ProteinAR

an iOS Application for Protein Visualisation and Design in Augmented Reality

A dissertation submitted to the National University of Ireland, Cork
in partial fulfilment of the requirements for the degree of
Master of Science in Interactive Media
in the College of Science, Engineering and Food Science

2020

By
Thao Phuong Le
School of Computer Science & Information Technology

Contents

List of Tables

List of Figures

Abstract

Dummy text

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Acknowledgements

I would like to thank...

Chapter 1

Introduction

In recent years, along with the advancement of technology, there are major advances in molecular biology. Technology has become a great help for scientists and biologists aiding their research and make things easier to study. This project focuses on protein structure displaying and protein structure design.

Protein is not a single substance. There are many different proteins in an organism or in a cell that comes in every shape and size, doing a unique and specific job ("Introduction to proteins and amino acids (article) — Khan Academy", n.d.). Proteins are considered as the "ultimate players in the processes that allow an organism to function and reproduce" (Stephenson, 2016).

Proteins are formed by linear chains of amino acids. A linear chain of protein is called a polypeptide. Each protein is formed by one or more of polypeptide chains, linked together in a specific order ("Introduction to proteins and amino acids (article) — Khan Academy", n.d.). Protein are the fundamental components of all living cells (Widłak, 2013). Protein has a countless number of functions in a cell or organism that are extremely important in the biology of many organisms. They form enzymes to speed the reactions up by break-down, link-up, or rearrange the substrates ("Introduction to proteins and amino acids (article) — Khan Academy", n.d.). They from hormones to control specific physiological processes such as "growth, development, metabolism and reproduction" ("Introduction to proteins and amino acids (article) — Khan Academy", n.d.). To maintain these roles, the shape of a protein is critical. If the shape changes, the protein will lose its functionality. There are four levels of protein structure: primary, secondary, tertiary, and quaternary ("Introduction to proteins and amino acids (article) — Khan Academy", n.d.). Knowing the structure of a protein makes understanding how that protein works much easier. By being able to manipulate a structure of a protein, scientists can create hypotheses about how to affect, control modify them to, for example, design mutations to change functions.

This year of 2020 has once again proven the importance of molecular biology study. As of this year, we all have experienced the Severe Acute Respiratory Syndrome Coronavirus-2 (SARS-CoV-2) as is it "a newly emerging, highly transmissible and pathogenic coronavirus in humans that cause the global public health emergencies and economic crises" (Mittal et al., 2020). The number of infections worldwide had reached millions, including thousands of

deaths. To find a cure, much researches have been carried out. Some research developed on the protein structure of SARS-CoV-2 has provided some insights into its evolution. As Wiesława has pointed out: "The chief characteristic of proteins that allows their diverse set of functions is their ability to bind other molecules (proteins or small-molecule substrates) specifically and tightly." (Widłak, 2013). The characteristic of SARS-CoV-2 is the protein spikes that cover the surface. The virus uses this to bind with and enter human cells (Wrobel et al., 2020). The spike of SARS-CoV-2 is very stable and thus help to bind to human cell tightly. Therefore, analysing the structure of theses spikes could provide clues about the virus's evolution. The study of the structure of the spike protein can aid with drug discovery and vaccine design. Understanding the new importance of implementing IT in Biology research, this project aims to aid with protein structural study and raise interest in protein design. Due to the shortage of time and lack in experience, this project only provides the first step into bringing the visualisation of protein into AR-display and creating simple protein structure in an iOS application using the framework ARkit. The main goal of this project, however, is to **visualise protein structure on AR using an iOS App and let user interact with the structures.** There are various previous studies on protein visualisation on 3D and VR, however, there has not been much on AR, especially AR app on iOS. This project proposed the implementation of displaying protein structures to serve as a trial for future study and research as it might make displaying more appealing than simple 3D and also cut down on the side effects of VR. All the protein models that are to be displayed are retrieved from RCSB Protein Data Bank.

The app aims to visualise protein structures in two ways. First way is to directly display the complex protein models from RCSB and the second way is visualising the design of simple protein structure. The second function is implemented so that this project's application is not only appealing to biologists and scientists but also can be used by anyone curious about biology. Being able to construct a protein structure as a mini-game might make it easier for users to understand more about protein structure.

In this project, a mobile application for iOS system was developed: ProteinAR. ProteinAR has two main categories: **Education** and **Mini-game**. The **Education** category assumes that the users have already known about proteins, they can input the name of protein and get the 3D visualisation of the protein structure in AR. Users can study the protein by zooming in, turning, flipping the protein structure. Due to the complexity of protein structure, even under microscope it is not easy to look through. Thus, being able to zoom in and all other interactions will definitely benefit researchers. Moreover, since this is on a mobile app, users can interact and discuss the structure with other users at the same time, which can be considered as a promising tool for study and research on proteins.

The **Mini-game**, on the other hand, is user-friendly to users who are not familiar with proteins or biology in general. Users can add the polypeptide chains namely: Flex Coil, Rig Coil, Helix, Sheet onto each other to create a protein. This might make the concept of protein sounds more appealing to the user and thus, motivate the wish to study more about protein. In this game, users are also able to interact with the polypeptide chains and protein models.

Besides, ProteinAR also integrates other functions to make the app more interesting such as enable taking photo of the proteins, or recording the video during the process as well as

provide users some information about protein.

This paper will elaborate the background and research, the problems and solutions, design and implementation and the final evaluation of the project. In the short period of time and the given circumstance of Covid-19, there were some limitations to the project, which would also be mentioned in the paper.

Finally, the paper will conclude some critical points in dealing with the fairly new AR technology, especially using ARKit on: - The feasibility of retrieving and displaying PDB contents - The usability of the app (AR) - Room for future work

Some important technical notes about the project: ProteinAR was designed on Xcode 12, based on Swift, on MacBook OS version: Catalina 10.5.5. There is no support for AR on MacOS, thus, the built-in simulator will not be able to display the AR function and can cause some other errors. The project was run and tested on an iPhone. The attached demo video is recorded on iPhone X, iOS version 14. Other versions of Xcode or macOS or iOS might not be able to get ProteinAR running and thus might generate some unwanted errors.

Chapter 2

Analysis: Review & Research on the field and existing products

2.1 Introduction to Protein

As mentioned, proteins are "the most important macromolecules in all living organisms" (Rashid et al., n.d.). Sequences of amino acids that bounds into linear chains create proteins. These chains have a specific folded three-dimensional (3D) shape, which enables the protein to perform a certain task (Rashid et al., n.d.).The shape of the protein defines the tasks of it, thus, knowing the protein structure is very important. There are four different levels of protein structures: Primary Structure, Secondary Structure, Teritiary Structure, and Quaternary Structure. A sequence of amino acids in a chain form a *primary structure*. These chains, then, would fold into three different shapes of Helix, Coil or Sheet where the alpha helix, the beta sheet, and the random coils are positioned, which is called the *secondary structure*. The combinations of these formed chains of helix, coil and sheet (polypeptide chains) would form a 3D structure – the *tertiary structure* of a protein. The *quaternary structure* is a large assembly of multiple polypeptide chains (Figure 2.1).

To understand a protein's function, understanding the structure of the protein is necessary. In the same way, designing a protein from the structure will help to design its function. In this app, users will get to design protein structure by combining different protein secondary structures of helices, coils, and sheets to form tertiary structures.

2.2 Existing Solutions to Protein visualisation

"Proteins are three-dimensional (3D) objects" (Ratamero et al., 2018). The key to understanding protein functions is to understanding protein structure. Computer models for protein has become very popular for a long time. Many projects were developed to make 3D viewing of protein possible such as PYMOL, CHIMERA, VMD, ISOLDE, etc.

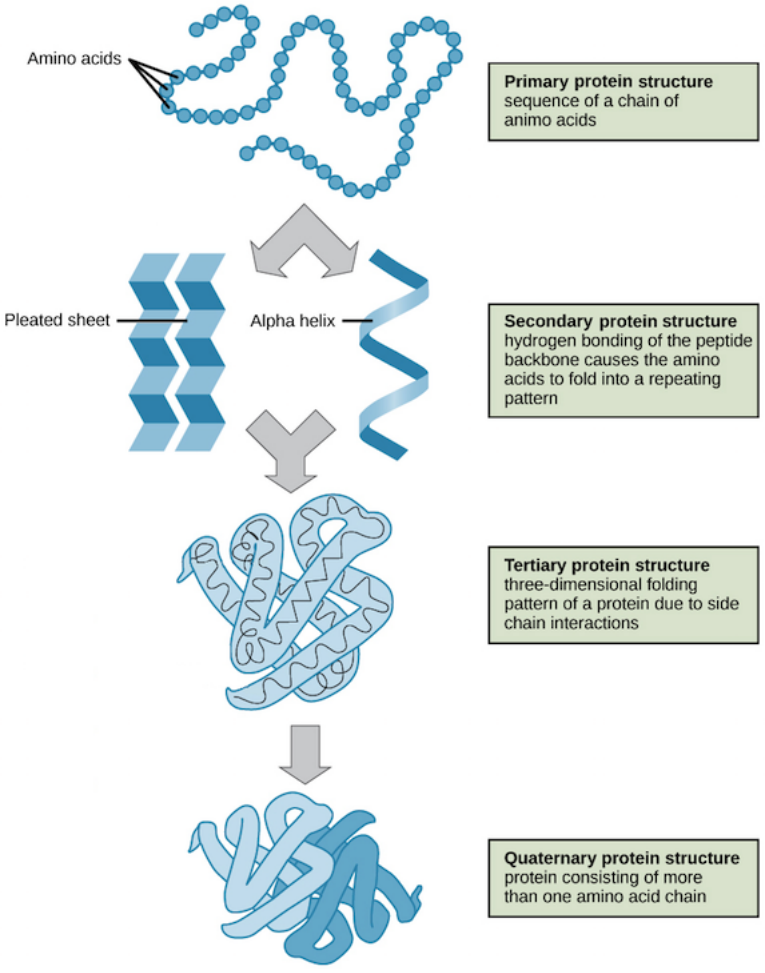

Figure 2.1: Orders of protein structure - source Khan Academy("Introduction to proteins and amino acids (article) — Khan Academy", n.d.)

2.2.1 Protein Visualisation in Mobile Applications

There are numerous mobile applications in which protein are visualised in 3D. The RCSB Protein Data Bank (the single worldwide repository of protein data) also provide a mobile app to provide data access and visualisation. Basically, the protein can be downloaded directly from the PDB from RCSB and displayed in 3D. This app is based on the open-source molecular viewer NDKmol. However, NDKmol can only be used on Android and not iOS. Jmoll is another Android app that connects to the RCSB PDB, visualising the protein in 3D once the protein name is typed in. There are some molecule viewers that can run on iOS devices. Unfortunately, most of them are no longer in used or was having troubled, thus, being removed from the Apple App Store. iMolview can still be used, however, the interface is not very user friendly.

