TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

ePuppet – A Mobile Application that Automates and Expands the Digital Figure Creation Process for the eShadow Platform

Πολυτεχνείο Κρήτης

Koukis Alexandros

Table of Contents

Abstract..4

Chapter 1: Introduction.....................................5

1.1 Traditional Shadow Theatre...5

1.2 Digital Shadow Theatre..5

1.3 ePuppet..7

1.4 Tackling the Problem...8

1.5 Thesis Outline..12

Chapter 2: Technological Background.............13

2.1 Mobile Applications..13

2.1.1 Web Applications...13

2.1.2 Native Applications..14

2.1.3 Hybrid Applications...14

2.2 Framework Selection..15

2.2.1 Software Stack...15

2.3 Image Processing...19

2.3.1 Image as a 2D Array of Pixels..20

2.3.2 Figure/ Background Distinction...20

2.3.3 Rectangle Packing..21

2.4 Project Structure, Typescript and Plugins......................................22

2.4.1 Project Structure...22

2.4.2 Typescript Core Concepts..24

2.4.3 Native Storage..24

2.4.4 Device Camera...25

Chapter 3: Functional Requirements and Use Cases...27

3.1 Functional Requirements...27

3.2 Use Cases..28

3.3 Personas...30

Chapter 4: Implementation..............................33

4.1 Architecture..33

4.2 Application Flow and Activities..36

4.2.1 The Home Page..38

4.2.2 Puppet Options Page..41

4.3 Image Data Manipulation..43

4.3.1 Background Transparency..44
4.3.2 Selecting Parts..46
4.3.3 Selecting Joints...49
4.3.4 Final Editing Page...53
4.4 Other Challenges...55
4.4.1 Base64 Images and Blob Data Form....................................55
4.4.2 Uploading Files on the Cloud...56
Chapter 5: Experimental Evaluation and Real Use Cases..59
5.1 System Usability Scale Questionnaire..59
5.1.1 Experimental Process...60
5.1.2 Results..60
5.2 Real Use Cases...62
5.2.1 Kindergartens, Thessaloniki...62
5.2.2 Christmas Edition...64
Chapter 6: Conclusion..66
6.1 Contributions...66
6.2 Future Work...67
References...68
Glossary of Acronyms...70

Abstract

This thesis presents the design, implementation and evaluation of ePuppet, a mobile application that expands the use of eShadow, a digital storytelling platform inspired by the traditional Shadow Theatre, addressing creative expression and learning in the field of cultural heritage. eShadow enables the creation of digital stories within a project-based approach that may start from scenario development and include the creation of digital puppets and sceneries, the set-up and recording of story scenes and the final assembly of a digital story. eShadow supports an inclusive learning framework that enables teachers to design and implement interdisciplinary theme-centered projects in a variety of subjects (language, history, music, computing, etc.) to help students develop cross-curricular (horizontal) skills: speaking and writing skills, social cooperation and research skills, digital and cognitive skills.

This ePuppet mobile application that is presented in this thesis enables the users of eShadow to develop their own digital puppets in a way that incorporates the creative process of developing traditional shadow theatre puppets and import them for further use in eShadow. This is achieved by digitizing pictures and drawings through a mobile device's camera. ePuppet offers two main modes of editing: The first is based on certain templates and facilitates the creation of digital puppets that have a certain structure (i.e. 2-, 3- or 4-part puppets) following the corresponding structure of Greek traditional shadow theatre puppets. The second one enables users to create any kind of puppet with any number of constituent parts.

ePuppet is built as a Hybrid Mobile Application using the Ionic Framework for theming and prototyping, Apache Cordova as the native wrapper, granting access to the device's native features such as the Local Storage and the Camera and Angular 2, which is a set of TypeScript libraries that implement core and optional functionality for mobile applications. The image processing is executed inside the HTML 5 canvas element using the WebGL2.0 JavaScript API as it enables GPU-accelerated usage of image processing and effects as part of the Web Page canvas.

Following the design and implementation of ePuppet, a thorough experimentation was undertaken to confirm its usability. The evaluation results show that the mobile application is highly usable and can effectively support the task of digital puppet creation in order to enhance the eShadow experience.

Keywords: shadow theatre; mobile application; learning; education; figure; camera; digital puppet; analog puppet;

Chapter 1: Introduction

1.1 Traditional Shadow Theatre

Shadow theatre is a storytelling tradition using flat articulated puppets which are held between a light source and a translucent screen or scrim. The cut-out shapes of puppets may include translucent color or other types of detailing. Shadow theatre is popular in various cultures in more than 20 countries, mainly in Asia. In Greece it used to be very popular at the end of the 19th and beginning of 20th century in both urban and rural areas. Although its popularity declined as cinema and TV invaded the Greek society, it still remains a favorite entertainment for many children and adults and a way for personal expression as a means of dramatized storytelling (Moumoutzis et al., 2019).

Shadow theatre is a medium with significant educational value within the wider context of drama and performance arts. This is linked to the ability of shadow theatre to activate people and promote their creativity as they engage in diverse activities during the preparation and the actual performance of shadow theatre plays. In particular children and adults find their own ways to act and imitate, create dialogues, get inspired and convey their own messages, direct, become stage designers, sing, strengthen their self-confidence giving life to the puppets, improvise and create their own stories. Thereby they cultivate their oral speech skills and develop in multiple modes their intelligence (multiple intelligences) in an entertaining manner. In addition, they get familiar with the creative inquiry-based process of scenario development and subsequent artistic work to prepare puppets and sceneries employing painting and handicraft. Finally, they are given the opportunity to discover identities of peoples and regions through the musical, linguistic and cultural particularities of certain shadow theatre characters (Hatzigianni et al., 2016).

1.2 Digital Shadow Theatre

Digital theater and puppetry platforms offer a new approach on the traditional Shadow Theatre, in part due to the creative potential of such software for entertainment purposes, but more importantly, due to the educational value. Firstly, applications of this kind offer ample room for creative expression. The setting can range from traditional or cultural events to studying modern day problems, providing teachers with the flexibility to adapt performances based on specific learning needs of their students. Additionally, since these platforms enable collaborative game-like interactions, they have great appeal among younger students.

Following these ideas, TUC/MUSIC Lab has developed eShadow (http://www.eshadow.gr/) with the aim to digitize and modernize the traditional Shadow Theater. eShadow is being used in elementary schools across Greece and beyond. It is a modernized version of the traditional Shadow Theater and provides an alternative to traditional storytelling techniques. With eShadow, the user can create digital shadow theater plays, record and share them. It also provides the ability for live communication and collaboration over the Internet.

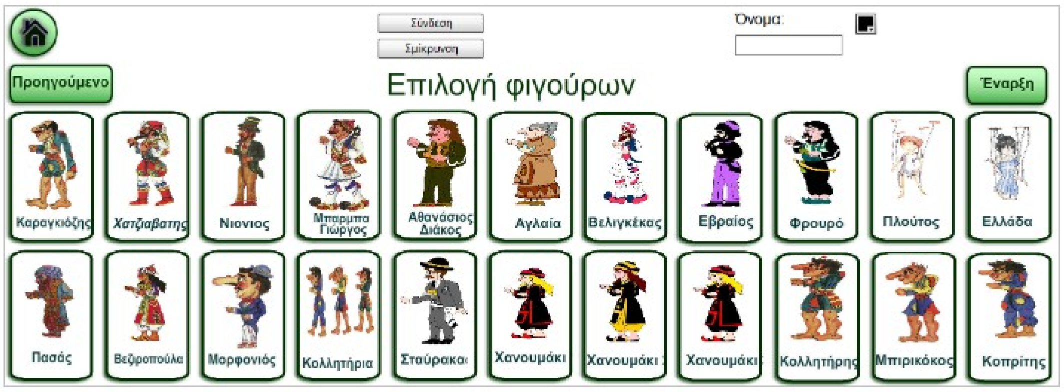


Figure 1: Selecting digital puppets in eShadow.

The goal of eShadow is to become a tool of creative expression and learning. It is being used by dozens of teachers, mainly in elementary schools, to create performances on topics such as economic crisis, environmental protection and folk songs. It is also used to create educational performances for learning basic concepts of mathematics and computer science (Christoulakis et al., 2013).

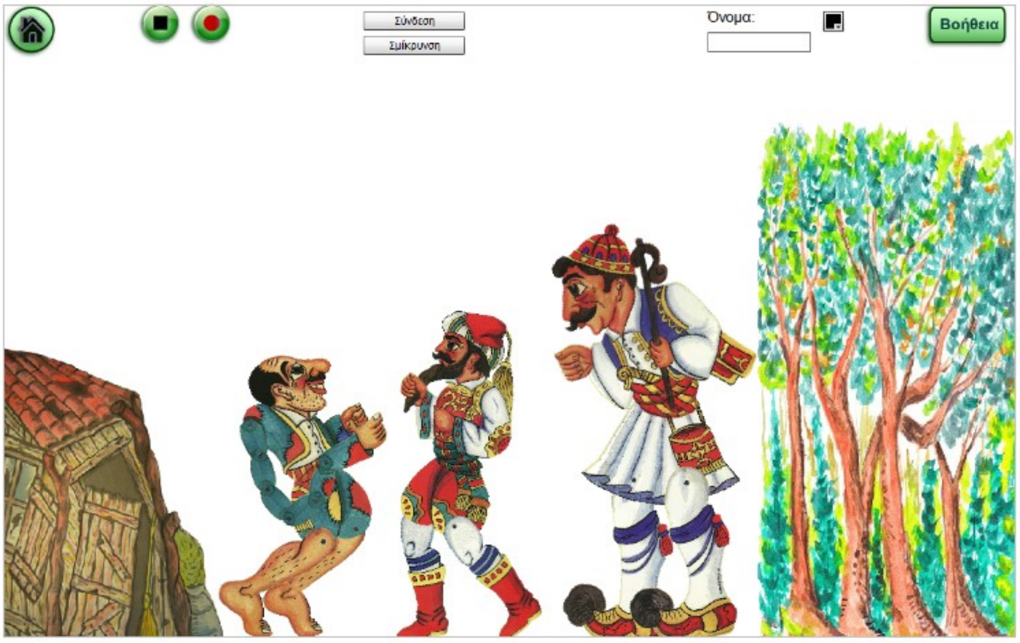


Figure 2: Enacting a scene with three puppets and two sceneries in eShadow.

Although the use of the platform has been successful, there is one profound limitation: The ability to create new figures and sceneries without the required knowledge attached to the process, both graphical and technical. Considering that the average user is either a teacher or a student, it is unrealistic to expect the know-how to create a figure from scratch and import it in eShadow. As a result, users are limited in their choice of content by the themes that come with the installation or they rely on more knowledgeable peers in case they would like to extend the offered library of digital puppets and sceneries.

The first attempt to overcome this limitation was eShadow Editor, an application created by A. Moraiti in her diploma thesis with the intent to create custom digital figures for further use in eShadow. The application was thoroughly tested and used across schools and institutions (Moraiti et al., 2016) but was still restrictive in the sense that figures could be created from a fixed set of building blocks inside the program. Digitizing analog puppets was not supported. These restrictions limited the educational scenarios that the editor could support and were the inspiration for the functionality of ePuppet developed in this thesis.

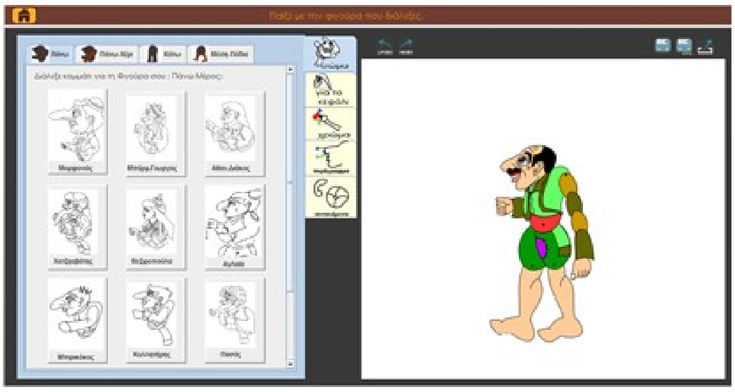


Figure 3: Remixing a digital puppet in eShadow Editor.

1.3 ePuppet

The aim ePuppet is to offer an effective mobile solution that is based on integrating the digital and analog worlds and support teachers and students that wish to work on traditional shadow theatre and transition to the digital version in a smooth and pedagogically rich manner.

The application was designed as a mobile app for several reasons. The process of digitizing an analog figure requires capturing an image of this figure, which can only be done through mobile devices, especially in a class setting. Given that a mobile device has to be used for this step, we decided that any additional processing should be performed on the same device in order to offer more flexibility to the users. Additionally, mobile applications are easier to install, maintain and are preferred by users in contrast to their desktop counterparts.

The goal of ePuppet is to primarily provide an all-in-one solution for end users to digitize, customize and import custom figures and scenery in the eShadow platform. The most important qualities of ePuppet are ease of use, minimal design and as little user input as possible in order to streamline the process.

1.4 Tackling the Problem

Transforming analog puppets into digital is a very broad problem that can be solved in different ways. The choice usually lies on the digital format we would like to achieve at the end of the digitization process. However, given that the end target is eShadow, we had to examine what exactly is a digital puppet in the context of eShadow. The structure of digital puppets is very similar to the structure of traditional shadow theater puppets. In particular, they consist of two or more parts which are joined together by joints. Each part An example can be seen below:

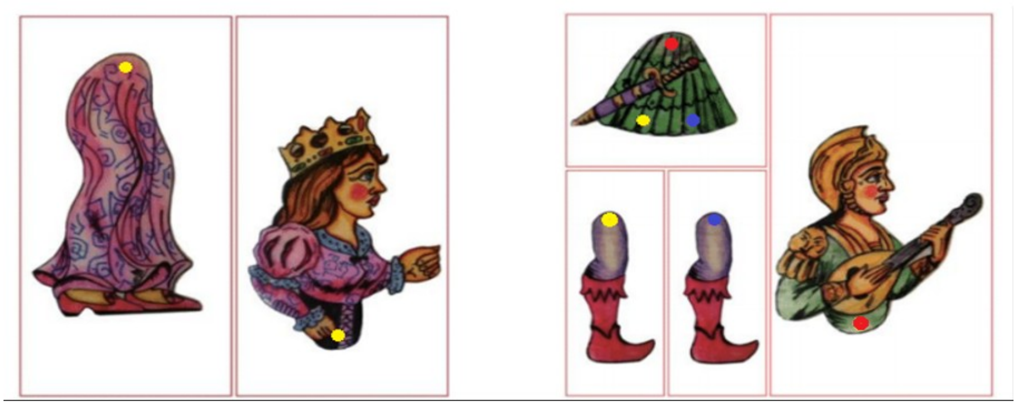


Figure 4: Synthesis of shadow theatre articulated puppets. In both cases the red rectangles represent the boundaries of areas used to place each constituent part.

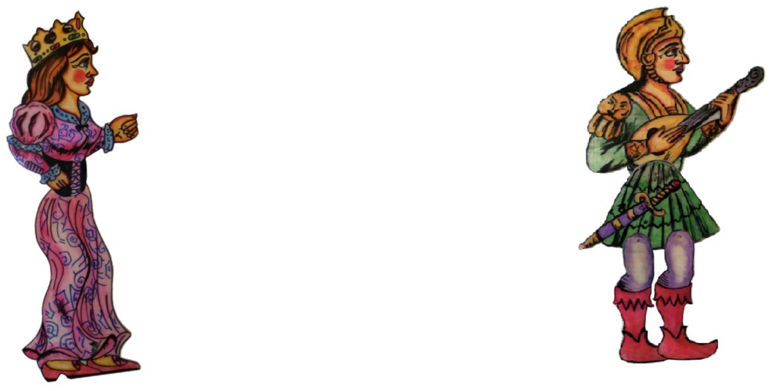
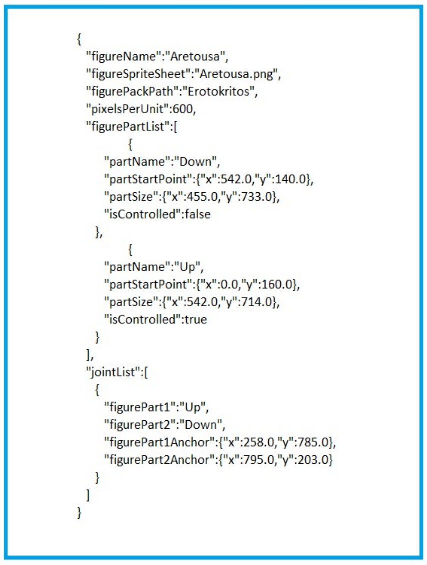


Figure 5: The final output of the above process.

In order to tackle the problem at hand, we first have to break down what a figure is, in the scope of eShadow. eShadow is built on Unity 3D (https://unity.com/), and a figure basically consists of three separate files:

- The thumbnail file, used both in eShadow and ePuppet in order to indicate which figure we are referring to. It is a png image file that contains the figure data after applying the necessary transparency to the background but before we perform any additional editing.
- The Sprite Sheet, which is a 1024x1024 pixel PNG formatted image consisting of the individual parts of the figure, contained in non overlapping rectangles so that eShadow can use pixel coordinates to display each separate part correctly.
- The JSON file, which contains all necessary data regarding the figure. In the header, the figure name, relevant sprite sheet and scale (pixelsPerUnit) are set. This part is followed by two lists, each containing every part and joint that is necessary to reconstruct the image in eShadow. The structure of each object in the part list is the following: the name that must be unique in order to identify each part for the joint list, the starting coordinates in pixels on the Sprite Sheet, the size of the part in pixels and a flag that determines if the part can be controlled in eShadow (is clickable). The structure for the joint list contains information for the two parts that will be joined (see part list description), and the anchors that refer to the coordinates of the joint point on each of the two parts.



```
{
  "figureName":"Aretousa",
  "figureSpriteSheet":"Aretousa.png",
  "figurePackPath":"Erotokritos",
  "pixelsPerUnit":600,
  "figurePartList":[
          {
     "partName":"Down",
     "partStartPoint":{"x":542.0,"y":140.0},
     "partSize":{"x":455.0,"y":733.0},
     "isControlled":false
    },
          {
     "partName":"Up",
     "partStartPoint":{"x":0.0,"y":160.0},
     "partSize":{"x":542.0,"y":714.0},
     "isControlled":true
    }
  ],
  "jointList":[
    {
     "figurePart1":"Up",
     "figurePart2":"Down",
     "figurePart1Anchor":{"x":258.0,"y":785.0},
     "figurePart2Anchor":{"x":795.0,"y":203.0}
    }
  ]
}
```

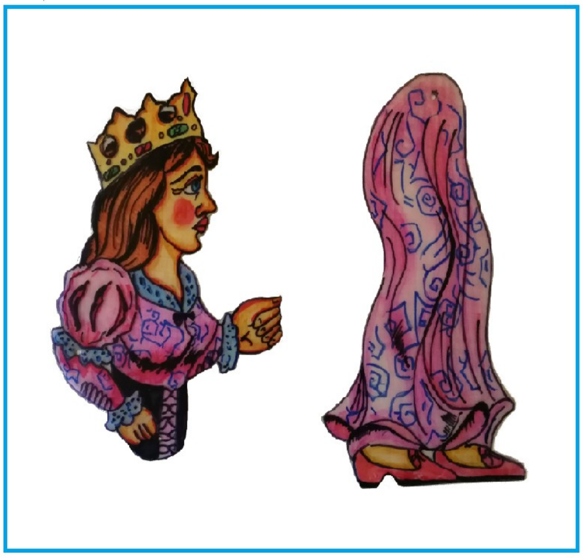
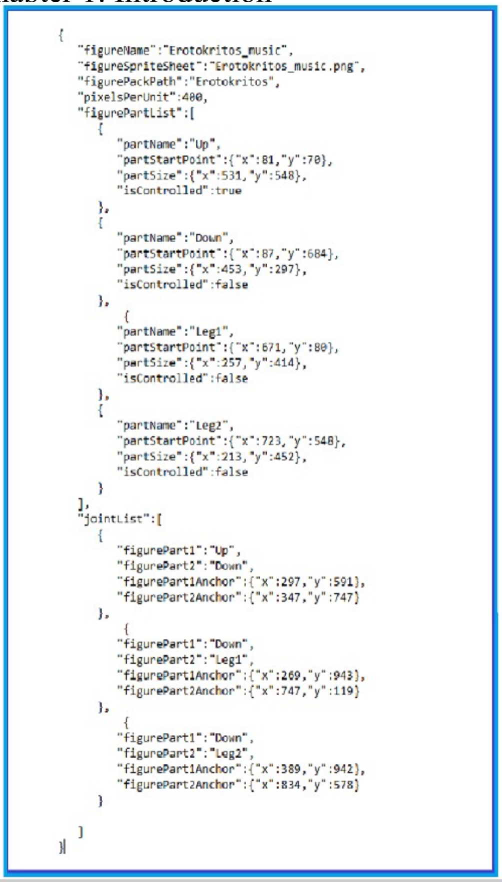


Figure 6: Representation of a 2-part digital puppet in eShadow. On the right, the sprite sheet of the 2-part puppet is presented, while on the left, the relative JSON description is depicted.

