triangulos para juntar con el centro en ambas caras
    asignarTriangulo(triangulos, indexT,
        indexV,
        indexV + 1,
        tamV - 1);
```

Después de unir los círculos con el medio, hay que unirlos entre ellos, en el caso del primer círculos, hay que unirlo con el segundo y el último. En la primera iteración no se pueden unir, porque los vértices de esos círculos aun no se han generado, por lo que se unirán al final.

A partir de la segunda iteración se unen los vértices del círculo actual con el círculo anterior, primero se generan los triángulos verdes.

```
//El vertice del primer circulo se tienen que unir con el del ultimo
if (i > 0) {
    //Al unir circulos entre si hay que juntarlos por los vertices mas cercanos
    ver[0,0] = vertices[indexV];
    ver[0,1] = vertices[indexV + 1];
    ver[1,0] = vertices[indexV - numVerticesUnion];
    ver[1,1] = vertices[indexV - numVerticesUnion + 1];
    cercanos = verticesMasCercanos(ver);

    //triangulos para juntar los circulos en ambas caras
    asignarTriangulo(triangulos, indexT + 6,
        indexV - numVerticesUnion + cercanos[1],
        indexV + cercanos[0],
        tamV - 1);
```

Después de haber generado los triángulos verdes se unen los morados según el parámetro "numVerticesUnion".

```
        //se juntan los vértices exteriores segun el valor de numVerticesUnion
        for (int ii = 2; ii < numVerticesUnion; ii = ii + 2) {
            ver[0, 0] = vertices[indexV + ii];
            ver[0, 1] = vertices[indexV + ii + 1];
            ver[1, 0] = vertices[indexV - numVerticesUnion + ii];
            ver[1, 1] = vertices[indexV - numVerticesUnion + ii + 1];

            cercanos = verticesMasCercanos(ver);

            asignarTriangulo(triangulos, indexT + ii * 6,
                indexV - numVerticesUnion + ii + cercanos[1],
                indexV + ii + cercanos[0],
                triangulos[indexT + 7]);

            asignarTriangulo(triangulos, indexT + ii * 6 + 6,
                indexV - numVerticesUnion + ii + cercanos[1],
                triangulos[indexT + 7],
                triangulos[indexT + 6]);
        }
    }
}
```

Para terminar se une el primer círculo con el último.

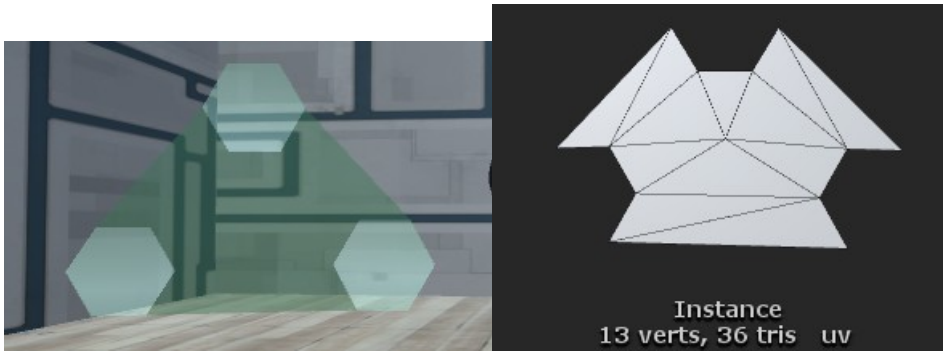
```
    //se unen los vertices del primer circulo con el del ultimo
    ver[0,0] = vertices[0];
    ver[0,1] = vertices[1];
    ver[1,0] = vertices[tamV - numVerticesUnion - 1];
    ver[1,1] = vertices[tamV - numVerticesUnion + 1 - 1];
    cercanos = verticesMasCercanos(ver);

    asignarTriangulo(triangulos, 6,
        cercanos[0],
        tamV - 1,
        tamV - numVerticesUnion - 1 + cercanos[1]);

    for (int ii = 2; ii < numVerticesUnion; ii = ii + 2) {
        ver[0, 0] = vertices[ii];
        ver[0, 1] = vertices[ii + 1];
        ver[1, 0] = vertices[tamV - numVerticesUnion - 1 + ii];
        ver[1, 1] = vertices[tamV - numVerticesUnion - 1 + ii + 1];

        cercanos = verticesMasCercanos(ver);

        asignarTriangulo(triangulos, ii * 6,
            tamV - numVerticesUnion - 1 + ii + cercanos[1],
            ii + cercanos[0],
            triangulos[8]);

        asignarTriangulo(triangulos, ii * 6 + 6,
            ii + cercanos[0],
            triangulos[8],
            triangulos[6]);
    }
```

Finalmente se genera el nuevo mesh con los arrays creados y se devuelve.

Indice

```
    /*se genera el nuevo mesh con los vértices, circulos
    y triángulos utilizados*/
    Mesh mesh = new Mesh();

    mesh.vertices = vertices;
    mesh.uv = uv;
    mesh.triangles = triangulos;

    return mesh;
}
```

Ejemplo unión no Simple, resolución = 6, numVerticesUnion = 4.



Indice

Congelar

Este script implementa la funcionalidad de congelar el modelo humano para cuando se esté haciendo captura de movimiento con la Kinect, de forma que se quede quieto al pulsar un botón y se vuelva a mover al volver a pulsarlo.

La primera versión de este script creaba un duplicado del original y el original lo desactivaba, sin embargo, después de implementar los Cloths, tuve que crear una segunda versión porque solo me cogía las colisiones del modelo original a pesar de que estuviese desactivado, y para implementar la funcionalidad de guardado tenía que guardar el modelo con las colisiones y los cloths haciendo referencia a ellas, por lo que programé una segunda versión.

En la segunda versión, en lugar de crear un duplicado del original, utiliza el original, para ello hace un recorrido por los hijos del objeto y les desactiva los scripts que se encargan de leer los mensajes OSC de NiMate para que deje de actualizar las posiciones del cuerpo. Cuando se vuelve a pulsar el botón, se reactiva la funcionalidad de captura.

```
public GameObject modeloKinect;
bool congelado;
// Start is called before the first frame update
0 referencias
void Start()
{
    //Por defecto no esta congelado
    congelado = true;
}
// Update is called once per frame
0 referencias
void Update()
{
    //Si no estaba congelado y se ha pulsado el boton c, se congela
    if (Input.GetKeyDown("c") && congelado)
    {
        foreach(Transform child in modeloKinect.transform)
        {
            if (child.GetComponent<extOSC.Components.Events.OSCReceiverEventVector3>() != null)
            {
                child.GetComponent<extOSC.Components.Events.OSCReceiverEventVector3>().enabled = false;
            }
            if (child.GetComponent<Rotation.OSC_rotation>() != null)
            {
                child.GetComponent<Rotation.OSC_rotation>().enabled = false;
            }
            else if(child.GetComponent<Rotation.OSC_rotation_brazos>() != null)
            {
                child.GetComponent<Rotation.OSC_rotation_brazos>().enabled = false;
            }
            else if(child.GetComponent<Rotation.OSC_rotation_piernas>() != null)
            {
                child.GetComponent<Rotation.OSC_rotation_piernas>().enabled = false;
            }
        }
        congelado = false;
        print("Congelado");
    }
```

