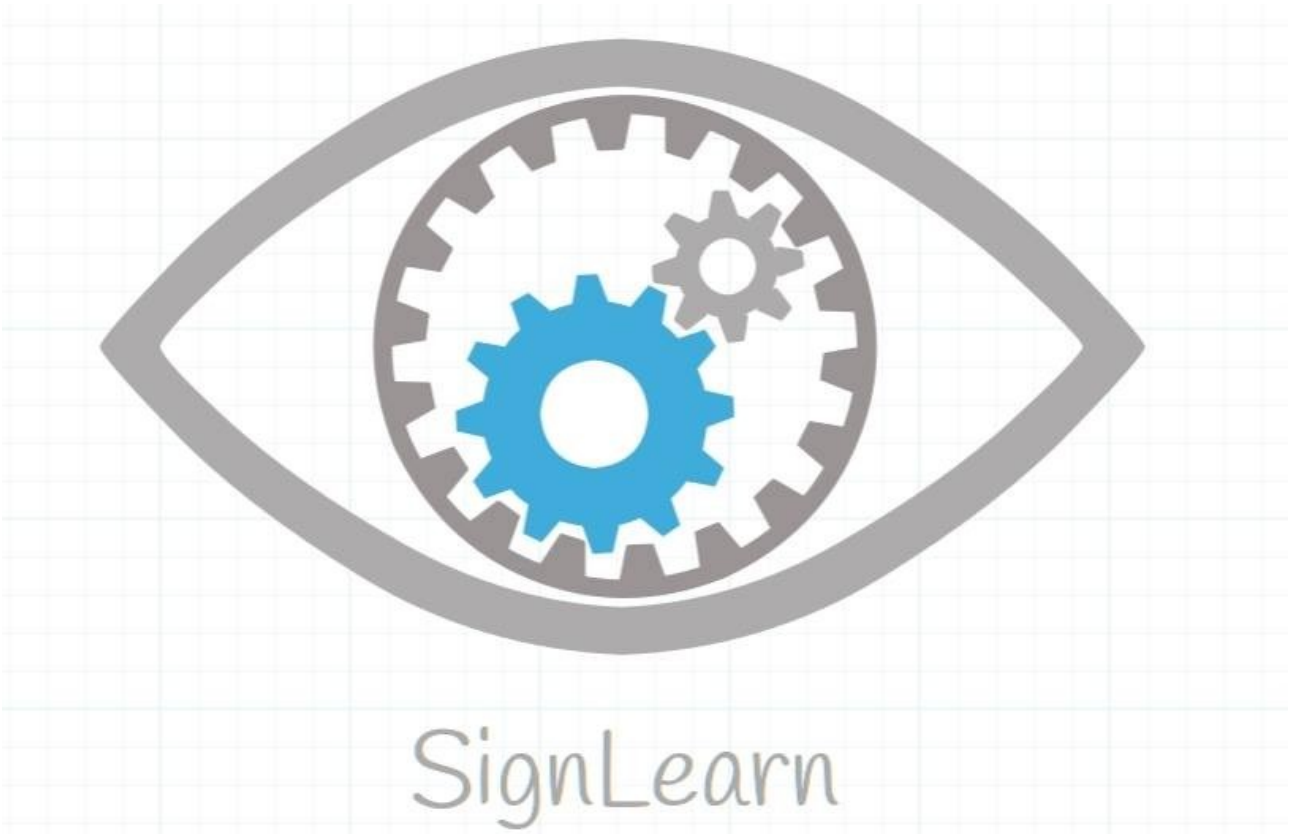


3º Grado En Ingeniería Informática

Computación Ubicua e Inteligencia Ambiental

Tecnologías de la Información

Hugo Bárzano Cruz

Cristina Rosillo Arenas

¿Qué es SignLearn?

SignLearn se presenta como una aplicación didáctica orientada a perfeccionar el aprendizaje de las señales de tráfico, necesarias en especial a los conductores de vehículos y peatones.

