Universidad de Cantabria

Facultad
de
Ciencias

ChronoStreetTurist3.0
(ChronoStreetTurist3.0)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERIA INFORMÁTICA

Autor: Martín del Saz-Orozco Bolado

Director: Pablo Sánchez Barreiro

Co-Director: Ánder Bermúdez González

Junio - 2025

Agradecimientos

En primer lugar, quiero agradecer a mis padres María y Pepe, y a mi novia Maja por la grandísima ayuda y motivación que han supuesto para mí a lo largo de estos 4 años de aprendizaje. En segundo lugar, a mi tutor académico Pablo y cotutores Luis y Ánder de HP por haber estado al pie del cañón durante todo este proyecto, permitiéndome desarrollarlo de la mejor manera posible. Por último, quiero agradecer a todos mis, primero, compañeros de carrera y que luego han acabado siendo unos grandes amigos, gracias a los cuales estos 4 años se han pasado volando.

Contenido

Resumen

El objetivo de este trabajo es la evolución de una aplicación conocida como *ChronoStreetTourist*. En su primera versión la aplicación solo permitía superponer imágenes antiguas de monumentos sobre imágenes actuales haciendo uso de la cámara del teléfono mediante técnicas de realidad aumentada. En la segunda versión se introdujo un sistema para la socialización de la aplicación, permitiendo a los usuarios añadir nuevos monumentos y actualizando varios aspectos visuales de la aplicación. En esta tercera evolución se ha querido actualizar la arquitectura del sistema para independizarla de servicios de alojamiento de imágenes que estaban dando problemas, añadir un mapa que permita visualizar todos los monumentos que se encuentran actualmente alojados en la aplicación y por último integrar un sistema de notificaciones que avise a los usuarios cuando se encuentren cerca de un monumento contenido en la aplicación.

Palabras clave

Realidad aumentada, Unity, Vuforia, Monumentos, Aplicación móvil, Android, C#, JavaScript

Abstract

The objective of this work is the evolution of an application known as ChronoStreetTourist. In its first version, the application only allowed the overlay of old images of monuments onto current images using the phone's camera through augmented reality techniques. In the second version, a system was introduced to promote social interaction within the app, allowing users to add new monuments and updating various visual aspects of the application. In this third evolution, the goal has been to update the system architecture to make it independent from image hosting services that were causing issues, add a map to visualize all the monuments currently hosted in the application, and finally integrate a notification system to alert users when they are near a monument included in the app.

Keywords

Augmented reality, Unity, Vuforia, Monuments, Mobile app, Android, C#, JavaScript

1. Introducción. Objetivos y Metodología

1.1. Introducción

El proyecto pertenece al programa Observatorio HP 2024-2025, el cual se nos ofertó en la Universidad de Cantabria a comienzos del curso universitario 2024-2025. A raíz de ahí surge esta oportunidad de evolucionar el proyecto *ChronoStreetTourist*, el cual cuenta ya con dos versiones anteriores. La primera versión la cual estaba plenamente centrada en el desarrollo de una realidad aumentada capaz de sobreponer fotos antiguas de monumentos sobre ellos en la actualidad. La segunda versión la cual se encargó de la socialización de la aplicación permitiendo a distintos usuarios subir nuevos monumentos a un nuevo almacenamiento en la nube. Desde el punto de vista del usuario, esta aplicación permite utilizar dos funcionalidades. Una de las funcionalidades es escanear monumentos con realidad aumentada, pudiendo ver superpuestas imágenes antiguas de estos encima. Por ejemplo, utilizando esta aplicación, se podría visualizar gracias a la realidad aumentada como era la puerta de Alcalá hace 100 años. La otra funcionalidad es agregar nuevos monumentos a la base de datos de la aplicación para que nuevos usuarios los puedan escanear.

Para realizar la superposición de imágenes, ChronoStreetTourist utiliza *Vuforia* [1], un motor de realidad aumentada. Vuforia tiene una base de datos, denominada *Vuforia Target Database*, con *targets*, que son las imágenes o localizaciones que tiene que reconocer junto con los metadatos de cómo superponer otras imágenes sobre estas localizaciones, o *targets*. Las imágenes a superponer deben estar alojadas fuera de esta base de datos, en el alojamiento de base de datos que se considere adecuado. En las versiones previas de *ChronoStreetTourist* se utilizaba *cloudaccess.net* para el almacenamiento de las imágenes a superponer. *cloudaccess.net* es un sistema de almacenamiento en la nube de archivos de todo tipo, desde imágenes hasta documentos pdf o vídeos. Este servicio requiere ser activado mensualmente, lo que generaba ciertas complicaciones. Por ello, se plantea la necesidad de sustituir *cloudaccess.net* por otro servicio de alojamiento con menos restricciones. Además, en esta nueva versión de *ChronoStreetTourist* se desea integrar un mapa que muestre la ubicación de todos los monumentos almacenados en la base de datos y diseñar un sistema que envíe notificaciones emergentes a los usuarios cuando estos se sitúen cerca de un monumento recogido en la base de datos. Estos objetivos se comentan con más detalle en el siguiente apartado.

1.2. Objetivos

El proyecto se divide en tres fases, cada una con un objetivo diferente.

1. Simplificación de la arquitectura de la aplicación.
2. Mapa con los monumentos.
3. Notificaciones emergentes al acercarse a un monumento.

1.2.1. Simplificación de la arquitectura de la aplicación

Este objetivo constituye la parte más compleja e importante del proyecto. Consta de los siguientes objetivos:

1. Volver a poner en funcionamiento la aplicación, puesto que la antigua arquitectura de esta constaba de dos partes (un servidor remoto y la aplicación en sí) de las

cuales el servidor remoto no se encuentra operativo. Este servidor remoto se alojaba en una plataforma de hosting llamada *cloudaccess.net*, la cual, si bien era gratuita, requería de una reactivación de la licencia gratuita de forma mensual. Por esta condición ha quedado en desuso y no se encuentra en funcionamiento la aplicación.
2. Hacer la aplicación funcional de manera independiente y para ello prescindir completamente del servidor remoto, siendo sustituido por un servicio o servidor el cual no requiera de activación periódica. El servicio elegido a utilizar es el que proporciona *Imgur*, puesto que es un servicio con acceso mediante una API tipo *Rest*, lo que permite un uso sencillo del mismo, y además es un servicio gratuito y sin restricciones de activación temporales.
3. Mantener la funcionalidad que ofrecía el servidor remoto, para poder contar con los metadatos e imágenes de los monumentos.

1.2.2. Mapa con los monumentos

Una vez conseguido el primer objetivo se quiere mejorar la aplicación. Integrando un mapa interactivo el cual muestre todos los monumentos almacenados en la base de datos. Para ello, se deberán realizar las siguientes operaciones:

1. Crear un mapa interactivo, el cual resulte familiar al usuario y fácil de utilizar.
2. Generar puntos de interés en el mapa que muestren la ubicación de los monumentos almacenados en la base de datos.
3. Mostrar detalles al ver cada punto de interés, mostrando su foto antigua, así como enlaces con información de interés sobre el mismo.
4. Permitir al usuario obtener su geolocalización actual para ubicarse en el mapa.

1.2.3. Notificaciones emergentes al acercarse a un monumento

Por último, se desean implementar notificaciones emergentes de la aplicación las cuales avisen a los usuarios cuando se encuentren cerca de un monumento disponible en la aplicación. Para ellos, se deberán llevar a cabo las siguientes operaciones:

1. En base a la ubicación actual del usuario, recorrer todos los monumentos o *targets* de la base de datos y buscar cual o cuales se encuentran a menos de 100 metros del usuario.
2. Notificar al usuario de que tiene un monumento cerca, en caso de haber encontrado un monumento en la base de datos que se encuentre a menos de 100 metros del usuario y no haya sido notificado a este previamente.

1.3. Metodología e Infraestructura

1.3.1. Metodología

Respecto a la metodología de este trabajo, el desarrollo cubre todo el ciclo de vida del software a excepción de la fase de captura de requisitos, la cual había sido realizada por la empresa. Se sigue una metodología ágil de tipo *scrum* [2]. Todas las fases del ciclo de vida se desarrollarán de manera habitual, exceptuando la fase de pruebas, de la cual solo se realizarán pruebas manuales al tener la imposibilidad de la automatización de las pruebas en el entorno de desarrollo elegido.

Dentro de la metodología ágil de tipo Scrum los roles se asignan de la siguiente manera:

- Scrum team: está compuesto por Martín del Saz-Orozco Bolado, el cual es el único desarrollador.
- Product owner: está compuesto por Ander y Luis, de HP SCDS.
- Scrum Master: este rol no lo adopta nadie en concreto.

Respecto al esquema del proceso de desarrollo se establecen los siguientes pasos:

- Reuniones cada 2 semanas entre el Scrum team y el Product owner: dichas reuniones cumplen a la vez el cometido de *Sprint Planning Meeting* como de *Product review*, puesto que sirven para comunicar lo que se ha realizado durante cada sprint y a la vez para acordar que se realizará en el próximo Sprint. Estas reuniones marcarán siempre el final de un Sprint y comienzo del siguiente.
- Las reuniones de tipo Sprint Planning Meeting (solo del Scrum team), Daily Scrum Meeting y Sprint Retrospective carecen de sentido en nuestro caso debido a que el Scrum team está compuesto por una sola persona.

A lo largo del proyecto en total se han llevado a cabo 6 sprints, desarrollando en cada uno de estos sprints una de las 6 historias de usuario que se habían especificado previamente. Para el seguimiento del proyecto se ha hecho uso de las facilidades que brinda *GitLab* para la gestión de proyectos. A continuación, la Figura 1 muestra a modo de ejemplo, el estado del tablero *Kanban* [3] de *Gitlab* para el sprint 4.

