

---

# CAPÍTULO 1

---

## Casos de Uso

### 1.1. Introducción

En este capítulo se presentan los distintos casos de uso que se implementaron con el fin de integrar los algoritmos comentados en capítulos anteriores en pequeñas aplicaciones que funcionen *de punta a punta*. Se buscó resolver individualmente los diferentes desafíos técnicos que una aplicación real de realidad aumentada para museos puede llegar a tener. Estas últimas no serán más que una combinación guionada de cada uno de estos casos de uso.

A lo cargo del capítulo se verán entonces los tres casos de uso implementados: “interactividad”, “video” y “modelos”. El primero presenta un modelo simple sobre el marcador que responde a toques con cierto movimiento y un audio en particular, el segundo soluciona el problema de proyectar un video sobre el marcador de forma consistente con el movimiento del usuario. El último caso de uso muestra cómo es posible importar modelos a ISGL3D de manera de lograr realidades aumentadas mucho más interesantes que si tan sólo se hicieran con las primitivas del *framework*, por detalles ver capítulo ??.

### 1.2. Caso de uso “interactividad”

#### 1.2.1. Comentarios sobre el caso de uso

En este caso de uso se implementa la parte interactiva de la aplicación. Al enfocar el marcador, se puede ver un cubo sobre el QISet de la esquina superior izquierda. Ver Figura 1.1. Si el cubo es tocado a través de la pantalla del dispositivo, este se anima y se reproduce un audio que indica la posición del cubo en el instante de ser presionado. Inmediatamente después, es desplazado hacia el QISet de la esquina superior derecha. Nuevamente, si el cubo es tocado a través de la pantalla del dispositivo, este se anima y se reproduce un audio que indica la posición del cubo en el instante de ser presionado. Inmediatamente después, este se desplaza hacia el QISet restante. Lo anterior sucederá de forma cíclica, cada vez que se presione sobre el mpdelo.

Esta funcionalidad es fundamental si lo que se quiere implementar es por ejemplo una audioguía interactiva. Podría pensarse una aplicación en la que el cubo anterior se reemplace por flechas 3D, y que estas sean ubicadas conjuntamente en distintas partes de una obra. Entones, al seleccionar cada una de las flechas, se podría reproducir un audio con información referente a esa zona o punto en particular.

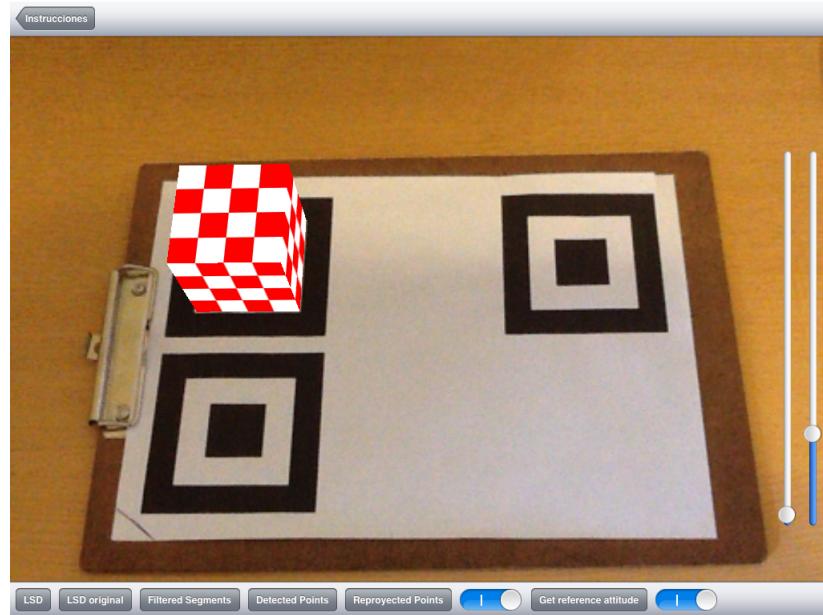


Figura 1.1: Captura de pantalla del caso de uso “interactividad”. Se puede ver al cubo apoyado sobre el QLSet de la esquina superior izquierda y los diferentes controles que ayudan a la depuración del código.

