Universitat Politècnica de Catalunya (UPC)

Facultat d'Informàtica de Barcelona (FIB)

Master in Innovation and Research in Informatics

Computer Graphics and Virtual Reality

A WebXR-based plattform for mixed geometry-based and image-based exploration of cultural heritage models

Author:

Arnau Farràs Llobet

Tutor:

Carlos Andujar Gran

Co-Tutor:

Marc Comino Trinidad

June 2021

Abstract

The great advances in 3D digitization and upward trend in remote services have led to a growth in demand for virtual museums and applications to explore cultural heritage. While it is true that virtual museums are not a novelty, many of these applications are still not very accessible or present several limitations. In this master's thesis, we have developed a web application to explore cultural heritage models without restrictions. The application offers the user a collection of high-quality photographs from the environment surrounding the user, which can be projected onto the model. Our application is also compatible with virtual reality headsets providing the same features in a much more immersive environment. After developing our application, we have carried out a pilot user study to assess its effectiveness objectively and know the issues that should be addressed in the final product.

Contents

1 Introduction

Thanks to the great advances in laser scanning and photogrammetry in recent decades, 3D digitization of cultural heritage has become a common practice in this domain [29]. The digitalized cultural heritage has proven to be particularly useful and demanded in many fields. One of its main advantages would be that it allows preserving the geometrical information of the works in digital format, thus protecting them from any deterioration caused by time or the human being himself [11]. Besides, working with scanned models allows historians in a much easier and faster way the virtual reconstruction of structures damaged or deteriorated by time or other historical events, thus avoiding using the original pieces, which are usually much more fragile or difficult to handle [9] [26].

Another interesting application and the one we will focus on in this work is that the virtual reconstruction of cultural heritage allows both historians and regular users to study and visualize highly detailed reconstructions of monuments and works of art and in a virtual way, using his own computer or mobile device, without the need to go in person to the place where the original work is located and without having to install and use any specialized software.

There are many techniques to digitalize cultural heritage. Between them, the most popular ones are photogrammetry and 3D terrestrial laser scanning [29]. The first one uses the matching points between a collection of overlapping photographs to compute the full 3D mesh. In the case of laser scanning, we measure the distance from the scanner to the surface, measuring the time elapsed between the transmission and reception of the laser pulse [27]. Both methods will give us pointclouds of the artwork or the monument. There are techniques to generate 3D meshes from these pointclouds. However, these often present holes or other artifacts. Currently, some algorithms solve a large part of these imperfections, but in some cases, the manual repairing of the mesh is still necessary. To recreate the texture of the model, the same scanners can usually capture the color at each scan point. Although the quality of the generated texture is often very poor, in some cases, it is sufficient for the desired task. In the circumstances where more detail is needed, there are methods such as projecting real images onto the mesh itself.

In this work, we present a novel web-based application for exploring cultural heritage. The models displayed are highly detailed digital 3D reconstructions from real monuments and artworks. Being a web application, it does not require the installation of additional software. It is accessible from the browser, desktop computers, and any mobile device with an internet connection.

The application offers a complete interface that allows both professionals and ordinary users to navigate within the 3D scene. Through different modes and configuration options, the user can select and inspect specific regions of the model as well as view and compare real photos of the chosen area. Finally, the application is compatible with virtual reality headsets. If available, the user can explore and interact with the scene in a VR environment, thereby providing better immersion.

In an increasingly globalized world where there are already countless monuments and digitized works of art, our application allows users to visit them in a comfortable and efficient way. It also supposes a solution to any mobility problem so that it would be helpful in situations such as the current health emergency due to COVID-19.

2 Previous work

2.1 Cultural Heritage Digitalization

The evolution of humanity comes from the learning of the previous generations. For this reason, the preservation and exhibition of cultural heritage assets can be considered a priority for every nation. With the beginning of the computer era, digitalization plays an important role in that [11]. We already have publications from the 60's like the ones by George Cowgill talking about the potential of raster devices applied in the archaeological field [17].

From then until now, progress in the field of digitization has gone hand in hand with technological evolution. Among all the methods, photogrammetry and 3D laser scanning stand out. As pointed by F. Fassi et al [19], comparing the effectiveness of both methods to reconstruct complex and extensive heritage areas, the decision to use laser or photogrammetry should depend on the project purpose. On the one hand, terrestrial laser scanning proves to give a higher precision in small areas. However, when the laser has to travel long distances, the results are not as good. On the other hand, using photogrammetry, the precision of the topological information will be lower than laser scanning but could be our solution if the goal is to digitalize large areas.

There are many other examples of the use of these techniques: Similar to F. Fassi, Naci Yastikli [30] compared both methods to reconstruct historic buldings. Lucia Arbace et al, 2012 [10], used 3D scanning with the help of advanced modeling to create digital 3D replicas of a fragmented terracotta statue (The Madonna of Pietranico). Annabelle Davis et al [18], uses laser scanning, photogrammetry and 3D photografic methods applied to the reconstruction of rock art. In 2013, Pedro Martín Lerones et al [26], used the projection of 2D artistic images to previously scanned models. This was useful to emulate the primitive appearance of the artwork and compare the evolution and the effects of deterioration over time.

2.2 Web navigation and VR for 3D enviorments

We start from the desire to offer the user an interactive application in a 3D environment. As we see in the work by J. Jankowski and M. Hachet on advances in interaction with 3D environments [23], this is a highly explored field. Their study reviews an extensive catalog of interaction techniques for navigation, selection, and manipulation in 3D environments not only at the level of specialized software but also at the level of Web-based environments. The ultimate goal is to help other researchers and developers find the interaction techniques that best fit the needs of their project.

When presenting a 3D environment representing a work of art or a monument that exists in the real world, it is essential to offer the user the maximum possible immersion. To achieve this goal, a very effective way is through virtual reality (VR) [22]. In this field, we have two main interaction paradigms, mixed reality and virtual reality itself. The first one, also commonly called augmented reality, is a combination of what we perceive with our senses and a synthetic virtual scene that would add additional information to the real environment. In the second one, and in which we will focus on this project, the immersion is complete. Commonly through Head Mounted Displays (HMDs), the visual senses are replaced to perceive only the virtual scene.

Virtual reality has already been brought to the web environment before. Until now, the WebVR [7] interface has been trendy for this purpose. However, it is currently considered obsolete due to the emergence of WebXR [8], which adds new features, including augmented reality.

2.3 Showing cultural heritage

Currently, we can already find projects that allow any user to visualize 3D environments of cultural heritage, whether they are small artwork, architectural structures, or even large open spaces. A good example could be the case of Photo Tourism [28] that present a software to explore large unstructured collections of photographs, generating a 3D scene that can be explored freely by the user. To achieve that, they use image-based rendering techniques that allow a smooth transition between the different photographs. In addition, Photo Tourism provides additional information through annotations about the current place that is being visited or the element that the user is looking at, as well as browsing other nearby or relevant photographs.

Similar to Photo Tourism, Paolo Brivio et al [13] present Photocloud, a system for interactive exploration for large datasets using several thousand photographs calibrated over 3D data. All the images of the model are displayed at the bottom bar as thumbnails. These are grouped by similarity and are sorted according to relevance given the current position and orientation of the camera. A very interesting feature is that the user can not only navigate between the proposed images but can also project them in the 3D model thus achieving a higher level of detail.

Mila Koeva et al (2016) [25] used a diferent aporach to visualize Cultural Heritage. Using a set of high-resolution spherical panoramas in a Virtual Reality environment followed by sounds, video, and informative texts, they achieve a significant effect of immersion to the user. Another feature to highlight is that it is a web-based portal which makes it highly accessible. Another example of visualization of Cultural Heritage through VR can be seen in the work of Choromański et al [14] where they implemented a visualization based on multi-source 3D data from archives of Museum of King John III's Palace (Wilanów). The aim of their work was to help the user to familiarise with the Museum architecture and history in the closest way as it could be in a real visit. Dimo Chotrov, and Angel Bachvarov [15] present another web-based framework with virtual reality integration that allows the user to navigate freely in the 3D scene. The advantage of this last work is that it simplifies the loading of models through templates and configuration files, making it accessible and easy for any user to use their own models without the need for high programming skills.

Although Photocloud and Photo Tourism present a work very similar to what we are looking for, they require the installation of specific software that makes them less accessible and more limited in terms of use on different devices. On the other hand, Mila Koeva not only presents a visualization of cultural heritage in a web environment but also offers the possibility of using VR to achieve more immersion, even so, the fact that the navigation is through the change between panoramas limits interaction. In our work, we present a complete and free 3D navigation so that you can investigate more specific details of the scene. Dimo Chotrov, and Angel Bachvarov give the user full navigation but beyond this the interaction with the scene is very poor. In our project we seek that the user can select specific areas of the environment to be able to observe in more detail, like it was done in Photocloud but maintaining the advantages of being a web-based portal and immersion through VR.

3 Application goals

The first step when developing our application is to be very clear about its goals and the series of tasks that users should be able to perform with it.
In our case, we will split these objectives into different categories (visualization, navigation, interaction, VR, accessibility)

1. Visualization
    - (a) The user should be able to visualize different models of cultural heritage composed by a 3D mesh and a texture.
    - (b) The user will be able to explore and inspect high quality photographs of the model from a collection.
    - (c) The user will be able to visualize the selected 2D photographs in a separate view and inspect the photo more closely in fullscreen.
2. Navigation
    - (d) The user will be able to move and rotate the camera in any direction without constraints inside the 3D scene.
    - (e) The user will be able to zoom and pan inside the scene
    - (f) The user will be able to zoom and pan inside the secondary 2D view from the selected photograph.
    - (g) The user will be able to highlight in the 3D scene the region of the selected photograph as well as project the 2D image into the scene.
3. Interaction
    - (h) The user will be provided with an interface with a list of settings to switch the model, enable/disable GUI elements and decide how the 2D photographs are displayed.
    - (i) The user will be able to use different methods and algorithms that allows choosing the relevant photographs from the suggested ones.
    - (j) The user will be able to select a region from the scene to show relevant photos from the chosen area.
    - (k) The user will be able to switch between the 2D/3D views in fullscreen.
    - (l) The user will be able to teleport to the capture position of a selected photograph from the collection.
4. VR
    - (m) The user will be able to use a VR headset. The application will track the head position to update the camera orientation.
    - (n) The user will be able to use VR controllers to perform the same interaction tasks in the virtual environment.
5. Accessibility
    - (o) The application will not require the installation of any software.
    - (p) The application will be available from the browser using an internet connection.
    - (q) The application will support the keyboard and mouse as well as touch controls for mobile devices.

4 Application description

Now that the goals have been defined, we are going to describe the operation of the application and explain how it solves each one of them. We will only describe the application from a user point of view. Implementation details (for example, how photos are scored and clustered) will be discussed Section 5.

4.1 Starting the application

Currently, the application is hosted on the following server:
https://laurelin.synology.me

The application is based on the Three.js library, which uses WebGL to display the scene. Due to this, the application can be used in any browser compatible with WebGL such as Google Chrome 9+, Firefox 4+, Opera 15+, Safari 5.1+, Internet Explorer 11, and Microsoft Edge. (Objectives o, p)

If the browser is compatible, the user only needs to access the server URL from any device. Given that some of the models and photographs are shown in the application are not in the public domain, access is protected with a login page with a username and password.

4.2 Scene navigation

Once inside the application, a loading bar will appear. Some models and textures are heavy, and the loading time will depend mainly on the quality of the network connection, although it should not take more than a few seconds (Figure 1).
When the loading process has finished, the scene will be shown from an arbitrary position. By default, the "Pedret" model will be loaded as shown in Figure 2.

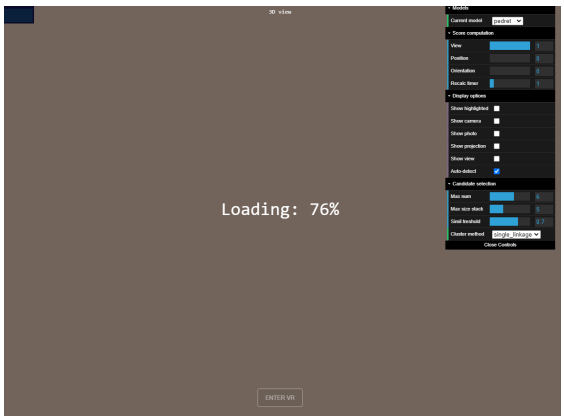

Fig. 1: Application during the loading process

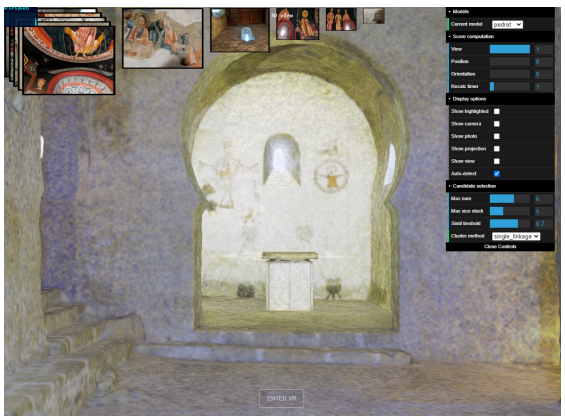

Fig. 2: Pedret model startup

We can explore the scene by changing the position and orientation of the camera using the keyboard and the mouse. The controls are intuitive and the usual ones in this type of application.