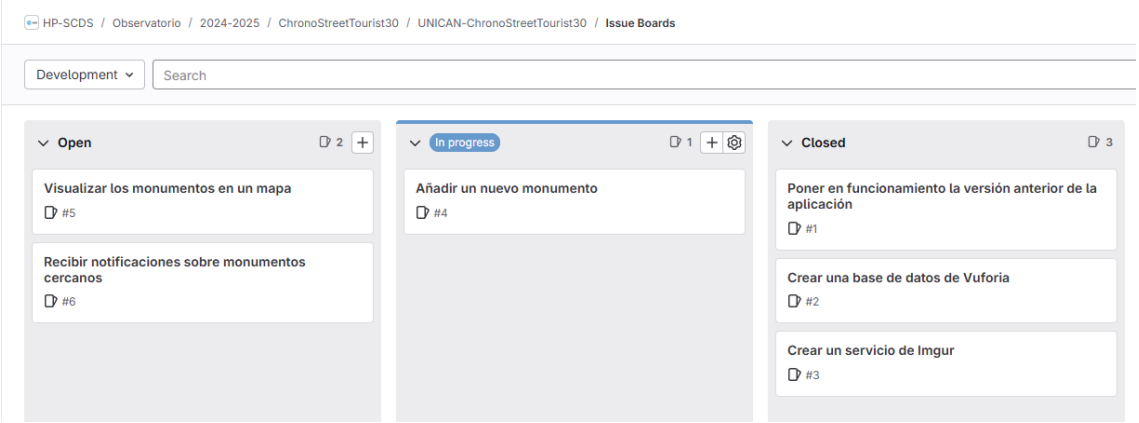

Figura 1. Estado del tablero en el sprint 4

Cabe destacar que este tablero no es exactamente equivalente al tablero Kanban que se utilizaría en una metodología ágil tipo *Scrum*, puesto que solo muestra el progreso de las historias de usuario y no el progreso de las tareas internas respecto a cada historia de usuario. Aun así, se ha decidido utilizar este al querer unificar las herramientas relativas a la metodología en un mismo lugar, y además al ser solo un desarrollador no se consideró estrictamente necesario la subdivisión en tareas al realizarlas todas uno mismo.

1.3.2. Infraestructura

Para el control de versiones del desarrollo se utilizó *Git* [4], en este caso almacenado en un repositorio de Gitlab, el cual proporciona HP como soporte al proyecto.

Respecto a la configuración de las ramas de dicho repositorio se siguió el esquema *Gitflow*, por lo que existían las siguientes ramas en el repositorio:

- *main*: rama donde se integrarán las distintas releases del desarrollo de la aplicación. Su tiempo de vida abarca desde el inicio hasta el final del proyecto.
- *develop*: rama donde se harán las fusiones de las distintas ramas de desarrollo con el objetivo de poder unir los nuevos desarrollos con los anteriores. Su tiempo de vida abarca desde el inicio hasta el final del proyecto.
- *feature*: ramas donde se desarrollarán las distintas funcionalidades establecidas para cada sprint del proyecto. Su tiempo de vida abarca desde el inicio hasta el final de cada sprint, de tal manera que existirá una rama por cada sprint realizado. En este caso se hace esta variante de *Gitflow* puesto que el proyecto esta desarrollado por una sola persona, en caso de haber sido desarrollado por más de una persona, se habrían creado tantas ramas *feature* como funcionalidades se hubieran desarrollado en cada sprint.

Respecto a la integración continua no se lleva a cabo en este proyecto, puesto que no existen pruebas automatizadas que se puedan ejecutar.

1.3.2.1. Herramientas de desarrollo

En este apartado se explicarán de manera breve y concisa todas las herramientas utilizadas clasificadas por funcionalidades.

Servicios Web y APIs:

- **API de Imgur**: Utilizada para alojar imágenes de forma externa y obtener URLs accesibles desde la aplicación.
- **Vuforia Cloud Database**: Servicio en la nube para reconocimiento de imágenes y Realidad Aumentada, integrado con Unity.
- **GitHub Pages**: Servicio de GitHub que permite alojar contenido estático (como páginas web) directamente desde un repositorio, usado para mostrar o servir contenido en la aplicación. Se utiliza para alojar la página con el mapa de monumentos.
- **Cloudflare Workers**: Plataforma para desplegar funciones en la nube, usada para lógica de servidor ligera o para servir contenido dinámico. En nuestro caso sirve como proxy para poder hacer peticiones desde GithubPages a la base de datos de Vuforia, debido a una serie de cuestiones técnicas muy concretas que se detallarán más adelante.

Entornos de desarrollo:

- **Unity**: Motor de desarrollo de videojuegos utilizado para crear entornos interactivos y experiencias 3D.
- **Visual Studio**: Editor de código enfocado a C# el cual en este caso sirve para la edición de los distintos scripts de Unity.
- **Visual Studio Code**: Editor de código multidisciplinar el cual en este caso se utiliza para el desarrollo de una web en HTML y JS.

Lenguajes de programación:

- **C#**: Lenguaje principal utilizado en los scripts de Unity.
- **JavaScript**: Usado para lógica en **Cloudflare Workers** y para la página en **GitHub Pages**.
- **HTML**: Utilizado para la creación del contenido web asociado al mapa de monumentos.

Plugins de terceros para el desarrollo en Unity:

- **ImgurSharp**: biblioteca de libre uso la cual permite subir fotos a *Imgur* haciendo un uso fácil de su API.
- **VuforiaWebService**: plugin NuGet el cual simplifica las llamadas a la API de VuforiaWebServices , pudiendo así interactuar con la base de datos en la nube que esta alberga.
- **Unity-webview**: plugin que permite introducir páginas web dentro de las distintas pantallas de la aplicación desarrollada en Unity.

2. Especificación de Requisitos y Arquitectura

2.1. Versión anterior de ChronoStreetTurist

Para poner en contexto el proyecto, en este apartado se explicará el funcionamiento de la aplicación original de manera breve y concisa de manera que se puedan entender mejor los requisitos de este proyecto.

La versión 2.0 de ChronoStreetTurist constaba de una interfaz inicial con 2 opciones; *Escanear un monumento* y *Añadir monumento*. Dicha interfaz se muestra en la Figura 2.

Figura 2. Interfaz de inicio

Al elegir la opción de *Escanear monumento*, como se muestra en la Figura 3, se accede a una interfaz que muestra lo que ve la cámara del dispositivo en uso, junto con 3 iconos:

- En la esquina superior izquierda, un icono de una *Cámara* el cual si se presiona hace una foto de lo que ve la cámara.
- En la esquina superior derecha, un icono que dice *AR* el cual se utiliza para activar o desactivar la realidad aumentada de los monumentos sobre la cámara del móvil.
- En la esquina inferior derecha, un icono de una *Casa*, el cual permite volver al menú principal de la aplicación.

Figura 3. Interfaz para Escanear monumento

Al elegir la opción *Añadir monumento*, se pasa a una nueva interfaz la cual tiene 2 opciones principales, como se puede observar en la Figura 4 (Izq.):

- *Automático*: Opción que conduce a la pantalla de *Añadir monumentos de forma automática*, la cual se explicará en detalle más adelante.
- *Manual*: Opción que conduce a la pantalla de *Añadir monumentos de forma manual*, la cual se explicará en detalle más adelante.

Además, en la parte inferior se encuentran dos iconos:

- En la izquierda, un icono de una *Casa*, el cual permite volver al menú principal de la aplicación.
- En la derecha, un icono de *Información*, el cual conduce a la pantalla de información, la cual se explica más adelante.

Al acceder a la interfaz para añadir automáticamente un monumento se muestra un formulario, que se puede observar en la Figura 4 (centro), el cual permite introducir:

- El nombre del usuario, que puede ser anónimo si se selecciona la opción.
- La imagen antigua del monumento que se quiere añadir.
- La URL que contiene información del monumento.

A continuación, se encuentran 2 botones:

- *Reestablecer campos*, que vacía la información que se ha metido en el formulario.
- *Abrir cámara*, que abre la cámara para, a continuación, escanear el monumento en la actualidad.

Al acceder a la interfaz para añadir un monumento manualmente se muestra un formulario, que se puede observar en la Figura 4 (dcha.), que permite introducir:

- El nombre del usuario, que puede ser anónimo si se selecciona la opción.
- La imagen actual del monumento que se quiere añadir.
- La imagen antigua del monumento que se quiere añadir.
- Las coordenadas X, Y, Z del monumento que se quiere añadir.
- Las escalas X, Y, Z del monumento que se quiere añadir.
- La URL que contiene información del monumento.

A continuación, se encuentran 2 botones, uno de ellos al igual que en el caso automático nos permite *Restablecer campos*. El otro botón con nombre *Enviar*, nos permite enviar el nuevo monumento cuyos datos hemos proporcionado en el formulario a la base de datos.

Figura 4. (Izq.) Interfaz para Añadir monumento (Centro) Interfaz para Añadir un monumento automáticamente (Dcha.) Interfaz para Añadir un monumento manualmente

La interfaz de información de monumentos es común a toda la aplicación, pero dependiendo desde donde se acceda esta pantalla se muestra una información u otra. No obstante, la esencia es la misma en todos los casos, una interfaz con información y un botón para volver a la interfaz anterior. Como ejemplo de una de estas interfaces se muestra la Figura 5.

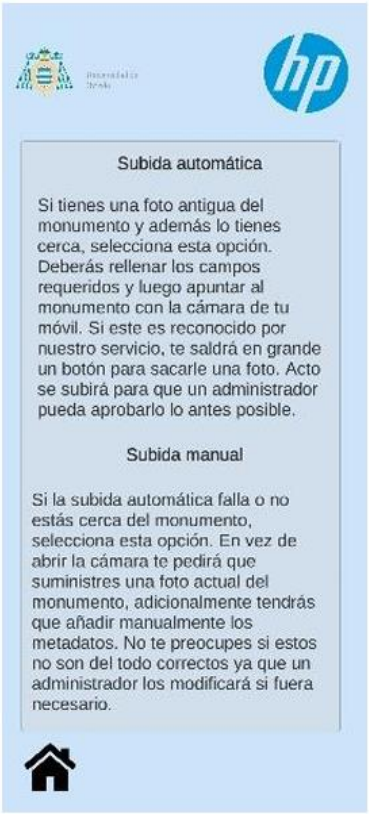

Figura 5. Ejemplo de interfaz de información

2.2. Requisitos funcionales

En el caso de este proyecto, los requisitos funcionales fueron proporcionados inicialmente por el equipo de HP SCDS, por lo tanto, no existió proceso de captura de requisitos. A partir de estos requisitos iniciales, se ha llevado a cabo un proceso de especificación mediante *historias de usuario* [5].