```
    //Si ya estaba congelado y se ha pulsado el botón c, se descongela
    else if (Input.GetKeyDown("c") && !congelado)
    {
        foreach (Transform child in modeloKinect.transform)
        {
            if (child.GetComponent<extOSC.Components.Events.OSCReceiverEventVector3>() != null)
            {
                child.GetComponent<extOSC.Components.Events.OSCReceiverEventVector3>().enabled = true;
            }
            if (child.GetComponent<Rotation.OSC_rotation>() != null)
            {
                child.GetComponent<Rotation.OSC_rotation>().enabled = true;
            }
            else if (child.GetComponent<Rotation.OSC_rotation_brazos>() != null)
            {
                child.GetComponent<Rotation.OSC_rotation_brazos>().enabled = true;
            }
            else if (child.GetComponent<Rotation.OSC_rotation_piernas>() != null)
            {
                child.GetComponent<Rotation.OSC_rotation_piernas>().enabled = true;
            }
        }
        congelado = true;
        print("Descongelado");
    }
}
```

GuardarPrefab y SerializeMesh

Este script implementa la funcionalidad del guardado de dibujos.

En la scene hay un objeto llamado "GuardarPrefab", el cual es el objeto que se va a guardar como prefab y va a contener los dibujos que se van haciendo y el modelo humano.

La forma de guardar un prefab es sencilla, solo hace falta un objeto y una ruta.

Parámetro de entrada :

objetoGuardar : Objeto que se va a guardar como prefab.

ruta : Ruta donde se va a guardar el objeto en el directorio del proyecto.

nombreFichero : Nombre con el que se va a guardar el prefab.

Formato : Formato con el que se va a guardar el prefab.

```
public class GuardarPrefab : MonoBehaviour
{
    public GameObject objetoGuardar;
    public string ruta = "Assets/DibujosGuardados/";
    public string nombreFichero = "IntroduceElNombre";
    public string formato = ".prefab";

    // Update is called once per frame
    0 referencias
    void Update()
    {
        //si se pulsa la tecla m, guarda objetoGuardar como un prefab en la ruta asignada, con el nombre asignado
        if (Input.GetKeyDown("m")) {
            string rutaCompleta = ruta + nombreFichero + formato;
            bool success;
            var prefab = PrefabUtility.SaveAsPrefabAsset(objetoGuardar, rutaCompleta, out success);
            if (success) {
                print("Prefab_Guardado");
            }
            else {
                print("ERROR, No se ha guardado el prefab");
            }
        }
    }
}
```

A pesar de que crear un prefab sea sencillo, el problema está en que los objetos no se guardan del todo bien. Los mesh instanciados durante la ejecución, como las uniones de planos y los cloths no se guardan.

Para poder almacenar los mesh instanciados, he encontrado un script llamado llamado "SerializeMesh" en un foro de internet. [9]

El funcionamiento del script es simple, como parámetros a serializar se utiliza un array de Vector3 para los vértices del mesh, un arrray de Vector2 para los uvs y un array int para los triángulos.

Cada vez que se asigna el script a un objeto, automáticamente se ejecuta la función start(), la cual llama a la función Serialize() que almacena en las variables los vértices, uvs y triángulos del mesh del objeto.

La función Awake() se ejecuta de forma parecida a Start() con la diferencia de que start() se ejecuta

solo cuando el componente está activado y Awake() se ejecuta antes que start, sin la necesidad de que el componente esté activado.

De esta forma, los mesh del prefab se cargarán incluso antes de haberlos instanciado en el espacio, para ello en Awake() se llama a la función rebuild que reconstruye el mesh con las variables serializadas.

```
namespace TheTide.utils {
    [ExecuteInEditMode]
    [RequireComponent(typeof(MeshFilter))]
    23 referencias
    public class SerializeMesh : MonoBehaviour {
        [HideInInspector] [SerializeField] Vector2[] uv;
        [HideInInspector] [SerializeField] Vector3[] verticies;
        [HideInInspector] [SerializeField] int[] triangles;
        [HideInInspector] [SerializeField] bool serialized = false;
        // Use this for initialization

        0 referencias
        void Awake() {
            if (serialized) {
                GetComponent<MeshFilter>().mesh = Rebuild();
            }
        }

        0 referencias
        void Start() {
            if (serialized) return;

            Serialize();
        }
```

```
        public void Serialize() {
            var mesh = GetComponent<MeshFilter>().mesh;

            uv = mesh.uv;
            verticies = mesh.vertices;
            triangles = mesh.triangles;

            serialized = true;
        }

        2 referencias
        public Mesh Rebuild() {
            Mesh mesh = new Mesh();
            mesh.vertices = verticies;
            mesh.triangles = triangles;
            mesh.uv = uv;

            mesh.RecalculateNormals();
            mesh.RecalculateBounds();

            return mesh;
        }
    }
```

La clase "SerializeMeshEditor" genera botones en el editor para poder ejecutar las funciones Serialize y Rebuild manualmente.

Indice

```
#if UNITY_EDITOR
    [CustomEditor(typeof(SerializeMesh))]
    0 referencias
    class SerializeMeshEditor : Editor {
        SerializeMesh obj;

        0 referencias
        void OnSceneGUI() {
            obj = (SerializeMesh)target;
        }

        22 referencias
        public override void OnInspectorGUI() {
            base.OnInspectorGUI();

            if (GUILayout.Button("Rebuild")) {
                if (obj) {
                    obj.gameObject.GetComponent<MeshFilter>().mesh = obj.Rebuild();
                }
            }

            if (GUILayout.Button("Serialize")) {
                if (obj) {
                    obj.Serialize();
                }
            }
        }
    }
#endif
}
```

Con este script ya tengo una solución para que los mesh de los objetos se guarden, lo único que tengo que hacer es modificar los scripts que dibujan objetos, para que se le añada al objeto el script cada vez que se defina un mesh.

A pesar de que ahora tengo una solución para los mesh, los Cloths son los que dan problemas, porque si se guarda un objeto con el cloth en funcionamiento, este no se puede guardar bien.

La solución que he desarrollado para este caso es, cuando se active la funcionalidad de guardado, antes de guardar el objeto, se elimina el componente cloth y se le añade un script al objeto que se encargue de volver a asignar el cloth cuando se cargue el objeto.

(No tengo imágenes del script, porque se me ha quedado en el PC del laboratorio durante la cuarentena)

Tras haber realizado estos arreglos, la funcionalidad de guardar es funcional.

SeleccionarDibujo

Este script se encarga de seleccionar el dibujo acrivo en función de la tecla del teclado pulsada de "0" a "9". La tecla "0" se utilizará para seleccionar el dibujo manual y de la "1" a la "9" se podrán almacenar hasta 9 dibujos de ejemplo. Por defecto el script comienza con la opción "0" seleccionada.

Parámetros de entrada :

ObjetosScript : Es un array de tamaño indicado por el usuario en el que se almacenan los objetos que se pueden seleccionar. El script puede seleccionar hasta un máximo de 9 objetos, si el tamaño del array es mayor, los objetos que sobrepasen la posición novena no se tendrán en cuenta.

ModeloKinect : Aquí se almacena el modelo humano, se utiliza para restablecer su posición inicial cada vez que se selecciona un objeto.

La implementación del script es sencilla, en la función start() se inicializan las variables y se desactivan los objetos del array "ObjetosScript".