- **Move forward/backward**: Mouse wheel.
- **PAN**: Mouse right button + drag.
- **Change camera orientation**: Mouse left button + drag.

For mobile devices, the *touchpad* can be used to perform the same actions. (Objectives a, d, e, q)

4.2.1 Image collection

A very powerful tool is the image collection. It can be activated and parameterized from the application interface. The collection is located at the upper part of the screen and consists of sets of images grouped in stacks along the screen's horizontal axis (Figure 3).

The images shown are thumbnails of the actual photographs of the model. The images belonging to the same stack share similarity between them. On the other hand, the size of the stack indicates the degree of relevance for the user according to the current position in the scene and other parameters, where the largest is the most relevant.

When hovering the mouse over an image, this will be shown in the auxiliary view (if enabled). While hovering, the user can use the wheel button to explore images from the current stack. Finally, by *clicking* on the image, we can automatically move to the position where the photograph was taken. (Objectives b, l)

Fig. 3: Application showing the image collection

4.3 Secondary view

One of our most important objectives of this project was that the user could explore the cultural heritage environment in very high quality. Unfortunately, the meshes and textures obtained from the scan are sometimes quite limited. Even projecting the photos in the 3D environment can generate errors or deformations. To solve this problem, we have added the secondary view. It comes disabled by default but can be enabled in the options menu.

As we can see in Figure 4, the secondary view is presented as a small box in the lower right corner and shows a projection of the scene. The camera's position, orientation, and projection are based on the last image selected from the collection.

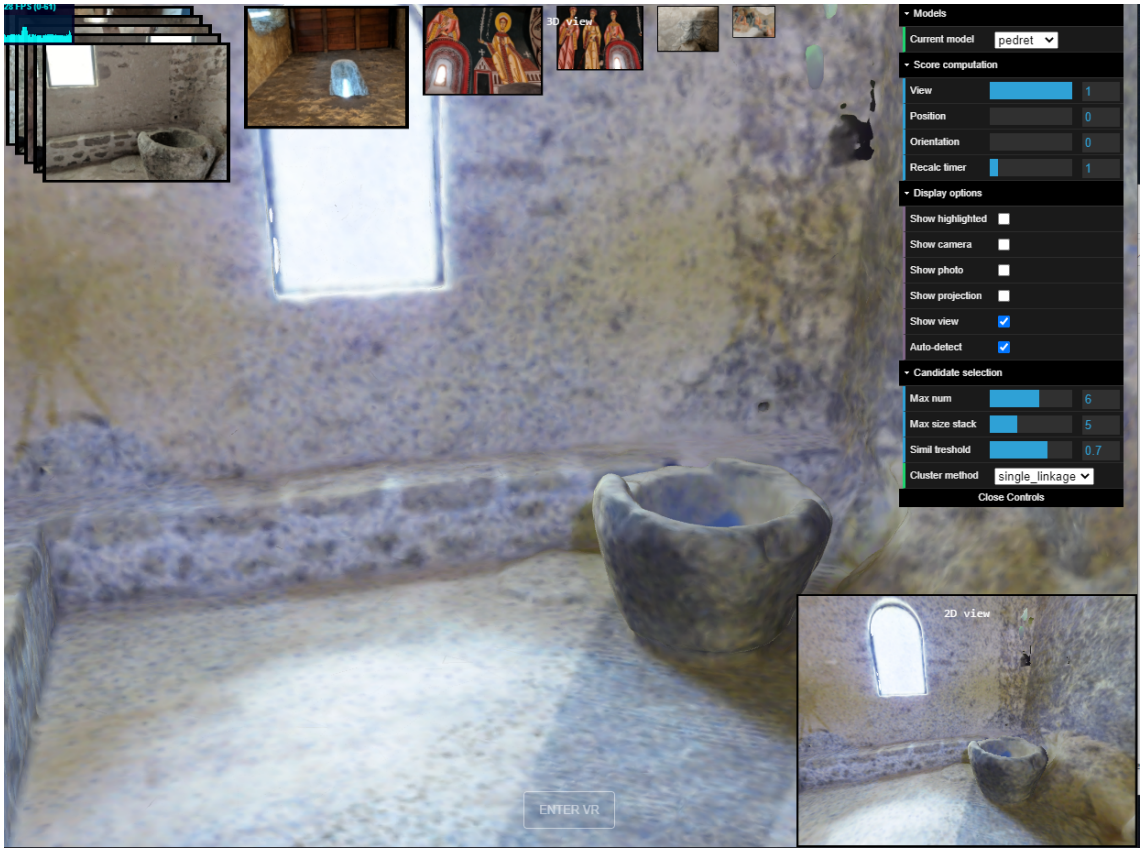

Fig. 4: Secondary view (bottom right) showing the projection of the scene based on the first image of the collection.

We can decide if we want the same photograph to be shown inside the view through the options menu. In this way, the user can understand better how the image is related to the model and its delimitation (Figure 5). Also, it may be the case in which we want to inspect these photographs more thoroughly. For this, the application offers the possibility to see the secondary view on fullscreen. By *clicking* on it, the main and secondary views are swapped. (Objectives b, k)
While we are in fullscreen, we can inspect the photograph in more detail by using zoom and pan, as can be seen in Figures 6, 7. (Objectives c, f)

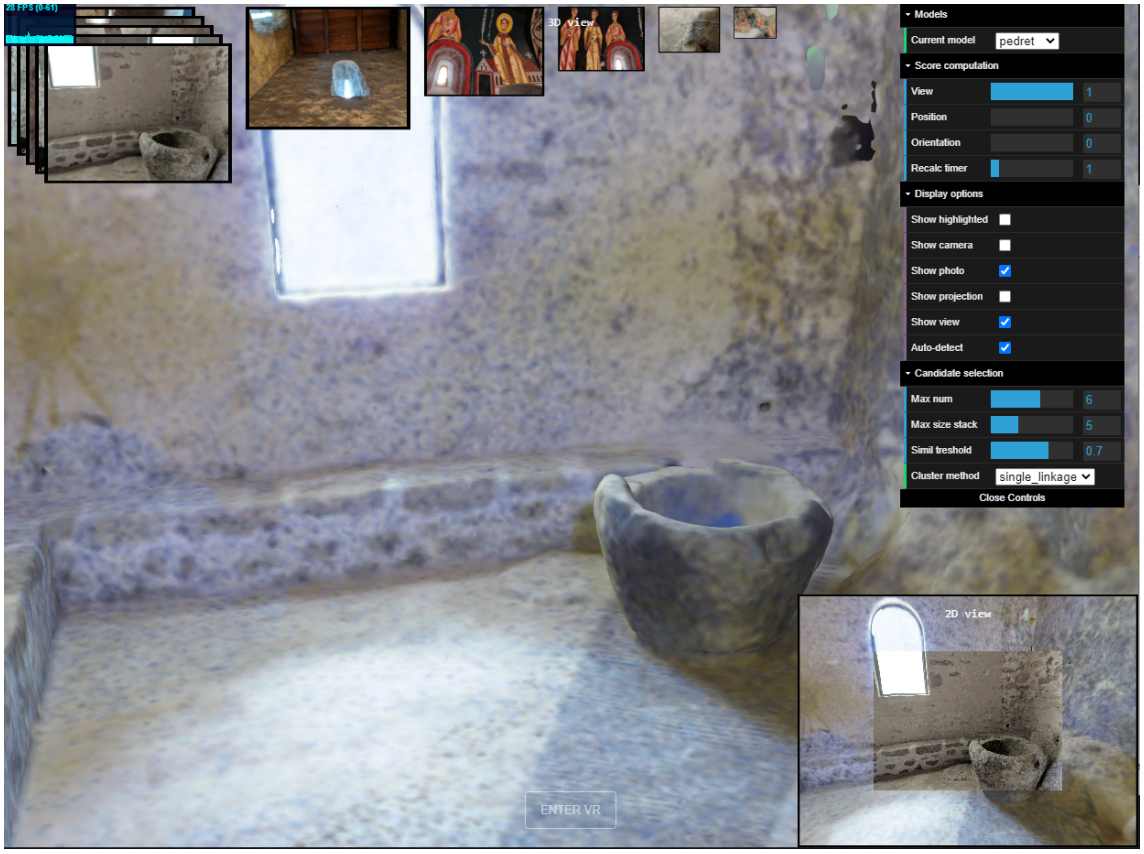

Fig. 5: Secondary view showing the selected image.

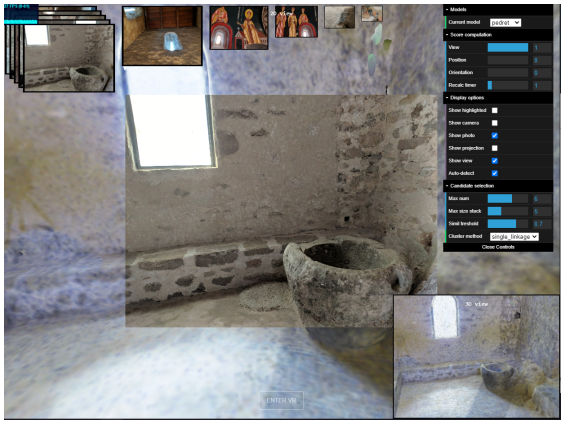

Fig. 6: Secondary view in fullscreen showing the selected image.

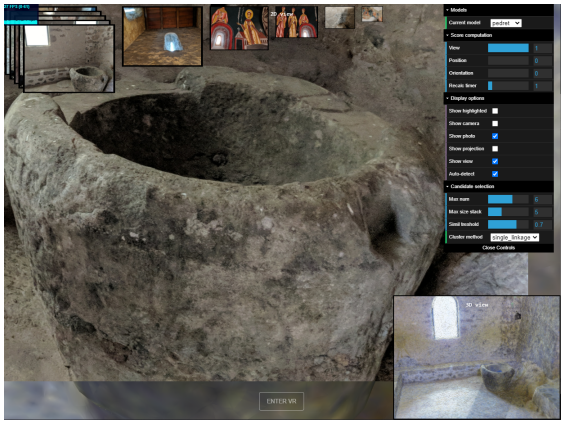

Fig. 7: Secondary view zoomed to apreciate the small details in high quality.

4.4 User interface and interaction

Since the tasks and goals can vary according to the user and the model, the application offers an extensive menu of options on the upper right side to customize the interaction. (Objective h)

Scene interaction and tools:
One of the most interesting tools is the selection box. At any time, the user can select an area of the scene (Ctrl + mouse drag). Automatically, the collection is filled with the most relevant photos according to the chosen area. Additionally, each image is highlighted with an approximation of the selected area. Given that sometimes the photo collections are very extensive, this tool becomes handy to find the most representative images of a particular area. Check the behaviour of this tool in Figures 8, 9.

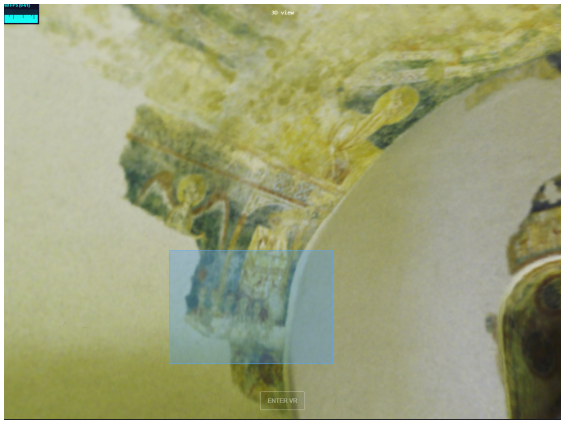

Fig. 8: User selecting a region in the scene. All the UI elements are hidden to facilitate the selection task.

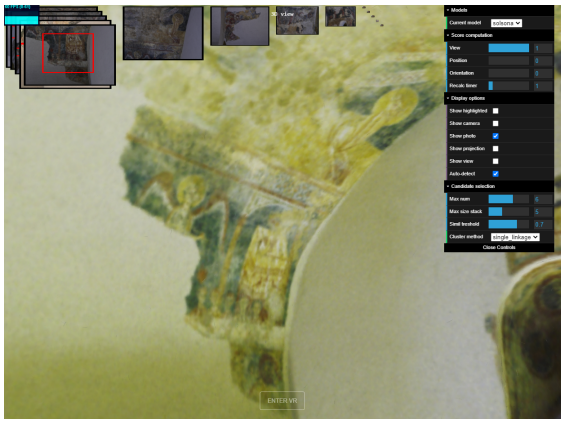

Fig. 9: The collection is filled depending on the specified region. The selection area is highlighted in the thumbnails.

Apart from this, in the options menu, we can activate some elements for the scene.

- Photo area: Displays a red transparent area that highlights the area covered by the selected photo (Figure 10).
- Photo projection: Similar to the previous one, the photograph is projected on the scene. It becomes helpful to see the real picture in the 3D environment, but depending on the angle of observation, some areas will be deformed (Figure 11).
- Secondary view: Enables/disables the 2D auxiliary view.
- Show camera: Shows the camera frustrum of the selected image in the scene (Figure 12).
- Show photo: Shows the photo in the secondary view. Also, if the "show camera" option is enabled, the 2D image is shown in the principal view (Figure 13).

