Contents

1 Research **2**
1.1 Related Projects . . . 2

2 Design and Implementation **3**
2.1 Design . . . 3
2.1.1 System Architecture Diagram . . . 3
2.1.2 Site Maps . . . 4
2.1.3 Application Structure . . . 7
2.1.4 Design Patterns . . . 8
2.2 Implementation . . . 9
2.2.1 Development Tools . . . 9
2.2.2 Front-End Technologies . . . 10
2.2.3 Back-End Technologies . . . 11
2.2.4 Implementation of Key Functionality . . . 11
2.2.5 VR Tagging System . . . 11
2.2.6 VR Viewer . . . 13
2.2.7 Data Storage . . . 14

Appendices **15**

A Deployment Manual **16**
A.1 Deploying the VR Tagging Engine . . . 16
A.2 Deploying the VR Viewer . . . 16

B System Manual **18**
B.1 Working on the VR Engine . . . 18
B.1.1 API Endpoint Map . . . 18
B.2 Working on the VR Viewer . . . 19

C Usecase Diagram **20**

Chapter 1

Research

1.1 Related Projects

After gathering requirements from our client we decided to try and find applications with a similar design or feature set, to decide on a simple way to build the overall systems architecture and refine the initial requirement that we got from the client. However, there did not seem to be much on the market that matched the clients full description i.e. a system at allows a normal user to tag virtual reality videos, with text, html, audio . . . , and then allows them to play those tagged videos back on a mobile device such as the Samsung Gear VR.

The closest we could find to an app that fulfilled the clients specification was an application called "ThingLink" at `http://demo.thinglink.com/vr-editor` which did most of the tasks that the client wanted performed but the ability to edit display 360 video comes at a fee of 125 per user per month which is prohibitively expensive for charity work, as well as that there is no guaranty that the off-the-shelf software can be updated to fit all of the clients needs as requirements change. But ThingLink did give us the idea to create a Client/Server application that would export custom data files that any client could use. And in the tradition of other android file formats we decided the best thing to do would be to use a renamed zip file with a metadata file inside it. Our system however, would improve on the software by using as many Open Source components as possible in an effort to reduce our clients maintenance troubles as well as fulfil one of the ethical requirements that they had given us.

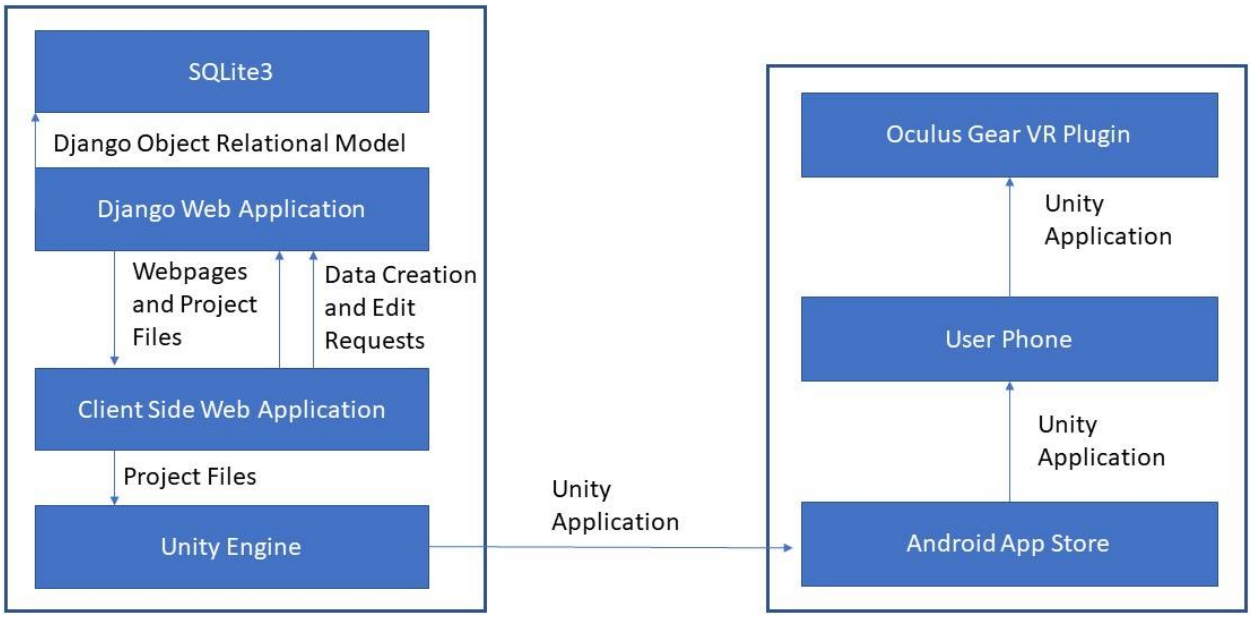
With that in mind we looked at the VR Player FREE `https://play.google.com/store/apps/details?id=com.vimersiv.vrplayerfree` for inspiration on how to build an interface and relayed my findings to my partner Jasper, informing him about the limitations of the Gear VR's input system and that the interface might need to be made simpler and more streamlined so as to better show up on the Gear VR. VR Player FREE was also one of the applications that was brought up in meetings as a possible related project as it did support 360 VR video that the client wanted but it did not support any tagging features and didn't provide an editor or any equivalent capability, ruling it out as it did not meet the clients requirements. Our system however will be different to VR Player FREE as it will have the ability to display tagged files.