Esta aplicación también se utilizó con fines de *debugging* o depuración de la integración de cada uno de los bloques. Se le agregaron las siguientes funcionalidades:

- La posibilidad de ver dibujados sobre la imagen los segmentos detectados por LSD. En sus versiones original y optimizada.
- La posibilidad de ver dibujados sobre la imagen los segmentos filtrados pertenecientes al marcador. Así como también las esquinas detectadas de cada uno de los cuadriláteros que lo forman.
- La posibilidad de variar el umbral utilizado para el filtrado de segmentos.
- La posibilidad de ver las esquinas de cada uno de los cuadriláteros que forman al marcador reproyectadas según la pose del dispositivo obtenida.
- La posibilidad de prender o apagar el filtro de Kalman.
- La posibilidad de aumentar o disminuir el ruido de medición del filtro de Kalman.
- La posibilidad de elegir si usar o no la fusión de la estimación de pose con los sensores.

En la Figura 1.1 también se puede ver cómo es la interfaz de usuario de este caso de uso, en donde se puede elegir entre todas las funcionalidades anteriores. El mismo fue fundamental para evaluar el desempeño de los algoritmos utilizados funcionando en tiempo real. Gracias a estas funcionalidades se pudieron definir las condiciones para las cuales el conjunto de todos los bloques funciona mejor. Se fue variando la distancia al marcador y se ajustó el umbral para el filtro de segmentos. Además, se pudieron ajustar los parámetros del filtro de Kalman y se pudo comparar el desempeño de la estimación de pose utilizando solamente información de la cámara con el resultado obtenido de la fusión de sensores. Fue en este caso de uso que se evaluó cualitativamente el desempeño de la

versión optimizada de LSD, respecto del de la versión original.

Si bien todas estas pruebas y ajustes si hicieron previamente en una computadora y con imágenes de prueba, fue necesario contar con una aplicación en la que se pudiera ver sobre el dispositivo al conjunto de los algoritmos funcionando en tiempo real.

### 1.2.2. Detalles constructivos

#### 1.2.2.1. Objetos ISGL3D interactivos

La manera de agregar interactividad a un nodo ISGL3D es bastante sencilla. En primer lugar, debe configurarse su propiedad *interactive* de forma positiva y luego se le debe ejecutar el método *addEvent3DListener*:

```
Isgl3dTextureMaterial * material = [Isgl3dTextureMaterial
materialWithTextureFile:@"red_checker.png" shininess:0.9
precision:Isgl3dTexturePrecisionMedium repeatX:NO repeatY:NO];

Isgl3dCube* cubeMesh = [Isgl3dCube meshWithGeometry:60 height:60 depth:60 nx:40 ny:40];

Isgl3dNode * _cubito = [self.scene createNodeWithMesh:cubeMesh andMaterial:material];

_cubito.interactive =YES;

[_cubito addEvent3DListener:self method:@selector(objectTouched:) forEventType:TOUCH_EVENT];
```

En el código anterior, primero se crea un nodo llamado “\_cubito” con la primitiva de un cubo y cierto material. Luego, se indica que sí se quiere que dicho nodo tenga interactividad y finalmente se lo configura para que cuando “\_cubito” reciba eventos del tipo *TOUCH\_EVENT*, o lo que es lo mismo, cuando se lo toque; se ejecute el método *objectTouched*, definido en la misma clase que esta escrita el código (*self*).

En este caso de uso lo que se hizo en *objectTouched* no fue más que cambiar la posición del cubo en la escena y reproducir un audio dependiente de la posición del mismo.

