INTRODUCCIÓN

La realidad aumentada

La realidad aumentada, en adelante RA, es el término que se usa para definir una visión a través de un dispositivo tecnológico, directa o indirecta, de un entorno físico del mundo real, cuyos elementos se combinan con elementos virtuales para la creación de una realidad mixta en tiempo real.

Según Ronald Azuma, desarrollador y lider de proyecto en New Media, *Intel Corporation*, y uno de los pioneros en el campo de la realidad aumentada, dice que la RA consta de las siguientes características:

- Combina elementos reales y virtuales.
- Es interactiva en tiempo real.
- Está registrada en 3D.

A su vez, consta de unos requisitos mínimos para poder ser emulada, los cuales son:

- Una pantalla, donde mostrar la combinación de los elementos reales captados por algún dispositivo y los elementos virtuales generados por un software.
- Un conjunto de dispositivos que capturen los elementos del entorno y nuestra situación como son una cámara, acelerómetro, acelerometro... de tal forma que permita al software tener referencias de como y donde debe mostrar sus elementos virtuales.
- Un hardware relativamente potente, para poder realizar los cálculos necesarios para mostrar el entorno que captura la cámara y ser capaz de hacer frente al software que genera los elementos virtuales y combinarlos en la pantalla con los reales.
- Un software capaz de reconocer el entorno y calcular donde y como debe representar los elementos virtuales combinados con los reales para conseguir una visión de RA.

Con el continuo avance de los dispositivos móviles y la potencia, a un coste aceptable para la mayoría, de la que constan ahora mismo estos, permiten que cualquier usuario pueda hacer uso de la RA, ya que sus smartphones cumplen todos los requisitos para ello.

Alcance

Actualmente, el hecho de que la era de los smartphones ha puesto practicamente en la mano de la mayoría de las personas uno de estos dispositivos hace que la

RA esté ahora mismo en su punto mas alto y que su crecimiento sea cada vez mayor.

Ahora mismo, tiene diferentes aplicaciones en diversos campos como son:

Información interactiva

En este caso se utiliza para dar a conocer de forma interactiva, información acerca de un elemento cercano al usuario de tal forma que se puede mostrar un tipo de información u otra en función de la interacción entre el usuario y dicho elemento.

Ahora mismo este area está muy presente en museos como forma dinámica de conseguir una inmersión del usuario con lo que está viendo y hacer de su visita una experiencia más atractiva e incluso hasta más productiva.

Dentro de este campo también se hace uso de la RA en herramientas utilizadas para el intercambio de información en proyectos profesionales o con fines comerciales como el de mostrar catálogos productos de una forma interactiva.

Entretenimiento

En la actualidad, aunque no está todavía muy popularizado su uso, existen una gran cantidad de videojuegos que hacen uso de está tecnología que, junto con diferentes mecánicas que se adaptan a ella, generan una novedosa experiencia para entretener el usuario.

Ciencia y desarrollo

Aunque todavía no está normalizado el uso de la RA en este campo, si están naciendo númerosos proyectos con el fin de desarrollar está tecnología para utilizarse en areas como la medicina y la construcción entre otras.

Otros

Además de los ya mencionados, existen todavía muchísimas areas en las que tienen cabida en sus tecnologías la RA además de muchos otros usos que o bien están por descubir o no se consta todavía de la tecnología suficiente como para integrarlo.

Algo que todavía está en una fase temprana de desarrollo pero que tiene un futuro prometedor es el uso de la RA para generación de terreno de forma que se pueda reconocer el mismo y en función de su forma generar un medio en relación un entorno que todavía no estaba registrado. Esta tecnología dota de un sinfín de posibilidades añadidas a las ya existentes para el uso de la RA en muchos otros campos.

ESTADO DEL ARTE

Impacto

... Puestos de trabajo relacionados con la RA, inversión destinada a su desarrollo, estudios y empresas que trabajan con RA, crecimiento del nivel de investigación en el sector.

Historia y evolución

... Breve descripción de como ha evolucionado la RA a lo largo de los años.

Avances

... Lo que se está investigando, mejorando o desarrollando sobre RA.

Algunos ejemplos

... Ejemplos de aplicaciones y usos de la RA que esten ahora mismo vigentes o que se esten desarrollando.

TECNOLOGÍA

Cómo funciona

... Resumen y de las diapositivas de Guillermo del drive drive_pdf)

Herramientas para el desarrollo

... Vuforia y otras alternativas para la programación de RA.

Vuforia

Vuforia es el framework que hemos utilizado para hacer toda la parte de RA. Es gratuito y tiene una gran comunidad, así como buenos ejemplos y tutoriales. Facilita mucho el trabajo, y se puede utilizar con diferentes SDK (Android, iOs, Unity 3D, ahora gafas de realidad virtual...).

Cómo genera realidad aumentada Básicamente, Vuforia superpone a la imagen tomada por la cámara de, en este caso, nuestro smartphone, cualquier modelo en tres dimensiones que queramos sobre la posición de un *detector* que le hayamos indicado. De esta manera, tenemos un "fondo" con la imagen tomada por la cámara, con modelos en tres dimensiones "por encima". Además, nos mantiene siempre los objetos de tres dimensiones en el mismo punto del espacio, por lo que si movemos nuestra cámara, cambiará la perspectiva desde donde vemos el objeto, pudiendo girar alrededor de éste. El comportamiento puede ser diferente, dependiendo de como lo hayamos configurado (podemos hacer que el objeto persista aun que perdamos de vista el detector).

Plataformas de desarrollo Vuforia proporciona paquetes para trabajar directamente con el SDK de Android o el de iOS, así como para Unity3D. Utilizando Unity3D podemos exportarlo después a una aplicación de Android o iOS también, aun que no quedaría de una manera tan "pulida" como desarrollándola directamente con el SDK del sistema operativo deseado. Nosotros hemos decidido utilizar el paquete para Unity3D porque los tres teníamos unos conocimientos básicos en desarrollo con Unity, además de que nos permite exportar después el proyecto al sistema operativo que quisiéramos.

Unity como herramienta y C# como lenguaje Unity funciona con algo a lo que han llamado *escenas*, que serían como diferentes situaciones del juego. En toda escena hay una jerarquía de objetos que la componen, y de cada objeto pueden *colgar* otros objetos, además de que se pueden añadir (por medio de código programable) otros objetos a esa jerarquía de manera dinámica. Todos los objetos de Unity tienen una serie de componentes, el más básico sería el de su situación en las tres dimensiones (o dos), su escala y su rotación con respecto a los tres planos. Éstos componentes permiten configurar los objetos de manera sencilla, encapsulando funcionalidades. Esta forma de "componer" los objetos no es casual: es la más utilizada en programación de videojuegos.

Además, Unity cuenta con una extensísima comunidad de desarrolladores, así como tutoriales, guías, dudas resueltas... solo con los tutoriales que proporciona la propia gente de Unity podemos hacer un juego casi de cada uno de los tipos más comunes de juegos.

Unity nos proporciona por defecto el calculo de colisiones entre objetos, gravedad, eventos de teclado o ratón... en pocos minutos podemos hacer cosas sencillas pero que con otras herramientas, o programándolo directamente a mano con un lenguaje de programación cualquiera como podría ser Java o C++, nos llevarían bastante más tiempo.

