Hand-held portfolio for 3D animated models

Author: Jose Luis Recatalá Bort
Supervisor: Raúl Montoliu Colás

Universitat Jaume I, Video Games Design and Development Degree, 03/06/2018

Abstract

This document is presented as a Project Memory for the development process of an application that allows the user to create his/her own personalized portfolios for 3D animated models.

Such project consists of developing an application for Android mobile devices with two basic functionalities. On the one hand, the possibility of allowing the user to import his/her own 3D animated models from the memory of his/her device in runtime, with the help of an integrated file browser. On the other hand, a series of tools that the user will be able to use to interact with the previously imported model, in order to create unique compositions by editing the rotation of the model, the zoom of the camera, the current frame of the animation, or the background scenario. Once finished, the user will be able to export those compositions as image files, so they can be easily shared through the Internet.

The software used to develop the application is Unity Engine 2017.

Index of contents

1 Technical proposal........ 1
1.1 Summary........ 1
1.2 Keywords........ 1
1.3 Introduction and project motivation........ 2
1.4 Related subjects........ 2
1.5 Project goals........ 2
1.6 Task and time planning........ 3
1.7 Expected results........ 4
1.8 Tools........ 4
2 Design document........ 5
2.1 Functional description........ 5
2.2 User interface........ 7
3 Development process........ 12
3.1 Introduction........ 12
3.2 Interactions with 3D models........ 13
3.2.1 Rotation tool........ 14
3.2.2 Zoom tool........ 15
3.2.3 Play/pause animation tool........ 17
3.2.4 Background change tool........ 19
3.2.5 Screenshot tool........ 21
3.3 Import of 3D models........ 23
3.3.1 File browser implementation........ 25
3.3.2 Model importer preparation........ 28
3.3.3 Data saving and loading........ 29
3.4 Scene navigation........ 31
3.4.1 Standard navigation........ 32
3.4.2 Quick access buttons........ 34
3.5 Visual design........ 35
3.5.1 User interface design........ 35
3.5.1-A Mockup elaboration........ 36
3.5.1-B 2D design process........ 37
3.5.1-C Exportation process........ 37
3.5.2 3D model design........ 38
3.5.2-A Geometry design........ 39
3.5.2-B Texturing........ 40
3.5.2-C Model animation........ 42
4 Results........ 45
4.1 Introduction........ 45
4.2 3D models import functionality........ 46
4.3 Interaction tools........ 47
4.4 Final results........ 47
5 Conclusions........ 48
5.1 Introduction........ 48
5.2 Visualization of 3D models........ 48
5.3 Scene composition and exportation........ 49
5.4 Model import and creation of customized portfolios........ 49
5.5 General conclusions........ 49
6 References........ 50

Index of figures

Figure 1 Mockup: main screen.......... 7
Figure 2 Mockup: main Screen, after selecting a model.......... 7
Figure 3 Mockup: model deletion dialogue.......... 8
Figure 4 Mockup: model deletion confirmation.......... 8
Figure 5 Mockup: file browser.......... 9
Figure 6 Mockup: confirmation for importing a file.......... 9
Figure 7 Mockup: error when trying to import a model.......... 10
Figure 8 Mockup: model successfully imported.......... 10
Figure 9 Mockup: 3D scene.......... 11
Figure 10 Free rotation versus horizontal rotation.......... 14
Figure 11 3D model viewer without background.......... 20
Figure 12 Mockup sample images.......... 36
Figure 13 Graphic defnition of a T vertex.......... 40
Figure 14 No-textured model versus textured model.......... 41
Figure 15 Template texture and final texture.......... 41
Figure 16 Geometry and skeleton in a standing pose.......... 44
Figure 17 Geometry and skeleton in a running pose.......... 44
Figure 18 File browser final appearence.......... 46
Figure 19 Main screen final appearence.......... 46
Figure 20 3D model viewer final appearence.......... 47

Index of tables

Table 1 Developed tools for interacting with 3D models.......... 13
Table 2 Import tools from the Asset Store.......... 24

1 Technical Proposal

1.1 Summary

This section presents the Technical Proposal for the end-of-degree project of the Video Games Design and Development Degree. Such project consists of the development of an application for Android mobile devices whose basic function is to show three-dimensional models and play their animations through a predefined 3D scene. This will allow the user to show his/her work quickly, easily and attractively in any situation that requires it. The application will be developed using Unity Engine, and will include tools for modifying different visualization parameters, obtaining screenshots for an easy sharing, and even updating the work with new models and animations created by the user.

1.2 Keywords

- Personal portfolio.
- 3D modeling and animation.
- Unity engine.
- Android operating system.
- Mobile application.

1.3 Introduction and project motivation

This end-of-degree project aims to develop an application that may be useful for those with a 3D design oriented profile who seek to show their work through a more physical approach, and not digital.

The programming of the application is established as the part of the project with the highest weight, so most of the time will be dedicated to implement functionalities of the application, starting with the simplest ones and leaving the most complex ones for the end, given its higher level of difficulty and the consequent need to search for documentation.

So, first, those functionalities related to the treatment of three-dimensional models will be implemented, as its rotation, the play of animations and the capture of screenshots, followed by those that allow the navigation between the different screens and the general control of the application through touch controls. Finally, the import of models from file will be implemented, which will allow the user to load his/her own models.

Once the programming section has been completed, the different visual elements that the application requires will be designed. This is, on the one hand, a 2D interface that allows users to receive information and interact with buttons and other elements, and, on the other hand, a few three-dimensional models to be able to test the features implemented for the application.

The motivation to carry out this project lies in the lack of tools of these characteristics, as well as in the bet for personal treatment when showing the own work and/or seeking for employment in the field of 3D design.

1.4 Related subjects

- Programming I and II (VJ1203 & VJ1208).
- Game Engines (VJ1227).
- 2D Design (VJ1209).
- Video Game Art (VJ1223).
- 3D Design (VJ1216).

1.5 Project goals

1: Develop an application that allows to visualize three-dimensional models and reproduce their animations.

2: Allow the composition of scenes that contain those models for its later exportation as image files.

3: Offer the user the possibility of importing his/her own models and animations, thus being able to create a personalized portable portfolio.

1.6 Task and time planning

Next, the content of the project in terms of specific tasks to be carried out, with its corresponding temporary cost is exposed. These tasks are divided into three blocks, depending on the area they are related to.

PROGRAMMING (160h)

- Interface functionalities (30h):
· Implement menu navigation (10h).
· Implement button functions (20h).

- Interactions with the models (50h):
· Implement the rotation of the models (20h).
· Implement the camera zoom (5h).
· Implement the functionalities for displaying the animations (10h).
· Implement a screenshoting tool (5h).
· Implement the scenario change (10h).

- Model import in runtime (80h):
· Implement a system file browser (30h).
· Implement the model import in the current senario (50h).

DESIGN (80h)

- Design the interface elements (20h).
- 3D models (25h):
· Model the 3D geometry (10h).
· Texturize the model (5h).
· Animate the model (10h).

- Scenarios (25h):
· Model the 3D geometry (10h).
· Texturize the model (15h).

- Compose scenarios in Unity (10h).

DOCUMENTS (60h)

- Elaborate the Technical Proposal (5h).
- Elaborate the Design Document (5h).
- Elaborate the Project Memory (40h).
- Prepare the final presentation (10h).

1.7 Expected results

After completing the project, it is expected to obtain a tool provided with easy and quick access that will help the designer to show his/her work in the situations that require it, being able to use it to elaborate a hand-held portfolio that contains his/her most representative works, as well as to share them with his/her contacts for a better diffusion.

The final appearence should be something similar to *Overwatch* Hero Gallery [1] where the 3D models for every character in the game can be found, as well as the corresponding animations. Other examples and sources of inspiration can be found in the *Super Smash Brothers* Trophy gallery [2], or in the Lolking 3D model viewer [3].

1.8 Tools

- Unity Engine [4]
- Monodevelop [5]
- 3D Studio Max [6]
- LibreOffice Writter [7]

2 Design Document

2.1 Functional description

As stated in the Technical Proposal document, the application that is intended to be developed must offer the user a series of operations and tools with which to interact with his/her own three-dimensional models, in order to create his/her own virtual viewer. Such viewer will allow users to display and share their work in a simple and quick way. The functions that the application must offer in order to achieve this objective are listed and detailed below:

- 3D model importation from file: the most important feature, that gives the project meaning as a personal portfolio. This function will allow the user to search their own 3D models on their mobile device, in order to load them into the application and be able to view them and play their animations. In order to search for the models, the application will open a simple file browser engine, which will show the user the file folders in the system, as well as their content. Once the desired file has been found and selected, it will be loaded into the application, as long as the file is recognizable. In order to load such files, they must have been exported in an easily treatable and recognizable format, so the FBX format has been chosen, since most current 3D editing softwares can export and import files with such extension. The names of the imported models will appear in a list on the main screen, to ease their search. The option "import new model" will always appear at the end of this list of models, as one more element of it. By tapping on it, the mentioned file browser will open.

- Visualization of three-dimensional models: after importing a 3D model, it will be possible to visualize it through a predefined three-dimensional scene. To visualize a model, the user just has to press the name of the desired file in the list of models on the main screen. Once this is done, a tab will be displayed offering the user the possibility of viewing the model, as well as eliminating it, through two similar buttons. By pressing the visualize button, the application will load a predefined 3D scene and place the three-dimensional model in the center. In this scene, it will be possible to interact with the model to visualize it better through simple touch controls. By tapping anywhere on the screen and draging the finger, the model will rotate horizontally in the direction in which the finger was dragged, so that the model can be viewed from any angle. In addition, it will also be possible to zoom-in or zoom-out, simply by pinching the screen.

- Deletion of three-dimensional models: if the user decided that, in order to keep his/her portfolio updated and organized, its necessary to eliminate one of the previously imported models. This can easily be done by pressing the right button in the tab that is displayed when tapping on the desired model. This action can not be undone, so it is convenient to show a confirmation message before proceeding with the delete operation. After a model is deleted, it must disappear from the list of models.

- Playing animations: in addition to the basic visualization features discussed above, it will also be possible to play the animation of the model (if it is animated). By default, the animation will be paused in its first frame when entering the model viewer. By pressing the play button, the animation will be played and repeated in a loop. Pressing the same button again will stop the animation. In addition to this button, the user can also handle a horizontal slider to advance or rewind the animation to the frame they want. This slider will only be interactable while the animation is paused, although while the animation is being played, it will update its position according to the current frame. The user will be able to return to the main screen by simply tapping the back button of the device.

- Change the 3D scene: the application will come by default with a variety of three-dimensional backgrounds, each with its geometry, textures, lighting, and so on. The user will be able to change the background of the 3D scene when viewing a model to try to find the best composition of which to take a screenshot. This change will be carried out through a button, which when pressed will change the current scenario to the next in an internal list. If the current scenario is the last one, pressing the scenario change button will show the first one again, so the scenario change will be treated in a cyclical and organized manner.

- Take screenshots: taking a screenshot will be as simple as pressing the capture button. By pressing this button, the user interface will be hidden, and a screen capture will be made, which will show the model in the current frame, with the chosen rotation and zoom, and placed in the scenario that the user chose. After finishing the capture process, the interface will be displayed again. All the images extracted from the application using this method will be later found in the images gallery of the Android device.

2.2 User Interface

In this section, a mockup for the user interface of the application is shown, as well as explanations of how the functionalities are triggered and transitions between screens. It is worth to mention that, since this is an application whose goal is to clearly show artistic works, there is no other possible screen orientation than the horizontal for the user interface, given its panoramic capability.

First, there is the main screen, the one shown when the app is opened. Figure 1 shows how this screen has a list of the imported models. As stated above, the last element of this list will always be the "Import model" option. The first time the app is opened, this list will be empty, except for the "Import model" option.

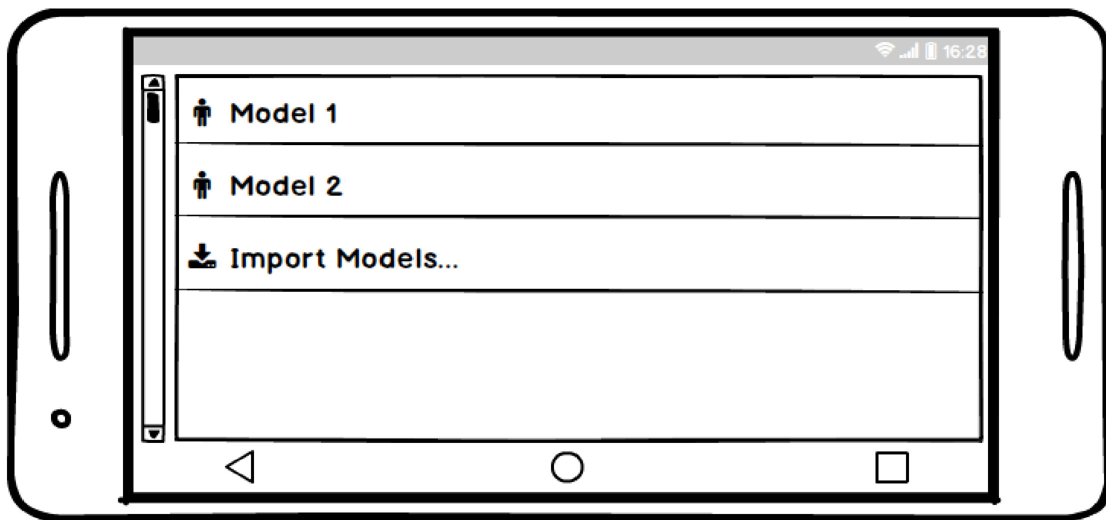


Figure 1: Main Screen.

By taping any model listed on this screen, a tab will be shown, which will offer some options to work with the selected model. Figure 2 shows how such tab is displayed.