#### 1.2.2.2. Reproducción de audio en Objective-C

Para reproducir audios en Objective-C primero la clase en la que se quiere reproducir el audio debe importar el framework *AVFoundation* y luego debe implementar el protocolo *AVAudioPlayer-Protocol*. El código que se debe escribir es el siguiente:

```
NSURL *url =[NSURL fileURLWithPath:[NSString stringWithFormat:@"%@/ %@",
[[NSBundle mainBundle] resourcePath],audio.mp3]];

AVAudioPlayer * audioPlayer =[[AVAudioPlayer alloc] initWithContentsOfURL:url error:nil];

audioPlayer.numberOfLoops=0;

audioPlayer.delegate = self;

[audioPlayer play];
```

En la primera línea se genera un *url* que indica cuál es el audio a reproducir y luego se le asigna a una instancia de la clase *AVAudioPlayer*. Se dice que no se quiere reproducir el audio en bucle,

se asigna a la clase en la que se esta escribiendo el código como la delegada de *audioPlayer*, una instancia de *AVAudioPlayer*, y finalmente se le da inicio al audio. Luego de reproducido el audio, se ejecuta autamáticamente el método de firma:

```
- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player successfully:(BOOL)flag;
```

En este código es en donde se indica que la próxima vez que se presione sobre el cubo, se querrá reproducir un audio distinto.

### 1.2.2.3. Dibujar en ISGL3D

Lo que se hizo fue crear una clase nueva, del tipo *UIView*, a la que se la llamó “claseDibujar”. Esta fue agregada como *subView* de la *view* en donde se muestra el video por detrás de lo que dibuja ISGL3D. Dicha clase se configuró para que fuera transparente y del mismo tamaño que la pantalla del *iPad*. *claseDibujar* cuenta con una cantidad de propiedades a las que se les asignan los diferentes puntos o segmentos que se quieren dibujar; son del tipo “puntero a entero” y “puntero a *float*” respectivamente. Luego, un método llamado *drawRect* es el que se encarga de dibujar cada uno de los puntos y segmentos. Los puntos se dibujan con las siguientes líneas de código:

```
CGContextRef context = UIGraphicsGetCurrentContext();  
  
CGContextStrokeRect(context, CGRectMake(punto_X, punto_Y, 4, 4));
```

En la primera línea de código se crea un contexto. Un contexto contiene ciertos parámetros y toda la información específica del dispositivo, requerida para poder dibujar. En la segunda línea se dibuja cada punto como un rectángulo centrado en el punto en cuestión y con 4 píxeles de ancho y largo. Los segmentos se dibujan con las siguientes líneas de código:

```
CGContextRef context = UIGraphicsGetCurrentContext();  
  
CGContextStrokeLineSegments(context, puntos, 2);
```

En la primera línea de código se crea un contexto (este paso puede saltarse si ya fue creado anteriormente), y en la segunda se dibuja la línea. La variable “puntos” es un arreglo de dos variables del tipo *CGPoint*, cada una de ellas tiene dos valores en precisión simple correspondientes a las coordenadas de un punto. Además, se le configura al segmento una anchura de 2 píxeles.

Finalmente, es bueno aclarar que *claseDibujar* se instancia y se destruye cuadro a cuadro; el método *drawRect* se invoca cada vez que se instancia la clase.

## 1.3. Caso de uso “video”

### 1.3.1. Comentarios sobre el caso de uso

Este caso de uso proyecta un video sobre el QlSet de la esquina superior izquierda del marcador de manera consistente con la pose del dispositivo. Ver Figura 1.2. La aplicación de esta solución técnica es directa. Tan sólo ajustando un par de parámetros el video podría ser proyectado dentro del marco de un cuadro, sobre uno de sus extremos, sobre una pared blanca o incluso sobre un mapa. Esto puede ser de gran interés para un museo, por ejemplo como complemento a una audioguía. A continuación se explican brevemente algunos detalles técnicos que fue necesario solucionar para lograr implementar este caso de uso.