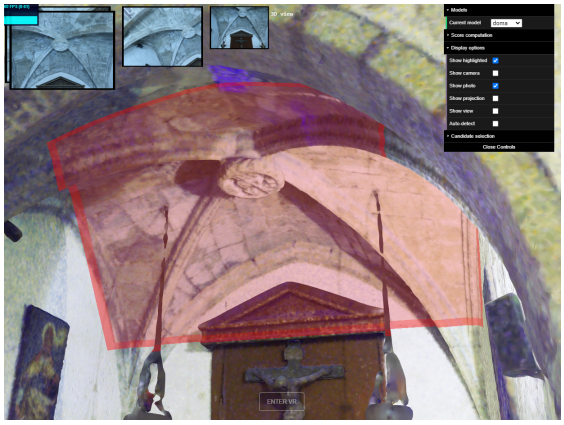

Fig. 10: Showing the highlighted area of the selected image.

Fig. 11: Showing the projection in the scene of the selected image.

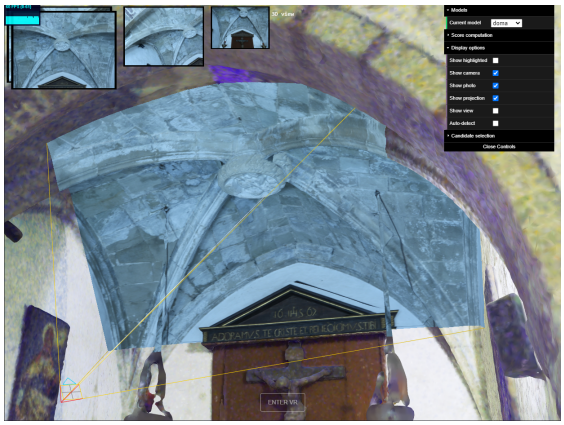

Fig. 12: Showing the camera frustum in the scene.

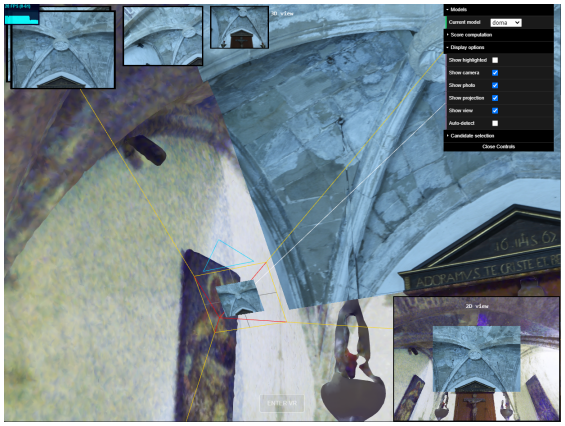

Fig. 13: Showing the image in the secondary view and in the camera near plane.

Photo collection:
There is a set of options that allow us to customize the distribution of the collection of photographs (Figure 14). On the one hand, we can define the maximum number of images allowed in the same stack and the maximum number of stacks that can be displayed. On the other hand, we can choose the algorithm that decides which images to show and how they are distributed in the stacks:

- Basic clustering: A fast but not very precise method.
- Single linkage clustering: A much more precise but slower method.

Both methods are explained in more detail in Section 5.

Additionally, we have a checkbox that allows us to activate the automatic suggestion mode. This mode automatically renews the photo collection according to the camera's position and orientation in the scene. Three control bars allow us to tell the influence of the user camera's position, orientation and projection when generating the collection. The fourth bar serves to define the waiting time for the new computation.

Fig. 14: The red rectangle shows the specific tool menu for the camera collection.

4.5 Virtual Reality mode

If you have a virtual reality headset with controllers, the application can be used in an immersive environment. In the virtual environment, you can perform the same tasks as in the normal one. In this case, the camera's orientation is tracked by the headset, and the UI elements are displayed as floating panels within the environment. With the virtual controllers the user can use the pointer to interact with the UI elements. Also, the joystick is used to navigate inside the scene. (Objectives m, n)

Fig. 15: User enjoying the VR experience.

Fig. 16: Scene displayed in VR mode.

5 Implementation

In this section, we will explain in detail the implementation of each aspect that takes part of our application. We also will describe the development process and decision-making in each of the features.

5.1 Application Structure

First of all, we are going to see a general description of the entire application that will help us to summarize the key steps which we will go into in detail in the next subsections.

Our goal was to develop a web application that would allow us to represent 3D models and provide a virtual reality environment. The most common way to achieve this is to use WebGL [6], a low-level API implemented in Javascript that allows rendering 3D graphics on any web browser. Still, developing a WebGL application from scratch that meets our goals can take a long time. For this reason, we decided to base our application on Three.js [5].

Three.js is a javascript library that acts as a layer on top of WebGL. This library allows the creation of scenes in which we can add meshes with geometry and texture. You can also add lights and cameras. Finally, we can create a renderer in charge of drawing the scene from a specific camera. All these elements can be handled or modified in a simple way. We do not have to worry about anything else as the Three.js layer itself will take on the task of translating all this to WebGL.

Another clear advantage that Threejs provides us is that it also allows us to abstract from the WebXR API [8], thus facilitating the implementation of virtual reality features. You can check the layered model description of our application in Figure 17.

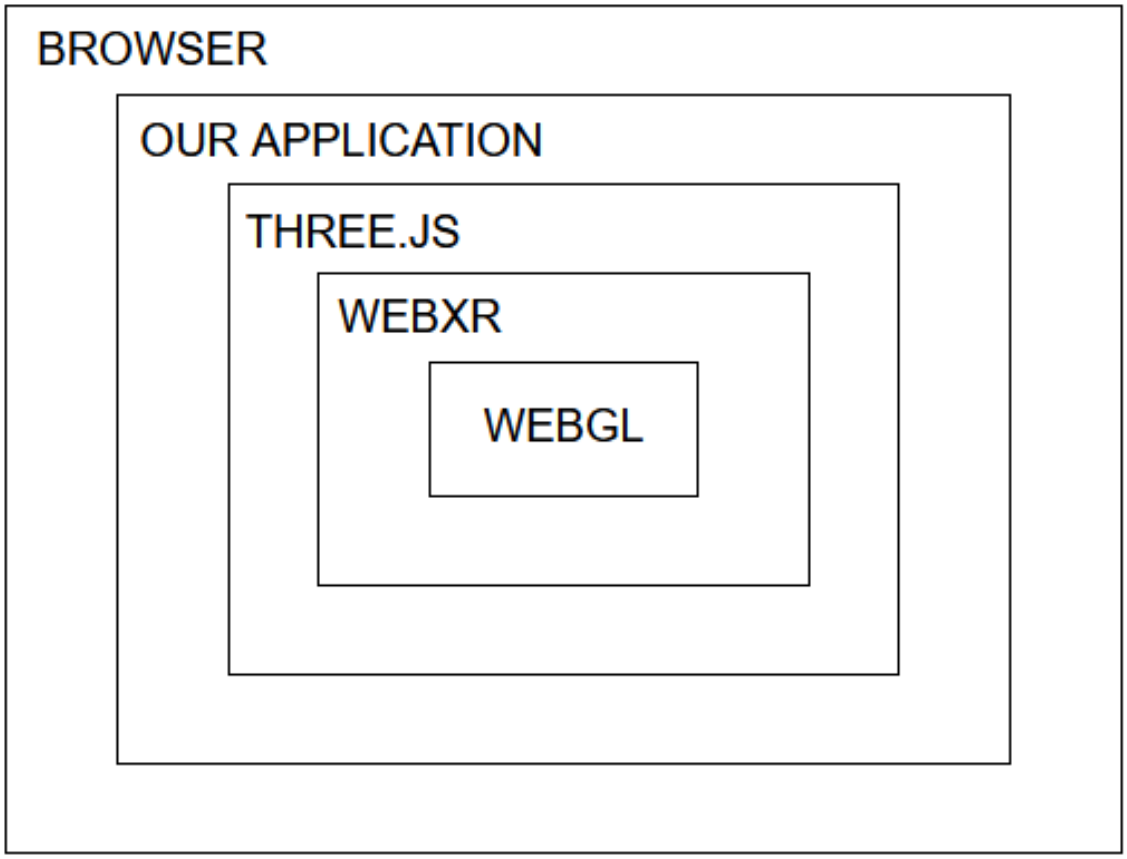

Fig. 17: Layered model description of our application.

5.1.1 The pipeline

We start from a series of models of cultural heritage obtained by laser scanning as well as a collection of photographs of the actual structure. Working directly with this data is not very convenient as it comes distributed among several files, so we will process them to centralize each camera's information. In this step, we will also precompute the information about the relationship of the cameras between them, which will be helpful to us to determine the photos that are relevant to the user.

Once we have processed the information from the cameras, we pass it on to the application. Using the Threejs structures, we create the scene with the model and the cameras, and we also create the collection of photographs from the precomputed information. At this point, we can decide whether to display the scene in the browser or use the WebXR features to display it in a virtual reality environment. Figure 18 can help you to visualize the pipeline.

In order to access the application from the internet, both the application and the assets are hosted on a personal server, fully accessible but password-protected.

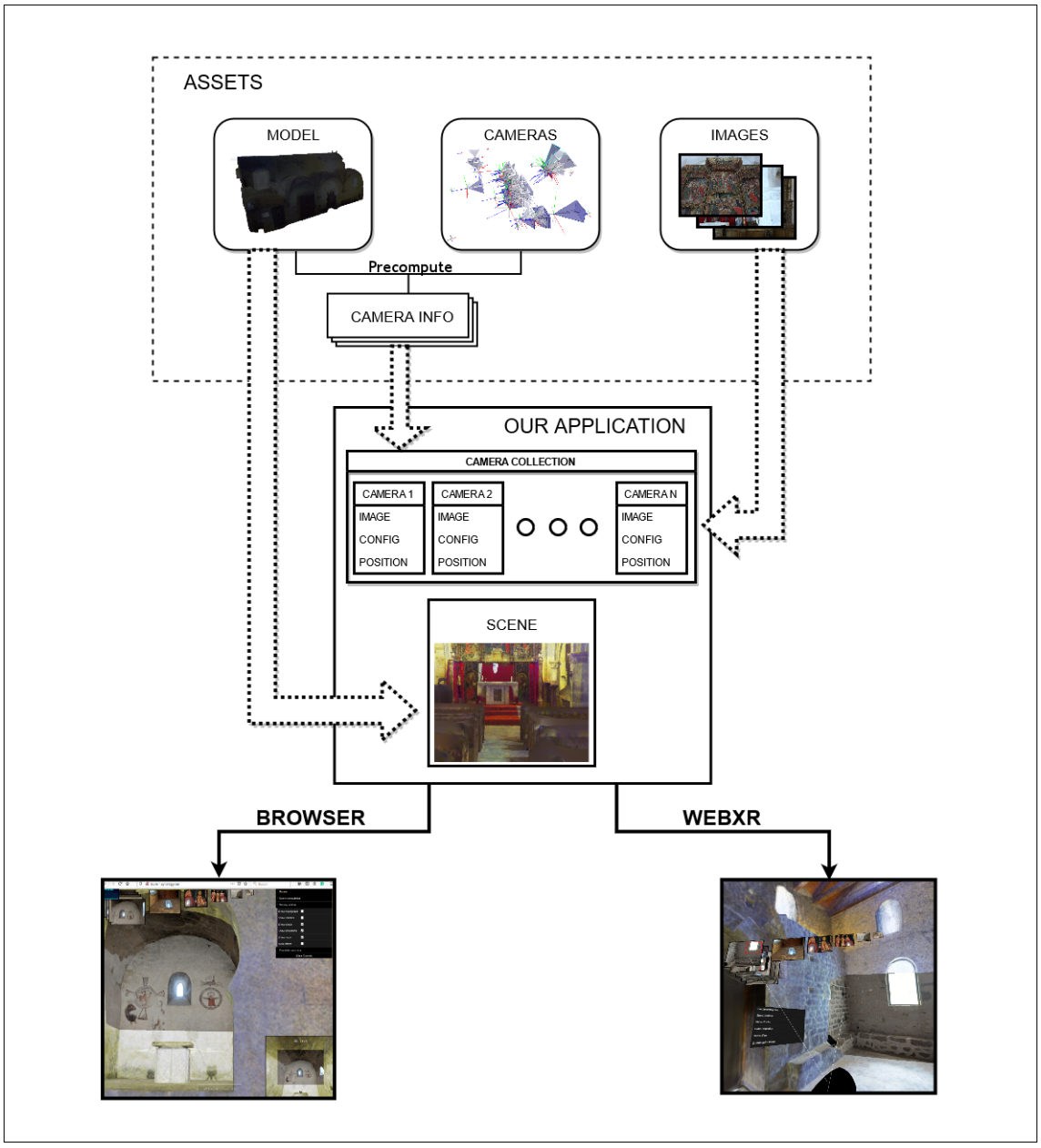

Fig. 18: Simplified pipeline of our application.

5.2 The models

In this section, we will discuss the process of generating the input files of the application. More specifically, we will talk about how we have obtained the models and photographs of the cultural heritage and how we have processed these files to convert them into the data that our application needs.