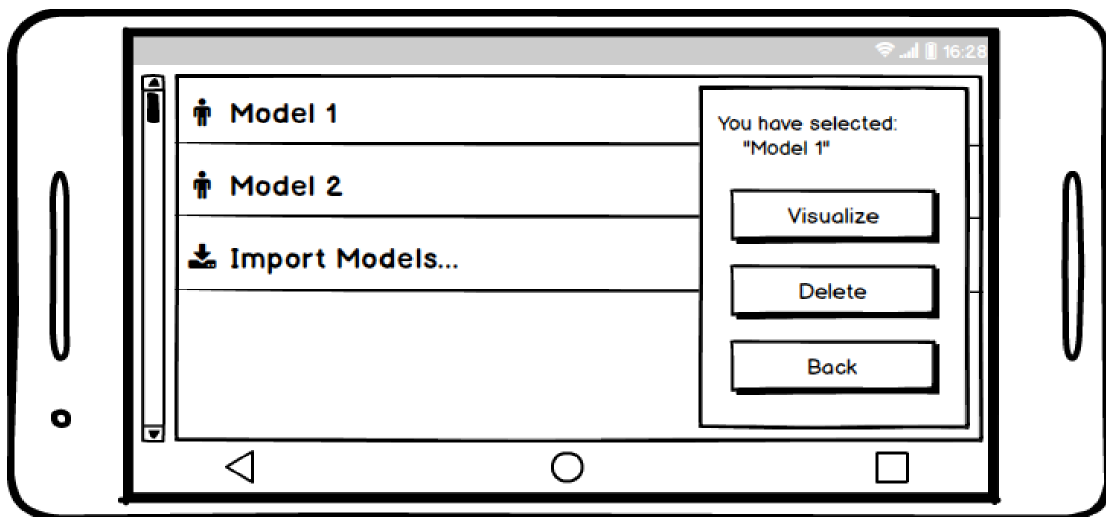


Figure 2: Main Screen, after selecting a model.

By pressing the "Delete" button, the app will show a dialogue window containing a warning message and asking the user for confirmation. If the user goes on with the deletion process, the selected model will disappear from the main screen list and a message of confirmation will be shown. Figures 3 and 4 illustrate this.

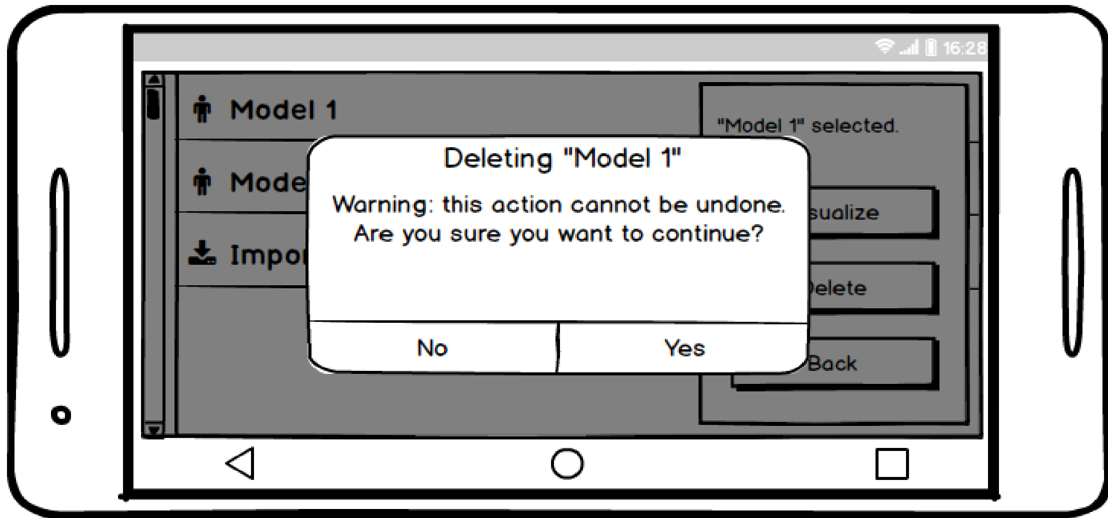


Figure 3: Model deletion dialogue.

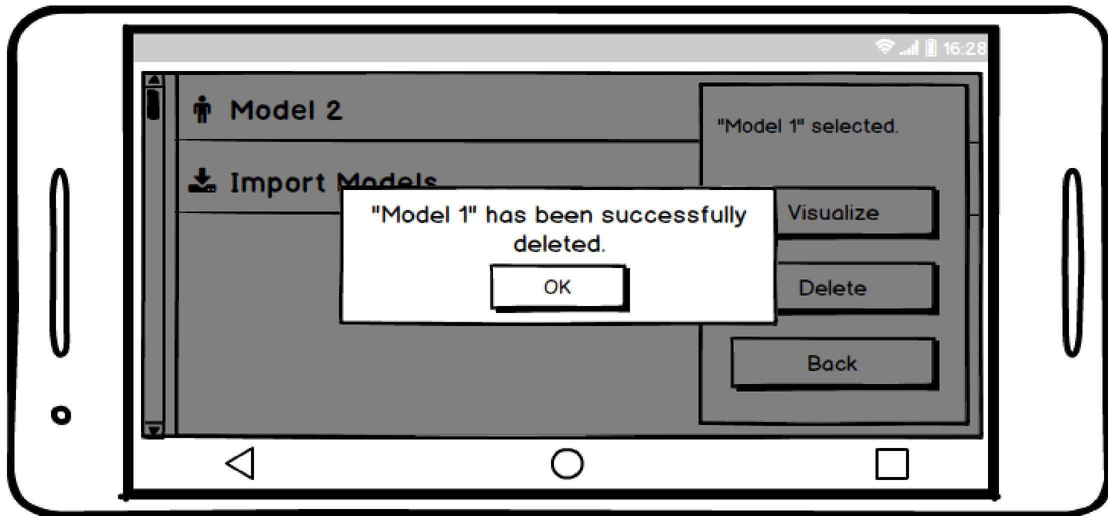


Figure 4: Model deletion confirmation.

To import a new model, the user may use the "Import model" option, at the end of the list on the main screen. After doing this, a simple file browser will be shown, where the user can search for the file he/she wants to load, looking inside the folders in his/her device. Figure 5 shows how the file browser will look like.

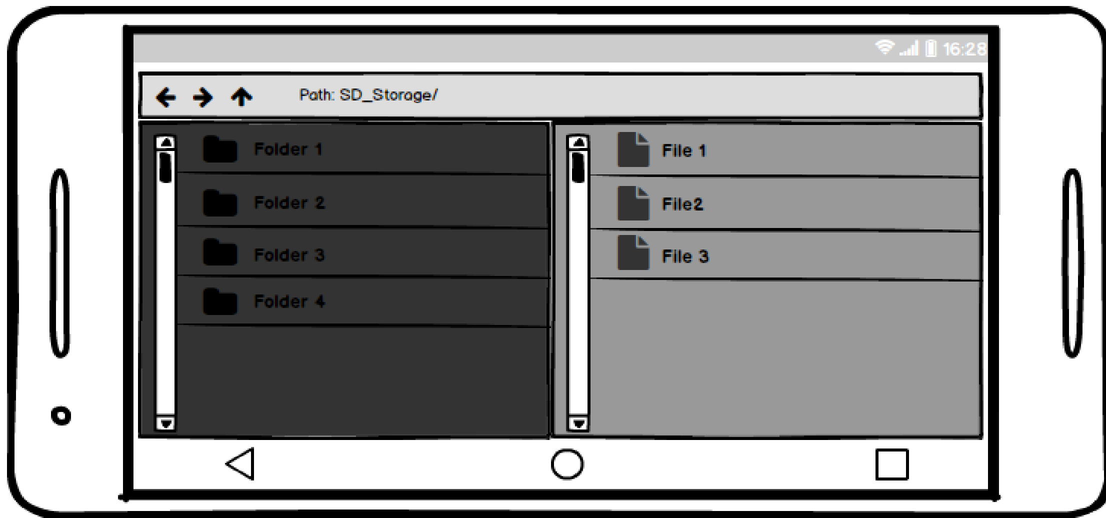


Figure 5: File browser.

When the desired file is selected, the app will show a new dialogue window, asking the user for confirmation. Selecting "Yes", will import the model into the app, unless the extension of the selected file is not appropriate. If that is the case, a message will be shown to acknowledge the user. Otherwise, the model will be successfully loaded and its name will appear in the model list of the main screen from now on. Figures 6, 7 and 8 illustrate how the screen will look like in these cases.

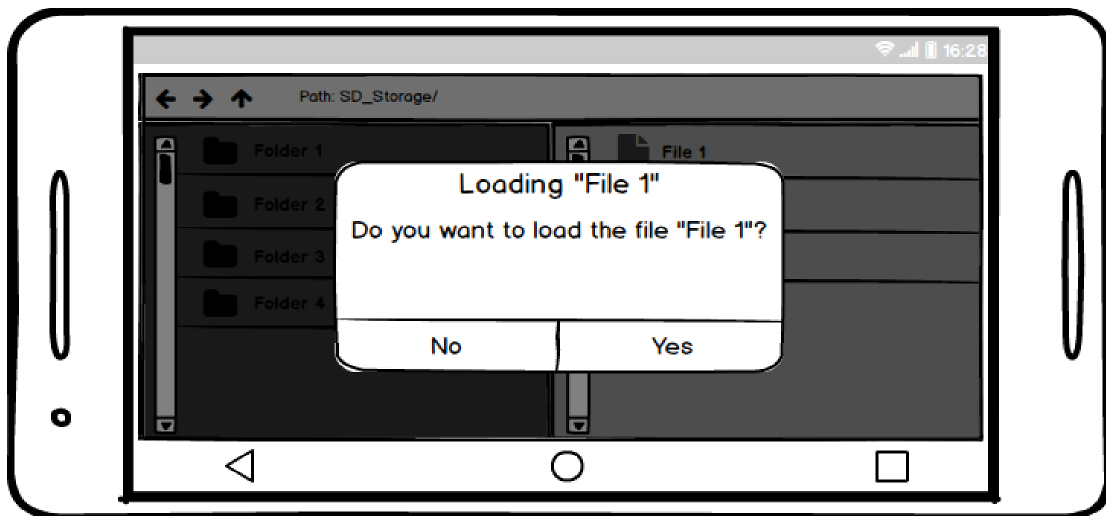


Figure 6: Confirmation for importing a file.

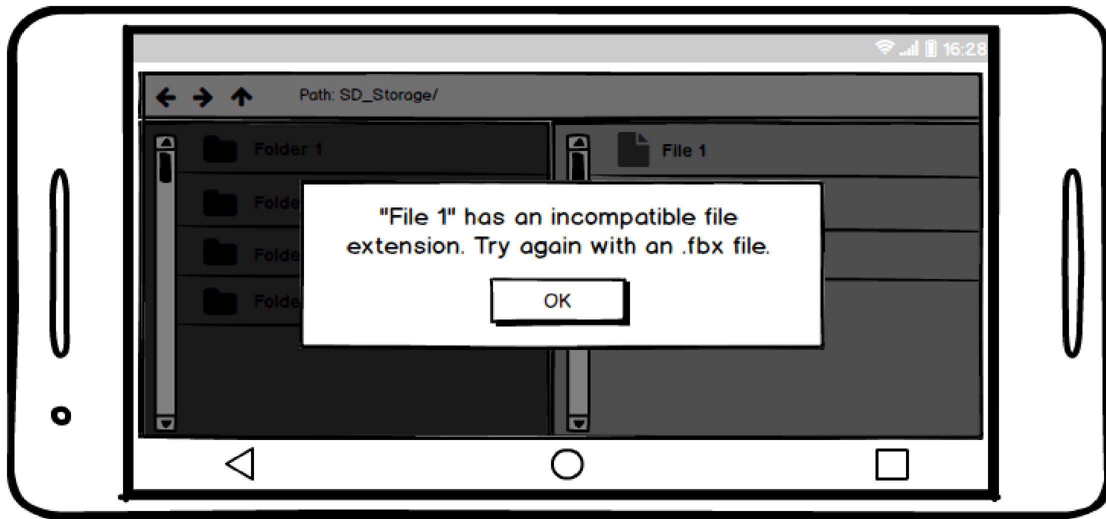


Figure 7: Error when trying to import a model.

Figure 8: Model successfully imported.

Back in the main screen, when the user taps on a model from the list, and chooses the option "visualize", the app will load a 3D scene to show the model, as shown in Figure 9. Here, the user can rotate horizontally the model, as well to zoom-in/zoom-out the camera for a better visualization. Using the play button and the slider in the bottom of the screen, the user can control the play of the animation, aun using the upper buttons they can perform a screenshot and change the background of the 3d Scene.

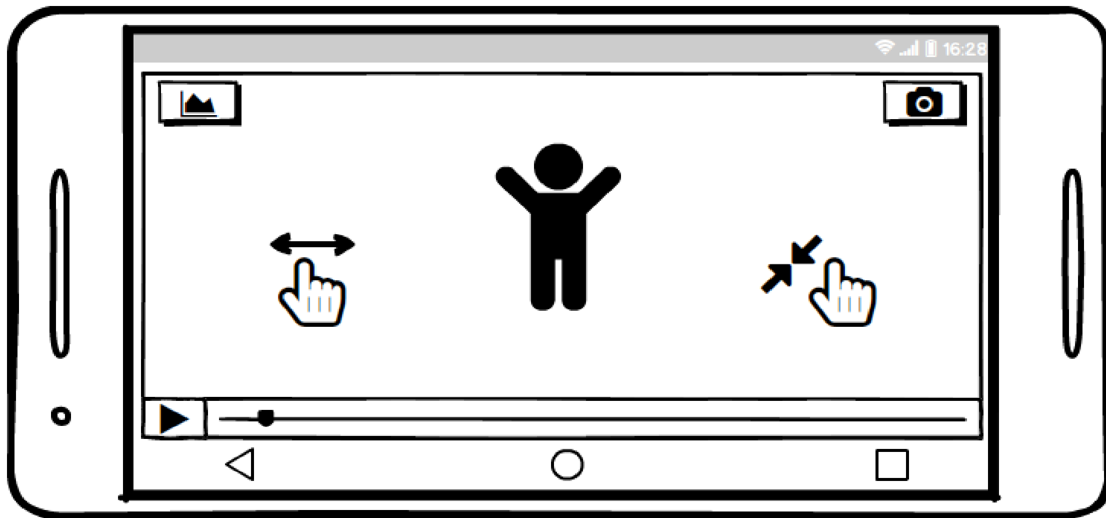


Figure 9: 3D scene.

