Trabajo de Fin de Grado

Grado en Ingeniería Informática

Sistema de realidad virtual para la delimitación de la copa en imágenes de la cabeza del nervio óptico

Virtual Reality system for the delimitation of the cup in images of the optic nerve head

Daniel Daher Pérez

La Laguna, 1 de Julio de 2017

D. **Rafael Arnay del Arco**, con N.I.F. 78.569.591-G profesor Ayudante Doctor de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Jose sigut Saavedra**, con N.I.F. 43.786.043-T profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A N

Que la presente memoria titulada:

"Sistema de realidad virtual para la delimitación de la copa en imágenes de la cabeza del nervio óptico"

ha sido realizada bajo su dirección por D. **Daniel Daher Pérez**, con N.I.F. 43.831.174-M.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 30 de Junio de 2017.

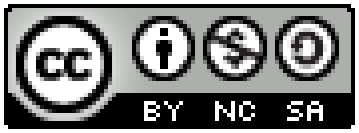
Agradecimientos

Agradecer a mi tutor Rafael por su guía en la realización de este Trabajo de Fin de Grado, su consejo y por siempre responder a mis dudas cuando me han surgido. De la misma forma agradecer también a mi cotutor Jose por dar su sincera opinión durante el desarrollo del trabajo.

A mi pareja, persona sin la cual no estaría terminando a día de hoy mi grado en Ingeniería Informática. Si he sido capaz de llegar hasta aquí en este tiempo y forma ha sido gracias a ti, *gracias* por ser mi apoyo incondicional y por siempre estar a mi lado.

A mi abuelo, por ser la persona que más me ha marcado en mi vida, por siempre confiar en mí y darme ánimos, por darme todo. Este proyecto marca para mí el fin de una etapa en mi vida, y creo que si he podido finalizarla es gracias a ti, espero que estés orgulloso de mí.

Licencia



Resumen

El objetivo de este trabajo es la creación de una aplicación móvil de realidad virtual para ayudar al profesional médico a a detectar una estructura del fondo del ojo conocida como "copa" que puede ayudar en el diagnóstico temprano del glaucoma.

Una forma de diagnosticar el glaucoma es conocer la relación entre dos estructuras del fondo del ojo que se conocen como "copa" y "disco". La delimitación del disco óptico se realiza a través de información puramente visual, ya que suele haber un cambio evidente de color en los bordes de esta estructura. La copa, sin embargo, es una estructura difícilmente distinguible usando información visual, pero que se puede detectar más fácilmente usando información 3D ya que se suele presentar un cambio de profundidad en los bordes de ésta. Este proyecto realizará una representación en realidad virtual del fondo del ojo que se desee inspeccionar para así obtener una retroalimentación imposible de otra forma y que el profesional médico pueda delimitar la copa de una manera eficaz e intuitiva.

Palabras clave: Glaucoma, Copa, Representación 3D, Imágenes estéreo.

Abstract

The aim of this paper is to create a mobile virtual reality application to help the medial professional detect a fundus structure known as "cup" that may aid in the early diagnosis of glaucoma.

One way to diagnose glaucoma is to know the relationship between two structures of the fundus of the eye that are known as "cup" and "disk". The delimitation of the optical disk is done through purely visual information, since there is usually an evident change of color in the edges of the structure. The cup, however, is a structure difficult to distinguish using visual information, but it can be detected more easily using 3D information since there is usually a change of depth in the edges of it. This project will perform a virtual reality representation of the background of the eye to be inspected in order to obtain an otherwise impossible feedback and that the medical professional can delimit the cup in an effective and intuitive way.

Keywords: *Glaucoma, Cup, 3D Representation, Stereo Images*

Índice General

Índice de figuras

Índice de tablas

Capítulo 1. Introducción

1.1 Motivación

El glaucoma es una enfermedad del ojo que se caracteriza por una pérdida gradual de la visión del paciente. Por lo general, no presenta síntomas en la primera fase de la enfermedad y sin el apropiado tratamiento puede llevar a la ceguera. Este proyecto ha surgido con el fin de facilitar la detección de un tipo de estructura en el fondo de ojo que puede ayudar en el diagnóstico temprano de esta enfermedad.

La "copa" y el "disco óptico" son estructuras que se encuentran en la retina. Estableciendo una relación entre la apariencia de estas estructuras, se pueden diagnosticar enfermedades como el glaucoma, incluso en etapas iniciales. La detección del disco óptico es relativamente sencilla, porque visualmente destaca. La copa presenta más dificultades para ser distinguida con pistas visuales. Sin embargo, en los bordes de la copa se produce un cambio en la profundidad del fondo de ojo que puede ser detectado usando información tridimensional.

Este problema es lo que nos lleva precisamente a este proyecto. Mediante herramientas de visualización en tres dimensiones se pretende ayudar de una forma eficaz e intuitiva en la detección del glaucoma.

1.2 Objetivos

Este Trabajo de Fin de Grado tiene como objetivo final la realización de una aplicación Android en Unity 3D para realidad virtual que facilite al profesional médico la delimitación de la "copa" en imágenes estéreo de fondo de ojo. Para llevarlo a cabo contamos con una base de datos de este tipo de imágenes.

La aplicación desarrollada permitirá visualizar en 3D estas imágenes y, a través de un interfaz sencillo, se podrá delimitar el contorno de la copa usando información de profundidad.

Lo que se va a realizar es una representación 3D del fondo de ojo, partiendo de las fotos estéreo ya mencionadas. Utilizando unas gafas de Realidad Virtual, un dispositivo Android con la aplicación y un mando bluetooth (en este caso se ha usado el iPega PG-9025) podremos usar la aplicación.

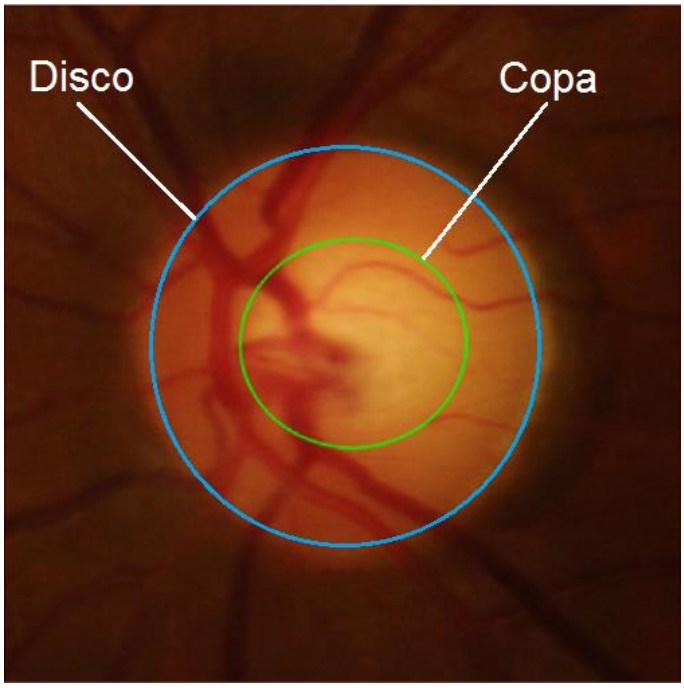


Figura 1. Ejemplo de segmentación de copa y disco

El control absoluto de la aplicación lo tendremos con el mando. Una vez abierta la app podremos movernos por encima de la superficie de la representación 3D, hacer zoom, cambiar de foto o empezar a seleccionar distintos puntos a través del cursor. Este cursor nos permite seleccionar zonas de la superficie donde crearemos un punto, estos puntos delimitan la zona correspondiente a la "copa", puntos que se irán intersectando creando la impresión visual de que están conectados para que sea más intuitivo. Las coordenadas de los píxeles correspondientes a los puntos seleccionados se podrán guardar para ser usadas en otros programas externos que determinen , si el ojo padece glaucoma o no.