```
public class SeleccionDibujo : MonoBehaviour
{
    public GameObject[] objetosScript;
    public GameObject modeloKinect;

    private int active;
    private int len;
    private Vector3 posInicial;
    private Quaternion rotInicial;

    // Start is called before the first frame update
    0 referencias
    void Start()
    {
        posInicial = modeloKinect.transform.position;
        rotInicial = modeloKinect.transform.rotation;
        len = Mathf.Min(objetosScript.Length, 9);

        for(int i = 0; i < len; i++)
        {
            objetosScript[i].SetActive(false);
        }

        active = 0;


    }
```

En el Update, compruebo si se ha pulsado una de las teclas entre "0" y "9" con 10 ifs y en el caso de que se pulse de "1" a "9" desactivo el objeto anterior activo o el dibujo manual y activo el nuevo objeto. Si se pulsa la tecla "0" se desactiva el objeto activo y se rehabilita el dibujo manual.

Indice

```
// Update is called once per frame
0 referencias
void Update()
{
    if (Input.GetKeyDown("0"))
    {
        if (active > 0)
        {
            objetosScript[active - 1].SetActive(false);
        }
        GetComponent<PainterDiscosPlanosColorV7_raton>().enabled = true;
        active = 0;
    }

    if (len > 0 && Input.GetKeyDown("1"))
    {
        if(active == 0)
        {
            GetComponent<PainterDiscosPlanosColorV7_raton>().enabled = false;
        }
        else
        {
            objetosScript[active - 1].SetActive(false);
        }
        objetosScript[0].SetActive(true);
        establecerPosicionRopa();
        active = 1;
    }
```

...

...

```
    if (len > 8 && Input.GetKeyDown("9"))
    {
        if (active == 0)
        {
            GetComponent<PainterDiscosPlanosColorV7_raton>().enabled = false;
        }
        else
        {
            objetosScript[active - 1].SetActive(false);
        }
        objetosScript[8].SetActive(true);
        establecerPosicionRopa();
        active = 9;
    }

}
```

```
void establecerPosicionRopa()
{
    modeloKinect.transform.position = posInicial;
    modeloKinect.transform.rotation = rotInicial;
}
}
```

Indice

Emparentar2

El script "Emparentar2" es un script que se encarga de emparentar un objeto a 2 padres para que se adapte al movimiento de ambos modificando la posición, la orientación y el tamaño del mismo cuando algún padre se mueva. Este script se utiliza en la funcionalidad de enganchar objetos al cuerpo.

Parámetros de entrada :

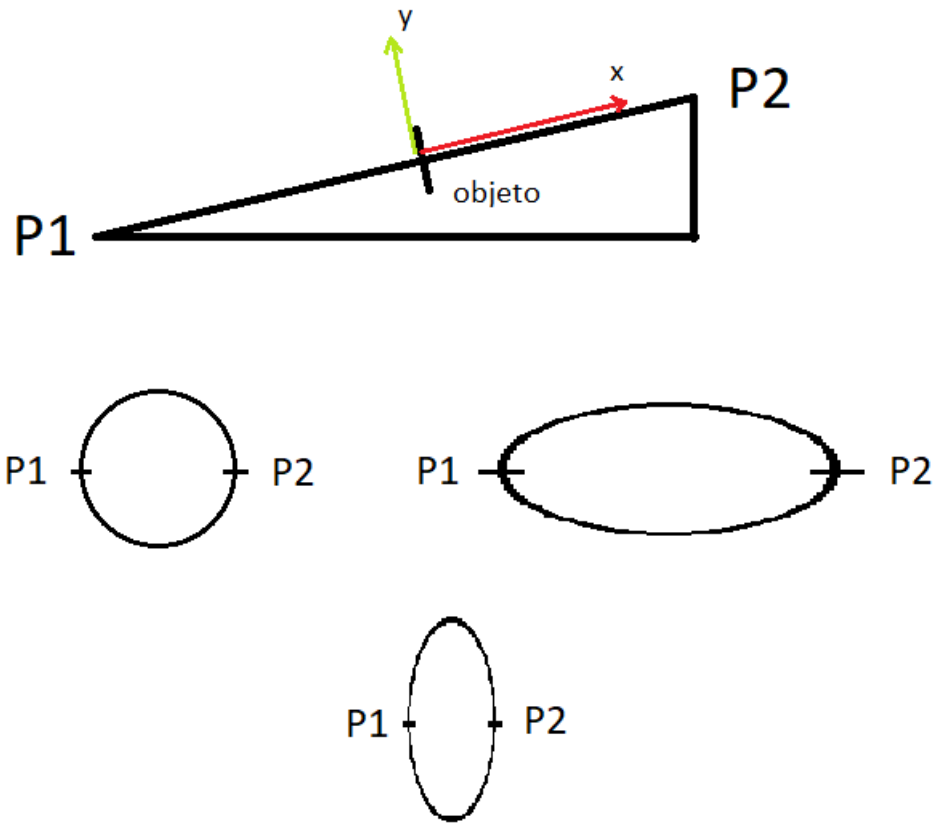
objeto : El objeto que se va a emparentar.

padre1 : El primer padre.

padre2 : El segundo padre.

limite : El límite de estiramiento en los ejes "y" y "z" cuando la distancia entre los padres es pequeña.

El funcionamiento del script consiste en crear un objeto y asignar su posición entre los 2 objetos y la orientación de forma que el eje "x" del objeto esté apuntando a la misma dirección de la recta que une los 2 padres y altera las dimensiones del objeto según como varia la distancia entre los 2 padres con respecto a la posición inicial.



Cuando los padres se alejan, el objeto aumenta su tamaño en el eje "x", pero reduce su tamaño en los ejes "y" y "z". Cuando se acercan, se reduce el tamaño en "x", pero se aumenta el de "y" y "z" hasta un límite. El límite está para evitar que un objeto se estire hasta el infinito.

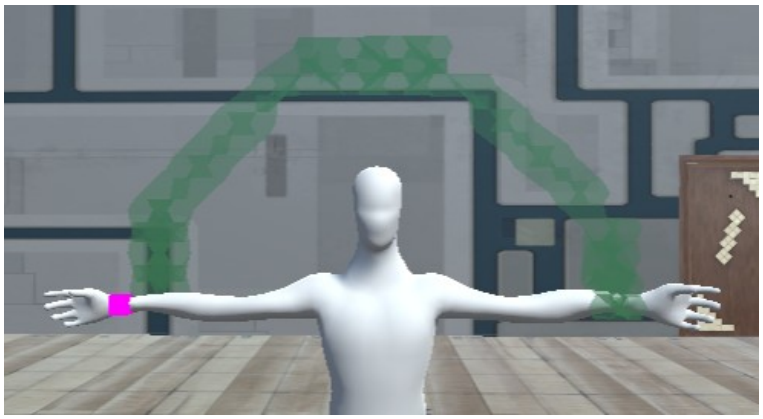
Indice

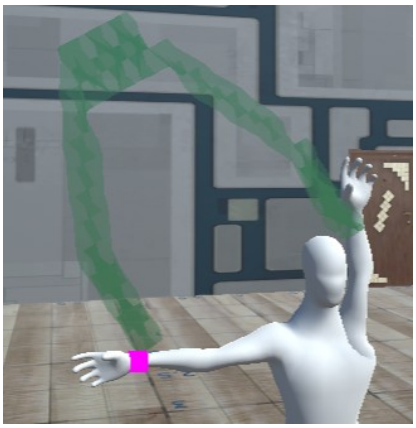
Una vez creado el objeto auxiliar, se emparenta el objeto original al objeto auxiliar, de forma que el objeto original no se vea alterado, y permitiendo que cuando el objeto auxiliar altere sus coordenadas y tamaños, el objeto original también se vea afectado de la misma forma.