3 Project Development

3.1 Introduction

The preparation of a Technical Proposal, which reflects the general idea of the project and the time cost of each of its sections, is a very important step to take into consideration when starting a project like this. Similarly, the elaboration of a Design Document, which shows an outline of what the final result is intended to be in terms of appearance and functionality, is very useful to establish a base over which start building the project. However, taking the step between the aforementioned design process and the development process can result really hard and confusing if an appropriate procedure is not followed.

A good methodology to be followed in order to focus and know where is better to start, and so optimize efforts and avoid consuming more time than necessary, is to analyze the project to differentiate every task in which it is divided, establish objectives and assign a priority to each task based on such objectives or the influence they will have over other parts of the project.

Based on this methodology, in the next sections it is explained, orderly and in detail, the development process, delving into the work done for each one of the tasks that the project consists of.

3.2 Interactions with 3D models

Analyzing the tasks that make up the project, as well as their priority, it can be said that if there is a better part from which start, this is from the programming section, which importance far exceeds that of the artistic section.

As it has been reflected both in the Technical Proposal and the Design Document from the first sections, the developed application has a strong visual component. It displays a three-dimensional model on the screen and allows the user to interact with it and its surroundings through simple tactile controls to modify its visualization at the user's own pleasure.

That is why this visual component is of great importance, as well as the correct functioning of the tools that allow the user to interact with it, placing it in a more imperative position than other sections of the project.

Although they will be explained in further sections, Table 1 resumes each one of the developed tools, explaining briefly what their purposes are, how are they operated, and what are the expected outputs after operating them.

Tool	Purpose	Controls	Output
Rotation tool	Change the rotation of the 3D model so that it can be seen from different angles.	Tap with a finger, and drag it across the screen.	The rotation of the model is updated in runtime, based on the length and direction of the user's drag.
Zoom tool	Change the field of view of the camera so that the scene can be observed in detail.	Pinch the screen with two fingers, and drag them across the screen.	The field of view is updated in runtime, based on the length and direction of the user's drag.
Play/Pause animation tool	Play or pause the animation of the model, depending on its current state.	Press a button to pause the animation. Press it once more to start playing it.	If the animation is currently being played, it is stopped. Otherwise, it starts being played.
Background change tool	Switch between backgrounds for the 3D scene.	Press a button to hide the current background and show the next one.	The background of the scene is changed.
Screenshot tool	Take a picture of the current state of the scene.	Press a button to take the screenshot.	A picture of the current state of the scene is taken and saved in the Android image gallery.

Table 1: Developed tools for interacting with 3D models.

3.2.1 Rotation tool

The rotation had to be an intuitive and easy to use tool, which would allow viewing the model from different angles by just tapping the screen of the device with a finger, and then dragging it horizontally across the scene.

At first, the idea of allowing the user to rotate the model around any axis (x, y and z, the Euler's axes) was considered. However, it was quickly dismissed, since something like that would make the idea of posing animated humanoid models in three-dimensional scenarios meaningless. Instead of allowing a total rotation around the three axes, it was decided that the rotation around a single axis, the vertical one, which would lead to the horizontal rotation of the model, was a better option. As seen in Figure 10, free rotation made some models look unnatural.

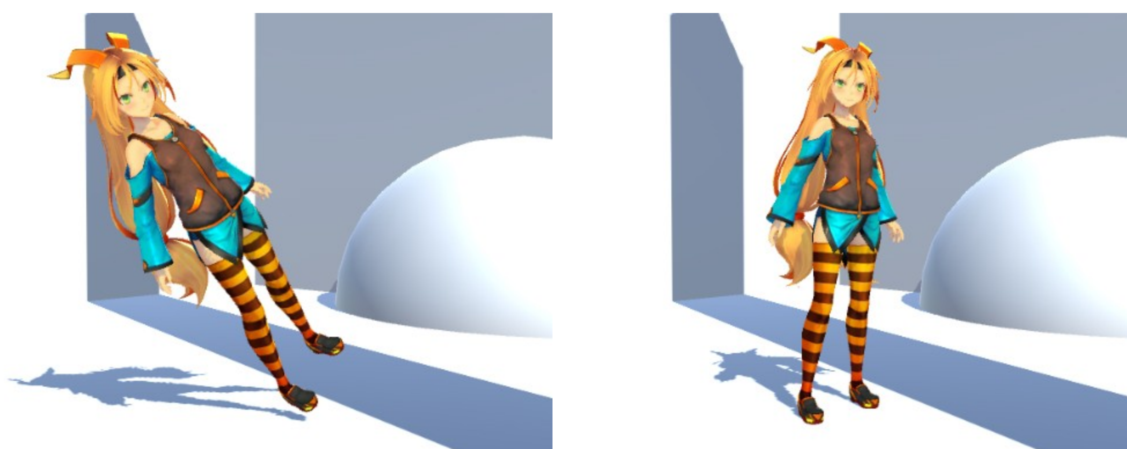


Figure 10: Free rotation (left) versus horizontal rotation (right).

Once the basic concept for the functionality was established, a first version of a simple script was written, in order to detect the movement of a finger dragging across the screen of the device. This script was responsible for collecting and storing the starting and ending coordinates of the user's drag over the screen, thus drawing a two-dimensional vector between the positions established by these coordinates, whose module (or length) would be later converted into degrees, so the model could know how much should it rotate.

This first version succeeded in establishing a standard for reading and storing the coordinates of the user's input. However, it lacked a mechanism to store and update the model rotation, so whenever the user tried to rotate the model after a previous drag over the screen, the model rotation returned to its original value (that is, 0 degrees in each axis). Furthermore, the value sign of the module of the vector resulting from the drag over the screen was not taken into account, so the rotation was performed in the same direction every single time.

In a second version, it was possible to correct these minor errors. The problem of the rotation restarting with every new input was solved by storing the rotation of the model in a variable and updating it each time the user modified it. By doing this, the script could know at any moment the rotation of the object, accessing it every time it had to be modified.

On the other hand, to solve the problem of the rotation only working on a single direction, it only had to be taken into account whether the value of the drag vector module was positive or negative. Thus, when the value turned out to be negative, the rotation would be performed to the left and, in case of being positive, to the right.

After finishing this second version, a new problem emerged. The rotation of the model should be performed whenever the user dragged his/her finger across the screen. However, if the user tapped first any element of the user interface, such as a button or a slider, and then dragged the finger, the rotation of the model was modified. This should not happen, since it would imply that interacting with the elements of the interface would interfere in the final rotation of the model.

To prevent this from happening, some kind of mechanism such as a function that would distinguish wether the user was trying to interact with the interface or not, was needed. This function read the coordinates of the user's input and checked by raycast if it was in contact with any element of the interface, in order to return a boolean variable according to the results. A call to this function was placed before the rotation modification code, so the value of the returned boolean established whether the modification should be performed or not. This code can be checked in Code 1.

```
//Función que comprueba si el cursor toca un elemento de la interfaz.
private bool CursorSobreInterfaz() {
    //Variable "PointerEventData" que usaremos para trabajar con información sobre el cursor
    PointerEventData posicionActual = new PointerEventData(EventSystem.current);
    //La posición actual del cursor, en coordenadas X e Y.
    posicionActual.position = new Vector2(Input.mousePosition.x, Input.mousePosition.y);
    //Lista donde se guardarán los contactos del cursor con otros elementos de la interfaz.
    List<RaycastResult> resultados = new List<RaycastResult>();
    //Se lanza un raycast desde el cursor para intentar hallar contactos, y si los hay, se al
    EventSystem.current.RaycastAll(posicionActual, resultados);
    //Si la longitud de la lista es mayor que 0, es que ha habido algún contacto, y que por t
    return resultados.Count > 0;
}
```

Code 1: Function that checks if the User Interface (UI) has been touched.

Taking the changes made into account, it can be said that this third version meets all the requirements that the rotation tool should take care of and, therefore, it can be considered finished and polished.

A demo video of the rotation tool can be found by following the next link: https://drive.google.com/open?id=1llIKxReofCyKv_6mgrX1rqHre8jSr87E .

3.2.2 Zoom tool

Keeping the focus in the strong visual component the application has, that is the possibility of showing animated three-dimensional models on the screen in an attractive and natural way, it becomes evident the need for a tool that allows the user to expand and reduce the field of view to be able to notice more detailed information in the geometry of the model, in its textures or even in its animations. The zoom is a common tool which can be found easily in any 3D edition software, so it is something the user is supposed to be familiar with, and thus it is appropriate to think that the inclusion of such a tool is a good option to complete the toolbox for the application.

The first step to develop this tool was to think about the most appropriate way to carry out the zoom-in/zoom-out operation, giving rise to three key proposals.

The first one of them relied on a "zoom mode" button, which would override the rotation controls and allow the user to zoom by tapping and dragging one finger across the screen. On the one hand, this option was simple, given its easy implementation, which would have recycled almost completely the model rotation code, thus eliminating the need to implement new controls. However, on the other hand, it would have been uncomfortable to rely on a button to change from rotation mode to zoom mode. Also, the idea of dragging a finger horizontally across the screen to enlarge/reduce the field of view was not intuitive at all. Taking this into consideration, this proposal was dismissed almost immediately.

The second proposal took the idea of controlling the zoom through a button, but taking away the change between rotation mode and zoom mode, as well the cancellation of the rotation controls. Even simpler than the previous one, this proposal relied on two similar functions: one for expansion and another for reduction, that respectively decreased or increased (to a certain extent) the field of view of the camera, each one of them being assigned to a different button. By doing this, it would be easy to control the zoom level of the camera without having to change between rotation and zoom modes. However, it was a bit orthopedic, since the zoom level could not be controlled with total precision and several attempts could be required to find the desired level of zoom. It was decided to also dismiss this idea, due to the lack of comfort and precision.

The third and last proposal was to implement the method that nowadays is considered standard for zooming images on mobile devices, which allows the user to control the zoom level by pinching the screen of the device with two fingers. This completely eliminated the need to interact with buttons, keeping the user interface simple and clean, and also gave the user full control and precision over the zoom level. The only problem was the possibility that zoom controls and rotation controls overlapped, so it would be necessary to control that the rotation would only be executed when the user made a single touch on the screen, since pinching the screen needs two touches.

Thus, it would be necessary to store the coordinates of both touches, as well as calculate the distance between the initial coordinates (those stored at the moment the user touches the screen) and the final ones (the ones stored after dragging the fingers across the screen), which is, as explained before, the module of a vector defined by two positions (only this time two modules are needed). This distance, multiplied by a scaling factor, will result in a value, which added to the value of the current field of view of the camera will produce that desired zoom-in/zoom-out effect. In the Code 2, the full zoom tool code is provided.

In a similar way to the rotation tool, it had to be taken into account if the value of the distances was positive or negative, and also assign a maximum value and a minimum value to the field of view of the camera to avoid problems with infinite zooms.

A demo video that shows the usage of the zoom tool can be found through the following link: https://drive.google.com/open?id=1hbYCJoFvG5VXkN0Vu24kBJrgM0xvJAVK .

```
public Camera camaraPrincipal;      //Cámara a la que se aplicarán las operaciones de zoom.

public float zoomEnPerspectiva = 0.05f;     //Factor de escalado del zoom en modo Perspectiva.
public float zoomEnOrtografica = 0.05f;     //Factor de escalado del zoom en modo Ortográfica.

void Update(){
    //Para que el script sea efectivo, el dispositivo debe poder almacenar más de un toque en pantal
    if (Input.touchCount == 2) {
        //Almacenamos ambos toques para calcular posteriormente la distancia entre sus coordenadas.
        Touch toquePrimero = Input.GetTouch (0);
        Touch toqueSegundo = Input.GetTouch (1);

        //La distancia entre las coordenadas iniciales y las finales de un toque determina si amplia
        //Así pues, creamos un Vector2 que almacene las coordenadas de inicio y de fin de ambos toq
        Vector2 toquePrimeroDistancia = toquePrimero.position - toquePrimero.deltaPosition;
        Vector2 toqueSegundoDistancia = toqueSegundo.position - toqueSegundo.deltaPosition;

        //Ahora simplemente calculamos la distancia inicial y final para poder calcular más tarde la
        float distanciaInicial = (toquePrimeroDistancia - toqueSegundoDistancia).magnitude;
        float distanciaFinal = (toquePrimero.position - toqueSegundo.position).magnitude;

        //Calculamos pues la diferencia entre distancias.
        float diferenciaDeDistancia = distanciaInicial - distanciaFinal;

        //El aumento/reducción se calcula multiplicando la diferencia de distancias por el factor de
        camaraPrincipal.fieldOfView += diferenciaDeDistancia * zoomEnPerspectiva;
        //Establecemos un máximo y un mínimo razonables para el campo de visión.
        camaraPrincipal.fieldOfView = Mathf.Clamp (camaraPrincipal.fieldOfView, 40f, 60f);
```

Code 2: Zoom tool code.

3.2.3 Play/Pause animation tool

Rotation and zoom tools are very useful for appreaciating in more detail the shape and appearence of the three-dimensional model, but without a tool responsible of playing and pausing its animations, the application loses a lot of potential, especially when it is intended to be used as an artistic portfolio, in which 3D animations are an important feature, providing a dynamism and variety that otherwise would not be possible.

The play and pause animation tool had to have several well differentiated parts. In first place, a mechanism that would allow the user to play and pause the animations of a three-dimensional model easily and effectively. Since both the rotation and the zoom tools used touches and drags on the screen, it would have been difficult to use touches for this one as well, since the functions could easily overlap. Instead, it was decided to use a simple button, that by a first press would play the animation of the model, and by another press would pause it.

Nowadays, many video players for mobile devices, despite offering the possibility of touching the screen to pause or play a content, keep a button on its interface dedicated to this same purpose. So, even though it may not be the most intuitive and common option, it was still a good choice.

Secondly, the user had to have some kind of control over the animation, such as the possibility of changing its current frame (that is, the current state of a video or, in this case, animation), as well as the playback speed. Following the example of video players, a kind of slide bar is usually the most common element for controlling frames, so it was decided to add a slider next to the mentioned play/pause button. However, the user would only be able to interact with this bar while the animation was paused. If the animation was currently playing, this slider would constantly update the current frame of the animation instead. For the playback speed control, it was decided to use two buttons: one to accelerate the speed, and another to slow it down.