2.2.2 Protein Visualisation in VR

The advancement of implementing VR in Protein Display

Visualising protein on computer in 3D has been a great step, however, it still lacked the immersion and a true feeling of 3D presence, leads to limitation in analysing protein structure. Virtual Reality (VR) provides a wide field of view on an immersive display and a better perception of the protein structure by head-tracking. Furthermore, VR enables user to have the freedom of hand controllers for simple manipulation and interaction with the protein instead of the conventional manipulation on 2D using trackpad, mouse and keyboard (Goddard et al., 2018). This makes VR entrance into the world of protein visualising/molecular biology more than welcomed. HMDs[1] are commonly used because they are easy to use and becoming more and more common and affordable. VR games have become popular, thus the tools for programming software that are compatible with HMD are better and cheaper. Project such as REALITYCONVERT, AUTODESK, MOLECULE VIEWER are well developed, providing good resource for further development on protein display in VR (Ratamero et al., 2018). UNITY is largely used with the combination of HMDs such as OCULUS RIFT and HTC VIVE to display and manipulate protein (Ratamero et al., 2018).

There have been many advanced projects of implementing VR in molecular biology. The MOLECULAR RIFT – an open source tool that creates a virtual reality environment steered with hand movements, incorporate OCULUS RIFT as the display to create the virtual setting (Norrby et al., 2015). The combination of a virtual reality experience with natural acts such as hand movements creates a much better experience for the users than just experiencing the 3D (Norrby et al., 2015).

Other research shows that the technology in displaying Protein in VR is advanced, however, tools that are designed to be installed on desktop systems are often tedious (K. Xu et al., 2019). The configurations might be different with systems and therefore causing errors and sharing between system is difficult. With the help of Web Graphics Library (WebGL), web-based applications such as JMOL, ASTERVIEWER are more straightforward as VR experiences can be directly accessed with common web browsers. However, there are many limitations for these web-based applications because it would only support a few file types and cannot perform

[1]Head Mounted Display

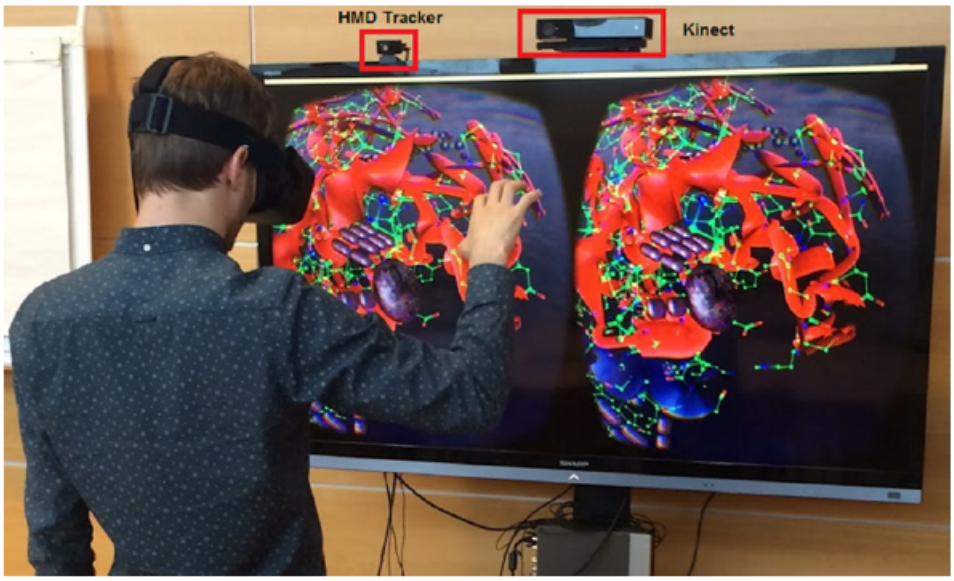

Figure 2.2: Oculus Rift (HMD) and Kinect v2 sensor placement used during Molecular Rift development

complex tasks for analytical purpose (K. Xu et al., 2019). A few solution of an integrative cloud-based system that can directly access databases and uses VR technology to visualise and analyse macromolecular structures were proposed, such as VRMOL. This might be the new direction for protein visualising in VR.

The limitations of using VR in Displaying Protein

Even though the VR implementation in displaying protein has come far and will still go further in the future, there are still some inevitable limitations. First, the limitations in the associated hardware, software may lead to an unsuccessful application of VR, which leads to the inaccuracy and imprecise in the results of using the application. With the increasing development of VR techniques and the popularity that VR games are gaining, software and hardware to be integrated with VR are becoming more compatible, but not without limitations. They are still costly and need to be increased in fidelity (Liu et al., 2018) Secondly, it is the unnatural feeling of using VR. Even though VR offers a realistic view, the users have to be wearing goggles which are not transparent and thus, blocking the vision of the real world. Furthermore, the head movements are unnatural because users will have to try to move their heads in order to see things. New HMDs are better because they are much lighter but mostly VR devices are still quite bulky and not that easy to use yet. Thirdly, most VR users claim to have motion sickness. This happens because of the difference the bodies and the eyes experience at the same times. The actual physical actions and the actions that are carried out in VR might be different and therefore, causing motion sickness to the users. Due to this problem, when using VR, users cannot use it for a long time.

2.2.3 Protein Visualisation in AR

Similar with Virtual Reality, Augmented Reality (AR) generate the realism by displaying the 3D models in a real-world context. However, unlike VR where the whole vision of the users is taken away and replaced by another completely different scene, AR's defined characteristic is that it added a layer onto the vision. While VR creates an immersive experience for users by shutting out the real physical world, AR maintains the realism of the world, allowing users to see whatever they are seeing plus more. With AR, the users have free movements while projecting images. Commonly speaking, there are the most two well-known types of AR technology implementation. The first one is implemented on AR smart-glasses such as the Microsoft HoloLens, Google Glass, Apple Glass. Contrast to VR goggles, AR glasses looks just like sunglasses or even normal glasses, thus, causing no bulky feelings to the users. The second type of implementations are on AR apps such as Pokemon Go. In this type of implementation, camera's phones are used to track the surroundings environment as well as adding a layer on top of the screen to show external information.

As AR is gaining popularity, more projects are being done but not much as it is an extension of VR, it is still very new. Some studies show that AR being used in science such as molecular displayed has yielded in good results for students, as it takes less imagination and makes things more easy to understand (Cai et al., 2014). However, there are not many AR apps available to support education, specifically in visualizing molecules.

As mentioned, there are not many projects done on visualisation molecules on AR. Unlike VR, where there are a various number of HDMs incorporated software and app for protein visualisation, on AR, apps are more commonly used. There are only a few apps that can be found. BiochemAR is one of those. BiochemAR was released in 2019 and are available on both App store (for iOS) and Google Play (for android). According to the developers, Sung and her team, the idea of the app is to create a simple, easy-to-use teaching tools for both teachers and students in the class room (Sung et al., 2020). The main function is to display protein in AR by scanning a QR code, thus the design is very basic. When a QR code is scanned, the app will use the phone/ tablet/ smart devices' built-in camera to bring the protein structure into life through VR as shown in Figure 2.3.

As the main purpose is to make things simple and easy to use for teachers and students, there is no other functions as well as interactions between users and the protein. Those are just visualised and users can move the phone around to look at the protein in different angles and size.

Having the same idea, another app called AR Assited Visualisation was developed in 2020 to visualize protein. These proteins are not written under QR code form but instead printed out on paper as in Figure 2.4.

Similar with BioChemAR, AR Assisted Visualisation only display protein structure in 3D, without any interacting elements.

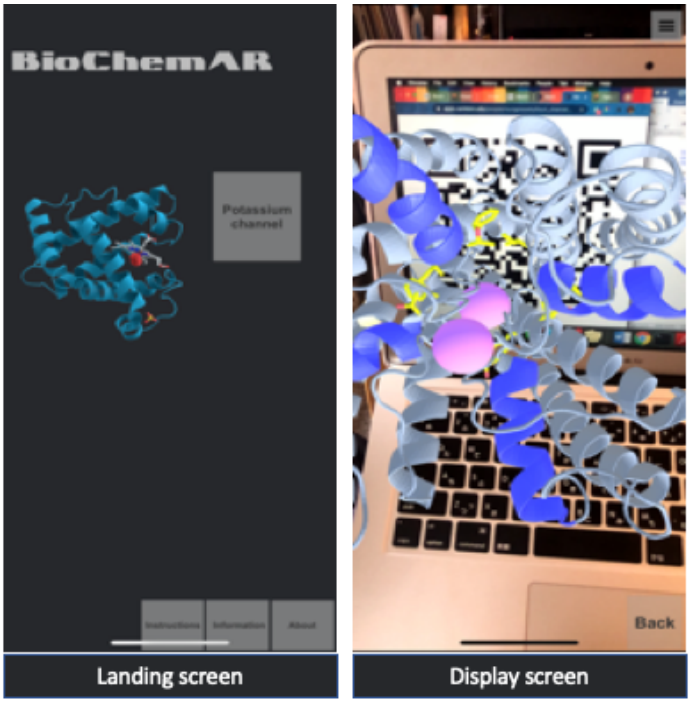

Figure 2.3: BiochemAR app screen shot

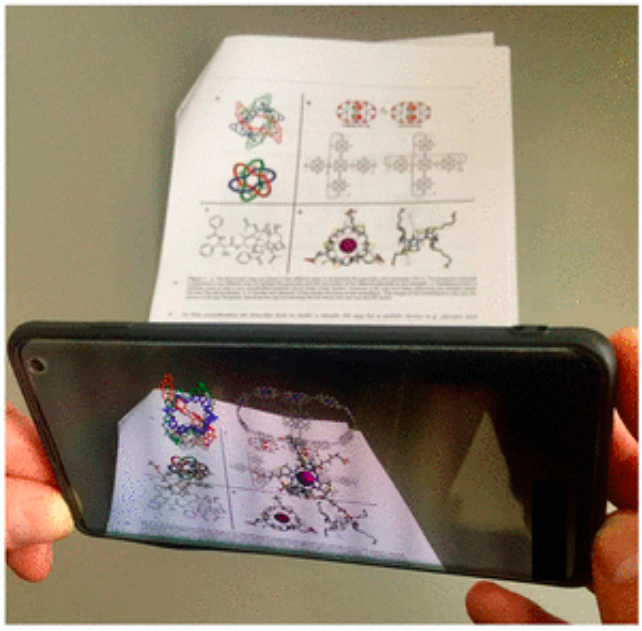

Figure 2.4: AR Assisted Visualisation App (Eriksen et al., 2020)

2.2.4 Finding summary

In a recent research on American Chemical Society and Division of Chemical Education, it seems that when undergraduate students get to create their own AR protein visualisation, they were enthusiastic in doing so and thus, their learning was enhanced when the AR module was inserted to their upper level biochemistry class (Argu, 2020). With the trend of online learning, the application of AR would promise a better curriculum for biochemistry.

Integrating protein visualisation on Mobile Apps is a good solution because of its availability. Most students have access to a mobile phone and it is handy to bring around as it is not bulky or need specific customisation.

The AR apps on protein visualisation are still new and young (released in 2019 and 2020). Thus, there is not much user interactions and functions to it. To use these apps mentioned in this thesis, a certain document with information of the protein, whether it be a figure of a protein or a QR code has to printed in order to get the AR visualisation. Moreover, the proteins can be viewed but cannot be interacted with in anyway. Furthermore, these apps are one-side oriented as users can only view protein but cannot create any.