```
void Start()
{
    dist = Vector3.Distance(padre1.transform.position, padre2.transform.position);
    distini = dist;

    //se crea el objeto auxiliar que se va a utilizar para emparentar el objeto original
    medio = new GameObject();
    medio.transform.position = (padre1.transform.position + padre2.transform.position) / 2;
    medio.transform.rotation = Quaternion.LookRotation(padre2.transform.position - padre1.transform.position, Vector3.up);

    Vector3 aux = objeto.transform.localScale;

    aux.x /= medio.transform.localScale.x;
    aux.y /= medio.transform.localScale.y;
    aux.z /= medio.transform.localScale.z;

    objeto.transform.localScale = aux;

    objeto.transform.SetParent(medio.transform);
}
```

```
void Update()
{
    dist = Vector3.Distance(padre1.transform.position, padre2.transform.position);
    medio.transform.localScale = new Vector3(Mathf.Min(distini / dist, limite), Mathf.Min(distini / dist, limite), dist / distini);
    medio.transform.position = (padre1.transform.position + padre2.transform.position) / 2;
    medio.transform.rotation = Quaternion.LookRotation(padre2.transform.position - padre1.transform.position, Vector3.up);
}
```

Ejemplo del funcionamiento del script :





Indice

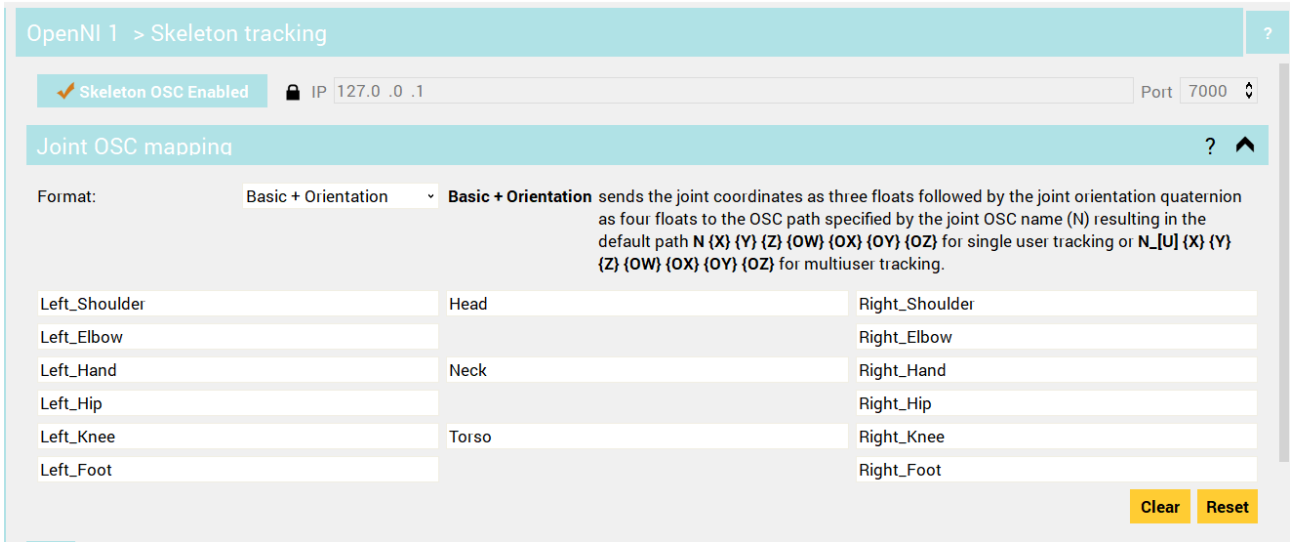
Integración Hardware-Software

Kinect con Ni mate y Unity

Para utilizar la kinect en unity y capturar movimiento he utilizado el Software NiMate, la librería extOSC de la AssetStore de Unity y unos scripts programados por mi.

Cuando se conecta la kinect al PC y NiMate está iniciado, este detecta automáticamente la kinect y crea una pestaña en la que se puede observar la captura.

Una vez que se ha comprobado que el programa captura la Kinect, hay que configurarla para que envié información del esqueleto vía OSC, para ello entro a la pestaña de "Skeleton tracking".



En esta pestaña, en la parte de arriba hay un cuadro que indica que está ya enviando mensajes OSC por la IP 127.0.0.1 en el puerto 7000, estas opciones se pueden modificar haciendo click en el candado. En este caso, como estoy utilizando Unity y NiMate en el mismo PC, puedo dejar puesta la IP por defecto, si lo tuviera que hacer en ordenadores separados, crearía una LAN y cambiaría la IP.

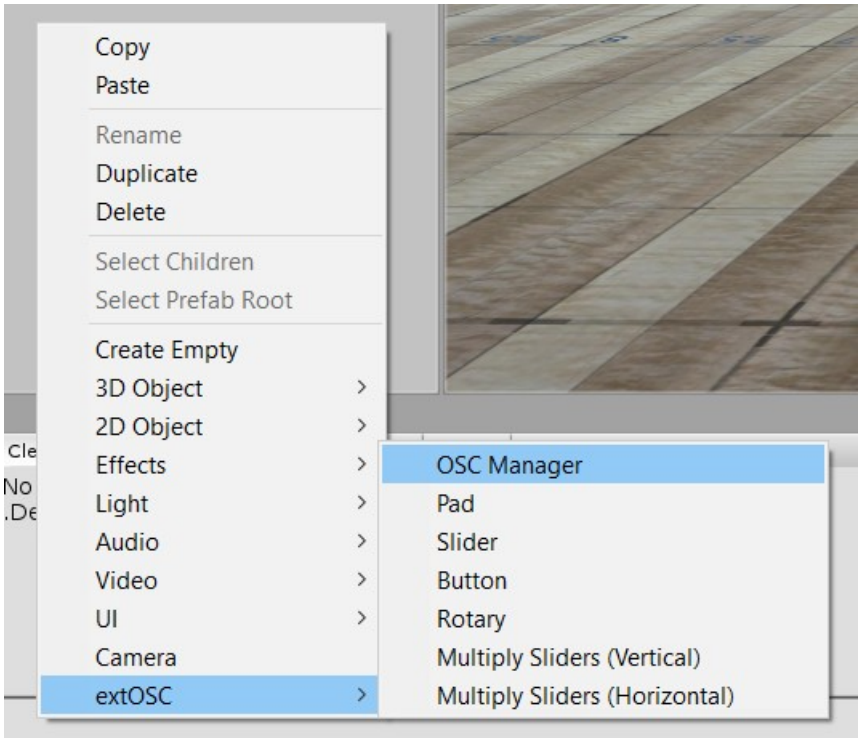
En la parte de abajo el programa indica qué información va a enviar ("Format") y los identificadores de las distintas partes del cuerpo que va a utilizar. Por defecto el cuadro de selección de "Format" viene como "Basic" de forma que solo enviá un vector de 3 elementos con la posición de las partes del cuerpo. En mi caso, como estoy utilizando un modelo humano, me interesa que envíe también la orientación del cuerpo, así que marco la opción de "Basic + Orientation" para enviar un vector de 7 elementos, en los primeros 3 la posición y en los últimos 4 la orientación. También el programa permite capturar a más de un usuario, en ese caso, el formato el formato cambia, pero en mi caso no lo utilizo.

En la parte de abajo de la pestaña hay una parte importante donde se puede modificar las coordenadas de la captura, la escala del cuerpo y también se puede invertir el movimiento derecha-izquierda. En mi proyecto, para que se ajuste al centro de la habitación y a la altura del suelo, tengo que hacer pequeñas correcciones en las coordenadas y como la izquierda y la derecha están invertidas, he invertido la escala en z (también se podría haber hecho marcando el cuadro de "Mirror left and right").

Como estoy utilizando la versión gratuita de NiMate, todos los parámetros se tienen que reconfigurar cada vez que se inicie el programa.

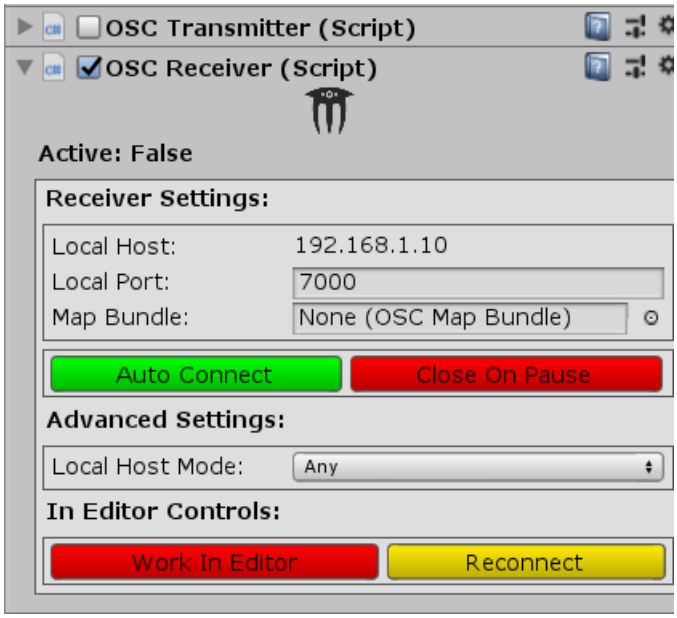
Indice

Por el lado de Unity, para utilizar la librería extOSC, en la Scene de Unity, hay que crear un objeto con los scripts que se encargan de la transmisión-recepción de mensajes, en mi proyecto lo he llamado "Controlador OSC"



En el "Controlador OSC" hay un transmisor y un receptor, en este caso, como no voy a enviar nada desde Unity, desactivo el transmisor (El transmisor también lo utilizo para otra funcionalidad, pero para la captura de movimiento no me hace falta).

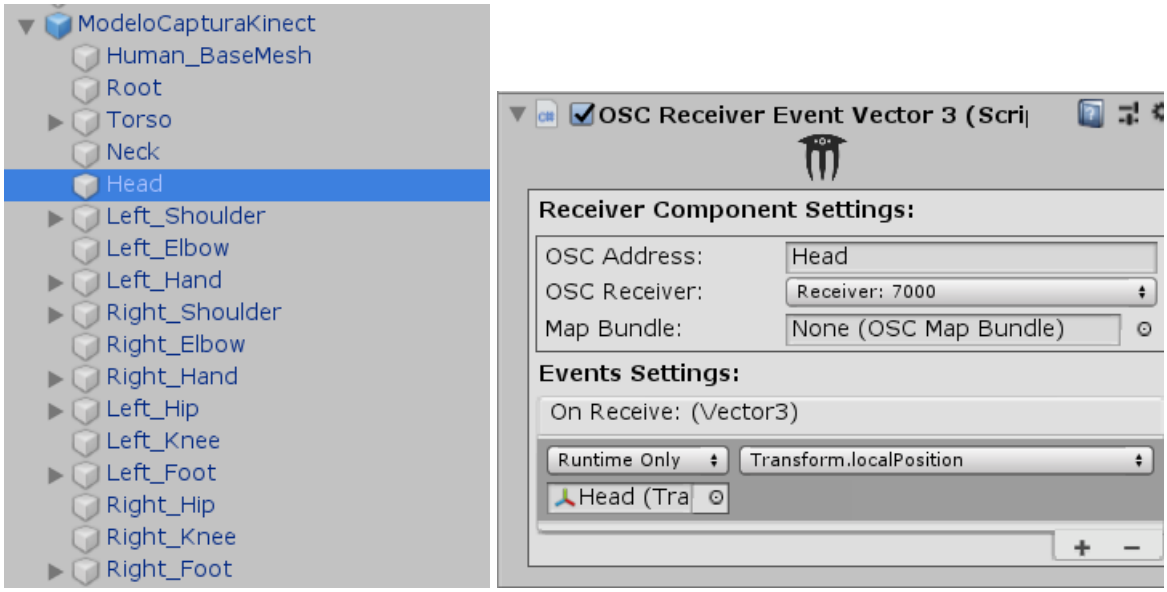
Por defecto, el receptor se configura automáticamente con la IP, como mucho habría que cambiar el puerto, pero como en NiMate envío por ese puerto, no necesito cambiar nada.



Ya está configurado el receptor, ahora lo último que falta y lo que mas tiempo me ha llevado es recibir e interpretar los mensajes y asignarles a las distintas partes del modelo las coordenadas correspondientes.

Indice

Para recibir los mensajes, la librería extOSC viene ya con scripts que permiten leer tipos básicos, entre ellos el tipo Vector3. Este script lo utilizo para leer los 3 primeros elementos del mensaje, donde viene la posición de las partes del cuerpo. Para ello, a cada parte del cuerpo, le he asignado el script "OSCReceiverEventVector3" y como parámetros, le asigno en "OSCAddress" el identificador correspondiente a ese parte del cuerpo, en "OSCReceiver", le asigno el receptor del objeto "ControladorOSC" y en "EventsSettings" le pongo que me transforme la posición en coordenadas locales.



Recibir e interpretar la posición es fácil y sencillo, pero para la rotación la cosa se complica bastante.

Para empezar, no hay ningún script predefinido que me permita leer los 4 últimos elementos del mensaje, por lo que he tenido que mirar como funciona internamente la librería para crear scripts que me permitan leer las 4 últimas posiciones del mensaje. Los scripts creado se llaman "OSC_rotación", "OSC_rotación_brazos" y "OSC_rotación_piernas". La estructura de los 3 Scripts es similar entre ellos, pero cambian algunos cálculos a la hora de interpretar las coordenadas.

Como parámetros de entrada, los scripts necesitan el identificador de la parte del cuerpo "Address" y el objeto "ControladorOSC".

En la función Start(), inicializa las variables y crea un escuchador que se encarga de recibir los mensajes dirigidos a su identificador.