Los *scripts* los podemos escribir en C#, Boo o un lenguaje "parecido" a JavaScript. Nosotros hemos decidido utilizar C#, ya que era la opción que más nos convencía por varias razones:

- Hemos leído que es más eficiente. [http://answers.unity3d.com/questions/7567/is-there-a-performance-difference-between-unitys-j.html]
- Los tres teníamos conocimientos previos de Java, y C# es muy similar a Java en cuanto a sintaxis.
- Es el más usado por la comunidad. [http://forum.unity3d.com/threads/boo-c-and-javascript-in-unity-experiences-and-opinions.18507/]

Todos los *Scripts* utilizados en Unity heredan de la clase **MonoBehaviour**, la cual permite a estos *scripts* integrarse con la ejecución interna de Unity. Toda clase que herede de MonoBehaviour tiene los métodos Start(), Awake(), Update(), FixedUpdate(), y OnGUI(). Éstos se ejecutan en diferentes momentos del juego.

- **Awake()**: el primer método al que se llama, antes incluso de que el objeto asociado esté habilitado en la escena. Se utiliza para inicializaciones o referencias entre *scripts*.
- **Start()**: se ejecuta después de *Awake()*, justo antes del primer *Update()* y después de que se active el objeto.
- **Update()**: se ejecuta en cada *frame*. Ésto hace que dependa del procesador y del equipo donde se ejecuta. Se usa para actualizaciones comunes como mover objetos no físicos, recoger entrada del usuario...
- **FixedUpdate()**: el intervalo entre una ejecución y otra es consistente y siempre el mismo. Se utiliza para actualizaciones como ajustar objetos físicos.
- **OnGUI()**: se utiliza para gestionar y renderizar eventos de la *Interfaz Gráfica de Usuario* (*Graphic User Interface*, **GUI**). Sólo es llamada si el objeto está habilitado.

Unity + Vuforia Vuforia nos proporciona un paquete de extensión de Unity 3D el cual debemos importar para trabajar. Éste paquete contiene diferentes *prefabs* que nos harán la tarea muy sencilla.

Lo que debe tener toda aplicación de RA hecha con Vuforia y Unity 3D es una ARCamera (cámara de RA). A ésta hay que indicarle el *product key* que nos da Vuforia desde su portal para desarrolladores, además de ésto, se le indicará el paquete de *targets* propios (lo explicaremos más adelante en profundidad). Es la unidad mínima de desarrollo de RA.

Una vez hecho esto, tendremos diferentes opciones para lanzar los objetos de RA, que deben colgar en la jerarquía de Unity de cualquiera de los siguientes *prefabs*:

- **Frame Markers**: son marcadores muy sencillos que son proporcionados por la gente de Vuforia en su paquete. Se pueden utilizar para calibrar la cámara, pero no tienen una gran calidad a la hora de ser detectados. Son lo más sencillo para comenzar una aplicación de prueba.

- **Image Targets**: imágenes propias del desarrollador. Funcionan como los Frame Markers, pero éstas deben ser importadas desde un paquete generado por el portal de desarrolladores de Vuforia, el cual nos indicará la calidad de esa imagen para ser detectada.

[caputra calidad image targets]

- **Multi-Targets**: son varios *ImageTargets* que representan las diferentes caras de un prisma en tres dimensiones.
- **Cylinder Targets**: *ImageTarget* que envuelve un cilindro, para representar, por ejemplo, una botella u otro objeto similar.
- **Text Recognition**: nos permite detectar textos, ya sean del diccionario proporcionado por Vuforia de palabras en inglés (más de 100.000 palabras diferentes) o de uno creado por nosotros mismos.
- **Object Recognition**: sirve para configurar un objeto en tres dimensiones que no sea ninguno de los anteriores.
- **Smart Terrain**: el más sorprendente para nosotros. Permite reconstruir el entorno del usuario de la aplicación en tres dimensiones. Nosotros no lo hemos utilizado pero es fascinante. (https://www.youtube.com/watch?v=JvE_7filGsY) [imagen smart terrain]

Con cualquiera de estos objetos, la funcionalidad por defecto (que podemos modificar creando nuestras propias clases que hereden de las que nos da Vuforia) es que al detectarse (ya sea un ImageTarget, un Text Recognition, etcétera) se comenzarán a mostrar todos los objetos que cuelguen de él en la jerarquía de Unity.

MEMORIA

Introducción

El trabajo que se ha realizado para este proyecto es un videojuego. Este, está ambientado en el espacio y hace uso de la realidad aumentada como entorno.

La idea principal del proyecto es la de implementar juegos con diferentes mecánicas y adaptarlos a su uso con RA y la tecnología móvil. Con este fin, creemos que se puede hacer de cualquier espacio, un lugar mucho más interesante haciendo uso de esta tecnología, ya que no solo imaginas la realidad del lugar si no que puedes interactuar con él.

Con el fin de hacer uso de la tecnología actual que se nos brinda para el desarrollo de la RA, hemos implementado tres juegos que creíamos viables, y que nos permiten poder experimentar con ella. Estos tres juegos son el Arkanoid, un Space invaders adaptado al uso de la RA y WaterPipes. Los tres, están unidos

haciendo uso de un hilo argumental que no solo da coherencia a los juegos, sino que obliga al usuario a moverse por la facultad para poder completar los niveles.

Antecedentes

El proyecto que hemos realizado para este TFG, es un proyecto nuevo y que ha sido diseñado e implementado desde el principio por nosotros. En años anteriores se realizaron trabajos de fin de grado dedicados a la Realidad Aumentada en museos, como el realizado el año pasado (2014/2015) para el Museo de América. Nuestro proyecto guarda muchas similitudes con el citado anteriormente, como el uso de Realidad Aumentada en museos para mejorar la experiencia del visitante. Por tanto, este trabajo puede que sea nuestro antecedente, aunque el concepto de proyecto sea distinto, ya que ellos utilizaban la AR como medio de información. Y nosotros añadimos los videojuegos en RA como medio de entretenimiento en la visita.

Objetivos y motivación

Desde la primera reunión con nuestro Director del proyecto, nuestro objetivo fue el desarrollo de una aplicación que mejorara la experiencia del usuario en un museo, en este caso, el museo de la Facultad de Informatica, "García-Santesmases". Pero no solo esto, si no que nuestro objetivo era hacerlo a través de los videojuegos y utilizando la realidad aumentada.

Esto lo logramos mediante una "búsqueda del tesoro", guiando al visitante a que recorra el museo en busca de misiones que tendrá que ir completando (minijuegos) para poder pasar a la siguiente. Así, al finalizar la visita, el jugador habrá recorrido el museo de una forma amena y divertida. La realidad aumentada, es una tecnología que aunque ya lleva muchos años, no se conocía a nivel usuario, pero el incremento tan alto del uso de los Smartphones en los últimos años en la sociedad, ha permitido crear más aplicaciones que todo el mundo pueda utilizar, ya que hoy en día casi cualquier persona de una edad comprendida entre los 16 y 55 años tiene un dispositivo móvil que le permite ejecutar una aplicación de RA en cualquier lugar y momento.

Nuestra motivación principal en la realizacion de este proyecto, fue profundizar en el desarrollo de videojuegos con una herramienta tan innovadora como lo es la Realidad Aumentada. Por lo que enseguida comenzamos a investigar sobre aplicaciones creadas anteriormente y descubrimos que los museos se estan haciendo cada vez mas eco de los beneficios de aplicaciones como esta para atraer al publico. Esto nos motivo mas, ya que nos ponia delante una opcion real de desarrollo.