Figura 1.2: Captura de pantalla del caso de uso “video”. Se puede ver al video proyectado sobre el QlSet de la esquina superior izquierda

### 1.3.2. Detalles constructivos

El desafío técnico es, dados 4 puntos dinámicos cualesquiera sobre la pantalla, poder reproducir un video cuyas esquinas se ajusten a esos 4 puntos. Por lo tanto, en la implementación del presente caso de uso, de toda la lógica de estimación de pose, solamente se hace uso de los bloques de detección, filtrado y determinación de correspondencias. En particular, no se utiliza el algoritmo POSIT visto en el Capítulo ???. Teniendo entonces detectados los límites dentro de los que se quiere reproducir el video, parecería que el problema está resuelto, pues sólo basta con ubicar al video dentro de los mismos. Sin embargo, *Xcode* no permite posicionar en forma directa una *view* (que será la que embeba al video) tomando como límites cuatro puntos cualesquiera.

Si simplemente se quiere reproducir un video, y no se quiere procesar su contenido, la solución más simple es utilizar la clase *MPMoviePlayerController* que hereda de *NSObject*. Una alternativa similar es hacer uso de la clase *MPMoviePlayerViewController* que hereda de *UIViewController* y tiene como única propiedad una del tipo *MPMoviePlayerController*. *MPMoviePlayerController* tiene como atributo una *view*, del tipo *UIView*, y son las esquinas de este atributo a las que se las quiere posicionar sobre 4 de los 36 puntos detectados por el filtro. La clase *UIView* tiene un atributo *frame* que es del tipo *CGRect*:

```
theMovie.view.frame = CGRectMake(0, 0, 60, 60);
```

En el código anterior *theMovie* es del tipo *MPMoviePlayerController*. De esta manera, parecería que los videos solamente pueden ser reproducidos sobre rectángulos y no sobre cualquier cuadrilátero genérico. Sin embargo algo que sí se les puede hacer a las instancias de la clase *UIView* es una transformación afín o incluso, de forma más genérica, una homografía.

### 1.3.3. *CGAffineTransform* y *CATransform3D*

La clase *UIView* tiene una propiedad llamada *transform* que es del tipo *CGAffineTransform*. Las primeras letras de esta clase (*CG*) refieren a la API **Core Graphics** utilizada ampliamente como herramienta para resolver problemas de *rendering* y cualquier tipo de transformación en 2D.

La clase *UIView* además tiene una propiedad llamada *layer* que es del tipo *CALayer* y que permite realizar transformaciones del tipo *CATransform3D*. Las primeras letras de estas dos clases (*CA*) refieren a la API **Core Animation** que es utilizada para generar animaciones y transformaciones sobre objetos 3D solamente indicando un punto inicial y final para el objeto (también es posible agregar efectos para la transición). En definitiva, para resolver el problema del caso de uso existen *a priori* dos alternativas posibles: *CGAffineTransform* y *CATransform3D*.

Se pueden generar fácilmente instancias de transformaciones afines invocando la siguiente función:

```
CGAffineTransform CGAffineTransformMake (
    CGFloat a,
    CGFloat b,
    CGFloat c,
    CGFloat d,
    CGFloat tx,
    CGFloat ty
);
```

que toma 6 *CGFloats* (números reales en coma flotante y precisión simple), y crea una *CGAffineTransform*, donde cada uno de los valores anteriores se corresponde con los elementos de una matriz transformación afín de la siguiente manera:

$$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{pmatrix}$$

Así entonces, de los 9 valores de la matriz, 2 de ellos son nulos por tratarse de una transformación afín y otro de ellos es un factor de valor constante 1. Resolviendo el sistema como se muestra en la sección 1.3.4 y obteniendo los restantes 6 valores, se le puede asignar transformaciones a la propiedad *transform* y realizar la transformación deseada. Sin embargo, una transformación afín preserva colinealidad y realaciones entre distancias; lo que realmente se necesita es una transformación proyectiva. Se decidió entonces estudiar la posibilidad de utilizar una *CATransform3D*.