The figures below depict the JSON files and the corresponding sprite sheets for the 2-part and 4-part puppets presented above:



```
{
   "figureName":"Erotokritos_music",
   "figureSpriteSheet":"Erotokritos_music.png",
   "figurePackPath":"Erotokritos",
   "pixelsPerUnit":400,
   "figurePartList":[
      {
         "partName":"Up",
         "partStartPoint":{"x":81,"y":70},
         "partSize":{"x":531,"y":548},
         "isControlled":true
      },
      {
         "partName":"Down",
         "partStartPoint":{"x":87,"y":684},
         "partSize":{"x":453,"y":297},
         "isControlled":false
      },
          {
         "partName":"Leg1",
         "partStartPoint":{"x":671,"y":80},
         "partSize":{"x":257,"y":414},
         "isControlled":false
      },
      {
         "partName":"Leg2",
         "partStartPoint":{"x":723,"y":548},
         "partSize":{"x":213,"y":452},
         "isControlled":false
      }
   ],
   "jointList":[
      {
         "figurePart1":"Up",
         "figurePart2":"Down",
         "figurePart1Anchor":{"x":297,"y":591},
         "figurePart2Anchor":{"x":347,"y":747}
      },
          {
         "figurePart1":"Down",
         "figurePart2":"Leg1",
         "figurePart1Anchor":{"x":269,"y":943},
         "figurePart2Anchor":{"x":747,"y":119}
      },
          {
         "figurePart1":"Down",
         "figurePart2":"Leg2",
         "figurePart1Anchor":{"x":389,"y":942},
         "figurePart2Anchor":{"x":834,"y":578}
      }

   ]
}
```

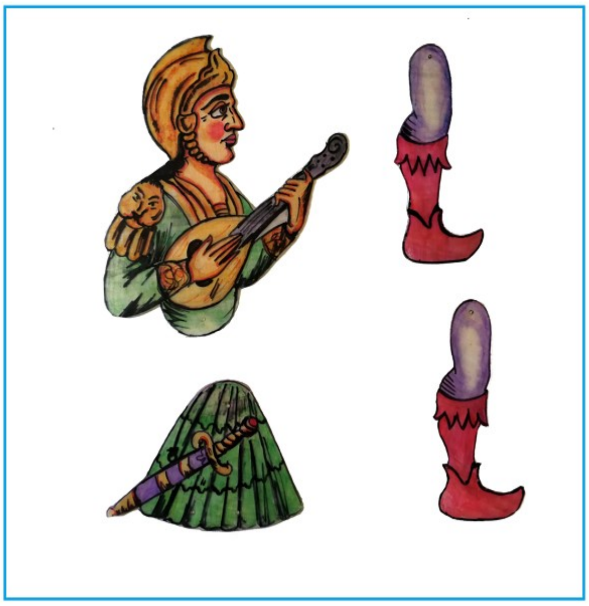


Figure 7: Representation of a 4-part digital puppet in eShadow. On the right, the sprite sheet of the 4-part puppet is presented, while on the left, the relative JSON description is depicted.

To synthesize such puppets, one has to develop their parts in certain material, cut them and join them together at the correct joint positions. The female puppet consists, as mentioned, of two parts which are associated with one joint. These two parts are illustrated separately in Fig. 6 as well as their connecting points (joint anchors) to form the joint of the puppet. The selection of connection points, especially for the bottom part affects not only the aesthetics of the puppet but also its behavior during handling due to the force of gravity.

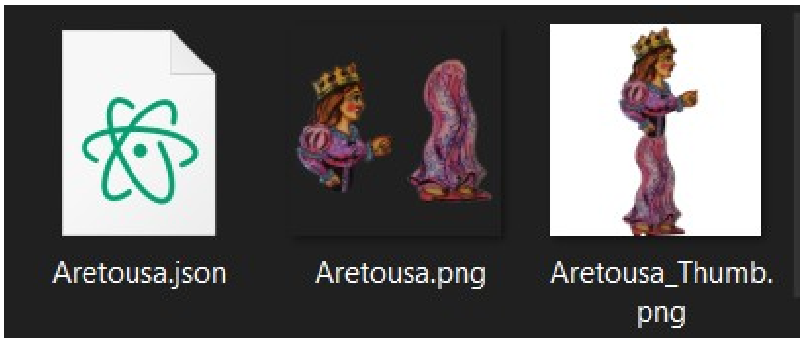


Figure 8: Any eShadow figure consists of these three files.

As seen above, the output of ePuppet should be a set of three files named accordingly, so that Unity 3D can utilize them further. Given that the input is always a captured image from the phone's camera, any additional information the app needs in order to generate the aforementioned files properly, must come in the form of user input. The following information is essential for the figure creation process and must be explicitly conveyed to the application, since it is not included to, or inferred from, the captured image:

- The name of the figure, which will be used for all three files needed in eShadow (see 'FigureName' field in the JSON file).
- Which pixels are actually part of the figure depicted in the image, as any pixels outside of those must be converted to transparent (see 'FigurePartList' field in the JSON file).
- The pixel coordinates of the parts in the original image where the figure is dissected, which will then be used to construct the sprite sheet (see 'FigurePartList' field in the JSON file).
- The pixel coordinates of the joints between parts on the sprite sheet, in effect, pairs of (x, y) coordinates that indicate where each part joins with another (see 'JointList' field in the JSON file).

This information is included in the JSON file. This file consists of three main areas. It may come from either analog or digital means. A point can be made for both cases, and thus, both ideas are supported. In the full digital approach, the user inserts a name for his new figure, then captures an image of it on a distinct, mono color background. They then proceed to split the figure in parts with touch gestures as well as create the specific joint points of each part.

However, given that the user base of this application is very diverse, the digitization process can be simplified further, to the point of only requiring a single click. Users can prepare the figures manually, by cutting each part on the paper and then placing those parts on a distinct, mono color background. Then, using the available 2, 3 and 4-part templates as overlays on the camera, they can calibrate the parts so that they fit in the overlay boxes and then simply capture the image. The application automatically assumes that each box contains a part and each set of same colored dots is a joint. This method does restrict the diversity of the figures though, since the templates are limited and must have specific sizes.

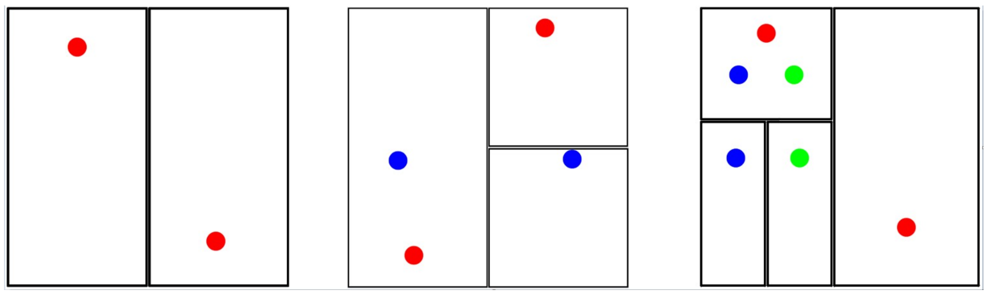


Figure 9: Available templates in ePuppet: These templates cover 2-, 3- and 4-part figures that are cut to fit the capturing boxes above.

1.5 Thesis Outline

In the following chapters of this thesis the framework that was followed is thoroughly explained. The first two chapters have an introductory and theoretical tone. The rest of the chapters focus on the implementation process along with the software used to accomplish the final result, along with the testing process and user evaluation.

The second chapter provides an insightful review of technologies used in modern web applications. Furthermore, it analyzes how object recognition and pixel data manipulation is performed today. Finally, a review and comparison of the most powerful frameworks is provided. The chapter also explains which technologies of the above are used in this project and why.

The third chapter showcases the steps we undertook in order to specify the MVP (Minimum Viable Product), by defining the functional requirements of ePuppet with the help of Use Cases and Personas.

The fourth and most extensive chapter is dedicated to presenting fundamental parts of this application. The flow within the application is presented in detail and noteworthy coding snippets are displayed and explained.

The fifth chapter provides information regarding the evaluation process that took place throughout development. It reviews different techniques used in order to receive feedback, their effectiveness, along with any enhancements that were made after receiving feedback.

In the sixth chapter the conclusion of the whole development process is presented, along with thoughts and suggestions about future extensions. Finally we present the list of references used throughout this document, and a glossary of acronyms.

Chapter 2: Technological Background

This chapter presents the scope of ePuppet along with important design decisions that were made prior to its implementation. We will start with State of the Art technologies on web application development frameworks in Section 2.1 and proceed to present the software stack that was used throughout the ePuppet implementation in Section 2.2, justifying every choice in detail. Additionally, in Section 2.3, we will delve into image processing techniques and background detection algorithms that were considered as potential solutions for the background extraction process and the sprite sheet creation. These choices are the foundation of the application and defined the development process significantly. Finally, in Section 2.4 we will present some core Typescript concepts and will provide on overview of the plugins we used in our implementation.

2.1 Mobile Applications

A mobile application, also referred to as a *mobile app* or simply an *app*, is a computer program or software application designed to run on a mobile device such as a smart phone, a tablet, or a smart watch. Mobile applications often stand in contrast to desktop applications which are designed to run on desktop computers, and web applications which run in mobile web browsers rather than directly on the mobile device.

In modern day development, there are three distinct mobile application categories, each with its own set of advantages and disadvantages. Thus the choice usually lies within the nature of the problem at hand. In the following sections we will present the relevant State of the Art technologies and explain the thought process behind every choice in the software stack.

2.1.1 Web Applications

By definition, a web application is an application that is accessed via a web browser over a network such as the Internet. They differentiate from websites in the sense that they offer additional functionality and interactivity, however the line between the two can be vague.

Developing a web app can be simple and quick. It’s often a good way to test out an idea before investing in developing a native mobile app. However, if the web app is relatively simple and designed for desktop users over mobile, a large portion of the user base is excluded.

Until recently, web apps lacked the functionality of native apps, like the ability to send push notifications, work offline, and load on the home screen. However, there have been a few improvements to browsers and web apps that offer these features over the past years. Apps that take advantage of these features are called Progressive Web Apps (PWAs). It should be noted that they still lag behind in functionality compared to mobile applications are currently mostly used for prototyping purposes or websites with enriched functionality.

2.1.2 Native Applications

A native application, is a software application built in a specific programming language, for the specific device platform, either iOS or Android. Native iOS apps are written in Swift or Objective-C and native Android apps are written in Java. Native applications are downloadable from their respective app stores. Due to the fact that the development environment is specifically tailored to the needs of the platform, they come with significant advantages over the alternatives:

- **Fast and Responsive**

  Native apps offer the fastest, most reliable and most responsive experience to users. Many elements come preloaded and the user data is fetched from the web rather than the entire application, providing great offline functionality. This is unlikely to change in favor of web applications.

- **Access to wider device functionality**

  Native application development comes with out-of-the-box support for device features such as the camera, compass, microphone, etc. While the alternatives also offer similar functionality, it is often less responsive and documented, which significantly increases development time.

- **UI/UX matching to platform conventions**

  Native applications are, in essence, extensions of the device's default applications and thus, offer a native feel by emulating the experience of those default apps. Familiarity is imperative for the success of an application as users tend to respond negatively when alienated.

2.1.3 Hybrid Applications

A hybrid app is a software application that combines elements of both native and web applications. Hybrid apps are essentially web apps that have been put in a native app shell. Once they are downloaded from an app store and installed locally, the shell is able to connect to whatever capabilities the mobile platform provides through a browser that's embedded in the app. The browser and its plug-ins run on the back end and are invisible to the end user.

Hybrid apps work similar to Web apps but like native apps, are downloaded to the device. Similar to Web apps, hybrid apps are typically written in HTML5, CSS and JavaScript. Hybrid apps run code inside a container. The device's browser engine is used to render HTML and JavaScript and native APIs to access device-specific hardware. Although a hybrid app will typically share similar navigation elements as a Web app, whether or not the application can work offline depends on its functionalities. If an application does not need support from a database, then it can be made to function offline.

Over the recent years, hybrid applications have taken over the mobile app industry due to their adaptability and speed of development. They offer unique advantages over the alternatives:

- **Cross Platform – One Codebase**

This is the main reason to use hybrid applications over any alternative. It cuts down on development time significantly as it is possible to reuse a big portion of the codebase. In effect, the application is written once regardless of the target platform and is then wrapped accordingly to target the required platforms. Based on the aforementioned, hybrid applications are preferred over native ones if the goal is to either reach the end user faster, reach a wider range of users, or both.

- **Maintenance**

Similar to the above point, hybrids applications are easier to maintain and update over the course of their life-cycle both due to the fact that all platforms operate on a single codebase and because web technology is simpler than native app technology in general.

2.2 Framework Selection

Since the aim of ePuppet is to support and expand the existing eShadow software, rapid development and testing is essential. Additionally, since the target audience includes students and teachers, a cross platform approach allows for branching to a wider user base. Those requirements point towards a Hybrid App framework. Furthermore, the nature of the project calls for minimal UI styling which further enforces the hybrid application approach.

Due to the preceding reasons, the Ionic Framework (https://ionicframework.com/) was chosen for the creation of the application as it offers a complete hybrid app development kit. In addition, it is open source, well documented and widely used. The specific collection of components used to support the execution of the application, also called the software stack, will be presented in the next section.

2.2.1 Software Stack

A software stack is a collection of independent components that work together to support the execution of an application. The components are stacked one on top of each other in a hierarchy. Typically, the lower level components in the hierarchy interact with hardware or the back-end, while the higher level components in the hierarchy perform specific tasks for the end user. Components communicate directly with the application through a series of complex instructions that traverse the stack.

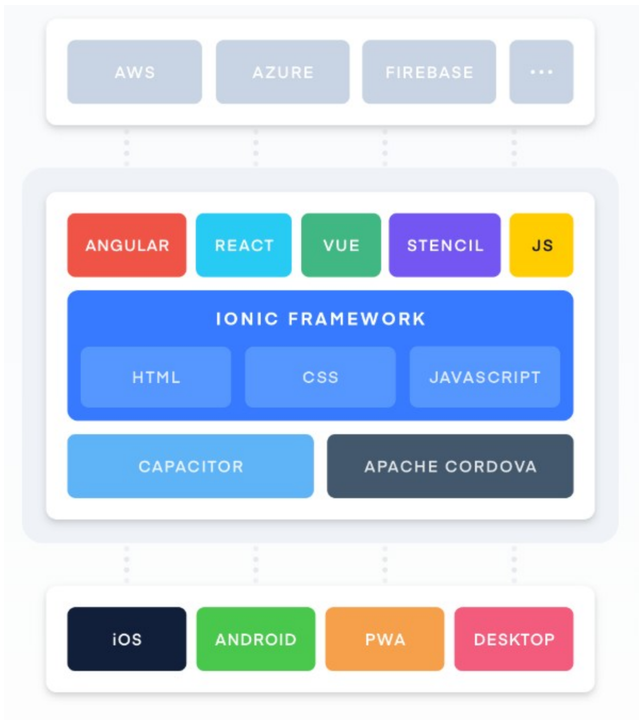


Figure 10: Software Stack

In the case of Ionic, a wide array of technologies can be combined to produce hybrid mobile applications as shown in the following figure. Apache Cordova is a framework that runs JavaScript apps in a Web View which has additional native extensions. This is what essentially makes Ionic applications hybrid. From the beginning of Ionic, Apache Cordova has been an integral part of the project. Cordova applications use less and more simple code, which iterates faster development and execution. Recently Ionic has launched their own framework called Capacitor that is meant to replace Apache Cordova in their software stack in the future. Both Capacitor and Cordova manage a Web View and provide a structured way of exposing native functionality to your web code. However, Capacitor is still in its early testing phase so we decided on the more mature choice for a production level application.

A brief comparison for each component library will be provided in the following pages. It should also be noted that, since the local device storage has been used for all data storing purposes in ePuppet, any comparison for back-end components will be omitted.

- **React** (https://reactjs.org/)

  React, developed by Facebook, was initially released in 2013. Facebook uses React extensively in their products. It combines the UI and behavior of components. In React, the same part of the code is responsible for creating a UI element and dictating its behavior. The library has a fixed overhead of 116 KB per application regardless of the app's requirements.

- **Angular** (https://angular.io)

  Angular, developed by Google, was first released in 2010. It is a TypeScript based JavaScript framework. A substantial shift occurred in 2016 on the release of Angular 2 (and the dropping of the "JS" from the original name – AngularJS). Angular 2+ is known as just *Angular*.

  In Angular, components are referred to as directives. Directives are just markers on DOM elements, which Angular can track and attach specific behavior too. Therefore, Angular separates the UI part of components as attributes of HTML tags, and their behaviors in the form of JavaScript code. This is what sets it apart when looking at Angular vs React. Angular is not a library like React but a framework in itself and that comes with stricter rules and more out-of-the-box features.

- **Vue** (https://vuejs.org)

  Vue, also known as Vue.js, was developed by ex-Google employee Evan You in 2014. Over the last three years, Vue has seen a substantial shift in popularity. Vue 3, currently in alpha phase, is planning to move to Typescript.

  In Vue, UI and behavior are also a part of components, which makes things more intuitive when looking at Vue vs React. Also, Vue is highly customizable, which allows you to combine the UI and behavior of components from within a script. Further, you can also use pre-processors in Vue rather than CSS, which is a great functionality. Vue is great when it comes to integration with other libraries, like Bootstrap. The library has a fixed overhead of 91 KB per application regardless of the app's requirements.

When considering which option to choose, the main criteria used were:

- Load Times
- Overhead
- Community Support and Documentation availability
- Maturity
- Compatibility with HTML5 canvas for image processing purposes

Both Angular and React are ahead of their competition currently, as the two most mature and well documented choices in the field. They come with powerful debugging tools, a plethora of components and efficient load and build times. Both offer a vast knowledge base from passionate developers. However, Angular allows for a tailored bundle size, specific to the application on demand, without carrying unnecessary overhead. This is imperative, considering that the app size and load time were a priority during development.

Additionally, Angular separates the behavior of a component from its UI which has proven to be extremely useful when dealing with custom elements such as the HTML5 canvas. The most important difference, however, and the one that influenced our choice the most was how data binding is performed on each case. Data binding consists of the data synchronization process between Model and View in the MVC design pattern. Angular uses a **bidirectional data-binding** process and mutable data. This means that when changes in the UI's input occur, then the model state will also change and vice-versa. Contrarily, React works with a **unidirectional (or one-way) data-binding** process. Thus, a UI element modification does not change the state of a component. In the case of ePuppet, two way interaction between the UI and the model is imperative to achieve the image processing functionality required.

It should be noted that Angular requires a deeper understanding of DOM elements and their handling as it completely separates the two. It also has a steeper learning curve over its competitors but this was not an issue during development due to previous familiarity and experience. Below we present the final Software Stack at the time of writing this thesis.

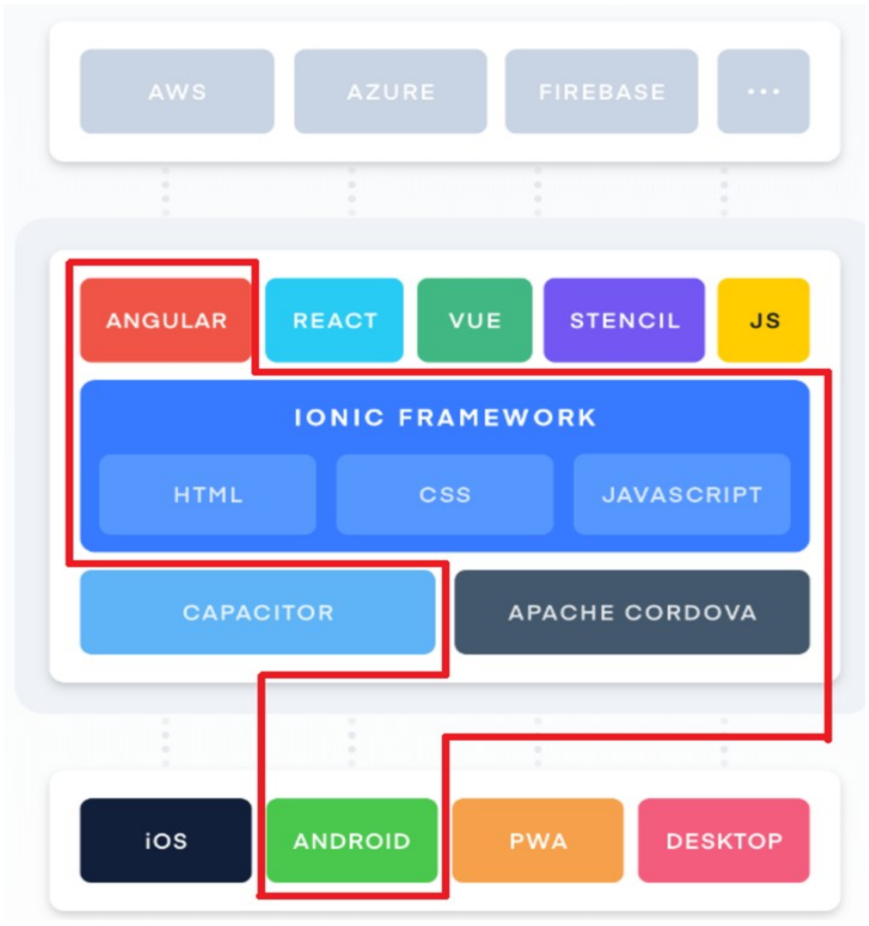


Figure 11: The final software stack is highlighted in red.