Plan de trabajo

Diseño del Proyecto

Inicialmente nos reuniremos para definir nuestros objetivos para este proyecto y así posteriormente diseñar un primer boceto de lo que será nuestra aplicación. Como la idea fundamental del proyecto está bastante clara, realizar una aplicación con varios mini juegos por el museo "García-Santesmases" de la facultad, con RA; decidimos investigar qué tipo de juegos podemos implementar, para ello realizaremos un estudio de los juegos ya existentes en museos con y sin RA, e iremos al museo de la facultad. Durante la visita al museo, y ya con una idea de lo que podemos o no desarrollar, obtendremos una lista de los posibles juegos que podemos realizar en distintas partes del museo.

Una vez elegidos los juegos, haremos un pequeño estudio del alcance del proyecto y decidiremos cuales son los juegos más viables, teniendo en cuenta el tiempo de desarrollo y pero sobretodo la experiencia del usuario. Aunque la aplicación se basa en varios minijuegos repartidos por diferentes zonas del museo, no hay que olvidar que el objetivo principal es mejorar la experiencia del usuario y para conseguirlo es fundamental crear un hilo argumentativo que facilite al usuario encontrar los juegos y lo más importante añada emoción y diversión a la visita. Por lo que, cuando ya tengamos los juegos y sepamos la distribución por el museo, debemos crear un hilo argumentativo que una de una manera amena todos los juegos entre sí. Cuando tengamos una versión estable, realizaremos pruebas con usuarios de distintos perfiles, para poder obtener un estudio de viabilidad de la aplicación más fiable y detallado.

ARKANOID

Historia El Arkanoid es un juego de los 80, donde el jugador controla una plataforma que impide que una bola se salga de la superficie que limita el juego. El objectivo principal de la bola es el de destruir todos los ladrillos o bloques de la pantalla sin salirse de esta. La misión del jugador es la de, no solo impedir que la bola se salga de la escena, si no la de situar la plataforma que maneja de tal forma que consiga que la bola rebote y destruya todos los bloques.

[imagen de la portada del arkanoid]

A lo largo del juego puede haber un gran numero de variaciones que complican el juego o que facilitan las cosas. La plataforma del jugador puede modificar su tamaño, u obtener mejoras como disparos para romper los bloques. La bola, puede verse modificada mediante variaciones de tamaño o en la fisica del juego como romper varios bloques sin rebotar o incluso multiplicarse. Y los bloques pueden variar en su posición e incluso desplazarse con el fin de finalizar el juego cuando llegan abajo.

[imagen de la escena de arkanoid]

Nuestra versión Es nuestra verión las cosas son sencillas. El usuario proviene de defender la facultad (*nave espacial tfg-III*) de unos invasores, y tras finalizar, recibirá una misión, que es la de despejar el campo de batalla para poder despegar la nave. Para ello debe abrir paso reciclando escombros, haciendo uso del panel de reciclaje que se situa en la tercera planta, y que se accede a él, apuntando al código QR de *Super Mario* situado junto a una de las consolas.

[imagen del qr de super mario]

Una vez se reconozca el ImageTarget (*el QR de Super Mario*), aparecerá un viejo televisor, y en su pantalla se observa el juego y dos contadores. El objetivo es sencillo, hay que despejar los escombros. Hay un minuto para realizar dicha tarea o hasta que la bola atraviese la deathzone. Cuantos mas residuos recicle, mayor será la puntuación que arrastrá al siguiente juego.

Una vez en situación, mientras enfoca al target para no perder de vista la escena, el jugador deberá arrastrar su dedo por la pantalla de su smartphone para poder desplazar la plataforma y controlar donde rebota la bola. Mientras hace eso, la bola irá rebotando y el tiempo decrementando. A medida que destruya objetivos, la puntuación crecerá y el tiempo se irá agotando.

[captura del juego enfocado en el museo de la facultad]

El juego está ambientado en un entorno espacial, haciendo referencia al hilo del juego pero solo de forma superficial. Lo que conforma el tema del espacio es lo que aparece en la pantalla del televisor que contiene la escena. Pero, en realidad, la escena tiene una connotación *retro*, donde un antiguo televisor al lado de una de las consolas que en un pasado ejecutaba el juego, quiere recrear la forma primitiva de jugar al Arkanoid; *en un televisor de tubo* y con un juego sencillo que era para lo que daba de sí la tecnología y los medios de la época.

Con esta recreación, lo que se intenta es, no tanto simular un juego entretenido para cumplir con el objetivo de la historia, si no el recrear una escena pasada basada en este juego. De esta forma, se hace uso de la RA en este caso, no como medio de entretenimiento a través las mecánicas que proporciona, si no generando una escena para mejorar la experiencia del visitante del lugar, el museo de la facultad.

Implementación

Diseño La escena se conforman por los siguientes elementos:

- ARCamera e ImageTarget como elementos principales en una escena de RA. La propia librería de *Vuforia* nos provee de ellos, solo hay que definir el Image target al que reacciona la escena y añadirlos; del resto ya se encarga su lógica interna.

- Un modelo 3D de un televisor que hace de contendor de la escena y que no tiene ninguna mecánica asociada.

- La bola, en la que se han implementado dos componentes; uno físico para hacer que esta rebote, y un script cuya finalidad es la de destruir los bloques, o mejor dicho, la de lanzar la animación de destrucción del bloque y finzalizar el juego al acabar con todos los bloques, o, al salirse de la zona de juego. Para evitar probloemas y por lógica del juego, aunque la escena sea 3D los compoenentes sobre este juego son en 2D.

- **Ball Material**: Es un material de unity que, junto al componente *Rigid-Body2D*, elimina la gravedad en el objeto, configura a este oara poder rebotar, consigue que la fuerza aplicada sobre la bola sea infinita, de forma que no pare jamás de desplazarse; y, define las coordenadas en las que puede desplazarse el objeto (x e y) y en las que puede rotar (ninguna).

  [imagen del inspector con mostrando el componente ballMaterial]

- **BoxCollider2D**: Para controlar todo el tema de colisiones. En este caso, el collider que envuelve a la bola es un cuadrado rotado 45 grados para evitar problemas en los rebotes.

  [imagen del boxcollider de la bola del arkanoid]

- Los bloques, que se reparten por la escena esperando a que la bola colisione con ellos y recibir así, la orden de que se lance su animación de destrucción y se aumente el marcador.

- **Audio source**: el componente que da sonido a la acción de destrucción.

- **Animacion**: la animación de destrucción que se lanza al colisionar con la bola. Esta se genera por interpolación, ya que es el sistema del que nos dota Unity, y lo que hace es reducir la escala del objeto a cero.

[imagen de los 4 bloques de Arkanoid]

- La plataforma: Esta contiene un componente que reacciona a los eventos de la pantalla táctil y que le permite deplazarse, además también tiene un *RigidBody2D* cuyas constraints impiden que pueda verse desplazado en el eje vertical o rotado.

- Canvas: Contiene la información del estado del juego. Este objeto y sus hijos, a diferencia del resto, en escena, se muestra en función a la pantalla del dispositivo en el que se ejecuta el juego y no en función de lo que enfoca la cámara. Sus hijos son, el marcador, el contador de tiempo y la pantalla de precarga que muestra la información de la ejecución del juego al usuario.

Desarrollo Para el Arkanoid, se han tenido que implementar las siguientes lógicas de juego:

1. **ScreenDragListener.cs**: Se encarga de capturar las pulsaciones en pantalla. En su método *Update()* se comprueba con cada llamada si la pantalla ha sido pulsada; si es así, se guarda el punto inicial donde se detectó la pulsación y se guarda, de forma que al volverse a llamar al método, esta vez, no solo se comprueba si ha habido contacto con la pantalla, sino que, además, se comprueba si ha habido variación en la posición del punto.