Esta aplicación recibirá el nombre de GLFinder3D.

1.3 Estructura de la memoria

Tras haber visto la introducción empezamos con el contenido en sí de la memoria:

- Capítulo 2. En este capítulo hablaremos de las aplicaciones similares a GLFinder3D, las herramientas que hemos usado para su desarrollo y la metodología de trabajo que se ha seguido en el trascurso del proyecto.
- Capítulo 3. Este capítulo es el más extenso y central de la memoria, donde hablaremos del desarrollo de la aplicación y de todas sus fases y etapas. La representación 3D o la conexión de un mando bluetooth son puntos que serán tratados aquí.
- Capítulo 4. Contendrá las conclusiones del proyecto y las líneas futuras de trabajo de la aplicación
- Al final de la memoria podremos ver un presupuesto del proyecto así como la bibliografía correspondiente a este documento

Capítulo 2. Antecedentes y diseño

2.1 Antecedentes

Este campo, y desde el punto de vista que está planteado este TFG, es algo novedoso, por lo que no hay demasiados trabajos similares. Sí existen una serie de proyectos pertenecientes al campo de la prevención del glaucoma, un conjunto de apps para android en los que el usuario puede hacer tests para así, en cierta medida, detectar si padece esta enfermedad.

Algunos de estos ejemplos son:

- **Viewi,** es un dispositivo que, conectado a un Smartphone, puede ayudar a prevenir el glaucoma. No es una suplantación de los sistemas tradicionales de detección del glaucoma, pero se espera que sea un método adicional extendido y con cierta fiabilidad, sobre todo para su uso en países en vías de desarrollo por su bajo coste. [1]
- **VisualfieldsEasy** es una app para iPad que realiza unos pequeños tests visuales que pueden ayudar a prevenir enfermedades oculares, como puede ser el glaucoma. También tiene como objetivo que su uso se propague por poblaciones con acceso limitado a la atención sanitaria. [2]

2.2 Herramientas

Para la realización de este proyecto se ha utilizado como herramienta principal y única el software de desarrollo Unity 3D.

Unity 3D es un motor de videojuego multiplataforma creado por Unity Technologies. Para el entendimiento de la plataforma Unity así como de sus funciones nativas y scripts ha sido fundamental la existencia de la documentación de Unity [4][5]. Debido a su gran capacidad de procesamiento

y su amplio abanico de tecnologías en muchas ocasiones su uso no se limita a la creación de videojuegos como tales, sino como en el caso de este TFG, a la creación de una herramienta que, haciendo uso del 3D, pueda ser de utilidad en otro campo totalmente distinto campo original.

Google tiene a nuestra disposición una librería para el desarrollo de aplicaciones de realidad virtual en dispositivos Android, *Google VR SDK* [3]. El SDK de Google VR contiene una gran cantidad de scripts y *prefabs* (instancias de objetos en Unity) que podremos añadir a nuestro proyecto en Unity para facilitarnos la implementación de la Realidad Virtual en el dispositivo móvil, como puede ser el seguimiento de la cabeza (*head tracking*), añadir nuevas funcionalidades o mejorar ya existentes. Esta librería toma un papel fundamental en este proyecto ya que para poder crear la representación 3D de las fotos estéreo es necesario utilizarla.

2.3 Metodología

Podemos englobar el desarrollo de la app en tres partes diferenciadas. La representación 3D del fondo de ojo, la conexión del mando bluetooth al dispositivo móvil con el cursor integrado y la selección de la máscara o coordenadas de la copa.

Para verlo con más detalle vamos a dividirlo y nombrarlo todo de la siguiente forma:

- **Representación 3D:**
  - Origen fotografías estéreo en 2D
  - ¿Cómo se realiza la representación?
- **Mando bluetooth:**
  - Mapeo de botones y uso
  - Movimiento y zoom
  - Cambio de fotografía
  - Cursor integrado a la app
- **Selección:**
  - Creación de puntos

- Intersección dinámica
- Interacción dinámica de la selección
- Cálculo de coordenadas (píxeles)
- Almacenamiento de coordenadas (píxeles)

Capítulo 3.
Desarrollo del proyecto

Este capítulo será el central de la memoria, ya que aquí es donde se detallarán todas las fases del desarrollo del proyecto que ya se han nombrado en el capítulo anterior.

3.1 Representación 3D

3.1.1 Origen fotografías 2D

Como ya hemos mencionado para realizar la representación partimos de fotografías estéreo, esto son fotografías que están hechas desde dos puntos levemente separados, creando un pequeño *offset* o diferencia, lo que nos permitirá crear el efecto 3D.

Gracias al SDK de Google y a la distorsión nativa de la realidad virtual (distorsión de lentes) nuestro cerebro interpretará como 3D las fotografías en 2D.

3.1.2 Representación

Una vez que sabemos lo que vamos a hacer y la idea detrás de ello... ¿Cómo se lleva a cabo la representación?

El concepto base es el siguiente: para poder ver en 3D, cada ojo tiene que ver sólo la parte que le corresponde de la escena, es decir, el ojo izquierdo tiene que poder ver sólo la parte izquierda de la foto estéreo y el ojo derecho la parte derecha. Para realizar esto empezamos a trabajar en Unity, haciendo uso de las librerías de google.

Lo primero que necesitamos es el prefab *GvrHead*, lo que se correspondería con nuestra 'cabeza' en la aplicación, el punto base desde donde vamos a ver todo. Este prefab tiene incluido el script *GvrHead* que ,normalmente, es el que se encarga del seguimiento de la cabeza pero para este proyecto vamos a

desactivar estas casillas ya que no nos interesa. Lo que sí vamos a añadir es el script *GvrViewer* que es el que genera el efecto de Realidad Virtual a la aplicación. Tendremos que tener siempre activado la opción de *VR Mode Enabled* y la distorsión en *Native*. Teniendo nuestro *GvrHead* lo que necesitamos ahora es una cámara, que será la que nos permitirá ver en nuestra aplicación. Esta cámara no va a tener agregado nada en especial, ya que es la cámara a la que redirigiremos lo que vean nuestros 'ojos'.