With all these ideas about the functionalities of the tool, it only remained to write the necessary code so that everything started to work.

The functionality of playing and pausing the animations of the three-dimensional models was simple to implement. The only thing to take care of was to find the animation of the model and put it into operation. This was possible thanks to a Unity component, that is attached to anything that has an animation, namely, a 3D model, a sprite, or even an element from the interface. That is the Animation component. Once this component is assigned to a 3D model, playing its animation, as well as pausing it, is as easy as writing a single line of code.

Since the responsible for both playing and pausing the 3D model animation was going to be a single button, it was necessary that the function assigned to it could be able to distinguish in some way when it had to do one thing and when the other one. A simple method to carry out this distinction could be to check the playback speed of the animation, and proceed accordingly. If its value is 0, that means that the animation is currently paused, and therefore the necessary code to start playing it must be executed. Otherwise, the animation is currently being played, and thus it has to be paused. This is shown in the Code 3.

```
//Función para detener o reanudar la animación actual.
public void PausarAnimacion(){
    //Necesitamos tener acceso al componente Animation.
    Animation animacion = GestorModelos.Instancia ().GetModelo ().GetComponent<Animation> ();
    //Siempre y cuando haya un componente Animation, haremos lo siguiente:
    if (animacion != null) {
        //Si la velocidad no es 0, le cambiamos el valor a 0.
        if (animacion [animacion.clip.name].speed > 0) {
            animacion [animacion.clip.name].speed = 0;
        }
        //De lo contrario, cambiamos su valor a 1.
        else {
            animacion [animacion.clip.name].speed = 1;
        }
    }
}
```

Code 3: Play/pause animation function.

Making the slider or time bar functional was also easy, thanks to the aforementioned Animation component. Since the values of the slider are normalized (that means that its minimum value is 0, and its maximum value is 1), the most optimal option was to also normalize the number of frames in the animation.

By doing this, the initial position of the slider would correspond to the initial frame of the animation, the final position would correspond to the last frame, and thus, any position of the slider between these two will correspond to the appropriate frame. Knowing this, the only thing to take care of was to make the slider update its position based on the normalized value of the current frame of the animation whenever the animation was being played and, when the animation was paused, make the frame of the animation take its normalized value based on the position of the slider. This is shown in Code 4.

```
//Bucle principal.
void Update () {
    //Siempre y cuando haya un componente Animation, haremos lo siguiente:
    if (animacion != null) {
        //Si la animación se está reproduciendo (velocidad de reproducción distinta a 0):
        if (animacion [animacion.clip.name].speed != 0) {
            //Hacemos que la posición del slider se mueva conforme el fotograma actual de la animación.
            barraDeTiempo.normalizedValue = animacion [animacion.clip.name].normalizedTime;
            //Si la animación llega al final, hacemos que empiece de 0.
            //Esto es necesario para que el slider se actualice correctamente.
            if (animacion [animacion.clip.name].normalizedTime >= 1) {
                animacion [animacion.clip.name].time = 0;
            }
        }
        //De lo contrario (velocidad de reproducción igual a 0):
        else {
            //Hacemos que el fotograma actual de la animación se mueva conforme el slider.
            animacion [animacion.clip.name].normalizedTime = barraDeTiempo.normalizedValue;
        }
    }
}
```

Code 4: Slider management implementation.

For the functions that had to be assigned to the buttons responsible of accelerating and slowing the animation, a simple solution was to increase or decrease the playback speed of the animation by a fixed amount with every press. This was possible by accessing the Animation component once more. The only thing that had to be taken into account were the maximum and minimum values. These values were necessary so that the playback speed of the animation could not become too high, nor below zero, in which case the animation would be played in reverse.

In case an imported model was not animated, the Animation component would not be found, and therefore none of these features of the play/pause tool could be operated, although they would still be visible in the interface.

A demo video of the play/pause animation tool can be found by following the next link: https://drive.google.com/open?id=1XgIxW2WHUuLV1sKBx7JIHuEdg3gr69Mb .

3.2.4 Background change tool

Showing an animated three-dimensional model over an empty background was an unattractive feature, as can be seen in Figure 11. It was evident that an application with such emphasis on the idea of showing artistic works needed an appropriate canvas or background where the works of the users could be placed. But with a generic background it would be very difficult to encapsulate the general idea of every model, as well as defining an appropriate environment. So it was decided to implement a tool that would allow the user to change the background of the 3D scene, being able to choose one from a list of backgrounds that the application would have as a default feature.

This tool had to use some kind of mechanism that the user could activate easily, in order to change the current background for another one that could fit better the concept of the work. To avoid the overlapping between the tactile controls of the tools, it was decided to use a simple button, which would cause the background change with just a single press.

Figure 11: 3D model viewer without background.

But first, a couple of backgrounds which could be used to test the tool would be necessary. So, using basic 3D primitives of Unity such as planes, quads, cubes and spheres, two simple backgrounds were created. These auxiliar compositions would be useful to perform tests for the implementation of the background change tool, as well as the general lighting of the scene, the angle and position of the camera, and the projections of the shadows of the model over the stage and vice versa. Once these backgrounds were designed, the necessary code to make the tool operative was the next thing to take into consideration.

The main problem was to maintain visual cleanliness in the viewer, since adding an emerging window just to offer the user the possibility to choose an option from a list in order to change the scene background, would have implied to temporarily cover the model, something that had to be avoided by all means. So it was decided to look for an alternative solution, one that depended exclusively on pressing a single button to perform the background change.

Thus, it was decided that a good solution was to create a list of three-dimensional backgrounds, which could be traversed based on an index that increased as the button was pressed. The element corresponding to the current index remained visible in the scene, while the others were deactivated so that they could not be seen. Once reached the end of the list, the index would start again from the beginning. By doing this, the control of the background would be covered with just the press of a button.

However, it is worth mentioning that this solution is valid only due to the characteristics of the project itself, which requires only a few backgrounds. If a greater number of backgrounds would be needed, this solution would be a very bad one, since it would require to press the button many times to go through the entire list of backgrounds, not to mention that a mistake as simple as pressing the button one more time than needed would imply having to travel again through the list to find the desired background.

Taking this into consideration, a good proposal for an alternative solution would be to use two buttons, one to increase the index of the list, and another one to decrease it. But, without doubt, the best solution for that case would be the one that has been dismissed from the beginning: an emerging window that offers the user a list of backgrounds to choose, at the cost of sacrificing the visualization of the model.

However, given the prototypical nature of the project, only a few backgrounds were needed, and thus the one-button solution was more than enough. This solution is shown through Code 5.

```
//Inicialización.
void Start(){
    //Inicializamos el índice a 0.
    indiceLista = 0;
    //El primer escenario activo será el primero de la lista.
    escenarioActual = listaDeEscenarios [0];
    //Se activa el escenario (todos están desactivados por defecto)
    escenarioActual.SetActive (true);
}

//Función que permite el cambio de escenario.
public void CambiarEscenario(){
    //El escenario actual se desactiva.
    escenarioActual.SetActive (false);
    //Si no se ha llegado al final de la lista de escenarios:
    if (indiceLista != listaDeEscenarios.Length - 1) {
        //Se avanza en la lista aumentando el índice.
        indiceLista++;
    }
    //De lo contrario:
    else {
        //Se reinicia el índice a 0.
        indiceLista = 0;
    }
    //Se actualiza el escenario actual, y se activa para que se vea.
    escenarioActual = listaDeEscenarios [indiceLista];
    escenarioActual.SetActive (true);
}
```

Code 5: Implementation of the Background change tool.

A video showing how the background change tool works can be found by following the next link: https://drive.google.com/open?id=1IXxJkst8SLmMnfzomB_ja8ksi3HhqWxc .

3.2.5 Screenshot tool

Although the main objective of the application is to show personal artistic work in a physical approach, it is also interesting to offer the user the possibility of sharing his/her work through the Internet. That is why the idea of implementing a tool to save the scene that the user has made, combining their model and a background, through an image is quite convenient, since it is easier to share an image than a 3D scene.

Nowadays, the use of social networks through mobile devices is more common and frequent than ever (WhatsApp, Facebook, Instagram, etc.), and sharing images through these social networks is very simple.

So, such tool could be beneficial for several reasons. First, posting and sharing personal work on social networks increases the scope and diffusion of it. Secondly, saving images of the work done helps to keep a register and to elaborate a more consistent portfolio, and although it is not the best medium to show 3D modeling and animation, it does help to get a general idea of the final appearence of the work.

Finally, it should be noted that an image needs much less space than a three-dimensional model, so saving screenshots of the models made, will help to save space, keeping images in the device instead of 3D files.

Nowadays, tools to take screenshots are very common and easy to find in any mobile device. However to operate them it is usually necessary to run the current application in background. For this case, it was intended something easier to use and more intuitive, so it was decided to integrate a screenshot taking tool in the application itself, which could be operated simply by pressing a button.

Pressing that button would capture what is currently being displayed on the screen, and convert it into a bidimensional texture, which would later be saved in a JPG format image inside the images gallery of the mobile device (the JPG format has been chosen because of its balance between image quality and file size).

To write the code, first a function was created to take the screenshot. Thanks to the ReadPixels and Apply methods of the Texture2D class of Unity, which read and store the color information of each pixel in a defined area (in this case, the whole screen) and transfer that information into a texture, creating such function was very simple. However, saving that screenshot as an image inside the image gallery of the mobile device, needed some research and extra job.

Searching about the matter in the Unity forums and official documentation was very useful, since it revealed that, for this to be possible, it was necessary to make use of certain Android classes and activities, whose usage is illustrated below, through Code 6.

```
//El canvas que contiene la interfaz.
public GameObject canvas;
//El nombre que recibirán las capturas de pantalla.
private string nombreDeCaptura;
//String con el nombre de la clase Java que usaremos para almacenar la captura de pantalla en la galería.
protected const string MEDIA_STORE_IMAGE_MEDIA = "android.provider.MediaStore$Images$Media";
//Una actividad.
protected static AndroidJavaObject m_Activity;

//Función que guarda la imagen en la galería.
protected static string SaveImageToGallery(Texture2D a_Texture, string a_Title, string a_Description){
    using (AndroidJavaClass mediaClass = new AndroidJavaClass(MEDIA_STORE_IMAGE_MEDIA)){
        using (AndroidJavaObject contentResolver = Activity.Call<AndroidJavaObject>("getContentResolver")){
            AndroidJavaObject image = Texture2DToAndroidBitmap(a_Texture);
            return mediaClass.CallStatic<string>("insertImage", contentResolver, image, a_Title, a_Description);
        }
    }
}

//Función que convierte una textura2D en un mapa de bits.
protected static AndroidJavaObject Texture2DToAndroidBitmap(Texture2D a_Texture){
    byte[] encodedTexture = a_Texture.EncodeToPNG();
    using (AndroidJavaClass bitmapFactory = new AndroidJavaClass("android.graphics.BitmapFactory")){
        return bitmapFactory.CallStatic<AndroidJavaObject>("decodeByteArray", encodedTexture, 0, encodedTexture.Length);
    }
}

//Actividad Android.
protected static AndroidJavaObject Activity {
    get {
        if (m_Activity == null) {
            AndroidJavaClass unityPlayer = new AndroidJavaClass ("com.unity3d.player.UnityPlayer");
            m_Activity = unityPlayer.GetStatic<AndroidJavaObject> ("currentActivity");
        }
        return m_Activity;
    }
}
```

Code 6: Functions to use Android images gallery.

So, just by adding a few simple adjustments, such as hiding the user interface before taking the screenshot and showing it again once the process finished, encapsulate the capture function in a coroutine so that it does not interfere in the process of hiding/showing the interface, and make the name of the resulting image to be generated automatically, the screen capture tool was considered to be succesfully implemented. The result can be seen in Code 7.

```
//Función para tomar una captura de pantalla.
public void CapturarPantalla(){
    canvas.SetActive (false);
    StartCoroutine (CorutinaCapturarPantalla (Screen.width, Screen.height));
}

//Corutina para tomar una captura de pantalla.
private IEnumerator CorutinaCapturarPantalla(int anchura, int altura){
    yield return new WaitForEndOfFrame ();
    nombreDeCaptura = System.DateTime.Now.ToString ("yyyy-MM-dd_HH-mm-ss");
    Texture2D textura = new Texture2D (anchura, altura);
    textura.ReadPixels (new Rect (0, 0, anchura, altura), 0, 0);
    textura.Apply ();
    yield return textura;
    string ruta = SaveImageToGallery (textura, nombreDeCaptura, "ScreenShot");
    canvas.SetActive (true);
}
```

Code 7: Screenshot functions.

A demo video that shows the usage of the screenshot tool can be found by following the next link: https://drive.google.com/open?id=1ZVcDXKDayCiyHsRdXYkeFqYD61BbfXgT .

3.3 Import of 3D models

Even taking into consideration the importance of the implementation of the 3D models interaction tools, the user would be unable to use such tools without the possibility of importing any of his/her artistic works into the application itself.

Given that the implementation of the different interaction tools was initially meant to have a much lower time cost, and since the tasks related to them were supposed to have a lower difficulty, it was decided that the implementation of an importer tool for three-dimensional models, as well as of any support tool that could be needed, such as a file browser dialog, was not the top priority.

Once all the functionalities that allowed the user to interact with their work were implemented, the next step was to identify the tasks to be performed for the model importer implementation process, check their weight over the entire project, and assign a proper priority to each one of them.

In the first place was the 3D model import tool itself, whose importance was vital, since it was the characteristic that gave meaning to the project as a hand-held portfolio for 3D artistic works.

Although it was considered a crucial part of the project, it was not considered at first as the most difficult one since, apparently, there were many external tools in the Unity Asset Store capable of importing 3D models into a Unity project (although later it was found that this was not the case). This is why, at the beginning, priority was given to supposedly more difficult tasks, such as the implementation of the file browser engine or the serialization and deserialization system, whose mission was to keep all the information of the application after its closure, so that after reopening it, everything could be found in its place.

However, after an exhaustive search of documentation about this matter, it became quite evident how extremely difficult it was to develop a tool for importing 3D models at runtime for mobile devices from scratch. The amount of knowledge that this process demanded about internal procedures of the Unity game engine, texture processing, format reading, and modification of native Unity classes, as well as, of course, Java and Android in general, was overwhelming.