Se ha diseñado de forma que no sólo éste dirigida a un tipo de público, sino que podemos enmarcarla en varios ámbitos. Algunos de ellos puede ser por ejemplo:

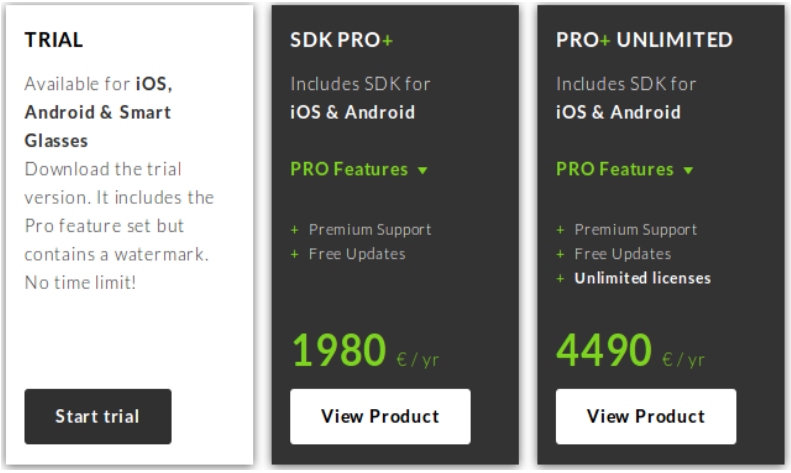
- El infantil, usándola como una herramienta sencilla y divertida para reforzar los conocimientos de “educación vial”.
- También se puede incluir como material extra en centros formativos como autoescuelas, para ayudar al alumnado a aprender los signos usados en la vía pública y así obtener su permiso de conducción.
- Y por supuesto como simple aplicación de entretenimiento, donde podemos poner a prueba nuestro conocimiento de las señales de tránsito.

El dominio de la aplicación no sólo se limita a las señales de tráfico, sino que por ejemplo, podría utilizarse para aprender el significado de las señales de peligro, riesgo, seguridad y salud tanto en una vía pública como en un lugar de trabajo o en el interior de cualquier vivienda o establecimiento comercial.

SDK (Software Development Kit)

Para realizar esta aplicación se ha utilizado la herramienta software Wikitude, se trata de un proveedor de tecnología móvil de realidad aumentada (AR). Fundada en 2008 y que inicialmente se centró en proporcionar experiencias de realidad aumentada basada en la localización.

La SDK de Wikitude es su principal producto, ésta incluye el reconocimiento de imágenes y de seguimiento, modelo 3D, la superposición de vídeo y AR basados en localización.
Wikitude está disponible para los sistemas operativos iOS y Android, y está optimizado para varios dispositivos de gafas inteligentes. El principal problema de éste producto es su precio:



Para realizar este proyecto, se ha utilizado el versión TRIAL cuyo principal inconveniente son las marcas de agua que se superponen a la vista de la cámara.

Implementación

Debemos incluir la SDK dentro de nuestro proyecto con Andoid-Studio, para realizarlo podemos hacerlo siguiendo los pasos descritos en la SETUP GUIDE ANDROID.

El primer paso en la implementación de es registrarnos en Wikitude, obtener la licencia de prueba y crear un Target Collectión, que no es mas que la lista de imágenes objetivo que queremos que sean detectadas por nuestra aplicación.

Figura 1

Como podemos observar en la *Figura 1*, cada imagen objetivo tiene un número de estrellas. Éstas nos indican la facilidad con la que nuestra aplicación reconocerá dichas imágenes. Por tanto cuanta mayor calidad tenga la imagen, será más fácil su captura.

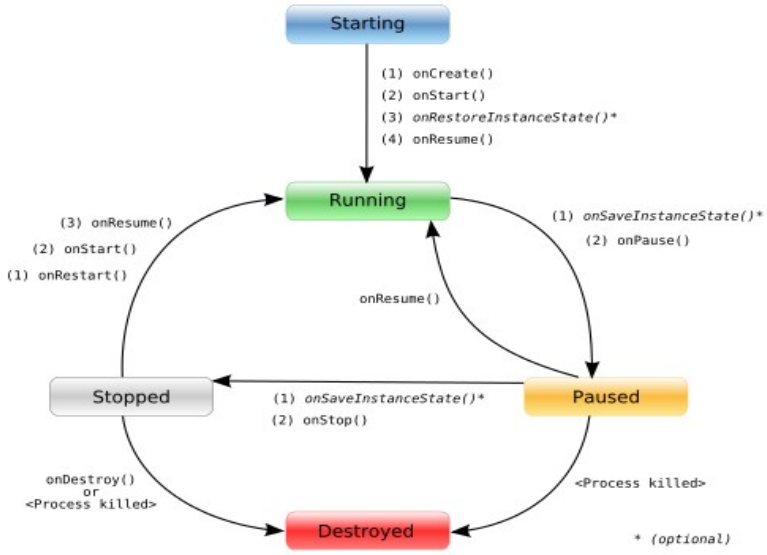
Una vez subidas las imágenes, descargaremos el Target Collection, el cual se representa mediante un fichero ".wtc". Dicho fichero será añadido a nuestra aplicación a través de la siguiente ruta:

/assets/base/assets/targetCollection.wtc

Una vez que tenemos la colección de imágenes, podemos centrarnos en el código de la aplicación. Esta parte de la implementación la hemos divido en dos bloques. El primer bloque, hace referencia a la parte de código java y el segundo bloque a la parte de código javascript.