Chapter 2

Design and Implementation

2.1 Design

2.1.1 System Architecture Diagram



SQLite3 is a simple file-based relational database that works by writing all of it's changes into a single db.sqlite3 file. It is would used by the application to store all of the project metadata such as project-video lists and tagging meta-information. It forms the model part of a traditional model-view-controller (MVC) architecture.

Django Web Application is a web application framework written in python and uses the WSGI (Web Server Gateway Interface) to communicate with a webserver to allow it to serve web requests. The application will translate the web-requests that it is given, update the database and reply with a given HTML response. It forms the controller of the MVC architecture.

Client Side Web Application is mostly made of HTML and JavaScript that is rendered out from templates by the Django app. It provides an interface to create/read/update/destroy/export projects/videos/tags as required. This forms the view part of a traditional MVC architecture.

Unity Engine The Unity Engine is the game engine that we will be using to perform all of the cross-platform OpenGL rendering and video playing that is necessary to simulate virtual reality and perform 3d tagging.

Android App Store is a virtual storefront that allows users to download and run applications on their Android Phones.

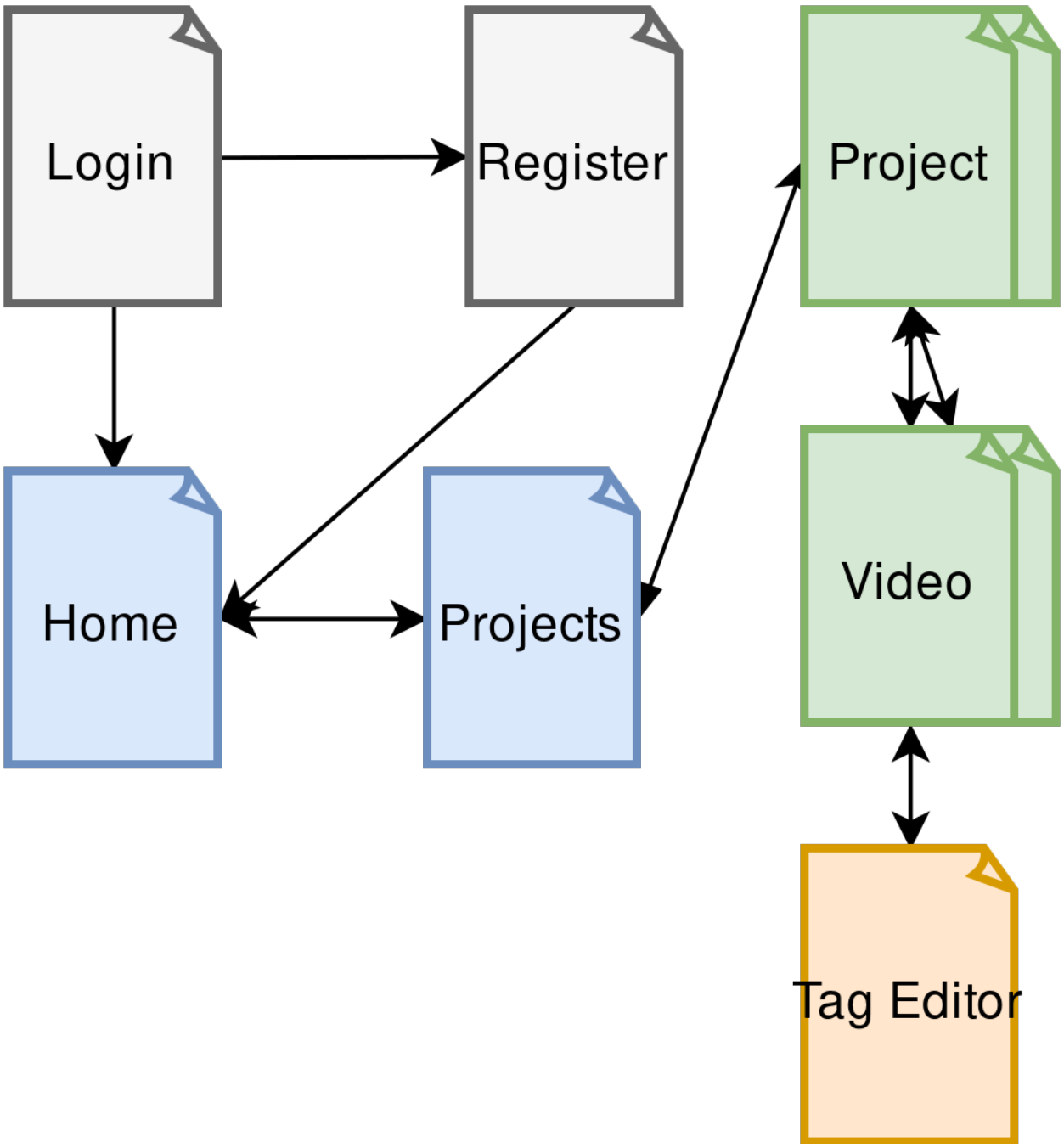
User Phone will run the Unity Application allow the user to interact with the VR space, choose which video to play and handle the display of all the tags.

Oculus Gear VR Plugin is a library that manipulates the Camera in a Unity Scene to track with the user's head movements and allow them use the Gear VR to look around in 3D space.