2.3 Image Processing

Image processing is a method to perform some operations on an image, in order to get an enhanced image or to extract some useful information from it. It is a type of signal processing in which input is an image and output may be image or characteristics/features associated with that image. Nowadays, image processing is among rapidly growing technologies. It forms core research area within engineering and computer science disciplines too.

Image processing basically includes the following three steps:

• Importing the image via image acquisition tools.

• Analyzing and manipulating the image.

• Output in which result can be altered image or report that is based on image analysis

Having set the fundamentals regarding the mobile web application design decisions in the previous sections, the image processing aspect of the project is next presented. Because ePuppet needs to process photographs of drawings so that it can isolate the parts of a figure, we had to solve how to detect which pixels are background and replace them with transparent pixels, and how to arrange the parts in order to fit in a sprite sheet the same size as the original image, also called rectangle packing. Below, we present pictures of the aforementioned practices.

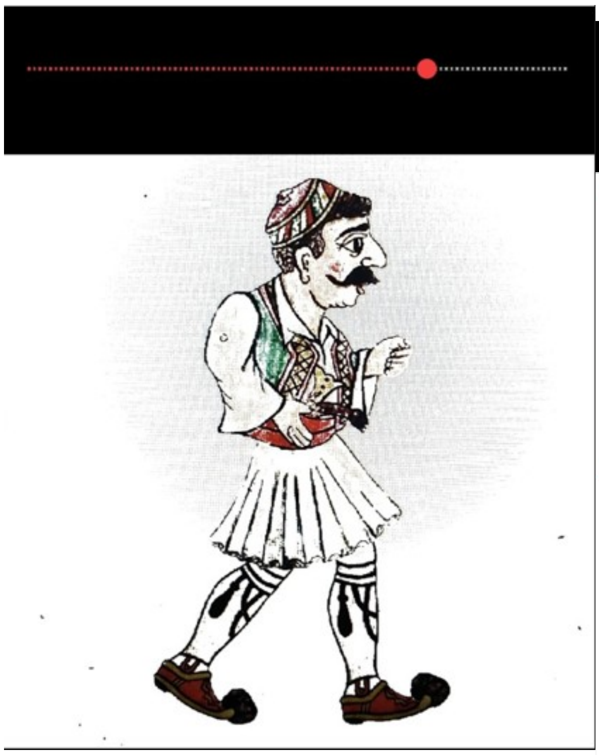


Figure 13: Background extraction as implemented in ePuppet.

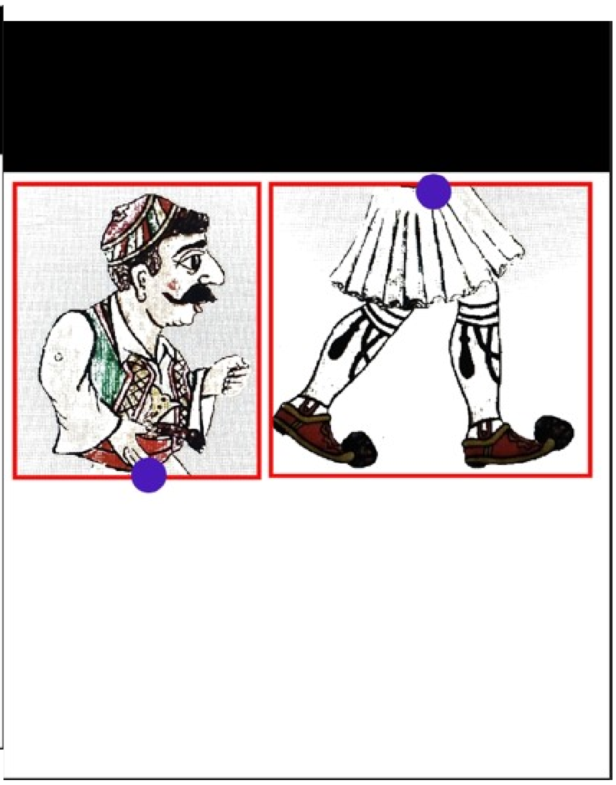


Figure 12: Rectangle packing as implemented in ePuppet.

2.3.1 Image as a 2D Array of Pixels

Working with pixel data is common in image processing applications and as a result very well documented. However, when adding the constraint of TypeScript as our primary language, many options tend to not be supported. On the bright side, the HTML5 canvas supports WebGl and as a result, is usable by any and all web applications.

- HTML5 canvas

  The Canvas API provides a means for drawing graphics via JavaScript and the HTML <canvas> element. Among other things, it can be used for animation, game graphics, data visualization, photo manipulation, and real-time video processing.

- WebGL canvas

  WebGL enables web content to use an API based on OpenGL ES 2.0 to perform 2D and 3D rendering in an HTML canvas in browsers that support it without the use of plug-ins. WebGL programs consist of control code written in JavaScript and shader code (GLSL) that is executed on a computer's Graphics Processing Unit (GPU). WebGL elements can be mixed with other HTML elements and composited with other parts of the page or page background.

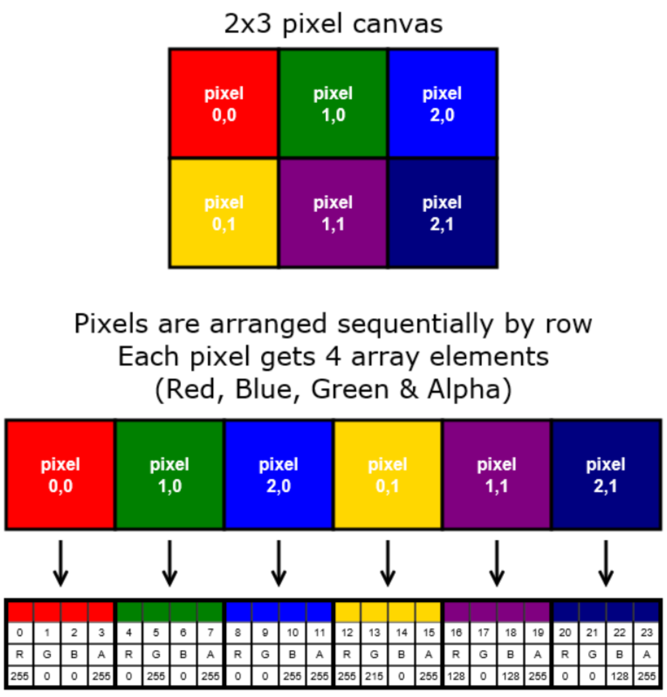


Figure 14: Pixel Data on HTML5 canvas

2.3.2 Figure/ Background Distinction

There is no universally correct answer as to which method is best and it is usually decided on a case by case basis. State of the Art suggests that a machine learning algorithm would yield the best results in the long term, however such a practice would require significant data before it becomes efficient, which is currently not available. Pattern recognition algorithms tend to fail in implementations like this project due to the fact that the patterns are highly irregular.

The inspiration for tackling this problem came from the fact that there are places inside an image that are guaranteed to be background, since they are outside of the part boxes. Using the aforementioned information we can deduce that simply comparing other pixels of interest to those background pixels results in a very good estimate of whether a pixel contains figure data or not. However, this implementation comes at a very steep cost: the backgrounds of the captured images must be purely white or very close.

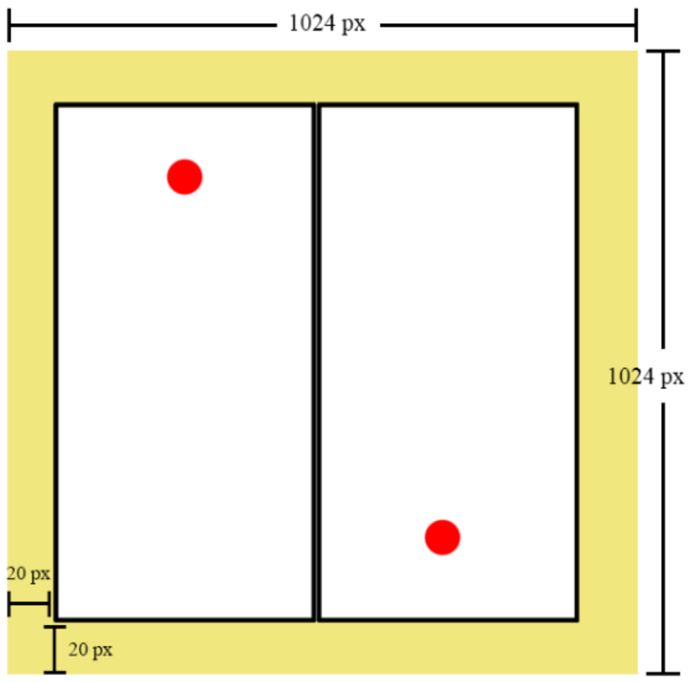


Figure 15: The highlighted area is always out of bounds of a figure, so it is safe to use as an indicator of the background color.

2.3.3 Rectangle Packing

Packing problems are a class of optimization in mathematics that involve attempting to pack objects together into containers. The goal is to either pack a single container as densely as possible or pack all objects using as few containers as possible. Many of these problems can be related to real life packaging, storage and transportation issues. Each packing problem has a dual covering problem, which asks how many of the same objects are required to completely cover every region of the container, where objects are allowed to overlap.

Rectangle Packing is a specific packing problem that we had to solve, where the objective is to determine whether a given set of small rectangles can be placed inside a given large polygon, such that no two small rectangles overlap. This algorithm was used in order to create the final Sprite Sheet, as all parts should fit on a new 1024x1024 image, or throw an error in case the small rectangles do not fit.

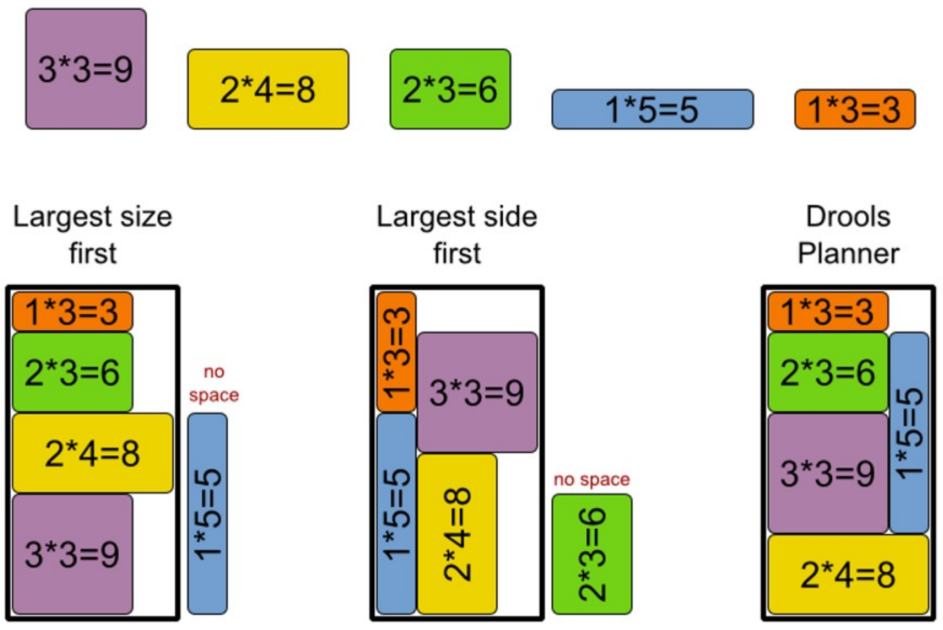


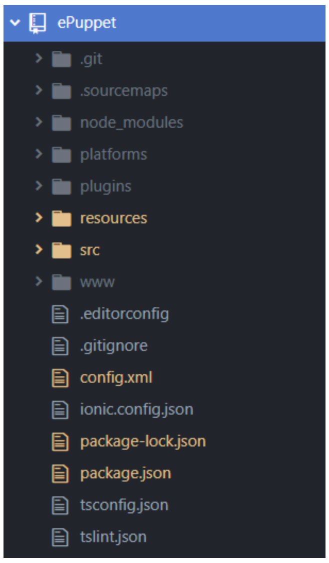
Figure 16: Example solutions for rectangle packing problems.

2.4 Project Structure, Typescript and Plugins

This section provides a concise summary of the plugins and libraries we used in the implementation. In section 2.4.1, the base project structure is shown, along with a brief explanation of each important sub-folder and file. It is important to note that ionic apps come with essential files, also called overhead. Section 2.4.2 provides insight on some necessary typescript only functions that are widely used in the project and, finally, Sections 2.4.3 and 2.4.4 explain the two most important plugins we used in the implementation, the Camera and the Device Storage. These plugins were essential in the implementation of ePuppet as they provided an out of the box solution for communicating between the device hardware and ePuppet.

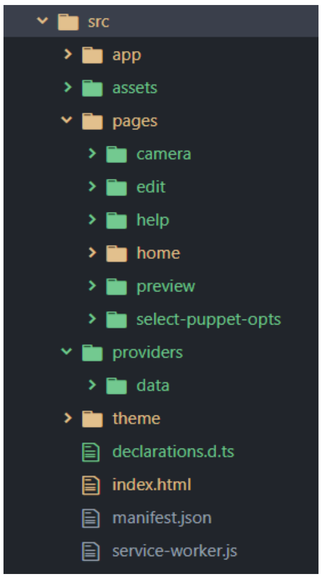
2.4.1 Project Structure

The structure shown below can be found in every Ionic Angular project. By generating a blank application with the *ionic start* command on the CLI we ensure that all the necessary dependencies are included and any initial setup is performed correctly in order to have a working Ionic app, albeit a blank one. We then, further populate the Src folder with our custom code.



```
ePuppet
  .git
  .sourcemaps
  node_modules
  platforms
  plugins
  resources
  src
  www
  .editorconfig
  .gitignore
  config.xml
  ionic.config.json
  package-lock.json
  package.json
  tsconfig.json
  tslint.json
```

Figure 18: Ionic Angular project structure



```
src
  app
  assets
  pages
    camera
    edit
    help
    home
    preview
    select-puppet-opts
  providers
    data
  theme
  declarations.d.ts
  index.html
  manifest.json
  service-worker.js
```

Figure 17: Src Folder contents

- **node_modules**

  This folder contains the bulk of API data used in the project as described further in package.json below. It is generated each time we rebuild our project based on the dependencies and specifications of package.json.

- **Platforms**

  This folder contains all the necessary platform specific data. In ePuppet's case, the only currently supported platform is Android.

- **Plugins**

  Any cordova plugins used throughout the project are installed here, such as the camera which is a focal point of this application.

- **resources**

  Miscellaneous media files that are essential for any mobile application, such as the splash-screen and the app icon.

- **Src**

  The main folder that hosts most of our hand written code. The two main subfolders are the Pages and Providers. Pages are designed to be rendered to the DOM, including navigation properties and HTML components. Providers don't have any visible components but communicate with every page and assist as a global function/ variable carrier.

- **www**

  Building a hybrid application requires a www folder where the output of the build is generated and is the starting point of the application launch.

2.4.2 Typescript Core Concepts

TypeScript is a language that aims at easing development of large scale applications written in JavaScript. TypeScript adds common concepts such as classes, modules, interfaces, generics and (optional) static typing to JavaScript. It is a superset of JavaScript: all JavaScript code is valid TypeScript code so it can be added seamlessly to any project. The TypeScript compiler emits JavaScript. Below, we present some of the core concepts that are used in Typescript:

- **Type Checking**

  TypeScript always points out the compilation errors at the time of development only. Because of this in the run-time, the chance of getting errors is very less whereas JavaScript is an interpreted language. It also includes a feature that is strongly typed or supports static typing. That means Static typing allows for checking type correctness at compile time. This is not available in JavaScript.

- **Promises**

  Asynchronous programming allows a user to go about his business in an application, while processes run in the background, thus enhancing the user experience. With asynchronous programming, the user can move to another screen while the function continues to execute. This is especially important in mobile applications since they often run on older devices. In addition, promise-based functions guarantee that functions are executed in the correct order. A promise is a TypeScript object which is used to write asynchronous programs. A promise is always a better choice when it comes to managing multiple asynchronous operations, error handling and better code readability.

2.4.3 Native Storage

Installing any Ionic Cordova plugin requires the following commands on the Ionic CLI. Afterwards, any time we want to make use of the plugin, we just have to import it in the specific page and declare an object of the appropriate type in the constructor.

```
ionic cordova plugin add cordova-plugin-nativestorage
npm install @ionic-native/native-storage
```

Native Storage is a plugin that allows us to tap into the device storage, either internal or external in the form of SD cards. This is necessary for two prime reasons: First of all, this app is designed to work offline most of the time so it cannot auto-upload or backup and this is exemplified by the fact that our primary user base are children, thus the need for security. We had to implement both write and delete functions for our files so as not to clog the storage. Each application installation automatically creates a data folder under the reverse domain name of the app. In our case it is com.tuc.epuppet. When first launching the app, it checks for a Figures and Sceneries Subfolder under the main folder and if they do not exist, it creates them. After naming a figure file, another subfolder is created under Figures named after it. This folder includes the three required files for eShadow.

2.4.4 Device Camera

As seen above in the Storage Section, using the device camera requires a plugin that is installed through the following commands in the CLI:

```
ionic cordova plugin add cordova-plugin-camera-preview
npm install --save @ionic-native/camera-preview@4
```

This plugin comes with decent customization regarding the quality and size of the image. The following code snippet shows the options we used for the figure capturing process:

```
pictureOpts: CameraPreviewPictureOptions = {
width: 1024,
height: 1024,
quality: 100
};
```

```
cameraPreviewOpts: CameraPreviewOptions = {
x: 0,
y: (window.innerHeight - window.screen.width)/2,
width: window.screen.width,
height: window.screen.width,
camera: 'rear',
tapPhoto: false,
previewDrag: false,
toBack: true,
alpha: 1
};
```

The width and height attributes set the size of the output PNG image in pixels and the quality attribute its definition on a scale of 0 to 100, which affects the size of the output and its compression relative to the original snapshot.

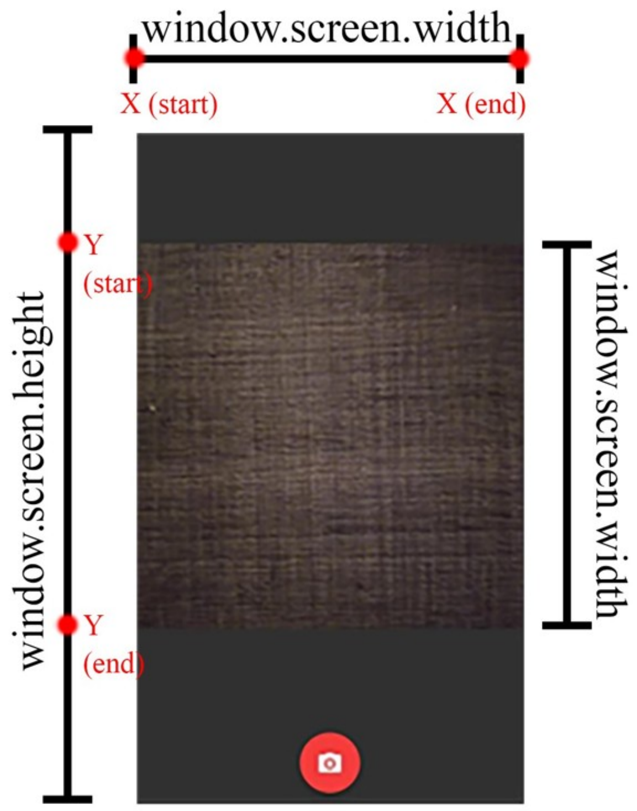


Figure 19: Camera Plugin sizes

The `CameraPreviewPictureOptions` structure contains the necessary information regarding the photograph capturing area, along with important parameters for the capture. Attributes x and y are the starting points of the capture area on our screen and their ending points are determined by the width and height attributes. In our case, we need the capture area to be a square and aligned to the center of the view. It is important to note that, since mobile devices have variable screen sizes, the capturing area must be relevant to the device's screen size. Using relative sizes in order to adjust ePuppet to any mobile device was a core challenge throughout the implementation so as to include as many devices and potential users.

Chapter 3: Functional Requirements and Use Cases

The first step of a mobile application design process is to determine the data that the mobile app will display to the users, the data it will collect, user interactions with the finished product, and the user journeys within the app. There is a plethora of tools available that assist in the aforementioned process but the most common way to extract this information is through the use of use cases and functional requirements.

A **Functional Requirement** (FR) is a description of the service that the software must offer. It describes a software system or its component. A function is nothing but inputs to the software system, its behavior, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform. Functional Requirements in Software Engineering are also called **Functional Specification**. In software engineering and systems engineering, a Functional Requirement can range from the high-level abstract statement of the sender's necessity to detailed mathematical functional requirement specifications. Functional software requirements help us to capture the intended behavior of the system.