```
public class OSC_rotation : MonoBehaviour
{
    #region Public Vars
    public string Address = "/Address";

    [Header("OSC Settings")]
    public OSCReceiver Receiver;

    public Quaternion qini;
    #endregion

    #region Unity Methods

    0 referencias
    protected virtual void Start()
    {
        Receiver.Bind(Address, ReceivedMessage);
        qini =transform.localRotation;
    }

    #endregion
```

Cada vez que se recibe un mensaje se ejecuta la función "ReceivedMessage", que se encarga de interpretar el mensaje, transformar las coordenadas para que se puedan utilizar en el modelo y asignarlas al modelo.

```
    #region Private Methods

    1 referencia
    private void ReceivedMessage(OSCMessage message)
    {
        var array = message.GetValues(message.GetTypes()[0]);
        var aux= new Quaternion(
            (float)array[4].Value,
            (float)array[5].Value,
            (float)array[6].Value,
            (float)array[3].Value
            );

        var aux2 = aux * qini;
        var aux3 = aux2.eulerAngles;

        aux3[0] = aux3[0];
        aux3[1] = aux3[1]-180;
        aux3[2] = aux3[2];


        transform.rotation = Quaternion.Euler(aux3);
    }

    #endregion
}
```

En este caso, este es el script "OSC_Rotacion", está pensado para ser utilizado en las partes centrales del cuerpo (cabeza, cuello y torso), para este caso, he sumado la rotación inicial del cuerpo a la del mensaje y he tenido que cambiar el sentido de la rotación en "y".

Indice

Para el script "OSC_rotacion_brazos" he cambiador el sentido de la rotación en el eje "z" en lugar de "y".

```
aux3[0] = aux3[0] ;
aux3[1] = aux3[1] ;
aux3[2] = aux3[2] + 180 ;
```

Para el script "OSC_rotación_piernas" a parte de cambiar el sentido de la rotación en el eje "y", también he invertido el valor de las coordenadas en "y" y "z" con un "-".