Ahora tenemos que crear dos cámaras más, *Camera Left* y *Camera Right*, estas serán las cámaras que visualizarán por separado nuestra foto estéreo. La cámara *Left* la parte izquierda y la cámara *Right* la derecha. Ambas cámaras tendrán que tener desactivado el componente *Camera* ya que no queremos que visualicen nada sino que lo que van a ver se va a "redirigir" a la cámara padre anteriormente creada. Lo que sí hay que añadirles es el script *GvrEye* y cada cámara tendrá que tener seleccionada la opción correspondiente en el campo *Eye* del script (*Left* o *Right*). Ya tenemos la estructura para la visualización de nuestra app:

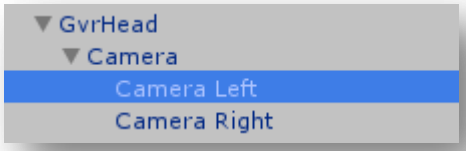


Figura 2. Estructura visualización Unity

Este último script (*GvrEye*) va a tener que ser modificado para hacer que la cámara izquierda sólo vea la parte izquierda y la derecha sólo la derecha, esto tiene que ver con las capas o *Layer* de Unity que se explicará en cuanto se llegue al objeto "Ojo".

La modificación quedaría así:

```
if (eye == GvrViewer.Eye.Left)
    cam.cullingMask = (1 << LayerMask.NameToLayer ("Left") | 1 << 0);
else
    cam.cullingMask = (1 << LayerMask.NameToLayer ("Right") | 1 << 0);
```

Figura 3. Modificación GvrEye

Lo que se realiza aquí simplemente es, mediante un acceso a los componentes de la cámara, 'filtrar' lo que vemos. Lo que hacemos es que el ojo izquierdo sólo tenga permitido ver los objetos con la capa *Left* y el derecho la *Right*. También se añade un último filtro que es la capa con código '0', y esa es la capa por defecto, esto permitirá que se pueda ver tanto el cursor como los puntos que se instancien más adelante.

Por último, tenemos que crear los objetos que tendrá nuestra imagen, los objetos 'Ojo Izquierdo' y 'Ojo derecho', que no son más que dos *gameObject* (instancia de cualquier cosa en Unity) que renderizarán en 3D nuestras imágenes en estéreo, cada uno su parte correspondiente. La forma en la que estos gameObject 'Ojo', seleccionan la imagen, la 'dividen' por la mitad y la renderizan de forma automática es mediante un script que veremos en un punto más adelante.

A cada uno de los objetos "Ojo" hay que ponerles los atributos *Layer* que nos interesan en el editor de Unity, *Left* y *Right*. Esto viene a ser como una etiqueta para poder referenciar a estos objetos. Esta misma etiqueta es lo que tenemos en cuenta cuando les decimos mediante el script *GvrEye* a las cámaras qué es lo que queremos que nos muestren. De esta forma podemos poner en un mismo punto en el espacio las dos imágenes. Cuando estamos usando la app, en cada 'ojo' solo vemos la parte correspondiente a ese ojo, y existe un offset con el que se sacaron las imágenes que 'simula' la diferencia de distancia entre nuestros ojos, y finalmente la distorsión nativa de la realidad virtual con las lentes de nuestras gafas de realidad virtual. Todo esto completa la representación 3D, ya que para nosotros estamos viendo un objeto 3D en un mundo 3D.