Las historias de usuario se han formulado siguiendo el formato: "Yo, como [rol de usuario], quiero [funcionalidad], de forma que [objetivo]". Además, cada historia se acompaña de sus correspondientes criterios de aceptación, los cuales definen las condiciones que deben cumplirse para considerar la funcionalidad como correctamente implementada y validada. A continuación, las Figuras 6, 7 y 8 muestran, a modo de ejemplo, tres historias de usuario pertenecientes a este trabajo. En estas tres figuras, figura en grande el nombre que recibe cada historia de usuario. A continuación, el estado de la historia de usuario, cuando fue creada y por quién. Por último, se describe la historia de usuario con el formato establecido y se especifican los criterios de aceptación necesarios para dar dicha historia de usuario por completada.

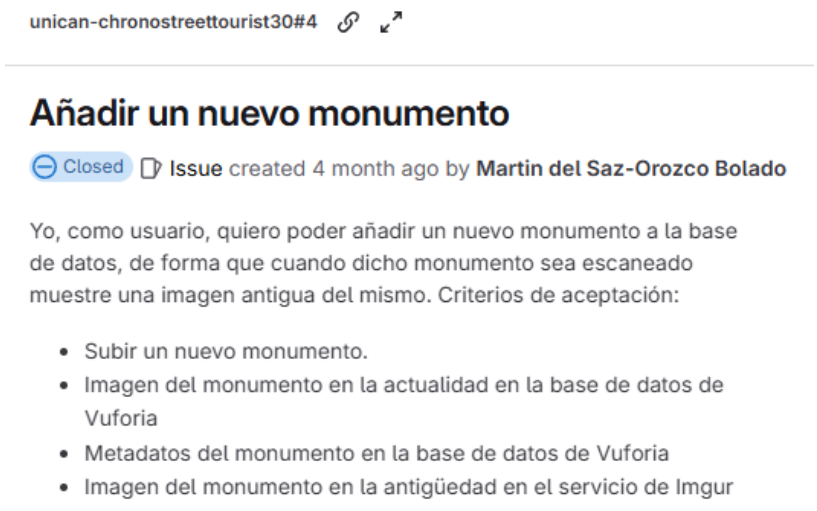

Figura 6. Historia de usuario Añadir un nuevo monumento

Figura 7. Historia de usuario Recibir notificaciones sobre monumentos cercanos

Figura 8. Historia de usuario Visualizar los monumentos en un mapa

Este tipo de especificación ha sido utilizado para todos los requisitos funcionales del sistema. En total en el proyecto se han especificado 6 historias de usuario.

2.3. Requisitos no funcionales

De cara a la especificación de los requisitos no funcionales de este proyecto se ha llevado a cabo un análisis del estándar de calidad ISO 25010 [6]. A partir de dicho análisis, se ha considerado necesario hacer hincapié sobre los requisitos de *comportamiento temporal* y *capacidad*.

A continuación, se especifica sobre cada requisito el motivo de destacar su importancia en este proyecto y la forma con la que se verificará que dichos requisitos se cumplen.

Comportamiento temporal

Uno de los aspectos más críticos en aplicaciones móviles que hacen uso de recursos en línea y tecnologías como la realidad aumentada es la rapidez con la que el sistema responde ante las acciones del usuario. En el caso concreto de esta aplicación, funcionalidades como la carga del mapa con la localización de los monumentos y la subida de nuevos elementos a las bases de datos (*Vuforia* e *Imgur*) pueden verse afectadas por tiempos de espera excesivos si no se gestionan adecuadamente los recursos y los flujos de red. Dado que este tipo de aplicaciones requieren mantener la atención del usuario y proporcionar una experiencia fluida e inmersiva, se ha considerado fundamental priorizar este requisito.

La verificación del cumplimiento de este requisito se llevará a cabo mediante pruebas de rendimiento sobre las funcionalidades críticas. Se establece como umbrales máximos aceptables para operaciones como la carga del mapa, un tiempo inferior a 3 segundos bajo condiciones normales de red y para la subida de monumentos, menos de 10 segundos incluyendo la transferencia de imágenes y metadatos. Estas pruebas permitirán evaluar de forma objetiva si el comportamiento temporal de la aplicación se encuentra dentro de los márgenes definidos como adecuados.

Capacidad

La capacidad del sistema se considera un requisito no funcional clave, entendida como la necesidad de optimizar el uso de recursos disponibles, especialmente teniendo en cuenta las limitaciones presupuestarias del desarrollo. Al tratarse de un proyecto sin financiación, no es viable asumir costes elevados derivados de un uso intensivo de servicios externos. Esto afecta particularmente a los servicios de *Google Maps* y *Vuforia*, cuyas APIs pueden implicar tarifas en función del número de peticiones. Por tanto, resulta esencial diseñar la aplicación con criterios de eficiencia, limitando el número de llamadas innecesarias y reutilizando datos en caché siempre que sea posible.

Para verificar el cumplimiento de este requisito se realizará una monitorización manual que permita contabilizar el número de peticiones realizadas a cada uno de los servicios externos, haciendo uso de las interfaces de analíticas que estas APIs ofrecen. En caso de encontrarse cerca de un límite que exceda las licencias gratuitas se deberá prescindir temporalmente de dicho servicio hasta que se consiga financiación o una nueva licencia gratuita.

2.4. Arquitectura

En este apartado se explicará el funcionamiento de los diferentes elementos implementados en este proyecto. De forma separada se detalla el funcionamiento de:

- El nuevo funcionamiento de la aplicación a la hora de añadir monumentos.
- El funcionamiento del nuevo mapa que muestra todos los monumentos que se encuentran en la base de datos.
- El funcionamiento del nuevo sistema de notificaciones de la aplicación al encontrarse el usuario cerca de un monumento para escanear.

Funcionamiento de añadir monumentos

La Figura 9 detalla cómo se realiza la incorporación de nuevos monumentos en la aplicación actualmente. El proceso comienza con la subida de la imagen del monumento antiguo desde el dispositivo al servicio de *Imgur* mediante la API correspondiente. Este servicio retorna un enlace con dicha imagen. Con este enlace junto a los datos de coordenadas, escala, posición y la imagen actual del monumento, se sube la imagen actual y sus metadatos a la base de datos de *Targets* de *Vuforia*. Esta operación conlleva la interacción con la API de *Vuforia Web Services*, a la cual se le realiza una petición HTTP de tipo POST con todos los datos mencionados previamente.

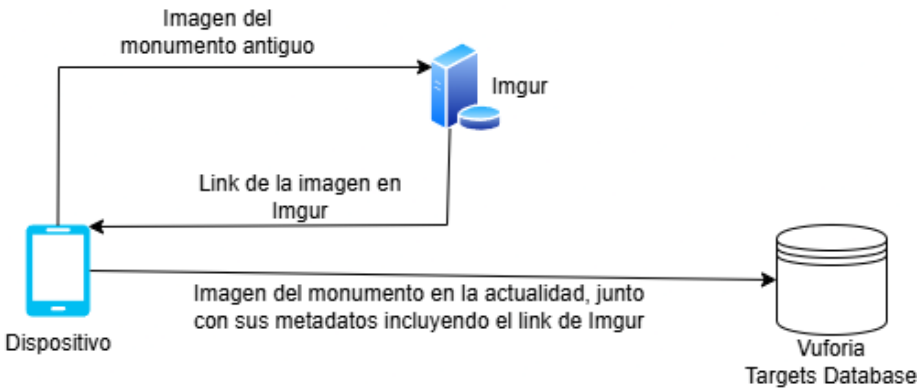

Figura 9. Diagrama de arquitectura para añadir monumentos

Funcionamiento del sistema de notificaciones

La Figura 10 especifica cómo funciona el sistema de notificaciones de la aplicación. Al iniciar la aplicación, el dispositivo realiza una petición HTTP GET a un *Worker* de *Cloudflare* pidiéndole una lista en formato JSON con todos los monumentos. La función de este *Worker* es la de hacer de intermediario entre la API de *Vuforia Web Services* y la página web que aloja el mapa de monumentos, con el objetivo de evitar las restricciones de peticiones sobre navegadores de tipo CORS. Las restricciones de tipo CORS (Cross-Origin Resource Sharing) son reglas de seguridad que impiden que una aplicación web en un dominio haga peticiones a recursos de otro dominio, a menos que el servidor de destino lo permita explícitamente mediante cabeceras específicas, lo que en este caso no es posible modificar al tratarse de la API de *Vuforia*. Se ha desplegado en *Cloudflare* aprovechando el hecho de que este ofrece este servicio de forma gratuita y con la posibilidad de monitorizar las analíticas de uso del mismo. Aprovechando que dicho *Worker* se desplegó para el mapa, se reutiliza en el sistema de notificaciones para simplificar la lógica de peticiones GET de la aplicación a solo una hacia el *Worker*.

El worker de Cloudflare realiza una petición GET a la base de datos de Vuforia con el objetivo de obtener todos los identificadores de los monumentos. A continuación, por cada monumento, se realiza una petición GET a la BBDD de Vuforia para obtener los metadatos de los monumentos, o targets, conforme a la terminología del proyecto. Esta forma de proceder, a pesar de ser ineficiente por las múltiples peticiones GET que se realizan a la API de *Vuforia Web Services*, es la única manera de obtener los metadatos de cada uno de los monumentos, o targets.

Con toda esta información, el *Worker* de *Cloudflare* elabora la respuesta para el dispositivo y envía la lista de monumentos en formato JSON. De esta lista de monumentos, la aplicación extrae el título de cada monumento, sus coordenadas y el código de la imagen dentro del sistema de alojamiento *Imgur*. A continuación, el dispositivo realiza una búsqueda en la lista que tiene almacenada con la información sobre cada monumento de la aplicación. En caso de encontrar uno o más que se encuentren a menos de 10km del dispositivo, éste genera una notificación con la información sobre el monumento que se encuentra cerca de él. La distancia originalmente se especificó como 100 metros, pero al estar la aplicación en una fase de implementación y para facilitar las pruebas de la misma, puesto que no se cuenta con un gran número de monumentos en ella, se decidió establecer en 10km. La revisión de la lista de monumento cercano se repite cada 5 minutos, generándose nuevas notificaciones en caso de que se encuentren nuevas coincidencias.