```
aux3[0] = aux3[0];
aux3[1] = -aux3[1]+180;
aux3[2] = -aux3[2];
```

El resultado final de todo esto es que ahora se puede hacer captura de movimiento con la kinect y verlo en tiempo real en unity de manera fluida.

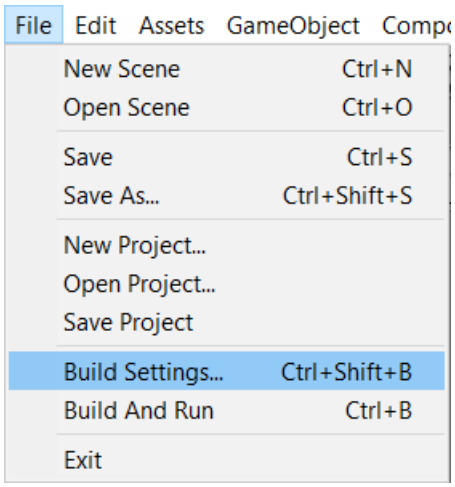
A pesar de que funcione, hay algunas imperfecciones de la captura, algunas debido a que estoy utilizando la kinectV1, como por ejemplo, cuando se cruzan los brazos por la parte del torso. También hay algunos fallos con el modelo en si, sobre todo en las piernas, y en los hombros. Otros fallos pueden ser por la forma en la que lo he programado.

Oculus Rift S con Unity

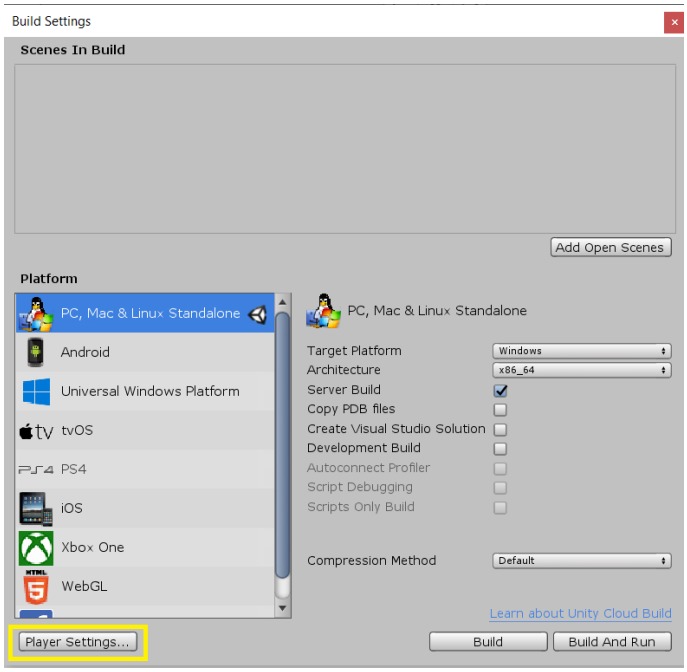
Para utilizar las Oculus Rift S en unity, he utilizado la aplicación de Oculus y la librería de Oculus en Unity.

Para conectar las Oculus al PC hay que conectar 2 cables, un cable USB 3.0 y un cable Display que hay que conectarlo a la tarjeta gráfica. Una vez conectado, la aplicación automáticamente reconoce las gafas y te pide configurarlas si no lo estaban. En el caso de que haya que configurarlas, hay que delimitar la ubicación del suelo y dibujar una zona donde supuestamente el usuario se va a mover, para en el caso en el que se salga de la zona, de de un aviso.

Dentro de Unity, para habilitar la realidad virtual, hay que abrir la ventana de "Build Settings".

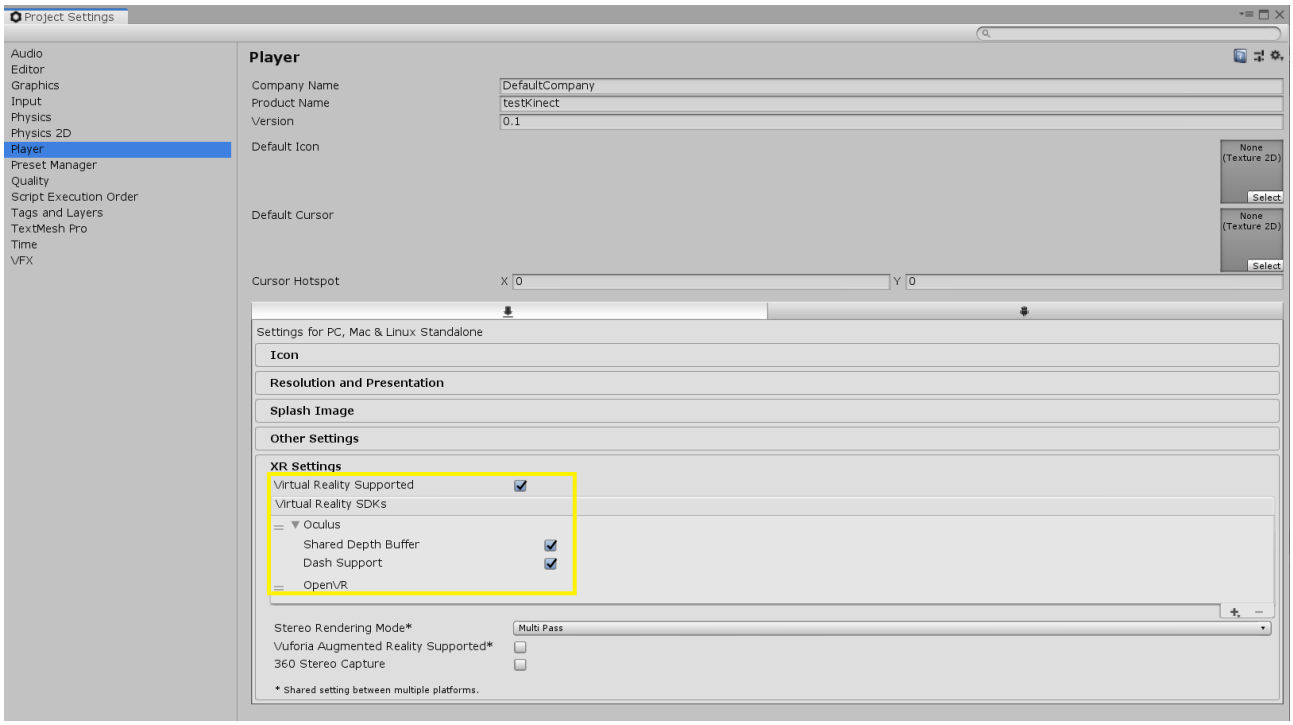


Dentro de esa ventana, hay que abrir la ventana de "Player Settings"



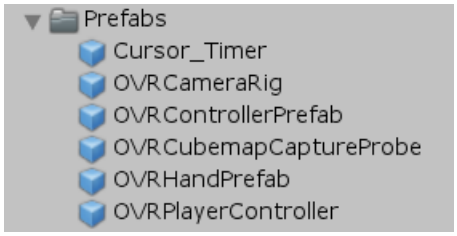
Indice

Dentro de "Player Settings" hay que ir a la pestaña "Player" y en "XR Settings" habilitar el soporte de realidad virtual.



Una vez hecho esto aun sin la librería de Oculus, cuando se ejecute el proyecto, la imagen se verá por las gafas.

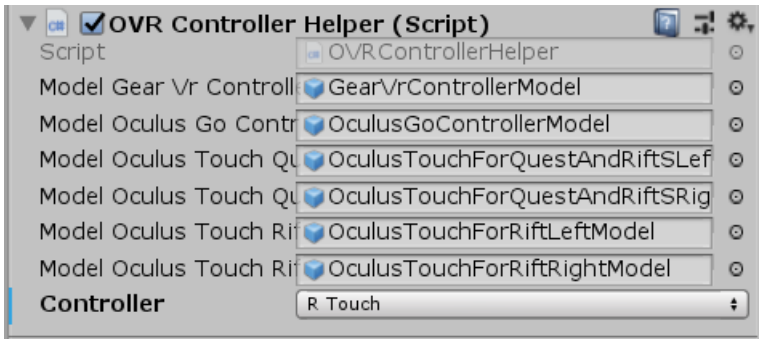
La librería de Oculus permite utilizar los mandos de las gafas en Unity, cuenta con prefabs para añadir un objeto jugador que pueda interactuar con el entorno a través de los controles.



Como objeto jugador utilizo una combinación de "OVRPlayerController" con "OVRController prefab" como modelo de los mandos.

Indice

En el "OVRControllerPrefab" hay que modificar en el script "OVRControllerHelper" el parámetro "Controller" y elegir el que se corresponda, para que se muestre el mando adecuado.



La única modificación que he hecho al "OVRPlayerController" es añadirle un script "CharacterCameraConstraint" que corrige el giro de cámara cuando se mueve el stick analógico del mando derecho para cambiar la orientación. Sin el script cuando se rota la cámara, el personaje también se desplaza y este script que lo he sacado de la propia librería arregla ese problema.

Con todo esto las Oculus Rift S y sus mandos se pueden utilizar en Unity.

Indice

Oculus Quest con el PC y Unity a través de una LAN inalambrica

(No tengo acceso al material para poner imágenes y tampoco puedo especificar la opciones que he tocado para que funcionen las Quest)

Las Oculus Quest, a diferencia de las Rift S, utilizan su propio sistema operativo basado en android. El objetivo de conectar las Quest a Unity es que la persona que está utilizando las Quest pueda ver la posición virtual y las cosas que va dibujando el usuario con las Rift S.