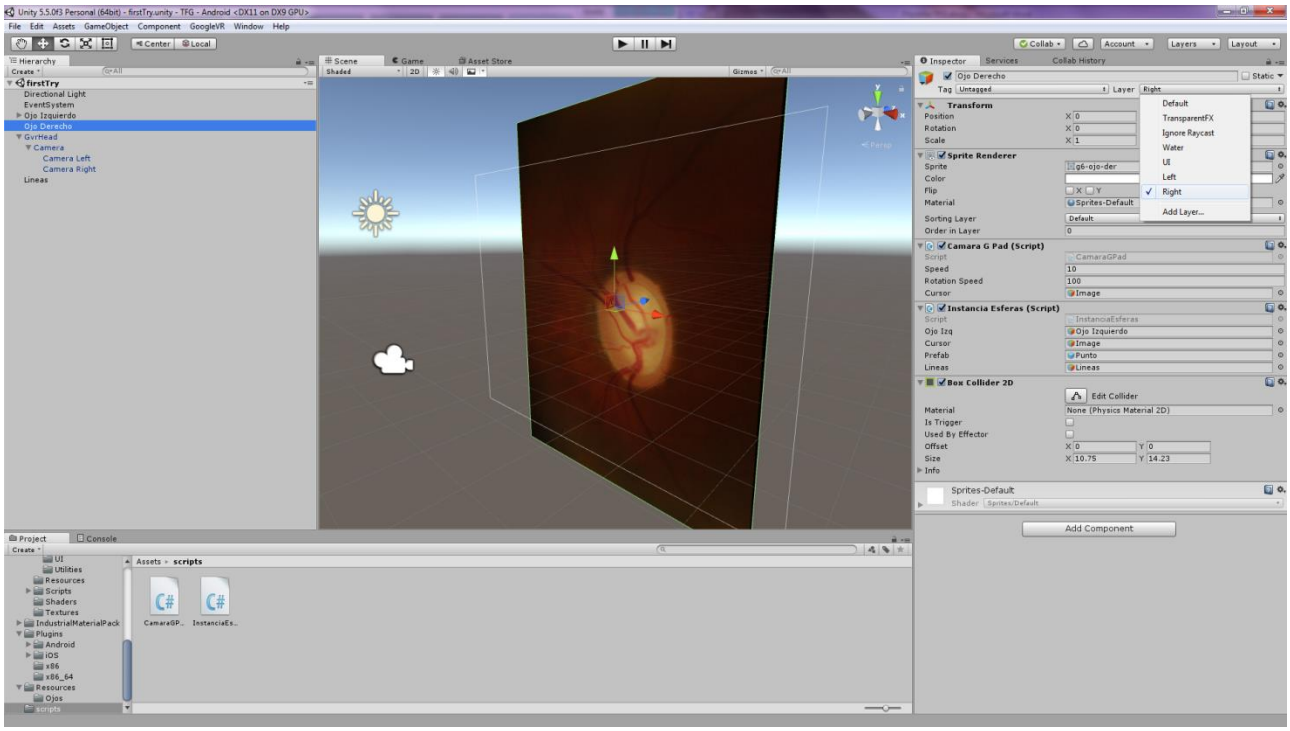


Figura 4. Editor de Unity mostrando el atributo *Layer* de un objeto Ojo

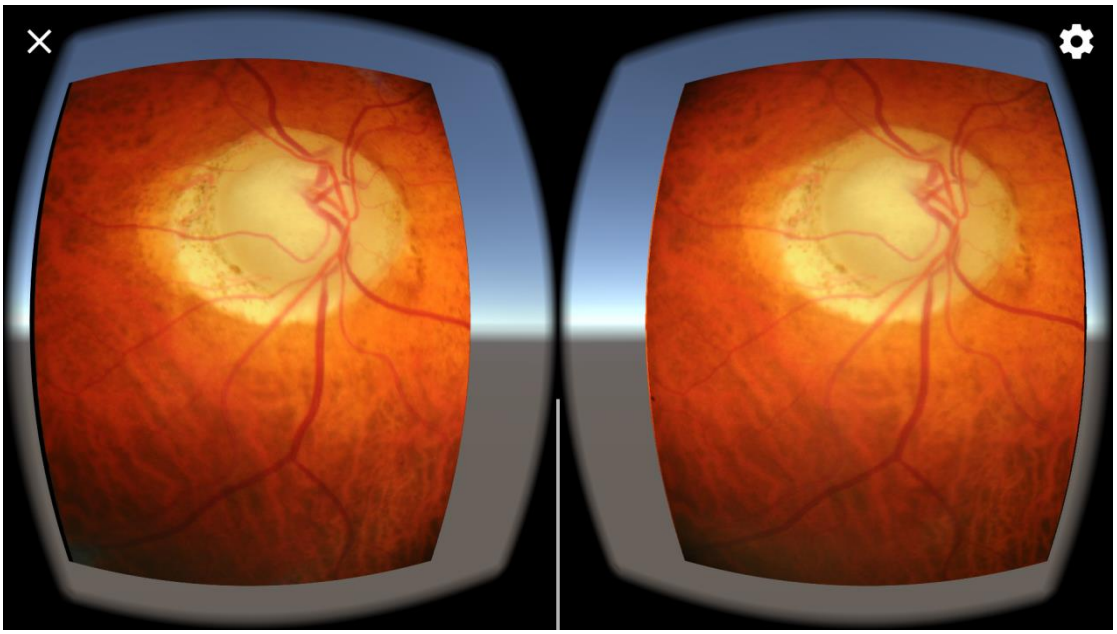


Figura 5. Representación 3D en dispositivo Android. (Sólo representación)

3.2 Mando bluetooth

3.2.1 Mapeo de botones y uso

Vamos a utilizar un mando bluetooth para manejar la aplicación. Como ya se mencionó, para este proyecto se ha usado el iPega PG-9025.

El problema de usar un mando que puede no ser el predefinido (Xbox Controller for Windows) como en este caso, es que Unity no sabe qué entrada corresponde a qué botón, es decir, no tenemos el mapeado de botones hecho, por lo que tenemos que, mediante prueba y error, ir descubriendo qué entrada corresponde qué botón del mando. Se pueden crear 'input' para la app desde Edit/Project Settings/Input, hasta que se encuentren los deseados. Después de esta investigación el mapeado del mando con los botones que vamos a utilizar quedaría así:

Código Unity/ Axis	Botón Mando	Función
X axis	Joystick izq. Eje X	Desplazamiento eje X
Y axis	Joystick izq. Eje Y	Desplazamiento eje Y
3rd axis	Joystick der. Eje X	Movimiento del cursor. Eje X
4th Axis	Joystick der. Eje Y	Movimiento del cursor. Eje Y
joystick button 5 / joystick button 4	Gatillos izquierdo y derecho	Ampliar, alejar
joystick button 7	Gatillo trasero der.	Siguiente fotografía
joystick button 6	Gatillo trasero izq.	Anterior fotografía
joystick button 0	Botón A	Insertar / seleccionar punto
joystick button 1	Botón B	Aceptar cambios y salir selec.
Joystick button 10	Botón Start	Guardar las coordenadas en fichero

Tabla 3.1. Mapeado de botones en Unity

Como vemos en la tabla, las dos primeras filas tienen como código X axis e Y axis, esto es porque los ejes X e Y del joystick primario sí que los detecta Unity sin especificar un código en sí, sino especificando el *Axis*. Para el resto de botones no basta con especificar el código 'joystick button x' correspondiente, sino también el eje *(Axis)* que, como ya mencionamos, es un campo del Input de Unity, un input X en un axis Y no tiene por qué ser el mismo input X en un axis Z. Aunque no estoy seguro del límite de este atributo, en esta app por ejemplo, en este mando los axis 3 y 4 pertenecen a los ejes horizontal y vertical del joystick derecho y el axis 5 a los botones.

3.2.2 Movimiento y zoom

Después de ver la tabla 1 ya sabemos que podemos desplazarnos por la superficie de la representación 3D con el joystick izquierdo del mando y podemos hacer zoom con los gatillos izquierdo y derecho, esto se realiza gracias al script *CamaraGPad*.

Este script es muy sencillo, lo primero que hace es encargarse de obtener los valores de los joysticks/gatillos pulsados:

```
float zoom = Input.GetAxis("Acercar") * speed;      // Alejar / acercar
float posX = Input.GetAxis("Horizontal") * speed;  // Horizontal // joystick izq
float posV = Input.GetAxis("Vertical") * speed;    // vertical
```

Figura 6. Movimiento y zoom

Una vez obtenidos los valores de movimiento y zoom podemos modificar el objeto acorde:

```
if (Input.GetKeyDown (KeyCode.JoystickButton4)) {
    if (zoom < cero)
        zoom *= -1.0f;
    transform.Translate (posX, posV, zoom);
}
else
if (Input.GetKeyDown (KeyCode.JoystickButton5)) {
    if (zoom > cero)
        zoom *= -1.0f;
    transform.Translate (posX, posV, zoom);
}
else
    transform.Translate(posX, posV, cero);
```

Figura 7. Actualización transform. Mov. Zoom

Todo esto siempre dentro de la función Update() del script ya que es la función que se llama automáticamente cada frame y nos permite iterar en el tiempo.

3.2.3 Cursor integrado

Para el manejo de la app se ha creado un *cursor virtual* que podremos manejar con el mando, para así facilitar la selección y marcado de puntos. Hablando en términos de Unity, este cursor es una 'UI Image' cuyo movimiento estará ligado al joystick derecho de nuestro mando. Este objeto es 'hijo' del objeto Ojo Izquierdo (es indistinto si el izquierdo o derecho) para que sus coordenadas siempre sean en función del padre. Si se realiza zoom el cursor tiene que ir ligado a los ojos, de otra forma el cursor podría quedar detrás del objeto ojo y desaparecer.

La forma de modificar el movimiento del objeto es igual al descrito para el movimiento y el zoom, se guardan los valores obtenidos de la entrada del joystick derecho y se va modificando la posición 3D del cursor acorde a dichos valores.

3.2.4 Cambio de fotografía

Para llevar a cabo el cambio de fotografía en tiempo de ejecución es necesario haber precargado primero todas las fotografías en nuestro código de una manera que indicamos a Unity que lo que hemos cargado va a ser utilizado en el futuro. Esto se realiza de la siguiente forma:

1. Crear la carpeta *Resources* dentro de la sección (*Assets*).
2. Incluir en la carpeta ya creada todas las fotografías estéreo que deseemos utilizar (mayor el número de fotografías, mayor el peso y aumento del tiempo de carga de la app), en este caso se han ubicado en *Resources/Ojos*. En este proyecto se han utilizado 20 fotografías estéreo distintas a modo de prueba.
3. Aunque no es obligatorio, ya que se puede realizar mediante código, para evitar errores de visualización en Android, se ha ido una por una seleccionando las fotografías y modificándolas con los detalles que deseados.
    a. En tipo de textura se elige *Sprite 2D*
    b. En *Advanced/* se activa la casilla *Read/Write Enabled*
    c. En *Filter Mode* se elige *Point (no filter)*

d. Se selecciona sobrescribir las opciones para Android y se elige en formato *RGB 24 bit*

e. Se aplican los cambios

4. Una vez ya tenemos las fotografías y con las opciones deseadas, las cargaremos en un array dentro de nuestro código con la función *Resources.LoadAll<Sprite>("Ojos")* que cargará todos los sprites dentro de la carpeta Ojos.