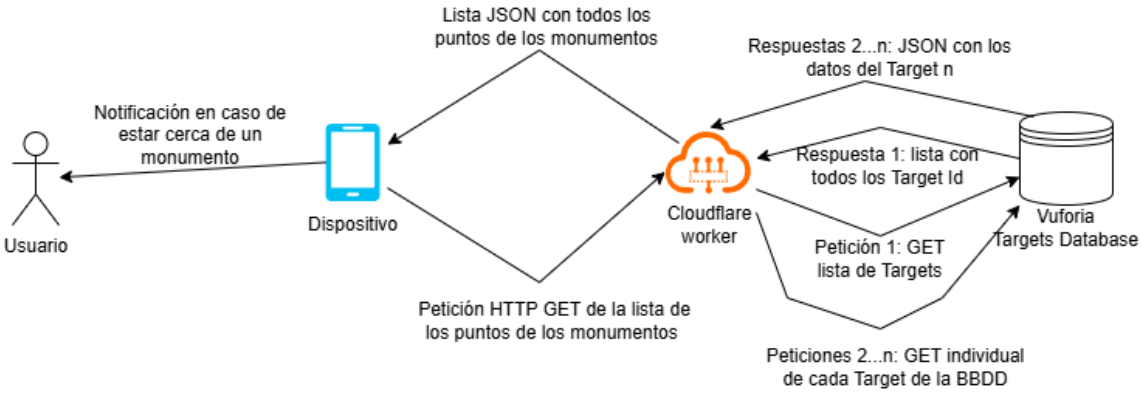

Figura 10. Diagrama de arquitectura del sistema de notificaciones sobre monumentos cercanos

Funcionamiento del mapa de monumentos

La Figura 11 ilustra cómo funciona el mapa de monumentos de la aplicación. Al solicitar el mapa de monumentos, el dispositivo se conecta a la web estática almacenada en una dirección específica de *Github Pages*. La elección de *Github Pages* como lugar para el despliegue fue condicionada por la facilidad de cambios que este ofrece, puesto que simplemente modificando los archivos de un repositorio se despliega la página web; además del hecho de que dicho servicio sea gratuito. Para mostrar la página web solicitada, se hace uso del plugin de *unity-webview*, el cual permite mostrar un documento HTML dentro de una aplicación de *Unity* [7]. Merece la pena recordar que las versiones previas de esta aplicación estaban desarrolladas sobre *Unity*, lo cual suponía una restricción a la hora de la implementación de este mapa y que condicionó de esa manera la forma en que está implementado.

La web almacenada en *Github Pages* realiza una petición HTTP GET al *Worker* de *Cloudflare* para obtener una lista en formato JSON con la información de los monumentos que necesita mostrar. El *Worker* de *Cloudfare*, como en el caso anterior, realiza una petición

GET a la base de datos de *Vuforia* con el objetivo de obtener todos los identificadores de los monumentos y luego realiza peticiones individuales para recuperar sus metadatos. Con la información devuelta por *Vuforia*, se recuperan las imágenes alojadas en *Imgur* y se retorna la web con monumentos.

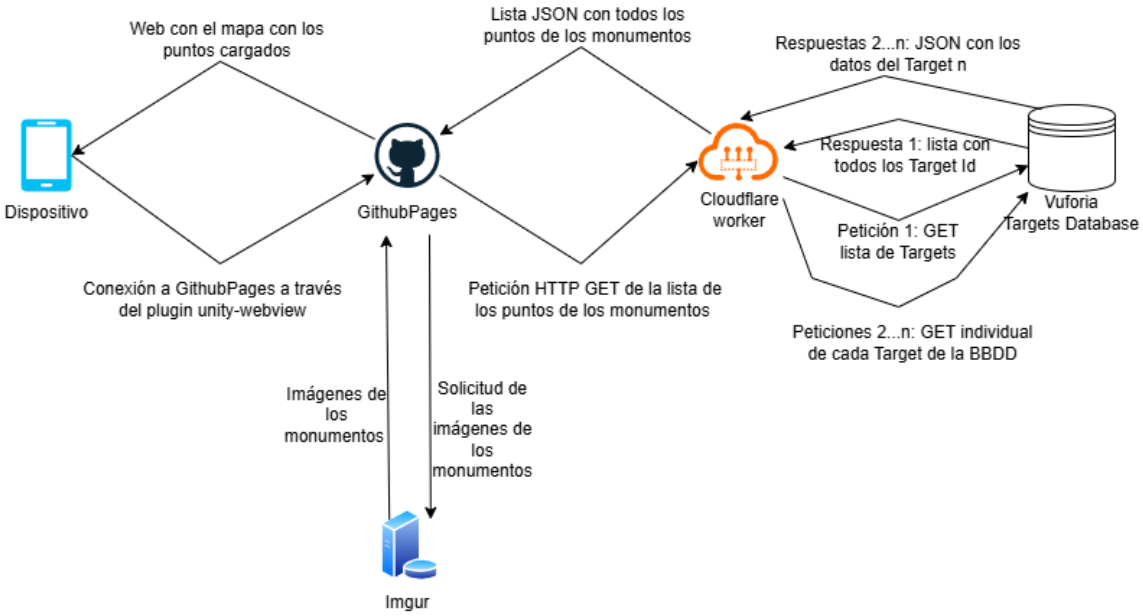

Figura11. Diagrama de arquitectura del mapa de monumentos

3. Implementación

Esta sección ilustra mediante pequeños ejemplos cómo se ha llevado a cabo la implementación de la aplicación. Más concretamente, se describen los flujos de ejecución para la subida de imágenes y la visualización del mapa de monumentos.

3.1. Subida de imágenes

3.1.1. Formularios

La subida de imágenes se trata de una funcionalidad ya implementada en versiones anteriores de la aplicación. En este proyecto se ha modificado ligeramente los formularios de subida de imágenes. Los nuevos formularios que se ilustran en las Figuras 12 (izq.) y (dcha.). Se ha sustituido el campo de *Usuario* por *Nombre del monumento*, se ha modificado la entrada de la ubicación de *X*, *Y* y *Z* a *Longitud* y *Latitud*. También se ha añadido una casilla de verificación en caso de que el usuario desee que los campos de *Longitud* y *Latitud* se autocompleten con su ubicación actual. Puesto que el proyecto está desarrollado en *Unity*, este formulario se implementa utilizando diferentes elementos dentro de una escena 2D de Unity. Cada elemento, incluyendo los campos de texto para introducir información y los textos descriptivos estáticos, tiene un identificador único el cual se puede referenciar desde los diferentes scripts de C# que se encargan de programar la lógica de la interfaz de usuario de la aplicación.

Figura 12. (Izq.) Interfaz para añadir monumento automáticamente actual (Dcha.) Interfaz para añadir monumento manualmente actual

El Listado 1 muestra, a modo de ejemplo, cómo se accede a los elementos de un formulario. Primero se definen las variables de tipo *GameObject* donde se tendrán las referencias a los elementos. Después se hace uso del método *initializeAll* el cual se encarga de obtener todas las referencias.

```
1. //Components' definition
2. GameObject label_image, label_current_image, ...;
3. GameObject xPos, yPos, zPos, xEs, yEs, zEs;
4. GameObject label_old_image_auto, url_field_auto, name_field_auto;
5. GameObject toggle_auto, toggle_manual;
6. /*Initialize game objects*/
7. void initializeAll()
8. {
9.          if (xPos is null) xPos = GameObject.Find("xPos_field");
10.         if (yPos is null) yPos = GameObject.Find("yPos_field");
11.         if (zPos is null) zPos = GameObject.Find("zPos_field");
12.         if (xEs is null) xEs = GameObject.Find("xEs_field");
13.         if (yEs is null) yEs = GameObject.Find("yEs_field");
14.         if (zEs is null) zEs = GameObject.Find("zEs_field");
15.         if (urlTexto is null) urlTexto = GameObject.Find("url_field");
16.         if (nameLabel is null) nameLabel = GameObject.Find("nombre_field");
20.         ...
21. }
```

Listado 1. Ejemplo de como se instancia y accede a los diferentes GameObject

El Listado 2 describe cómo se obtiene la ubicación automáticamente. Primero se ha de solicitar al usuario que active los permisos necesarios para que la aplicación pueda acceder al sistema de GPS del teléfono, como se muestra en las líneas 2 a 7. Dichas líneas se ejecutan en la función *Start* del script, la cual solo se activa una vez al iniciar el script cuando se accede a la interfaz que lo tiene asociado. Después se comprueba

periódicamente si el servicio de ubicación de Unity está inicializado o no y, en caso de no estarlo, se inicializa, tal como muestran las líneas 8 y 9 del Listado 2. Dichas líneas se ejecutan dentro de la función *Update* del script, la cual al tratarse de un script de *Unity* se ejecuta una vez por fotograma de la aplicación. Este concepto de fotograma viene dado por el hecho de ser un motor de aplicaciones más enfocado a videojuegos.

```
1. //Location permissions (only needed for Streetview)
2. if (!UnityEngine.Android.Permission.HasUserAuthorizedPermission
          (UnityEngine.Android.Permission.CoarseLocation)) {
3.         UnityEngine.Android.Permission.RequestUserPermission
                    (UnityEngine.Android.Permission.CoarseLocation);
4. }
5. if (!Permission.HasUserAuthorizedPermission(Permission.FineLocation)) {
6.         Permission.RequestUserPermission(Permission.FineLocation);
7. }
          (...)
8. if (autolocation && Input.location.status == LocationServiceStatus.Stopped) {
9.         Input.location.Start();
10.}
```

Listado 2. Ejemplo de como se solicita la ubicación al usuario y se verifica periódicamente