Para hacer que esto funcione, primero hay que crear un proyecto de Unity a parte del principal con funcionalidades reducidas, lo ideal sería que estuviese en el mismo proyecto, pero no es posible porque al compilarlo para android hay librerías y funciones que no se pueden utilizar y que son imprescindibles para algunas funcionalidades, como guardar un dibujo en un prefab, por lo que se ha optado por hacer un proyecto a parte. En el proyecto a de las Quest he hecho ajustes en la ventana de "Build Settings" y "Player Settings" para cambiar la plataforma a android y deshabilitar y habilitar opciones para que el proyecto pueda funcionar en las gafas.

Para comunicar las Quest con el PC, he conectado las Quest y el PC a una misma LAN, de forma que puedan enviar mensajes entre los 2 utilizando mensajes OSC con la librería extOSC, de forma parecida a lo que he hecho entre NiMate y Unity.

Para esto he creado versiones modificadas del script "PainterDiscosPlanosColor" de forma que cuando se dibuje un objeto en el espacio, el script envía las coordenadas del objeto que se ha dibujado con las Rift S a las Quest para que estas dibujen el mismo objeto en la misma ubicación, de la misma forma cuando se junten y unan los objetos, la única excepción es con la propiedad Cloth, en las Quest la propiedad Cloth funciona cuando se instancia durante la ejecución, solo funciona si la propiedad ya está preinstanciada cuando se ejecuta el proyecto.

Por el lado de los mandos, también he creado script para que comuniquen las posiciones de los controles, de forma que el usuario de las Quest pueda ver los controles del usuario de Rift S.

En ambos proyectos he tenido que configurar el "ControladorOSC" con las IPs y los puertos por donde van a enviar y recibir. En el proyecto principal en el "OSCTransmitter" he puesto la IP de las Quest y en el "OSCReceiver" la IP del PC. En el proyecto de las Quest lo he puesto al revés, en "OSCTransmitter" la IP del PC y en el "OSCReceives" la de las Quest.

Tras haber hecho esto, lo ideal sería que el usuario de las Rift S pudiese ver al de las Quest, pero no lo he podido implementar con mensajes OSC, porque a pesar de que recibir los mensajes con las Oculus no me ha dado problemas, no he podido conseguir enviar mensajes al exterior desde las gafas (si que se pueden enviar a ellas mismas, por lo que no creo que sea problema de la librería).

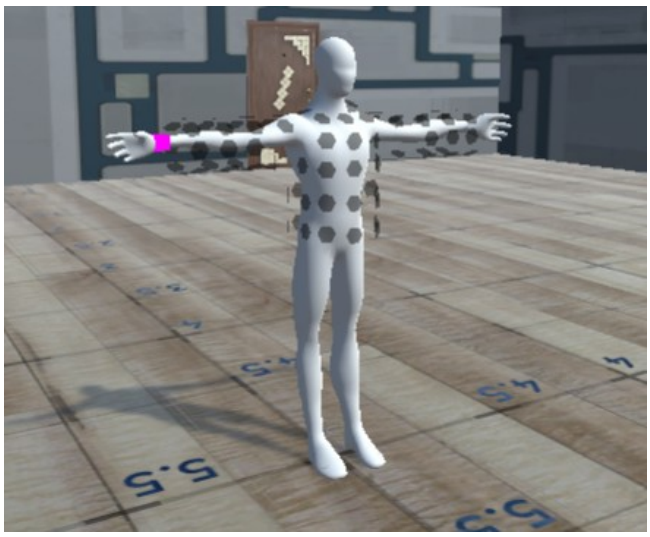
Como estas pruebas las hice la misma semana en la que comenzó el confinamiento no he podido probar con otras librerías.

6. Resultados

Con la aplicación dibujando planos se pueden dibujar prendas de ropa como las siguientes :

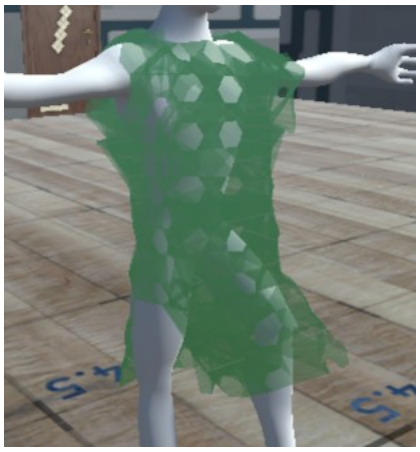
Los primeros ejemplos son de ropa con las físicas del cloth activadas, a la izquierda se puede ver el dibujo hecho con los planos ,a la derecha el resultado de unirlos y generar el cloth y abajo se puede ver como reacciona la ropa al ejecutar una animación.

- Un jersey

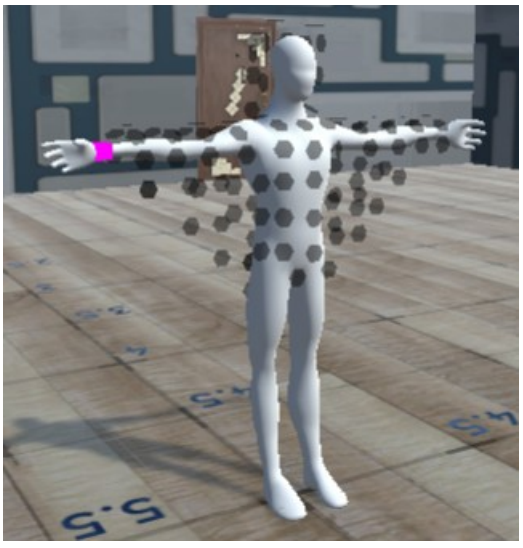


- Un vestido largo





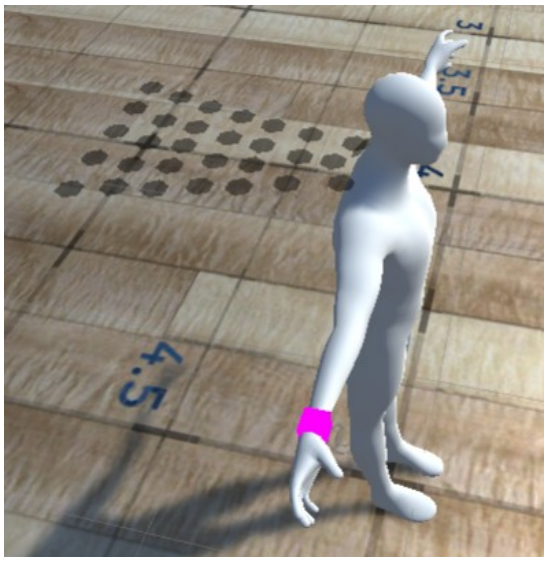