Implementing something like that without a base, would have required more resources and more time than was available for this end-of-degree project, not to mention that it would have been a too much difficult task for a single developer.

It was undoubtedly a great problem that jeopardized the integrity of the project and threatened to make it impossible to complete it without having to modify its characteristics, renouncing the import of models in runtime, which, without a doubt, was the most important feature of the entire application. It was necessary to look for an alternative solution.

After looking for information in the Unity forums, it became quite clear how many developers demanded a 3D model import tool of these characteristics, easy to access and utilize. Among the most recommended answers, numerous links were found to the Unity Asset Store, where it was possible to find a multitude of import tools, mostly free, whose technical specifications explained that, apparently, these tools were in fact able to work in runtime.

Employing any of these tools seemed like a viable solution. However, most of these tools did not work as expected. Some were very old and, after numerous Unity updates, they had become obsolete. Others worked well, but only when using certain 3D file formats (OBJ, STL, etc). Some tools were only able to work inside the Unity editor, and not in real runtime. But nevertheless, the biggest problem of them all was that none of them worked on Android devices, and, since the application was expected to run exclusively in Android devices, that made them useless for this project. Table 2 shows some of these importers.

Importer	Supported file extensions	Platform	Runtime
Runtime OBJ Importer [8]	OBJ	Windows/Mac	Yes
Asset Importer [9]	FBX	Windows/Mac	No
3DS Loader runtime importer [10]	3DS	Windows/Mac	Yes
OBJ Mesh IO [11]	OBJ	Windows/Mac	Yes

Table 2: Import tools from the Asset Store.

It seemed that it was an unsolvable problem, and that the project was destined to fail. But one last search on Unity forums showed, through various user opinions, a specific tool that could be found in the Asset Store: TriLib [12]. Among its technical specifications, the wide variety of formats of 3D files that it admitted was explained. It also seemed to work in both the Unity editor and the actual runtime. And, in addition, the most important feature: it was able to run on Android mobile devices.

The tool also had many functionalities that the project did not need, such as their own rotation and animation play/pause tools, which had already been implemented for this project, the possibility of downloading 3D models from the Internet to import them directly into the application, scripts that modified the behaviour of the Unity editor to carry out tests, and a large etcetera.

Within the technical specifications, a downloadable test version was included for free, which served to verify that, indeed, each and every one of the technical specifications were met.

It was the solution that the project was looking for and, although it was not a free tool, knowing that several weeks had passed by without the project showing a single sign of improvement, it seemed to be the only possibility to save the situation. Not to mention that, after seeing the great performance of the free trial version attached to the technical specifications by the developer, it was clear that the tool was of high quality and that it deserved the 25 dolar price.

Therefore, with an apparently valid solution, priorities were reassigned for the tasks related with the importer implementation process. Since, instead of developing an importer from scratch, an external tool was going to be used, this part lost priority (though not importance). So, now, the implementation of a file browser engine was placed in a higher priority position.

The next step was to prepare the model importer. This implied getting rid of all those features that the tool possessed that were not necessary for the project, as well as making the necessary adaptations for the code to conduct the inclusion of the file browser engine and the 3D models interaction tools that had already been implemented.

And the last step was the serialization and deserialization functionality implementation, which is a very important feature to take into consideration.

3.3.1 File browser implementation

Even having managed to solve the problem with the 3D models importer tool, it was evident that some kind of system was needed to offer the user the possibility of searching, among the files of their mobile device, a three-dimensional model for its later import into his/her personal portfolio.

The basic proposal was to implement a pop-up dialog window that would show the user the hierarchy of directories of their mobile device, so they could navigate between those directories to find a proper 3D model to import. Once found the desired 3D model, its file path would be stored in a text string, which would later be used by the importer to locate and load the model.

Based on this simple idea, three basic elements were clearly differentiated that would establish guidelines for the operation of the file browser engine:

- The files, contained within folders, which were the end goal of the user. Depending on their extension, it would be able to import them or not, in which case it would be appropriate to show an error message.

- The directories or folders, in which files (or more folders) could be found. The user could be located at a specific moment based on the combined path of these directories.

- The parent directories, folders from an immediately superior level to the one in which the user was located. The path of these directories would have to be saved so that it would be possible to go through the hierarchy in reverse.

Once these three elements were differentiated, it was necessary to indicate Unity that the project needed to work with the directory hierarchy of the system, otherwise it would be impossible for the browser to access the files in the memory of the mobile device. This was possible thanks to the Unity IO class, which offers exclusive methods for dealing with files and directories.

It was also necessary to think about the appearance of the file browser engine, and which elements of the interface were going to compose it. Since the file browser was going to be a pop-up dialog window that showed the user a list of selectable options, and that in the directory hierarchy of any device there are usually many folder branches, it was obvious that the list of options could become really long. Thus, the best thing for containing this was, without a doubt, a tactile scroll container, whose content would be a list of touchable elements.

In addition, it also seemed appropriate to add an escape mechanism, so that, in case of human mistake, or in case the user decided to stop searching for a model, the user could close the file search dialog window and perform other actions normally. So, taking all this into consideration, it was decided to start writing the code that would allow the file browser to operate correctly.

In first place, a function for locating the user in an initial directory was created. This function also collected the paths of all the elements contained in that directory, whether they were files or other folders, and created a list within a scroll with one touchable element for each path obtained. This initial directory seemed to change from one device to another, but it was usually established at the root of the internal memory. This is shown in Code 8.

```
//Actualiza los elementos que se muestran en el buscador..
private void ActualizarElementos()
{
    //Se destruyen los elementos previos.
    DestruirElementos();
    //Se crea un elemento para el directorio base.
    CrearElemento(TipoDeElemento.DirectorioPadre, "[Parent Directory]");
    //Se almacenan todos los directorios que contiene el directorio base.
    var directorios = Directory.GetDirectories(_directorio);
    //Y por cada uno de ellos se crea un elemento en el buscador.
    foreach (var directorio in directorios) {
        CrearElemento(TipoDeElemento.Directorio, directorio);
    }
    //Se hace lo mismo con los archivos.
    var archivos = Directory.GetFiles(_directorio, "*.*");
    if (!string.IsNullOrEmpty(Filtro) && Filtro != "*.*") {
        archivos = archivos.Where(x => Filtro.Contains(Path.GetExtension(x).ToLower())).ToArray();
    }
    foreach (var archivo in archivos) {
        CrearElemento(TipoDeElemento.Archivo, Path.GetFileName(archivo));
    }
}
```

Code 8: Update elements function.

So, the next step was to create a function that would make the file browser engine update the list of elements displayed based on the user's selection. The only necessary thing was to take into consideration three possible cases, each corresponding to one of the three types of elements discussed previously:

- In case of having selected a file, it would not be necessary to continue advancing further in the folder hierarchy. Instead, it would be necessary to store the file name of the selected file, combine it with the path of the folder it was located in, and return the resulting path as a text string.

- In case of having selected a folder, it would be necessary to change the path of the parent directory for the path of the current folder, as well as calling the function for obtaining the elements of the directory.

- Finally, in case of selecting the parent directory element, it would be necessary to ascend in the hierarchy, as well as obtaining the path of the parent directory based on the current directory path, in order to call the function again to obtain the elements of a directory.

In short, this function was defined as a simple switch that checked the nature of the element selected (file, directory, or parent directory), and in case it was not a file, it would update the current directory path, as well as the list of elements based on that path. In case a file was selected, it would return its path in a text string. This function is shown through Code 9.

Finally, since the file browser was a pop-up dialog window that was not always going to be visible, two similar, very simple functions, were created that simply activated or deactivated the elements of the interface that composed the file browser engine.

```
//Función que determina qué ocurre al seleccionar un elemento.
public void ControlarSeleccion(TipoDeElemento tipoDeElemento, string nombreArchivo) {
    switch (tipoDeElemento) {
    //Si se pulsa en el directorio padre:
    case TipoDeElemento.DirectorioPadre:
        //Se actualiza el directorio actual y se llama a la función de actualización.
        var directorioPadre = Directory.GetParent (_directorio);
        if (directorioPadre != null) {
            _directorio = directorioPadre.FullName;
            ActualizarElementos ();
        }
        else {
            MostrarNombresDirectorios ();
        }
        break;

    //Si es un directorio lo que se ha pulsado:
    case TipoDeElemento.Directorio:
        //Se actualiza el directorio actual y se llama a la función de actualización.
        _directorio = nombreArchivo;
        ActualizarElementos();
        break;

    //Y si es un archivo:
    default:
        //Se devuelve la ruta completa del archivo y se cierra el buscador.
        ArchivoAbierto(Path.Combine(_directorio, nombreArchivo));
        EsconderBuscador();
        break;
    }
}
```

Code 9: Selection handle function.

3.3.2 Model importer preparation

Preparing the importer to work in perfect synchrony was more complicated than initially expected. Although the tool was very complete, attaching it to the project was not as simple as simply importing the Unity package into an empty scene. It was necessary to study the tool in depth and find out how it worked.

Thanks to the attached documentation, which could be described as a very complete list with explanations of classes and methods (i.e., an API reference), it was possible to understand better the way some scripts interacted with other ones, as well as which ones were vital for the project and which ones were not. Thus, the basic operation of the tool was divided into three different levels.

First, in the most internal level, a script with a huge number of methods worked as a library. All these methods were responsible for carrying out complicated tasks that had to do with the translation of data from a 3D file of known format, to a three-dimensional object with a format that Unity could understand (GameObjects).

For this purpose, other auxiliary scripts were used, located in a second level, which were in charge of modifying the native Unity classes to facilitate the tasks of the library script methods. Some of the modified classes were Transform, which represents the physical appearance of objects in Unity, or 2D texture, which, as its name suggests, is responsible for the correct visualization, storage and reading of the textures of three-dimensional objects.

These modifications were not trivial. They required a deep knowledge of the environment and just by looking at the code of these scripts it could be seen that behind them there was a huge work.

Lastly, in a last outermost level, were the scripts that were in charge of calling the functions of the library script, so that, in a simpler and most superficial way, all the pertinent methods were triggered, so that the file of a 3D model could become part of a Unity scene, like any other GameObject, at runtime.

And so, it was discovered that to adapt the importer to the file browser engine, and to the rest of the tools developed so far, it was in this last layer where the pertinent code had to be written, accessing the library of the innermost layer in a simple and totally parallel manner.

It was also dispensed with all those native features of the tool that were not necessary or important enough for the project, such as the ability to connect to the Internet to download models at runtime, rotation and animation play/pause tools (since they were implemented in the first phases of the project already), modifications for the editor of Unity, and so on.

Taking into consideration everything studied so far, which had taken weeks of work, it became even more evident that it would not have been possible to develop something as complex in the short space of time available for this project, and that using an external tool was the only viable alternative.

So, to adapt the importer to the file browser, the only thing needed was to create a script that could get the path that returned the function of the file browser engine script, to use it as a parameter when calling the model import method, which would trigger a series of internal methods that would eventually result in the import of a 3D file into the current scene.

This model would be the one that would later appear in the viewer scene, where, by using the tools developed in the first phases of the project, it would be possible to interact with it. The preparation of the model, as well as the transition from the importer scene to the viewer scene, will be explained in future sections.

3.3.3 Data saving and loading

Now that both the file browser engine and the 3D models import tool were functional, it seemed that the only remaining task was to program the navigation between scenes. That is, to perform a transition from the scene of the importer to the scene of the viewer and vice versa, to conclude the section of programming. It was even possible to perform the first tests with the importer to verify that it already worked correctly. However, it was still necessary to work on a fundamental aspect: the saving and loading of data.

Even though everything would have worked perfectly, without a data loading and saving system, the user would lose everything he/she had done with the application just after closing it. It was necessary to create some kind of system that could be responsible for keeping a registry of the user's actions, so that, when closing the application and opening it again, it would be possible to find the user's works as he/she had left them.

This involved two things. On the one hand, it was needed to store somehow references to the imported models, so that they could survive the closure of the application and, on the other hand, some type of mechanism responsible for reading these model references so it could be possible to load them later into the application.

Something like this was a problem, since the effects of any operation that was carried out through the scripts of the application in runtime, were only visible through the current execution, disappearing completely with the closure of the application and, to carry this out, it was necessary that the references to 3D models persisted between executions.

A quick search in the official documentation of Unity brought to light a possible solution: what is known as persistent data.

After building an application, when installing it on a mobile device, a folder tree is created within the directory defined on the device for that purpose. These folders contain information about the application, whether it is cache data, user preferences, analytics, and so on. And one of them stores information accessible from the application itself, thanks to the Application.persistentDataPath Unity method, which returns the path of this mentioned folder.

Based on this, it was decided that a good solution would be to save a text file inside this folder, which would contain a list of file paths belonging to the models imported by the user. By doing this, each time the user imported a model, it would simply be necessary to update the list of file paths and proceed to update the text file. And, when opening the application after a closure, it would simply be necessary to read this text file to get the file paths of every previously imported model. Something like this could be achieved through a process of serialization and deserialization.

Serializing and deserializing can be defined as a process of translation and subsequent interpretation. A series of data is converted into an easily understandable format, so that the process can then be carried out in reverse later on. Thereby, something complex like a list, with its elements ordered by indexes, can become a single text string to keep it as persistent data. Later, the interpretation of such text string would result in the same list, intact and ready for its reading.

In this case, it was decided that the optimal option was to use the JSON utility, because it works with text file format and is very easy to use.

First, it was necessary to create the list of routes. This was just a regular list of text strings that was updated every time the user imported a new model, thanks to a simple function called at the end of the import process.

Then, it was necessary to serialize that list and save it in the persistent data, so a function that would take care of the serialization process was written. This function was called every time the list of file paths was updated, either because a new path was added, or because some path was deleted.

And, of course, a homologous function was necessary, which would be in charge of the deserialization process. This function was automatically called whenever the main scene was loaded, in order to keep the file path list constantly updated.

Thus, by means of a simple JSON file in the folder of the persistent data and two simple functions that modified its content, it was possible to guarantee the survival of the user data between executions of the application.