5.2.1 Obtaining models

All the models that we show in the application have been obtained using laser scanning. This work was carried out by Comino et al [16]. The following steps represent a very brief description of the process:

The first step consisted of traveling to each of the cultural heritage sites to take the 3D scans and the photographs:

- Església de Sant Quirze de Pedret (Berga, Barcelona). Figure 19.
- Museu Diocesà i Comarcal de Solsona (Solsona, Lleida) Figure 20.
- La Doma (La Garriga, Barcelona) Figure 21.

Fig. 19: Pedret

Fig. 20: Solsona

Fig. 21: La Doma

In each of the places, a laser scanner Leyca RTC360 was used to scan the scene. Thanks to this scanner, the point cloud of the model and panoramic photos of the scene were obtained, which will be converted later into cubemaps. Additionally, a set of photographs of the scene are taken to show in the application. Some captures were taken on different days so that we can see some changes in the lighting.

In order to know in which position the photographs are, we follow the following process:

1. We identify image features in each of the faces of different cubemaps constructed from the panoramic images provided by the scanner.
2. For each photography, we compute their image features and match them to the cubemaps' ones.
3. Now, since we have registered the 3D position and camera information of the cubemaps, we can obtain the intrinsic and extrinsic parameters of the cameras using the matched features. This step is called *Structure from motion*.

All this process is carried out through a software called COLMAP [2].
COLMAP is a general-purpose Structure-from-Motion (SfM) and Multi-View Stereo (MVS) pipeline with a graphical and command-line interface. Thanks to the command-line interface, all the above steps can be automated in one script. In Figure 22 you have a simplified representation of this process.

We obtain the triangular mesh from the point cloud by using reconstruction algorithms [24]. In some cases, the generated mesh presents some artifacts that should be fixed. To address this issue, we used another software called MeshLab [3], which not only allows us to fix these artifacts but also permits us to visualize both the model and the cameras.

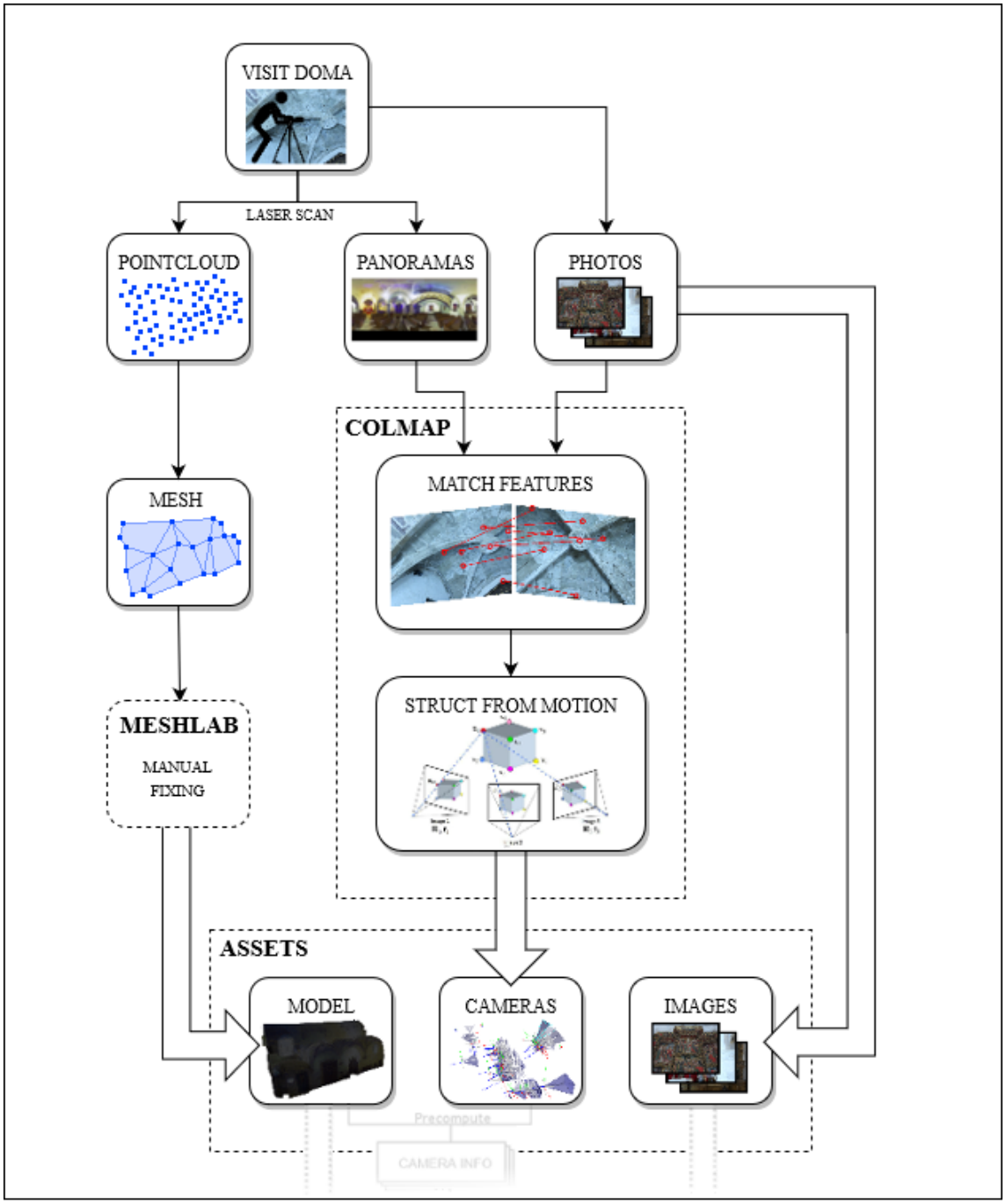

Fig. 22: The pipeline to generate the input of the application.

5.2.2 Processing the output from COLMAP

Once the COLMAP commands have been executed, it returns the mesh in *.ply* format and a set of files containing information about the cameras:

- **cameras.txt**: for each camera, it gives us its identifier, the size in height/width, the focal length, and the principal point. Keep in mind that the same camera *ID* can correspond to multiple photographs.
- **points3D.txt**: this file contains information about the features found in the images. In our case, this data is useless, so it is discarded.

- **images.txt**: This file contains information about the photographs. Each entry includes the camera identifier, the name of the image file, and the rotation and translation of the camera. For each image, it also gives us the 2D coordinates of the found features.
- **bundler.out**: Contains a list including the rotation matrix, the translation vector, and the focal length of each camera. The internal format of bundler files is described in the Bundler User's guide [1].
- **list.txt**: Contains a list with the image filenames in the same order as the entries in the bundler.out file.

As the relevant information is split, working directly with these files is inconvenient. To solve this, we have created a python script that generates a single json file that contains the list with all the relevant information for each image. Specifically, we first iterate the bundler.out file to obtain the position and orientation of each image. For each entry, we discover its image filename by checking in the list.txt file. Now, we can match the filename in the images.txt file to obtain the camera ID of each image that will help us find the intrinsic parameters of each camera in the cameras.txt file.

5.2.3 Precomputating extra information

From this list, we could already represent both models and cameras on the scene. However, to make the corresponding calculations to know which are the relevant photos for the user in a particular moment, it is necessary to know which points of the mesh are visible from each of the cameras. This requires casting multiple rays from each camera.

Considering that the meshes we work with are highly dense, computing those raycasts using Threejs is quite expensive. Furthermore, since both the model and the cameras are static, the output of these raycasts can be precomputed. The process consists of dividing the frustum of each camera into a 10x10 grid. Then we cast rays from each of these points (100 rays in total). All these 3D positions are stored in a file to be accessed from the application when necessary. Check figure 23.

Another piece of information that we are interested in precomputing is the similarity between the cameras. In our case, we understand similarity as the percentage of the image that is shared between two photographs. Since we have already precomputed the mesh points visible in each photo, to calculate the similarity between two photos, we will project each of the points of the first camera on the frustum of another camera and calculate the percentage of points found inside the frustum (notice that we are not taking into account any possible view occlusions). Next, we carry out the same task but this time from the second camera to the first. Finally, we take the average of the two percentages, giving us the similarity between the two. Check figure 24.

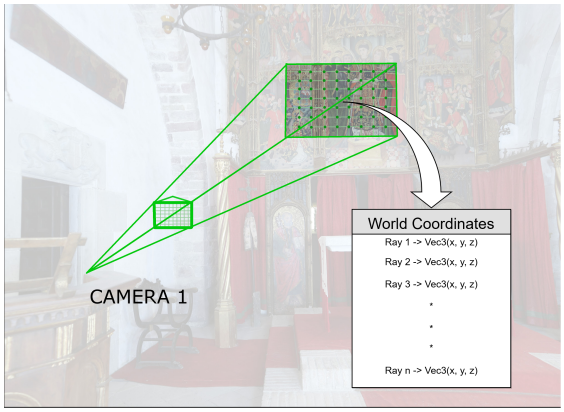

Fig. 23: Diagram representing the precomputation process to obtain the world space position of the ray intersections. The coordinates found at the green dots in the grid are saved in a list to be used later.

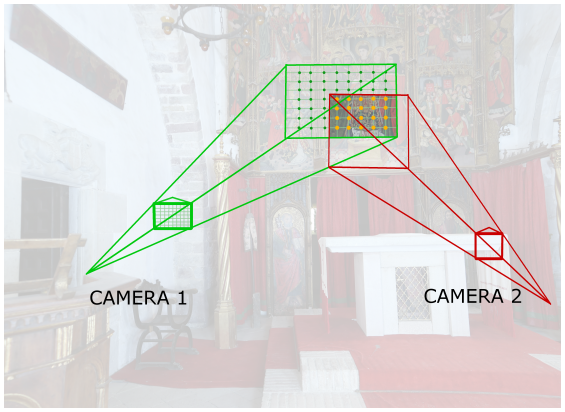

Fig. 24: Diagram representing the precomputation process to obtain the similitude value between two cameras. The orange dots inside the highlighted area represent the rays of the camera 1 that can be seen from the camera 2.

5.3 Preparing the scene

In this section, we will take a look at how the main scene of our application is structured and how the information that we have previously precomputed is processed and represented.

5.3.1 Loading models and cameras

The first thing the application does when it starts up is to call the ***init()*** function responsible for initializing the Three.js *renderer* and creating an empty *scene*. From this point, we move on to loading the cameras and the default model. To do this, we read the file ***cameraInfo.json*** asynchronously and iterate through the list of cameras. For each of them, we create and store in a list an object that we call *CameraInfo*, which contains the following parameters:

- Photo filename
- Width and height
- Transformation matrix
- Focal lenght
- Camera
- Sprite
- Similitudes vector
- Rays intersection vector

Most of the data is obtained from the *JSON* file, such as the name of the file, the camera's intrinsic/extrinsic parameters, and the ray vectors and similarities that contain the values that we have previously precomputed. From this data, we can create a *Camera* that is a Threejs object. This object will be useful to project the scene from the selected camera. On the other hand, the *Sprite* is another Threejs object that consists of a quad with a texture (the image taken with the camera). This object can be added later to the scene to display the image in the photo collection. To reduce the loading time as well as the *RAM* used by the application, we will load a much smaller version of the image instead of the real one.

During the initialization step, we will also create a Threejs *Mesh* to save the model we want to represent (by default *Pedret*). Fortunately, Threejs accepts *.ply* files, so it is not necessary to transform the model. We also load the texture of the model, which is usually quite large and takes up a large part of the loading time.

5.3.2 Main camera and navigation

Now we have created the scene, but there one last element missing: the user's camera.

It will be a perspective camera. The initial position and orientation are decided according to the loaded model. Other parameters such as FOV, near and far have been chosen arbitrarily after experimenting with different settings.

To control the camera, Threejs provides us with a set of predefined controls. One of them is the *OrbitControls* which allows us to move the camera with the mouse. These controls work as follows:
Holding the left mouse button the camera moves around a fixed point to which the camera is facing. With the mouse wheel, we move closer or farther from this point (Figures 25 and 26). We can also do panning, which consists of moving the orbit point and the camera together in the direction of movement.

To improve the user experience, we have rewritten part of the original *OrbitControls* code so that when you zoom in from a certain threshold, the orbit fixed point moves forward. In this way, the user can navigate forward without restrictions.

An important fact to remark is that there are no light sources in the scene. The main reason is that we want to visualize the model and the projections of the images with the original information of the capture.

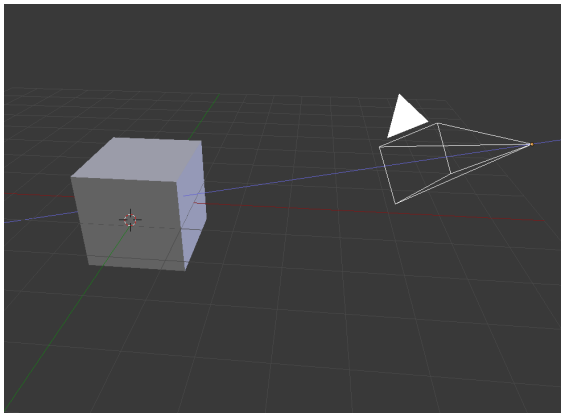

Fig. 25: Representation of the forward movement of the camera using orbit controls. The camera moves along the blue line, aproaching or moving away from the invisible target.