This project's application: ProteinAR's purpose is to not only let users directly view the shape of protein in AR, interact with the protein by gesture touch on the screen but also allow users to design and create their own proteins. The majority of mobile apps to visualise protein are only in 3D, and most of the times on Android. Therefore, the open-source API for protein visualisation directly from the PDB files are limited. This project will have to start from little availability in pre-developed techniques.

With further work being put into, ProteinAR can be applied to be used in teaching to make lessons more interesting and understandable for students as well as motivate students to do higher level in Biochemistry.

Chapter 3

Methodology

ProteinAR is an app designed to run on an iOS system. It was written in Swift 5, on Xcode. The dataset in which protein files are downloaded from is directly connected to RCSB PDB. There were some other sources of protein data websites such as Protein Parameter or Protein Structure Function and Prediction I-TASSER Server were used to test out the application during the process of making. The app only runs fully on an iOS device, not a built-in simulator due to the requirement to use the camera to achieve the AR function.

3.1 Softwares used

3.1.1 Xcode

Xcode is an integrated development environment (IDE) for MacOS. It was first released in 2003, enables developers to create apps for Apple platforms. Xcode supports sources codes for various programming languages including C, C++, Objective-C, Swift, etc. Xcode has a built-in Interface Builder to construct graphical interfaces. During the making process, Xcode has a few version upgrades. The latest update was Xcode version 12. With every version updates, there are few changes in codes and functions as the main goal is to build more compact and user-friendly interfaces.

Advantages of using Xcode

ProteinAR is written on Swift, a native language for iOS apps, released by Apple and since Xcode is the native IDE of Apple, the compatibility is perfect, making the app and tests run faster and less errors. Xcode is a highly intuitive IDE where there is the main storyboard interface, visualising the designs elements of an app, with various built-in function to customise the design, from background colours to framing and a built-in library for easy adding and changing elements such as icons, pictures, text labels, etc (Introducing Xcode 12, n.d).

Disadvantages of using Xcode

ProteinAR used the built-in ARKit package. As this requires camera accessibility, tests cannot be run on the built-in iPhone simulators but instead, a real iPhone device. This creates a great disadvantages as iPhone iOS version keeps on updating, and thus, being incompatible with Xcode if Xcode is not the up-to-date version, which means MacOS should always stay as the latest version. Xcode's disk size is large, thus, downloading takes a great amount of disk space and time. Moreover, in some updates, the packages supports change, meaning there might be some errors that needed to be fixed with the newer version.

3.1.2 UCSF Chimera

UCSF Chimera (or Chimera) is developed by the University of California. This program allows interactive visualisation of protein data. Once a PDB file is downloaded, Chimera can open the files in a 3D form and allow users to export the files in various types such as *.dae, .x3d, .obj*.

3.2 Language used: Swift

Swift is a powerful programming language for Apple platform. Apple released Swift from 2014, taking ideas from various other languages (Rust, Haskell, Ruby, Python, C, etc.,) but it bares most similarities to Objective-C ("Swift - Apple Developer", n.d.).

Advantages of using Swift

Swift was always considered as one of the *Most Loved Programming Language* on Stack Overflow for many years as it is highly interactive, with concise and expressive syntax which runs fast. There are several improvements comparing to other languages: there is no need for semi-colons, UTF-8 based encoding is used, Strings are Unicode-correct, etc. It is also designed for safety as by default, Swift objects can never be *nil*. As a successor to C and Objective-C, Swift includes low-level primitives such as types, flow control and operators as well as object-oriented features such as classes, protocols, and generics ("Swift - Apple Developer", n.d.). Overall, Swift is a simple and straight-to-the-point coding language.

Disadvantages of using Swift

As mentioned above, there were a few version updates of Xcode during the programming process. Swift is a new language, thus, are being changed constantly to reach perfection. Therefore, the syntax and packages might change from times to times. It is relatively new, therefore, solutions to coding problems might be too new to have an answer, which was the most challenging in using Swift as the main coding language.

3.3 ARKit API

The technology to develop Augmented Reality was ready for mobile devices, however, it is too complex as algorithms for detecting objects in real world and displaying virtual object need

to be created, and these are very complex for developers. This is why Apple released ARKit in 2017 as a software framework, making developing an AR iOS app so much easier. It is an API that supplies numerous and powerful features to handle the process of building Augmented Reality apps and games for iOS devices.

Apple has been acquiring many AR companies, thus, the ARKit is built on all of these acquisitions. One of the major ones was the German company Metaio, which IKEA initially used to let customers display IKEA furnitures in their own home. Ferrari also used Metaio's technology to allow customer changing colours of cars in showroom, and looking at car's internal features. In 2017, Apple acquired SensoMotoric Instrument, a company specialized in eye tracking technology to use in AR. Other companies that specialized in other parts of AR technology are being acquired by Apples throughout the year. By doing this, the features of ARKit on iOS devices are frequently newly added and updated. ARKit is continuing to grow, making the creating of AR apps easier than ever (Wang, 2018).

3.3.1 Basic understanding of the ARKit

There are three layers that works simultaneously in ARKit ("Introduction to ARKit - Design+Code", n.d.) as shown in Figure 3.1.

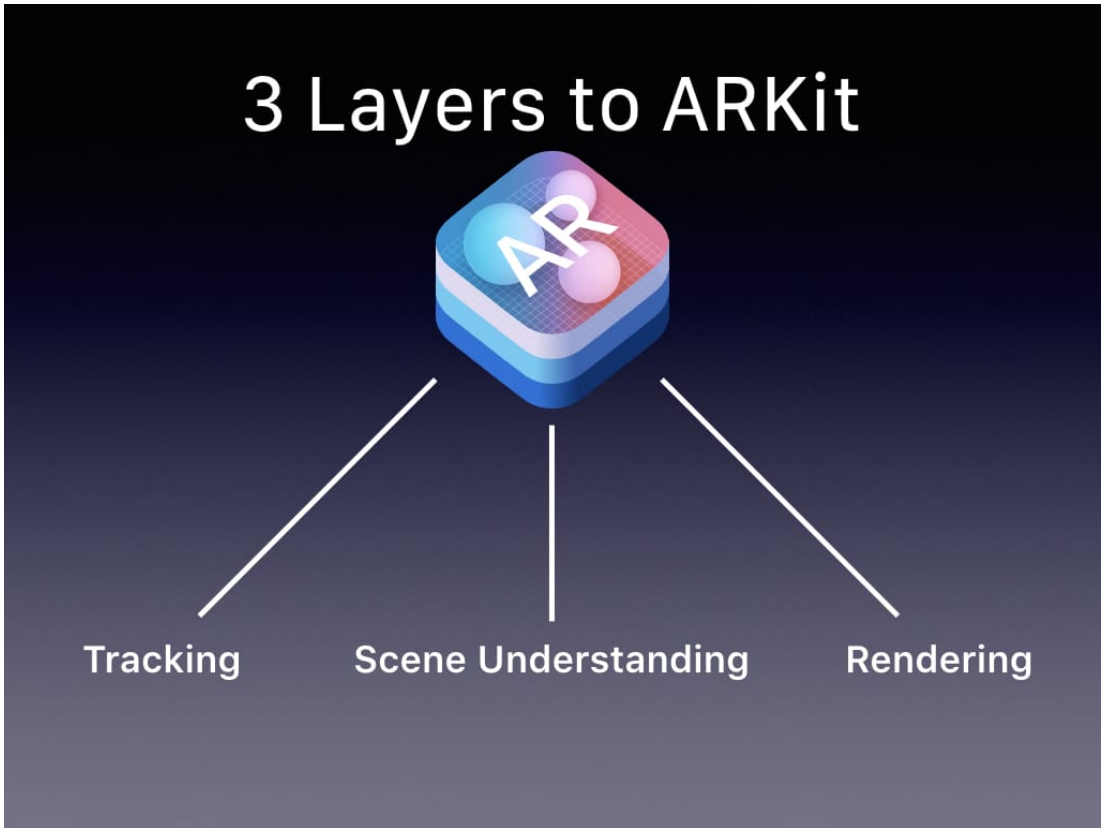

Figure 3.1: Three Layers to ARKit

Tracking is the key function of ARKit. Without ARKit, it would be very complex for developers to write algorithm to track a device's position, location and orientation in the real world. **Scene Understanding** is the layer that allows ARKit to analyse the environment presented by the camera's view to adjust and provide information in order to put place a virtual object on it. **Rendering** is the process where ARKIt handles the 3D models to put them in a scene such as SceneKit, Metal, RealityKit.

3.3.2 Language and System Requirement for ARKit

As mentioned in this thesis, since augmented reality requires access to cameras and high resolution display, ARKit apps can only be run on modern iOS devices:

- iPhone SE, iPhone 6s and later
- iPad 2017 and later
- All iPad Pro models

To develop an iOS app, Xcode is the best IDE to be used as it also has the built-in simulator program to mimic different iPhone and iPad models. However, with ARKit integrated, the app cannot be tested on the simulators but has to be on a real iOS devices listed above, connecting through its USB cable. Both Swift and Objective-C can be used to create an ARKit app. This project chose Swift as the language because it is much easier to learn and run faster. ARKit framework allows developers to be able to focus on the features of the app rather than on the AR required technologies such as detecting, displaying and tracking virtual object in the real world.

3.4 Database used: RCSB

ProteinAR downloads PDB files directly from RCSB.

PDB (Protein Data Bank) file format provides a standard representation for macromolecular structure data. These are obtained from X-ray diffraction and NMR studies (About RCSB PDB, n.d). RCSB was the first open access digital data resource for Protein Data Bank. It provides access to 3D structure data for all biological molecules. RCSB is a global archive where PDB data are available for free ("RCSB PDB: About RCSB PDB: Enabling Breakthroughs in Scientific and Biomedical Research and Education", n.d.). The data acquired on RCSB are data submitted by biologists and biochemists around the world. On the website, users can search for any protein name and the 3D structure will be displayed and can be interacted with. Information about the protein will also be displayed, and PDB files can be simply downloaded. During the process of making ProteinAR, some other sources for protein data were used including I-TASSER and ProtParam. ProtParam displays all the parameters for protein once the amino acid sequence is entered. ProtParam is a simple designed encoded website, allowing the GET method to get information from the server to the app easier, however, the PDB files contains 3D structure information of the protein is not available, therefore, it was used as a test to see if the POST and GET method work well in the app for similar website.

I-TASSER predicts protein structure and function after users enter the sequence of amino acids. Similar with RCSB, I-TASSER allow free downloading of the PDB files, where the structure of protein is already created in 3D and can be opened using UCSF Chimera. The cons of using I-TASSER is that the data cannot be downloaded in real-time because user need to enter their emails into the server and get the PDB files back a few hours later. As the goal of ProteinAR is to visualize protein structures and display them instantly, RCSB was chosen for the database as it fits the goal.