El script, tiene una interfaz interna. En el método *Start()*, se comprueba todos los componentes que implementan dicha interfaz en la escena y se almacenan en un aray. Cada vez que se detecta y cálcula un movimiento de *drag* sobre la pantalla del dispositivo, se recorren los componentes del array. Estos reciben, a través del método, las variaciones en los ejes de la pulsación; y así pueden realizar las acciones correspondientes.

“‘c# public class ScreenDragListener : MonoBehaviour { ... private Component[] onDragListeners; ... void Start () { onDragListeners = (Component[])GetComponentsInChildren(typeof(OnDragListener)); }

```
void Update () {
  ...
  if (horiz != 0 || vert != 0) {
                foreach(OnDragListener onDragListener in onDragListeners){
                    onDragListener.onDrag (vert, horiz);
                }
            }
            touchOrigin = myTouch.position;
        }
    } else {
        foreach(OnDragListener onDragListener in onDragListeners){
            onDragListener.onRelease();
        }
    }
}

...
public interface OnDragListener {
      void onDrag (float vertical, float horizontal);
      void onRelease();
}
```

} “‘

2. **BallIA.cs**: No tiene mucha explicación, aplica una fuerza en dos coordenadas que se le asignan como atributos en el momento en el que está activa y se toque por primera vez la pantalla.

3. **DefaultTrackableEventHandle.cs**: Es un script que viene por defecto asociado al *ImageTarget* de *Vuforia*. En este caso, hemos hecho una leve modificación sobre este sript, donde hemos añadido un array de componentes que, cuando se detecta el QR en escena (*OnTrackingFound()*) se recorre y estos se van activando.

4. **DieOnCollide.cs**: con dos parámetros que definen las etiquetas de los *gameObjects* que se comportarán como enemigo (el gameObject que delimita la deathzone) y los objetivos (los bloques que se destruyen al colisionar con ellos). Implementa el método *OnCollisionEnter2D()*, que es aquel al que llama el collider 2D cuando entra en collisión el *gameobject* con otro objecto con collider. Cuando se detecta una colisión, se comprueba la etiqueta del objeto con el que se ha producido y si es de tipo enemigo, se destruye el propio objecto y se pasa a la siguiente escena. Si es de tipo objetivo, se llama a la animación de destrucción de este, se aumenta el marcador y se comprueba si quedan mas, para que, en el caso de que no, pasar también a la siguienet escena.

“‘c# public class DieOnCollide : MonoBehaviour {

```
public string enemyTag;
public string targetTag;
private int numEnemies;
void Start(){
    numEnemies =  GameObject.FindGameObjectsWithTag(targetTag).Length;
}

void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == enemyTag) {
        Destroy(gameObject);
        GlobalGameManager.getInstance().loadArkanoidWaterPipes();
    }
    if (collision.gameObject.tag == targetTag) {
        collision.gameObject.GetComponent<Animation>().Play();
        ScoreManager.getInstance().pointUp();
        numEnemies--;
        if(numEnemies == 0){
            GlobalGameManager.getInstance ().loadArkanoidWaterPipes();
        }
    }

}
```

} “‘

Los mencionados son los mas importantes, ya que son los que dotan de lógica al juego, aunque existen más de los que se hacen uso como el globalGameManajer, que es el encargado de navegar entre escenas, ScoreMarkerObserver y TimeMarkerObserver, que pintan en el canvas el estado del juego o un script que se añade a la escena para controlar el back de nuestro dispositivo. Además, hay también otros scripts sencillos que controlan el resto de los objectos de la escena.

Conclusiones La RA tiene muchos usos como se menciona en el estado del arte. En este caso, el objetivo principal no es el de aprovecharnos de las dinámicas que nos provee para convertirlas en objeto de entretenimiento, si no el de dotar de dinamismo una parte del museo, ya que aportamos información de uno de los objetos expuestos en forma de videojuego (una versión propia que funcionaba en dicha máquina).

Una de las partes complejas en el desarrollo del juego, es la de configurar una forma de jugar en la que se le haga posible al usuario interaccionar con su dispositivo mientras apunta al target. Para esto se han hecho diferentes pruebas, y se ha llegado a la configuración actual teniendo en cuenta los siguientes factores:

- **El usuario** tiene una posición incómoda al utilizar su smartphone. Él tiene que apuntar a la imágen mientras interacciona con el teléfono para evitar que la bola caiga. La mejor solución en este caso es la de implementar una mecánica basada en detectar el arraste del dedo de la pantalla y que sea este movimiento el que desplace la lataforma en el juego. De esta forma, el usuario puede apuntar sin problema al QR mientras a su vez puede jugar. Esta forma de hacer las cosas consigue los dos objetivos, una mantener la cámara fijada en el target, y dos la de hacer cómodo junto con esto el poder manejar la plataforma.
- **La complejidad del juego**. En este caso se implementa un juego sencillo, muy intuitivo y corto, porque el jugador, por la posición que tiene, no puede pasar mucho tiempo jugando. El juego tiene como máximo un minuto de duración. Y es creemos que es la apropiada ya que en este período se puede jugar sin cansarse y acabar el juego si se tiene la habilidad suficiente sin que el cansancio de la posición evite dicho propósito.
- **La escena**. Es cierto que el modelo 3D del televisor no ayuda mucho a la jugabilidad. Pero en este caso en el que recreear una escana es casi mas importante que la experiencia del jugador en el juego. Se ha preferido penalizar un poco la jugabilidad (tampoco en exceso) para añadir este elemento de forma que añadiendo este objeto a la escena, se haga una referncia a la forma de jugar de la época.

[imagen de un usuario apuntando al target y jugando]

Con este juego respecto al desarrollo de RA con *Unity* y *Vuforia*, las conceptos sintetizados son:

- Aunque se trate de una escena 3D el uso de componentes 2D en los objetos de la escena simplifican mucho el trabajo, ya que al no tener que contar con un eje más; controlar el movimiento de la bola, los rebotes y las colisiones es mas sencillo. Al fin y al cabo es como mantener el eje restante en un objeto 3D bloqueado pero esto lo hace por ti, por lo que te ahorras muchos quebraderos de cabeza a la hora de controlar casuísticas que se pueden escapar por ser detalles tan pequeños.

- El manejo de los componentes *Collider* y *RigidBody* que en esta escena hay que modificar el material por defecto para cambiar la lógica de la gravedad, el rozamiento, etc; que en este caso, no es el que está por defecto,ya que en este juego no hay nada de eso. Además hay que hacer uso de las constraints del *RigidBody* para poder bloquear las rotaciones y desplazamientos de algunos de los objetos de la escan que no interesan que puedan moverse.

- La depuración del juego en modo escena en vez de hacer uso de la cámara. En Vuforia es normal el uso de una webcam para probar tu escena, pero en este caso, para probar el juego, era incómodo el tener que estar usando dicho elemento, por lo que, desactivando dicha opción de depurar la webcam en el *ImageTarget* y colocando la *ARCamera* apuntando a nuestro modelo 3D podíamos depurar simplemente haciendo click en el play del IDE de Unity sin preocuparnos de apuntar al QR.

Lo malo de este planteamiento es que traía un problema consigo, los eventos de pantalla; por defecto, *Unity* no detecta como eventos de pantalla en un dispositivo Android el uso del ratón. Para solventar esto se hizo uso de una aplicación llamada *UnityRemote*, que lo que hace es mostrar en la pantalla de nuestro dispositivo la escena que se está ejecutando en el IDE; y de esta forma podemos capturar los eventos que se realizan sobre esta pantalla.