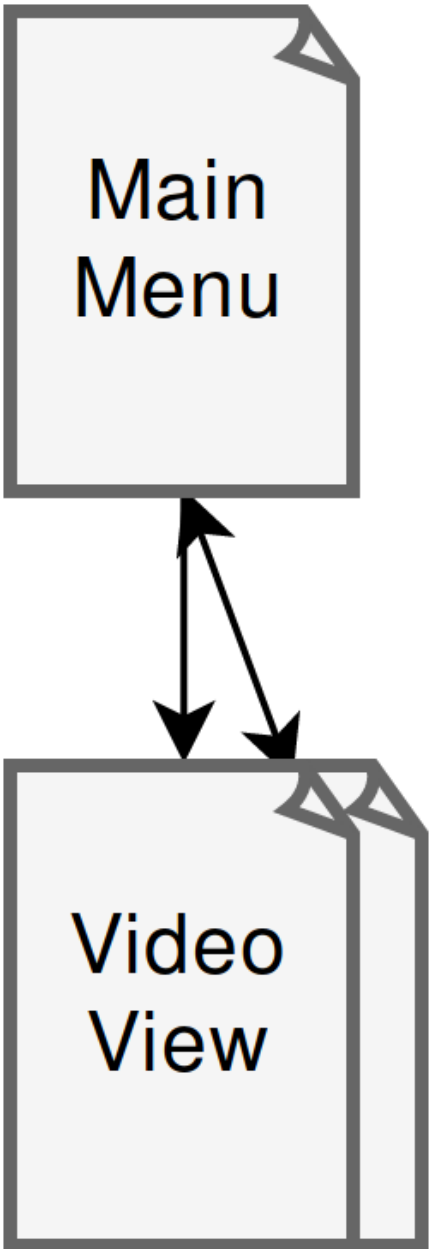
2.1.2 Site Maps

All traversals are user driven, two-way, navigation.