Chapter 4

Analysis: Thesis Problems and Solution functions

4.1 Main problems of the project

ProteinAR is an iOS application to visualise the three-dimensional (3D) structure of protein. The project was set with three main goals:

(1) Provide an **educational experience** for users: download and visualise in real time protein structure with data from RCSB PDB, using the user-typed input protein name. As mentioned in chapter 2, there are a few existing apps on visualising protein on AR. However, these apps need to scan a code/ an image to display the protein which lead to the limitation in displaying the protein as the protein needed to be prepared in some form already (QR codes, images). The apps that allow direct protein structure viewing in 3D by entering proteins names also are available but not in AR. Thus, the *first main problem* to solve is to make it possible for the app to connect to RCBS PDB server, download the protein model, and display it on AR after the user type in the name of the protein.

(2) Provide an **entertaining experience** for users: user can put together the polypeptide chains (coils, helix, sheet) to create new protein. As the existing apps on protein visualisation are more focused on just displaying the protein, this project is set to bring some entertaining element by adding the mini-game function in which users can create new proteins. The *second main problem* to solve is to enable users to create new proteins from the combination of coils, helices, and sheets. For this project, because of the biological complexity of the quaternary structure protein, the new protein created will be in tertiary form.

(3) Provide an **interactive experience** for users: user can interact with the protein models or the polypeptide chains that are displayed on the screen by touching them to scale the models, to move the models around and to rotate the models. The findings from chapter 2 shows that little effort has been put in interactive elements of the existing products. The main function of the products are mainly just to show the protein. Therefore, the *third main problem* to be solved is to enable interacting with the 3D models in AR.

4.2 Functional requirements to solve problems

4.2.1 Educational purpose: Visualising Protein from RCSB PDB server

There are a few problems that needed to be solved in order to visualise the proteins. Firstly, the app needs to be able to send request to the RCSB PDB server. Secondly, the app needs to be able to download the files from the server. Then, the app needs to be able to track the downloaded files' location. Finally, the app should be able to pull the files out and display them as an AR layer on the screen.

Send request and download the files

As mentioned, the app needs to be able to send information (user input) to the server and get the files back. Based on this thinking, the first try was to use the *POST* and *GET* method. This can be achieved by using *HTTP Request* in Swift. *HTTP POST Request* allows the app to post information onto the destination URL where the specified embedded method is *POST*. The way this can be achieved is firstly, to go on to the website that needed to be post on, inspect its element to find the action method as well as the parameters needed to be used in this method.

Similarly, *HTTP GET Request* allows the app to get information from the destination URL where the method is specified as *GET*. The approach is the same with *POST*, usually the parameters can be found by inspecting the source code of the website, most of the times under *form action*.

To test out the function, ProtParam was a good start as the website only consists of string type data. The URL for both *POST* and *GET* are the same and the methods are in the form action, which was no trouble to find. However, since there is no PDB files on ProtParam, RCSB PDB has to be the data source. On RCSB PDB, the methods of *POST* and *GET* do not exist in the form function. The PDB files are directly downloaded by a separate URL in which only the only changing part (parameter) is the name of the protein. Understanding this, ProteinAR uses *URLSesssion* and *downloadTask()*. *URLSession* makes network transfers easy and *downloadTask()* fetches the contents of a specified URL, saves it to a local file and calls a completion handle. The *URLSession* tracks the storing place of the download task while it happens. This will be explained more with codes in chapter 6

Display the file

When the files are downloaded, they are saved in the *.pdb* format. In Swift, when a file is downloaded, it is downloaded to a temporary location, and then can be moved to the *Document Directory*. ProteinAR specifies the format of the download by saving it as "name of the protein.pdb". The solution to display the *.pdb* file in visualisation is to convert it to *.dae* files and then load it on the *SCN*Scene as a scene. In order to load the file, one solution is to use move all downloaded items into the project folder using *moveItem*. However, this will make the app heavy because the data will be kept there and had to be loaded every time the app runs. Therefore, the solution that this project uses is to make use of *Core Data*. Each time a file is

save into the *Document Directory* it is also saved as an attribute of the *Protein Entity* that was pre-defined in *Core Data*. Each entity will have a value of *name* and *location*. For loading the file, there needs to be a converter which convert the downloaded *.pdb* file to *.dae* file. This converter will automatically convert any downloaded *.pdb* file in the *Document Directory* into *.dae* so that it can be loaded as a *SCNScene* in the app. Unfortunately, due to the limitation of time, the converter could not be made and thus, this remain the one problem that had not been solved in this project. For a smooth demonstration of how the ideal product should turn out, a sample folder of a few pre-downloaded models is imported.

4.2.2 Entertaining purpose: Create new proteins from combination

Add polypeptide chains to screen

The app needs to be able to display individual polypeptide chain when the user clicks the buttons. There are four types of polypeptide chains to be added: Flex Coil, Rig Coil, Helix, and Sheet. Each polypeptide chain is input into the project as a *.dae* model. In order for these models to be loaded on ARKit, it must be converted into *.scn* files. Each model consists of different nodes: the model, lighting, camera, etc. By using the pre-defined function of *SCNScene*, the 3D models can be loaded into the AR view. By passing on the name of each models as a parameter, only one function of adding is needed to add four polypeptide chains using four different buttons.

All of the models are loaded on screen at the same location as if the location is not specified, the models might go off-screen. However, this caused a problem because if the same model is added twice, they will lay on top of each other, causing misunderstanding for the users as they can only see one model on screen. To solve this problem, the app randomises the orientations of the models every time a new model is added to screen by using the pre-defined function of *eulerAngles* to specify the *SCNVector3* with random x, y, and z.

Combining polypeptide chains

After adding individual polypeptide chains on screen, ProteinAR must be able to combine these chains into proteins. The combinations might be successful and might not be. For this to happen, successful combinations of these chains are pre-loaded into the apps in a "Combinations" folder. In the code, an empty string array for protein name is created. Every time user adds a polypeptide chain to the screen, the name of the protein is added using *append* to the array. After user clicking the "Try" button to combine the polypeptide chains, the names in the array are joined using the *array.joined()* function. The models' name in the "Combinations" folder have a naming convention so that when the array are joined, the name it generated matches with the name of the models in the "Combinations" folder. See more chapter 6 for further understanding.

4.2.3 Interactive purpose: Interacting with the models

After the polypeptide chains or the protein models are loaded onto the screen, users should be able to interact with the models by touching them on screen. To make it happens, the

UIGestureRecognizer was used. There are three types of *Gesture Recognizer* used in ProteinAR:

UI Gesture	Gesture Description	Function in the app
Pinch Gesture	"A two-fingers gesture that moves the two fingertips closer or farther apart" (Wang, 2018).	Allows users to scale (zoom in, zoom out) on the models.
Rotation Gesture	"A two-fingers gesture that moves the two fingertips in a circular motion" (Wang, 2018).	Allows users to rotate the models in any angle.
Pan Gesture	"Press a finger on the screen and then slide it across the screen" (Wang, 2018).	Allows users to move the models on the screen.

Table 4.1: Interacting Gestures in ProteinAR

4.3 Non-functional requirements

4.3.1 Core Data

Core Data is a popular framework provided by Apple to manage the model layer object in an application. Core data can automate solutions to common tasks associated with object life cycle and object graph management, including persistence (Core Data Programming Guide, n.d.). In this app, in order to manage the downloaded proteins' pdb files, Core Data is used with Protein defined as an Entity with the properties of name and location, stored as *String*. *NSManagedObject* instance was created by defining the *NSEntityDescription* and an *NSManagedObjectContext*. *NSFetchRequestResults* is used to fetch the stored data and display them on the screen.

4.3.2 Constraint

Although it is not mandatory; the app should be able to run on different iOS devices without problem. As the screen size of different iOS devices are different, if the app was designed on the view of iPhone 11 but run on iPhone 6, the buttons might be off screen or other elements might move around, making it impossible to navigate through the app. This is the reason why constraints are important in developing an iOS app. ProteinAR do not have too many elements on the screen at the same time, however, the *Auto Layout* was chosen as the solution for the constraints. Using *Auto Layout*, every new view that is a layer on top of a view are made into a *childView* attaching to the *parentView* which makes it easy for the anchor to be pinched to the *parentView*. *NSLayoutConstraint* was used to keep the elements in place.

4.3.3 Protein Combination Models Database and Polypeptide chains Database

As mention above, the combinations of polypeptide chains are kept in a “Combination” folder and has a naming convention that makes it easy for addressing the model. It is the combination of the names of the polypeptides, which makes it possible for the array to be combined into the new names.

When users tap on the individual polypeptide buttons, the models are displayed distinctively without having to tap on the “Try” button. When the “Try” button is tapped, the models on screen combined. In order to do this, the separate models are kept in a different folder, under a different function to avoid the collision.

Chapter 5

Project Design

5.1 Application Skeleton Design

The structure of ProteinAR is very simple. It consists of four main screens including the landing screen as shown in Figure 5.1.

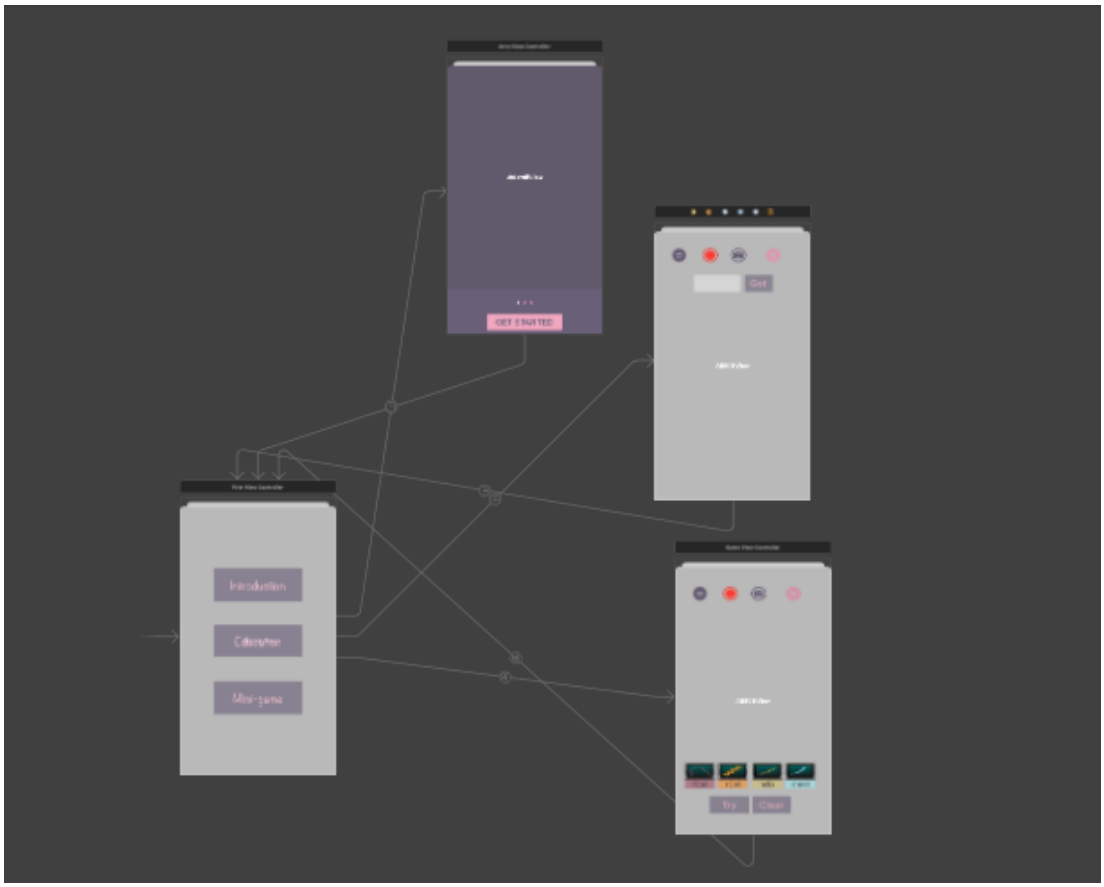

Figure 5.1: The skeleton of the app

On the landing view (first view) (Figure 5.2, there are three buttons, leading to the three other views of the apps.