▫ Primer bloque, se centra en el main activity, o actividad principal, en dicha actividad se implementa el método *onCreate()*, encargado de configurar la interfaz de la clase principal y de establecer la clave de licencia para Wikitude.

Además se implementarán los métodos necesarios para controlar el ciclo de vida de la aplicación. El ciclo de vida de una aplicación, son los distintos estados por los que puede pasar:



– Código asociado a la actividad principal:

```
public class MainActivity extends AppCompatActivity {
    ArchitectView architectView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // la ruta del architectView en nuestro XML
        this.architectView = (ArchitectView) this
                .findViewById(R.id.architectView);
        final StartupConfiguration config = new StartupConfiguration("Z0Xqu1d/kjk...");
        this.architectView.onCreate(config);
    }
   /*
    * Ciclo de vida en nuestra actividad
    */
    @Override
    protected void onResume() {
        super.onResume();
        if ( this.architectView != null ) {
            this.architectView.onResume();
        }
    }
    @Override
    protected void onPause() {
        super.onPause();
        if ( this.architectView != null ) {
            this.architectView.onPause();
        }
```

```
    }
    @Override
    protected void onStop() {
        super.onStop();
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        if ( this.architectView != null ) {
            this.architectView.onDestroy();
        }
    }
    @Override
    public void onLowMemory() {
        super.onLowMemory();
        if ( this.architectView != null ) {
            this.architectView.onLowMemory();
        }
    }
    @Override
    protected void onPostCreate( final Bundle savedInstanceState ) {
        super.onPostCreate( savedInstanceState );
        // IMPORTANTE cargamos el ARchitect worlds (codigo web: HTML CSS javaScript)
        this.architectView.onPostCreate();
        try {
            this.architectView.load("base/index.html");
            this.architectView.onResume();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}
```

Dentro de éste mismo bloque, también resulta interesante comentar el método *onPostCreate()* que es el encargado de cargar todo el código de carácter web. Lo que cabe a destacar de éste método es que carga el fichero *index.html* que contiene las referencias javascript que requiere wikitude denominadas *architect.js* y también el fichero donde nosotros hemos implementado la lógica de la aplicación denominado *ar.js.*

- Código asociado al fichero *index.hmtl:*

```
<!DOCTYPE HTML>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title></title>
    <script src="architect://architect.js"></script>
    <script type="text/javascript" src="js/ade.js"></script>
</head>
<body>
<script type="text/javascript" src="js/ar.js"></script>
</body>
</html>
```

▫ Segundo bloque, hace referencia al fichero *ar.js* que contiene todo el código javascript. En dicho fichero es donde hemos implementado la lógica de la aplicación.

Para alcanzar un mayor grado de comprensión, se van a ir detallando las distintas partes de éste fichero que intervienen en un escenario de uso de la aplicación. El escenario en cuestión es el de la detección y desarrollo de la aplicación para la señal del paso de peatones.

Para comenzar, se define la variable *World* y dentro de ella, toda la lógica de la aplicación. Al comienzo de esta declaración, lo que se realiza es la inicialización del *Tracker* con las imágenes objetivo que contiene nuestro Target collection:

```
var World = {
   loaded: false,
   init: function initFn() {
      this.createOverlays();
   },
   createOverlays: function createOverlaysFn() {
      // inicializamos el Tracker con el Target collection
      this.tracker = new AR.Tracker("assets/tracker.wtc", {
         onLoaded: this.worldLoaded
      });
```