A **Use Case** describes how a user uses a system to accomplish a particular goal. A **Use Case Diagram** consists of the system, the related use cases and actors and relates these to each other to visualize:

- what is being described? (system),
- who is using the system? (actors),
- and what do the actors want to achieve? (use cases),

thus, use cases help ensure that the correct system is developed by capturing the requirements from the user's point of view. While a use case itself might drill into a lot of detail (such as, flow of events and scenarios) about every possibility, a use-case diagram can help provide a higher-level view of the system, providing the simplified and graphical representation of what the system must actually do.

3.1 Functional Requirements

ePuppet intends to provide an all-in-one solution for end users to digitize, customize and import custom figures and scenery in the eShadow platform. The most important qualities of ePuppet are ease of use, minimal design and as little user input as possible in order to streamline the process as discussed in Chapter 1. Since we made no assumptions on the level of technical expertise that the end user should have before using the application, learnability is a core requirement for ePuppet. We should also provide as wide a range as possible for figures that can be processed through the application as we aim to embolden creative expression of users.

The functional specification for this project shaped up as follows:

- The UI should be intuitive and provide guidance for each step
- The application should automate as many steps of the figure digitization as possible in order to achieve minimum complexity
- The steps mentioned above should be easy to understand and explain
- Users should be able to use templates to digitize figures in one click or use an advanced process to dissect any figure they want
- The background used during the image capturing should be as devoid of multiple colors and different light sources as possible to achieve maximum efficiency
- Deleting and reviewing existing figures should be possible
- A tutorial should be available
- An undo function should exist during the part and joint selection processes
- Figures should be saved on the device's local storage for improved security
- Upload functionality should be implemented to circumvent the need for USB cable transfers from the mobile device where ePuppet runs, to the desktop computer where eShadow is installed.

3.2 Use Cases

In this figure we present the Use Cases of an ePuppet user that wants to digitize a figure:

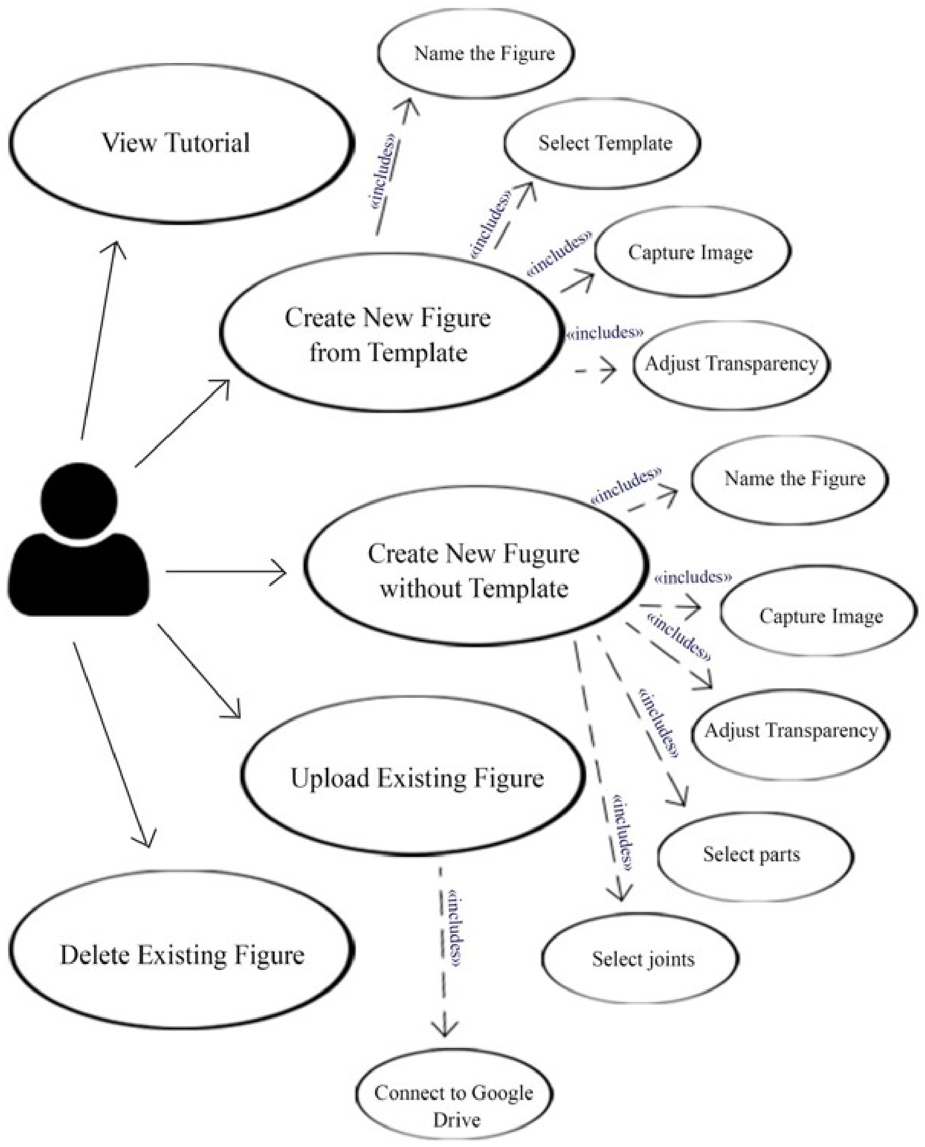


Figure 20: ePuppet common Use Cases.

Here we will detail the aforementioned Use Cases and their challenges:

- Use Case: **View Tutorial**

  The user downloads ePuppet for the first time. He wants to use the application to digitize paintings but has no prior knowledge of the process. He opens the menu and selects one of the available tutorials, depending on what kind of figure he has available.

- Use Case: **Create New Figure from Template**

  The user wants to create a Figure that is already available in four parts on paper. He selects “Start new Project” and navigates to the Puppet options Page. He provides a valid name for his figure and selects the “4-part template” option from the list of available templates. On the next screen, he aligns the physical parts with the template on the camera and captures the image. He then adjusts the transparency slider to achieve the optimal result and finalizes the process.

- Use Case: **Create New Figure without Template**

  The user wants to create a Figure that is still in one piece, so he is unable to use templates. He selects “Start new Project” and navigates to the Puppet options Page. He provides a valid name for his figure and selects the “free figure creation” option from the list of available templates. On the next screen, he aligns the physical parts with the template on the camera and captures the image. He then adjusts the transparency slider to achieve the optimal result. He, then, is asked to select which are the individual parts of the figure. After determining the parts, he selects the joint points between them and then finalizes the puppet.

- Use Case: **Upload Existing Figure**

  The user has created a digital puppet using one of the methods mentioned above and now wants to import it in eShadow. The upload button on the figure is currently disabled as he first has to open the menu and connect to his Google Drive. The upload button now becomes enabled.

- Use Case: **Delete Existing Figure**

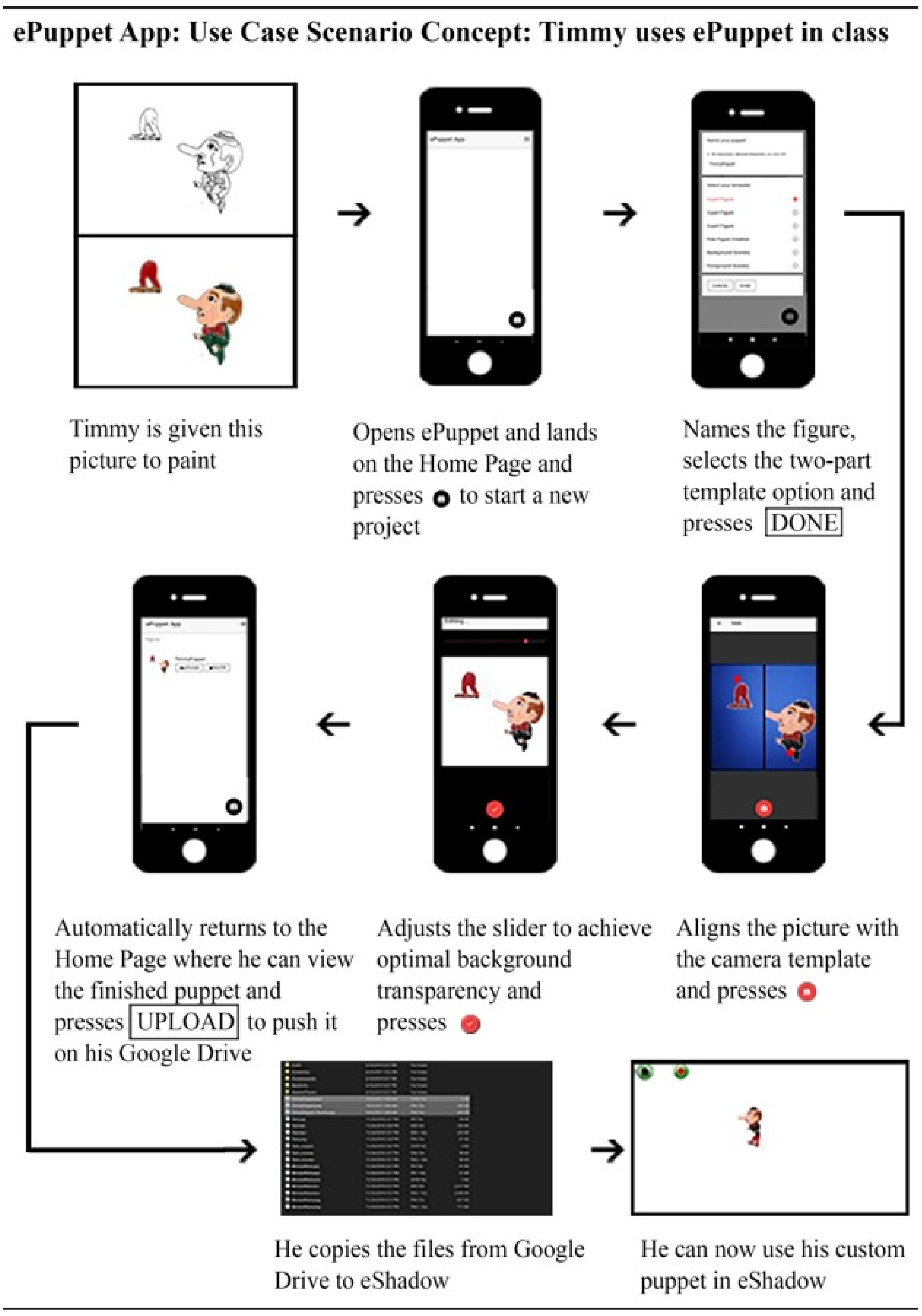
  The user wants to delete an existing figure to remake it or because he has already imported it in eShadow and is no longer needed in ePuppet. He selects delete and is prompted to confirm the deletion. By clicking “yes” he removes all data from the device.

3.3 Personas

Here we present specific use case scenarios for two imaginary user that capture how two target groups, teachers and students, can use ePuppet in their creative work with eShadow:

- Timmy, who is an elementary school student using ePuppet to create a simple digital puppet from a template.
- Mr Garison, who is a teachers using ePuppet to create a digital puppet from a picture.

This approach to describe how an application is to be used using fictional personas is widely used to simulate real scenarios with potential users and their interaction with the application. We often created personas such as the one displayed here in order to grasp the viewpoint of as many potential users as possible over a specific scenario inside ePuppet, which was invaluable, especially for the UI elements and the user experience in general.



ePuppet App: Use Case Scenario Concept: Timmy uses ePuppet in class

Timmy is given this picture to paint

Opens ePuppet and lands on the Home Page and presses ● to start a new project

Names the figure, selects the two-part template option and presses DONE

Aligns the picture with the camera template and presses ●

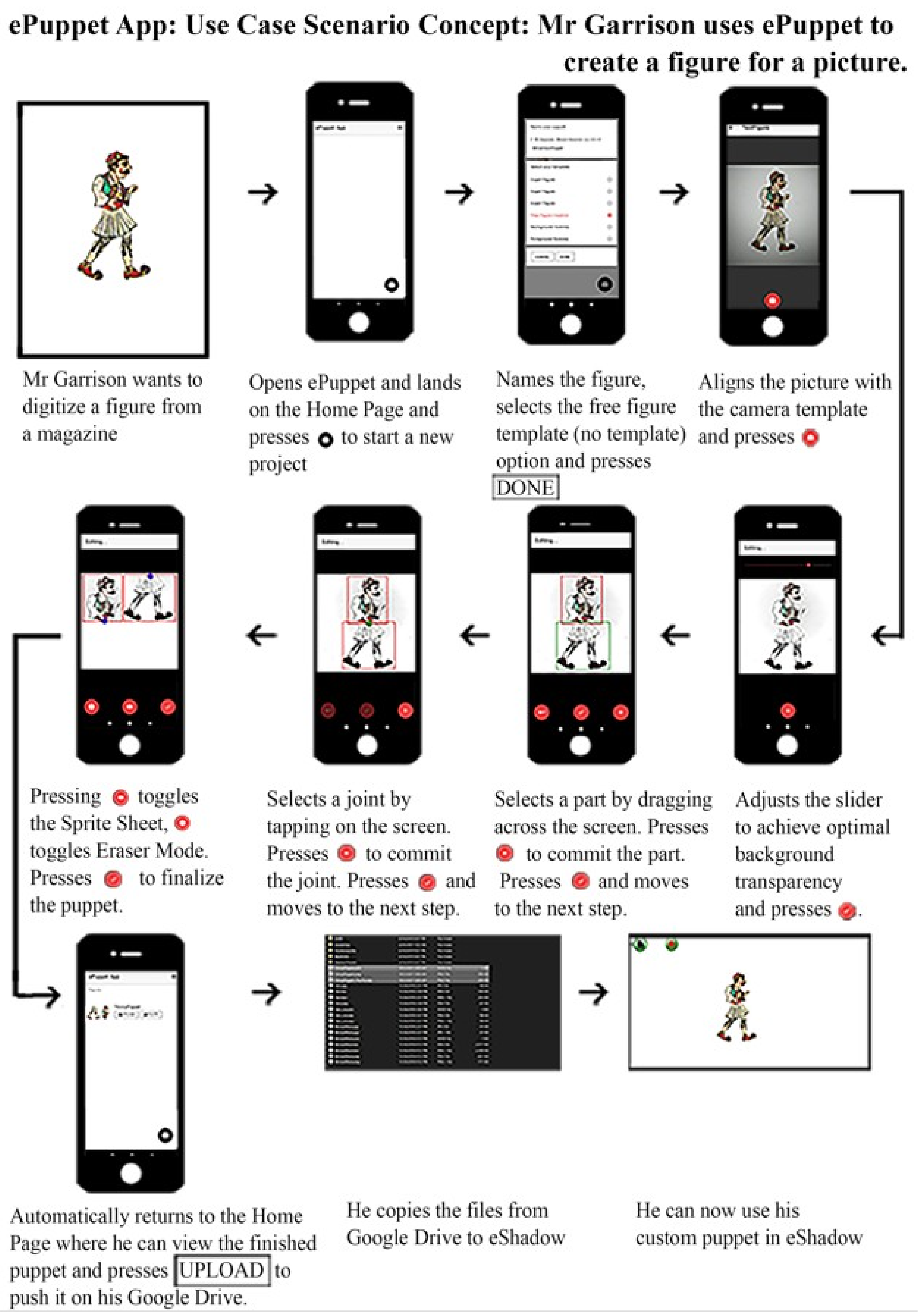
Adjusts the slider to achieve optimal background transparency and presses ●

Automatically returns to the Home Page where he can view the finished puppet and presses UPLOAD to push it on his Google Drive

He copies the files from Google Drive to eShadow

He can now use his custom puppet in eShadow

Figure 21: Use Case Scenario of Timmy, a user of ePuppet wants to create a two part figure.



ePuppet App: Use Case Scenario Concept: Mr Garrison uses ePuppet to create a figure for a picture.

Mr Garrison wants to digitize a figure from a magazine

Opens ePuppet and lands on the Home Page and presses ● to start a new project

Names the figure, selects the free figure template (no template) option and presses DONE

Aligns the picture with the camera template and presses ●

Adjusts the slider to achieve optimal background transparency and presses ●.

Selects a part by dragging across the screen. Presses ● to commit the part. Presses ● and moves to the next step.

Selects a joint by tapping on the screen. Presses ● to commit the joint. Presses ● and moves to the next step.

Pressing ● toggles the Sprite Sheet, ● toggles Eraser Mode. Presses ● to finalize the puppet.

Automatically returns to the Home Page where he can view the finished puppet and presses UPLOAD to push it on his Google Drive.

He copies the files from Google Drive to eShadow

He can now use his custom puppet in eShadow

Figure 22: Persona 2: Mr Garrison, an elementary school teacher wants to digitize a figure from a picture.

Chapter 4: Implementation

This chapter discusses the implementation of ePuppet. Section 4.1 provides insight on the top level architecture of ePuppet as an Ionic Angular application. Section 4.2 delves into the application flow with the assistance of flowcharts. A brief display of all possible screens is shown, along with the navigation between them. The distinction between the two editing modes is also highlighted as separate paths in the flow.

An activity diagram is also included, which highlights distinct actions that can be undertaken in ePuppet. Section 4.3 focuses on the image data manipulation specifically, since this is the most challenging point of the application. In section 4.3.1 we explain the algorithm that we implemented to define which areas in the picture are background and how we replace these areas with transparency. The next sections focus on the process of dissecting the captured image into non overlapping parts and the definition of the joints between these parts. The most important code snippets are included and explained in detail. Finally, section 4.4 explains various other implementation points of interest and challenges such as the interface that connects and realizes the Google Drive upload sequence and the transformation of PNG images into BLOBs, which is a mandatory step before being able to use the device storage.

4.1 Architecture

Mobile application architecture is a set of techniques and patterns required to design and build a mobile application based on industry and vendor-specific standards. Application architecture provides a road map as well as best practices to follow when building an application so that the final application is well structured. As it is the skeleton of a mobile application, if the architecture missing any vital parts, it could endanger the success of the entire project as a whole. The complexity of crafting high-quality architecture depends on the size of the application being built and applying the proper architecture will allow for time and cost savings because different models work best in different scenarios. A common practice for top level architecture of mobile applications is to use the MVC (Model – View – Controller) software design pattern. A brief description of this pattern is displayed bellow.

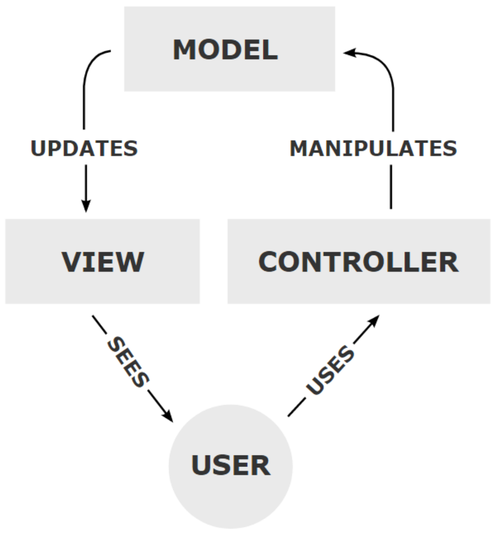


Figure 23: MVC design pattern.

Below we detail the three main components in this pattern:

- **Model**

  The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.

- **View**

  Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

- **Controller**

  Accepts input and converts it to commands for the model or view. In addition to dividing the application into these components, the model–view–controller design defines the interactions between them:

  - The model is responsible for managing the data of the application. It receives user input from the controller.
  - The view renders presentation of the model in a particular format.
  - The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

As with other software patterns, MVC expresses the "core of the solution" to a problem while allowing it to be adapted for each system. Particular MVC designs can vary significantly from the traditional description. In the case of ePuppet, we used Ionic Angular as the framework. In MVC if we make any change in the view it does not get updated in model. But Angular uses 2-way binding to enable the MVVM design pattern. MVVM basically includes 3 things:

1.Model

2.View

3.View Model

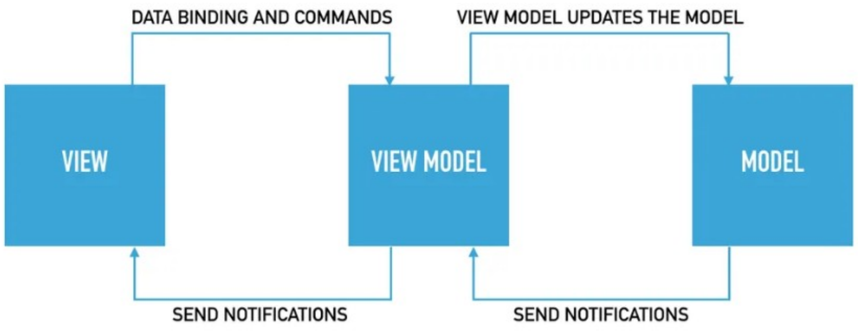


Figure 24: MVVM design pattern.

Controller is actually replaced by View Model in MVVM design pattern. View Model is nothing but a JavaScript function which is again like a controller and is responsible for maintaining relationship between the view and the model, but the difference here is that if we update anything in the view, it gets updated in the model. Inversely changing anything in the model, shows up in the view, which is what we call 2-way binding.