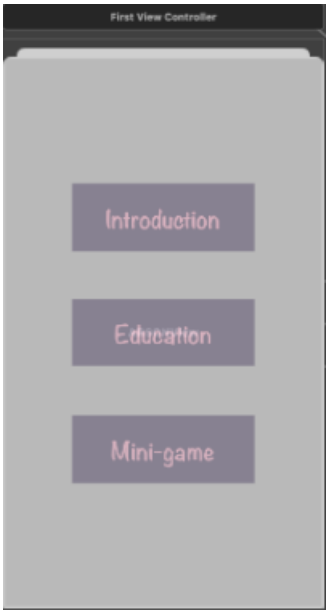

Figure 5.2: The First Screen View – Landing view after launch screen

(1) By tapping on *Introduction*, the segue will bring up the introduction view. Instead of using multiple screens connecting from the introduction, there are three sub-screens added by using *page control* on the *Introduction Screen View* to give information about the app as shown in Figure 5.3. Using *Page Control* keeps the coherency for the same contents, at the same

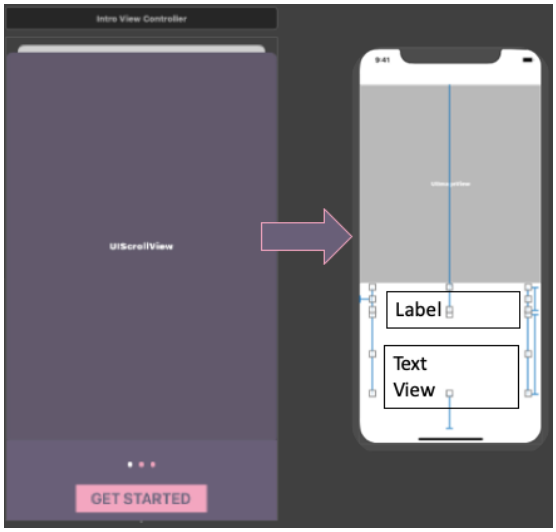

Figure 5.3: Introduction View Controller and Page Controller

time keep less words per screen, makes it more appealing to users. Users can click on the *GET STARTED* button to go back to the first view to start explore the options or simply just drag

the screen down and away.

(2) Tapping on the *Education* will bring users to the Education View Controller as shown in Figure 5.4. The main function on this screen is for user to input the protein's name and get

Figure 5.4: Education View Controller

the pdb file back, thus the design is simple with a *textfield* and a *GET button*. To make the app more appealing, there are four other buttons with four other minor actions on top of the screen. These actions are *Menu, Screen Record, Screen Capture* and *Exit* as shown in Figure 5.5. These functions will be explained in the next section: design solutions.

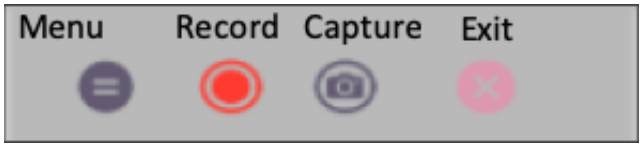

Figure 5.5: Four buttons on top of Education View Controller and Game View Controller

(3) Tapping on the *Mini-game* will bring up the *Game View Controller* (Figure 5.6). In this view, users can create new protein by combining the coils, helix and sheet in different orders simply by adding each polypeptide onto the screen by tapping on them, and press *Try*. Similar to Education View Controller, the four buttons on top of the screen are kept.

5.2 Solution Design

Figure 5.7 shows the whole design of the solution for the app.

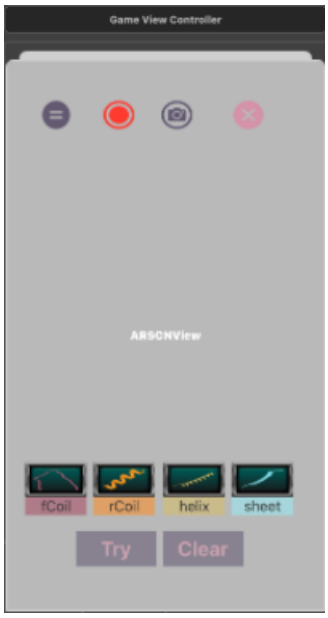

Figure 5.6: Game View Controller

5.2.1 Solution for top four buttons

The top four buttons are the same on both Education View Controller and Game View Controller. This creates the coherency through-out the app. However, the downside is that all functions and buttons have to be duplicated on the two views, causing heavier memory load for the app.

- *Menu* is the function that gives users extra options. The extra options on Education View Controller and Game View Controller are slightly different. On Education View Controller, when the user press *Menu*, an *Alert Service* is used, where the options rise up from the bottom of the screen, giving users 4 options: *More About Protein*, *Help*, and *PDB 101*. While *More About Protein* get users directly to the homepage of RCSB PDB and *PDB 101* links to the PDB 101 page on the RCSB website, the *Help* options bring a small pop-up screen layer on top of the AR scene. This pop-up screen contains some guide lines on how to use and navigate around the *Education View* and *Game View* accordingly. Since this is more of a demo-app, the options only directly open link to the RCSB website, however, for future work development, more in-depth options can be integrated to create a more scientific experience for users.

- *Record* is the function to record the AR screen and then save the recorded video to the phone's *Camera Roll* if the users choose to do so. In interacting with a protein or create a new one, users might want to record the process as there might be interesting and new findings for future study. When users long press on the *Record* button, the recording process will start. By doing so, there will be a pop-up on screen asking for permission to save the recorded file to the *Camera Roll*. The recording can be stopped simply by tapping on the record button. The app will then bring up a *Preview screen*, allowing

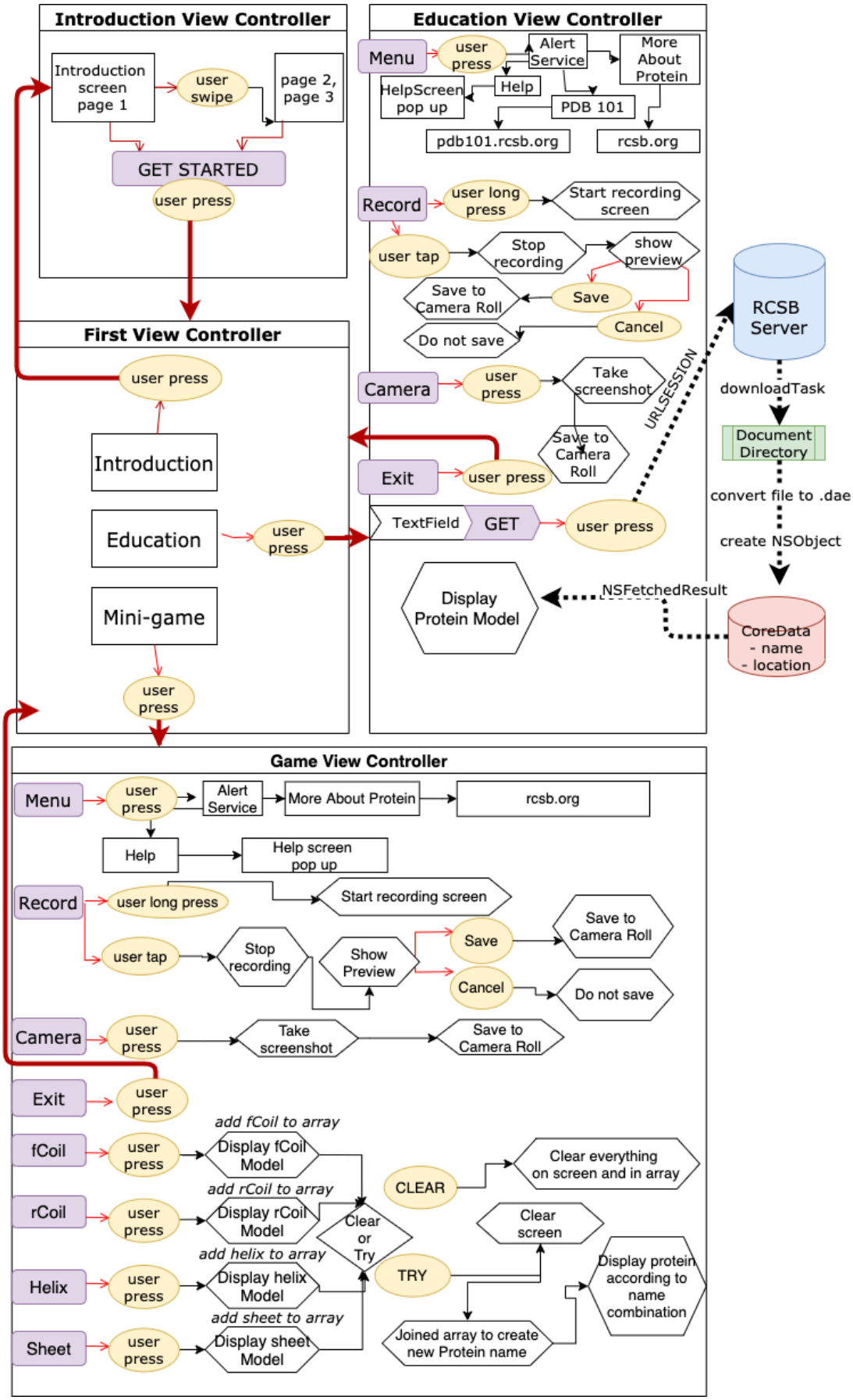

Figure 5.7: Application Solution Design

users to watch the recorded video before deciding to save or to not save the video. The two actions of *long-pressing* and *tapping* are enabled using the *UIGestureRecognizer* of the ARKit. The iPhone already has a screen recording functions built in. However, the *Screen record* function is better because it only records things happening on the AR Scene, excluding all buttons and un-wanted information.

- *Camera* is the function to capture the AR screen into a shot and save the photo to the phone's *Camera Roll*. When user taps the *Camera* button, the screen will be captured and save immediately. The UIButton flashes colors to indicate that the shot has been taken. Similar to the *Record* function, even though iPhone already has the screen-capture function, that will include all the buttons on the screen, which is not desirable. Therefore, with this *Camera* function, users can save a photo of just the protein they want.