A continuación declararemos los elementos que aparecerán superpuestos a las imágenes detectadas. Estos elementos son de carácter HTML y definen cada una de las opciones que el usuario puede elegir. La imagen objetivo del paso de peatones, tiene tres definiciones posibles, de las cuales dos son incorrectas y una es correcta. Si el usuario elije una definición incorrecta perderá puntos y si elije una definición correcta ganará puntos. Estos elementos html han sido definidos de la siguiente manera:

```
//###################PASO DE PEATONES##############################################
var p_1=false;
var p_2=false;
var p_3=false;
var htmlUno = new AR.HtmlDrawable({
                html:"<html><head><meta name='viewport' content='target-
                 densitydpi=device-dpi, width = 320, user-scalable = 0'>"+
                "</head>"+"<body style='font-size:30px'><div><strong>1 -
                 Advertencias Acústicas Prohibidas.
                 </strong></div></body></html>"
          }, 0.55, {
                viewportWidth: 520,
                viewportHeight: 150,
                backgroundColor: "#3B83BD",
                offsetX: -0.2,
                offsetY: 0,
                horizontalAnchor: AR.CONST.HORIZONTAL_ANCHOR.RIGHT,
                verticalAnchor: AR.CONST.VERTICAL_ANCHOR.TOP,
                updateRate: AR.HtmlDrawable.UPDATE_RATE.LOW,
                allowDocumentLocationChanges: true,
                onClick : function(){
                                if (p_1 ==false) {
                                    this.backgroundColor= "#FE0000";
                                       contador-=1;
```

```
                                        p_1=true;
                                    }
                                }
            });

var htmlDos = new AR.HtmlDrawable({
                html:"<html><head><meta name='viewport'
                content='target-densitydpi=device-dpi, width = 320, user-
                scalable = 0'>"+"</head>"+"<body style='font-
                size:30px'><div><strong>2 - Carril Reservado Para Autobuses.
                </strong></div></body></html>"
            }, 0.55, {
                 viewportWidth: 520,
                 viewportHeight: 150,
                 backgroundColor: "#3B83BD",
                 offsetX: +0.8,
                 offsetY: 0,
                 horizontalAnchor: AR.CONST.HORIZONTAL_ANCHOR.RIGHT,
                 verticalAnchor: AR.CONST.VERTICAL_ANCHOR.TOP,
                 updateRate: AR.HtmlDrawable.UPDATE_RATE.LOW,
                 allowDocumentLocationChanges: true,
                 onClick : function(){
                                if (p_2 == false) {
                                    this.backgroundColor= "#FE0000";
                                    contador-=1;
                                    p_2=true;
                                }
                            }
                        });

 var htmlTres = new AR.HtmlDrawable({
                html:"<html><head><meta name='viewport' content='target-
                densitydpi=device-dpi, width = 320, user-scalable = 0'>"+
                "</head>"+"<body style='font-size:30px'><div><strong>3 -
                Situación de un paso para peatónes</strong></div></body></html>"
                }, 0.55, {
                viewportWidth: 520,
                viewportHeight: 150,
                backgroundColor: "#3B83BD",
                offsetX: +0.35,
                offsetY: -0.25,
                horizontalAnchor: AR.CONST.HORIZONTAL_ANCHOR.RIGHT,
                verticalAnchor: AR.CONST.VERTICAL_ANCHOR.TOP,
                updateRate: AR.HtmlDrawable.UPDATE_RATE.LOW,
                allowDocumentLocationChanges: true,
                onClick : function(){
                                if(p_3 == false){
                                  this.backgroundColor= "#24E711";
                                  contador+=1;
                                  p_3=true;
                                }
                            }
                 });
```

Como podemos observar, en la creación de las variables *AR.HtmlDrawable* se establece el contenido html, las dimensiones y la posición en la que se situarán con respecto a la imagen detectada. Es interesante resaltar el hecho de que inicialmente, los htmls tiene como propiedad el color en segundo plano establecido a azul. Dentro de ella, esta implementado el método que controla el evento que se ejecuta al pinchar sobre ellas (*OnClick)*, dicho método es el encargado de cambiar el color de fondo a rojo o verde en función de si hemos elegido la respuesta correcta o nos hemos equivocado. En dicho método también se incrementa la variable contador, que será utilizada para representar el marcador de puntos. Por ultimo, resaltar el uso de las variables p_1, p_2 y p_3 las cuales se usan para controlar que al pulsar más de una vez sobre la misma definición el marcador no cambie de manera indefinida.