4.2 Application Flow and Activities

In this section, we present the main pages of the application, along with any possible navigation among them. The functionality of every button will be explained in the following sections. In the first figure, we present an activity diagram that highlights the inputs, processes and outputs of the standard workflow of ePuppet for any given activity initiated by the user.

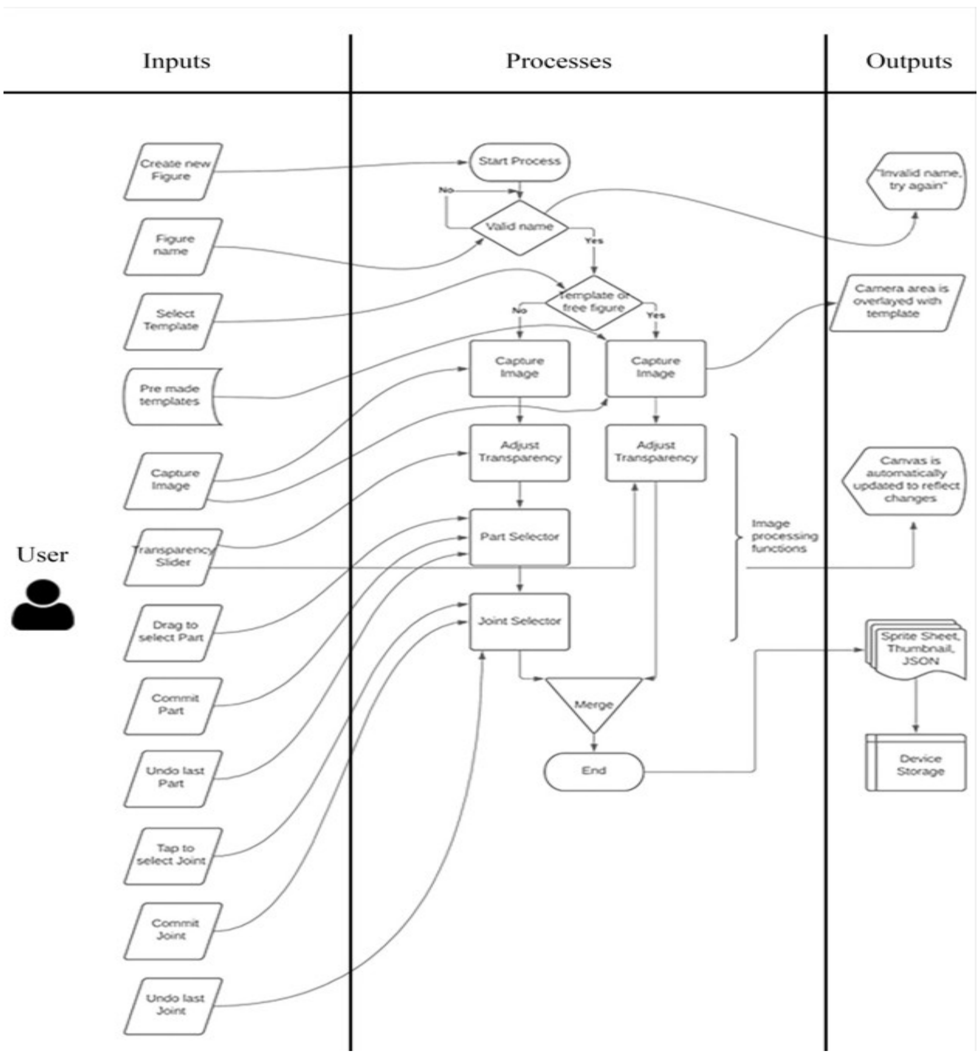


Figure 25: ePuppet main workflow highlighting the Inputs, Processes and Outputs.

In this figure, we present a complete flowchart of the pages and the transitions among them. This flowchart is the navigation map of ePuppet.

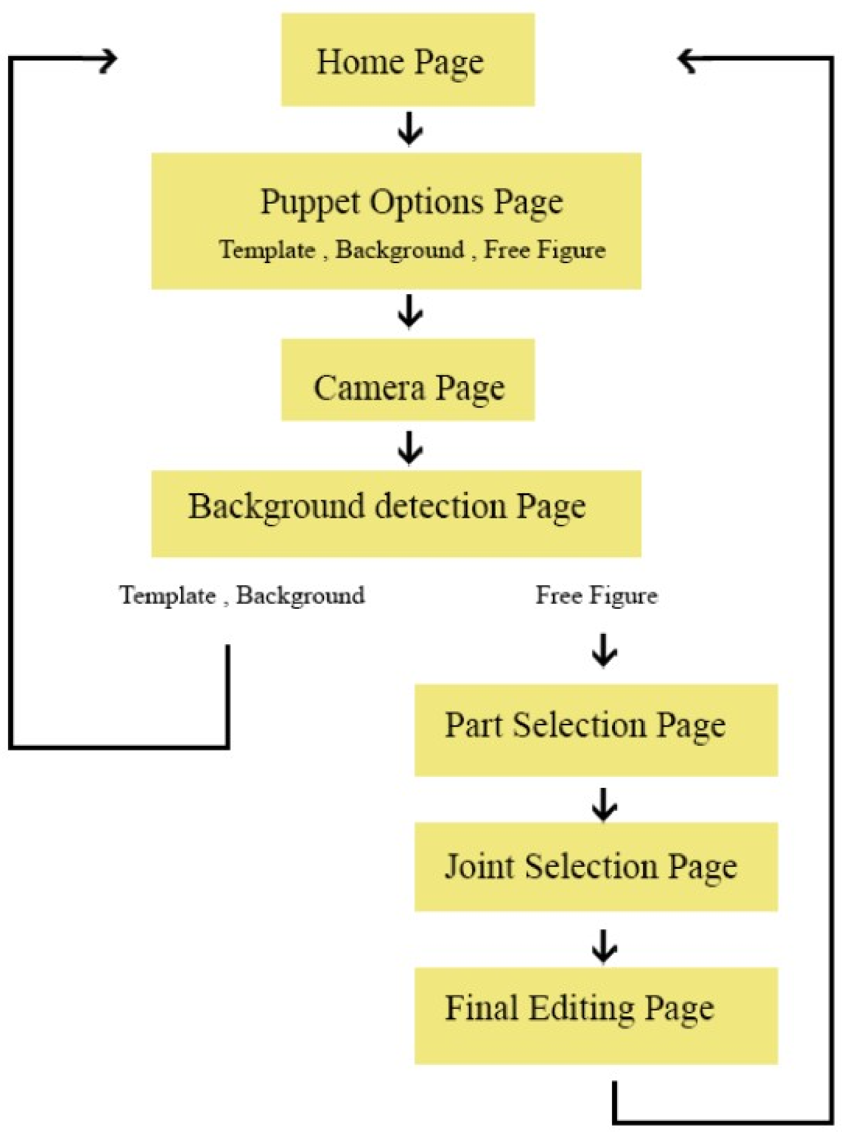


Figure 26: ePuppet Flow Chart highlighting the transition between pages. Each of these pages will be detailed in the following Sections.

4.2.1 The Home Page

When loading the application, the user always lands on the Home Page. This view contains three important points of interaction that will be further explained in this section. Below, a screenshot of the page is included, for easier reference. However, it is worth noting that the application must check each time whether this is the first time it has been launched after the installation or not, as the startup behavior is different in each case.

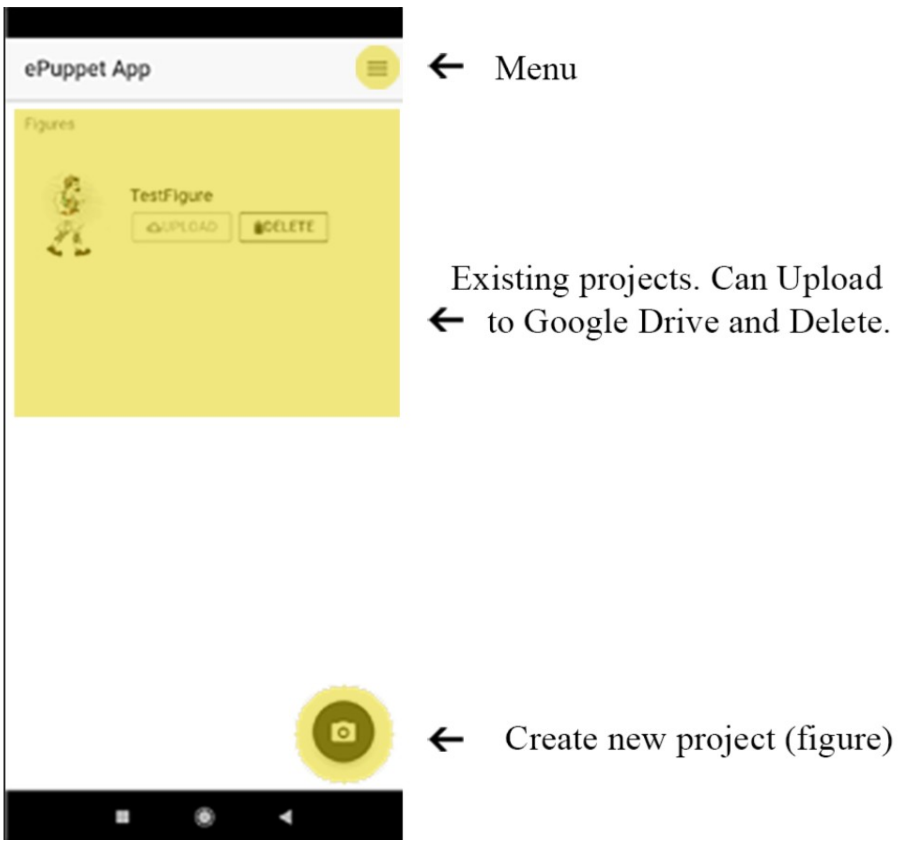

ePuppet App
Figures
TestFigure
UPLOAD
DELETE
Menu
Existing projects. Can Upload to Google Drive and Delete.
Create new project (figure)

Figure 27: ePuppet Home Page

The very first time that the application is launched after the installation, the base folder structure must be initialized, essentially creating the Figures and Sceneries Folders under com.tuc.epuppet as mentioned in section 2.4.3. Additionally, the user must be prompted to accept the usage of any device features the app requires, as dictated by Google for Android and Apple for iOS. In ePuppet’s case, permissions are required for using the Device Camera and Storage. Finally, this first time usage check is valuable in order to prompt the user to go through a tutorial, which is currently not implemented as it adds unnecessary steps during development.

As seen in the figure above, the user can interact with a number of elements on the Home Page to achieve a goal from those presented in Chapter 3 as Use Cases.

- **Menu**

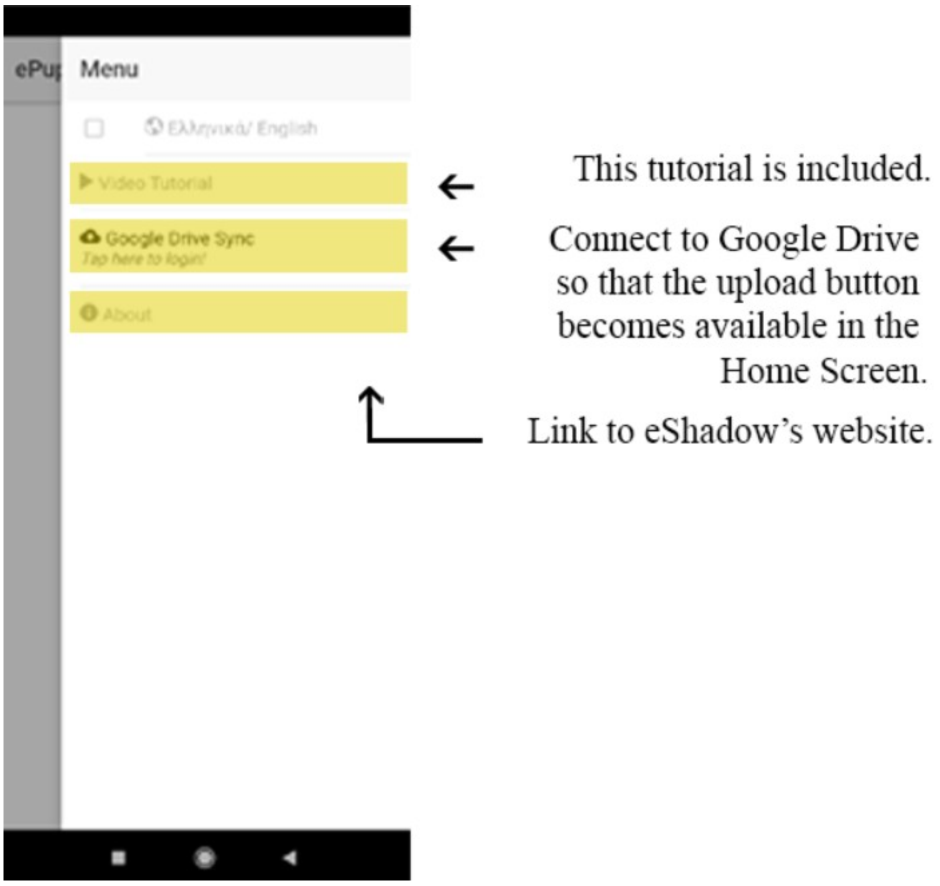

ePup
Menu
Ελληνικά/ English
Video Tutorial
Google Drive Sync
Tap here to login!
About
This tutorial is included.
Connect to Google Drive so that the upload button becomes available in the Home Screen.
Link to eShadow's website.

Figure 28: ePuppet Menu. Note that Greek are not currently supported.

- The menu offers basic functionality that cannot be placed directly in the view, in order not to clog the UI. The **tutorial** is a slideshow of explanatory images that go through the whole application and creation sequence. Since most buttons contain simple icons, it is often difficult for a new user to distinguish the functionality of each button, so a tutorial is important in that aspect. The tutorial was added later during development after user feedback made it clear that it was necessary.
- The **Google Drive Login** button prompts the user to login through Google using oAuth2.0, in order to be able to upload finished projects on the cloud for easier importing in eShadow. This process will be further explained in section 4.4.2.
- The **About** button links to the eShadow website. This is intended to be used along with the new eShadow version that targets mobile and tablet devices.

- **Start New Project**

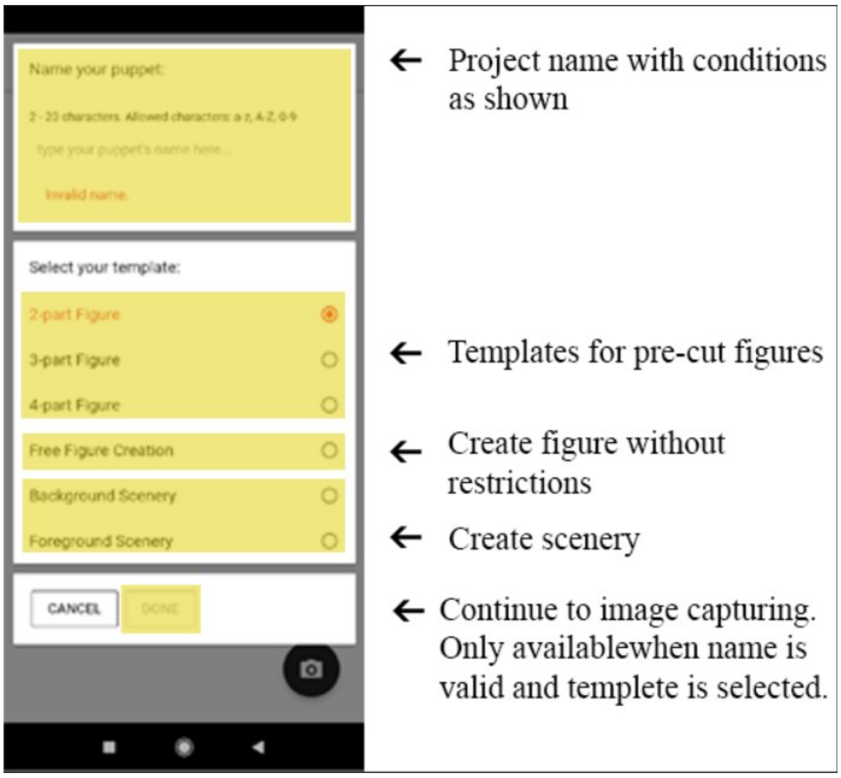
  This button initiates the figure/ scenery creation process by pushing the next view, where the user is prompted to provide necessary information regarding his/ her new project before proceeding further.

- **List of Projects**

  This area of the view is occupied by a list of finished projects. A discreet separator is implemented to distinguish figures from scenery. The list is dynamically rendered in the view, so that it can be updated in real time. If no projects exist, this area is empty. Furthermore, tapping on an item of the list offers a preview of the project. Removing any unwanted project is implemented through the Delete button. Finally, the Upload button is by default disabled, as it requires that the user logs in his Google account through the menu before becoming available.

4.2.2 Puppet Options Page

This page provides all the necessary parameters that must be taken into consideration before digitizing a figure. The name that will be used in all three output files is required here, along with a selection of all available editing templates for figures and sceneries. Both a valid name and a template are required to proceed further. The UI for the Puppet Options Page is presented here:



Name your puppet:

2 - 20 characters. Allowed characters: a-z, A-Z, 0-9

type your puppet's name here...

Invalid name.

Select your template:

[✓ 2-part Figure]
3-part Figure
4-part Figure
Free Figure Creation
Background Scenery
Foreground Scenery

CANCEL
DONE
← Project name with conditions as shown
← Templates for pre-cut figures
← Create figure without restrictions
← Create scenery
← Continue to image capturing. Only availablewhen name is valid and templete is selected.

Figure 29: New Project Creation Options

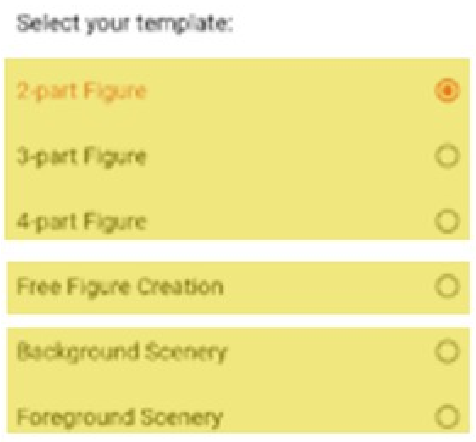
After starting a new project, the first step is to name the project (either a figure or some scenery). This is the name that will be used for all three necessary files and will be displayed in eShadow, For example, given the name *test*, after the figure creation process, three files will be created named *test.png*, *test_Thumb.png* and *test.json*. Only Latin characters and numbers are permitted in order to avoid problems across devices, and the DONE button is disabled until a valid name is given. The following code snippet indicates the naming policy we enforced. Most disallowed characters, such as special characters, were avoided due to the fact that these names are used as file paths.

```
name: ['', Validators.compose([Validators.maxLength(20),
Validators.minLength(2), Validators.pattern('[a-zA-Z0-9]*'),
Validators.required])]
```

The naming restrictions imposed are the following, as seen above:

- 2-20 characters
- allowed characters are all uppercase and lowercase English letters
- numbers
- no special characters are allowed

Additionally, this is the point where the user opts to either use a premade template to further simplify the process, or create a custom figure with additional steps. Only one option can be active at a time, and this selection impacts every following step. This selection also loads the correct template on the camera if necessary (2, 3, 4-part template or no template for Free Figure Creation) and prepares a JSON template for further use in the next steps.



Select your template:

- 2-part Figure [✓]
- 3-part Figure
- 4-part Figure
- Free Figure Creation
- Background Scenery
- Foreground Scenery

Figure 30: Only one of these options can be selected at a time, and determines the figure creation process that will be realized in the next steps.

4.3 Image Data Manipulation

The central part of the Edit Page is occupied by the HTML5 canvas. WebGL enables web content to use a JavaScript API based on OpenGL ES 2.0 to perform 2D and 3D rendering in an HTML canvas in browsers that support it without the use of plug-ins. WebGL programs consist of control code written in JavaScript and shader code (GLSL) that is executed on a computer's Graphics Processing Unit (GPU). WebGL elements can be mixed with other HTML elements and composited with other parts of the page or page background. However, it is impossible to achieve the required functionality by using a single canvas. This can be deduced by going through the edit process step by step. The aforementioned canvas will, from now on, be called canvas #1.

Assuming we use a single canvas, the first step is to place the image data on the canvas with the proper scaling, since captured images are 1024 x 1024 pixels but the canvas is a square of screenWidth x screenWidth which is variable based on the size of the device and its resolution. The HTML that holds the canvas stack is presented here and explained further below:

```
  <div class="ion-canvas">
      <canvas #canvas1 style=”z-index:0;”>

      </canvas>
      <canvas #canvas2 style=”z-index:1;”>

      </canvas>
      <canvas #canvas3 style=”z-index:2;”>

      </canvas>
      <canvas #canvas4 style=”z-index:3;”
      (touchstart)="handleStart($event)" (touchend)="handleEnd($event)"
      (touchmove)="handleMove($event)">

      </canvas>
 </div>
```

The next step involves sampling the image edges for the background color and then converting any pixels of similar color to transparent pixels. This step can be done in canvas #1 without compromising the next steps. If we are using a template, no additional steps are required after this one, and the figure is created successfully.