5.2.2 Solution for getting pdb files from RCSB Server and storing it in CoreData

This is one of the critical functions of the project. It requires the app to be able to download the PDB files, save it and then display it on the screen. In order to achieve the download function, there were many trial-and-error-methods as mentioned in the previous section of using the *HTTP Request POST* and *HTTP Request GET* method. In the process of making the task possible, Alamofire was also considered an option. Alamofire is a Swift-based HTTP networking libray for iOS which simplifies a number of common networking tasks. However, after a few tries, the conclusion was that it was not necessary since the main task of the function is just to download a PDB file. This can be achieved using *URLSession* with *downloadTask()*. The downloaded destination is pre-defined to the internal *Document Directory*. To save the downloaded file's information to CoreData, firstly, a CoreData model was created with an *Entity* Protein. This *entity* has two attributes of *name* and *location*. Both of these attributes' type are *String*. If the download process is successful (the file exists, the connection was stable, etc.), at the same time of downloading, a new *NSObject* is created with the two attributes. These will be saved as an *Entity* in the CoreData database. *NSFetchRequestResult* is used to fetch the data in CoreData database back to the app. This process is visualise in Figure 5.8.

5.2.3 Solution for visualisation of protein models from pDB files

As mentioned, in order to visualise protein models from downloaded pDB files, a converter to convert file type *.pdb* to file type *.dae* must be made. The ideal design is as shown in Figure 5.8. UCSF Chimera was used in the process of converting, however, it is only compatible with MacOS, not iOS. This is the missing solution from the project.

5.2.4 Solution for combining polypeptide chains into a protein

As mentioned in the previous section, each polypeptide chain is designed to be referred to as a value in an array. Every time user press on a polypeptide chain's button, that model of protein is displayed, at the same time, that model's name is added as a value in the array. After these

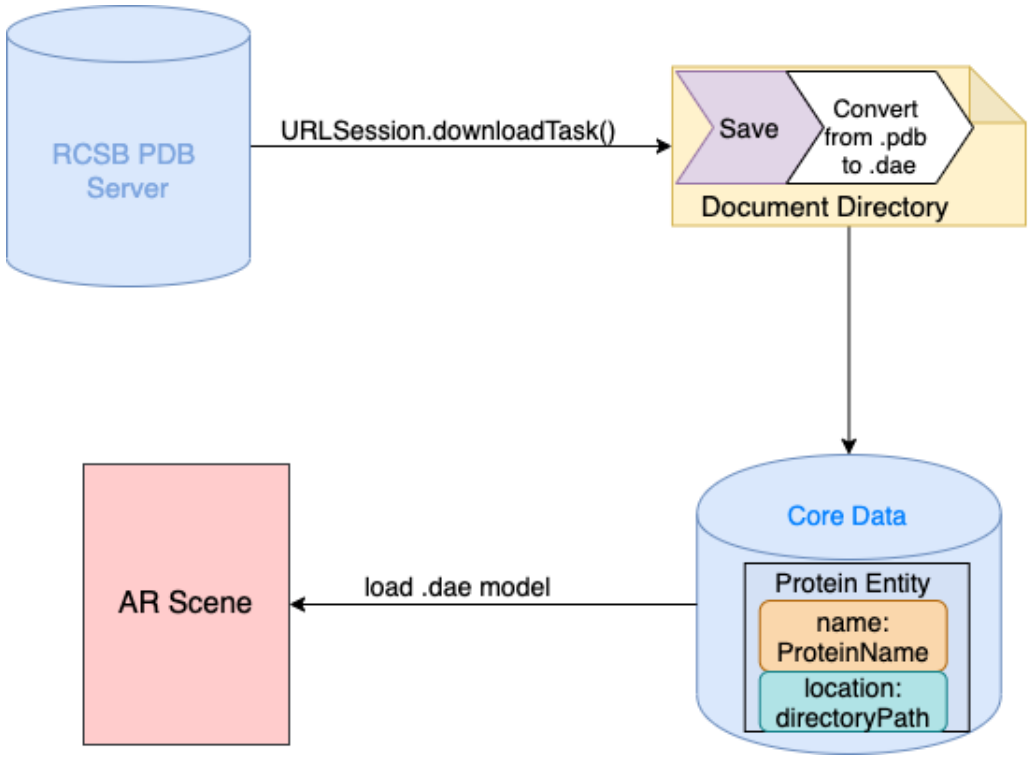

Figure 5.8: Process of Downloading, Saving and Displaying downloaded protein Model

actions, if the *Clear* button is pressed, not only the models on the screen is deleted but also the values in the array are emptied. On the other hand, if *Try* is pressed, all the values in the array will combined into a new name. First, the screen will be cleared and then the new name protein model will be displayed. Together with this, a text of "Congratulations, you have created a new protein made of . . . " will also be displayed if the combination is valid. If the combination is not valid, no models will be displayed. The errors will be caught and, on the console, "This model does not exist" will be printed. As for user's side, a 3D text of "Sorry, this combination cannot be made" will appear on screen. The simplification of the design can be found in Figure 5.9.

5.3 User interface (UI) Design

5.3.1 Introduction to user interface

In order to ask the computer/smart devices to do anything, users need to communicate with them. The way users (human) can communicate with the product (software, app, website) is through interacting with the user interface (UI) of that product. The purpose of a UI is to enable users to control a computer or a device they are interacting with, by giving orders and receiving feedbacks in a chain to complete a task. The user interface of any software or website does not only create the first impression to the products which makes user decide instantly if they want to use the product, but it also plays a big role in keeping the users. With a

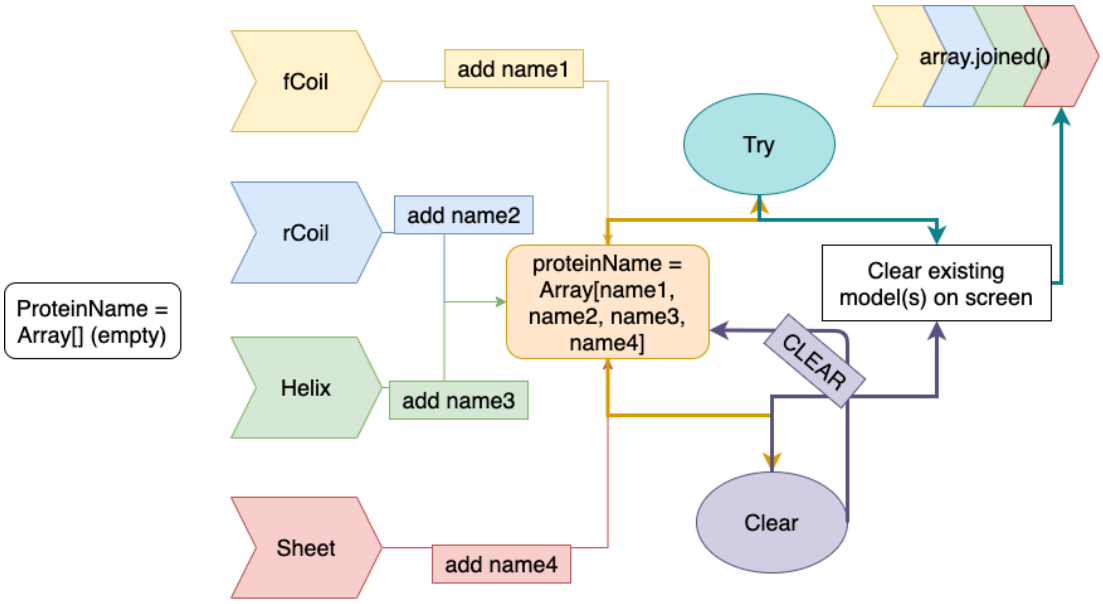

Figure 5.9: Create a new protein name from existing ones

complicated or not efficient UI, users would not want to keep using the product because it takes memory efforts. Therefore, an UI should be *intuitive* – be kept simple where no training should be needed to operate, *efficient* – functions are precise and on point, and *user-friendly* (User Interface, n.d) Currently, there are three formats of user interfaces (What is User Interface Design, n.d.):

- **Graphical User Interfaces (GUIs)** – interactions happens through visual representations on digital control panels such as a computer desktop, a website interface.
- **Voice-controlled interfaces(VUIs)** – interactions happens through voice representation such as Siri, Google Home or Alexa.
- **Gesture-based interfaces** – interactions happen through physical motions in 3D spaces in VR games.

The UI of ProteinAR is categorized as a GUI since users interact with the device through the visual representations of functions on a phone screen.

5.3.2 ProteinAR's user interface design

Logo design

The logo for the app was designed simple with just a letter P, short for Protein. This was created in GIMP and then exported to various sizes to fit to maintain the resolution in different views (refer to Figure 5.10. Other designs with symbols or words were considered but sticking to the "Simplicity is the best" approach, the logo ended up with only one simple letter and two

colours, makes it easy to remember for users. This is not a new approach. Simple logo design with only one letter can be found with popular apps such as Facebook app or Google app.

Figure 5.10: App's logo in different sizes.

Colour scheme

The logo, to the flash screen, the buttons and popup view elements in the app follow the same colour scheme. In ProteinAR, analogous colour scheme was chosen as shown in Figure 5.11.

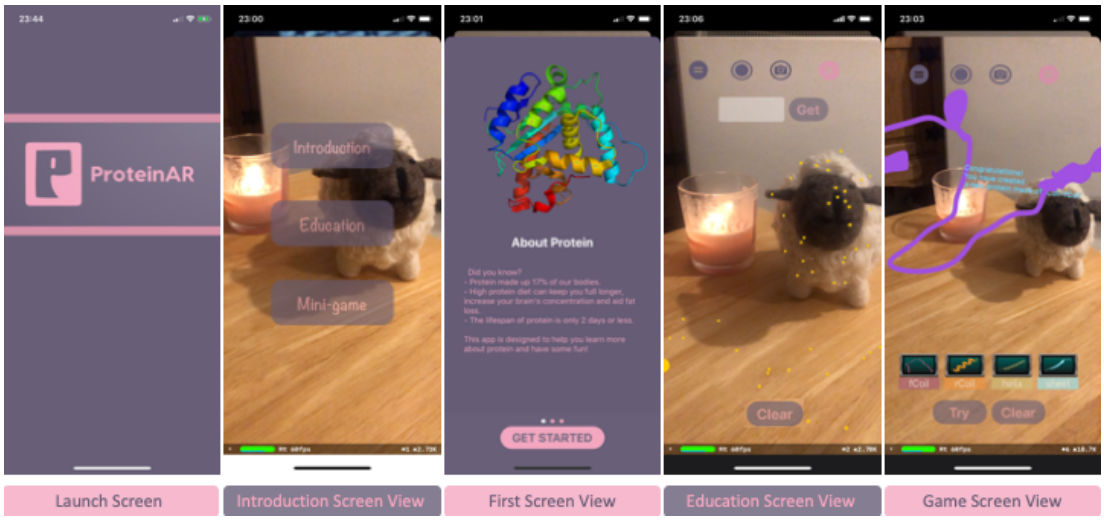

Figure 5.11: UI design of ProteinAR

This is one of the traditional colour palettes which is the combination of related colours that

are placed next to each other on the colour wheel. Analogous is known to be one of the most used colour pallets because they are harmonious and pleasing to the eyes. ProteinAR uses two colours from the Pinks and Mauves colour sections as shown in Figure 5.12.