All these functions can be checked through Code 10.

```
// Función que agrega los scripts necesarios al modelo.
public void AgregarComponentes(){
    modelo.AddComponent<RotarModelo> ();
    modelo.GetComponent<RotarModelo> ().velocidadDeRotacion = 4;
}

//Función para almacenar las rutas de los modelos importados.
public void AlmacenarRuta(string ruta){
    if (!rutas.Contains (ruta)) {
        rutas.Add (ruta);
        Debug.Log (ruta);
        SerializarListaDeRutas (rutas);
    }
}

//Función que serializa la lista de rutas de los modelos.
public void SerializarListaDeRutas(List<string> rutas){
    string listaRutas = JsonConvert.SerializeObject (rutas);
    PlayerPrefs.SetString ("rutas", listaRutas);
}

//Función que deserializa la lista de rutas de los modelos.
public void DeserializarListaDeRutas (){
    string rutasJson = PlayerPrefs.GetString ("rutas");
    if (rutasJson != "") {
        rutas = JsonConvert.DeserializeObject<List<string>> (rutasJson);
    }
}
```

Code 10: Saving and loading functions (with helper functions).

3.4 Scene navigation

At this point, the project was divided into two well differentiated parts or scenes (Unity scenes): a scene for the model viewer, where the user could visualize his/her artistic works and interact with them, and another scene for the model importer, where the user could search for 3D files in the folder system of his/her device, to import them into the application later. So, the next step was to implement transitions between these two scenes.

It is obvious to think that the file importer scene should be loaded in the first place, before the viewer's scene, since otherwise there would not be models to show.

Implementing a scene transition in Unity is trivial. Thanks to the existence of the Scene Manager, allowing the change from one scene to another is as simple as writing a single line of code. However, in order to design the scene transitions effectively, it was necessary to think about all the possible situations and how to trigger them, in this case:

- Move from the importer scene to the viewer scene.
- Move from the viewer scene to the importer scene.

In the first case, the transition from one importer scene to the viewer scene would have to be carried out immediately after the model import process, which inevitably involves carrying out the pressing of one of the buttons of the interface.

In the second case, it would be optimal to offer the user the possibility of returning to the importer scene at any time, and the best way to do this seemed to be through a simple back button located on the interface itself. Thus, it was decided that to implement the scene changes, the different buttons of the user interface would be used.

3.4.1 Standard navigation

As stated before, establishing a scene transition in Unity is very easy by using the Scene Manager. However, loading a scene from another one is not the only thing that has to be taken into consideration for this project.

In the scene where the model importer can be found, the file of a 3D model is searched through the folder system of the device, in order to import it as a 3D object into the application. However, when loading a scene, every single element that is contained in the current scene is destroyed, which in this case would imply to destroy the imported model in the process. Thus, the viewer scene would be loaded without a model to show. Some type of mechanism would be necessary to allow the model to survive the transition between the importer scene and the viewer scene.

At first glance, the usage of the method DontDestroyOnLoad, which allows an element inside the game engine to persist in the space even after loading a new scene, seemed a good option. This should allow the model to exist even after a transition between scenes.

However, this solution was not as optimal as first thought, since certain operations with the model had to be performed during the transition between one scene and another, such as creating a quick access button for the model in the importer scene and placing it in a dynamically generated scroll, or nesting the imported model in a hierarchy of objects in the viewer scene and add an Animation component to it, to be able to access their animation later. That is to say: at the time of the transition, the model required to interact with elements both from the importer and viewer scenes at the same time, and this was at first glance impossible, since while one scene was active, the other remained inactive.

Therefore, even the model would be able to persist between transitions using the DontDestroyOnLoad method, it would not be able to prepare it so the user could interact properly with it.

It was necessary to rethink the problem. If ensuring the survival of the model was not the point, since it was necessary to access certain elements from both scenes at the same time, a good proposal could be to store those needed elements in some way at a point that would be accessible from any scene. So it was decided to use a singleton class (that is, a class that refererences to itself) to store those essential elements for the preparation of the 3D animated model.

Within this singleton class, which can be checked in Code 11, functions that stored and returned the imported model were written, so that there was always an accessible reference to it. Functions that added the necessary components to the model were also created, so that when loading the viewer scene all the controls and tools could work properly.

In addition, the singleton class also contained the necessary elements to store the file path of the imported files in a list of strings that would later be used to create the quick access buttons, as well as the serialization and deserialization functions that ensured the correct saving and loading of the data, as well its subsequent reading when opening the application again after a closure, using a self-generated JSON file.

Then, the only thing that had to be taken into consideration was to attach the singleton class to a manager object (which in Unity can be nothing more than an empty GameObject) existing in both scenes, in order to access its content easily and from anywhere, as well as to make this manager object indestructible between scenes. And as explained before, this is easy to achieve thanks to the DontDestroyOnLoad method.

Once the manager object, with the singleton class attached to it, was prepared, the only thing left was to implement the functions that would allow the application to switch between scenes. In the viewer scene, such thing was easy to achieve, since, as stated before, a button was created with the only purpose of returning to the importer scene.

In the importer scene, on the other hand, there is more than one way to trigger a transition to the viewer scene. As explained before, the importer consists of a list of buttons contained in a scroll, whose content is updated as new models are imported into the application. In the first run of the application, there will only be one button, the one that allows the user to import a new model, which will open the file browser so that the user can choose a model to import it. Thus, it will be right after this import process when the transition will happen. But for the other buttons, the quick access ones, it was more complicated to proceed.

```
// Clase singleton para gestionar modelos entre escenas.
public class GestorModelos {
    // Instancia de la propia clase.
    static GestorModelos instancia;
    //Modelo 3D (post importación)
    GameObject modelo;
    [SerializeField]
    //Lista de rutas donde guardaremos las rutas de los modelos importados.
    List<string> rutas;

    // Constructor de la clase.
    GestorModelos(){
        modelo = null;
        rutas = new List<string> ();
    }

    // Función que crea una instancia de la clase en caso de que no haya.
    public static GestorModelos Instancia(){
        if (instancia == null) {
            instancia = new GestorModelos ();
        }
        return instancia;
    }

    // Getter para obtener el modelo.
    public GameObject GetModelo(){
        return modelo;
    }

    // Setter para actualizar el modelo.
    public void SetModelo (GameObject nuevo_modelo){
        modelo = nuevo_modelo;
    }
```

Code 11: Singleton class basic structure.

3.4.2 Quick access buttons

Controlling that, after an import, the model manager picks up the imported model and moves it to the viewer scene, has some degree of complication, but with the singleton class, which supports the communication between scenes, can be achieved, as explained in the previous section. However, this is possible because the import function receives the file path where the model is located, since it is obtained directly from the file browser that opens up when the button that imports a new model is pressed.

However, the quick access buttons do not use the file browser engine: its utility lies in reimporting a model that has already been imported previously, without having to search for it in the system again.

These buttons are automatically created when the application starts, thanks to the serialization and deserialization process. Thus, when the information of the last execution is deserialized, for each file path obtained, a new quick access button is created in the scroll from the main menu. And as these buttons appear, it is necessary to dynamically control that each button is responsible for importing the model associated with it, so that the scene transition can be performed.

And this is where the difficulty of the process lies, since in Unity, the functions that are assigned dynamically to the buttons can not receive parameters. This makes it impossible to dynamically assign functions that receive as a parameter the file path of a three-dimensional model to the quick access buttons, and therefore makes it impossible to perform a transition to the viewer scene, since there is no imported model to visualize.

Some research about the matter on the Unity forums and documentation ilustrated that the only way to solve this problem, that at first glance had no solution, was to make use of lambda expressions. These expressions, which easy usage is exemplified below through Code 12, allowed the dynamically assigned functions to receive parameters for their correct operation.

```
//Función que deja listo un botón para que reimporte un modelo determinado.
void PrepararBoton (GameObject boton, int indice){
    //Índice para recorrer la lista de rutas.
    int indiceLista = indice;
    //Se le agrega la función de reimportación.
    boton.GetComponent<Button>().onClick.AddListener (() => PrepararVentanaOpciones(indiceLista));
    //Se le cambia el texto.
    boton.GetComponentInChildren<Text> ().text = ObtenerNombre (listaRutas [indiceLista]);
}
```

Code 12: Lambda expressions usage sample.

And by doing so, the import function could now be assigned to the quick access buttons, receiving the file path of their associated 3D model, obtained thanks to the deserialization process, as a parameter, and thus allowing the model to be passed to the manager, so the scene transition could be performed.

3.5 Visual design

Once the programming tasks were finished, it could be affirmed that the application was already functional. However, the need to work on certain visual elements, in order to provide the application with a more professional and attractive look, became evident. This involved getting rid of the Unity native 2D elements, such as the default sprites of the buttons and sliders, whose appearance was very flat and basic, and substitute them by other ones designed in line with the general idea of the application.

To fill this gap, it was necessary to design and elaborate a user interface, to facilitate the global visualization of the application and allow the controls to be executed in a simple and comfortable way.

In addition to the creation of a user interface, it was also necessary to work on different three-dimensional elements, such as scenarios, which would be used as dynamic backgrounds for the 3D model viewer, as well as models to test the implemented tools, which implied designing their shape and also animating them.

It should be noted that, except for the animation tasks, the methodology to be followed for modeling the backgorunds and standard models was going to be the same, so, to avoid explaining the same process several times, the modeling process of a standard model will be used to explain the procedure.

Thus, it is concluded that the design process included the following phases, which will be explained in detail in the following sections:

- User interface design: 2D design process through which it is intended to obtain an appropriate user interface for the application, starting with a simple sketch, and improving it iteratively until the ideal result is obtained for its export and subsequent inclusion in the application.

- 3D model design: design process whose objective is to generate three-dimensional objects for illustrative and test purposes, and which includes tasks such as modeling geometry, texturing it and animating its components.

3.5.1 User interface design

The design of the user interface is a really important step in the development of any application. Not only for the visual appearence, but for what is known as user experience.

The user experience can be defined as a series of factors that the user perceives through interacting with an application, whose results directly influence the conception that the user generates about the used application, which can be positive or negative.

Through the interface design process, it is intended to make the perception of the user as positive as possible, although the visual appearence is not the only element involved: it is also important that the controls respond well, that the application execution is fluid, that the tools provided are useful and pleasant to use, and so on.

It is also an iterative process. It starts with a very schematic initial design which is very susceptible to future changes and, based on this scheme, something bigger is gradually built up, substituting the basic ideas by more developed ones, until the final result is achieved.

Given the general theme and the main purpose of the application, it was decided that the user interface had to be clean and simple, without shocking details that could cause the user look away from what actually mattered: his/her own work. Following this general idea of design, the main proposal was the use of subtle gradients of dark and pale colors, since, in general, the interfaces with excessively clear and saturated colors are uncomfortable to look at. For shapes (buttons, containers, sliders, windows, etc.), simplicity would also be used: straight lines and slightly rounded corners, since this type of compositions transmit professionalism and cleanliness.

Good examples of interfaces with all these characteristics are Krita's [13], a free 2D design program with a very friendly interface for the designer, and Artstation's [14], a web page where artists of different artistic genres can post their works and see other people's, whose use is quite standardized given the professionalism it transmits.

So, to start with the process, the first step was to design a basic scheme or sketch that showed the basic layout of the elements of the interface, taking into account the functionality of each one of them. That is, a mockup.

3.5.1-A Mockup elaboration

The elaboration of the mockup is an essential step in any project of this type. It is not only useful to form a basis on which to start designing the user interface, but it is also really useful during the initial phases of the project to elaborate a Design Document, where an idea of what the final result, in terms of appearance and functionality, is intended to look like, is shown.

This scheme does not have to be necessarily detailed, as shown in Figure 12. With a few lines and scribbles it is more than enough to capture the general idea, as can be seen in the first sections of this report, where the design document prepared for this project is shown.

Figure 12: Mockup sample images.

Nowadays, there are numerous programs whose purpose is to facilitate the development of the mockup, allowing to capture the general idea of an application for mobile devices in a very short amount of time, for example, Balsamiq Mockups 3 [15], the tool used in this project.

3.5.1-B 2D design process

Once the mockup was designed, the next step was to start designing the elements of the interface.

To do this, using the scheme obtained by making the mockup as a base, it was needed to draw the elements of the user interface in a digital image editing program dedicated to artistic design tasks.

Normally, to carry out this type of tasks, the most common and recommended is to use vector image editing software, such as Adobe Illustrator [16] or Inkscape [17], since they allow the images designed and edited to be scaled to any size without loss of quality . This makes them optimal for designing interfaces for mobile applications, since a different screen resolution is used for each device.

However, given the unique and prototypical nature of the project, it was decided that the best option was to renounce the diversity of resolutions to avoid losing more time than necessary. So, it was decided to use a raster image editing tool (Krita) instead, since its use is usually more comfortable and faster.

To avoid suffering loss of quality due to the scaling of images, it was decided to elaborate the interface designs to a rather large size (1920 x 1080). By doing this, the need to enlarge images was eliminated, which usually causes a loss of image information much higher than the fact of reducing its size.

Once the elaboration of the user interface elements was finished, the only thing left to do was to save each and every one of those elements as individual images with an appropriate format, through a process known as exportation process.

3.5.1-C Exportation process

The process of exporting the elements of the interface was slow and tedious, and more things had to be taken into account than initially expected.

To begin with, each element of the user interface had to be on separate layers, so if during the visual design process of the user interface this had not been taken into account, it was necessary to separate them one by one, which in many cases meant having to design one or more elements again.

Once separated by layers, each one of the elements was copied and pasted into a new document whose size was adapted to that element, so that, when saving the image, the resulting element had the correct size and not the complete interface (that is, so that the size of a simple button was not 1920 x 1080 pixels, where the majority of the pixels were empty pixels).

Once an element was separated and adjusted its size in pixels, it had to take into account the use that was going to be given to that element.

In the case of a panel, or a background, without rounded corners and without the need for transparencies, it was already possible to proceed with the saving of the image, in JPEG format. This choice was made based on the fact that the JPEG format does not support transparencies and occupies less space in memory.

Otherwise, if it was a button, an icon, or any other element of reduced size and that used transparency, it was necessary to carry out a few more operations:

- First of all, to avoid display problems in Unity when minimally scaling this type of elements, it was convenient to leave a margin of one pixel around the original image. This prevented the edges of the element image from distorting.