However, if we are creating a figure without using templates, the next step in the creation process is the part selection. This step requires additional layers of canvases. Canvas #1 maintains the original image, which is needed in this step as well as the joint selection afterwards. It is also used to create the thumbnail. Another three canvases are then stacked on top of the initial, with an incremental z-index so as to create a stack. Canvas #4 is the top canvas, meaning that if this canvas is fully occupied with non transparent data, no information from the canvases with a lower z-index are visible. During the part selection, canvas #2 is a copy of canvas #1. Canvas #3 is where the selected parts are visualized with the help of red rectangles. This canvas is a transparent layer that, when a part is committed, is redrawn to include the rectangle. Since this canvas is transparent, canvas #2 is still visible, albeit overlay-ed with the selected parts. Canvas #4 is different from the rest, in that it recognizes touch gestures such as single Taps or Drag events. Dragging on the area creates a temporary green rectangle that can either be committed (drawn on canvas #3) or removed by starting another touch gesture.

The next step of the process is the joint selection. Canvas #2, which was unused in the previous step, now becomes a copy of canvas #3, which contains the red rectangles that indicate the selected parts and is otherwise transparent. Canvas #1 still holds the original image. Canvas #3 is where the selected joints are visualized as pairs with the help of same colored dots. This canvas is a transparent layer that, when a joint is committed, is redrawn to include the dot. Since this canvas is transparent, canvas #1 and #2 are still visible, albeit overlay-ed with the selected joints. Canvas #4 functions similarly to the part selection step, recognizing Taps, and either drawing on canvas #3 on joint commit, or remove the dot by tapping on another part of the screen.

The final step presents the Sprite Sheet we generated in the previous steps on canvas #2 and uses canvas #4 to detect touch gestures in eraser mode (touching on the canvas while eraser mode is ON converts those pixels in transparency). Canvas #3 is used as an overlay for the Sprite Sheet which can be toggled to show dots and rectangles. The thumbnail is extracted from canvas #1 and the Sprite Sheet from canvas #2.

4.3.1 Background Transparency

In Section 2.3.2 we discussed the problem of detecting the background and replacing it with transparent pixels. Given that the dominant use case is figure creation from a painting, we assume that the background of the figure at the time of capturing the picture will be a single color area (paper, desk, whiteboard, etc.). WebGL uses the RGBA color scheme, where each parameter (red, green, and blue) defines the intensity of the color between 0 and 255 and Alpha represents the opacity with values ranging from 0 to 255, where 0 is full transparency and 255 is full opacity. For example, RGBA(255, 0, 0, 255) is displayed as red, because red is set to its highest value (255) and the others are set to 0.

White in RGBA color representation scheme is represented by these values (255, 255, 255, 255). However, even given perfect conditions, it is unrealistic to expect that pixels from a captured image will be pure white. After testing relevant values, we came to the conclusion that a range of +/-50 for each Red, Green and Blue value for each pixel can be considered background, without compromising figure information.

Below, the function that reads the image and breaks it down to pixel data is presented:

```
this.ctx.readPixels(0, 0, 1024, 1024, this.ctx.RGBA,
this.ctx.UNSIGNED_BYTE, this.og_pixels);
```

The first four inputs indicate which area of the image will be crawled through, with the intent to acquire pixel data. They are, in order, x-index start, y-index start, x-index end, y-index end. Since the image is 1024 x 1024 pixels, this example includes the whole image. The next input determines that the color scheme of the output will be RGBA. The sixth input declares the data type of the output array. The last input is the array where the pixel data will be stored for further processing.

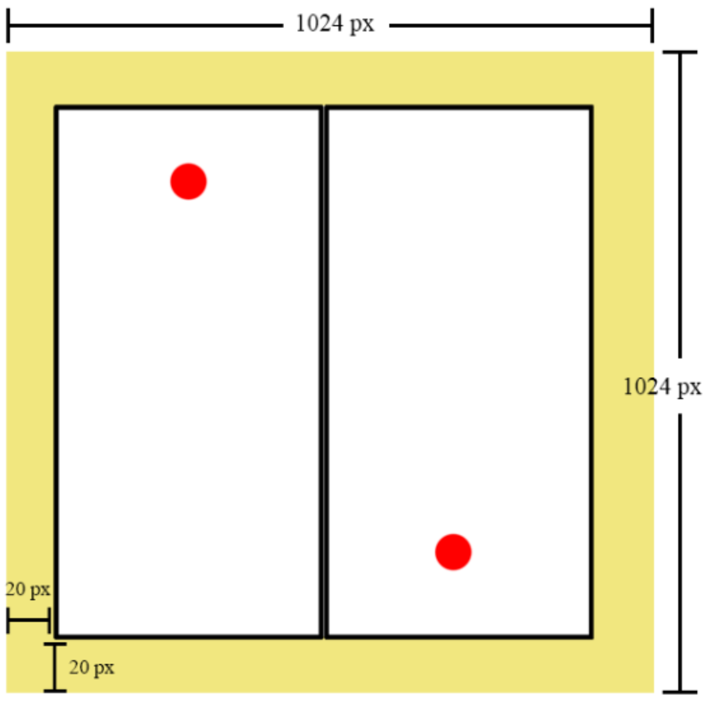


Figure 31: The highlighted area is presumed to be explicitly background, therefore can be safely sampled in order to get the median color values.

Using this function with specific bounds that only include the edges of the image, and then aggregating the values, we can determine median values for Red, Green and Blue that, when combined, are considered background as presented above. We then use the following function to compare each pixel of the original image to these values, and if they match, we overwrite the pixel with transparency:

```
replacerFunc(range){
      this.edited_pixels = new Uint8Array(this.pixelDataLength);
      for (let i = 0; i < this.pixelDataLength; i+=4) {
        if(this.og_pixels[i]>this.medianRed-range
          && this.og_pixels[i]<this.medianRed+range
          && this.og_pixels[i+1]>this.medianGreen-range
          && this.og_pixels[i+1]<this.medianGreen+range
          && this.og_pixels[i+2]>this.medianBlue-range
          && this.og_pixels[i+2]<this.medianBlue+range){
            this.edited_pixels[i] = 255;
            this.edited_pixels[i+1] = 255;
            this.edited_pixels[i+2] = 255;
            this.edited_pixels[i+3] = 0;
        }
        else{
          this.edited_pixels[i] = this.og_pixels[i];
          this.edited_pixels[i+1] = this.og_pixels[i+1];
          this.edited_pixels[i+2] = this.og_pixels[i+2];
          this.edited_pixels[i+3] = 255;
        }
      }
  }
```

This code snippet scans the array that holds the pixel data for the processed image in a for loop, and checks if each specific pixel is within the +/- 50 range, keeping or replacing it with a white, transparent pixel if it meets the criteria. The array that holds the pixel data is iterated in intervals of four.

4.3.2 Selecting Parts

After the background detection and replacement step, the user is prompted to divide the figure in parts. Meanwhile, the application initializes the canvases and loads the image in the first canvas. The part array is now initialized and is empty until a part is committed through the keepSelection() function. The relevant UI is indicated below for easier reference:

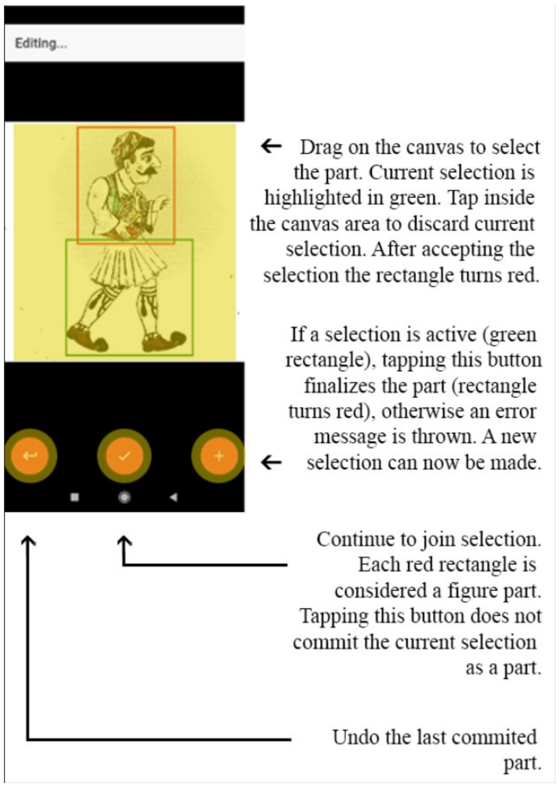

Editing...
← Drag on the canvas to select the part. Current selection is highlighted in green. Tap inside the canvas area to discard current selection. After accepting the selection the rectangle turns red.
If a selection is active (green rectangle), tapping this button finalizes the part (rectangle turns red), otherwise an error message is thrown. A new selection can now be made. ←
Continue to join selection. Each red rectangle is considered a figure part. Tapping this button does not commit the current selection as a part.
Undo the last commited part.

Figure 32: Part Selection Page

In Section 3.3.1 we went into detail regarding the canvases and their functionality. During this step, Touch gestures are recognized on the canvas as indicated on the previous sections. The three functions presented below, implement the functionality that is described in the figure above. Note that only important parts of the functions are included, in order to keep the chapter organized and readable. The function that is displayed below initializes the part selection process. This function is called after capturing an image in the previous step:

```
  partSelector(){
  ...
    this.tempimg.onload = () => {
      this.redrawingFlag = false;
      this.ctx2d.drawImage(this.tempimg,0,0);
    };
    this.tempimg.src = this.tempURL;
   ...
  }
```

After initializing the canvases and loading the captured image in the first canvas, ePuppet is ready to receive user input regarding the part selection. Whenever the user drags his finger across the canvas during this phase, he selects an area, which is highlighted in a green border. The green border area is the user's current selection and is not yet considered a part that has been committed. Thus, any other new gesture on the canvas will remove the current selection in order to begin a new one. The functionality that allows the user to save his current selection as a part is included in the following function. It should be noted that committing a part will push it the part array which holds the saved parts and is used at the end of this step to create the Sprite Sheet and the JSON file accordingly. Whenever the commit button is pushed, this function is executed, and the current selection is redrawn as a red rectangle to signify that this part is now permanent.

```
  keepSelection(){
    this.ctx2d4.clearRect(0,0,this.canvas4.width,this.canvas4.height);
    this.ctx2d3.beginPath();
    this.ctx2d3.lineWidth = 7;
    this.ctx2d3.strokeStyle = "red";
    this.ctx2d3.rect(this.lastX,this.lastY,this.currentX-
this.lastX,this.currentY-this.lastY);
    this.ctx2d3.stroke();
    if(this.lastX>this.currentX){
      let swap = this.lastX;
      this.lastX = this.currentX;
      this.currentX = swap;
    }
    if(this.lastY>this.currentY){
      let swap = this.lastY;
      this.lastY = this.currentY;
      this.currentY = swap;
    }
    this.partListArray.push({
      startX: this.lastX,
      startY: this.lastY,
      endX: this.currentX,
      endY: this.currentY
    });
```

`keepSelection()` is called whenever we commit a part to the part array. The output that the user perceives is the red rectangle that signifies the part on canvas #3, pushes the part coordinates in the `partListArray` which holds the completed parts and, finally, resets canvas #4 and re enables touch events to prepare for the next selection.

```
  discardSelection(){
    this.partListArray.pop();
    this.ctx2d3.clearRect(0,0,this.canvas4.width,this.canvas4.height);
    for(let i = 0; i < this.partListArray.length; i++){
      this.ctx2d3.beginPath();
      this.ctx2d3.lineWidth = 7;
      this.ctx2d3.strokeStyle = "red";
this.ctx2d3.rect(this.partListArray[i].startX,this.partListArray[i].star
tY,this.partListArray[i].endX-
this.partListArray[i].startX,this.partListArray[i].endY-
this.partListArray[i].startY);
      this.ctx2d3.stroke();
    }
  }
```

`discardSelection()` is used if the user wants to undo the last committed part. By calling `partListArray.pop()`, the last element of the array is removed and then we redraw the whole array from scratch in the next render cycle.

Once the part array is complete, we move to the Joint Selection Page. Before doing so, the Sprite Sheet must be created as a background process (the user has no visual cue this is happening). Creating the Sprite Sheet, however, comes with its own challenges. Since the rectangles can and should be overlapping, there is a case where the parts cannot fit in the Sprite Sheet, as it is the same size as the original image. In Section 2.3.3 we briefly explained the problem of Rectangle Packing. Below, we present the algorithm that was used to achieve optimal Rectangle Packing. If the problem cannot be solved, an error is thrown, and the Part Selection process begins anew. This is the rectangle packing algorithm we used:

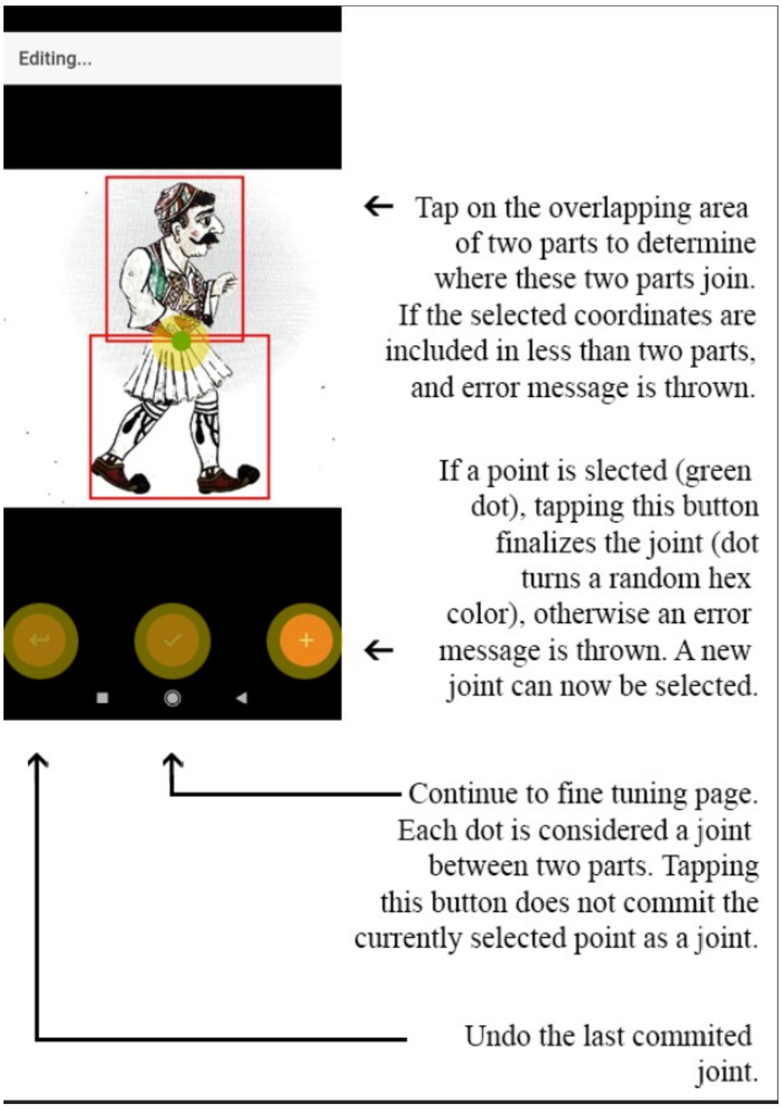
```
  findBestFitBox(box){
      let smallest = Infinity;
      let boxFound;
      for(let i=0;i<this.spaceBoxes.length;i++){
        if(this.spaceBoxes[i].w >= box.partSize.x &&
this.spaceBoxes[i].h >= box.partSize.y){
            let area = this.spaceBoxes[i].w * this.spaceBoxes[i].h;
            if(area < smallest){
                smallest = area;
                boxFound = i;
            }
        }
      }
      if(boxFound===undefined){
        alert("Parts are too large to fit. Please try with smaller
margins!");
        while(this.partListArray.length){
          this.partListArray.pop();
        }
      }
      return boxFound;
  }
```

4.3.3 Selecting Joints

After the Part Selection step, the user is prompted to define the joints between the parts of the previous step. During the transition between the two steps, the Sprite Sheet is updated to reflect the part array and the JSON template is updated with the parts' names and coordinates. We decided that displaying the Sprite Sheet instead of the initial image would be counter intuitive and more time consuming during this process, since it would require two clicks instead of one.

Checking if a joint is valid, in the sense that it must be inside of exactly two parts, is trivial, given that we maintain information about the original image in this step, so it is simply a matter of iterating through the whole part array and counting hits. If, at the end of this process, the count of hits is exactly two, then the joint is valid. The UI includes three main buttons that are, in order from left to right: Undo, Done, Commit Joint. The commit button, in this step, may not always succeed, as this is the point when the validity of the rule explained above is tested. If the joint does not meet the criteria, an error message is thrown, and the user is returned to an idle state with no joint currently selected. The functionality of each button will be explained in detail along with their relevant code below.

The relevant UI is indicated below for easier reference:



Tap on the overlapping area of two parts to determine where these two parts join. If the selected coordinates are included in less than two parts, and error message is thrown.

If a point is slected (green dot), tapping this button finalizes the joint (dot turns a random hex color), otherwise an error message is thrown. A new joint can now be selected.

Continue to fine tuning page. Each dot is considered a joint between two parts. Tapping this button does not commit the currently selected point as a joint.

Undo the last commited joint.

Figure 33: Joint Selection Page

In Section 4.3.1 we went into detail regarding the canvases and their functionality. During this step, Touch gestures are recognized on the canvas as indicated on the previous sections. The three functions presented below, implement the functionality that is described in the figure above. Note that only important parts of the functions are included, in order to keep the chapter organized and readable:

```
jointSelector(){
...
    let ctrlPtX = this.partListArray[0].endX -
this.partListArray[0].startX;
    let ctrlPtY = this.partListArray[0].endY -
this.partListArray[0].startY;
    for(let i = 0; i < this.partListArray.length; i++){
      let boxw = this.partListArray[i].endX-
this.partListArray[i].startX;
      let boxh = this.partListArray[i].endY-
this.partListArray[i].startY;
      let boxtemp = {x:boxw,y:boxh};
      this.figurePartList.push({
        partName: "Part"+i,
        partStartPoint: {x:0,y:0},
        partSize : boxtemp,
        isControlled : false
      });
      if(boxw==ctrlPtX && boxh==ctrlPtY){
        this.figurePartList[i].isControlled = true;
      }
      ….
  }
```

After filling the part array and clicking next, ePuppet is ready to receive user input regarding the joint selection. Whenever the user taps his finger on the canvas during this phase, he selects a point, which is highlighted in a green dot. The green got is the user's current selection and is not yet considered a joint that has been committed. Thus, any other new gesture on the canvas will remove the current selection in order to begin a new one. The functionality that allows the user to save his current selection as a joint is included in the following function. It should be noted that committing a joint will push it the part array which holds the saved joints and is used at the end of this step to update the Sprite Sheet and the JSON file accordingly. Whenever the commit button is pushed, this function is executed, and the current selection is redrawn as a random hex color dot to signify that this joint is now permanent.

```
keepJoint(){
    console.log("Hey from keep Joint!!!");
    let color = this.getRndColor();
    if(this.whatPartDidIClick(this.lastX,this.lastY) && this.jtPt1 !=
this.jtPt2){
      this.ctx2d4.clearRect(0,0,this.canvas4.width,this.canvas4.height);
      this.ctx2d3.beginPath();
      this.ctx2d3.arc(this.lastX,this.lastY,30,0,2*Math.PI);
      this.ctx2d3.fillStyle = color;
      this.ctx2d3.fill();
      this.tmpJtList.push({
        x: this.lastX,
        y: this.lastY
      });
      this.jointList.push({
        figurePart1: this.jtPt1,
        figurePart2: this.jtPt2,
        figurePart1Anchor : {x:this.jtNewX1,y:this.jtNewY1},
        figurePart2Anchor : {x:this.jtNewX2,y:this.jtNewY2},
      });
    }
    else{
      alert("Please, click on a valid part of the sprite sheet and/ or
different parts.");
    }
  }
```

`keepJoint()` is called whenever we commit a joint to the joint array. It draws the unique colored dot that signifies the joint on canvas #3, pushes the part coordinates in the `jointListArray` which holds the completed joints and, finally, resets canvas #4 and re enables touch events to prepare for the next selection. The if/ else clause deals with the problem of selecting an invalid joint. An invalid joint is a joint that its coordinates are not included in exactly two parts. The `whatPartDidIClick(this.lastX, this.lastY)` function is presented below and implements the aforementioned checks:

```
  whatPartDidIClick(x,y){
    let gottem = false;
    let bumpcount = 0;
    for(let i = 0; i < this.partListArray.length; i++){
      if((x>this.partListArray[i].startX &&
x<this.partListArray[i].endX)
      && (y>this.partListArray[i].startY &&
y<this.partListArray[i].endY)){
        bumpcount++;
        if(bumpcount==1){
          gottem = false;
          this.jtPt1 = this.figurePartList[i].partName;
          this.jtNewX1 = this.figurePartList[i].partStartPoint.x + (x -
this.partListArray[i].startX);
          this.jtNewY1 = this.figurePartList[i].partStartPoint.y + (y -
this.partListArray[i].startY);
        }
        else if(bumpcount==2){
          gottem = true;
          this.jtPt2 = this.figurePartList[i].partName;
          this.jtNewX2 = this.figurePartList[i].partStartPoint.x + (x -
this.partListArray[i].startX);
          this.jtNewY2 = this.figurePartList[i].partStartPoint.y + (y -
this.partListArray[i].startY);
        }
        else{
          gottem = false;
        }
      }
    }
    return gottem;
  }
```

Finally, `discardJoint()` is used if the user wants to undo the last selected part. By calling `jointListArray.pop()`, the last element of the array is removed and then we redraw the whole array from scratch. The function is displayed below:

```
  discardJoint(){
    this.jointList.pop();
    this.tmpJtList.pop();
    this.ctx2d3.clearRect(0,0,this.canvas4.width,this.canvas4.height);
    for(let i=0;i<this.jointList.length;i++){
      let color = this.getRndColor();
      this.ctx2d3.beginPath();

this.ctx2d3.arc(this.tmpJtList[i].x,this.tmpJtList[i].y,30,0,2*Math.PI);
      this.ctx2d3.fillStyle = color;
      this.ctx2d3.fill();
      this.ctx2d3.beginPath();
    }
  }
```

Once the joint array is complete, we move to the Final Editing Page. At this point we have created another PNG image that represents the Sprite Sheet. It includes non overlapping parts of the initial figure, along with a JSON file that details the part and joint arrays. During the final step, the Sprite Sheet is displayed for better visual clarity, as well as minor tweaks before finalizing the figure.