A continuación, como se muestra en el Listado 3, haciendo uso de una variable común a todo el script de tipo booleano, cuyo nombre es *autolocation*, y dos métodos encargados de comprobar el estado de la casilla de verificación con nombres *isAutoLocation* y *toggleChangedEvent*, cada vez que se marca o desmarca la casilla de verificación se invoca al método *toggleChangedEvent* ilustrado en las líneas 9 a 28. Este método se encarga de comprobar si se trata del caso automático o manual (líneas 11 y 24). Haciendo uso de *isAutoLocation* activa o desactiva los campos de texto. El método *isAutoLocation* se encarga de verificar si la casilla de verificación está activada, actualiza la variable global y retorna *true* en caso de estarlo y *false* en caso de no estarlo. Como ejemplo ilustrativo en las Figuras 13 (Izq.) y (Dcha.) se muestran ambos casos con la casilla activa y desactivada.

```
1.  Boolean isAutoLocation(Boolean b)
2.  {
3.        (... Cargado de la referencia a toggle)
4.       autolocation = toggle.isOn;
5.
6.       if (toggle.isOn) return true;
7.       return false;
8. }
9.     public void toggleChangedEvent(Boolean b)
10.     {
11.         if (b)
12.         {
13.             if (isAutoLocation(b))
14.             {
15.                 xPos.GetComponent<UnityEngine.UI.InputField>().interactable = false;
16.                 yPos.GetComponent<UnityEngine.UI.InputField>().interactable = false;
17.             }
18.             else {
19.                 xPos.GetComponent<UnityEngine.UI.InputField>().interactable = true;
20.                 yPos.GetComponent<UnityEngine.UI.InputField>().interactable = true;
21.             }
22.
23.         }
24.         else
25.         {
26.             (... Misma lógica que en el otro If pero para distintos GameObject)
27.         }
28.     }
```

Listado 3. Ejemplo de como se verifica el estado de la casilla de ubicación automática

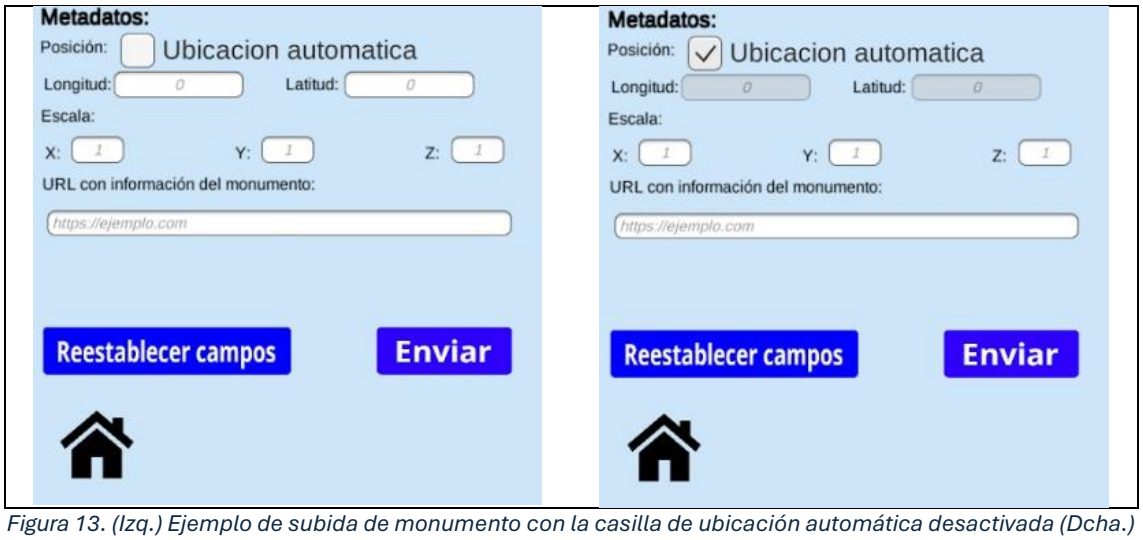

Figura 13. (Izq.) Ejemplo de subida de monumento con la casilla de ubicación automática desactivada (Dcha.) Ejemplo de subida de monumento con la casilla de ubicación automática activada

Por último, como se muestra en el Listado 4, a la hora de subir un monumento se comprueba si la ubicación automática está activa por medio de la variable *autolocation* (línea 3) y en caso de estarlo se hace uso de la ubicación actual del usuario para conformar el nombre del monumento el cual incluye las coordenadas actuales del usuario como ubicación del monumento, como se muestra en las líneas 5 a 10. En caso contrario, simplemente se coge la ubicación introducida a mano por el usuario, como se muestra en la línea 14.

```
 1. string name = "Error";
 2. if (autolocation) {
 3.       double longitud = 1.2f, latitud = 2.4f;
 4.       //Capturar la ubicacion actual
 5.       latitud = Input.location.lastData.latitude;
 6.       longitud = Input.location.lastData.longitude;
 7.       //Truncamos
 8.       latitud = Math.Truncate(latitud* 1_000_000) / 1_000_000;
 9.       longitud = Math.Truncate(longitud * 1_000_000) / 1_000_000;
10.       name = latitud.ToString().Replace(".", "_").Replace(",", "_") + "-" +
              longitud.ToString().Replace(".", "_").Replace(",", "_") + "-" +
              monumentName + "-" + idImgur;
11. }
12. else
13. {
14.       name = xPos.GetComponent<UnityEngine.UI.InputField>().text.
                  Replace(".","_").Replace(",", "_")
                  + "-" +
                  yPos.GetComponent<UnityEngine.UI.InputField>().text.
                  Replace(".", "_").Replace(",", "_")
                  + "-" + monumentName + "-" + idImgur;
15. }
```

Listado 4. Ejemplo de como se conforma el nombre del monumento y se utiliza la ubicación para ello

3.1.2. Comunicación con Imgur

La comunicación con *Imgur* se realiza en el momento en el que se acciona el botón de *Enviar* en el formulario de subida de monumentos. Dicha acción conlleva la ejecución de una serie de funciones que se implementan en el script en C# cuyo nombre es *FileManager.cs* y del cual se extrae el Listado 5 que se explica a continuación. Para la subida de imágenes a *Imgur*, primero se debe acceder a la ruta que proporciona el usuario de la imagen antigua del monumento (línea 5), cargar dicha imagen en una variable en formato binario y, haciendo uso del plugin *ImgurSharp* y una credencial de la API de *Imgur* se realiza

la subida de la imagen (líneas 4 a 6). A partir de la subida se obtienen varios datos, de los cuales solo son relevantes dos de ellos en nuestro caso:

- La dirección URL donde se aloja la imagen subida a *Imgur*. Esta dirección se utilizará para incorporarla en los metadatos que se subirán a la base de datos de Vuforia. Este valor se captura como se muestra en la línea 7 del Listado 5.
- El código identificativo de la imagen en Imgur. Dicho código se incorporará a la cadena de caracteres que da nombre al monumento, de manera que se pueda recuperar posteriormente en la implementación del mapa de monumentos de la aplicación. Este valor se captura como se muestra en la línea 8 del Listado 5.

```
1. async void UploadAll()
2. {
3.     //Upload the old image to Imgur and get the URL as an string
4.     Imgur imgur = new Imgur("CODIGO_DE_API_IMGUR");
5.     MemoryStream ms = new MemoryStream
           (File.ReadAllBytes(label_old_image.GetComponent<UnityEngine.UI.Text>().text));
6.     ImgurSharp.Image img = await
           imgur.UploadImageAnonymous(ms, "name", "title", "description");
7.     String oldImage = img.Link;
8.     String imageId = img.Id;
```

Listado 5. Ejemplo de subida de una imagen a Imgur

3.1.3. Incorporación a Vuforia

La incorporación de un nuevo monumento a *Vuforia* se realiza inmediatamente tras la comunicación con *Imgur*, en el mismo script con nombre *FileManager.cs*. De cara a la comunicación con Vuforia se realizan tres pasos principales:

- Primero se carga la imagen en una variable binaria, se convierte al formato de imagen JPG, tal y como se muestra en las líneas 3 a 10 del Listado 6. A continuación, se comprueba que dicha imagen no excede un tamaño de 2 Megabytes y en el caso de ser así se reduce el tamaño de esta a un valor inferior a 2 Megabytes. Para la comprobación del tamaño y compresión de la imagen se implementa un bucle que verifica que el tamaño del archivo no supere los 2MB y en caso de hacerlo se reduzca la calidad de dicho archivo un 20% por cada iteración del bucle, tal y como se muestra en las líneas 14 a 19 del Listado 6.
- Luego se genera un archivo con extensión *.meta* temporal el cual contiene los datos necesarios para que el monumento pueda ser escaneado posteriormente. El formato de dicho archivo se genera con el patrón que se adjunta en la Figura 14. La generación de dichos metadatos se lleva a cabo como se muestra en el Listado 7.
- Finalmente haciendo uso de un paquete NuGeT, que permite la comunicación con la base de datos de Vuforia, se suben los datos del monumento al servicio, tal y como se ejemplifica en las líneas 4 a 11 del Listado 8.


```
{
        "position":["0","0","0"],
        "url":"https://i.imgur.com/3AT0VIq.jpeg",
        "scale":["1","1","1"],
        "location":["0","0","0"]
}
```


Figura 14. Ejemplo de metadatos de una imagen

```
1. private string ConvertImageToJPG(string filePath) {
2.         (...)
3.         Texture2D texture = new Texture2D(2, 2);
4.         byte[] fileData = File.ReadAllBytes(filePath);
5.         texture.LoadImage(fileData);
7.         // Comprimir imagen como JPG
8.         byte[] jpgData = texture.EncodeToJPG(100);
9.         string tempPath = Path.Combine(Application.persistentDataPath, "temp.jpg");
10.        File.WriteAllBytes(tempPath, jpgData);
11.        // Reducir tamaño si es necesario
12.        FileInfo fileInfo = new FileInfo(tempPath);
13.        int quality = 80;
14.        while (fileInfo.Length > 2097152 && quality > 0) {
15.            jpgData = texture.EncodeToJPG(quality);
16.            File.WriteAllBytes(tempPath, jpgData);
17.            fileInfo.Refresh();
18.            quality -= 20;
19.        }
20.        Destroy(texture);
21.        return tempPath;
22.    }
```

Listado 6. Ejemplo de cargado, conversión y compresión de una imagen a JPG

```
1. void generateMetadata(String url) {
2.    metadatos="{\n\"position\":[\""+xPos.GetComponent<UnityEngine.UI.InputField>().text
                + "\",\"" + yPos.GetComponent<UnityEngine.UI.InputField>().text
                + "\",\"" + "0" + "\"],";
3.    metadatos += "\n\"url\":\"" + url + "\",";
4.    metadatos +="\n\"info\":\""+urlTexto.GetComponent<UnityEngine.UI.InputField>().text
                + "\",";
5.    metadatos += "\n\"scale\":[\"" + xEs.GetComponent<UnityEngine.UI.InputField>().text
                + "\",\"" + yEs.GetComponent<UnityEngine.UI.InputField>().text
                + "\",\"" + zEs.GetComponent<UnityEngine.UI.InputField>().text
                + "\"]\n}";
6. }
```