5. No hay que olvidar que estas fotos son fotos estéreo, es como decir que en realidad son una foto que hay que dividir a la mitad. Para esto tenemos que crear una textura distinta para cada mitad, eligiendo las coordenadas que nos interesa coger de la textura base que es la foto, para sí poder aplicar estas nuevas texturas a los objetos (sprites) ojos correspondientes. Un ejemplo en código realizando el cambio al ojo izquierdo, teniendo en cuenta que *source* es la textura estéreo a la que queremos cambiar, sería el siguiente:

```
int xIzq = 0;
int yIzq = 0;
int widthIzq = Mathf.FloorToInt (source.width / 2);
int heightIzq = Mathf.FloorToInt (source.height);

Color[] pixelsIzq = source.GetPixels (xIzq, yIzq, widthIzq, heightIzq);

Texture2D texIzq = new Texture2D (widthIzq, heightIzq);
texIzq.SetPixels (pixelsIzq);
texIzq.Apply ();

ojoIzq.GetComponent<SpriteRenderer>().sprite =
    Sprite.Create (texIzq, new Rect (0, 0.0f, texIzq.width, texIzq.height),
        new Vector2 (0.5f, 0.5f), 100.0f);
```

Figura 8. Cambiar textura ojo izquierdo

En la Figura 8 podemos ver que lo primero que hacemos es crear los vértices del "rectángulo" que vamos a seleccionar como textura de la foto estéreo elegida. En este caso como nos interesa el ojo izquierdo vamos a seleccionar un recuadro que vaya desde el pixel (0, 0), origen de coordenadas arriba a la izquierda, hasta un ancho de la mitad del ancho total (primera mitad de la imagen) y con un alto completo. Luego lo que

hacemos es crear un vector que contendrá todos los valores de los píxeles definidos y se lo vamos a aplicar a nuestra nueva textura. Lo único que queda es actualizar el sprite actual (imagen) del ojo izquierdo con un nuevo sprite aplicándole la textura correspondiente a la primera mitad de la imagen estéreo. Habría que seguir un proceso similar para el ojo derecho.

3.3 Selección

3.3.1 Creación de puntos

Tenemos que obtener el contorno de la copa del ojo. Para realizar esto vamos a ir instanciando pequeñas esferas por nuestra representación 3D gracias al cursor virtual.

Moveremos, con el joystick derecho, el cursor hasta el punto que deseemos marcar como primer punto del contorno y, pulsando el botón A, instanciaremos la esfera. Iremos creando instancias de puntos sobre nuestra superficie, tantas como queramos.

La instancia de puntos, así como otras funciones que veremos a continuación, se realizan gracias al script *Seleccion*. Este script es un componente del gameObject 'Ojo Derecho'. Es indiferente a qué ojo lo añadimos.

El concepto en sí detrás de la instancia de los puntos sería el siguiente:

1. *Seleccion* lo primero que tiene que hacer es esperar a que se pulse el botón A en el mando, en el momento en que el botón sea pulsado proseguirá el algoritmo.
2. Acceder a la cámara principal de nuestro proyecto es lo primero, la cámara que se encarga de recoger lo que ven las cámaras 'ojos'.
3. Una vez obtenida la cámara, a través de la posición en ese momento del cursor virtual, va a ser lanzado un 'rayo' desde la cámara hacia el GameObject Ojo Derecho y cuando chocamos contra él paramos y calculamos las coordenadas (píxeles) del punto de colisión.

Especificando en el código:

1. Todo lo que realicemos será en la función Update() ya que queremos que se pueda hacer durante todo el transcurso de la aplicación. Aquí será donde pondremos un condicional de sólo proceder a la instancia de puntos si el botón A del mando ha sido pulsado, esto se consigue con un *Input.GetButton("A")* que devuelve *true* si el botón ha sido pulsado.
2. Vamos a proceder a realizar el 'rayo' a través de la cámara principal (*Camera.main*). Esto se realiza con la siguiente línea:

```
RaycastHit2D hit = Physics2D.GetRayIntersection(Camera.main.ScreenPointToRay(
        Camera.main.WorldToScreenPoint(cursor.GetComponent<RectTransform>().position)));
```

Figura 9. RaycastHit2D

Esta línea de código se encarga de lo yamencionado: lanza un rayo con la función *GetRayIntersection* desde la cámara principal, con punto de salida la posición del cursor en la pantalla, hasta el objeto Ojo para quedarnos con el objeto *hit* con el que hemos chocado. El parámetro que se le pasa es justo la posición nombrada.

   a. Para que el 'rayo' pueda detectar que choca contra el objeto Ojo tenemos que añadir a dicho objeto el componente *Box Collider 2D*. Para este proyecto se ha decidido añadir el componente al GameObject Ojo Derecho. Hay que tener en cuenta que sólo se añade a uno de los objetos Ojos porque sólo queremos una instancia de los puntos, si no se nos duplicaría todo.

3. Ya tenemos el objeto con el que nos hemos 'chocado', el objeto *hit* de tipo *RaycastHit2D*. El objeto *hit* tiene todas las opciones de cualquier otro objeto y además añade las coordenadas de mundo donde se ha cometido el choque. Lo que vamos a realizar es, mediante la función *Instantiate*, una instancia de un prefab 'esfera', que hemos pasado al script desde el editor de Unity, en la posición donde se ha producido la colisión.

3.3.2 Intersección dinámica

Desde el momento que tenemos dos puntos en nuestra representación podremos ver que existe una fina línea que los une y en el caso de añadir un tercero podemos ver que el tercer punto se conecta con el primero. Esta unión entre los puntos tiene como finalidad dar una apreciación visual de la máscara que estamos creando. Cuando hayamos creado todos los puntos deseados alrededor de nuestra selección de la copa, podemos apreciar visualmente cómo se unen los puntos que aunque no sea una máscara totalmente circular, nos da información acerca de cómo nos está quedando la selección.

Esto se realiza gracias a un GameObject que tiene como componente un *Line Renderer* , elemento que se encarga, dados una serie de puntos, de formar una recta en el espacio que pasa por dichos puntos. Para conseguir la unión entre objetos lo que hacemos es pasarle, mediante el script *Seleccion,* al *Line Renderer,* los puntos donde se van instanciando las esferas y aumentando en uno el atributo del componente *Line Renderer* correspondiente al número de tramos que tiene la línea.