4.3.4 Final Editing Page

This is the final page before returning to the Home Page. The view is displaying the final Sprite Sheet that will be upload to eShadow. An assistive overlay can be toggled on/ off that highlights parts and joints. An eraser mode has been implemented to clear imperfections from the part dissecting phase.

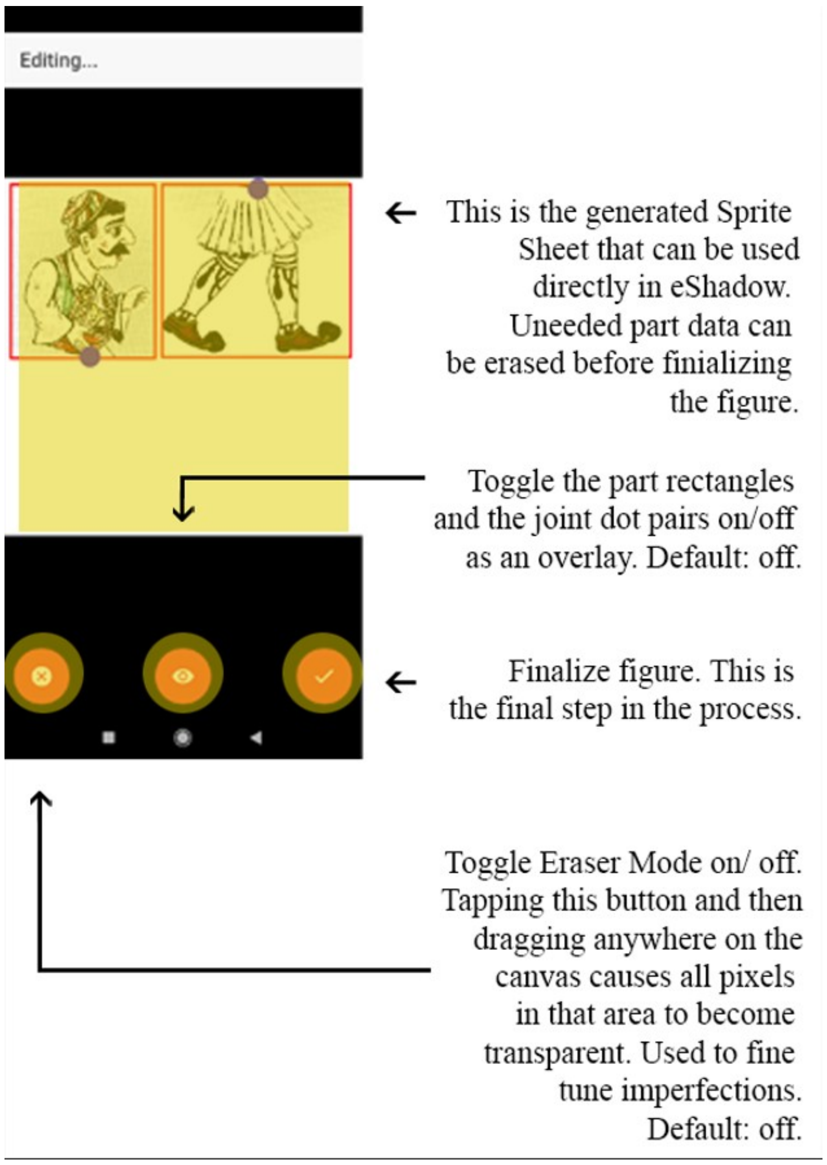

Editing...
This is the generated Sprite Sheet that can be used directly in eShadow. Uneeded part data can be erased before finializing the figure.
Toggle the part rectangles and the joint dot pairs on/off as an overlay. Default: off.
Finalize figure. This is the final step in the process.
Toggle Eraser Mode on/ off. Tapping this button and then dragging anywhere on the canvas causes all pixels in that area to become transparent. Used to fine tune imperfections. Default: off.

Figure 34: Final Edit Page with correction functionality and previewing the output Sprite Sheet

```
  eraseOnOff(){

    if(this.eraserFlag){
      this.toastEraser.dismiss();
      this.eraserFlag = false;
    }
    else{
      this.toastEraser = this.data.toastCtrl.create({
        message: 'You are in eraser mode.\nTouch to remove unnatural
edges in parts!',
        position: "top",
        dismissOnPageChange: true
      });
      this.toastEraser.present();
      this.eraserFlag = true;
    }
  }
```

The code displayed above is responsible for toggling Eraser mode on/ off. While eraser mode is ON, any gesture on the canvas results in the tapped area transforming into transparency. This functionality was implemented because, during the feedback collection phase, a lot of users pointed out that, often, while selecting the individual parts, a small area of other parts is incorrectly included in the rectangle. We, thus, provided a way to clear those imperfections in the last step.

4.4 Other Challenges

In this Section we will present miscellaneous points of interest that arose during development and cannot be categorized in the previous Sections of this chapter. These challenges involved the interaction of ePuppet with eShadow, along with the data management

4.4.1 Base64 Images and Blob Data Form

In Section 3.1.4 we explained the usage of the device's Camera to capture images for further processing. The method that realizes this, is presented below:

```
  takePicture(){
    this.data.presentLoading();
    this.cameraPreview.takePicture(this.pictureOpts).then((imageData) =>
{
        this.data.base64Image = 'data:image/png;base64,' + imageData;
        let newModal = this.modalCtrl.create(EditPage);
        newModal.present();
    }).catch((err) => {
      console.log("Error taking pic!", err);
    });
  }
```

The output of this function is a Base64 encoded string. Base64 is a mechanism to enable representing and transferring binary data in ASCII over mediums that allow only printable characters. This type of encoding is universal, meaning that that it can be interpreted by any and all computer systems. It can also be included in data URLs, which is a requirement for most software systems. Due to the aforementioned reasons, the camera defaults to this representation for the captured images.

However, Each Base64 digit represents exactly 6 bits of data. So, three 8-bits bytes of the input string/binary file (3×8 bits = 24 bits) can be represented by four 6-bit Base64 digits (4×6 = 24 bits). This means that the Base64 version of a string or file will be at least 133% the size of its source (a ~33% increase). The increase may be larger if the encoded data is small.

Furthermore, storing image data in the device storage or in the cloud (Google Drive etc.) is most commonly done in raw BLOB data form. In computers, a BLOB (binary large object), is a large file, typically an image or sound file, that must be handled (for example, uploaded, downloaded, or stored in a database) in a special way because of its size. The main idea about a BLOB is that the handler of the file (for example, the database manager) has no way of understanding the file in order to figure out how to deal with it. Based on this information, we have to implement a function that converts Base64 encoded images to BLOBs, in order to store them in the phone's storage or upload them to Google Drive. The code snippet below presents the described function:

```
  b64toBlob(b64Data: string, contentType: string, sliceSize: number) {
      contentType = contentType || '';
      sliceSize = sliceSize || 512;
      let byteCharacters = atob(b64Data);
      let byteArrays = [];
      for (let offset = 0; offset<byteCharacters.length;offset
+=sliceSize)      {
            let slice = byteCharacters.slice(offset, offset +
sliceSize);
            let byteNumbers = new Array(slice.length);
            for (let i = 0; i < slice.length; i++) {
                  byteNumbers[i] = slice.charCodeAt(i);
            }
            let byteArray = new Uint8Array(byteNumbers);
            byteArrays.push(byteArray);
      }
      let blob = new Blob(byteArrays, {type: contentType});
      return blob;
  }
```

4.4.2 Uploading Files on the Cloud

The output of our application is a folder containing three separate files that together construct the figure in eShadow. As discussed in the previous chapter, this folder can be found under the following path:

`InternalStorage/Android/Data/com.tuc.ePuppet/…`

To use those files on an eShadow supported device, we can currently connect our mobile phone with the device through USB, and move the files directly. Another way to achieve this, is by sending the files from our phone through e-mail or another application. Both of these methods are not ideal, since they assume that the user knows where files are stored in the Local Storage, and require that an external application is used. The solution to the aforementioned problem is implementing functionality in order to be able to upload the files directly through ePuppet. Since we built an Android application, it is safe to assume that users will have access to a Gmail account, as it is required for most of the phone's features. Thus, we decided on Google Drive as the supported Cloud. Users can log in to Google through the application's menu, and if the verification is successful, they can now upload individual figures. A list of available completed figures is shown on the

Home Screen, and the upload button becomes enabled after connecting to a Google account.

This process can be broken down to two distinct parts. Firstly, we need to authenticate the user with the Google servers. This is achieved using OAuth 2.0 flow. The steps we followed are presented below:

1. **Obtain OAuth 2.0 credentials from the Google API Console.**

   By visiting the Google API Console, we obtain OAuth 2.0 credentials such as a client ID and client secret that are known to both Google and our application. The set of values varies based on what type of application we are building.

2. **Obtain an access token from the Google Authorization Server.**

   Before the application can access private data using a Google API, it must obtain an access token that grants access to that API. A single access token can grant varying degrees of access to multiple APIs. A variable parameter called **scope** controls the set of resources and operations that an access token permits. During the access-token request, the application sends one or more values in the **scope** parameter.

   There are several ways to make this request, and they vary based on the type of application we are building. In the case of ePuppet, we simply open an IAB (In-App Browser) to redirect to Google.

   Some requests require an authentication step where the user logs in with their Google account. After logging in, the user is asked whether they are willing to grant one or more permissions that the application is requesting. This process is called **user consent**.

   If the user grants at least one permission, the Google Authorization Server sends the application an access token (or an authorization code that the application can use to obtain an access token) and a list of scopes of access granted by that token. If the user does not grant the permission, the server returns an error.

3. **Examine scopes of access granted by the user.**

   We then compare the scopes included in the access token response to the scopes required to access features and functionality of ePuppet, dependent upon access to a related Google API. This is imperative in order to use Google APIs in our application.

Having obtained the access token with the appropriate scope, we can now utilize it to make post requests to the Google Drive API. The endpoint used is presented below. Note that the `uploadType=multipart` attribute is required, as each upload attempt in ePuppet is a set of three files:

```
'https://www.googleapis.com/upload/drive/v3/files?uploadType=multipart'
```

The following code snippet indicates the HTTP post request we used to upload each file. It is worth noting that each file has to be converted to Blob form, using the function we presented in section 3.4.1 of this chapter, before the upload:

```
let payload = new FormData();
let meta = {name: dirEnt.name, mimeType: ".png"};
let iblob =
      this.b64toBlob(res.split("base64,")[1],'data:image/png;base64,',
512);
payload.append('metadata', new Blob([JSON.stringify(meta)], {type:
      'application/json'}));
payload.append('file', iblob);
iheaders.append('Authorization', 'Bearer ' + this.accessToken);


return this.http.post('https://www.googleapis.com/upload/drive/v3/files?
uploadType=multipart', payload, {headers: iheaders}).map(res =>
res.json()).subscribe(data =>{console.log(data);});
```

Chapter 5: Experimental Evaluation and Real Use Cases

Evaluation is an important step in the software development process in order to increase the reliability of the software. There are a number of different methods available to evaluate software that use different approaches to find outliers and errors, all with different requirements and possible results. In this thesis we have performed a series of tests on our own mobile application, ePuppet, developed for the Android platform. The results of our evaluation along with our experiences were reviewed and have helped us identify and implement enhancements on our application, both in terms of bug fixing, as well as quality of life changes. Most importantly, however, it assisted in defining what the end user expects from ePuppet and if the criteria of Ease of Use both in terms of accessibility and learnability are met.

5.1 System Usability Scale Questionnaire

The System Usability Scale (SUS) provides a quick, reliable tool for measuring the usability. It consists of a 10 item questionnaire with five response options for respondents; from Strongly agree to Strongly disagree (The System Usability Scale & How it’s Used in UX). SUS has become an industry standard. The noted benefits of using SUS include that it:

- Is a very easy scale to administer to participants
- Can be used on small sample sizes with reliable results
- Is valid – it can effectively differentiate between usable and unusable systems

When a SUS is used, participants are asked to score the following 10 items with one of five responses that range from Strongly Agree to Strongly disagree:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

Interpreting scoring can be complex. The participant’s scores for each question are converted to a new number, added together and then multiplied by 2.5 to convert the original scores of 0-40 to 0-100. Though the scores are 0-100, these are not percentages and should be considered only in terms of their percentile ranking.

Based on research, a SUS score above a 68 would be considered above average and anything below 68 is below average, however the best way to interpret your results involves "normalizing" the scores to produce a percentile ranking.

5.1.1 Experimental Process

To evaluate ePuppet, we decided to design an evaluation session within the context of the annual Science and Technology Day organized by TUC on 16/11/2019 as a means to find plausible users to test the application live. The pilot event was organized as a playful learning activity where the children first watched a special installation of eShadow having also the opportunity to interact with digital puppets using back projection and body shadow. After that, they were offered materials to create their own puppets on paper and finally they were able to digitize their creations using ePuppet, upload them in eShadow and use them to make short improvisations. After finishing the activity, the students were asked to fill the System Usability Scale (SUS) questionnaire. The process involved the following sequence:

- Users could either color-fill premade copies of figures or draw their own. This was used to simulate a realistic usage scenario as most children loved the concept of animating their own puppet.
- Afterwards, we provided the mobile devices in order to start the digitization process. We walked everyone through the application flow, answered any questions and helped create the digital puppet.
- Finally we imported the figures in eShadow via Google Drive and let the users play with their puppets.

The questionnaires were handed after the previous sequence in order to evaluate the ease of use and the quality of the application. In the following section we will detail the results of these questionnaires.

5.1.2 Results

In Section 4.1.1 we detailed how to analyze the results of SUS questionnaires. Below, we present the aforementioned results and the conclusions we were able to draw based on them.

It is worth noting that, although, post calculation, the scores range from 0 to 100, the results are still misleading. This occurs because the 0 – 100 scale refers to percentile ranking and not percentages. So, in effect, a score of 68 is considered 'average' and a score of 50 is considered 'below average'. The following figure displays the most widely used grading system for the SUS (Measuring usability with the System Usability Scale (SUS) by Jeff Sauro):

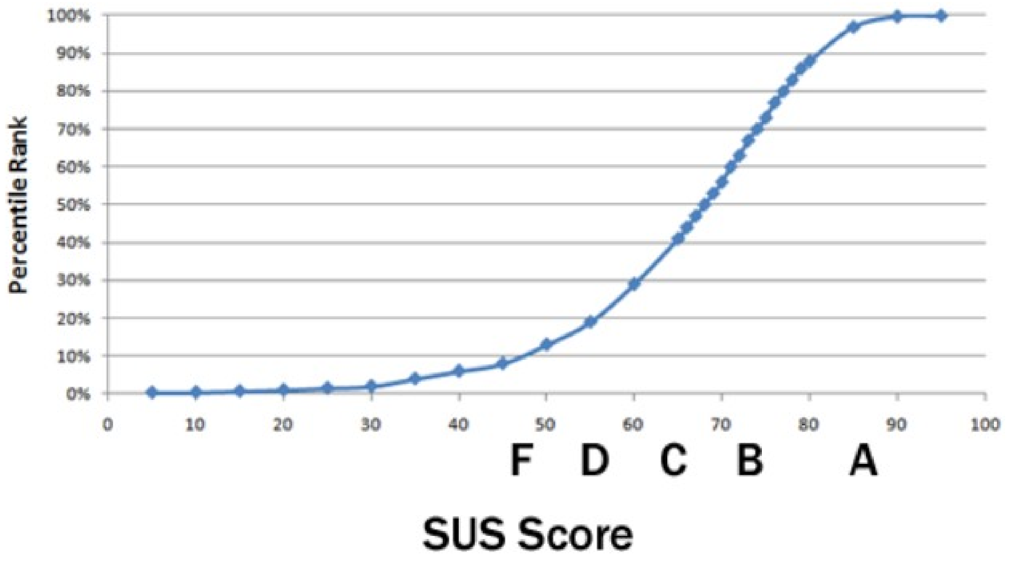
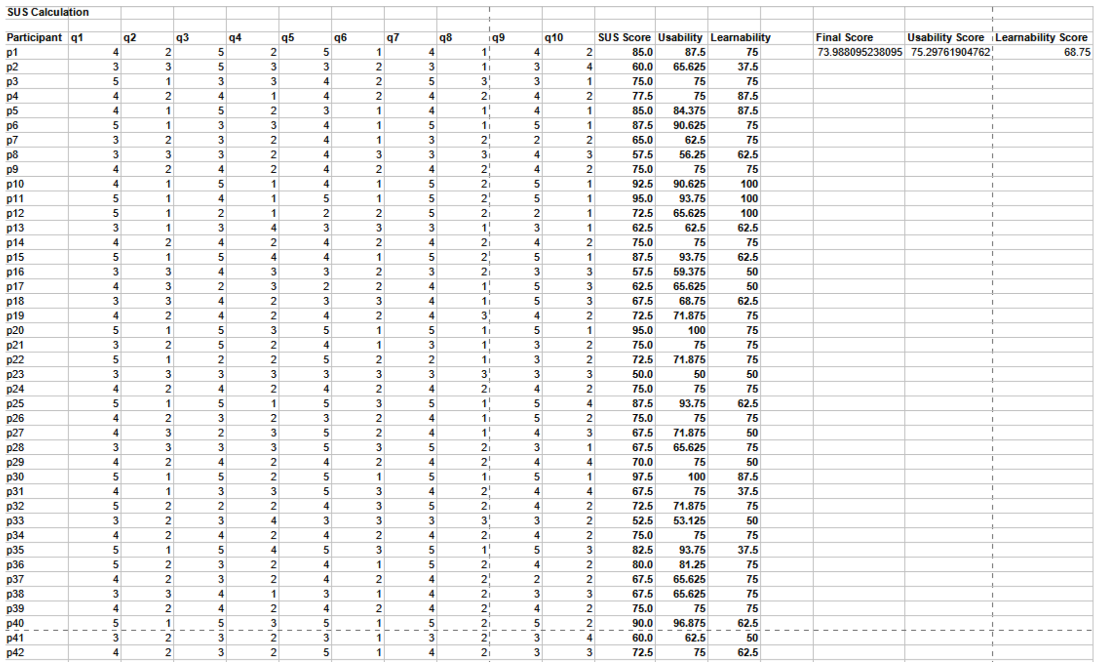


Figure 35: Percentile rankings associated with SUS scores and letter grades.