Una *CATransform3D* se define de la siguiente manera:

```
struct CATransform3D
{
    CGFloat m11, m12, m13, m14;
    CGFloat m21, m22, m23, m24;
    CGFloat m31, m32, m33, m34;
    CGFloat m41, m42, m43, m44;
};

typedef struct CATransform3D;
```

donde  $m_{ij}$  corresponde al elemento de la matriz ubicado en la fila  $i$  y la columna  $j$ . Así entonces, conociendo los valores que debe tomar la homografía, también es posible completar los elementos de esta matriz  $4 \times 4$  y asignarle la transformación a la propiedad *layer* de cualquier objeto del tipo *UIView*. Demás decir que una transformación 3D es algo bastante más genérico que una transformación proyectiva; pero si se anulan algunas entradas de la matriz, es posible lograr el tipo de transformación que se busca. En particular, la coordenada  $z$  debe ser nula:

$$\begin{pmatrix} m_{11} & m_{12} & 0 & m_{14} \\ m_{21} & m_{22} & 0 & m_{24} \\ 0 & 0 & 1 & 0 \\ m_{41} & m_{42} & 0 & m_{44} \end{pmatrix}$$

donde también en este caso se asume el valor unitario para  $m_{44}$  por ser tan sólo un factor de escala. Al igual que para la transformación afín, resolviendo la homografía como se ve en la sección 1.3.4 se obtienen los 8 valores restantes de la matriz.

### 1.3.4. Resolución de Homografía

A continuación se plantea la resolución del sistema de ecuaciones que, dados 4 punto correspondientes entre 2 planos, halla los parámetros de la homografía que los relaciona. Esta transformación proyectiva se puede expresar en forma matricial, en coordenadas homogéneas, de la siguiente manera:

$$\begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} i \\ j \\ k \end{pmatrix}$$

donde la matriz  $h_{3x3}$  representa la transformación homográfica, el vector  $(x, y, z)^t$  representa los puntos de referencia y el vector  $(i, j, k)^t$  respresenta los puntos transformados. Asumiendo un valor unitario para las coordenadas  $z$  y  $k$  la resolución del sistema se simplifica mucho y no se pierde generalidad. Imponiendo esto entonces, el sistema anterior se puede expresar de la siguiente forma:

$$xh_{11} + yh_{12} + h_{13} = i \quad (1.1)$$

$$xh_{21} + yh_{22} + h_{23} = j \quad (1.2)$$

$$xh_{31} + yh_{32} + h_{33} = 1 \quad (1.3)$$

Multiplicando la ecuación (1.3) por  $i$  e igualándola a la ecuación (1.1) se obtiene lo siguiente:

$$xh_{11} + yh_{12} + h_{13} = ixh_{31} + iyh_{32} + ih_{33} \quad (1.4)$$

o lo que es lo mismo:

$$xh_{11} + yh_{12} + h_{13} - ixh_{31} - iyh_{32} - ih_{33} = 0 \quad (1.5)$$

Procediendo de manera análoga y multiplicando la ecuación (1.3) por  $j$  e igualándola a la ecuación (1.2) se obtiene lo siguiente:

$$xh_{21} + yh_{22} + h_{23} = jxh_{31} + jyh_{32} + jh_{33} \quad (1.6)$$

o lo que es lo mismo:

$$xh_{21} + yh_{22} + h_{23} - jxh_{31} - jyh_{32} - jh_{33} = 0 \quad (1.7)$$

Las ecuaciones (1.3) y (1.7) se pueden expresar en forma matricial, de la siguiente manera:

$$\begin{pmatrix} x & y & 1 & 0 & 0 & 0 & -ix & -iy & -i \\ 0 & 0 & 0 & x & y & 1 & -jx & -jy & -j \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Teniendo entonces 4 parejas de puntos referencia y puntos transformados, y asumiendo  $h_{33}$  de valor unitario, se logran 8 ecuaciones con 8 incógnitas. Se tiene ahora un sistema compatible determinado, que puede ser expresado de la siguiente manera:

$$\begin{pmatrix} x_0 & y_0 & 1 & 0 & 0 & 0 & -i_0x_0 & -i_0y_0 \\ 0 & 0 & 0 & x_0 & y_0 & 1 & -j_0x_0 & -j_0y_0 \\ x_1 & y_1 & 1 & 0 & 0 & 0 & -i_1x_1 & -i_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -j_1x_1 & -j_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -i_2x_2 & -i_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -j_2x_2 & -j_2y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -i_3x_3 & -i_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -j_3x_3 & -j_3y_3 \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix} = \begin{pmatrix} i_0 \\ j_0 \\ i_1 \\ j_1 \\ i_2 \\ j_2 \\ i_3 \\ j_3 \end{pmatrix}$$