[imagen del depurador de Unity y la tablet capturando la escena]

- *OnTrackingFound()* es un método que que contiene el *ImageTarget* en uno de sus scripts por defecto, y quu en un principio, parece que se encarga de habilitar todos los hijos que cuelgan de él en el momento en que se reconoce el QR asignado. Pero no, uno de los problemas en el desarrollo fué, que aún sin tener enfocado el QR; aunque no se vieran los objetos de la escena, todos los componentes de estos si se ejecutaban. Esto es, porque lo único que hace el método *OnTrackingFound()*, solo es renderizar los objetos en escena cuando se detecte el *Imagetarget*, por lo que se modificó este método para, además, los scripts, que deshabilitados por defecto al comienzo de la escena, se habilitarán al llamarse este método.

SPACE INVADERS

Historia Quizá uno de los juegos *arcade* clásicos más conocidos. La primera versión salió al mercado en 1978, hace casi cuarenta años. Uno de los precursores del género *shoot 'em up*. El jugador controla una nave espacial que se mueve horizontalmente y debe hacer frente a *hordas* de alienígenas enemigos que atacan al jugador disparándole proyectiles. Además, aveces el jugador cuenta con pequeñas construcciones que hacen la labor de *búnker* donde ponerse a cubierto de los disparos, aun que éstos se van destruyendo.

[imagen space invaders clasico]

Éste juego está ampliamente extendido en la cultura popular ya que es uno de los grandes clásicos, por eso hemos considerado acertado incluirlo en nuestro proyecto.

Nuestra versión Nosotros hemos decidido darle un cambio a la jugabilidad del juego, y cambiar el sistema. En nuestra versión utilizamos la RA para que la experiencia sea completamente diferente. Nuestro juego arrancará al detectar el cartel de **FACULTAD DE INFORMÁTICA** (*Text Recognition*), mostrándonos unos invasores alienígenas sobre el cartel, y unas *defensas* bajo éste.

[imagen de como empieza el juego en el cartel de la facultad]

Para destruir a los invasores, lo que tenemos que hacer es mover nuestro *smartphone* para mover nuestra cámara y pulsar en la pantalla para realizar el disparo. Hemos pasado de manejar la nave defensora en tercera persona, a hacerlo en primera persona, con un punto de mira en el centro de la pantalla que nos marca en qué dirección irán los láseres de nuestra *torreta de defensa*, convirtiendo el juego en un *First Person Shooter*, y tendremos que hacerlo antes de que los enemigos consigan destruir el escudo de nuestra nave espacial (cuando pasa del verde al rojo). Iremos obteniendo puntos según destruyamos naves enemigas, y perderemos puntos al recibir impactos en el escudo, por lo que cuanto más rápidos seamos, más puntos obtendremos.

Con estos cambios, hemos conseguido (a nuestro juicio), transformar un clásico de los *arcade* en un nuevo juego que utiliza la RA dando una experiencia diferente.

Implementación Durante el desarrollo de este juego hemos encontrado unos cuantos escollos que superar, algunos que nos han llevado quebraderos de cabeza. Comenzamos desarrollando el videojuego utilizando un código QR como *activador* de la RA, pensando que después pasar a utilizar un texto no tendría complicaciones. Una vez teníamos el juego desarrollado con un código QR (*ImageTarget*), probamos a detectar texto propio, ya que Vuforia nos da un diccionario con miles de palabras en inglés, pero nosotros no queríamos detectar esas palabras, si no únicamente **FACULTAD DE INFORMÁTICA**.

Para ésto, seguimos los tutoriales de Vuforia, y, tras resolver algunas dudas, implementamos una escena sencilla en la que se mostraba una esfera sobre el texto. Después, intentamos transferir lo desarrollado con el *ImageTarget*, al texto.

Entonces surgieron los problemas. El primero que vimos, era que las proporciones de nuestros *Invasores* y las *Defensas* se quedaron muy pequeñas, haciendo imposible jugar cómodamente. La mejor manera que conseguimos para que se ajustaran los elementos del juego a un tamaño aceptable fue mediante un Script de C# que aumentaba la escala local (*local scale*).

El siguiente problema que encontramos fue al *insertar* nuestro Enjambre de *Invasores*. Por un lado, una vez aparecían los enemigos, si movíamos la cámara, éstos se quedaban en la misma posición respecto a la cámara, es decir, si cuando salían por primera vez estaban en la parte superior izquierda (por ejemplo) de la pantalla, y nos movíamos, seguían ahí, en vez de ajustar su posición con respecto al texto detectado. Tuvimos que cambiar la forma en la que los insertábamos en la escena, antes de manera dinámica y creando una instancia con un Script, y ahora como hijos del *GameObject* que representa al texto detectado. Éste tipo de problema (de la posición respecto a los objetos de RA) nos volvería a salir más adelante, pero con los proyectiles que lanzábamos para acabar con los enemigos.

En las primeras versiones del juego, lanzábamos un prisma (nuestro *Proyectil*) contra los enemigos. Según lanzábamos, el proyectil se iba dirigiendo en la dirección que tenía la cámara respecto al *ImageTarget* al realizar el disparo. Al pasar a utilizar el texto, ésto cambió. Vimos que nuestro disparo se mantenía siempre en el vector de dirección de la cámara, y cambiaba con éste. Es decir, nuestro proyectil **siempre** estaba en el centro de la pantalla, con lo cual se perdía toda la gracia al juego y su jugabilidad pasaba a ser bastante complicada. Éste problema nos dejó bastante confusos, ya que con cambiar el objeto de la RV (ImageTarget o TextRecognition) cambiaba el comportamiento del proyectil.

Para resolver ésto, cambiamos nuestro proyectil por un *láser*. Ahora al tocar la pantalla no se *lanza* un proyectil, si no que se dispara un *láser* que destruirá los enemigos que estén en el vector de dirección de la cámara. Así hemos conseguido solventar este extraño comportamiento.

Diseño El juego se compone de una única escena que contiene todo el juego. Básicamente se compone de:

- La cámara de Vuforia, que a su vez tiene los siguientes hijos:
  - El *canvas* con la **Interfaz de usuario** (puntos y mensajes de inicio y fin del juego).
  - El punto de mira que utilizamos para apuntar al disparar, que también está hecho con un *canvas*.

  - El **Cannon** (Cañón de disparo) que representa nuestra arma. Básicamente dibuja una línea hacia el infinito para que de la sensación de un puntero láser para apuntar, además, desde su posición se lanza el *raycast* que calcula las colisiones con los posibles enemigos.

- Un *GameObject* vacío llamado **SpaceInvadersGame** que contiene la clase *singleton* que gestiona el juego y la información mostrada por la interfaz.
- El **TextRecognition** que sirve para cargar la detección de textos. A éste le hemos añadido un diccionario propio de palabras para poder leer texto en castellano. El diccionario contiene únicamente dos palabras, **facultad** e **informática**. Hemos configurado el TextRecognition de manera que sólo busque las palabras que están en su *white list* (lista blanca), que son las dos antes mencionadas, así las operaciones son más ligeras ya que no tiene que comprobar las miles de palabras.
- **Word** representa a una palabra detectada por Vuforia. Se puede configurar para que represente cualquier palabra detectada o alguna en particular. Nosotros lo utilizamos para representar en particular la palabra **INFORMÁTICA**. Éste es el *GameObject* que sustituye al *ImageTarget* que utilizábamos en el pasado. Al detectar la palabra "INFORMÁTICA" en la cámara de RA, activa sus hijos y "avisa" al gestor del juego de que debe empezar a ejecutarse.
- Un Enjambre, que contiene la lógica para crear varios Invasores y posicionarlos a cada uno en su sitio, así como para moverlos todos juntos.
- Las copias de los invasores, las cuales disparan aveces a las defensas.