Ahora definiremos el html que representará el marcador con los puntos. Ya que éste html solo tiene carácter informativo, no es necesario que implemente el método *Onclick()*.

```
var htmlPuntuacion = new AR.HtmlDrawable({
                     html:"<html><head><meta name='viewport' content='target-
                     densitydpi=device-dpi, width = 320, user-scalable = 0'>"+
                     "</head>"+"<body style='font-size:300px'><div><strongstyle=
                     'color:#F1C40F;'>"+contador+"</strong></div></body></html>"
                     }, 0.15, {
                     viewportWidth: 150,
                     viewportHeight: 150,
                     offsetX: +0.15,
                     offsetY: 0,
                     horizontalAnchor: AR.CONST.HORIZONTAL_ANCHOR.RIGHT,
                     verticalAnchor: AR.CONST.VERTICAL_ANCHOR.TOP,
 });
```

Una vez creados los HTMLs iniciales y el marcador con la puntuación, empezaremos a definir el Trackable encargado de detectar la señal de paso de peatones y posicionar sobre ella los htmls declarados.

Para alcanzar un mayor grado de comprensión y puesto que este objeto es bastante extenso, vamos a dividirlo en varias partes para poder ir comentado que acciones se llevan a cabo en cada una de las partes y que métodos son implementados para controlar el comportamiento de la aplicación ante ciertos eventos.

Primero se establecerán los htmls que se van a superponer cuando la aplicación detecte la señal de paso de peatones. Dichos htmls son los establecidos en el array cam:

```
//################### TRACKABLE - PASO DE PEATONES ##############################

var paso_peatones = new AR.Trackable2DObject(this.tracker, "paso_peatones", {
   drawables: {
      cam: [htmlUno,htmlDos,htmlTres,htmlPuntuacion]
   },
```

A continuación implementamos el método *OnClick()* que se ejecutará cada vez que se lance el evento *OnClick()* de los objetos htmls asociados al Trackable en el array cam, es decir, cada vez que el usuario pulse sobre una definición. Se realizará de forma simultánea una actualización del marcador con la nueva puntuación.

```
onClick : function(){
            this.drawables.removeCamDrawable(3);
            var htmlPuntuacion = new AR.HtmlDrawable({
                                    html:"<html><head><meta name='viewport'
                                    content='target-densitydpi=device-dpi, width =
                                    320, user-scalable = 0'>"+ "</head>"+"<body
                                    style='font- size:300px'><div><strong
                                    style='color: #F1C40F;'>"+contador+"</strong>
                                    </div></body></html>"
                                    }, 0.15, {
                                    viewportWidth: 150,
                                    viewportHeight: 150,
                                    offsetX: +0.15,
                                    offsetY: 0,
                                horizontalAnchor:AR.CONST.HORIZONTAL_ANCHOR.RIGHT,
                                verticalAnchor: AR.CONST.VERTICAL_ANCHOR.TOP,
            });
             this.drawables.addCamDrawable(htmlPuntuacion,3);
           },
```

La manera de conseguir la actualización de puntuación es eliminando del array cam el html que representa el marcador, crear un nuevo objeto con los datos actualizados y volver a añadirlo al array cam del trackable.

El siguiente método a implementar es el *onEnterFieldOfVisión*, este método controla cuando una imagen objetivo entra en el campo de visión, es decir, es detectada. Con éste evento, conseguimos que el marcador permanezca con la puntuación correcta y actualizada entre los distintos Trackables. En otras palabras se encarga de mantener el marcador cuando pasamos de escanear el paso de peatones a escanear por ejemplo la señal de stop. La implementación es similar a la del evento *OnClick()*, se elimina el marcador del array cam, se crea un nuevo objeto con el valor actual de la variable contador y se superpone sobre la imagen objetivo.

```
onEnterFieldOfVision : function(){...},
```

Y para finalizar definiremos el método *OnExitFieldOfVision()* que se encarga de controlar el evento en el que la imagen objetivo sale del campo de visión, es decir, deja de detectar por ejemplo el paso de peatones. Éste método resulta interesante para mantener la fluidez de la aplicación ya que podemos aprovechar el evento para recargar los htmls de las definiciones que se superponen en el Tracker.

Esto es necesario debido a que cuando hacemos click en una de las definiciones y se cambia su color para indicar si nos hemos equivocado o no, la propiedad de ese html ya queda modificada y en futuras detecciones de la misma señal, aparecería desvelada la opción elegida por primera vez. Para

evitar eso, se procede de la misma manera, eliminado los htmls del array cam, creándolos de nuevo con color de fondo azul y volviéndolos a añadir al array cam para que se superpongan a la imagen objetivo en futuras detecciones. (La declaración de los htmls es similar a los iniciales por lo que se obviará en este manual)