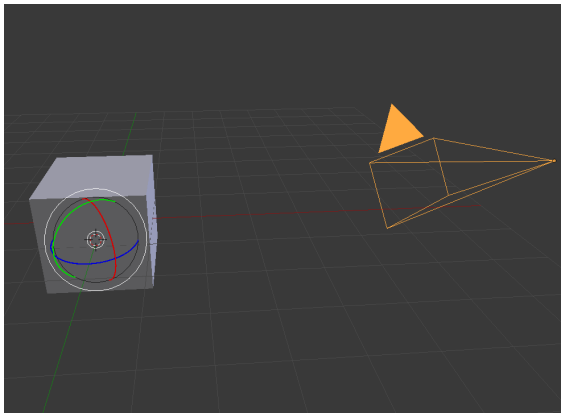

Fig. 26: Representation of the camera rotation while using orbit controls. The camera rotates arround the invisible target. The sphere represents the allowed axes of rotation.

5.4 Photo collection

Now that we have seen how the scene is prepared, we will talk about the implementation of the photo collection shown at the top of the screen. More specifically, we will talk about how the distribution and order of the photographs is decided and how the different customizable parameters are implemented.

5.4.1 Ranking images

When deciding which photos are most relevant to the user, the first step is to sort them according to their *score*. The total score for a photograph is determined by the weighted sum of different secondary scores. Each of these is normalized (between *0* and *1*), and the user decides the weight for each of them.

1. Distance score
This first score consists of measuring the distance between the user's camera and the position in which the capture was taken (Figure 27). The closer they are, the more score, where the value would be maximum when the two cameras are in the same *x, y, z* coordinates, obtaining a score of *1*.

To get the normalized score value from the distance value, during the initialization of the application, we measure the distance between the two most distant cameras. Then we use this value to compute the *score* for our camera, as we can see in the following equation:

$$Capt\_score = max(\frac{Max\_dist}{|Cam\_pos - Capt\_pos|}, 0)$$

2. Orientation score
This time, instead of taking the distance into account, we measure the difference in their orientations (Figure 28). More specifically, we measure the angle formed by the forward vectors of the two cameras. Next, to obtain the normalized value, we divide this value by *180*. An angle of *180* would correspond to two cameras looking in opposite directions, which would result in a score of 0.

3. Shared points score
This last one is the most accurate of them but also the one that requires the most computing time. It consists of counting the number of points of the model visible from the capture that are also visible from the user's camera. If all the points are visible, the maximum score of *1* would be obtained. (figure 29)
The points that are evaluated are those that we have previously precalculated for each camera. We consider that a point is inside the frustrum when the value in the *z-axis* in camera space is smaller than the one of our camera. Plus, after converting its coordinates from *world space* to *NDC*, its value is between *-1* and *1* along the x and y axes.

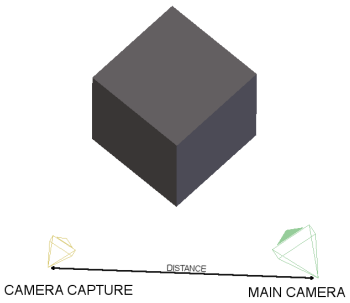

Fig. 27: Score computation using the distance between the main camera and the camera capture.

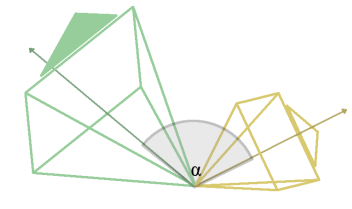

Fig. 28: Score computation using the angle difference between the main camera and the camera capture.

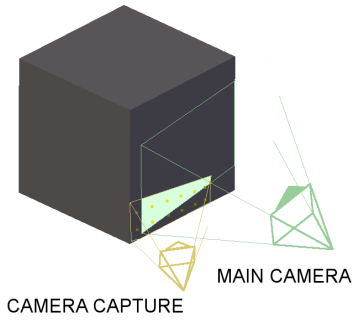

Fig. 29: Score computation using the intersection ray count from the capture camera in the main camera

5.4.2 Automatic image selection

When the automatic suggestion option is enabled, whenever the user moves the camera, the scores of the entire collection are recalculated according to the weights established by the user.

Note that during the camera movement, the application may recalculate all the scores several times, causing performance drops. To avoid this problem, we added a timer to prevent the calculation from taking place in each frame. This timer also restarts every time the position and orientation of the camera have changed. This way, the computation will only be carried out once the user has stopped moving the camera.

The user also can modify the timer waiting time according to his needs. If the weight assigned to the shared points score is 0, the computation is quite agile, so it allows setting a low value to the timer without affecting performance.

5.4.3 Manual image selection

An alternative to automatic selection is manual selection. It consists of selecting an area of the screen and then calculating the scores taking into account the selected area. In this process, only the *shared points score* is taken into account. As we mentioned previously, the score is obtained by counting the number of points visible from the user's camera. To adapt it to manual selection, we will only count the points that enter the selected area. Instead of delimiting between *-1* and *1* in *NDC* coordinates, we will use the offset corresponding to the selected area. This is shown in Figure 30.

To give visual feedback to the user, we use the Threejs SelectionHelper class, which allows us to create selection boxes.

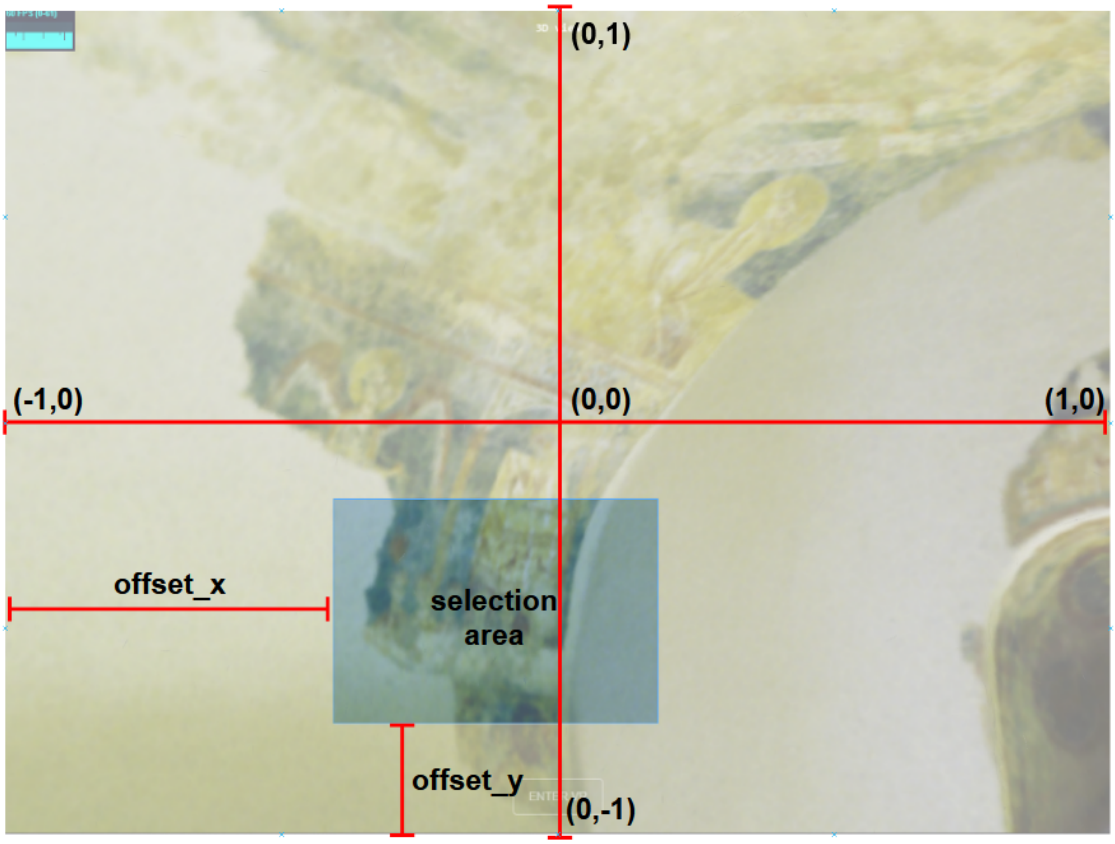

Fig. 30: Representation of the offsets used in the manual image selection mode.

5.4.4 Clustering images

When displaying the images, a common problem is that there is a lot of redundancy. Many images resemble each other. To achieve a collection of images with both variety and relevance, we have made a system of stacks where each stack contains images that are similar to each other. The user can choose between two different clustering algorithms to decide how these stacks are formed. Both of them use the similarity information that we precomputed during the preparation of the input data.

Basic clustering
This method begins by taking the image with the highest score. Then it goes through the ordered list of images, and for each one of them, we add it to the stack if their similarity value is greater than a given threshold (given by the user). If the stack is full, but there are still some similar images remaining, these are discarded.
Once the stack is complete, we move on to the next one using the images that have not yet been assigned or discarded.

The benefit of this method is that it is very fast. Unfortunately, its precision is quite low.

Single linkage clustering
The second method we offer is based on single linkage clustering [4]. The adaptation for our scenario works as follows:

1. Each of the images represents a single cluster. We save the similarity of each pair of clusters in a symmetric table. For now, this value is defined by the pre-computed similarity value.
2. Within all the pairs of clusters, we select the pair of clusters whose similarity is greater, and we join their elements, forming a single cluster.
3. Next, we update the table of similarities. Being a and b the clusters that we put together forming a new cluster (a, b) and the function $f(x)$ that determines the similarity of a cluster x, we eliminate a and b from the table and add (a, b), where the similarity value with each one of the other clusters k would be denoted as $f(a, b) = \min(f(a, k), f(b, k))$
4. We repeat steps 2-3 until we have the desired number of clusters.
5. For each cluster, we order the elements in decreasing order according to their score. We also order the stacks from left to right according to the maximum score.

Our implementation has a time complexity of $O(n^3)$. For our case, it is enough, although there are more efficient implementations of the algorithm.

5.4.5 Visualization

The computed clusters are displayed at the top of the screen from left to right, decreasing size to denote more or less relevance. In the options menu, the user has two customization parameters. One is used to establish the maximum number of clusters shown on the screen, and the other one determines the maximum number of photographs that a cluster can contain.

To display the photographs with *Threejs*, we have used quads with textures. Each photo is a *Threejs Mesh* that contains the geometry of a quad and a material with the corresponding texture. In order to create the black border in the images, we have modified the *fragment shader*, where we paint black if the x, y coordinates in *NDC* are outside the interval $(-1 + a, 1 - a)$ where a is an arbitrarily decided offset.

Instead of adding these objects to the regular scene, we used an auxiliary scene drawn after the 3D scene. In this scene, there are only the photographs, and it is drawn using an orthographic camera whose dimensions are decided and rescaled from the size of the window.

To decide the size and position of each image, we take 1/4 of the screen size. This portion will correspond to the first stack of photographs. For each photograph in the stack, we decide

its size by dividing the total size available by the number of photographs (adding a slight offset to generate the stair effect). For each following stack, we will take 1/4 of the remaining screen size and repeat the same process. This way, the stack size is being lowered along the horizontal axis.

One last point to note about the visualization of the photo collection is that we use thumbnails instead of the real photographs. In this way, we considerably reduce the loading time as well as the *RAM* used.

In the following code snipet we summarize the process to calculate the sizes of the elements in the photo collection. Figure 31 shows all the elements referenced in the code.

Pseudocode 1: Resizing and positioning the images in the photo collection.

```
remaining_width = max_width
start_x_stack = 0
offset_per_image = 10px

foreach(stack in collection)
{
        width_stack = remaining_width / 4
        width_image = width_stack - stack.size() * offset_per_image
        foreach(image in stack)
        {
                image.scale = width_image
                image.x = start_x_stack + offset_per_image * image.index
                image.y = offset_per_image * image.index
        }
        remaining_width = remaining_width * (3 / 4)
        start_x = start_x + width_stack
}
```

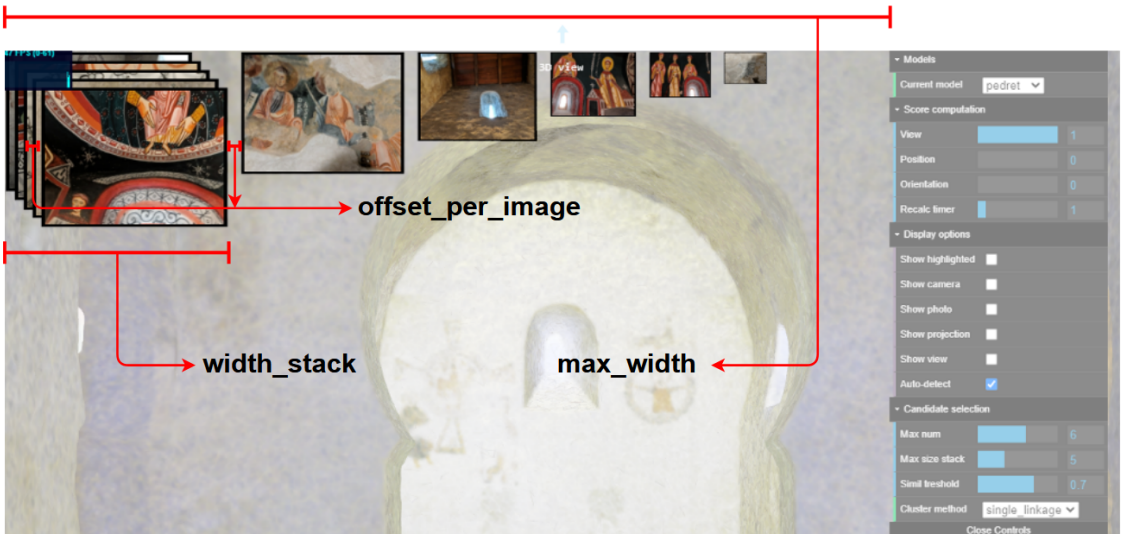

Fig. 31

5.5 Tools

Now that we have explained how navigation and selection work in the photo collection, let's take a look at the implementation of the additional tools. This set of tools work from the images selected by the user and serve to represent the photo information in the 3D scene.