VR Tagging Engine Sitemap

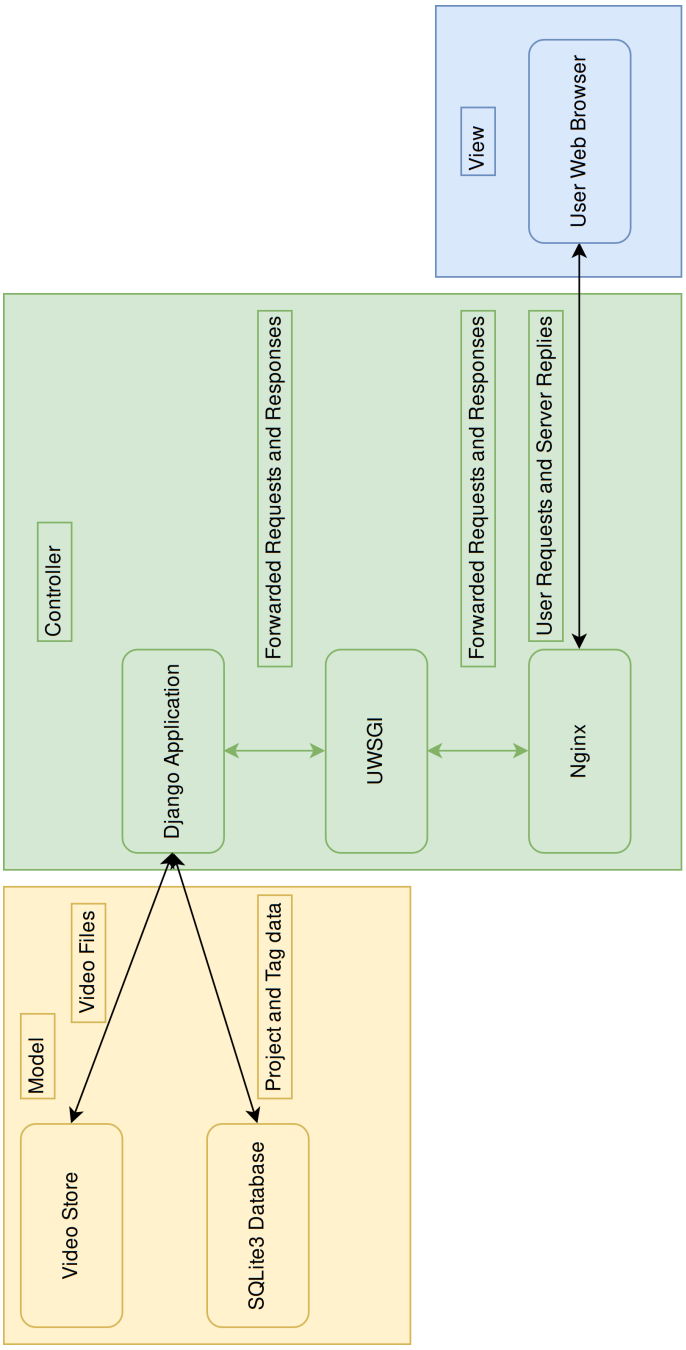


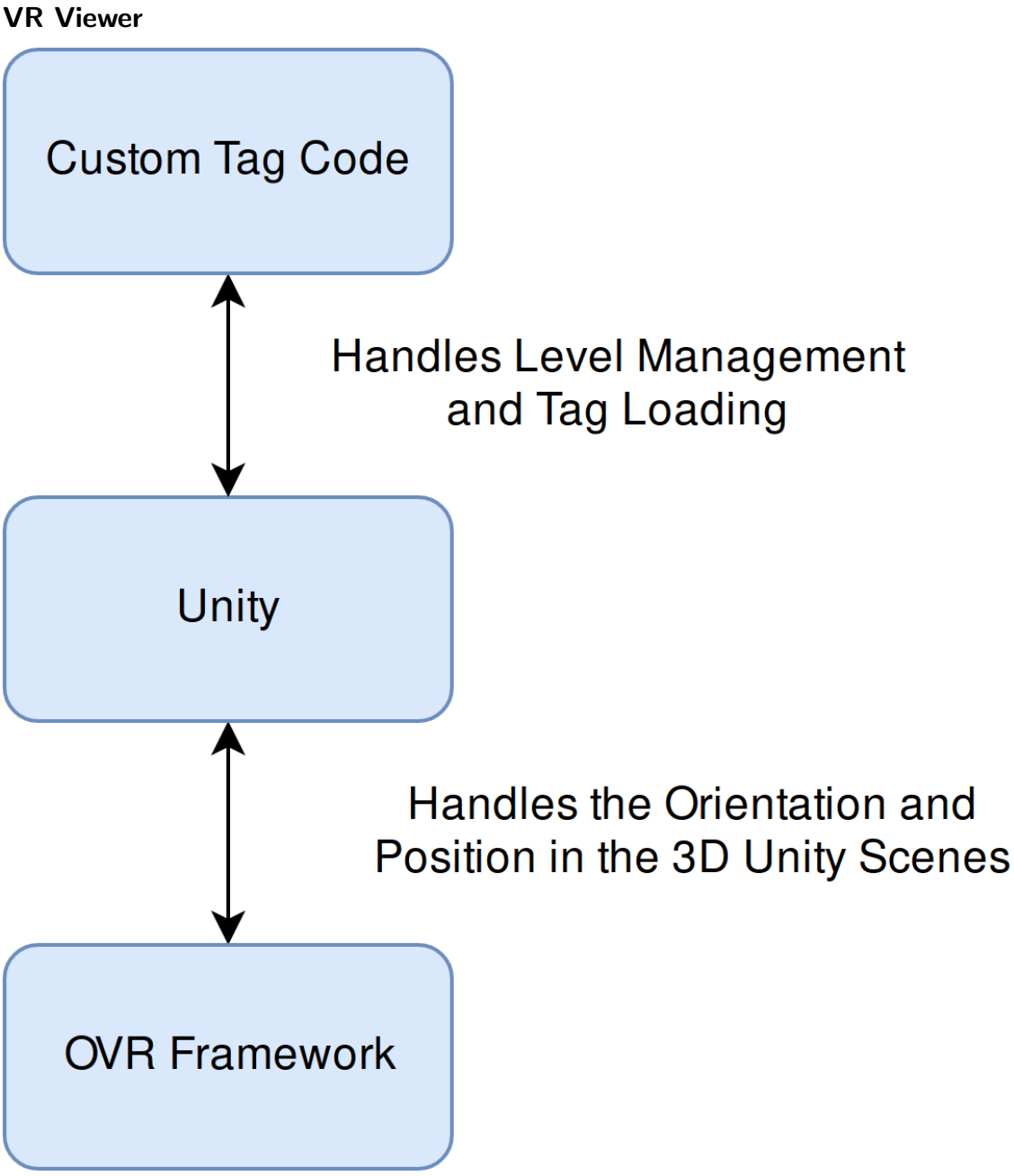
VR Viewer Sitemap



2.1.3 Application Structure

VR Tagging Engine





2.1.4 Design Patterns

Most of the design and architectural patterns we have implemented are as a result of out choice of framework and implementation. However, any design pattern that was not automatically included as a result of a framework as used to ensure logical consistency and practical maintainability. It also had the handy side effect of keeping repeated code to a minimum and allowed for rapid iteration on the projects. This was especially helpful as the build process would sometimes take upwards of 2 minutes and so reducing this time was crucial.

VR Tagging Engine Patterns

MTV — Model Template View This architectural pattern is the core of Django and seperates the concerns of "What data should be shown", "How the data should be shown" and "Which data should be shown". It allows the program to scale quickly and improves iteration times by reducing the amount of work needed to implement a new feature down to the creation of a new view.

Template Methods Django uses this behavioural pattern to great effect with it's class based views and they are used in this application to provide custom functionality with it's class based views. I would use the same pattern for each of the models, in that I would override the `__str__` method of each of the models to provide a string representation that is used in the administrator view for the application.

Adapter Django has a set of classes known as Models which implement the adaptor pattern which adapts the raw SQL interface of the SQLite3 database into a more useful python object containing many methods which form part of Django's object relational model. I could then use the adapted interface in my own code without any issue.

VR Viewer Patterns

Entity — Component Pattern The Entity — Component pattern means that all objects in Unity are either Entities (Unity refers to these as "GameObjects") which actually exist in the simulated space, or Components which are merely a method for describing a small set of reusable behaviors.

Composite Pattern GameObjects in Unity follow the Composite pattern. This means that there is no difference between a tree-like structure of GameObjects and a single GameObject. This is because both of these are objects of the same "GameObject" superclass and so can be composed in a recursive manner. This allows for complex scenes to take place as nested GameObjects inherit a local coordinate space from their parents, meaning that 3D transformations can be stacked together with ease.