During the testing day, we accumulated 42 valid questionnaires, with almost an even split between children and adults. Below, we present the basic processing done in Excel:



SUS Calculation

Participant	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	SUS Score	Usability	Learnability		Final Score	Usability Score	Learnability Score
p1	4	2	5	2	5	1	4	1	4	2	85.0	87.5	75		73.988095238095	75.29761904762	68.75
p2	3	3	5	3	3	2	3	1	3	4	60.0	65.625	37.5				
p3	5	1	3	3	4	2	5	3	3	1	75.0	75	75				
p4	4	2	4	1	4	2	4	2	4	2	77.5	75	87.5				
p5	4	1	5	2	3	1	4	1	4	1	85.0	84.375	87.5				
p6	5	1	3	3	4	1	5	1	5	1	87.5	90.625	75				
p7	3	2	3	2	4	1	3	2	2	2	65.0	62.5	75				
p8	3	3	3	2	4	3	3	3	4	3	57.5	56.25	62.5				
p9	4	2	4	2	4	2	4	2	4	2	75.0	75	75				
p10	4	1	5	1	4	1	5	2	5	1	92.5	90.625	100				
p11	5	1	4	1	5	1	5	2	5	1	95.0	93.75	100				
p12	5	1	2	1	2	2	5	2	2	1	72.5	65.625	100				
p13	3	1	3	4	3	3	3	1	3	1	62.5	62.5	62.5				
p14	4	2	4	2	4	2	4	2	4	2	75.0	75	75				
p15	5	1	5	4	4	1	5	2	5	1	87.5	93.75	62.5				
p16	3	3	4	3	3	2	3	2	3	3	57.5	59.375	50				
p17	4	3	2	3	2	2	4	1	5	3	62.5	65.625	50				
p18	3	3	4	2	3	3	4	1	5	3	67.5	68.75	62.5				
p19	4	2	4	2	4	2	4	3	4	2	72.5	71.875	75				
p20	5	1	5	3	5	1	5	1	5	1	95.0	100	75				
p21	3	2	5	2	4	1	3	1	3	2	75.0	75	75				
p22	5	1	2	2	5	2	2	1	3	2	72.5	71.875	75				
p23	3	3	3	3	3	3	3	3	3	3	50.0	50	50				
p24	4	2	4	2	4	2	4	2	4	2	75.0	75	75				
p25	5	1	5	1	5	3	5	1	5	4	87.5	93.75	62.5				
p26	4	2	3	2	3	2	4	1	5	2	75.0	75	75				
p27	4	3	2	3	5	2	4	1	4	3	67.5	71.875	50				
p28	3	3	3	3	5	3	5	2	3	1	67.5	65.625	75				
p29	4	2	4	2	4	2	4	2	4	4	70.0	75	50				
p30	5	1	5	2	5	1	5	1	5	1	97.5	100	87.5				
p31	4	1	3	3	5	3	4	2	4	4	67.5	75	37.5				
p32	5	2	2	2	4	3	5	2	4	2	72.5	71.875	75				
p33	3	2	3	4	3	3	3	3	3	2	52.5	53.125	50				
p34	4	2	4	2	4	2	4	2	4	2	75.0	75	75				
p35	5	1	5	4	5	3	5	1	5	3	82.5	93.75	37.5				
p36	5	2	3	2	4	1	5	2	4	2	80.0	81.25	75				
p37	4	2	3	2	4	2	4	2	2	2	67.5	65.625	75				
p38	3	3	4	1	3	1	4	2	3	3	67.5	65.625	75				
p39	4	2	4	2	4	2	4	2	4	2	75.0	75	75				
p40	5	1	5	3	5	1	5	2	5	2	90.0	96.875	62.5				
p41	3	2	3	2	3	1	3	2	3	4	60.0	62.5	50				
p42	4	2	3	2	5	1	4	2	3	3	72.5	75	62.5				

Figure 36: SUS results processing in Excel

The first thing of note is the Final Score, which is the average of each individual SUS score for each user. This score is an estimation of the usability of the application from the perspective of the user. By scoring a 74, which is above the aforementioned standard of 68, we could deduce that the application was usable by the audience at the time of this questionnaire. As shown in Figure 18, the application scored a 74, which places it in the 70% percentile rank, meaning that the product has higher perceived usability than 70% of all products tested with this scale.

However, the above evaluation is vague, since the product was usable under our guidance in real time. Delving deeper in the SUS questionnaires, we can notice that questions 4 and 10 are specifically pointing to the learnability of the product:

4. I think that I would need the support of a technical person to be able to use this system.

10. I needed to learn a lot of things before I could get going with this system.

The rest determine its usability. Thus, we split the two qualities and calculated two new scores that evaluate each of those separately. At this point, the scoring interpretation had to change to reflect the new values as Usability could score from 0 to 32 and Learnability from 0 to 8. Thus, the initial function to calculate the SUS score for each user had to change to reflect the usability and learnability relevant scores.

In figure 34, the Usability and Learnability scores are presented as different averages. Viewing the results in this manner, we can see that the actual usability score is higher that the Final Score by one unit, but the learnability aspect scored a 68.75 which is considered average in the scope of this testing method. This observation led to further discussion with testers about the need to include a tutorial of the figure creation process, which improved users' experiences significantly.

5.2 Real Use Cases

At the time of writing this thesis, ePuppet has already been used in real life scenarios in schools by students and teachers to create custom digital puppets. In this Section we will present two major projects that were realized through the use of ePuppet.

5.2.1 Kindergartens, Thessaloniki

An opportunity for initial testing arose during a collaboration with kindergarten teachers in Thessaloniki. This collaboration took place before the SUS evaluation and acted as our first reference point for enhancements and changes in ePuppet. These teachers wanted to use eShadow to create a virtual performance with figures created by their pupils during their study case (Hatzigianni et al., 2021).

Four public kindergartens (four to six year old children) from the prefectures of Thessaloniki and Imathia in Northern Greece (Macedonia region) participated in this study. Two of the kindergartens were situated in an urban area and two were situated in a rural area. Seven female kindergarten teachers with a mean age of 46 years and a mean of 16.5 teaching years, and 76 kindergarten children and their parents participated in the study (41 girls and 35 boys). The community (e.g. administrators, museums) around each kindergarten was also informed about the aims of the study.

Young children did not yet have the digital skills to create their own digital puppets using image processing software, as it was done with older children, usually with the help of a computer science teacher (Moumoutzis et al.,2017). They needed a simpler and more intuitive way of developing new digital puppets, as well as the need to combine this process with the traditional process of producing paper-based puppet We introduced them to ePuppet as a means to digitize the children's paintings easily.

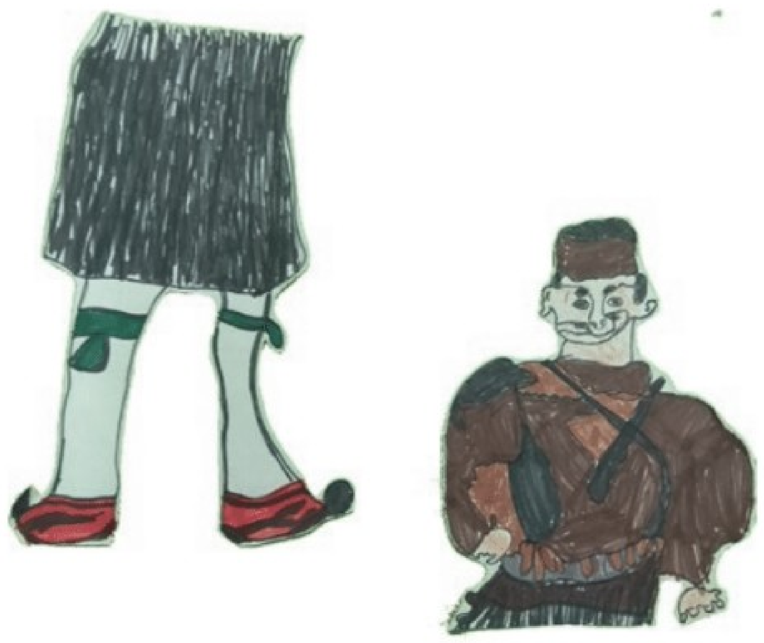


Figure 37: Figures created by students.

Initially, they familiarized themselves with the application and, under our guidance, created basic figures and imported them in eShadow. After this stage, they started importing the real drawings as figures. During this process we received invaluable feedback for the application is a real life scenario. Children were given ample opportunities to experiment with the software and the app. By the end of this phase, the digitization of the puppets and the preparation of the digital play were both completed. In addition, children practiced moving and ‘acting’ with their figures in the eShadow environment.

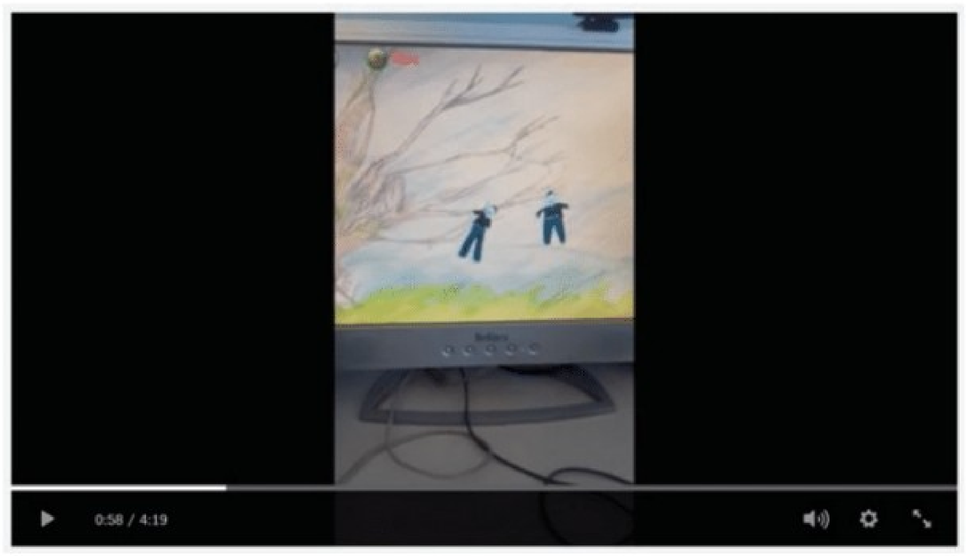


Figure 38: Children play with their puppets in eShadow.

Finally they offered to answer questions regarding their experience with the application individually. Each of these interviews followed the same pattern:

- Did the application succeed in simplifying the task at hand?
- Did the user need guidance in any of the steps of the creation process?
- If yes, which of these steps were unclear?
- Are there any features they felt were missing?

These interviews assisted immensely with the evaluation of the application, since the testers had freedom of time and space to discuss any problems that arose, as well as offer suggestions about the features included from a consumer standpoint.

5.2.2 Christmas Edition

Another opportunity arose during the Christmas of 2019 from the Mixed Puppetry Techniques Center (https://kemitek.gr/). This event included various projects such as sketching and drawing lab, analog and digital figure creation, and shadow theater performance creation. We collaborated with Paul Nowak for the digital figure creation event by developing an additional Christmas themed version of ePuppet to distribute during the event.

This event was a challenge since none of the us were present on the day. Paul familiarized himself with the platforms under our guidance and provided feedback as a dedicated user over the month leading to the event. He introduced the new users to eShadow and ePuppet with tremendous success. Throughout the day multiple families created puppets from sketches and participated in interactive performances with them. Some results of their work can be seen in the figure below:

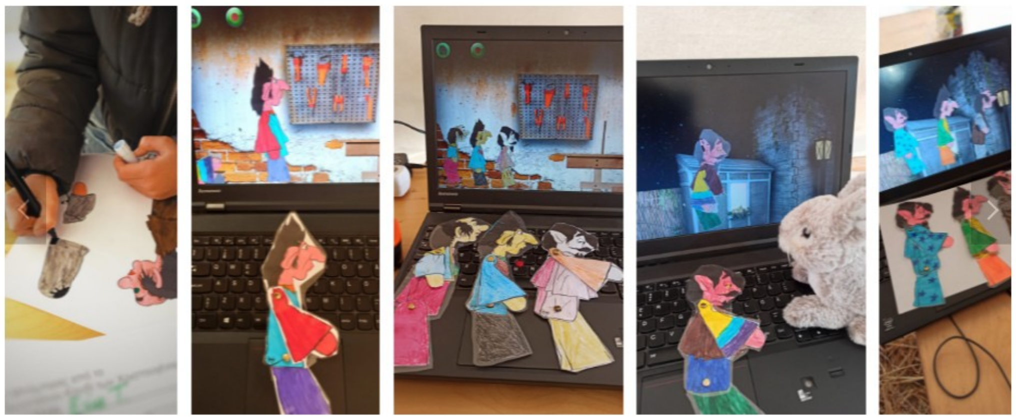


Figure 39: Various stages of the Figure creation process during the event.

Chapter 6: Conclusion

The complete development process that was undertaken with this thesis will be summarized in the following paragraph. Initially we defined the scope of ePuppet, by identifying where it fits in the current ecosystem of existing platforms, namely, eShadow and eShadow Editor. This step helped us decide on core design decisions such as the targeted platforms (Android Mobile, iOS, etc.) and ultimately choose the correct framework and software stack for this project. The next step involved defining the functional requirements of ePuppet which was realized through the use of two main techniques. Initially we designed use cases for every functional requirement we had anticipated and we then created personas that reflected the average users going through each specific use case scenario.

We then begun the implementation process, based on the required functionality. The main challenges we encountered were primarily during the image processing steps. We tackled the background transparency problem by scanning the outer edges of the picture and using the pixel data as a reference of what is considered background. We then used the HTML 5 canvas to further edit the picture in order to extract the Sprite Sheet and the JSON file. We overcame this step by stacking canvases on top of each other so as to maintain all the relevant information while creating the Sprite Sheet.

After developing a working prototype, we sought willing users to test ePuppet and provide feedback about existing and additional functionality. This was realized in many stages, ranging from personal presentations for interested teachers to large scale during the TUC open day and the Christmas edition event. Feedback is continuously provided currently as the application is used in schools and puppetry projects and is integrated in new versions of ePuppet.

It seems that ePuppet is on a good track in expanding eShadow, as it includes the users in the figure creation process, making users more engaged in the platform both for educational as well as entertainment purposes. This project will always be a work in progress as it evolves beside eShadow but the current iteration of the application is fully usable with minimal prior knowledge by the user. The main technical challenge of this work was the implementation of a potent figure creation tool, without compromising the ease of access (usability, learnability) given our intended user group.

6.1 Contributions

As we have mentioned above, the process of creating a figure of our application is perfectly adapted to the structure of the Greek Shadow Theater, a fact that in itself is an innovation. In particular, ePuppet overcomes the obstacles faced by students and teachers with low digital skills to develop their own digital puppets and opens up new opportunities for creativity and learning. It provides a means to digitize any figure that users can imagine and thus, enhances the eShadow experience and engagement with the platform.

ePuppet employs a model for digital puppets that directly correspond to the structure of analog puppets of the Greek Shadow Theater of Shadows. It provides functionality for easy composition of personalized figures either following a certain pre-defined structure (2- and 4-part puppets) or general puppets that can consist of any number of parts in any configuration. Note here due to the design of the application and the its architecture it is also possible to extend the set of predefined puppet type to accommodate new templates with minimal implementation effort.

Providing the above flexibility, ePuppet can be effectively used to create digital puppets that go beyond the themes found in traditional Shadow Theatre scenarios. Such new themes can be mythology, folklore, children's literature, fairy tales and more. The use of ePuppet that is reported in this thesis, showcase some indicative cases in these directions.

Compared to similar ready-made figure creation applications, ePuppet offers a great deal of creative freedom that combines the merits of analog puppet creation on paper or any other appropriate material and the easy of digitization. ePuppet effectively facilitates the creation of digital puppets by engaging its users in a creative workflow within which they first develop analog shadow puppets, thus offering them a learning experience that links traditional shadow theatre to digital technologies. The evaluation results clearly demonstrate the usability of ePuppet and its use. ePuppet, apart from the reported evaluation, is used in many schools within the context of project-based learning activities to enable students and teachers to actively explore various themes ranging from social topics to STEM learning in an engaging and creative manner.

ePuppet, contributes to education through the development of children's teamwork, understanding the composition and creation of an image, working with tangible materials, digitizing drawings, and the development of technological skills (especially in the first grades of primary school). These contributions have been thoroughly documented a number of scientific papers that are listed here:

- Moumoutzis, N., Koukis, A., Christoulakis, M., Maragkoudakis, I., Christodoulakis, S., & Paneva-Marinova, D. (2019, October). PerFECt: a performative framework to establish and sustain onlife communities and its use to design a mobile app to extend a digital storytelling platform with new capabilities. In *Interactive Mobile Communication, Technologies and Learning* (pp. 1002-1014). Springer, Cham.
- Hatzigianni, M., Gregoriadis, A., Moumoutzis, N., Christoulakis, M., & Alexiou, V. (2021). Integrating Design Thinking, Digital Technologies and the Arts to Explore Peace, War and Social Justice Concepts with Young Children. In

Embedding STEAM in Early Childhood Education and Care (pp. 21-40). Palgrave Macmillan, Cham.

- Moumoutzis N., Koukis A., Xanthaki X., Christoulakis M., Maragkoudakis I., Christodoulakis S., Paneva-Marinova D., Pavlova L. (2021) EPuppet: A Mobile App to Extend a Digital Storytelling Platform with New Capabilities. Internet of Things, Infrastructures and Mobile Applications. IMCL 2021. To appear.

6.2 Future Work

Although the project is fully functional and is currently being used by students and teachers, some of the functionality can be improved or extended.

The main method that can be reworked in the future, is the background extraction step. In this step we used a custom algorithm based on observation of the captured images. State of the Art suggests that machine learning is, in the long term, always the best possible solution for such a problem. Although machine learning requirements are steeper, both in terms of computational resources and in time spent teaching and calibrating the algorithm, the final process should be improved significantly. This would greatly enhance the ePuppet experience, given that the restrictions regarding the captured image (single color background, ambient light source) would be reduced or completely eliminated.

Some quality of life features that can be added in the future are additional languages, so that students are not required to understand English to use this application unsupervised, and wider platform support, to cover iOS and Windows mobile. Building for other platforms is an easy task, given that Ionic wraps the existing code in a native wrapper without the need for redeveloping the application. However, we did not have the resources (iOS, Windows devices) to test on other platforms.

Currently we do not support editing or changing an already finished figure. Only previewing the finished figure is supported. This functionality would improve the flexibility of the application and provide a means for post creation editing. This feature, along with collaboration elements that will be explained below, could transform ePuppet in a shared workspace for students to work on puppets collectively, highlighting cooperation and teamwork on a class level. The project could be expanded on a new direction, by implementing collaboration features during the figure creation process. This functionality can include team editing of a figure in a class, or teacher assistance and guidance.

References

Christoulakis, M., Pitsiladis, A., Moraiti, A., Moumoutzis, N., & Christodoulakis, S. (2013). eShadow: a tool for digital storytelling based on traditional Greek shadow theatre. In *workshop Proceedings of the 8th International Conference on the Foundations of Digital Games*.

Hatzigianni, M., Gregoriadis, A., Moumoutzis, N., Christoulakis, M., & Alexiou, V. (2021). Integrating Design Thinking, Digital Technologies and the Arts to Explore Peace, War and Social Justice Concepts with Young Children. In *Embedding STEAM in Early Childhood Education and Care* (pp. 21-40). Palgrave Macmillan, Cham.

Hatzigianni, M., Miller, M. G., & Quinones, G. (2016). Karagiozis in Australia: Exploring principles of social justice in the arts for young children. *International Journal of Education & the Arts*, *17*(25).

Lantz, J. L., Myers, J., & Wilson, R. (2020). Digital Storytelling and Young Children: Transforming Learning Through Creative Use of Technology. In *Handbook of Research on Integrating Digital Technology With Literacy Pedagogies* (pp. 212-231). IGI Global.

Measuring usability with the System Usability Scale (SUS) by Jeff Sauro: https://measuringu.com/sus/ (accessed Sep. 3, 2021)

Moraiti, A., Moumoutzis, N., Christoulakis, M., Pitsiladis, A., Stylianakis, G., Sifakis, Y., ... & Christodoulakis, S. (2016, October). Playful creation of digital stories with eShadow. In *2016 11th International Workshop on Semantic and Social Media Adaptation and Personalization (SMAP)* (pp. 139-144). IEEE.

Moumoutzis, N., Christoulakis, M., Christodoulakis, S., & Paneva-Marinova, D.: Renovating the Cultural Heritage of Traditional Shadow Theatre with eShadow. Design, Implementation, Evaluation and Use in Formal and Informal Learning. In DiPP 2018 Conference on Digital Presentation and Preservation of Cultural and Scientific Heritage. Vol. 8, Sofia, Bulgaria: Institute of Mathematics and Informatics–BAS, 2018, pp. 51-70, ISSN 1314-4006(Print), e ISSN 2535-0366 (Online)(2018).

Moumoutzis, N., Koukis, A., Christoulakis, M., Maragkoudakis, I., Christodoulakis, S., & Paneva-Marinova, D. (2019, October). PerFECt: a performative framework to establish and sustain onlife communities and its use to design a mobile app to extend a digital storytelling platform with new capabilities. In *Interactive Mobile Communication, Technologies and Learning* (pp. 1002-1014). Springer, Cham.

Moumoutzis N., Koukis A., Xanthaki X., Christoulakis M., Maragkoudakis I., Christodoulakis S., Paneva-Marinova D., Pavlova L. (2021) EPuppet: A Mobile App to Extend a Digital Storytelling Platform with New Capabilities. Internet of Things, Infrastructures and Mobile Applications. IMCL 2021. To appear.

References

The System Usability Scale & How it’s Used in UX: https://medium.com/thinking-design/the-system-usability-scale-how-its-used-in-ux-b823045270b7(accessed Sep. 3, 2021)

Glossary of Acronyms

SUS - System Usability Scale

API - Application Programming Interface

SDK - Software Development Kit

APK - Android Application Package

APP - Application

UI - User Interface

UX - User Experience

JSON - JavaScript Object Notation

TUC - Technical University of Crete

Wi-Fi - Wireless Fidelity

CLI - Command Line Interface

IAB - In App Browser

BLOB - Binary Large Object

MSV – Model – View - Controller