- Afterwards, it was necessary to choose an appropriate format for the export of the element. If it was a small element, with low variety of colors, an appropriate format was PNG-8, since this format saves space in memory in exchange for offering a lower variety of colors. If, on the other hand, the element was more detailed and used a larger color palette, the appropriate format would be PNG-24, although this would mean that the element would occupy a larger space in memory.

Once the appropriate format was chosen, it was finally possible to proceed to save the image. At the moment of saving it, it was also necessary to follow a certain methodology, which consisted in naming the elements following normalization rules.

This is not important in small projects where only one developer works, but in larger projects with a larger number of collaborators, it is crucial that graphic elements have a descriptive and easily archivable name, so it is advisable to follow this methodology, anyway.

Once this process was finished, it was necessary to repeat it from the beginning for each and every one of the elements of the user interface that had been previously designed.

3.5.2 3D model design

The objective of the user interface design process was to create interface elements that fit the general purpose of the application and that would provide the user with the most positive perception regarding the use of the tool.

The 3D design process, on the other hand, had the objective of generating three-dimensional virtual objects to include them later in the application. Thus, by including such 3D elements, it would be possible to test the capabilities of the application and the tools and functionalities implemented throughout the development process in a real case.

The generation of a three-dimensional model suitable for inclusion in the application went through three basic phases:

- Geometry design: the process by which the appearance of the model is shaped. This is achieved through the combination of different basic geometric elements using a series of tools provided by a 3D editing program, which results in a mesh of polygons.

- Texturing: the process by which the geometry of the model is assigned color and texture. For this, the geometry is divided into different parts separated by lines of union that define areas on which the textures will later be placed.

- Animation: the process by which the three-dimensional model ceases to be a static object and comes alive. The methodology to be followed is based on the principles of classical 2D animation, which used the superposition of frames to simulate movement.

In the following sections, each one of these phases will be explained.

3.5.2-A Geometry design

The design of the geometry is the simplest of the three phases, although certain aspects have to be taken into account so that they do not interfere with the procedure of the other phases.

To carry out the modeling process, a three-dimensional cube was used as starting base. Any basic primitive could have served to start: a cube, a sphere, a cylinder or even a plane. On this three-dimensional primitive a series of operations that aimed to change its shape and appearance were carried out.

In 3D Studio Max these operations are carried out at different levels, depending on the element on which is wanted to work, through tools that the program makes available to the designer.

There are other tools that allow operations on all geometry, and not only on the most basic elements that make it up. These tools are called modifiers, and cause noticeable changes to the final geometry of the model, such as causing all polygons in the mesh to be subdivided, or giving the model a smoother, more rounded final appearance without adding more polygons.

Given the large number of tools and modifiers that exist, not only in 3D Studio Max, but also in any other modeling program, explanations about most of them will be omitted, referring only to those most commonly used during the design process.

Thereby, among the most common tools for working geometry at the polygon level, the "extrude" tool was used, which generated new polygons over the selected ones and allowed to modify their length. It was also quite common to use the "bevel" tool, which worked in a similar way to extrusion, but also allowed adding a bevel on the edges, or the "inset" tool, which caused a polygon to contain a smaller version of itself inside.

At the edge level, it was very common to use tools to generate more edges around a primitive, connecting some edges with others, such as the "connect" tool, or the "bridge" tool.

And at the vertex level, the most used tool was undoubtedly "weld", which allowed merging several vertices into one. However, given the spatial characteristics of the vertices, the most common thing was to perform movement operations on them to reposition them in the three-dimensional space.

Once the mesh of the model was generated by editing the basic primitive using those tools, it had to be taken into consideration that the mesh met certain requirements.

First, the overall size of the polygons should be uniform. Otherwise, at the beginning of the animation process, the difference in size between different polygons could result in overlaps or display errors. In addition, a mesh which polygons have a uniform size is considered cleaner and more orderly.

Secondly, the polygons of the mesh had to be correctly connected to each other. This implied that there were no T vertices and that the normals of all the polygons were oriented in the same direction. Figure 13 shows the basic appearence of a T vertex.

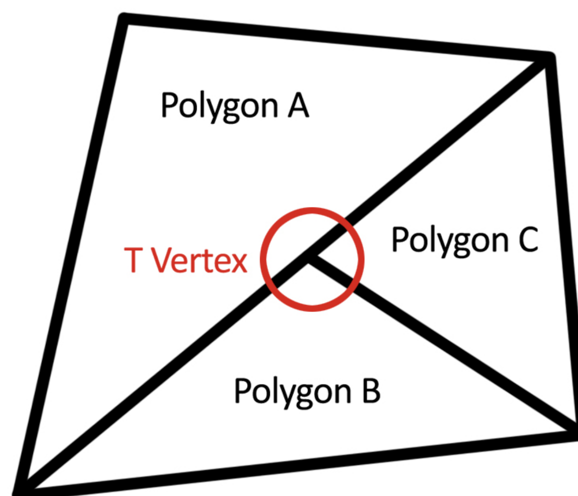


Figure 13: Graphic definition of a T vertex.

This would ensure the correct visualization of the model, allow the textures to be displayed correctly on the model and facilitate the animation process.

3.5.2-B Texturing

Once the mesh of a model is finished, the texturing process can be carried out.

This process can be defined as a sewing process, in which the geometry is divided into parts, on which a texture will be coupled, to subsequently sew all the parts together, as if it were cloth patches. So, the first step to texturize the model was to mark the seams by which the mesh would later be divided. This was achieved thanks to the UVW Unwrap modifier.

Figure 14 shows the difference between a model without textures and ist textured version.

Figure 14: No-textured model (left) versus textured model (right).

Once these seams were defined, the model was divided into different parts, which were flattened and placed neatly on a white texture. Through this process, the texture coordinates were defined. Thus, the empty texture with the parts of the model worked as a template, so that when drawing on that texture, it was actually drawing on the part of the model associated with the texture coordinates. Relationship between the template texture and the final texture is shown through Figure 15.

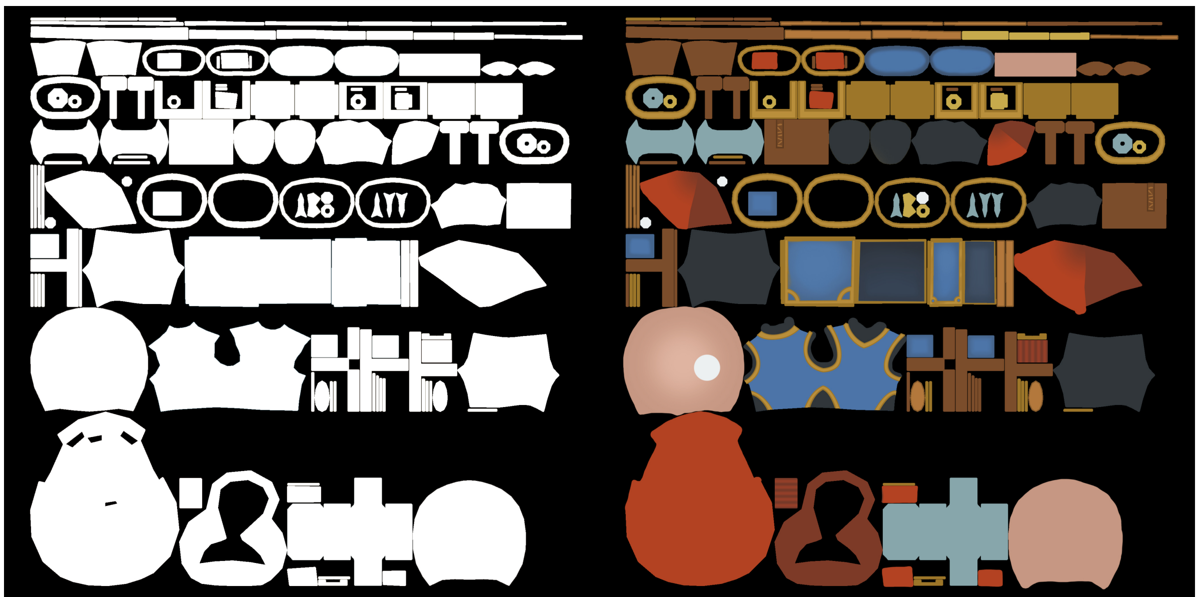


Figure 15: Template texture (left) and final texture (right).

3.5.2-C Model animation

The last phase, the animation process, was undoubtedly the most difficult and laborious. As mentioned before, the methodology to follow is the same as the one used to animate cartoons, which consisted of overlapping frames to simulate movement. Of course, this concept must be transferred to the three-dimensional environment.

In the same way that, in the past, characters were drawn in various poses and then each pose was overlapped, with a 3D model the same has to be done. Through skeletal animation, it is possible to generate a three-dimensional hierarchy of bones, so that each part of a model can be assigned to one of those bones. Thus, by moving these bones, the associated part of the model would also move accordingly.

Although it is true that 3D Studio Max offers the possibility of using predetermined biped skeletons (i.e., with an already created hierarchy of bones), these skeletons are somewhat limited in terms of movement freedom and number of limbs, so in most cases, the best option is to create a skeleton from scratch using the bone creation tool. This is more laborious, but in turn leads optimal bone hierarchies for the animation process.

These hierarchies establish father-son relationships between bones, which determine the degree of freedom that a bone has with respect to the rest. These relationships can be compared to the ones between the bones of a human skeleton: although when moving the fingers of one hand the position of these fingers change in space, when moving an entire arm the position of the fingers also changes, unavoidably. In other words, the movement of a bone causes the bones in its charge (that is, its children) to move as well.

So, in the first place it was necessary to define an appropriate skeleton. This involved designing a bone structure that fit the basic characteristics of the model.

En caso un modelo humanoide, estas características básicas son las proporciones y el número de extremidades. De este modo, si un supuesto modelo tiene unas piernas cortas, los huesos correspondientes a las piernas deben ajustarse a la longitud de estas; si tiene cuatro brazos, el esqueleto debe tener un conjunto hombro-brazo-antebrazo-mano para cada uno de ellos; si tiene cola, una serie de huesos debe prepararse para que esta pueda retorcerse apropiadamente; si tiene el pelo largo, deben acoplarse a los huesos de la cabeza una serie de huesos que puedan simular el movimiento del cabello, etcétera.

In case of a humanoid model, these basic characteristics are the proportions and the number of limbs. In this way, if a supposed model has short legs, the bones corresponding to the legs must adjust to its length; if it has four arms, the skeleton must have a set shoulder-arm-forearm-hand for each one of them; if it has a tail, a series of bones must be prepared so that it can be properly twisted; If it has long hair, a series of bones that can simulate hair movement must be attached to the bones of the head, etc.

Furthermore, if the character carries some kind of object, such as a hat or cane, bones must also be added for these objects. However, being independent objects that can be freely separated from the main body, those bones will be outside the main hierarchy as well.

Once a valid and appropriate skeleton was prepared for the model, the next step was to establish a relationship between the different parts of the model and the bones of the skeleton, so that, by moving these bones, the associated parts of the model would move accordingly.

3D Studio Max ofrece diferentes modificadores para llevar a cabo este proceso, como "Physique" o "Skin". "Physique" funciona creando áreas llamadas envelopes alrededor de los huesos, de modo que, al mover un hueso, todos los vértices del modelo que se encuentren dentro de su envelope se moverán en consecuencia. "Skin" por el contrario, emplea una tabla de relaciones, donde se establece un peso entre 0 y 1 para cada vértice asociado a un hueso, de modo que los vértices cuyo peso sea 1, seguirán el movimiento del hueso con exactitud, y cuanto menor sea este valor, menor será la influencia del hueso sobre los vértices.

3D Studio Max offers different modifiers to carry out this process, such as "Physique" or "Skin". "Physique" works by creating areas called envelopes around the bones, so that, when moving a bone, all the vertices of the model that are inside its envelope will move accordingly. "Skin" on the other hand, uses a table of relations, where a weight between 0 and 1 is established for each vertex associated with a bone, so that the vertices whose weight is 1, will follow the movement of the bone with accuracy, and the lower this weight, the smaller the influence of the bone over the vertices.

Even though the first is intuitive and easy to use, the second can achieve much more precise results, so it was decided to use the "Skin" modifier. Thereby, a table was created that contained all the vertices of the model and all bones of the skeleton, and a weight was defined for each possible relation.

Once the weights were assigned for all the vertices of the model, it was already possible to start designing the animation.

A good methodology to be followed is to use references. Watching videos of a specific movement or animations made by other artists is a great help to make more complete and natural animations. A very useful tool in this matter is Mixamo [18]. Mixamo is a website where it is possible to quickly and easily animate a humanoid 3D models, thanks to the large library of predetermined animations that it offers. For an animator, it is also a very useful source to find a lot of varied humanoid animations, being able to use them as references to elaborate their own animations.

It is also common to use storyboards, the sketching on paper of the different keyframes of the animation. They are very useful to establish a general guide of how the animation will flow. Once a simple storyboard has been elaborated and enough references have been collected, it is now possible to start animating the model.

Moving, rotating and scaling the bones of the skeleton, the model was made to pose. Each pose was recorded in a different frame of a time bar, so that, when advancing through the time bar, a frame transition was caused, which made it look like the model was actually moving. Figures 16 and 17 show how a model can be posed to generate several frames.

A quick export in FBX format, would result in a 3D file that contains both the geometry, textures and animations of the model inside, ready to be imported into the application to be viewed and archived.

Figure 16: Geometry (left) and skeleton (right) in a standing pose.

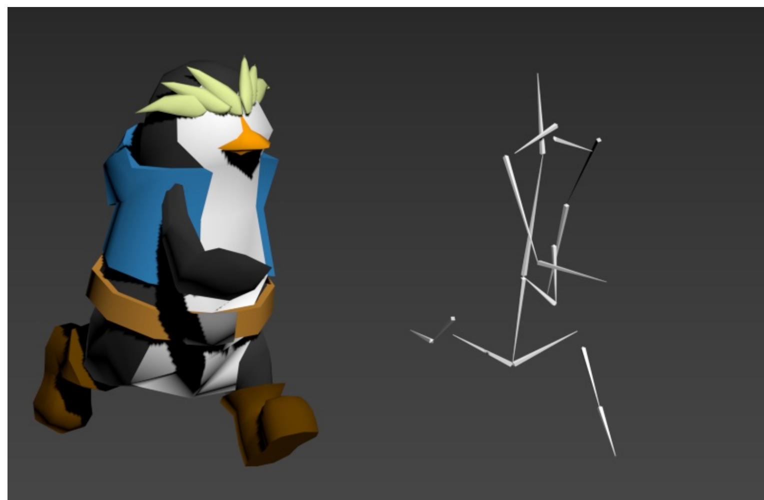


Figure 17: Geometry (right) and skeleton (left) in a running pose.