5.5.1 Highlighted area

This tool is used to highlight the area covered by the photograph in the 3D model (Figure 32). To achieve the effect, we have modified the shaders that draw the model. As we mentioned before, Threejs allows you to easily render models based on a 3D mesh and a texture. For this, Threejs uses a default shader which we have rewritten to implement the features we need.

Once the user has selected a photo, we can access the camera's information that has taken the capture (position, orientation, FOV, etc.). In addition, we also have the model view and projection matrix already calculated. What we want to do with this information is to detect which fragments of the scene are visible from the capture camera. The steps to achieve this are the following ones:

1. We use two *uniforms* to send the view and projection matrices of the camera that took the capture to the *vertex shader*.
2. We multiply the view and projection matrices with the scene's model matrix to obtain the model's vertices in screen space starting from the capture camera instead of the user camera.
3. We divide the coordinates obtained by the perspective (component w) and obtain the coordinates x, y, z in *NDC*.
4. We calculate the *gl_Position* by multiplying the matrices of the user's camera.
5. We send the *NDC* coordinates of the capture to the *fragment shader*.
6. In the fragment shader, we evaluate the calculated coordinates. If its x, y values are between -1 and 1 and $z > 0$, the fragment evaluated in the user's camera is within the frustum of the capture camera. We also detect the fragments on the edge of the image by subtracting a small offset to the -1, 1 range.
7. For the fragments that are within the range, we use the glsl *mix* function to mix the color of the texel corresponding to the fragment with a red color $vec3(1, 0, 0)$. For the fragments that are on the border, we proceed in the same way, but we give more weight to the red color in the mix function. Finally, the fragments out of range are drawn in the usual way using only the texture color. Check the result in Figure 32.

Pseudocode 2: Fragment shader modification to display the red area.

```
uniform bool showRedArea;    //Boolean that controls if the option is enabled.
varying vec2 vUv;            //Texture coordinates.
varying vec3 capturePos;     //Position of the fragment in NDC (secondary camera).
...
if(showRedArea
    && capturePos.x > -1.0 && capturePos.x < 1.0
    && capturePos.y > -1.0 && capturePos.y < 1.0
    && capturePos.z > -1.0 && capturePos.z < 1.0)
{
    if(capturePos.x < -0.95 || capturePos.x > 0.95
    || capturePos.y < -0.95 || capturePos.y > 0.95)
        gl_FragColor = vec4(mix(vec3(1.0,0.0,0.0), texture2D(texture1, vUv).xyz, 0.6), 1.0);
    else
        gl_FragColor = vec4(mix(vec3(1.0,0.0,0.0), texture2D(texture1, vUv).xyz, 0.8), 1.0);
}
else
    gl_FragColor = texture2D(texture1, vUv);
...
```

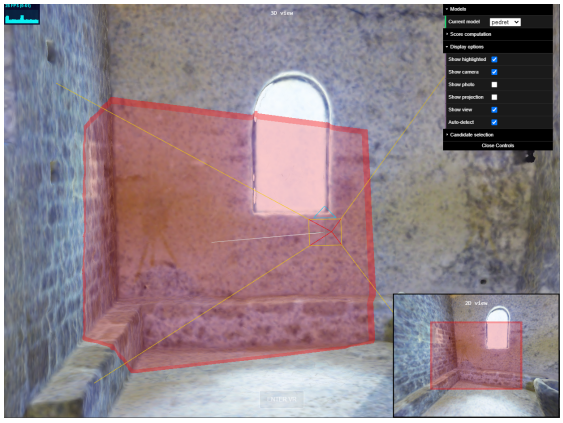

Fig. 32: The area covered by the photo is displayed by a transparent red projection

5.5.2 Image projection

A very similar tool to the previous one is photo projection. Instead of highlighting a red area on the model, we project the image on top of the mesh. The procedure is the same as before, except that on this occasion, during the *fragment shader* phase, we will map the *NDC* coordinates of the camera to texture coordinates. The texture coordinates have a range between 0 and 1, so they need to be converted from *NDC* is as follows:

$$texCoord = 1 + NDCcoord/2$$

Notice that we are not considering the real depth that each texel would have in the 3D object. This means that the same texel is drawn in each of the points that coincide in the same screen space coordinate of the capture camera. This can generate artifacts like the ones we can see in Figure 33.

This issue could be fixed by using an auxiliary draw pass to fill the depth buffer with the depth information. Still, we did not consider it worthwhile since the current result is acceptable enough. Also, this kind of improvement would lead to a decrease in performance and other types of artifacts that would have to be managed.

Fig. 33: This figure shows the depth artifact, the texture corresponding to the top of the altar is replicated on the floor and walls.

5.5.3 Show camera

This last tool is used to show the camera of the selected photograph in the 3D scene. For this, we use a Threejs object called CameraHelper. It requires a camera object which we already have, then will automatically create a representative model of the camera frustum.

Additionally, if we also enabled the show photograph option, we can see the photo placed in front of the frustum, as shown in Figure 34.

Fig. 34: This figure shows the camera tool, we can also see the photo in the near plane.

5.6 Secondary view

We are now going to talk about the secondary view. By default, this view is located in the lower right corner of the screen, in front of the rendered scene. We achieved this by limiting the size of the viewport to an arbitrary percentage of the screen size. Also, to differentiate the secondary view from the scene, we have added a small border. We achieve this effect by first making a clear $(0, 0, 0)$ in the viewport and then painting the scene but with a smaller viewport size.

5.6.1 Rendering the view

Instead of drawing the scene directly on the screen, we will use a different render target. Once drawn, we take the generated texture and put it on a Threejs *Mesh* (quad). Next, we add this object to a new scene with an orthographic camera whose size and position are chosen to fit the object. Finally, we draw the quad from the orthographic camera inside the viewport that we mentioned earlier.

5.6.2 Showing the image

In the tools menu, we can activate the option "*show photo*" (not to be confused with "*show projection*"). This option displays the selected photograph above the scene in the secondary view. Since the camera's configuration that projects the scene is the same as that of the camera that took the capture, the displayed image fits perfectly with the scene.

In order to show this image, we create a new *Mesh* corresponding to a quad with the photo as texture. The next step is to position this model in front of the camera in such a way that it fits with the scene. To achieve that, we carry out the following steps:

1. We apply the transformation matrix of the selected camera to position and orient the object with the camera.
2. We shift n units in the z vector in the camera's *eyespace*, where the value of n corresponds to the $z - near$.
3. Once the object is in front of the camera's vision frustum, we scale it in its x and y components to match the edges of the frustum in the *near* plane. To obtain the scaling values, we use the following formula:

```
var frustumHeight = 2 * camera_capture.near * Math.tan(camera_capture.fov * 0.5);
var frustumWidth = frustumHeight * camera_capture.aspect;
```

Notice that we also increase the FOV of the camera to add context around the photo. Figure 35 gives a visual representation of the previous steps.

One issue we encountered during the implementation of this feature is that the loading time of the photograph was very high. To solve this problem, we draw the thumbnail while we wait for the real photo to finish loading.

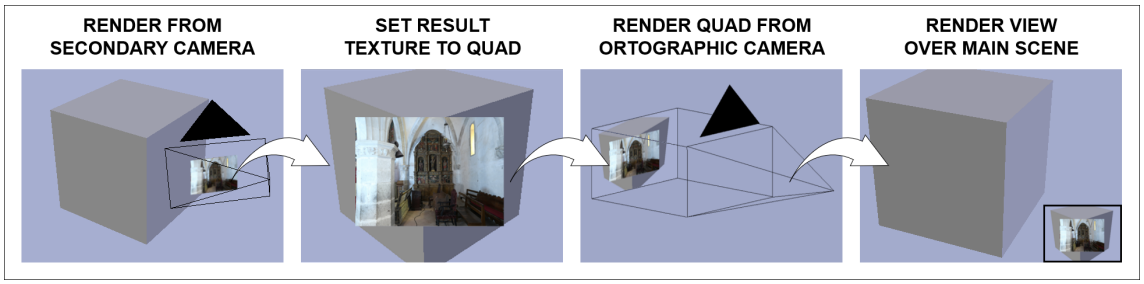

Fig. 35: Pipeline to render the secondary view.

5.6.3 Navigation

If we click on the secondary view, it will be shown in fullscreen, leaving the main view as the secondary in the left corner. For this, the only thing needed is to swap the rendering order and exchange the viewport sizes.

The reason we draw the scene on a plane and use an orthographic camera is to allow navigation of the secondary view in fullscreen mode. The main utility of this view is to allow the user to examine in detail the photographs and their context in the scene. Since the photo is superposed in front of the scene instead of being projected, if we used a perspective camera to navigate in this view, when moving the camera, the perspective would change and the image would no longer match the scene. If, on the other hand, we used an orthographic camera, the perspective of the photo would differ from the scene, so it would not match either.
For this reason to allow navigation, first, we draw the scene and the photo, respecting the perspective of the camera that took the capture, and once the scene is projected on a plane, we can navigate through an orthographic camera in front of that plane without risk to create a discordance between the photograph and the scene.

Note that in order not to lose quality in the shown image, the size of the target render is set to $sizeImage + offset$, where $sizeImage$ represents the actual size of the photograph and $offset$ corresponds to the part of the scene that stands out from the photograph. To calculate this last value, we calculate how much space it occupies in proportion within the near plane and then multiply this value by the size of the image.

Figures 36 and 37 show how the camera movement works in the photo collection scene with the orthographic camera.

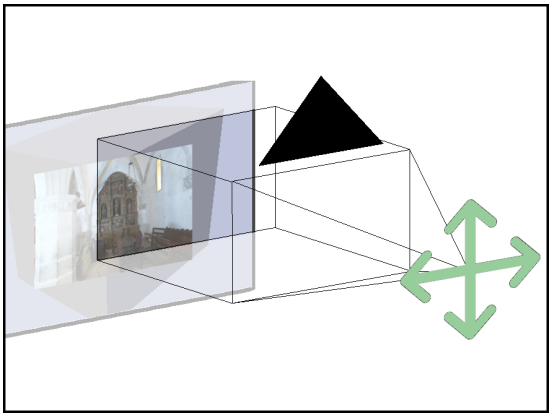

Fig. 36: Graphic representation of the grab navigation.

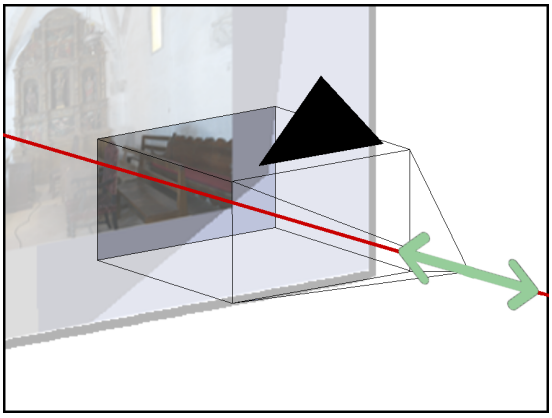

Fig. 37: Graphic representation of the zoom navigation.

5.7 VR mode

In this section, we will talk about the changes made to adapt the application to virtual reality. To allow VR in Threejs, it is enough to activate the *xrEnabled* parameter in the initialization of the renderer. We also add a button to decide whether or not to enter virtual reality. Once inside, when we call the renderer, instead of using the standard camera, we use a pair of cameras already prepared to follow the position and orientation of the headset. Threejs makes part of the work easier for us by automatically initializing and controlling the two VR cameras from our initial camera.

While it is true that Threejs allows you to view the scene on a VR headset directly, it was necessary to make some design changes to preserve visual coherence.

5.7.1 Navigation

Previously we used the Threejs *orbitControls* object to navigate within the scene. With the mouse movement, we controlled the orientation, and with the zoom, we could move forward or backward in the direction pointed by the camera.

In VR, the position and orientation of the camera are updated through the Headset sensors. *Threejs* decides the height at which the camera is located by mapping the actual height of the headset to the world space coordinates. This automatic assignment works quite well when the mesh of the 3D model corresponding to the ground coincides with the 0 coordinate on the vertical axis, but this may lead to some invalid positions since there are some models that contain sections with different heights. A good example is *Pedret*, where it is necessary to go upstairs to access the side section.

To solve this problem, every time we move to a new position in VR, we cast a vertical ray from under the model. Since none of our models has two or more stacked floors, the first intersection of the ray with the mesh will correspond to the new position on the ground.