La parte de código correspondiente al dibujo de las líneas sería el siguiente:

```
void pintarLineas() {

    contador = 0;

    for (int i = 0; i < instancias.Count; i++) {
        if (i == 0)
            primerPunto = instancias [0].transform.position;
        lineas.GetComponent<LineRenderer> ().SetPosition (i, instancias[i].transform.position);
        instancias [i].transform.hasChanged = false;
        contador = i;
    }

    lineas.GetComponent<LineRenderer> ().SetPosition (contador + 1, primerPunto);

}
```

Figura 10. Pintar líneas unión

Lo que aquí realizamos es un bucle en el que recorremos el array de todas las instancias de puntos creadas, almacenamos siempre el primer punto para conectarlo con el último, y vamos conectándolos entre sí. Al finalizar el bucle lo único que nos queda por hacer es conectar el último punto con el primero. Este método tiene que estar por separado ya que tendremos que llamarlo

cuando estemos modificando en tiempo de ejecución los puntos ya creados, esto lo veremos en el siguiente punto.

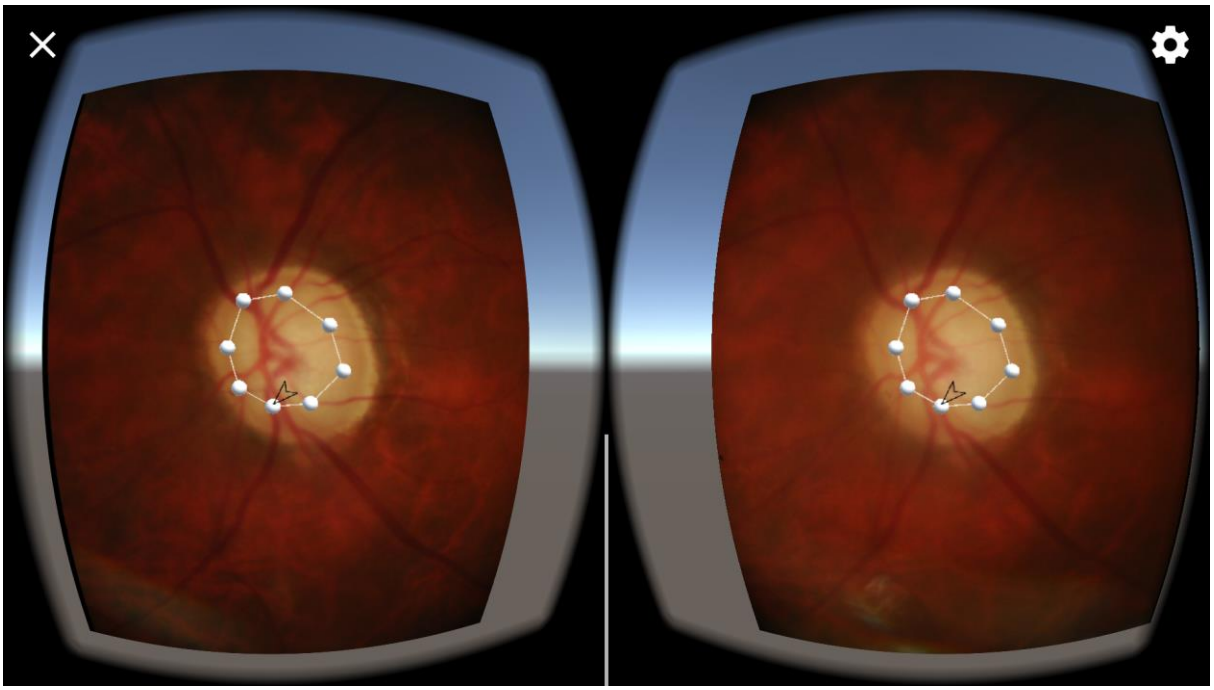


Figura 11. Puntos e intersección en tiempo de ejecución

3.3.3 Interacción dinámica de la selección

La aplicación nos permite, una vez ya creado un cierto punto, modificar su posición seleccionándolo. Lo único que tenemos que hacer es seleccionar un punto ya creado con el cursor de la misma forma que creamos uno nuevo, con el botón A. Una vez seleccionado un punto, veremos que desaparece nuestro cursor y se ilumina el punto que habíamos seleccionado, esto es entrar en el modo 'selección'. Ya en modo selección seremos capaces de mover por toda nuestra representación, con el joystick derecho, la esfera en cuestión. Cuando hayamos movido el punto a la posición deseada sólo tendremos que pulsar el botón B para aceptar los cambios y salir del modo selección, esto volverá a aparecer el cursor y apagar el punto, dejándonos como estábamos antes del modo selección.

Lo primero que nos viene a la mente es el cómo diferenciar entre insertar un nuevo punto y seleccionar uno ya existente. Al igual que el objeto Ojo tiene un componente *Box Collider 2D* como ya nombramos, el prefab del que estamos creando instancias cuando creamos un nuevo punto también tiene el mismo componente, convirtiéndolo en otro objeto posible de colisión.

La condición principal para poder crear una instancia de un punto es que el nombre del objeto tipo *RaycastHit2D* que llamamos *hit* anteriormente no sea 'Punto(clone)', ya que este es el nombre que cogen todas las instancias del prefab (cuyo nombre es Punto). Por lo tanto ya podemos sacar la condición de saber si estamos chocando contra un punto ya existente o no, esto simplemente es un condicional con la sentencia *hit.collider.name == "Punto(Clone)"*. Una vez detectamos que hemos colisionado con una instancia de punto procederemos a poner la variable booleana *modoSeleccion* a *true* lo que para el script quiere decir que entramos en modo selección y no permite acceder a ninguna parte del código sino a la correspondiente a este modo.

La parte central del código del modo selección es la siguiente:

```
if (modoSeleccion) { // hemos seleccionado una esfera

    if (primeraSeleccion) {
        cursor.SetActive (false);

        // activar halo (brillo)
        Behaviour halo = (Behaviour)seleccion.GetComponent ("Halo");
        halo.enabled = true;

        // deshabilitamos el movimiento de la camara/objeto
        ojoIzq.GetComponent<CamaraGPad> ().enabled = false;
        this.GetComponent<CamaraGPad> ().enabled = false;

        primeraSeleccion = false;
    }

    float movX = Input.GetAxis("RH");              // joystick derecho
    float movV = Input.GetAxis("RV");              // para cursor

    movX *= Time.deltaTime;
    movV *= Time.deltaTime;

    Vector3 movCursor = new Vector3 (movX, movV, 0.0f);

    seleccion.GetComponent<Transform> ().position += movCursor;
```

Figura 12. Modo selección

- Lo primero que vemos es que modificamos el atributo correspondiente a si un GameObject está activo en la escena o no del cursor como *false,* lo que lo desactiva.
- El prefab que hace de punto tiene un componente adicional del que no hemos hablado, y ese es el *Halo.* El *Halo* es el brillo que ya hemos nombrado, es un brillo que aparece alrededor del punto cuando lo seleccionamos. Este prefab aunque tenga este componente *Halo*, de forma predefinida lo tenemos desactivado, y lo que vamos a realizar aquí es activarlo ya que hemos entrado en el modo selección. *Aquí el objeto llamado 'selección' referencia a la instancia que hemos seleccionado.*
- Sólo nos queda desactivar los scripts *CameraGPad* encargados del movimiento del objeto, ya que no nos interesa que se puedan mover los objetos Ojo o que se modifique el valor de la posición del cursor, además de que vamos a utilizar para mover el prefab el joystick derecho encargado del movimiento del cursor.
- Este movimiento es lo único que queda, almacenamos los valores de entrada de los dos ejes del joystick derecho y modificamos la posición del prefab seleccionado acorde a dichos valores.
- El resto del algoritmo se encarga de esperar la entrada del botón B para aceptar los cambios y lo que hace es activar lo que hemos desactivado en la primera parte del código y poner a *false* la variable booleana *modoSeleccion* para salir del modo selección y volver al modo inserción normal.