[imagen de los invasores]

- Las defensas, un objeto en tres dimensiones que representa a las defensas del jugador. Van cambiando de color, desde el verde al rojo según van recibiendo impactos de los invasores (o del propio jugador que apunta mal, para ser algo más realistas).

Desarrollo Pasamos a explicar qué clases componen el juego y para qué las utilizamos.

- **Defense.cs**: gestiona las defensas del usuario. Marca el color de inicio y el de final que debe tener la defensa para calcular los colores intermedios. Además gestiona las colisiones.
- **Projectile.cs**: muy simple. Va asociada a los proyectiles y los destruye al pasar unos segundos en escena. Es para que los proyectiles que no impacten con nada, no se queden siempre en la escena.
- **Enjambre.cs**: se encarga de gestionar la inicialización del Enjambre y de sus invasores (colocándolos en la posición que les corresponda en función de cuantos sean y cuantas filas queremos que haya) y el movimiento del

Enjambre (del que "cuelgan" los invasores), así como la escala de los invasores. Además contiene la información para saber si se han eliminado a todos los invasores o no.

- **GameManager.cs**: es la clase que gestiona el juego en si. Es un singleton y se le llama desde la mayoría de los otros scripts. Gestiona la interfaz de usuario, mostrando mensajes y los puntos cuando empieza el juego, además de cuando se puede disparar, etcétera.
- **Invader.cs**: lógica del invasor. Gestiona los disparos de los invasores, la muerte de éstos y el sonido que hacen al ser destruidos.
- **TextInformaticaTrackableEventHandler.cs**: implementación propia de la clase *ITrackableEventHandler* de Vuforia. Va asociada al Text de RA. Al "encontrarse" y si no está instanciado ya (es decir, que no se ha "encontrado" varias veces), le dice al GameManager que comience el juego, indicándole dónde están el Enjambre y las Defensas en relación al Text.
- **TextTimer.cs**: de manera muy sencilla destruye el texto (de la interfaz gráfica) al que está asociado al pasar un tiempo dado una vez se ha habilitado. Lo utilizamos para mostrar los mensajes de texto de información.

Conclusiones

WATER PIPES

Historia

Este juego fue creado a finales de los años 80 bajo el nombre de "Pipe Mania" por "The Assembly Line", teniendo muchas versiones a lo largo de los años. Las primeras fueron realizadas por el estudio "LucasFilm Game" que lanzó el juego "Pipe Dream". Aunque posteriormente fueron lanzadas otras versiones del juego para PC, PS2, NintendoDS y PSP.

En la versión original, el juego consistía en ir colocando las tuberías que salían de forma aleatoria en un tablero (matriz de NxN) de tal manera que el agua pudiera fluir por ellas. El objetivo, era construir el mayor recorrido posible antes de que el agua lo inundara todo. Cada tubería llena de agua sumaba puntos y cada tubería que no hubiera sido inundada a la finalización del juego restaba puntos para obtener así la puntuación final. En cada nivel se iban complicando las cosas, esto lo conseguían poniendo posiciones de la matriz de juego en las cuales no se pudieran colocar tuberías. O comenzando antes a fluir el agua por las tuberías, dando al jugador menos tiempo de reacción.

Nuestra version

En nuestra versión del juego, existen algunas diferencias con respecto a la versión original. La principal, es la mecánica del juego ya que en está ocasión

el usuario acaba de exterminar al enemigo que estaba poniendo en peligro a la facultad, y ahora su misión cambia para poder despegar la nave y ponerse a salvo definitivamente. En esta última misión el jugador se encontrará frente a frente con los conductos de refrigeración de la nave, el problema es que debido a la batalla estos se han dispersado y si quiere despegar la nave tendrá que reordenarlos para que el agua pueda fluir correctamente. Al comienzo, el jugador verá una matriz rellena aleatoriamente con estos conductos, además de un salida y una meta. Por lo tanto, el objetivo del jugador es ir cambiando de posición las tuberías que encuentra en la matriz, entre sí, para que el agua pueda fluir del punto de origen al de destino. En esta ocasión, el jugador tiene 6 segundos por cada tubería para ir colocando las demás y otros 6 segundos adicionales desde que el juego comienza hasta que la casilla de salida comienza a llenarse. Para poder despegar la nave, el agua ha debido fluir por todas las tuberías necesarias para llegar a la meta. En cambio si el agua ya ha comenzado a fluir y el jugador no ha conseguido recolocar los conductos, en cuanto el agua no encuentre un camino factible la nave ya no se podrá despegar y el juego terminará.

Implementacion

Diseño

- Existen dos elementos fundamentales, en cualquier aplicación de AR, y que nos lo proporciona la librería de Vuforia, la AR Camera y el Image Target.
  - **AR Camera**: Es la cámara del juego, y a nivel de usuario del unity funciona como "main camera" de otros juegos sin AR.
  - **Image Target**: Este objeto también nos lo proporciona Vuforia, y cuando lo configuremos con el código QR (u otra imagen o texto) será el responsable de relacionar la escena con lo que nuestro dispositivo móvil este leyendo.
- Para este juego existe un solo tipo de objeto, el quad que representan a los conductos. Todos estos objetos son hijos del Image Target.
  - **Quad**: todo el juego se compone de 25 (matriz de 5x5) objetos Quad que representan los conductos de refrigeración. A cada objeto se les han incluido los mismo componentes:
    - *Box Collider*: para detectar la colisión y así poder intercambiar dos objetos entre sí, en este caso, solo existe la colisión entre el usuario y los objetos (Touch).
    - *Rigibody*: Elimina la gravedad de los objetos y congela ciertos movimientos de desplazamiento y rotación de estos, en este caso hemos querido bloquear todas las rotaciones y la dimensión Z del desplazamiento. Esto es debido a que aunque es un juego en 3D, la experiencia con el jugador es totalmente en 2D y esos movimiento no nos serán necesarios.

- **Animación**: Aunque el único objeto "visible" en el juego sean los conductos (Quad), cada uno de ellos contiene un GameObject como hijo. La funcionalidad de esté es la animación del agua cuando pasa por la tubería, por lo que cada uno de estos "hijos" contienen también los mismos componentes cada uno:
  - *Sprite Renderer*: Esto es necesario, ya que la animación se compone de un sprites.

  **Animator*: que contiene el Animator Controller de ese tipo de tubería. Cada controlador suele tener dos animaciones por tubería, que dependiendo de la salida de la anterior se reproduce una animación u otra.
- **Canvas**: Muestra la información del juego, en este caso se encarga de mostrar el tiempo restante que le queda al jugador y cuando el juego finaliza muestra la pantalla de marcadores, para que el jugador pueda introducir su nombre y así entrar en la lista de clasificados.
- **GameManager**: Es un GameObject vacio en la escena que contiene los scripts para la funcionalidad del juego:
  - *GameManagerWaterPipes*: es el encargado de generar toda la escena al inicio del juego.
  - *WaterController*: desde este script, se controla todo el flujo del agua.
  - *TimeController*: clase que decrementa el tiempo de juego y el tiempo que tarda el agua en pasar por cada conducto.