Adapter The Android WebView Plugin written by Amartya adapts Android's native WebView class into one that Unity can use and apply as a texture.

2.2 Implementation

2.2.1 Development Tools

Communication For communication with the client we will use a mixture of Slack messages and Email depending on the nature of the communication and if it needs to go on record as an official decision by the client.

File Sharing For the sharing of large binary files such as Unity projects we will use the file sharing service "Google Drive". This is because standard version control performs very poorly when

dealing with the large binary files usually associated with Unity projects, duplicating them with every revision unless they are exact duplicates of each other.

UI Design The design of the Web Application's Interface will be done in the Django templating language (which renders out to HTML5) and Javascript. The design of the VR Viewer's User Interface will be done in the Unity Editor.

Version Control We will use `git` as our primary method of version control as it is an industry standard with wide support for cloud based storage and branching.

IDE Amartya will be using vim to write the Web Application and Jasper will be using the MonoDevelop to write C# scripts for the Unity Application. Amartya will also use Android Studio to write a web-texture plugin for the VR Viewer.

Local Development Environments Amartya will be using VirtualEnv to set up a easily replicable deployment environment.

2.2.2 Front-End Technologies

VR Engine

The front end of the VR engine was written in the Django Templating language, a small language created by Django to allow for the creation of templates that could render to HTML when given a specific data set which the view has retrieved from the database. The language allows for looping constructs and conditional operations while generating HTML so little hard-coding had to be done to allow for multiple items and as the HTML was all generated on the server side there is little to no need for any client side work, meaning that the VR Tagging Engine can work from a variety of web-browsers.

However, the generated HTML could not be used for everything. In cases where a page reload could not be afforded, such as the Tag Editor, the only solution was to use JavaScript to forward requests such as Tag creation or deletion via AJAX and await the server's response. But this does mean that the VR Engine is very bandwidth efficient as all the calls can be batched up and sent to the server in one go ensuring that duplicated data is not sent. CSS was used to layout the app and ensure that it was visually appealing.

VR Viewer

The VR Viewer was created using Unity, a multi-platform game engine that natively supports VR and has the ability to use native plugins. This meant that it was perfect for creating the 3D scenes required to render the tagged VR content as well as loading Web Content as described by the client using a Native Plugin. The plugin was made in Android Studio and defined a class that extended the native Android Web View, overriding the `onDraw` method to render the view into a bitmap. This bitmap is then drawn into an OpenGL texture on the plugin's main thread. Unity then defines an external texture using the texture pointer created in the plugin and applies that texture to a plane which is then used as a web view tag. Similar processes are applied for all other types of tag, skipping the native plugin steps and using Unity defined helper classes instead.

Unity then handles VR input events from the Gear VR and uses them to play/pause the video while the user is interacting with the tags.

2.2.3 Back-End Technologies

VR Viewer lacks any persistent modifiable data and so does not really have a back-end as such. However the VR Tagger System is a full web application using Django and SQLite3 to handle incoming web-requests. Django was used as it is a very elegant and popular framework written in Python, this meant that it was not only a very simple application to write but also that it should be near trivial to find any developer willing to work on the project and that developer should be able to get accustomed to the environment effortlessly. SQLite3 was chosen as it was the simplest relational database that covered all the features that were needed, and as we do not expect to have a work load on the database (less than 1GB in total size and fewer than 5 concurrent accesses) it should be more than sufficient.

2.2.4 Implementation of Key Functionality

2.2.5 VR Tagging System

RQ1 — The system will have HTML5 tagging in VR space When the client is pointed at a tag editor URL, e.g. `/tagger/project/23/video/20/` which edits the video with id 20, which belongs to a project with id 23, the browser sends a request to the Django application. Which forwards the relevant data to the python function `video_editor`.

```
@login_required
def video_editor(request, project_id, video_id):
    vid = get_object_or_404(Video, id=video_id)

    if request.method == "POST":
        print(request.body)
        data = [item for item in json.loads(request.body) if "
            deleted" not in item]
        for item in json.loads(request.body):
            if item not in data:
                Tag.objects.filter(id=item["pk"]).delete();
        for obj in serializers.deserialize("json", json.dumps(
            data)):
            obj.save()
    return render(request, "tagger/video_editor.html", {"video
        ": vid, "tags": vid.tag_set, "tags_json": serializers.
        serialize("json", vid.tag_set.all())})
```

This loads the relevant video from the Django ORM and the checks if the Request was a POST request. If it was it then gets the JSON that was sent from the client and deserializes and saves the changed tags into the database, deleting tags that have been marked as such, it then re-renders the tag editor sends this data to the user's browser as a response.

RQ4 — Data structure of time indexes and coordinates kept with a collection... This mostly got changed as I realised the user didn't have to upload any content to the web server just to re-download it after the fact instead what the tags store now is something called a `remote_url` which allows the server to know at export time where and what the actual file is, while still allowing the user to rapidly iterate on the tags having to wait on the upload process. The code to create these tags is shown above in the `video_editor` method.

RQ11 — Allow loading in of videos To create a new video for a project the client must first navigate to a project, and then to the new video page of that project. The URL is usually as something resembling the following `/tagger/project/23/video/new/` where 23 is the id of the project that the video is being uploaded to. They then fill out the form that they are presented with and click upload.