Finalmente, si los puntos de referencia se definen como 4 puntos del modelo del marcador (en el caso particular de la imagen 1.2, se usan los puntos 4, 5, 6 y 7 del QLSet de la esquina superior izquierda del marcador), y los transformados son sus correspondientes filtrados e identificados cuadro a cuadro en las imágenes capturadas por el dispositivo; es posible transformar en cada instante a la *view* que contiene al video de manera de que se mantenga siempre en dentro de los límites que uno quiere.

## 1.4. Caso de uso “modelos”

### 1.4.1. Comentarios sobre el caso de uso

El presente caso de uso no hace más que importar modelos en ISGL3D de manera de agregarle valor a la realidad aumentada. La aplicación de este caso de uso es casi cualquier ejemplo de realidad aumentada en la que se espere contar con una escena con más que tan sólo primitivas agregadas de forma individual o conjunta. Es importante entonces, contar con una pequeña aplicación piloto para de forma controlada importar los modelos a la escena, ajustar sus tamaños, definir sus posiciones, probar diferentes configuraciones para las luces de la misma y hasta intentar animarlos; para así entonces a la hora de implementar una aplicación final, contar con las herramientas suficientes para que los modelos se vean de la mejor manera posible.

También resulta importante contar con una instancia de prueba para buscar y descargar distintos modelos 3D de internet; incluso puede ser un buen ejercicio probar editarlos, rotarlos o escalarlos en algún *software* de creación y animado de modelos. Finalmente, habrá que llevar a cabo todos los

pasos necesarios para darle al modelo el formato POD, necesario para ser importado en ISGL3D.

En la figura 1.3 se pueden dos imágenes de los modelos 3D pertenecientes a un perro chihuahueño y dos sillones, uno de 2 plazas y otro de 3, vistas desde dos ángulos distintos. Todos descansan sobre el marcador.

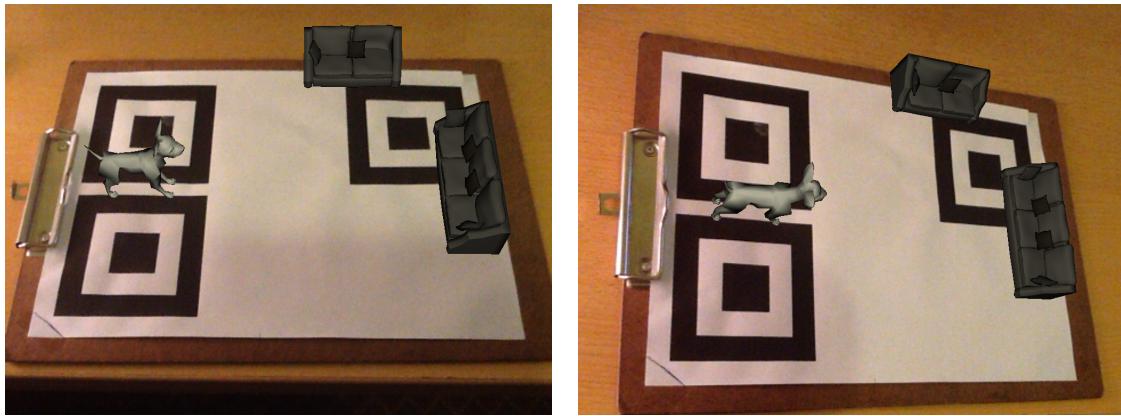


Figura 1.3: 2 capturas de pantalla del caso de uso “modelos”, desde dos ángulos distintos. Se pueden ver los modelos 3D pertenecientes a un perro chihuahueño y a dos sillones, uno de 2 plazas y otro de 3.

#### 1.4.2. Detalles constructivos

Los detalles constructivos de este caso de uso pueden consultarse en el Capítulo ??.

### 1.5. Conclusión