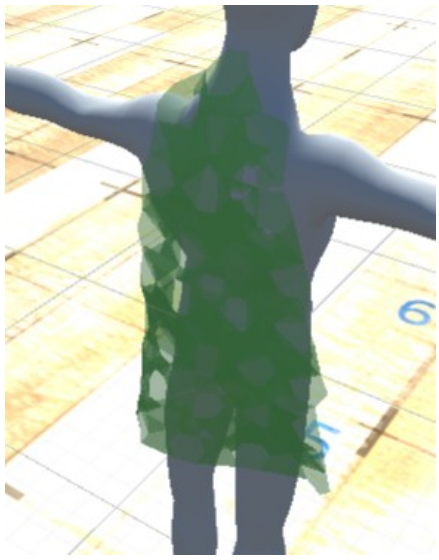
- Un poncho con capucha





Para la capa se ha creado la tela con la opción de engancharla automáticamente al cuerpo, como tiene planos cercanos al cuello, engancha esa parte para que el resto de la tela se quede colgando.

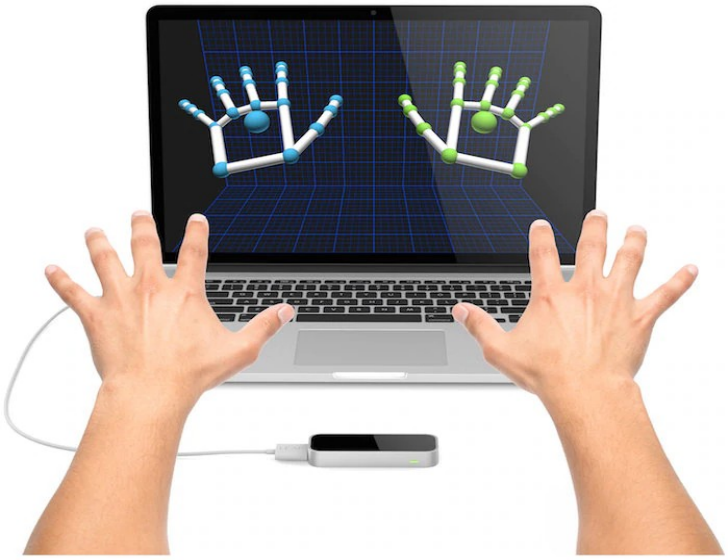
- Una capa enganchada al cuello

7. Otras Pruebas

Leap Motion

Una de las ideas que han quedado descartadas ha sido en utilizar un sensor llamado "Leap Motion", el cual es un sensor que detecta la posición de las manos y los dedos, para crear un objeto físico con el que el diseñador pueda dibujar. El motivo por el que esta idea se descartó es porque no encontramos ningún objeto transparente que se acomodara adecuadamente a las "Leap Motion", la última prueba que hicimos fue poner la "Leap Motion" dentro de un globo semitransparente. Si se inflaba el globo hasta un determinado punto, el sensor podía detectar las manos fuera del globo, pero surgieron otros problemas, porque inflar el globo de forma que el sensor se quedara dentro era demasiado complicado. Al final utilizamos los mandos de las Oculus para pintar.



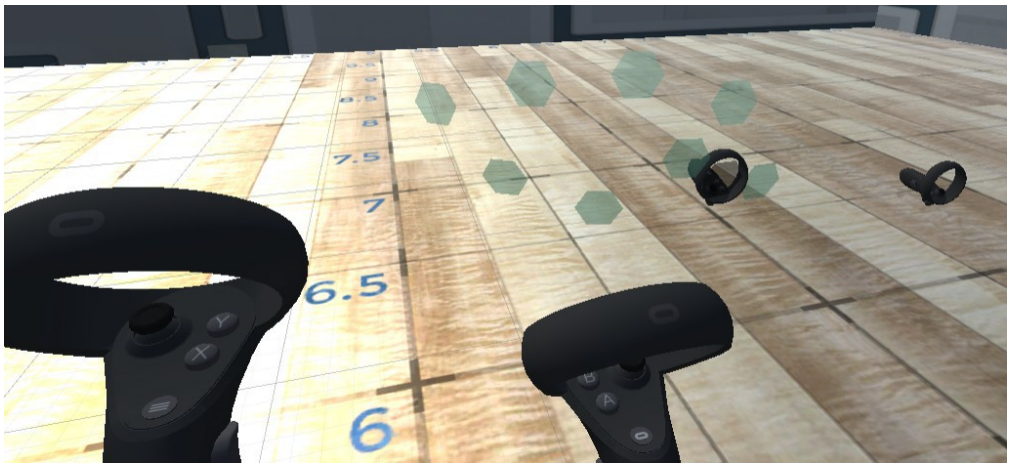
[10]

Co-diseño

La idea de utilizar las Oculus Quest es que otro usuario pueda ponerse las gafas y, a parte de ver qué está dibujando el usuario de las Rift S, también pudiese dibujar objetos con otro color y el usuario de las Rift S lo viese también. Esta idea se ha implementado a medias, porque no he conseguido hacer que las Quest envíen mensajes OSC al ordenador y tampoco he podido hacer más pruebas durante la cuarentena.

Rift S :

Quest :



Indice

8. Referencias

[1] SecondSkin: sketch-based construction of layered 3D models : Chris De Paoli, Karan Singh, https://dl.acm.org/doi/10.1145/2766948, https://www.youtube.com/watch?v=eyaYCdnv9Gg

Volver

[2] Garment modeling with a depth camera : Xiaowu Chen, Bin Zhou, Feixiang Lu, Lin Wang, Lang Bi, Ping Tan, https://dl.acm.org/doi/abs/10.1145/2816795.2818059, https://www.cs.sfu.ca/~pingtan/Papers/sigasia15.pdf

Volver

[3] Imagen e información de la Kinect obtenidas de : https://es.wikipedia.org/wiki/Kinect

Volver

[4] Imagen Oculus Rift S sacada de : https://www.amazon.com/-/es/dp/B07PTMKYS7

Información obtenida de : https://en.wikipedia.org/wiki/Oculus_Rift_S

Volver

[5] Imagen Oculus Quest sacada de : https://www.amazon.es/Oculus-Quest-All-Gaming-Headset/dp/B07P6Y5DNT

Información obtenida de : https://en.wikipedia.org/wiki/Oculus_Quest

Volver

[6] Página oficial de NiMate : https://ni-mate.com/about/

Volver

[7] Página de la librería extOSC: https://assetstore.unity.com/packages/tools/input-management/extosc-open-sound-control-72005

Volver

[8] Página de la librería de Oculus : https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022

Volver

Indice

[9] Script "SerializeMesh" obtenido del usuario "mmankt" de los foros de Unity, https://forum.unity.com/threads/solution-save-a-prefab-with-a-generated-mesh-es-without-creating-assets.463496/

Volver

[10] Imagen Leap Motion sacada de : https://store.roboticsbd.com/robotics-parts/211-leap-motion-controller-robotics-bangladesh.html

Volver