Durante el modo selección estamos modificando en tiempo de ejecución la posición de los puntos, pero las líneas que los unen también se modifican en tiempo de ejecución. Esto es gracias a una pequeña comprobación que se realiza al principio de la función Update() que comprueba si los puntos han cambiado de posición respecto a la anterior comprobación, y se debe a la existencia del atributo *gameObject.transform.hasChanged.* Este atributo lo posee todo gameObject como variable booleana y sólo está a *true* cuando el valor ha cambiado desde su origen o desde que se puso a *false.* La comprobación sería la siguiente:

```
for (int i = 0; i < instancias.Count; i++) {
    if (instancias [i].transform.hasChanged) {
        pintarLineas ();
        break;
    }
}
```

Figura 13. Comprobación de posiciones para líneas

3.3.4 Cálculo de coordenadas

El cálculo de coordenadas (píxeles) van de la mano, ya que ambos procedimientos se realizan dentro de la función *SaveOnFile()*. Esta función se llama cuando se pulsa el botón Start en el mando.

Lo que hacemos en esta función es recorrer todos las instancias que hemos creado (puntos), almacenando sus posiciones absolutas en el mundo. Una vez obtenidas estas posiciones de mundo vamos a realizar unas operaciones matemáticas para calcular a qué pixel corresponde en la imagen dichas posiciones.

```
foreach (Transform child in transform) {
    rend = transform.GetComponent<SpriteRenderer> ();
    tex = rend.sprite.texture as Texture2D;

    pixelWidth = rend.sprite.rect.width;
    pixelHeight = rend.sprite.rect.height;
    unitsToPixels = pixelWidth / rend.bounds.size.x * transform.localScale.x;

    screenPos = new Vector3 (child.position.x, child.position.y, transform.position.z);
    screenPos = transform.InverseTransformPoint (screenPos);

    int xPos = Mathf.RoundToInt (screenPos.x * unitsToPixels) +
        Mathf.RoundToInt (tex.width * 0.5f);
    int yPos = tex.height - (Mathf.RoundToInt (screenPos.y * unitsToPixels) +
        Mathf.RoundToInt (tex.height * 0.5f));

    string coords = "X: " + xPos + " Y: " + yPos;

    coordenadas.Add (coords);
}
```

Figura 14. Calcular coordenadas (píxeles)

Para que todos los puntos se muevan siempre junto a los objetos Ojo, por ejemplo haciendo zoom, todos los puntos cuando se instanciaron se crearon

como hijos del GameObject *Ojo Derecho*. Lo que vamos a realizar es un bucle en el que iteramos sobre todos los GameObjects hijos del *Ojo Derecho* para ir calculando cada una de sus coordenadas en píxeles.

Por cada hijo (punto) vamos a realizar lo siguiente:

- Calcular el ancho y alto del componente *Sprite* del objeto ojo (componente que renderiza la textura)
- Proceder a calcular la relación entre unidades de mundo y píxeles dentro de nuestra textura (x unidades equivalen a 1 pixel)
- Con la función *transform.InverseTransformPoint(posición instancia)* hacemos el cambio desde posiciones de mundo de nuestra instancia a coordenada local en nuestro objeto ojo. Esto es el número de unidades de Unity desde el punto (0, 0), que es el centro del objeto, hasta la posición donde se encuentra la instancia punto en cuestión.
- Ya tenemos la posición local de nuestra instancia y la relación entre las unidades de Unity y los píxeles en nuestra textura, por lo que sólo hay que realizar la multiplicación oportuna para hallar los píxeles de la imagen. Se le suma la mitad del tamaño de la textura porque queremos que nuestro centro de coordenadas sea la esquina superior izquierda y no el centro.

Finalmente hemos calculado la coordenada en píxeles de nuestra textura donde instanciamos un punto, ahora sólo queda almacenar estas coordenadas en una lista (en este caso de *string*) para escribirlos luego en un fichero.

3.3.5 Almacenamiento de coordenadas

Como ya ha sido nombrado en el punto anterior, tanto el cálculo como el almacenamiento de coordenadas se realizan en el método *SaveOnFile()*.

Llegados a este punto tenemos creado una lista de cadenas (*string)* que contiene las cadenas de caracteres que vamos a escribir en un fichero, los valores X e Y de todas las coordenadas que hemos seleccionado en nuestra aplicación.

Cada vez que esta función es llamada la primera línea de código se encarga de crear el fichero donde vamos a escribir todas nuestras coordenadas:

```
file = new FileInfo (Application.persistentDataPath +
    "/" + spritesOjos [indiceSprite].name + ".txt");
```

Figura 15. Ruta fichero

Lo que aquí realizamos es la instancia de un objeto *FileInfo* que es el tipo de objeto que nos permite el Input/Output en C# con Unity. A este objeto hay que pasarle como parámetro,la ruta donde queremos que se cree, con el nombre del fichero en sí. En este caso como nombre de fichero siempre se va a elegir el nombre de la fotografía del ojo actual, por ejemplo *G-1-L* implicando que es un ojo con glaucoma y que es una fotografía estéreo del ojo izquierdo del paciente. La sentencia *Application.persistentDataPath* devuelve la ruta base de la aplicación, en el dispositivo Android devolvería la ruta correspondiente a *Android/data/com.DanielDaher.GLFinder/files* ya que esa es la ruta de instalación y archivos de la aplicación. De esta forma creamos la instancia del objeto *FileInfo* eligiendo como ruta predeterminada la de la aplicación y como nombre el nombre de la fotografía estéreo.

Ya tenemos creado el objeto que tiene la ruta donde se creará el fichero, pero todavía no ha sido creado el fichero en sí, esto lo crea la siguiente parte del algoritmo:

```
StreamWriter w;

if (!file.Exists)
    w = file.CreateText ();
else {

    file.Delete ();
    w = file.CreateText ();
}

foreach (string coordenada in coordenadas) {
    w.WriteLine (coordenada);
}
w.WriteLine ("Número de coords:" + coordenadas.Count);
w.Close ();
```

Figura 16. Creación fichero

1. Creamos un objeto *StreamWriter*. Este objeto será el cual nos almacene el fichero creado, escribir y borrar de él.
2. Comprobamos si el fichero no existe, si es así lo creamos
3. Si actualmente existe lo eliminamos para crearlo de nuevo, vacío.
4. Realizamos un bucle para iterar sobre la lista de coordenadas. Escribimos una línea en el fichero por cada entrada en esta lista.
5. Añadimos una última línea con el número total de coordenadas que hemos creado.
6. Cerramos el objeto *StreamWriter* ya que no vamos a editar más el fichero

Capítulo 4. Conclusiones y líneas futuras

4.1 Conclusiones

El objetivo de este proyecto era crear una herramienta que mediante realidad virtual pudiera facilitar la labor al profesional médico de seleccionar la copa en una imagen de fondo de ojo.