Figure 5.12: Analogous Colour Scheme

Button design

As for the top four buttons of the Education and Mini-game screen, the main colour scheme is kept. As the first three buttons generate actions, they are in the same mauves colour and the exit one is in pink, which create the slight distinction of the functions. The two main function button of *Try* and *Clear* also follow the main colour scheme.

On the top four buttons, button icons are used instead of button labels. These icons are familiar to mobile app users, thus, makes the design more concise and easier to navigate.

(1) Menu button: There are many styles of menu buttons as shown in Figure 5.13. Each menu buttons generate a type of menu display for example, the *hamburger icon* opens a navigation drawers to more actions, the *kebab icon* are commonly seen on Android operating system, normally open a smaller inline menu. In this project, the chosen icon for *Menu* button is the *Veggie burger* style as it is common for this style to be placed on the top left of the screen, and generate more actions but less than a *hamburger icon*.

(2) Record and Camera button: Record button accepts two types of actions: long press

Figure 5.13: Different styles of menu buttons – source: ux.stackexchange

and tap. Long press generates the action of recording the screen and tap ends it. Long press action also changes the colour of the button to red, which is commonly associate with recording symbol colour. Tap brings it back to its original colour, symbolizes the end of the recording action. As for the camera button, the colour only flashes, implies the act of picture taking has been done.

As mentioned, the buttons in ProteinAR mainly follows the colour scheme of Pinks-Mauves. However, the four buttons to add polypeptide chains are the exceptions. These four buttons uses images as the buttons and to make it easy for users, they are labelled with their names. The designs of the buttons are inherrited from Tianshu Xu's 2019 Master project (T. Xu, 2019) and the label colours were chosen to be matched with the polypeptides' colours. Since this is a mini-game, the colourful elements will make it look more appealing to users.

5.4 Core Data Design

In ProteinAR, Core Data is designed with only two *Entity* of *name* and *location*. Core Data only stores the information of the location path, which is in the *Document Directory*, not the file itself. When the data is called, information stores in the attribute *location* of Core Data simply acts as a *String* value to concatenate with the file's name and extension to fetch the file that

Figure 5.14: Polypeptide chains button

are stored in the *Document Directory*. With this current functions of ProteinAR, implementing Core Data is not necessary as document path can be directly called into the displaying function. The reason Core Data was used is for future work, when more information of a Protein (protein parameters) needed to be stored.

Chapter 6

Project Implementation

6.1 Download and Visualise Protein Models

Due to its complexity, the process to download and visualise protein models will be explained in five steps.

6.1.1 Step 1. Set up Core Data (Figure 6.1 and Figure 6.2)

First, Core Data is set up by adding a new *Data model* from the *Core Data* section. It is important to add-on the *App delegate* if *Core Data* was added in later in the project because Xcode will not automatically generate those delegate and the database will not be set up. In this app, the database has only one entity with the name of *Protein* which has two *attributes* of *name* and *location*, defined in type *String* as in Figure 6.1. After that, *NSManagedObjectSubclass* were created where Protein is defined as a public class in *NSManagedObject* and so as function *fetchRequest*. In these subclass, the attributes of *name* and *location* are also declared as public variables (refer to Figure 6.2)

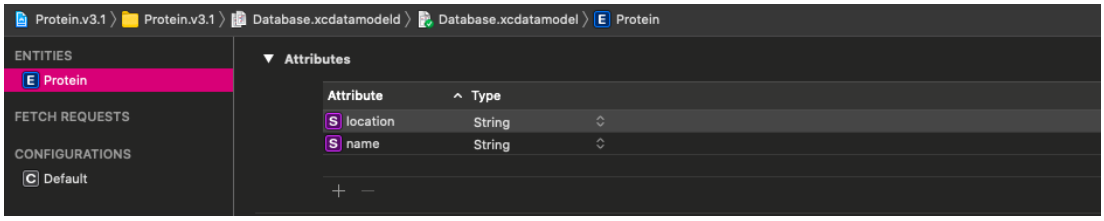

Figure 6.1: Core Data Entity and attributes

6.1.2 Step 2. Download from RCSB PDB using downloadTask (Figure 6.3 and Figure 6.4)

Download is the critical function in this process and since it is a long one, the code snippet will show the code in two parts with modification for easy explanation. The full functioning code

```swift
import Foundation
import CoreData

@objc(Protein)
public class Protein: NSManagedObject {

}
```

```swift
extension Protein {

    @nonobjc public class func fetchRequest() -> NSFetchRequest<Protein> {
        return NSFetchRequest<Protein>(entityName: "Protein")
    }

    @NSManaged public var location: String?
    @NSManaged public var name: String?

}
```

Figure 6.2: NSManagedObject subclass

with alternative options can be found in Appendix A or in the attached source code folder.

In the code shown in Figure 6.3, the URL to the source file is created. After observing how files are downloaded from RCSB, an URL pattern was found. Instead of using the *GET* method in *action form*, RCSB allows downloading the PDB files directly from an URL. The structure of these URL are the same for all of the different PDB files, starting with the same domain. The file name is the only part that needs to be changed, and it is the protein's name. With this logic, the URL to the source file was constructed using the parameter as the user-input-text to change the file name accordingly.

```swift
    func download() {
        //Create URL to the source file to download
        let parameter = textField.text
        let domain = "https://files.rcsb.org/download/"
        let fileExt = ".pdb"
        let fileURL = URL(string: "\(domain)" + parameter! + "\(fileExt)")!
        print(fileURL)

```

Figure 6.3: Download function part 1

The code snippet in Figure 6.4 shows how the code uses *URLSession* and *downloadTask()* to generate the download. *URLSession* provides API for downloading and uploading data to the specified URLs. This API helps performing background downloads. In this code, *default* type of *URLSession* is used instead of *shared* because it allows more freedom of configuration. *URLSessionConfiguration* defines the behaviour policies when the app downloads data from the server. There are a few types of *URL Session Tasks*. In this app, *download task* is used as it retrieves data in the form of a file and supports background downloads. The status code of *HTTPURLResponse* is important to be known because if the file cannot be downloaded, the problems could be addressed as there could be different reasons that triggers the unsuccessful downloads: the file does not exist (status code 404), the connection to the server was interrupted (status code 500) or something else is wrong with the code. When the code performs its download task, the file is store in a temporary location, as called in the code *temporaryURL*.

The file will be moved to an absolute path in the *Document Directory* by using the *File Manager* to remove and move item. In order to keep track of the downloaded files and create an absolute path, the name of the files are pre-defined by *destinationURL* (refer to Figure 6.4).

```
409         //Use URLSession and downloadTask
410         let sessionConfig = URLSessionConfiguration.default
411         let session = URLSession(configuration: sessionConfig)
412         let request = URLRequest(url: fileURL)
413         let task = session.downloadTask(with: request) { temporaryURL, response, error in
414                 //Get the httpresonse code to make sure file exists
415                 if let statusCode = (response as? HTTPURLResponse)?.statusCode{
416                     print("Successfully downloaded. Status code:\(statusCode)")
417                 }else {
418                     return
419                 }
420                 guard let temporaryURL = temporaryURL, error == nil else {
421                     print(error ?? "Unknown error")
422                     return
423                 }
424                 do {
425                     //download file and save as pre-defined format
426                     let documentsUrl =  try FileManager.default.url(for: .documentDirectory, in: .userDomainMask, appropriateFor:
                            nil, create: false)
427                     let destinationURL = documentsUrl.appendingPathComponent( parameter! + ".pdb")
428                     //manage the downloaded files
429                     let manager = FileManager.default
430                     try? manager.removeItem(at: destinationURL)// remove the old one, if there is any
431                     try manager.moveItem(at: temporaryURL, to: destinationURL)// move the new one to destinationURL
432                     //Save Files information to Core Data
433                     self.saveContext(name:parameter!, location: String(describing: documentsUrl))
434                 }
435                 catch let moveError {
436                     print("\(moveError)")
437                 }
438         }
439         task.resume()
```

Figure 6.4: Download function part 2

The alternative way is to move the downloaded file into the main app bundle as shown in Figure 6.5. The directory of the main app's bundle is created and the file can be moved by the same *moveItem()* method. In the source code, this alternative way is disable. The reason for this was previously explained: if all the downloaded files are saved into the main app's folder, the app will have to load them all every time it is loaded, making the app heavy and slow.

```
435                     //Create the path to the main app's Bundle
436                     let newFolderURL = Bundle.main.bundleURL
437                     let newFileURL = newFolderURL.appendingPathComponent("/Sample.scnassets" + parameter! + ".pdb")
438
439
440                     //manage the downloaded files
441                     let manager = FileManager.default
442                     try? manager.removeItem(at: destinationURL)// remove the old one, if there is any
443                     try manager.moveItem(at: temporaryURL, to: destinationURL)// move the new one to destinationURL
444
445                     try manager.moveItem(at: destinationURL, to: newFileURL)
```

Figure 6.5: Alternative: Move downloaded files to main app's folder

6.1.3 Step 3. Assign downloaded files to Core Data (Figure 6.6)

Firstly, the *context* is declared by *persistentContainer* and the *proteinManagedObject* is declared by starting with *nil*. Then, the function to save *proteinManagedObject context* is called inside of the *do* action in the download function (Line 433 -Figure 6.4). The function to save

context is displayed in Figure 6.6. When the download is successful, the attributes of *proteinManagedObject.name* and *proteinManagedObject.location* are saved into the Protein *Entity* as *String*. Since *proteinManageObject* is a global variable, it can be accessed anywhere in the code.

```
22      let context = (UIApplication.shared.delegate as! AppDelegate).persistentContainer.viewContext
23
24      var proteinManagedObject : Protein! = nil
25
26      var entity: NSEntityDescription!=nil
27
```

```
464     func saveContext(name: String, location: String) {
465         //proteinManagedObject = frc.object(at: IndexPath(row: 0, section: 0)) as? Protein
466         proteinManagedObject = Protein(context: context)
467         proteinManagedObject.name = name
468         proteinManagedObject.location = location
469         do {
470             try context.save()
471             print("Data saved to CoreData")
472         } catch {
473             print("Cannot create a new object")
474         }
475     }
```

Figure 6.6: Save and Assign downloaded file to attributes in Core Data

6.1.4 Step 4. Convert PDB file to Collada file

The process would be completed with a script converting *.pdb* file to *.dae* file type because ARScene only allow loading Collada models as its AR Scene. A few solutions were used to solve this problem. One of those is to borrow the converting from PDB to Collada script from UCSF Chimera. Since Chimera was written in Python, its scripts could be run in Swift because Python has a C interface API. However, the problem with this was Chimera is not compatible with iOS in the first place, thus, this solution could not be used.

OpenBabel was another a solution that was carried out. Unfortunately, OpenBabel is also just compatible to Android and MacOS, not iOS and since, was not able to be implemented.