Listado 7. Ejemplo de generación de los metadatos de monumento

```
1.bool uploadVuforiaTarget(string targetName, float width, string image, string metadata){
2.         bool success = false;
3.         try {
4.             var userCredential = new UserCredential();
5.             var initializer = new BaseClientService.Initializer() {
6.                 ApplicationName = "VuforiaWebService",
7.                 HttpClientInitializer = userCredential,
8.             };
9.             var keys = new ServerAccessKeys(vuforiaAccessKey, vuforiaSecretKey);
10.            var targetService = new TargetService(initializer);
11.            var newTarget = targetService.TargetList.Insert(keys,
                   new PostTrackableRequest {
                             Name = targetName,
                             Width = width,
                             Image = image,
                             ActiveFlag = true,
                             ApplicationMetadata = metadata
                   }).Execute();
12.            success = true;
13.        } catch (Exception ex) {
13.            debugFile($"General Error: {ex.Message}");
14.        }
15.        return success;
16.    }
```

Listado 8. Ejemplo de subida de un monumento a la base de datos de Vuforia

3.2. Mapa de monumentos

3.2.1. Proxy del listado de monumentos con Cloudflare Worker

Tal como, se comentó anteriormente la necesidad de implementar un proxy para obtener el listado de monumentos y su información se debe a una limitación de seguridad presente en los navegadores web, conocida como CORS. Esta medida impide que una página web, como la desarrollada en este proyecto en GitHub Pages, pueda comunicarse directamente con servicios externos como la API de Vuforia.

Dado que la página necesita acceder a esta API para mostrar los monumentos en el mapa, fue necesario introducir un intermediario o proxy y para ello se utilizó *Cloudflare Workers*. Este proxy actúa como puente entre la página web y la API de Vuforia, reenviando las solicitudes de forma segura y evitando que el navegador las bloquee por la política de CORS, la cual es un mecanismo de seguridad que los navegadores utilizan con el objetivo de controlar solicitudes HTTP de un dominio a otro y que en este caso estaban bloqueando las peticiones desde nuestra página *Github Pages* a *Vuforia Web Services*.

Dada la necesidad de implementar este proxy, se ha aprovechado para delegar en el parte de la lógica de funcionamiento del mapa. De esta forma, en vez de simplemente evitar la política de CORS, el proxy realiza todo lo siguiente cada vez que recibe una petición GET:

- Recibe como parte de las cabeceras de la petición GET las credenciales con las que se accede a la base de datos de Vuforia con la API, tal y como se muestra la obtención de la pareja de credenciales en el Listado 9.
- El proxy comienza realizando una primera petición GET a la API de Vuforia con la cual obtiene una lista con todos los IDs de los monumentos que se encuentran almacenados en la base de datos, tal y como se describe en el Listado 10. Es importante remarcar la necesidad de generar cabeceras con autenticación tal y como se describe en dicho listado en las líneas 1 a 8.
- Cuando tiene la lista con todos los IDs de los monumentos de la base de datos realiza tantas peticiones GET como monumentos se encuentran en dicha base de datos. En cada petición obtiene todos los datos de las imágenes, de los cuales solo almacena el nombre de los monumentos en una lista. Dicho nombre contiene más información que simplemente el nombre, la cual se usará en GitHub Pages y por ello se explica en el siguiente apartado. Este proceso se realiza tal y como se puede observar en el Listado 11.
- Una vez ha completado la lista con todos los nombres de los monumentos, retorna dicha lista en formato JSON, como se muestra en el Listado 12.

```
1.      const headers = request.headers;
2.      const vwsUrl = "https://vws.vuforia.com/targets";
3.      const secretKey = headers.get("Secret");
4.      const accessKey = headers.get("Access");
```

Listado 9. Ejemplo de captura en variables de los tokens de Vuforia de la cabecera

```
1.    const httpVerb = "GET";
2.    const md5 = "d41d8cd98f00b204e9800998ecf8427e"; //al ser una peticion sin body
3.    const contentType = "";
4.    let date = new Date().toUTCString();
5.    let requestPath = "/targets";
6.    let stringToSign = httpVerb + "\n" + md5 + "\n" + contentType + "\n" +
                date + "\n" + requestPath;
7.    let signature = await generarHMAC(stringToSign, secretKey);
8.    let authorizationHeader = 'VWS ' + accessKey+ ':' + signature;
9.    (...) Configuracion del resto de cabeceras de la petición
10.   const response = await fetch(vwsUrl, requestOptions);
```

Listado 10. Ejemplo de petición GET de la lista de IDs de Vuforia

```
1.    let responseData = await response.json();
2.    const results = responseData.results;
3.    console.log(results.length);
4.    let targets = []
5.    let counter = 0;
6.    for(let i = 0; i < results.length; i++){
7.      let targetid = results[i];
8.      requestPath = "/targets/" + targetid;
9.      (...) Firmado y generación de las cabeceras y petición GET
10.     const res = await fetch(vwsUrl + "/" + targetid, requestOptions);
11.     const target = await res.json();
12.     targets[counter] = target.target_record.name;
13.     counter++;
14.   };
```

Listado 11. Ejemplo de las peticiones GET de cada target a la API de Vuforia

```
1. try {
2.       return new Response(JSON.stringify({
3.         targets
4.       }, null, 2), {
5.         headers: { "Content-Type": "application/json",
6.        "Access-Control-Allow-Origin": "*", // Permite solicitudes desde cualquier origen
7.         "Access-Control-Allow-Methods": "GET, POST, OPTIONS", // Métodos permitidos
8.         "Access-Control-Allow-Headers": "Content-Type, Authorization, Secret, Access"
9.         }
10.      });
11.   } catch (error) {
12.     (...) Gestión de errores
13.   }
```

Listado 12. Ejemplo del retorno de la lista con los nombres de los monumentos

3.2.2. Página web estática del mapa en GitHub Pages

La página web estática que da soporte al mapa de monumentos de la aplicación se encuentra en GitHub Pages, donde solo se pueden desplegar páginas web junto con archivos de estilo y javascript, lo cual limita lo que se puede implementar en dicha página. En este caso, se aloja un documento HTML muy sencillo, el cual ejecuta un código JavaScript el cual se encarga de toda la lógica del mapa. Dicho script se compone de 4 funciones principales:

- **cargarMapa()**: Función encargada de insertar en la página todo lo relativo al mapa junto con la API de Google Maps, encargándose de llamar a *iniciarMapa*, tal y como se hace en las líneas 4 y 7 del Listado 13.

```
1. function cargarMapa() {
2.  (...) Gestión de errores
3.     const script = document.createElement("script");
4.     script.src =
         `https://maps.googleapis.com/maps/api/js?key=${api_key}&callback=iniciarMapa
         &libraries=marker`;
5.     script.async = true;
6.     script.defer = true;
7.     document.body.appendChild(script);
8. }
```

Listado 13. Ejemplo de la función para el cargado de la API de Google Maps

- **iniciarMapa()**: Función que inicializa el mapa (líneas 2 a 7 del Listado 14), llama a la función encargada de cargar todos los puntos de interés del mapa, llamando a la función *cargarPuntosVuforia*, en la línea 8, e implementa el botón que se encarga de centrar el mapa en la ubicación actual del dispositivo del usuario (líneas 10 a 25 del Listado 14).

```
1. function iniciarMapa() {
2.     const coordenadas = { lat: 43.4751, lng: -3.8078 };
3.     mapa = new google.maps.Map(document.getElementById("map"), {
4.       zoom: 12,
5.       center: coordenadas,
6.       mapId: "DEMO_MAP_ID"
7.     });
8.     cargarPuntosVuforia();
9.     // Botón para buscar ubicación actual
10.    const locationButton = document.createElement("button");
11.    locationButton.textContent = "Mi ubicación";
12.    locationButton.classList.add("btn-location");
13.    mapa.controls[google.maps.ControlPosition.TOP_CENTER].push(locationButton);
14.
15.    locationButton.addEventListener("click", () => {
16.      if (navigator.geolocation) {
17.        navigator.geolocation.getCurrentPosition((position) => {
18.          const userLocation = {lat: position.coords.latitude,
                   lng:position.coords.longitude,};
19.          mapa.setCenter(userLocation);
20.        },
21.        (error) => { switch (error.code) { //Casos de error }
22.        }, { enableHighAccuracy: true, timeout: 10000, maximumAge: 0 });
23.        }
24.      else { alert("Geolocalización no soportada en este navegador.");}
25.    });
25. }
```

Listado 14. Ejemplo de la función encargada de iniciar el mapa y el botón de ubicación actual

- **cargarPuntosVuforia()**: Función que realiza la petición al proxy de *Cloudflare* para traer la lista JSON con todos los monumentos, como se muestra en la línea 4 del Listado 15. La lista obtenida es posteriormente procesada, como se muestra en las líneas 7 a 32, convirtiendo una cadena de caracteres en las propiedades de un punto de interés del mapa. Dichas cadenas de caracteres tienen el siguiente formato: LATITUD-LONGITUD-NOMBRE-CODIGOIMGUR, donde se utilizan los guiones como separadores y los guiones bajos como puntos de separación para las coordenadas. Además en caso de ser la longitud o latitud negativas se colocarán dos guiones seguidos en la cadena. Una vez se ha dividido la cadena en las propiedades de un punto de interés del mapa, se llama a la función *agregarPuntoDeInteres*.

```
1. async function cargarPuntosVuforia() {
2.     let listaPuntos = [];
3.     (...) Generacion de las cabeceras para la petición GET al Worker de Cloudflare
4.     fetch("https://vuforia-chrono.msob523.workers.dev/", requestOptions)
5.         .then((response) => response.json()).then((result) => {
6.             listaPuntos = result.targets;
7.             for (let i = 0; i < listaPuntos.length; i++) {
8.                 let punto = listaPuntos[i].replaceAll("_", ".");
9.                 let longitud = "", latitud = "", nombre = "", imagen = "", parte = 0;
10.                for(let j = 0; j < punto.length; j++) {
11.                    if (parte == 0) {
12.                        if (j > 0 && punto[j] == "-") {
13.                            parte++;
14.                        } else {
15.                            longitud += punto[j];
16.                        }
17.                    } else if (parte == 1) {
18.                        if (punto[j] == "-" && punto[j-1] != "-") {
19.                            parte++;
20.                        } else {
21.                            latitud += punto[j];
22.                        }
23.                    } else if (parte == 2) {
24.                        if (punto[j] == "-") {
25.                            parte++
26.                        } else {
27.                            nombre += punto[j];
28.                        }
29.                    } else if (parte == 3) {
30.                        imagen += punto[j];
31.                    }
32.                }
33.                agregarPuntoDeInteres(mapa, parseFloat(longitud), parseFloat(latitud),
                            nombre, imagen);
34.            }
35.            return listaPuntos;
36.        })
37.        .catch((error) => {
38.            console.error(error);
39.        });
40. }
```