Se ha conseguido realizar una aplicación, mediante la herramienta Unity 3D, de realidad virtual para dispositivos Android. Esta app nos permite visualizar una representación 3D de una serie de fotografías estéreo de fondo de ojo. Mediante un mando bluetooth, se pueden obtener una serie de puntos para crear una delimitación de la copa con el resto del ojo. También podemos modificar estos puntos en tiempo de ejecución y previsualizar una intersección de cómo está quedando la máscara para que nos podamos hacer a la idea del resultado final. Una vez obtenida la máscara nos permite almacenar las coordenadas en píxeles en un fichero, con lo que gracias a herramientas externas, el profesional médico puede saber si el paciente está enfermo de glaucoma o no.

Como conclusión podemos decir que el objetivo de este proyecto se ha conseguido satisfactoriamente, se ha creado una aplicación que ayudará en la detección del glaucoma mediante fotos estéreo de fondo de ojo.

A modo de valoración personal de este Trabajo de Fin de Grado he de decir que siento que este desarrollo ha sido un primer gran proyecto en mi trayectoria como estudiante de Ingeniería Informática. Ha sido el primer trabajo al que le he dedicado tanto esfuerzo y tiempo, he aprendido mucho sobre la realidad virtual, Unity 3D y apps de esta índole en Android. He quedado más que conforme con el resultado final de la aplicación y creo que cumplirá su misión satisfactoriamente.

4.2 Líneas futuras

Con el objetivo de "sacar al mercado" esta app o generalizar su uso hay algunas mejoras principales que se podrían implementar.

Por un lado está el álbum de fotografías. En nuestro caso hemos cargado la aplicación con 20 fotografías estéreo en alta calidad, esto ha hecho que el peso de la *.apk* (instalable de la app en Android) aumentara de manera considerable. Esto se podría solventar vinculando la carga de estas imágenes a una nube por ejemplo, si las fotos las cogiera de un servicio en la nube como puede ser Dropbox o Drive, tanto el peso de la app como el tiempo de carga se mejorarían.

Otro punto a tener en cuenta podría ser el del mando. En el caso de GLFinder3D se ha usado un mando iPega, pero el usuario final podría no disponer de este mando en específico. Se podría automatizar mediante una interfaz el reconocimiento del mando, que salieran todas las opciones en pantalla y que se pidiera ir pulsando las teclas una a una, así con una gran matriz de botones posibles pues ir mapeando todas las entradas como se deseen. Esto es un borrador de la posible solución pero la idea en sí es la misma.

Capítulo 5.
Summary and Conclusions

5.1 Conclusions

The objective of this project was to create a tool using virtual reality that could facilitate the work of the medical professional to select the cup in an eye fundus image.

An application, using the Unity 3D virtual reality tool for Android devices, has been achieved. This app allows us to visualize a 3D representation of a series of eye fundus stereo photographs. Using a bluetooth controlller, you can obtain a series of points to create a boundary of the cup with the rest of the eye. We can also modify these points at runtime and preview an intersection of how the mask is remaining so we can get the idea of the final result. Once obtained the mask allows us to store the coordinates in pixels in a file, so thanks to external tools, the medical professional can know if the patient is ill with glaucoma or not.

As a conclusion we can say that the objective of this project has been successfully achieved, an application has been created that will help in the detection of glaucoma using stereo background photos.

From my point of view about this End of Grade Work, I have to say that I feel that this development has been a first great project in my career as a student of Computer Engineering. It has been the first job that I have dedicated so much effort and time, I have learned a lot about virtual reality, Unity 3D and Android apps of this type. I have been more than satisfied with the final result of the application and I believe that it will fulfill its mission satisfactorily.

5.2 Future lines

In order to "bring to market" this app or generalize its use there are some major improvements that could be implemented.

On one side is the photo album. In our case we loaded the application with 20 stereo photos in high quality, this has made the weight of the *.apk* (installable of the Android app) increased considerably. This could be solved by linking the load of these images to a cloud storage for example, if the photos were taken from a service in the cloud such as Dropbox or Drive, both the weight of the app and the load time would be improved.

Another point to keep in mind might be the controller. In the case of GLFinder3D an iPega controller has been used, but the end user may not have this specific controller. It would be possible to automate through an interface the recognition of the controller, to leave all the options on the screen and to be asked to press the keys one by one, as well as a large array of possible buttons and mapping all the entries as desired. This is a rough draft of the possible solution but the idea itself is the same.

Capítulo 6. Presupuesto

En este capítulo se realizará la estimación del presupuesto del desarrollo de GLFinder3D.

Elementos a tener en cuenta son por ejemplo un dispositivo Android en la que instalar la aplicación, las gafas de realidad virtual, mando bluetooth o las horas de trabajo del desarrollador. Para el desarrollo de este proyecto se utilizó un Samsung Galaxy S5, se tendrá en cuenta un teléfono similar.

En la siguiente tabla observaremos el desglose del presupuesto total por recursos.

Recurso	Precio
Smartphone Android	150€
Gafas de Realidad Virtual	30€
Mando bluetooth para Android	25€
Cuenta desarrollador Google para publicar la aplicación en la Google Play Store	25$/~22€
Tiempo de trabajo de desarrollador: 300h	25€/h

Tabla 6.1. Desglose de presupuesto.

Para todos los recursos relacionados con productos materiales se ha calculado su precio teniendo en cuenta un nivel de calidad de gama media, la suficiente para utilizar la aplicación en óptimas condiciones. Estos precios son bases, eligiendo recursos de mayor calidad el precio aumentará. Se ha detallado también el precio que conllevaría publicar la aplicación en la play store en el caso que en un futuro hiciera.

Con la tabla 6.1 llegamos a la conclusión de que un presupuesto base del desarrollo de GLFinder3D sería de unos 7727€.

Bibliografía

[1] Cambridge Consultants. Visionary Technology, Viewi. 2016.

https://www.cambridgeconsultants.com/media/press-releases/visionary-technology

[2] George Kongo Softwares. visualFields easy. 2015.

http://georgekongsoftwares.weebly.com/

[3] Google. Google VR SDK for Unity. 2017.

https://developers.google.com/vr/unity/

[4] Unity Technologies. Unity user Manual (5.6). 2017.

https://docs.unity3d.com/Manual/index.html

[5] Unity Technologies. Unity Scripting Reference (5.6). 2017.

https://docs.unity3d.com/ScriptReference/index.html