Exploring the scene in VR:
There are different methods for getting around in virtual reality. The first we considered was the use of joysticks to move continuously. However, it is well known that this method often causes motion sickness. Some techniques, such as using a variable *FOV*, proved to be successful in reducing this effect [20]. In our case, given the time constraint, we have opted for the use of teleportation [12]. This method is comfortable to use, easy to implement, and avoids the problems of motion sickness.

Choosing the teleport target:
In order to decide the teleport target, we need to detect the inputs of the virtual controllers. *Threejs* also facilitates part of the work by offering us an object that automatically synchronizes its position and orientation with the controller in the real world. In addition, the controller can be seen within the scene since it has a *Mesh* as well as a ray that indicates to the user where it is pointing. This object comes with a list of events that can be detected, such as connection/disconnection, movement, and selection (main trigger). This last event is the one that we are going to use to carry out our teleportation.

The most intuitive way to implement this method would be to cast a ray in the direction pointed by the controller and move the camera in the x and z coordinates where the ray intersects with the model. However, this method is not very comfortable for the user and often forces them to look continuously towards the ground. Other techniques are based on curved trajectories [21], which are more intuitive for the user.

Our approach is based on the use of a parabola (Figure 38) which is obtained by the following formula:

Pseudocode 3: Geting the target position using a parabola.

```
...
//Cursor position
const p = handController.getWorldPosition()
const v = handController.getWorldDirection()
//Parabola equation
const t = (-v.y  + Math.sqrt(v.y**2 - 2*p.y*gravity))/gravity
var cursorPos = new Threejs.Vector3(0,0,0)
cursorPos.addScaledVector(v,t)
cursorPos.addScaledVector(gravity,0.5**t**2)
return cursorPos
...
```

The formula allows us to obtain points on the trajectory of the parabola, which we use to draw a *Threejs* line in the scene. Additionally, we have added a 3D *Mesh* (arrow shape) at the point where the parabola reaches the ground to indicate in which exact position the camera will move.

When the user holds down the trigger of the right controller, we capture the event and display the parabola with the arrow in the scene. The shape and position are updated every frame while the user is holding the trigger. Once the user has decided which position he wants to teleport, he releases the trigger, and the camera is moved to the new coordinates. Check the result in Figure 39.

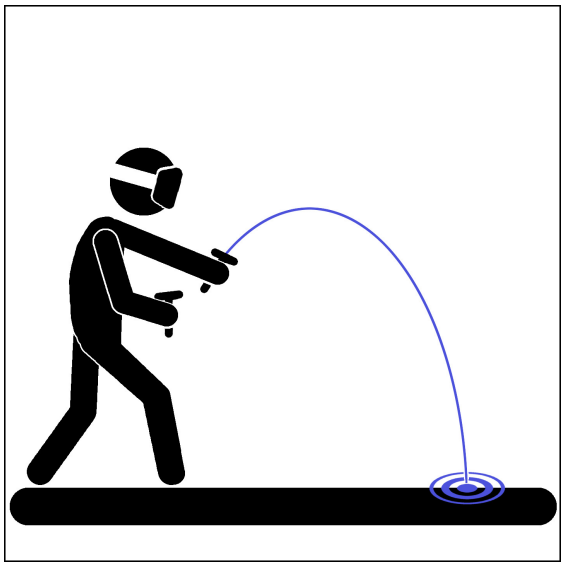

Fig. 38: Graphic representation of the teleporting feature using curve trajectories.

Fig. 39: User using the teleport tool in the VR mode.

5.7.2 UI

Previously for the options menu, we used the Threejs *GUI* class. This allowed us to create checkboxes, dropdown menus, and drag bars. Unfortunately, this menu was not an element inside the scene, instead it was an *HTML* element that we cannot see in the VR environment. However, thanks to the *HTMLmesh* class, we can convert this element into a Mesh that we can add inside the scene. In addition, this class allows us also to capture the selection events of the VR controllers, making this *Mesh* an interactive UI within the scene. Unfortunately, some elements such as dropdown lists did not work quite well this way, so they were removed from the interface or adapted to other types of UI elements.

At this point, we only need to choose where and how we position the UI inside the scene. To be able to access the UI at any time, the most logical implementation is to follow the user position. To do this, we create a *Threejs Group* object. This object allows us to join multiple objects as children of the group, which we can transform as we want, and the elements that compose it will follow the same transformations. In this case, what we will do is a group that we will call camera_group, which will contain the UI, the camera, the photo collection, and the secondary view.

The local position of the elements in this group has been chosen arbitrarily after experimentation with different configurations. At the end, we placed the UI on the left side with a slight rotation on the vertical axis so that it is facing the user. To prevent the interface from being too intrusive, it comes disabled by default, and it is only activated while the left trigger is held down.

We discovered while experimenting with the UI that if it followed the orientation of the headset at all times, it was very difficult to interact with the elements. Hence we decided that each time the user held down the trigger to activate the UI, it would be positioned according to the position and orientation of the camera, but then it becomes a static element until we deactivate it. Figure 40.

Fig. 40: UI displayed in the VR mode.

5.7.3 Photo collection

Remember that when we drew the photo collection, the *Meshes* that contained the images were set on a different scene. To adapt this to virtual reality, the *Meshes* will have to be visible in the main scene. To achieve this, we will add each one of them to a new *Threejs Group*. Inside that group, we position the images by the same procedure used in the normal mode with the difference that in each stack, we will add a little offset between the images on the *z-axis* to provide a sensation of volume in the scene. Next, we scale the group size by an arbitrary number so that it occupies the desired size along with the scene. Finally, we add the group inside the *camera_group*. This way, when we activate the image collection, it will be positioned in front of the camera. Notice how the photo collection is displayed in VR in Figure 41.

In order to detect the photograph that we want to select, we use the same system that we have used with the tools menu. We cast a ray from the controller and check if it intersects with any of the images. Once selected, if we press the trigger, it will move us to the capture position. If we want to scroll in one of the stacks, we only need to press the side trigger.

Fig. 41: Displaying the photo collection in VR.

Area selection
In VR mode, we also wanted to integrate the area selection mode. Remember that in normal mode, it was as simple as selecting an area on the screen and then calculating its coordinates in screen space. These coordinates were used to delimit the points that we counted inside that frustum fraction of the camera. To achieve the same effect in VR, the process we followed these steps:

1. When the user presses the area selection button (side trigger), we launch a ray from the controller towards the model. We keep the intersection point for later.
2. While the user keeps the button pressed, we cast rays to know the second selection point.
3. We project the two points to *camera-space* and pass it to the *fragment shader* that draws the model.
4. In the *fragment shader*, we check if the fragment stays within the range between the two points. If so, we mix the fragment's color with a $vec3(0, 0, 1)$ (blue color). Check Figure 42
5. When the user releases the trigger, we calculate the points inside the area from the last *camera-space* coordinates we have, just as we did in normal mode.

Fig. 42: Using the selection area in VR mode.

5.7.4 Secondary view

This element was easier to adapt since we already had the view drawn in a Threejs *Mesh*. Therefore, the only thing we had to do was introduce the element into the camera_group and decide the local position so that it would be comfortable to visualize. In this case, we have chosen the camera's right side with a slight rotation in the vertical axis to face the user. We consider unnecessary the additional zoom/drag controls in VR mode since the same user can move closer to or away from the panel by moving the headset. At the right side of Figure 43 we can appreciate the big plane showing the secondary view.

One problem we encountered was that interface elements sometimes were occluded by the Scene Mesh. To solve this issue, we opted to draw these elements without taking into account the depth buffer.

Fig. 43: Showing the secondary view in VR mode.

6 Pilot study

In this section, we will talk about the study carried out to verify the usability of our application. The study's main objective is to determine if the application is useful and easy to use to explore cultural heritage environments. To do this, we have developed a five-stage experiment. First, the participants will learn to use the application in web mode. In the second stage, they will be asked to perform some tasks. In the 3rd and 4th stages, the same process is repeated, but this time for virtual reality. Finally, in the last stage, the participants will answer a questionnaire.

Five people voluntarily participated in the study. Their ages were between 20 and 70 years old. Most of the participants had little or no experience in VR and navigating 3D environments. This last fact is important as it helps determine whether the application is accessible for inexperienced users to learn.

6.1 Experiment description

6.1.1 Training

As we have said previously, users follow a five-stage process. The first one consists of brief guided training for navigation the scene using the different tools. Once the training is completed, the user has about 2-3 minutes to freely familiarize himself with the environment. To be as objective as possible, we will use a specific model for the learning stage (*Pedret*), reserving the other two for the tasks. This way, we prevent the user from obtaining key information to perform the tasks during the learning phase.

6.1.2 Tasks

For the tasks, we will use two variables. The first is the task itself, and the second is the recommendation system used for it:

- **Automatic system**: The auto-detect option is enabled. The user has to solve the task starting from the images shown in the collection of photos recommended in this way.
- **Manual system**: The user can only use the area selection tool to generate the collection of photos.

Since the number of participants is relatively small, we have decided that the experimentation follows a *within-subjects* design. In our case, this means that all participants perform all tasks using both recommendation systems. Since the same task is solved twice, the second time it is done, it is easier for the user regardless of the recommendation system used. To counterbalance this fact conditions the study, we will randomize the order of the recommendation system used.

We have followed the following criteria to decide what tasks should be performed to give us useful information:

- The tasks are easy to understand and perform, enough for all users to complete them all without help.
- The result after completing the task is always correct. We do not ask answers that could invalidate the result due to ambiguity or misinterpretation.
- It is necessary to examine the collection of photographs to complete the tasks.

The following list contains the different tasks used in the study:

- **Doma**: navigate to the left-wing and read the inscription above the crucifixion of Christ.
- **Doma**: navigate to the altar and discover what elements hold the hands of the central figure in the mural.
- **Solsona**: Examine the front mural and discover what musical instruments are being played.
- **Solsona**: Examine the arch and discover the characters drawn on the inside face.

Doma tasks will be performed in web mode and Solsona's tasks in VR mode. Since we used different models for each mode, experimentation in web mode does not condition the performance of tasks in VR mode.

6.1.3 Questionnaire

Finally, the user is asked to answer a questionnaire of 17 questions. Mainly, we ask the participant to indicate which method considers most effective for each of the tasks and modes. We also ask them to evaluate more general characteristics such as the application's usefulness, the level of immersion and usability. In the Annex 7.1 section, you can find the exact procedure of each stage and the complete list of questions of the questionnaire.

6.2 Analysis of the results

6.2.1 Questionarie

Previous experience:
Half of the participants had previous experience in 3D navigation, either in video games or in other virtual museums. As far as virtual reality is concerned, most participants had never used a headset before, and those who had used it had minimal experience. Still, no one got dizzy during the VR experience.

Completion of tasks:
Unfortunately, most users needed some additional help to complete the tasks. All users understood the task they had to perform, but sometimes they forgot how to use any of the tools, or they got lost in the scene. Regarding the different tools to complete the tasks, there is no clear consensus on which tool is better. Some participants felt more comfortable with the automatic mode, and others preferred the manual.

App evaluation:
The majority of participants considered that the photos displayed in the collection of photographs were valuable and relevant. Regarding the application's usefulness in virtual museums, everyone considered it a helpful tool, being the web mode, the most practical to carry out the tasks, and the VR mode the most immersive.

In case you want more information, in the Figure 44, you can find the complete table with the results of the study.

6.2.2 Observations

The training phase lasted longer than expected:
It was quite difficult for the participants to learn to move around comfortably, especially for those who had no previous experience in 3D environments. On some occasions, they teleported using the selected photograph ending completely disoriented. On the other hand, sometimes they were clear about where they were but found it difficult to go or position themselves in a specific place. Regarding the different tools (photo collection, secondary view, selection), they also had difficulties assimilating and remembering how to use each of these features.

The interface was a bit confusing:
To select a photo, you just need to move the mouse over it in the collection. Due to this, sometimes the participants moved the mouse over one of the images inadvertently, causing the projected image to change, which was disconcerting. Another common problem was that during the automatic mode, sometimes the participant had found the desired photos. However, after zooming or moving the camera a little to take a closer look, the collection was unintentionally recalculated again as they left the automatic option enabled.

Participants did not use the tools as expected
Most of the time, the participants preferred to move repeatedly in the scene even though they already had the correct photograph on hand in the collection. They trusted that the photo they were looking for would be automatically projected. They found the secondary view a bit confusing

so they tried to complete the task without using it. Also, they hardly ever explored the stacked photos. Only after spending a lot of time experimenting with the application, participants ended up making good use of the tools.

6.2.3 Comments

Regarding the two methods to perform the selection of images (automatic and manual), the participants agreed that both were effective and that they complemented each other depending on the situation. Another aspect that the participants agreed on was that they were pretty annoyed that the base model was so blurry. They consider that with a little more quality, the application would improve a lot and it would also be easier to orient in the scene. Ultimately most of them agreed that the tools could be more intuitive.

7 Conclusions

In this master's thesis, we have developed an application for the exploration of cultural heritage models. The application offers the user a collection of high-quality photographs which can be used to move around the scene or examine more closely. In addition, thanks to the virtual reality mode, the user can explore the model in an immersive environment.

In this report, we have described the application development process, starting from obtaining of the scanned models using photogrammetry, followed by the implementation of the tools and visualization of the scene. Finally, we have conducted a small study to verify the effectiveness of the application.