Desarrollo

TouchObject Este script es el encargado de controlar el cambio de los conductos. Para ello, lo primero que hace es capturar en el método Update todos los toques en la pantalla y llamar a getObjectHit para obtener el objeto presionado. Este objeto lo obtenemos a través del método "GetNearestHitGameObject", que genera un rayo con origen en la AR Camera y con la dirección generada por el rayo. Y de esos se queda con el que haya menor distancia.

Una vez tenemos el objeto presionado, tenemos que saber si ya existe otro objeto pulsado para poder intercambiarlos, o si es el primer elemento que queremos cambiar. Si es el primero, lo único que haremos es cambiarle a ese objeto el tag a "ButtonPush" y si es el segundo, obtenemos el que ya hemos pulsado anteriormente e intercambiamos su posición.

GameManager Es el encargado de crear dinámicamente todos los objetos de la escena al comienzo del juego, y de llevar el control del tiempo que tiene el jugador para completar la misión. Lo primero que hace es añadir los objetos

prefabs de cada tipo de conducto(Horizontal, Vertical, Codo Derecho Arriba, Codo Derecho Abajo, Codo Izquierdo Arriba y Codo Izquierdo Abajo) en una lista de SquareReceiver para luego ir eligiendo de manera aleatoria los 23 conductos totales. A continuación crea la salida y la meta, ya que estos siempre van a tener una posición fija en el tablero. Y cuando ya está todo listo, lo único que falta es ir creando el resto del tablero. Va recorriendo todo el tablero desde la casilla 1 hasta la 23 y por cada una escoge aleatoriamente un tipo de conducto, de la lista anterior. Crea una instancia de ese tipo de objeto que será un hijo del Image Target. Dándole además las propiedades como la posición, el nombre, y la escala. Esta instancia la guardamos en un array del tipo SquareReceiver para luego poder acceder a él. El método Update controla el tiempo del juego.

WaterController Este script es, junto con el gameManagerWaterPipes, el más importante, ya que se encarga de controlar el flujo del agua, que es la funcionalidad básica de nuestro juego. En nuestro caso, el flujo de agua funciona de la siguiente manera:

Por cada tipo de conducto el agua puede fluir en dos sentidos cada vez, por lo tanto a partir de la entrada del agua de la casilla anterior se calculará la casilla siguiente por la que el agua debería de fluir. Y una vez que ya sepamos cual es la siguiente casilla por la que tenemos que llevar el agua, podremos comprobar si esa casilla es válida, es decir su entrada de agua coincide con la salida de agua anterior.

ESCENAS INTERMEDIAS

Entre cada escena; o juego del desarrollo, se ha establecido una intermedia donde un "humanoide" nos pone en situación antes de cada misión y nos sitúa acerca de cómo se ha llegado a dicha situación y a dónde debemos ir para resolverla.

Gracias al paquete de *Unity* animación por voz, *SAMBA*, el proceso de sincronizar el audio con la animación del discurso del modelo ha sido algo muy sencillo. Dicha animación consta de tres componentes:

1. El modelo, que en este caso era un *prefab* que venía configurado por defecto para soportar el componente de audio y de enfoque aleatoria.
2. Componente de animación de los músculos faciales, al cuál se le asigna un conjunto de audios de tal forma que la cara del modelo se articula de forma sincronizada con el audio proporcionado. El componente viene configurado para que el audio que le proporcionemos al modelo pueda ser interpretado por este con mas o menos enfásis o con diferenes estados de ánimo.
3. *Random eyes* es un componente que, asignado al modelo, articula sus ojos de tal forma que definiendo unos puntos objetivo, alterna y gesticula mirando a los diferentes objetivos a lo largo de la animación del objeto.

Además del modelo utilizado, también hubo que crear los audios que narran el hilo argumental de nuetro juego. Para esta tarea, se generarón los audio con la herramienta de la url *http://vozme.com/index.php?lang=es* que convierte a *.mp3, con una voz un "robótica", un texto dado.

En último lugar, la escena contiene también un texto donde se muestra de forma escrita todo lo que se va narrando. Esto es así porque pueden haber problemas de audio, o el usuario puede tener la necesidad de volver a ver el mensaje y sintetizar lo que el "robot" ha dicho para poder encontrar el próximo lugar donde aparecerá el siguiente minijuego.

[imagen de una escena intermedia]

SISTEMA DE PERSISTENCIA DE PUNTUACIONES

Al finalizar el juego, el usuario puede almacenar su puntuación, y ver el ranking de estas. Este sistema se hizo para enlazar el juego con lo que esun servicio web, ya que nos parecía una práctica interesante el hacer uso de la parte cliente que nos ofrece *Unity* para comunicarnos con servicios web. ;Se han implementado ambas partes en este caso, tanto el lado cliente que consume los servicios en *Unity*, como el lado del servidor, en el que se ha implementado un sistema de gestión de usuarios con una sencilla api REST en *PHP*, haciendo uso del framework *Symfony 2*.

Desarrollo

Diseño

- Base de datos: Se genera una base de datos SQL con una sola tabla, la de usuarios y los campos que deseabamos guardar: id, nombre y puntuación.

[imagen de phpMyAdmin tabla de usuarios]

- Operaciones: Para la parte REST; los servicios, solo pueden realizar las operaciones de leer y crear nuevos usuarios. En el panel de administración, se pueden realizar las *CRUD*, crear un nuevo usuario, leer los usuarios y ver el usuario en detalle, actualizar y eliminar.
- Panel de administración: Se accede con el usuario administrador a través de la siguiente URL http://augmentedreality.hol.es/web/users/ y en ella se pueden realizar las descritas en el apartado anterior.
- REST: la URL base es http://augmentedreality.hol.es/api/ y las operaciones son; de tipo *GET* **/users.[json | xml]** para obtener un listado de los usuarios y sus puntuaciones en uno de los formatos especificados.

Y para guardar la puntuación de un usuario, de tipo *POST* **/api/users** dónde se les manda, en el body de la petición, la puntuación del usuario y el nombre de este.

- Seguridad: De lado del servidor, *Symfony* nos provee de un sistema de seguridad basado en tres factores; la dirección a la que se accede, la autenticación para acceder a ella y una vez auténticado, el rol que tiene dicho usuario para poder acceder. En nuestro caso hay dos usuario. "admin", con el rol de administrador, que le permite entrar tanto en el panel de administración, como hacer uso de la api. Y "api_user", un usuario cuyo rol solo le permite usar los servicios web. El tipo de autenticación es a través de *Basic auth*. Consiste en añadir un campo en el *header* con la clave "Authorization" y como valor, la palabra "Basic" concatenada y separados por un espacio con la combinación de "username:password" codificada en *Base64*.

Implementación La implementación del sistema de persistencia se compone de dos partes, la parte del cliente, dónde se consumen los servicios web; y la parte del servidor, donde se configura el sistema.

Servidor

El lenguaje utilizado para implementar esta parte es PHP. Sobre este lenguaje se utiliza un framework de desarrollo llamado Symfony, que facilita mucho la tarea de construir este tipo de sistemas. Symfony hace uso de trabajar basada en el *Modelo Vista Controlador* y el flujo de trabajo sobre este framework consiste en simplificarte, bajo esta arquitectura, un montón de procedimientos y líneas de código que se repiten constantemente en este tipo de software.

A continuación se describe de forma esquematizada los pasos más importantes para desarrollar esta parte del proyecto.