Django will then dispatch them to the relevant view function

```
class VideoForm(ModelForm):
    class Meta:
        model = Video
        fields = ["title", "uploaded_video", "project"]
        widgets = {"project": HiddenInput()}

def video_create(request, project_id):
    f = VideoForm(initial={"project":project_id})
    if request.method == "POST":
        f = VideoForm(request.POST, request.FILES)
        if f.is_valid():
            f.save()
            return HttpResponseRedirect("/tagger/project/{}/
               video/".format(project_id))
        return render(request,
                "tagger/video_detail.html",
                {"video_form": f})
    else:
        return render(request,
                "tagger/video_detail.html",
                {"video_form": f})
```

A `VideoForm` is an python class that extends `ModelForm` a Django class that reads it's meta class to find a way of auto generating a form that can be used in a Django template. The project field is hidden as it is auto filled from the URL and the user should not be able to edit it directly. The form then has an is valid method that validates the contents of request.POST and request.FILES to check if they match the format given by the model. If they do then all of the data of the form is saved. As it is a model form this means that a Django object will be created with all of the settings and data from the request, and as the `FILES` object is passed in this includes any videos that are sent as part of a multi-part request. The user is then redirected to the project's video overview screen. If it

is not valid then the user is served a refreshed page, but with a form that contains errors. This means that when the form is generated the errors are automatically included in the generated HTML with a useful CSS class. If the incoming request is not a post request then a request with a VideoForm is made only setting the project value to the incoming project id.

RQ12 — Allow editing of custom project format The source code also contains methods called `project_list`, `project_create`, `project_detail`, `project_delete`, `project_export`, `video_list`, `video_create`, `video_editor` and `video_delete`. These functions perform the other aspects of the CRUD (Create/Read/Update/Destroy) that is required for the app.

2.2.6 VR Viewer

The VR viewer is starts by loading up the meta.xml file which is a section in the generated project file that states what videos are included in the project, what their tags are, if those tags are remote and so on. An example meta.xml looks as follows.

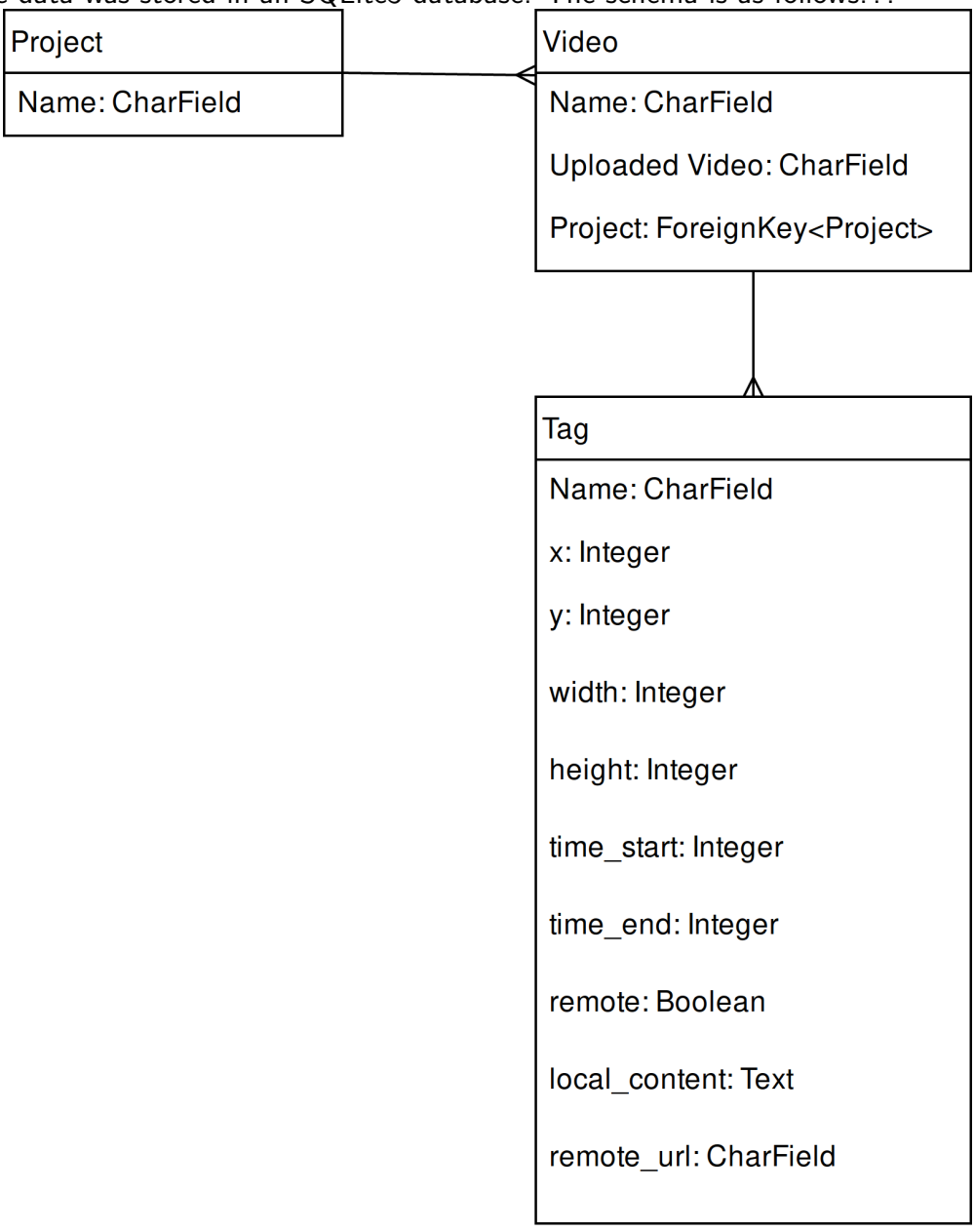
```
<!-- Handy Meta File -->
<project id="24" title="Testing the File Exporter">
    <video id="21" title="Star Warz" location="project/videos/
        output_ke4Kqgg.webm">
        <tag remote="false" x="50" y="50" width="500" height="500"
             tstart="55" tend="60" local_content="Dod Gamn
            Stormtroopers"/>
        <tag remote="true" x="50" y="50" width="1024" height="1024
            " tstart="62" tend="72" remote_url="www.google.com/?q=
            Darth"/>
    </video>
    <video id="22" title="BBB" location="project/videos/big-buck-
        bunny_trailer_9RD0r16.webm">
        <tag remote="false" x="50" y="50" width="50" height="50"
            tstart="5" tend="7" local_content="Tree"/>
        <tag remote="false" x="50" y="50" width="50" height="50"
            tstart="10" tend="11" local_content="Bunny"/>
        <tag remote="false" x="50" y="50" width="60" height="60"
            tstart="18" tend="23" local_content="You done goofed"/>
    </video>
</project>
```