Listado 15. Ejemplo de la función encargada de hacer la petición GET al Worker de Cloudlfare y procesarla

- **agregarPuntoDeInteres()**: Función encargada de añadir un punto de interés al mapa. En ella se genera un nuevo marcador con los parámetros recibidos, como se puede observar en las líneas 2 a 6 del Listado 16. A continuación, sobre dicho marcador se genera un pequeño desplegable el cual al ser pulsado mostrará la información de dicho punto de interés, tal y como se observa en las líneas 8 a 26, donde se genera un pequeño HTML con el título del monumento, su imagen en *Imgur* y las coordenadas.

```
1. function agregarPuntoDeInteres(mapa, lat, lng, titulo, imagen) {
2.     const marcador = new google.maps.marker.AdvancedMarkerElement({
3.         position: { lat, lng },
4.         map: mapa,
5.         title: titulo,
6.     });
8.     marcador.addListener("gmp-click", () => {
9.         if (!infoWindow) {
10.            infoWindow = new google.maps.InfoWindow();
11.        }
13.        const imgurID = imagen.replace(/\.(jpg|jpeg|png|gif)$/i, '');
15.        const embedHtml = `
16.            <div style="text-align: center;">
17.                <h3>${titulo}</h3>
18.               <blockquote class="imgur-embed-pub" lang="en" data-id="${imgurID}" data-
                           context="false">
19.                    <a href="https://imgur.com/${imgurID}">Ver en Imgur</a>
20.                </blockquote>
21.                <p>Coordenadas: ${lat.toFixed(6)}, ${lng.toFixed(6)}</p>
22.            </div>
23.        `;
25.        infoWindow.setContent(embedHtml);
26.        infoWindow.open(mapa, marcador);
27.       (...) Cargado de la librería de Imgur para mostrar la imagen del monumento
39.    });
40. }
```

Listado 16. Ejemplo de la generación de un nuevo punto de interés en el mapa

3.2.3. Incorporación de la web en la aplicación Unity

El motor de desarrollo de Unity por defecto no permite mostrar páginas web HTML dentro sus aplicaciones. Por ello, para poder cargar la web implementada anteriormente en *GitHub Pages*, se hace uso de un plugin para Unity cuyo nombre es *unity-webview*. Para hacer uso de dicho plugin se añade un nuevo elemento en la escena donde se quiera visualizar el mapa y a dicho elemento se le liga un script el cual inicializa (líneas 4 a 7 del Listado 17) y hace funcionar la web que se carga. En dicho script se inicializa el mapa, ajustando el tamaño del elemento en pantalla y cargando la dirección URL de la web del mapa (línea 10 del Listado 17). Adicionalmente se han tenido que implementar dos métodos auxiliares para ocultar y mostrar el mapa al cambiar de escena.

```
1. void Start() {
2.       (...)
3.         // Crear instancia de WebViewObject
4.         webViewObject = (new GameObject("WebViewObject")).AddComponent<WebViewObject>();
5.         webViewObject.Init(
6.             (...) Gestión de errores
7.         );
8.
9.         // Cargar la URL
10.        webViewObject.LoadURL("https://martin-dso-b.github.io/?some_key=" + apiKey +
                   "&accessKey="+ accessKey    + "&secretKey=" + secretKey );
11.        // Hacer visible el WebView
12.        webViewObject.SetVisibility(true);
13.        // Ajustar tamaño y posición según el RectTransform del Canvas
14.        AdjustWebViewSize();
15.    }
```

Listado 17. Ejemplo de la inicialización de la web del mapa en Unity

4. Pruebas y Despliegue

4.1. Pruebas

Debido a que la aplicación ha sido desarrollada en *Unity*, a lo largo del proyecto la fase de pruebas se limitó únicamente a pruebas manuales. Esto se debe a que Unity, al funcionar combinando escenas puesto que es un motor de desarrollo orientado principalmente a videojuegos y scripts los cuales se ejecutan desde las escenas de manera asíncrona no facilita la implementación de pruebas automatizadas. En este contexto, el comportamiento del sistema depende en gran medida de la interacción visual y de eventos en tiempo de ejecución, lo que dificulta la realización de pruebas unitarias tradicionales sobre los scripts implementados.

Por esta razón, las pruebas llevadas a cabo se han centrado en pruebas de integración y de interfaz, todas ellas ejecutadas de forma manual. Estas pruebas han permitido comprobar el correcto funcionamiento conjunto de los distintos componentes del sistema y validar la experiencia de usuario. Asimismo, se han enfocado en la verificación de los criterios de aceptación definidos en las historias de usuario, actuando como base para la validación funcional del producto.

4.1.1. Desarrollo de las pruebas

Como bien se ha comentado anteriormente, en el proyecto se han realizado únicamente pruebas manuales las cuales abarcaban pruebas de interfaz e integración a la vez. De cara a seguir un proceso de pruebas riguroso, se han especificado los casos de prueba a ejecutar manualmente siguiendo los criterios de aceptación de las distintas historias de usuario. Para cada historia de usuario se ha creado un listado de pruebas y a continuación se ha verificado que la funcionalidad que implementa dicha historia de usuario verdaderamente cumple los distintos criterios de aceptación.

A continuación, se muestra a modo de ejemplo en la Tabla 1 el proceso de pruebas realizado para la historia de usuario relativa al *Mapa de monumentos* la cual se especifica en la Figura 8.

Especificación de la prueba	Resultado
Verificación visual mediante la cual se observa que todos los monumentos que se encuentran en la base de datos de vuforia se localizan en el mapa	ÉXITO
Verificación visual y manual mediante la cual se observa que si se desplaza el mapa a una ubicación distinta a la mía y se pulsa el botón "Mi ubicación" efectivamente el mapa se centra en mi ubicación actual	ÉXITO
Verificación visual y manual mediante la cual se observa que al pulsar sobre un marcador de un monumento se abre una pequeña ventana donde se pueden visualizar el nombre, las coordenadas y una foto de dicho monumento.	ÉXITO

Tabla 1. Tabla de prueba para la historia de usuario del Mapa de monumentos

De la misma manera que en el ejemplo anterior se ha verificado la correcta implementación del resto de historias de usuario desarrolladas en este proyecto. Este proceso a pesar de no ser automatizado ha permitido evitar y corregir posibles fallos de implementación de las nuevas funcionalidades de la aplicación.

4.1.2. Pruebas de comportamiento temporal

De cara a la verificación del cumplimiento del requisito no funcional de comportamiento temporal, se ha realizado un proceso de pruebas manual. Para ello, tal y como se muestra en la Tabla 2, donde se establece una prueba a realizar, un tiempo máximo que se permite para dicha prueba, un tiempo promedio medido en un total de 10 intentos de realización de la prueba, y la desviación típica muestral; cuyos valores se muestran redondeados a los 2 últimos decimales.

Como podemos observar, el resultado obtenido en estas pruebas es satisfactorio, puesto que en ambos casos el tiempo promedio es inferior al tiempo máximo esperado y la desviación típica obtenida nos indica que los datos obtenidos son suficientemente precisos.

Prueba	Tiempo máximo esperado	Tiempo promedio medido	Desviación típica
Acceder a la interfaz del *Mapa de monumentos*	3 segundos	2.31 segundos	0,32 segundos
Realizar la subida de un monumento de forma manual	10 segundos	6.78 segundos	2.34 segundos

Tabla 2. Tabla relativa a las pruebas de comportamiento temporal

La forma de medición de estos tiempos obtenidos en dichas pruebas ha sido mediante la utilización de las funciones de depuración que ofrece Unity, generando dos entradas en el registro de depuración: una al iniciar la prueba y una al finalizar la prueba. Posteriormente, de manera manual se ha restado la marca de tiempo inicial a la final para obtener el tiempo de ejecución de la prueba.

4.1.3. Pruebas de aceptación y Product reviews

Puesto que en este proyecto se ha seguido una metodología ágil tipo *Scrum*, al final de cada sprint se llevaba a cabo una reunión de tipo *Product review*, en la cual se llevaban a cabo las pruebas de aceptación preestablecidas antes del sprint. Gracias a las pruebas de aceptación, se detectó que algunos criterios de aceptación no podían cumplirse, por lo que se llevaron a cabo modificaciones en dichos criterios de aceptación.

Un ejemplo es la historia de usuario con nombre *Recibir notificaciones sobre monumentos cercanos*. Dicha historia de usuario, en un principio, entre sus criterios de aceptación incluía uno el cual requería que la funcionalidad operase con el dispositivo bloqueado, en segundo plano. Este criterio de aceptación fue eliminado, puesto que la implementación de dicha funcionalidad no está soportada en *Unity*, al requerir de un segundo servicio Android encargado de llevarla a cabo. Al estar en un proyecto de una duración corta y con recursos limitados se decidió eliminar dicho criterio de aceptación por la alta complejidad que suponía cumplirlo.

4.2. Despliegue

De cara al correcto funcionamiento de la aplicación, esta requiere el despliegue o creación de una serie de servicios. A continuación, se describe cómo se han configurado estos servicios.