4 Results

4.1 Introduction

In this section, the obtained results after completing the whole process of design and development of the application are shown.

It is worth to mention that, as explained before, the developed application has two basic functionalities. On the one hand, the possibility of allowing the user to import his/her own animated three-dimensional models from the memory of his/her device into a 3D scene to visualize them. On the other hand, a series of tools that the user will be able to use to interact with his/her imported models, in order to create visual compositions by editing the basic parameters of the 3D scene.

Thus, these two basic functionalities are going to be analyzed to check if the obtained results meet the expected results. In addition, a video of the global result, where the final appearence and operation are shown, is attached, as well as the links to the 3D animated models made for this project.

4.2 3D models import functionality

The 3D models importer had to be an easy to use tool that, with just a few taps on the screen, would allow the user to import his/her own animated models into a 3D scene to visualize them. This involved taking two basic concepts into consideration: on the one hand, the possibility of locating 3D files in the memory of the device and, on the other hand, importing them just by tapping them once located.

The first concept was covered by implementing a simple file browser, whose final appearence is shown in Figure 18. Although the main idea was to build a more complex browser, as can be seen in Figure 5, it was decided to make mechanincs as simple as possible, so that the user could not be overwhelmed by excessive options. The resulting browser proved to be both efficient and easy to use.

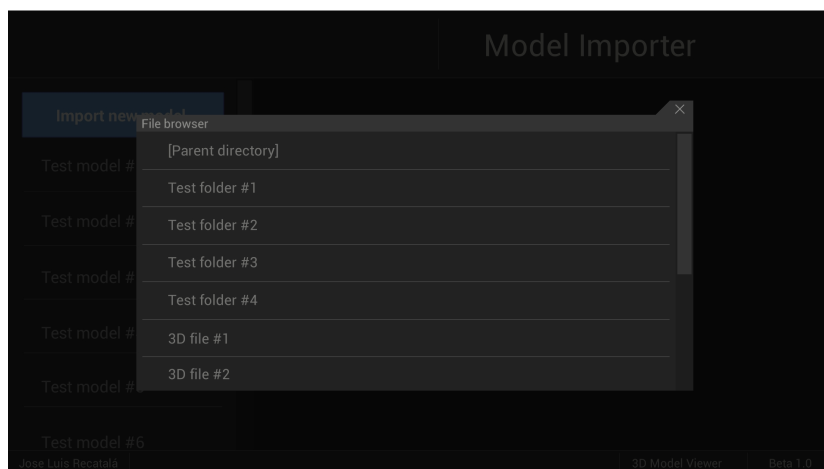


Figure 18: File browser final appearence.

The second concept was covered by implementing a list of models, as seen in Figure 19. Such list contained the models imported by the user, and by selecting one of them the user was able to either delete it or visualize it again. Thus, by just tapping on the desired model of the list, the user had full control over it, making it very easy to visualize models and keep the list clean and updated.

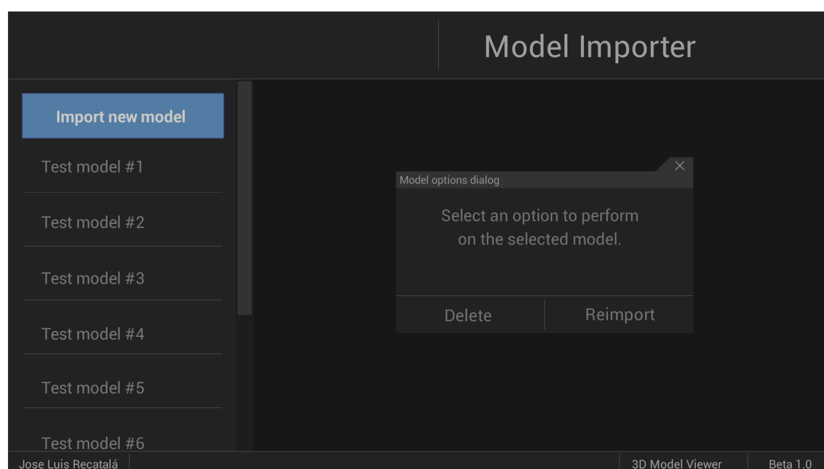


Figure 19: Main screen final appearence.

4.3 Interaction tools

The interaction tools had to allow the user to change the global visualization of the 3D scene where the model was. The tools had to be intuitive and easy to use. This implied putting special efforts on the implementation of the controls.

Since the tools had different purposes and not every one of them could be operated by tapping the screen, it was decided to make some tools operable through buttons, but keeping in mind the simplicity. Thus, while rotation and zoom tools were implemented to be operated by tapping and dragging across the screen, the other tools (play/pause animation, background change and screenshot) were designed to be operated through buttons. Figure 20 shows the final appearence of the 3D model viewer.

The resulting tools proved to be intuitive and easy to operate.

Figure 20: 3D model viewer final appearence.

4.4 Final results

By following the next link, a video that shows the final result of the application, both in terms of appearence and functionalities, can be found: https://drive.google.com/open?id=1nf3qEJvaKNCH9PAksefBT0ohnQjgj4Kx .

The models used to test this project can also be found by following the next link: https://drive.google.com/open?id=1pK2KukqN0yLNlqRMocrWrqkLAXTHl3t1 .

Finally, the application installer can be found by following the next link: https://drive.google.com/open?id=1UJVIDQG6MSlPFjLSvJ3ok0Enwnv6lFiY .

5 Conclusions

5.1 Introduction

During the initial phases of the project, specifically during the elaboration of the Technical Proposal, some goals or objectives were established, which had to be fulfilled after the end of the application. These objectives were directly related to the application, either by the very fact of completing its development, or by specific characteristics that it should offer to the user.

This section checks wether the objectives established during the initial phases of the project have been met or not, and gives a general conclusion about the work done during the whole project

5.2 Visualization of 3D models

The first of the objectives established that an application should be developed for allowing the user to visualize three-dimensional models through an Android mobile device.

As seen in the development section, the resulting application allows not only to visualize 3D models, but also to interact with them to visualize them from different perspectives in order to better appreciate details of their shape and appearance, thanks to the rotation and zoom tools. Moreover, if it is an animated model, it will also be possible to visualize its animation.

With all this, it can be said that the first objective has been fulfilled.

5.3 Scene composition and exportation

The second objective required that the possibility of recreating dynamic scenes composed by a model and a background or scenario was offered to the user, and that, once these elements were organized to their liking, they could make a screenshot of his work.

Thanks to the background change tool and the screen capture tool, whose implementation has been detailed in the corresponding sections of the development section, it can be affirmed that both functionalities have been made available to the user, so that he/she can make different compositions with his/her own work that can later be saved in an easily archivable and shareable image.

Thus, it is concluded that, indeed, the second objective has also been met.

5.4 Model import and creation of customized portfolios

The third and final objective established that the user had to be able to import his/her own models into the application, and that he/she should be able to store them in some way so that he/she could elaborate a personalized portfolio of animated 3D models.

Thanks to the model import tool and the implementation of a data storage and loading system, the resulting application allows the user to import various models from his/her device, and is responsible for keeping a registry of each one of them. In this way, after closing the application, this system is responsible for reading the registry so that it is possible to access again the previously imported models.

Thus, it can be said that the third objective has been fulfilled.

5.5 General conclusions

The idea of offering the user the possibility of importing his/her own models and sharing them in some way changed everything. Something like that arose from the personal need of a 3D artist to show his/her work at any time and place. Such a feature converted a simple circumstantial application into a possible useful tool for a 3D artist.

It also turned a relatively simple development process into a much more complex one, which needed a lot of documentation about programming aspects and the Unity game engine itself. But, step by step, the project grew with effort and perseverance.

Without a doubt, the fact of not being able to implement an own import tool is a failure, but something like that also helps to want to learn more, to not fall into the same fate in the future. And, in the same way, all the mistakes made during the project have led to learning, providing new knowledge and experiences.

The final result is better than expected and, in addition, each and every one of the proposed objectives has been met. It is concluded that the project of developing of a hand-held portfolio for three-dimensional animated models has been a resounding success.

6 References

[1] Overwatch – Hero gallery review:
https://youtu.be/rQ0PuyUOWA4?t=17m30s

[2] Super Smash Bros. Melee/Brawl – Trophy gallery reviews:
https://youtu.be/DjeUtKEjzo0?t=21s
https://youtu.be/O7n7CMYQIag?t=2m42s

[3] Lolking – 3D Champion models viewer examples:
http://www.lolking.net/models?champion=78&skin=0
http://www.lolking.net/models?champion=113&skin=0

[4] Unity Engine – Official webpage:
https://unity3d.com

[5] Monodevelop – Official webpage:
https://www.monodevelop.com

[6] 3D Studio Max – Official webpage:
https://www.autodesk.es/products/3ds-max/overview

[7] LibreOffice Writter – Official webpage:
https://es.libreoffice.org/descubre/writer

[8] Runtime OBJ Importer – Unity Asset Store:
https://assetstore.unity.com/packages/tools/modeling/runtime-obj-importer-49547

[9] Asset Importer – Unity Asset Store:
https://assetstore.unity.com/packages/tools/utilities/asset-importer-9224

[10] 3DS Loader runtime Importer – Unity Asset Store:
https://assetstore.unity.com/packages/tools/3ds-loader-runtime-importer-62536

[11] OBJ Mesh IO – Unity Asset Store:
https://assetstore.unity.com/packages/tools/modeling/obj-mesh-io-15862

[12] TriLib – Unity Asset Store:
https://assetstore.unity.com/packages/tools/modeling/trilib-unity-model-loader-package-91777

[13] Krita – Official webpage:
https://krita.org

[14] Artstation – Webpage:
https://www.artstation.com

[15] Balsamiq Mockups 3 – Download page:
https://balsamiq.com/download

[16] Adobe Illustrator – Webpage:
https://www.adobe.com/es/products/illustrator.html

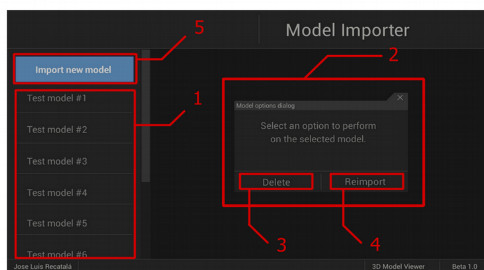
[17] Inkscape – Official webpage:
https://inkscape.org

[18] Mixamo – Webpage:
https://www.mixamo.com

Appendix 1: User's Guide

The objective of the developed application is to visualize 3D animated models, as well as their manipulation through internal tools to alter the visualization and generate results to share them later. Although the full development of the application has been detailed throughout the main document, it seems necessary to prepare a document to explain, step by step, how the final product works. This appendix is attached as a brief user's guide, through which it is intended to explain in detail the operation of the application. It is worth to be remembered that the application works only on Android devices.

The operation of the application is simple and easy to master thanks to the tactile controls, which allow to use all the features just by tapping on the screen a few times. After opening the application, the start screen will be displayed, which will contain the main menu. The following diagram briefly details each one of the elements of the menu.



1 Model list

2 Model options dialog

3 Delete option

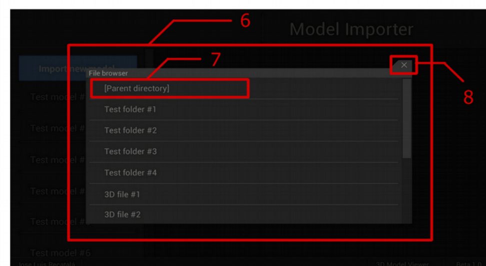
4 Reimport Option

5 Import new model button

6 File browser

7 Parent Directory

8 Close button



- In the **model list (1)** the models that have been previously imported are shown. If this is the first time the application is executed, this list will be empty.

- Tapping any of the models of this list will bring up the **model options dialog (2)**. This window offers the possibility of deleting the selected model, by choosing the **"Delete" option (3)**, or visualizing the model again, by choosing the **"Reimport" option (4)**.

- To import a new model, the **"import new model" button (5)** must be pressed. Pressing this button will display a simple **file browser (6)**. This file browser shows the files located in the memory of the mobile device, allowing to search for the 3D files to import.

- The navigation through the file browser is performed just by tapping the name of the desired folder. By doing this, the elements shown in the file browser will be updated, displaying the files and folders contained in the selected folder. A previos folder can be accessed again by tapping on **"Parent Directory" (7)**.

- Tapping the **"X" button (8)** in the upper right part of the dialog will close the file browser and show the main menu again.

By tapping on the name of a 3D file displayed in the file browser dialog, the selected file will be added to the model list in the main menu, and imported into the model viewer scene. The following diagram shows the elements that can be found in the model viewer scene.

1 Pause/playback button

2 Frame slider

3 Playback speed buttons

4 Screenshot button

5 Background button

6 Back button

- Using the **pause/playback button (1)** the animation of the imported model (if any) can be played or paused.
- Through the **frame slider (2)** the current frame of the animation can be modified. This slider can only be operated if the animation is paused.
- By tapping the **playback speed buttons (3)** the rhythm at which the animation of the model is played can be altered.
- The **screenshot button (4)** takes a picture of the current state of the scene (without including the user interface) and saves it in the image gallery of the device.
- The **background button (5)** causes the background scenario, over which the model is displayed, to change.
- The **back button (6)** closes the model viewer scene and displays the main scene again.

It is worth mentioning that the application lacks an escape mechanism to close its own process, so to close the application, it is recommended to use the process manager of the mobile device itself.

NOTE: *This version of the application is a prototype (3/6/2018), so, in future versions, important changes in its operation, as well as the implementation of new features, could be found.*