The Unity application then parses this data and uses it to create a main menu using the New Unity GUI sub-system. This menu has a maximum video count of 20, meaning requirement RQ18 is fulfilled. It does this by parsing the XML using the System.XML library and extracting a list of videos. It then iterates through the list and uses the title attribute of a video tag to generate the necessary title for the menu button. Upon clicking this button a new scene is loaded in which the application does several things simultaneously. It renders the video on to a sphere fulfilling requirements RQ2, RQ5, RQ14 and RQ16. It then re-parses the meta.xml file to get

access to all of the tags under the current video. For each tag that was created in this way, a `CreateTags` script creates a prefab which contains disabled variants of all the possible tag types. Then it performs a web-request to the remote url if the tag is a remote one and gathers the tag type from the mime-type of the resultant file. Otherwise the type of the tag is just text. The `CreateTags` script then Instantiates clones of that prefab in all of the places were they can be and enables the one that matched the type needed. This then proceeds to fulfil all of the other requirements.

2.2.7 Data Storage

The data was stored in an SQLite3 database. The schema is as follows...



Appendices

Appendix A

Deployment Manual

A.1 Deploying the VR Tagging Engine

To begin deploying the app follow these command line instructions

```
$ mkdir vr_tagger
$ cd vr_tagger
$ virtualenv .
$ source bin/activate
$ git clone https://github.com/Napoleon1769/COMP103_AppProject #
   change the URL as needed
$ cd COMP103_AppProject/prototype/web_app/
$ pip install -r requirements.txt
$ touch video_tagger/video_tagger/secrets.py
```

Now open `video_tagger/video_tagger/secrets.py` in your text editor and add this line to the file

```
SECRET_KEY = r"""^h*y_1+w2s#z(jelq4@^\blg21t*@9*r03)#2*xf7o5x%*v^(
   g"""
```

Note: If you are doing anything remotely security related you should replace the contents of this secret key with a new key from `http://www.miniwebtool.com/django-secret-key-generator/`.

Then you should run these commands

```
$ python video_tagger/manage.py migrate
$ python video_tagger/manage.py runserver
```

The server should be at http://localhost:8000/tagger/login/

Note: If you close the shell that is running the server you will have to resource `bin/activate` by running `source path/to/activate`.

A.2 Deploying the VR Viewer

Start by installing Unity and the Dependencies that it needs to build applications for the Android Operating System. Then follow the instructions as shown below.

1. Extract the Unity Project into a directory
2. Launch Unity and open the project that you just extracted with it
3. Download the completed project.zip from the VR Tagging Engine
4. Extract project.zip
5. Copy `meta\meta.xml` into the root Assets directory
6. Copy all of the videos into the Streamed Assets directory
7. Navigate to the `MainMenu` scene and find the `Canvas Object's Start Up Component's Game Asset` variable. Set this field to the new meta.xml file in the root assets folder.
8. Connect your Samsung S7 to your PC and enable both Development Mode and USB Debugging
9. Press `Ctrl-B` to build and run the application on the S7

Appendix B

System Manual

B.1 Working on the VR Engine

The VR Engine it a Django + Python application, meaning that all the best practices are well documented on it's official website `https://docs.djangoproject.com/en/1.11/`. All of the python code should comply with the PEP8 coding standard which can be found here `https://www.python.org/dev/peps/pep-0008/`. The tagger application is the one that handles the vast majority of URLs that the application can reach.

B.1.1 API Endpoint Map

login/ GET and POST supported. Handles all logins to the system

project/ GET only, login required. This view shows a generic list of all projects currently managed by the system.

project/new/ GET and POST supported, login required. This view handles the creation of new projects and any validation associated with this action.

project/$project-id/ GET and POST supported, login required. A view used to rename a project that is already existent

project/$project-id/delete/ POST only, login required. Deletes the project with the given id, all child videos and all grand-child tags

project/$project-id/export/ GET only, login required. Creates a project.zip file for the project with a given ID.

project/$project-id/video/ GET only, login required. This view shows a generic list of all videos owned by a specific project with the given project ID.

project/$project-id/video/new/ GET and POST supported, login required. Creates a new video owned by a specific project with a given project ID. The POST request to this endpoint should be multi-part data with a video attached.

project/$project-id/video/$video-id/ GET and POST supported, login required. Displays a tag editor on GET. POST data should be a JSON list of all the tags that should belong to this video. Marking with deleted if it needs to be.

project/$project-id/video/$video-id/delete/ POST only, login required. Deletes the video with the given video id and all child tags associated with it.