4.2.1. Vuforia Target Database

De cara a la obtención de una base de datos para almacenar las imágenes de los monumentos y sus metadatos se ha necesitado crear una cuenta en el portal de desarrolladores de *Vuforia*. Una vez registrado en dicho portal se solicita una licencia gratuita de tipo *Basic Plan*, la cual otorga acceso a la creación de una base de datos en *Cloud* dentro del *Target Manager*. Una vez se crea dicha base de datos, se interactúa con ella de dos formas posibles.

La primera forma de interacción con la base de datos es a través de la propia web de *Vuforia developers*, donde previamente se ha registrado el usuario. Aquí se pueden modificar las imágenes y sus metadatos almacenados, denominados *Targets*, de una forma sencilla. También se obtiene acceso a los dos tipos de credenciales de la base de datos, los cuales permiten interactuar con ella a través de la API de *Vuforia web services*.

La segunda forma de interacción con la base de datos es haciendo uso de las ya mencionadas credenciales para la interacción con la API de *Vuforia web services*. Esta forma de interacción es la que se utiliza en la aplicación desarrollada en este proyecto, haciendo uso de los dos tipos de credenciales que existen, las credenciales de *Cliente* y las credenciales de *Servidor*. Las credenciales de *Cliente* se utilizan en nuestra aplicación para el reconocimiento de monumentos, puesto que solo dan acceso a este tipo de operaciones de la API. Por otro lado, las credenciales de *Servidor* se utilizan en nuestra aplicación para la incorporación de nuevos monumentos a la base de datos y para la obtención de todos los datos necesarios para la implementación del mapa de monumentos.

4.2.2. Imgur

De cara al almacenamiento en la nube de las imágenes antiguas de los monumentos, con la finalidad de obtener un enlace directo a dichas imágenes para incluirlo en los metadatos de los *Targets* de la base de datos de *Vuforia*, se ha creado una cuenta gratuita en *Imgur*. Con esta cuenta gratuita se permite la creación de una *Application* en *Imgur*, la cual proporciona un identificador de cliente y una clave de acceso secreta. Con estos dos elementos, se puede hacer uso de la API de *Imgur* para subir imágenes y obtener el enlace directo a ellas. La interacción con *Imgur* solo se ha hecho mediante llamadas a la API que proporciona.

A raíz del desarrollo de la funcionalidad de *Añadir monumento*, se han generado una gran cantidad de imágenes en la API puesto que, en el proceso de pruebas manual, siempre era necesario adjuntar una imagen y si fallaba la subida a la base de datos de *Vuforia*, el último paso en el proceso, se generaban imágenes en *Imgur* a las cuales nunca se ha acabado dando uso. En el futuro este aspecto cabría mejorarlo, puesto que a pesar de ser un servicio gratuito no es conveniente sobrecargarlo con imágenes a las cuales nunca se va a dar ningún uso.

4.2.3. CloudFlare Worker

Debido a la necesidad de disponer de intermediario entre la API de *Vuforia Web Services* y la página web que da soporte al mapa en *GitHub Pages*, se optó por implementar esa necesidad en la nube con un *CloudFlare Worker*. Para ello fue necesario la creación de una

cuenta en el *CloudFlare Dashboard*, con la cual se implementó un *worker* que no es más que un proceso que se ejecuta en la nube.

El despliegue de este *worker* consistió en la creación de la aplicación y la codificación de la funcionalidad que este implementa, en este caso en *JavaScript*. En el proceso de implementación de dicho *worker* se realizaban pruebas sobre él realizando peticiones HTTP de tipo GET haciendo uso de la herramienta *Postman*. Y posteriormente una vez fue verificada que la funcionalidad estaba implementada correctamente, se realizaban dichas peticiones desde el *JavaScript* de la página que da soporte al mapa.

4.2.4. Github Pages

Con el objetivo de dar soporte al mapa de monumentos, se optó por desplegar una página *HTML* junto con un sencillo script *JavaScript* en el servicio gratuito que otorga para esto *GitHub Pages* a cada usuario de *GitHub*.

El despliegue de esta página web consiste en la creación de un repositorio con nombre *username.github.io*, en el cual basta con subir un archivo *index.html* y adicionalmente, archivos de tipo *JavaScript* o *CSS* con el objetivo de enriquecer un poco más el funcionamiento de esta página. Una vez subidos dichos archivos, es la propia *GitHub Pages* la encargada de desplegar en la dirección *URL* con formato *https://username.github.io*, la página web.

4.2.5. Google Cloud

Para implementar un mapa en la aplicación se optó por utilizar la API de *Google Maps*. Para tener acceso a dicha API es necesario tener creada una cuenta de *Google Cloud* y dentro de esta misma generar un *token* de acceso a la API de *Google Maps*.

En este proyecto se hace uso de una licencia gratuita la cual proporciona un número de usos del token limitado a un total de 10.000 usos.

4.2.6. Publicación de la aplicación

Como se ha comentado hasta ahora, para el funcionamiento de esta aplicación y todas sus características, son necesarios varios servicios diferentes. Algunos servicios, como el de *GitHub Pages, Imgur* o *CloudFlare Worker*, son gratuitos en su totalidad, pero solo en el caso de que su utilización no sea con fines comerciales. Y en otros casos, como es el de *Vuforia Targets Database* o *Google Cloud*, las licencias utilizadas en este proyecto han sido todas limitadas a desarrollo de aplicaciones y no a su despliegue comercial. Estos factores han supuesto que la aplicación actualmente no se encuentre publicada en ninguna tienda de aplicaciones para *Android*, como pueden ser la *Play Store*, *Huawei AppGallery* o *Samsung Galaxy Store*.

5. Conclusiones y Trabajos futuros

5.1. Experiencia personal

El afrontar el desarrollo de un proyecto, como es el de la actualización e implementación de nuevas funcionalidades de una aplicación como ChronoStreetTurist, ha sido un reto personal y de aprendizaje en distintos aspectos.

Por una parte, este proyecto ha servido para poner en valor los diferentes conocimientos obtenidos a lo largo de los años cursando el Grado en Ingeniería Informática, obligándome a utilizar una gran parte de estos fundamentos y conocimientos, así como refrescar aquellos que estaban más olvidados o peor comprendidos. En este proyecto he podido experimentar de primera mano el proceso de un desarrollo de software, en casi todas sus etapas. Por tanto, ha sido una experiencia educativa extraordinaria como preparación para el futuro laboral próximo.

Concretamente el desarrollo de este proyecto ha servido para comprender a fondo los siguientes conocimientos nuevos:

- La necesidad de interactuar con varios servicios web y sus APIs de tipo REST, han propiciado una mejor comprensión de cómo funcionan estos, ocasionando en mí una gran mejoría en la interacción con este tipo de servicios y aprendiendo a explorar el potencial del despliegue de servicios y del uso de herramientas como *Postman* para interactuar con ellos de una forma sencilla.
- Los diferentes requisitos planteados en forma de historias de usuario han conseguido enseñarme la capacidad de búsqueda de alternativas y diferentes soluciones a los problemas planteados. Ejemplo claro de esto han sido las interminables horas de trabajo invertidas en la sincronización de unos servicios con otros, como sincronizar *Vuforia Web Services* e *Imgur*, para subir imágenes nuevas a la base de datos, o encontrarme con restricciones de tipo *CORS* a la hora de la implementación del mapa de monumentos, propiciando así en el desarrollo de un intermediario.
- El tamaño de los archivos fuente que generan el desarrollo de aplicaciones en *Unity* ha supuesto un reto en el manejo del repositorio *GitLab* donde se alojaban dichos archivos. Este reto ha conseguido que aprenda a manejarme en repositorios *Git* con un gran volumen de archivos sin problema alguno y pudiendo solventar los conflictos que este tamaño de archivos origina.

5.2. Desarrollo futuro del proyecto

En caso de haber contado con un mayor margen de tiempo para el desarrollo del proyecto, se habrían abordado una serie de mejoras y funcionalidades adicionales que, por cuestiones de planificación y alcance, no han podido implementarse en esta versión. Por ello, a continuación, se proponen como posibles líneas de evolución y ampliación futura de la aplicación:

- Una posible mejora por implementar hubiera sido el soportar la generación de la aplicación para el sistema operativo *iOS* adicionalmente al sistema operativo *Android*. Esta mejora es alcanzable puesto que *Unity* soporta el desarrollo de aplicaciones en *iOS*, pero no es exactamente el mismo proceso de desarrollo que para *Android*, puesto que entre los distintos sistemas operativos existe una diferencia en la gestión de los permisos que obtiene la aplicación, así como la optimización a los distintos sistemas operativos.
- Otra posible mejora hubiera sido el servicio que dé soporte al sistema de notificaciones de la aplicación cuando dicha aplicación se encuentra en segundo plano. Este servicio constituye un gran reto que en este proyecto no se afrontó por

una cuestión de tiempo pero que es posible implementar con el tiempo y las herramientas necesarias.

- Tambien se propone una mejora en la funcionalidad de escaneo de monumentos pudiendo añadir en dicha pantalla, además del monumento superpuesto con realidad aumentada, diferentes desplegables u opciones para obtener más información sobre el monumento que se está visualizando.
- Por último, se propone aumentar las funcionalidades del mapa de monumentos, con la posibilidad de generar rutas de monumentos guiadas, que indiquen las direcciones a seguir para escanear diferentes monumentos en lugares cercanos o con relacion directa entre sí.

Todas estas funcionalidades son algunas de las posibles líneas de desarrollo a futuro de este proyecto, el cual ya ha superado una fase inicial de desarrollo y dos fases de evolución.

Referencias

[1] Dawid Borycki. "Programming for Mixed Reality with Windows 10, Unity, Vuforia, and UrhoSharp". Microsoft Press, Agosto 2018.

[2] Ken Schwaber and Jeff Sutherland. “The Scrum Guide”. Scrum.org. Noviembre 2020.

[3] David J. Anderson. "Kanban". Blue Hole Press. Abril 2010.

[4] Scott Chacon, Ben Straub "Pro Git". 2ª Edición. Apress. Noviembre 2014.

[5] Mike Cohn. "User Stories Applied". Addison-Wesley Professional. Marzo 2004.

[6] Joint Technical Committee ISO/IEC JTC 1,"Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model". International Standard Organization (ISO) e International Electrotechnical Commission (IEC), ISO/IEC 25010:2023, Noviembre 2023.

[7] Joseph Hocking . "Unity in Action". Manning, Marzo 2022.