We believe that we have achieved our goal and that the application proved to be helpful for exploring models of cultural heritage. Even so, we believe that there are still many features that could be improved in the future to give the user a better experience.

7.1 Future work

In the development of the application, we have been able to implement all the features that we set as objectives. Even so, due to the limitation of time, the application lacks a bit of polishing to be able to be presented as a definitive product. There are lots of enhancements that could improve the user experience.

Interface
As we have seen in the study, it has some limitations and is still not intuitive for users. A change that could substantially improve this aspect would be to change the photo selection system. Instead of selecting the photo automatically by hovering over, we could select it only when clicking on it. Once selected, it could be highlighted with a different color in the photo collection. This way, the user knows at every moment which photograph of the collection is working on. The selected photo would also be shown in the secondary view, and the options chosen in the tools menu would be applied to it. As the selection now replaces the teleport option, we could add a "teleport to photo" button in the tool menu to travel to the capture position.

Photo recomendation
Regarding the collection of photos, in some circumstances, the proposed photos did not make much sense. The main reason is because the photographs corresponding to areas obstructed by walls were not discarded. Fixing this problem implies more ray casts, which ends with a significant performance drop as the raycast implementation in Ihreejs is inefficient for large meshes. A good way to solve this could be to using a simplified mesh only for the raycast calls.

Another reason is that the method used to decide whether two photos are "similar" could be further elaborated. Currently, we calculate the average of the ray casts that coincide in both photographs. Although this is good enough for our purpouse, surely it could be combined with other methods to improve the results.

Acknowledgments

The digitization of the St. Quirze de Pedret models was partially funded by the Spanish Ministry of Economy and Competitiveness and FEDER Grants TIN2017-88515-C2-1-R, the Romanesque Pyrenees, Space of Artistic Confluences II (PRECA II) project (HAR2017-84451-P, Universitat de Barcelona) and the JPICH-0127 EU project Enhancement of Heritage Experiences: the Middle Ages. Digital Layered Models of Architecture and Mural Paintings over Time (EHEM). We would like to thank the Museu Diocesà i Comarcal de Solsona, Carles Freixes, Lídia Fàbregas, for kindly allowing ViRVIG to scan Pedret's mural paintings at MDCS. We would also like to thank the Ajuntament de la Garriga and Enric Costa for kindly allowing Marc Comino to scan the Doma church.

References

[1] Bundler v0.4 user's manual. https://www.cs.cornell.edu/~snavely/bundler/bundler-v0.4-manual.html. Accessed: 2021-06-21.

[2] Colmap. https://colmap.github.io/. Accessed: 2021-05-09.

[3] Meshlab. https://www.meshlab.net/. Accessed: 2021-05-09.

[4] Single linkage clustering. https://scikit-learn.org/stable/modules/clustering.html#hierarchical-clustering/. Accessed: 2021-05-16.

[5] Three.js. https://threejs.org/. Accessed: 2021-05-07.

[6] Webgl. https://www.khronos.org/webgl/. Accessed: 2021-05-07.

[7] Webvr api. https://developer.mozilla.org/en-US/docs/Web/API/WebVR_API. Accessed: 2021-04-17.

[8] Webxr api. https://https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API. Accessed: 2021-04-17.

[9] Lucia Arbace, Elisabetta Sonnino, Marco Callieri, Matteo Dellepiane, Matteo Fabbri, Antonio Iaccarino Idelson, and Roberto Scopigno. Innovative uses of 3d digital technologies to assist the restoration of a fragmented terracotta statue. *Journal of Cultural Heritage*, 14(4):332–345, 2013.

[10] Lucia Arbace, Elisabetta Sonnino, Marco Callieri, Matteo Dellepiane, Matteo Fabbri, Antonio Iaccarino Idelson, and Roberto Scopigno. Innovative uses of 3d digital technologies to assist the restoration of a fragmented terracotta statue. *Journal of Cultural Heritage*, 14(4):332–345, 2013.

[11] David Arnold. Computer graphics and cultural heritage: From one-way inspiration to symbiosis, part 1. *IEEE computer graphics and applications*, 34(3):76–86, 2014.

[12] Evren Bozgeyikli, Andrew Raij, Srinivas Katkoori, and Rajiv Dubey. Point & teleport locomotion technique for virtual reality. In *Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play*, pages 205–216, 2016.

[13] Paolo Brivio, Luca Benedetti, Marco Tarini, Federico Ponchio, Paolo Cignoni, and Roberto Scopigno. Photocloud: Interactive remote exploration of joint 2d and 3d datasets. *IEEE computer graphics and applications*, 33(2):86–96, 2012.

[14] K Choromański, J Łobodecki, K Puchała, and W Ostrowski. Development of virtual reality application for cultural heritage visualization from multi-source 3d data. *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, 2019.

[15] Dimo Chotrov and Angel Bachvarov. A flexible framework for web-based virtual reality presentation of cultural heritage. In *AIP Conference Proceedings*, volume 2333, page 140002. AIP Publishing LLC, 2021.

[16] Marc Comino, Antoni Chica, and Carlos Andujar. Easy Authoring of Image-Supported Short Stories for 3D Scanned Cultural Heritage. In Michela Spagnuolo and Francisco Javier Melero, editors, *Eurographics Workshop on Graphics and Cultural Heritage*. The Eurographics Association, 2020.

[17] George L. Cowgill. Computer applications in archaeology. page 331–337, 1967.

[18] Annabelle Davis, David Belton, Petra Helmholz, Paul Bourke, and Jo McDonald. Pilbara rock art: laser scanning, photogrammetry and 3d photographic reconstruction as heritage management tools. *Heritage Science*, 5(1):1–16, 2017.

[19] Francesco Fassi, Luigi Fregonese, Sebastiano Ackermann, and Vincenzo De Troia. Comparison between laser scanning and automated 3d modelling techniques to reconstruct complex and extensive cultural heritage areas. *International archives of the photogrammetry, remote sensing and spatial information sciences*, 5:W1, 2013.

[20] Ajoy S Fernandes and Steven K. Feiner. Combating vr sickness through subtle dynamic field-of-view modification. In *2016 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 201–210, 2016.

[21] Markus Funk, Florian Müller, Marco Fendrich, Megan Shene, Moritz Kolvenbach, Niclas Dobbertin, Sebastian Günther, and Max Mühlhäuser. Assessing the accuracy of point & teleport locomotion with orientation indication for virtual reality using curved trajectories. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[22] Athanasios Gaitatzes, Dimitrios Christopoulos, and Maria Roussou. Reviving the past: cultural heritage meets virtual reality. In *Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage*, pages 103–110, 2001.

[23] Jacek Jankowski and Martin Hachet. Advances in interaction with 3d environments. In *Computer Graphics Forum*, volume 34, pages 152–190. Wiley Online Library, 2015.

[24] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstruction. *ACM Trans. Graph.*, 32(3), July 2013.

[25] Mila Koeva, Mila Luleva, and Plamen Maldjanski. Integrating spherical panoramas and maps for visualization of cultural heritage objects using virtual reality technology. *Sensors*, 17(4):829, 2017.

[26] Pedro Martín Lerones, José Llamas, Jaime Gómez-García-Bermejo, Eduardo Zalama, and Jesús Castillo Oli. Using 3d digital models for the virtual restoration of polychrome in interesting cultural sites. *Journal of Cultural Heritage*, 15(2):196–198, 2014.

[27] G. Pavlidis, A. Koutsoudis, F. Arnaoutoglou, V. Tsioukas, and C. Chamzas. Methods for 3D digitization of Cultural Heritage. *J. Cult. Herit*, 8(1):93–98, 2007.

[28] Noah Snavely, Steven M Seitz, and Richard Szeliski. Photo tourism: exploring photo collections in 3d. In *ACM siggraph 2006 papers*, pages 835–846. 2006.

[29] N. Yastikli. Documentation of cultural heritage using digital photogrammetry and laser scanning. *J. Cult. Herit*, 8(4):423–427, 2007.

[30] Naci Yastikli. Documentation of cultural heritage using digital photogrammetry and laser scanning. *Journal of Cultural heritage*, 8(4):423–427, 2007.

Annex

Web tasks:

1. In the *Doma* model, if you navigate to the left-wing and, you will find a crucified Christ. Can you read the inscription at the top of this?
2. In the *Doma* model, navigate to the altar and look at the central illustration of the mural. Can you see what the central figure in the illustration is holding in each hand?

VR tasks:

1. In the *Solsona* model, take a look at the mural on the front wall of the room. There are some musicians, discover what type of instrument they are using.
2. In the *Solsona* model, take a look at the lower vignette of the inner part of the central arch. There is a humanoid figure and some letters next to it, can you read them?

Modes:

1. Using the automatic recommendation system.
2. Using the recommendation-by-selection system.

Stage 1: Web training

1. We give a basic description of the objective of the application and the study to the participants.
2. We open the application in the browser with the default model (*Pedret*).
3. We proceed to explain the navigation controls and the most relevant tools to explain.
4. The participant expends between 2-3 minutes to familiarize with the environment and tools freely.

Stage 2: Web tasks

1. We ask the user to perform one of the two tasks with one of the two modes.
2. We ask the user to perform the same task but this through the alternative mode.
3. We ask the user to perform the remaining task using the alternate mode.
4. We ask the user to perform the same task with the first mode used for the first task.

Stage 3: VR training

1. We open the application in the browser with the default model (*Pedret*).
2. We proceed to explain the navigation controls and the most relevant tools to explain.
3. The participant expends between 2-3 minutes to familiarize with the environment and tools freely.

Stage 4: VR tasks

1. We ask the user to perform one of the two tasks with one of the two modes.
2. We ask the user to perform the same task but this through the alternative mode.
3. We ask the user to perform the remaining task using the alternate mode.
4. We ask the user to perform the same task with the first mode used for the first task.

Stage 5: Questionnaire

1. For task (1) on the Web, I consider that the manual mode is more practical than the automatic. Score from 1 to 10 where 1 means "completely disagree" and 10 "completely agree".
2. For task (2) on the Web, I consider that the manual mode is more practical than the automatic. Score from 1 to 10 where 1 means "completely disagree" and 10 "completely agree".
3. For task (1) in VR, I consider that the manual mode is more practical than the automatic. Score from 1 to 10 where 1 means "completely disagree" and 10 "completely agree".
4. For task (2) in VR, I consider that the manual mode is more practical than the automatic. Score from 1 to 10 where 1 means "completely disagree" and 10 "completely agree".
5. At a general level, I consider that the manual mode is more practical than the automatic. Score from 1 to 10 where 1 means "completely disagree" and 10 "completely agree".
6. Rate from 1 to 10 the intuitiveness and comfort of navigation in Web mode (10 being the maximum score)
7. Rate from 1 to 10 the intuitiveness and comfort of navigation in VR mode (10 being the maximum score)
8. Rate from 1 to 10 the usefulness of the secondary view (10 being the maximum score)
9. Rate from 1 to 10 the relevance of the photos shown in the photo collection (10 being the maximum score)
10. Rate from 1 to 10 the usefulness of the collection of photos (10 being the maximum score)
11. Rate from 1 to 10 the usefulness of the projection of the photograph in the scene (10 being the maximum score)
12. Rate from 1 to 10 the usefulness of the Web mode to explore the model and perform tasks like the previous ones (10 being the maximum score)
13. Rate from 1 to 10 the practicality of VR mode to explore the model and perform tasks like the previous ones (10 being the maximum score)
14. Rate from 1 to 10 the level of immersion in the scene using the Web mode (10 being the maximum score)
15. Rate from 1 to 10 the level of immersion in the scene using VR mode (10 being the maximum score)
16. Rate from 1 to 10 the general usefulness of the application for use in virtual web museums (10 being the maximum score)
17. Answer *true* or *false* in the following questions:
    (a) At times I have become disoriented or lost in the scene.
    (b) I got dizzy in VR mode.
    (c) After the initial explanation, I have required additional help to navigate the scene and perform the tasks.
    (d) I have encountered difficulties completing one or more tasks.
    (e) This is the first time I have used a virtual reality headset.
    (f) I have previous experience (+ 4h) navigating in 3D environments either in other virtual museums or in video games.

Questions	P1	P2	P3	P4	P5	AVERAGE
Q1	4	10	5	3	3	5
Q2	5	10	9	3	3	6
Q3	4	10	5	3	7	5,8
Q4	5	10	9	3	7	6,8
Q5	3	10	5	3	6	5,4
Q6	5	9	5	6	6	6,2
Q7	7	9	7	9	7	7,8
Q8	8	10	6	7	8	7,8
Q9	5	9	8	7	6,5	7,1
Q10	6	10	9	9	7	8,2
Q11	6	9	10	9	8	8,4
Q12	6	10	8	8	8	8
Q13	9	8	7	9	7	8
Q14	6	3	5	3	2	3,8
Q15	9	10	10	8	9	9,2
Q16	10	9	10	6	7	8,4
Q17_a	1	1	0	0	0	0,4
Q17_b	0	0	0	0	0	0
Q17_c	1	1	1	1	1	1
Q17_d	1	1	1	0	0	0,6
Q17_e	1	1	0	0	1	0,6
Q17_f	0	0	1	1	0	0,4

Fig. 44: Study results