1. Instalación: Primero en local, y después bajo un host de pruebas (hostinger.es) se instala la versión 2.8 de symfony, con su paquete por defecto, bundle a partir de ahora. A est,e se le instalán además dos *bundles* externos, los cuales se encargan de facilitar el proceso de generar la parte *REST*. Estos son *JMSSerializerBundle* y *FOSRestBundle*; se descargan, y se añaden el el archivo AppKernel.php.

```
php ...  public function registerBundles() { $bundles = array( ...
new FOS\RestBundle\FOSRestBundle(), new \JMS\SerializerBundle\JMSSerializerBundle(),
new AppBundle\AppBundle() ); ...
```

2. Controladores: En este caso como, el proyecto era sencillo, sobre el propio AppBundle (Bundle por defecto de Symfony) se crean dos controladores, uno para la Api y otro para el panel, la parte web.

3. Modelo: Se crea la base de datos con la tabla de usuarios. A continuación, se meten los credenciales de esta sobre el archivo parameteres.yml para interactuar con ella. Una vez creada, se implementan, en el bundle por defecto, la entidad que serializa los datos, *Users.php* y el script *UserType.php*, que se encargará, construyendo un formulario, de comunicar el la vista y el controlador.

```
php /** * Users * * @ORM\Table(name="users") * @ORM\Entity */
class Users { /** * @var integer * * @ORM\Column(name="id", type="integer",
nullable=false) * @ORM\Id * @ORM\GeneratedValue(strategy="IDENTITY")
*/ private $id; /** * @var string * * @ORM\Column(name="name",
type="string", length=40, nullable=false) */ private $name; /**
* @var integer * * @ORM\Column(name="score", type="integer",
nullable=true) */ private $score; //getters and setters ...
```

4. Acciones del modelo: haciendo uso *Doctrine* (el *ORM* de *Symfony*, que se encarga de crear una capa intermedia que permite interactuar con la base de datos sin tener que utilizar el lenguaje de esta, en este caso SQL), se generan las operaciones CRUD en el controlador del panel, *UsersController.php*, y con ayuda de *FOSRestBundle*, las operaciones de mostrar y crear un nuevo usuario en el controlador de la api, *UsersRestController.php*.

```
php class UsersRestController extends FOSRestController { public
function getUsersAction() { $em = $this->getDoctrine()->getManager();
$users = $em->getRepository('AppBundle:Users')->findBy( array(),
array('score' => 'DESC') ); return array( 'users' => $users ); }
public function getUserAction($id) { $em = $this->getDoctrine()->getManager();
$user = $em->getRepository('AppBundle:Users')->find($id); return
array( 'user' => $user ); } public function postUserAction(Request
$request) { $user = new Users(); $form = $this->createForm(new
UsersType(), $user); $form->submit($request); if ($form->isValid())
{ $em = $this->getDoctrine()->getManager(); $em->persist($user);
$em->flush(); } return array( 'form' => $form ); } }
```

5. Rutas: En el archivo *routing.yml*, se generan dos rutas. Una que hace referencia al controlador de la parte web, donde se indica que las rutas van definidas en las accciones del controlador, indicando en el campo type *annotation*. Y la otra hace referencia a los servicios: indicando que las rutas quedan definidas por el nombre de la acción, indicando como type *rest*. Una de las funciones de FORestBundle.

```
yml app_web:  resource:  "@AppBundle/Controller/Web" type:  annotation
app_api:  type :  rest resource :  "@AppBundle/config/routing_rest.yml"
prefix :  /api
```

6. Seguridad: En el archivo *security.yml* se definen dos usuarios; “admin” y “api_user”, el primero con role de administrador y el segundo con rol de acceso a la api. A continuación se define el firewall, que deshabilita define que patrones de URL están permitidos de forma anonima y que indica que tipo de autenticación se va utilizar; en este caso *http basic*. Y en último lugar, se definen las rutas que necesitan permisos para acceder y que roles pueden acceder a ellas.

“‘yml security:

```
# http://symfony.com/doc/current/book/security.html#where-do-users-come-from-user-providers
providers:
    in_memory:
        memory:
          users:
            admin: { password: $2y$12$xmPAOHtOPJpNz/83dXYOHuTi3hPBv3bc/ZPh2ZD.0bOGJtpOOVzWW
            api_user:  { password: $2y$12$UDuOn4JCYK8ZibPqkDxXW.CFJHPkGTcvjmdAIcKZiGgUaEtqkV

firewalls:
    # disables authentication for assets and the profiler, adapt it according to your needs
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false

    main:
        pattern: ^/
        anonymous: ~
        http_basic: true

access_control:
    - { path: ^/api, roles: ROLE_API }
    - { path: ^/web/users, roles: ROLE_ADMIN }

encoders:
        Symfony\Component\Security\Core\User\User:
                algorithm: bcrypt
                cost: 12
```

“‘

Cliente

En esta parte se define como se consumen con unity los servicios implementados en el lado del servidor.

1. *Autenticación*: se define un diccionario a partir de del atributo headers de WWWForm (new WWWForm().headers). A este se le añade la clave

"Authorization" y como valor la cadena "Basic" más la codificacón base64 de la caderna formada por el nombre de usuario seguido de el carácter ':' y la contraseña.

```
c# private Dictionary<string,string> getAuthenticationHeader()
{ Dictionary<string,string> headers = new WWWForm().headers;
headers["Authorization"] = "Basic " + System.Convert.ToBase64String(
System.Text.Encoding.ASCII.GetBytes(USERNAME + ":" + USERPASSWORD));
return headers; }
```

2. *GetScore*: se instancia un objeto WWW, y en el constructor se añaden la url del servicio, y el diccionario de autenticación.

```
c# public WWW getScores() { return new WWW(URLBASE + "/api/users.json",
null, getAuthenticationHeader()); }
```

3. *PostScore*: el método es similar al del Get solo que en esta ocasión, además hay que generar el body creando un objeto de tipo WWWForm y añadiendo el campo "user[name]" con el nombre de la puntucación, y la puntuación en el campo "user[score]".

```
c# public postScore(string name, int score) { WWWForm form =
new WWWForm(); form.AddField ("user[name]",name); form.AddField
("user[score]",score); return new WWW(URLBASE + "/api/users",
form.data, getAuthenticationHeader()); }
```

4. Encapsulamos la petición para que se realice en un segundo plano y esperamos la respuesta.

```
c# public void OnClick(){ GetComponent<Button> ().interactable =
false; string name = input.text; StartCoroutine(WaitForRequest(RestApi.getInstance().postSc
input.text = "loading..."; } private IEnumerator WaitForRequest(WWW
www) { yield return www; if (www.error == null) { ...  } else {
...  } }
```

Conclusiones Este apartado, aunque quizás no estaba del todo realacionado con la RA, a nivel académico ha sido algo útil. Ese tipo de configuraciones nunca está de más conocerlas, ya que en el mundo laboral están a la orden del día. Además, no solo se aprende a configurar un sistema de persistencia de puntuación, si no se queda montado una base para un sistema de conexión a internet, que puede favorecer a futuras ampliaciones del proyecto.

Por otro lado; de cara al juego, serviría como aliciente para que los jugadores volvierán a repetirlo, con los conceptos de como funciona el juego mas claros y con idea de mejorar su puntuación anterior o a sus compañeros. Se podría decir que mejora un poco la experiencia del usuario y hace mas completo el proyecto.

BIBLIOGRAFIA

https://es.wikipedia.org/wiki/Realidad_aumentada#Definiciones

http://www.ronaldazuma.com/publications.html