B.2 Working on the VR Viewer

The purpose of the relevant scripts are as follows.

TagScript.cs Handles the tag object and deletes the tag when the time end for the tag is reached.

StartUp.cs This script runs when the main menu scene is loaded and creates the UI for the user by reading data from the xml document. It also sets the PlayerPrefs needed to access the data from the xml in the video player scene. It also handles user input from the Gear VR to choose a video to play.

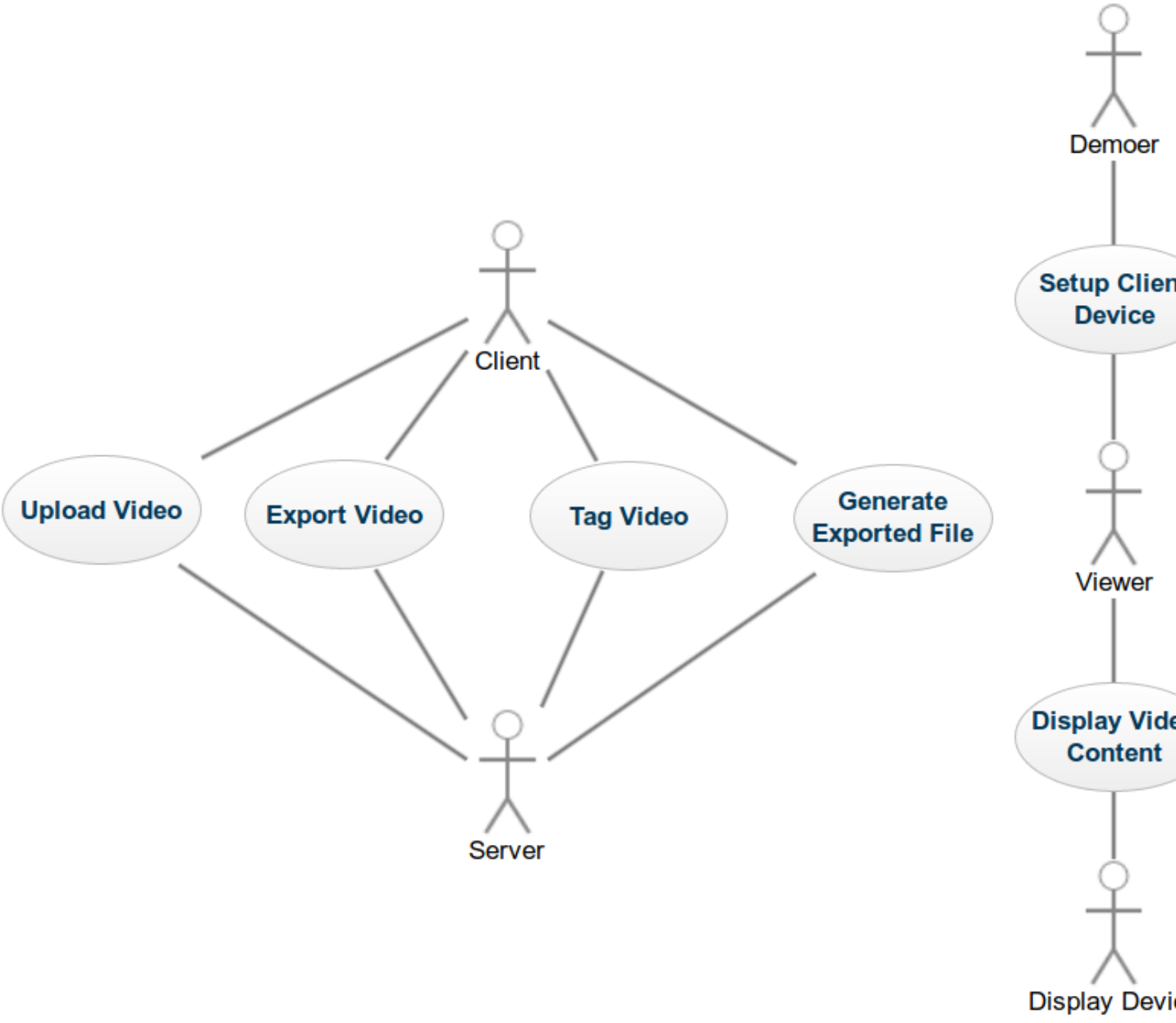
MoviePlayerSample.cs This script plays the video selected by the user on a prefab called sphere.

CreateTags.cs This script creates the tags in the video scene and instantiates them in the right position with the relevant content.

ApplyWebTexture.cs This script instantiates a WebView object and uses the functions associated with it to apply a texture to the mesh render of the game object that it is applied.

GlobalTexUpdate.cs This script handles all of the events from all of the ApplyWebTexture scripts in the scene. Forwarding `repaint` and `input` events.

WebView.cs This script is an Adaptor for the Android WebView Class which adapts it's interface into one that Unity can easily use.

Appendix C

Usecase Diagram