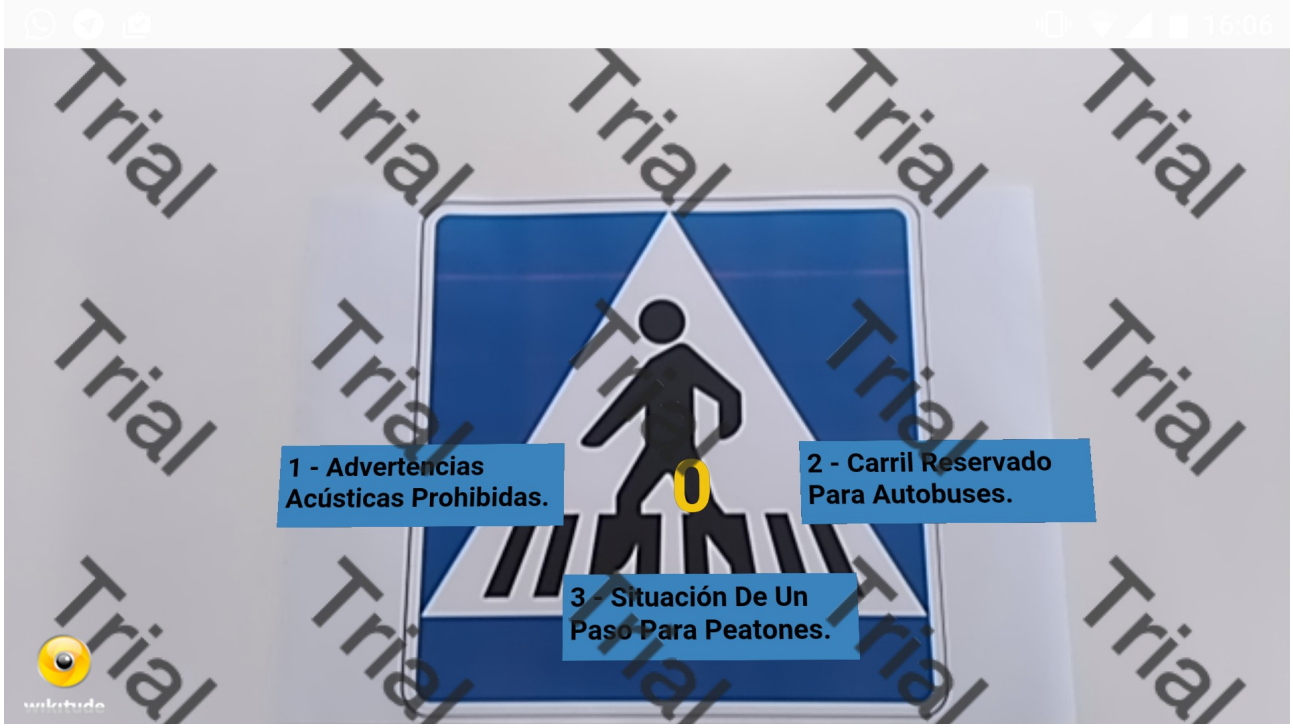
```
 onExitFieldOfVision : function(){
            this.drawables.removeCamDrawable(0);
            this.drawables.removeCamDrawable(1);
            this.drawables.removeCamDrawable(2);
            p_1=false;
            p_2=false;
            p_3=false
            var htmlUno = new AR.HtmlDrawable({ … });
            var htmlDos = new AR.HtmlDrawable({ … });
            var htmlTres= new AR.HtmlDrawable({ … });
                                                    });
            this.drawables.addCamDrawable(htmlUno,0);
            this.drawables.addCamDrawable(htmlDos,1);
            this.drawables.addCamDrawable(htmlTres,2);
            }
});

};
```

Para que la aplicación pueda detectar también la imagen de stop, se procede de igual forma, declarando un nuevo trackable, con la imagen de la señal de stop asociada. Lo único que cambia son las definiciones dentro de los htmls y las variables booleanas para controlar que no se seleccione una definición mas de una vez en la misma detección. Por ultimo, dentro del fichero *ar.js* lo que hay que hacer es inicializar la variable *WORLD* que indicamos al principio y que contiene todo lo especificado hasta ahora. Esto se realiza mediante:

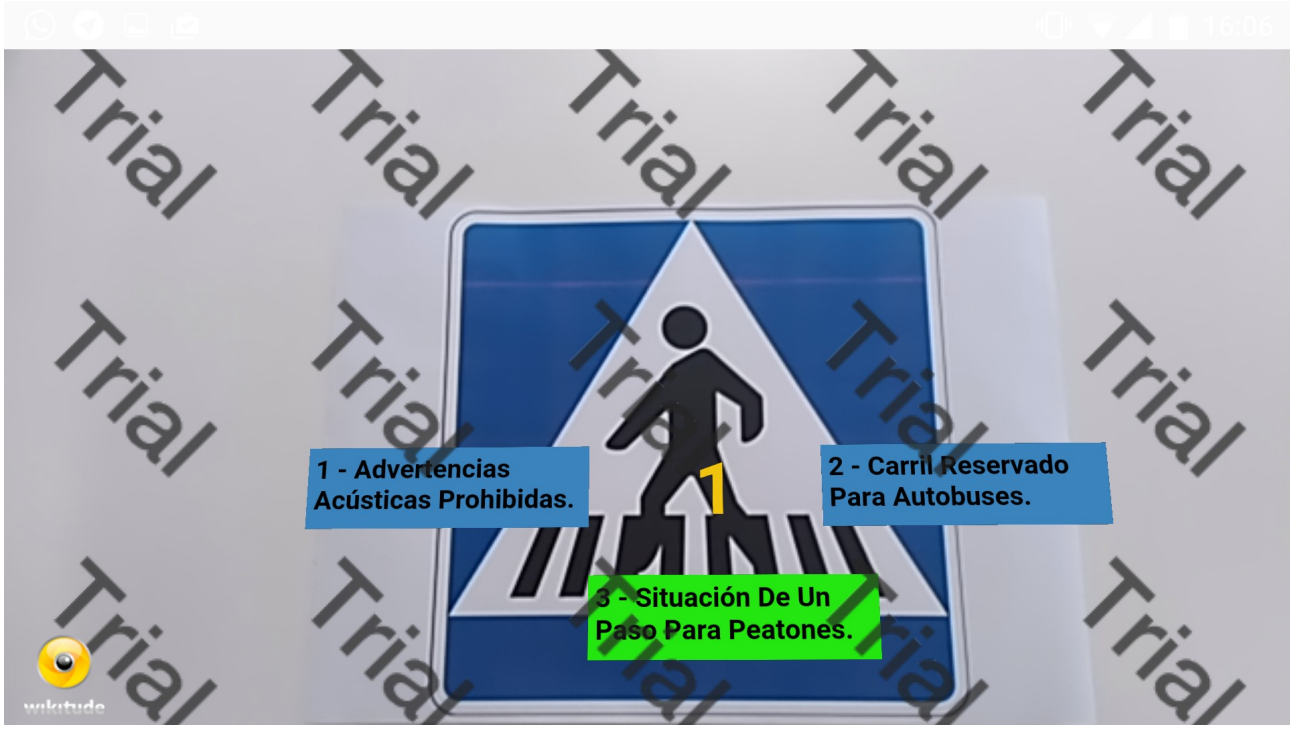
```
// principal
World.init();
```

Ejemplo de Uso

Para comprender el funcionamiento de la aplicación, se va a mostrar un ejemplo de uso. Inicialmente, se parte de una imagen, como por ejemplo la de paso de peatones y podemos observar como el marcador esta a 0:

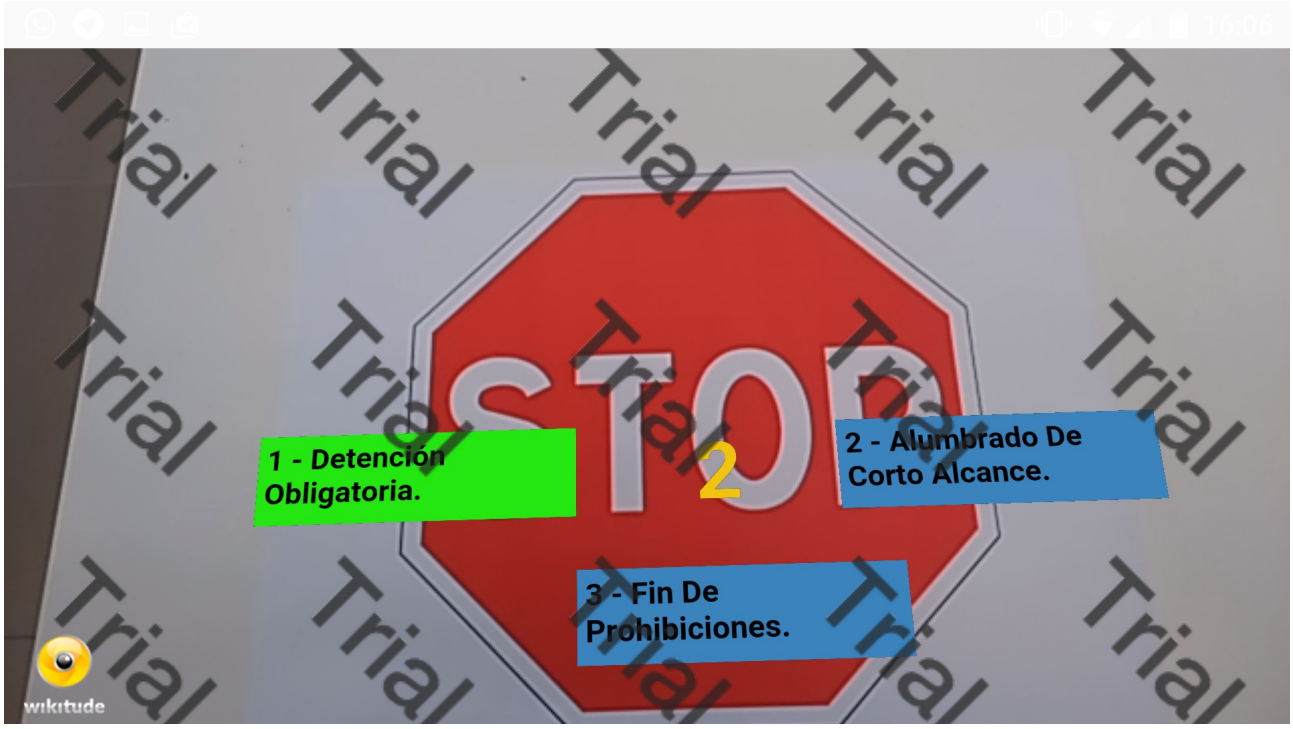


En caso de que el usuario elija la opción correcta, obtiene un punto:



Si ahora detecta otra imagen, como por ejemplo la señal de stop, podemos observar como las definiciones cambian, pero el marcador se mantiene:

Si el usuario acierta, sigue acumulando puntos:



Pero si comienza a equivocarse, comienza a perderlos:

En el caso de que el usuario cometa muchos errores, puede incurrir en una situación en la que deba puntos, de la cual solo puede salir si comienza a acertar señales:

Estudio de Mercado

Como parte del estudio de mercado con el objetivo de mejorar la distribución de la aplicación, se han llevado a cabo la siguiente encuesta sobre una muestra de 50 sujetos:

-Edad Medi de los Sujetos: 18 a y 56 años.
-Sexo: 25 mujeres y 25 hombres
-SO móvil: Android 45 y 5 IOS

La encuesta esta formada por las siguientes cuestiones:

1. ¿Tienes algunas nociones de seguridad vial?

El 95% de los encuestados contestaron que “SI” frente a un 5% que contestaron negativamente.

2. ¿Tienes el carnet de conducir?

El 75% de los encuestados contestaron que “SI” , un 15% contestaron que no y un 10% contestaron que estaban en proceso.

3. ¿Con que edad te sacaste el carnet de conducir?

El 80% contesto que entre los 18-28 años y el 20% restante, de los 28 en adelante.

4. ¿Cuantos puntos tienes?

El 90% de los participantes tienen entre 8 y 12 puntos y el 10% de 12 a 15 puntos.

5. ¿Utilizarías esta aplicación como herramienta de aprendizaje?

El 50% de los encuestados contestaron afirmativamente frente al 50% que contesto negativamente.

6. ¿Estarías dispuesto a pagar por esta aplicación?

El 20% de los encuestados estaría dispuesto a pagar por la aplicación, frente al 80% que no.

7. ¿Utilizarías esta aplicación fuera del contexto de la educación vial?

De el 50% de los encuestados que contestaron afirmativamente que utilizarían esta aplicación como herramienta de aprendizaje, solo el 60% la utilizaría fuera del contexto de la educación vial.

8. ¿Te resultaría interesante que una autoescuela te facilitara esta herramienta como parte del aprendizaje?

El 95% de los encuestados, contesto afirmativamente ya que supone una innovación dentro de la metodología de aprendizaje. El 5% restante indicó que preferiría el método tradicional.

Bibliografía

Wikitude ARchitect v2.0 API Documentation

Setup Guide Android

Desarrollando aplicaciones de Realidad aumentada con Wikitude (parte 1)

Desarrollando aplicaciones de Realidad Aumentada con Wikitude (parte 2)

Fuentes

El código fuente de la aplicación esta liberado en la plataforma para el control de versiones Github y puede ser consultado desde **aquí.**