RCSB PDB published an article on the releasing of their mobile version in 2015 which can help visualise the PDB file on both iOS and Android, however, as of 2020, it was no longer available on the App Store.

Thus, this remains an unsolved problem of this project.

6.1.5 Step 5. Fetch and Visualise PDB files (Figure 6.7)

The data which are saved into the Core Data would be fetched using *NSFetchRequestResult* and can be display easily using the attributes assigned into Core Data. If the PDB to Collada converter script can be made, the function could just be as easy as shown in Figure 6.7

Due to the unsolved problem of PDB to Collada converter, the app demo video shows some pre-downloaded protein models to give a complete image of how the app would be done if given more time in the future work.

```
180     //idea of the function that should be able to display the protein if there is a converter
181    func displayProteinreal(name: String) {
182     let proteinScene = SCNScene(named: proteinManagedObject.location! + "/" + name + ".dae")!
183         secondSceneView.scene = proteinScene
184     }
```

Figure 6.7: Function to display protein after being converted into Collada models

6.2 Create new Protein Models

6.2.1 Step 1. Import and name models (Figure 6.8)

Since ProteinAR uses ARKit and SceneKit to load the models on, for importing the models, first, a new file of *Scene Catalogue* must be made. In this project, the folder is named "Combinations". The provided combination model type was in *.dae*, however, to enable smooth loading for SceneKit, the files are converted into *.scn* type. These are named after the polypeptide chains names and their order in creating the combinations. See Figure 6.8 for more details of some examples. This naming convention makes it easy to pass as arguments and load models

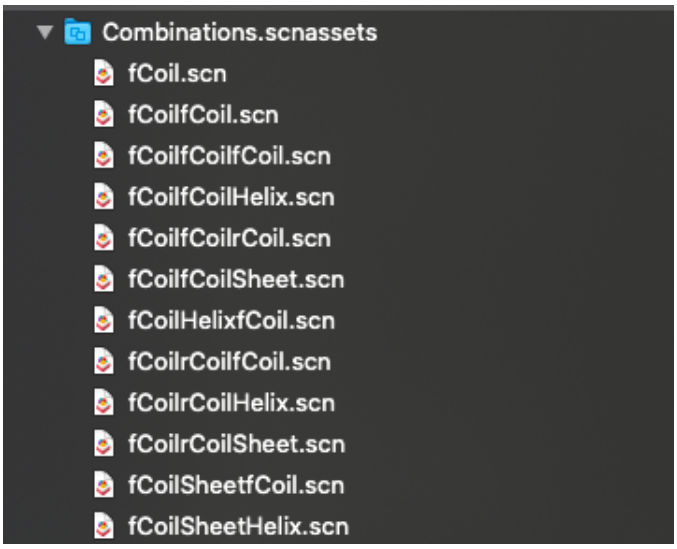

Figure 6.8: Combinations of polypeptide chains stored in a folder

in the following functions. To make the models more appealing, Phong shading is used and the colour of each and every model are randomly picked.

6.2.2 Step 2. Add polypeptide function (Figure 6.9, 6.10)

Figure 6.9 shows the function to add the polypeptides on the screen. In this function, the argument is pre-defined as *String* type and has the name of *name*. In *ARKit*, *SCNScene* is used to load the 3D models. Since all the models have the same format (inside of "Combinations.scnassets" folder and has the ".scn" extension), the models will easily be called by passing the names of the protein as arguments each time a Polypeptide button is pressed, as shown in Figure 6.10. The function also add a camera node to the screen at the position of (0, 0, 0) and

the position of the models are also fixed by *SCNVector3* to ensure that the models will always appear in front of the camera. One of the problems that users might encounter with using AR technology is that the space is infinity and so the models might be loaded in places that cannot be seen. Thus, it is more user-friendly to make sure the position of the models loaded are visible. The function also uses *eulerAngles* with a random *SCNVector3* to make sure that every time new models are loaded, they are at the same position, but with different rotation so they do not lay on top of each other perfectly, ensuring users that they have already added the same models more than one time.

```
// 4. Polypeptide Button Functions: To add protein onto the screens when button is pressed
    func addProtein(name: String) {
       lightOn()
        let proteinScene = SCNScene(named: "models.scnassets/" + name + ".scn")!
       //These models have only 1 rootnode as the model, add cameranode
        let cameraNode = SCNNode()
        cameraNode.camera = SCNCamera()
        cameraNode.position = SCNVector3(x: 0, y: 0, z: 0)
        scene.rootNode.addChildNode(cameraNode)

        let nodeName = proteinScene.rootNode.childNodes[0].name

        guard let proteinNode = proteinScene.rootNode.childNode(withName: nodeName!, recursively: true) else {return}

        proteinNode.scale = SCNVector3(x: 0.008, y: 0.008, z: 0.008)
        proteinNode.position = SCNVector3(x: -0.005, y: 0, z: -0.010)

        let randomx = Float.random(in: (-Float.pi)...(Float.pi))
        let randomy = Float.random(in: (-Float.pi)...(Float.pi))
        let randomz = Float.random(in: (-Float.pi)...(Float.pi))


        proteinNode.eulerAngles = SCNVector3(x: randomx, y:randomy, z:randomz)
        scene.rootNode.addChildNode(proteinNode)

        let centerConstraint = SCNLookAtConstraint(target: proteinNode)
        cameraNode.constraints = [centerConstraint]

        thirdSceneView.scene = scene
    }
```

Figure 6.9: Function to add protein to the screen

```
var proteinArray = [String]()

@IBAction func flexCoil(_ sender: UIButton) {
    addProtein(name: "fCoil")
    proteinArray.append("fCoil")
}


@IBAction func rigCoil(_ sender: UIButton) {
     addProtein(name: "rCoil")
     proteinArray.append("rCoil")
}


@IBAction func helix(_ sender: UIButton) {
    addProtein(name: "Helix")
    proteinArray.append("Helix")

}


@IBAction func sheet(_ sender: UIButton) {
    addProtein(name: "Sheet")
    proteinArray.append("Sheet")
}
```

Figure 6.10: Actions happen when users press on each Polypeptide Button

6.2.3 Step 3. Create new protein (Figure 6.11, 6.12)

To simplify the process, the combinations of protein are not completely new but instead, loaded from the "Combinations" folder and displayed. To create a smooth transition and generate the feeling of joining the polypeptide chains, the process has three small steps.

Clear everything off the screen

Clearing the screen off will make the transition to a new model feels more approved.

```
327     //Function to clear screen
328     func clearAll(){
329             print("deleting " + String(scene.rootNode.childNodes.count))
330
331             for node in scene.rootNode.childNodes
332             {
333                 print(node.name as Any)
334                 node.removeFromParentNode()
335             }
336
337         }
```

Figure 6.11: Function to clear models off the screen

As the models are added on the screen as nodes (model node, camera node, light node), simply removing all the node from *ParentNode()* would enable clearing off the screen.

Load a combination model

In Figure 6.10, it is shown that every time a button is pressed, besides loading a model onto the screen, it does something else. An empty array variable was declared in the beginning and every time a button is pressed, a value is added to the array. For example, when *fCoil Button* is pressed, "fCoil" is added to the array. These values in the array will then be joined using *array.joined()* in the function to create the combination name as shown in Figure 6.12. Without any separator, these joined values will become exactly like the names of models in the "Combinations" folder, which enable the code to run and load the models from there.

Display 3D text

These functions are called inside of function *createProtein*. If the combination that user created is valid, together with the model, the text will be loaded with "Congratulations", and followed by the names of the polypeptides in order of input. If the combination that user created is invalid, no model would be loaded and instead, only the 3D text will appear with "Sorry"! The combination of (*user-pressed buttons*) cannot be made. See Appendix A for this function.

```
288     func createProtein(){
289         clearAll()
290         lightOn()
291         let newProteinName = proteinArray.joined()
292         print(newProteinName)
293 
294         let newProtein = SCNScene(named: "Combinations.scnassets/" + newProteinName + ".scn")
295         if newProtein != nil {
296             displayText1()
297             let cameraNode = SCNNode()
298             cameraNode.camera = SCNCamera()
299             cameraNode.position = SCNVector3(x: 0, y: 0, z: 0)
300             scene.rootNode.addChildNode(cameraNode)
301 
302 
303             let nodeName = newProtein?.rootNode.childNodes[0].name
304 
305             guard let proteinNode = newProtein?.rootNode.childNode(withName: nodeName!, recursively: true) else {
306                         fatalError("Model is not found")
307                 }
308 
309 
310         proteinNode.scale = SCNVector3(x: 0.008, y: 0.008, z: 0.008)
311         proteinNode.position = SCNVector3(x: -0.005, y: 0, z: -0.005)
312         scene.rootNode.addChildNode(proteinNode)
313 
314             let centerConstraint = SCNLookAtConstraint(target: proteinNode)
315             cameraNode.constraints = [centerConstraint]
316 
317             } else {
318             print("Model is not found")
319             displayText2()
320         }
321 
322             thirdSceneView.scene = scene
323     }
```

Figure 6.12: Function to create a new protein

6.3 Interactive elements

6.3.1 The three gestures to interact with Protein Models (Figure 6.13)

ARKit is a very powerful framework as it cuts off a lot of coding work to enable gesture interaction. To enable gesture interactions, the first step is to drag and drop the gesture on *Main storyboard* from the built-in library. The three gestures used in ProteinAR are *Pinch Gesture* , *Rotation Gesture* and *Pan Gesture*. The gestures are initiated by *.state: .change*. As in ProteinAR, the goal of interaction is the full screen, the area of gesture is set to be *SCNView* and *hitTest* is used to run the gesture.

In the function to generate *Pinch Gesture* as shown in figure 6.13, *SCNVector3* is used with changeable (x, y, z) set in float. By using the two fingertips, users can zoom in and zoom out on the models. The other two functions of rotation and pan gesture are similar to pinch gesture and thus, will not be displayed in code here. The codes can be found in the Appendix A.

6.3.2 Other interactive elements (Figure 6.14, 6.15)

Although it is not a compulsory requirement of the app, more interactive display will make the UI more appealing, thus, some other gestures and touches are added in the app. This might not seem very obvious but it improves the user experience.

```
88      // Pinch Gestures
89      @IBAction func pinchGesture(_ sender: UIPinchGestureRecognizer) {
90          if sender.state == .changed{
91                  let areaPinched = sender.view as? SCNView
92                      let location = sender.location(in: areaPinched)
93                      let hitTestResults = thirdSceneView.hitTest(location, options: nil)
94
95                      if let hitTest = hitTestResults.first {
96                          let plane = hitTest.node
97
98                          let scaleX = Float(sender.scale) * plane.scale.x
99                          let scaleY = Float(sender.scale) * plane.scale.y
00                          let scaleZ = Float(sender.scale) * plane.scale.z
01
02                          plane.scale = SCNVector3(scaleX, scaleY, scaleZ)
03
04                          sender.scale = 1
05                      }
06                  }
07              }
```

Figure 6.13: Pinch Gesture function

Gestures Recognizer for button

For the *Record* button, the two gestures of *Tap* and *Long press* were added. There might be other methods to do this, however, creating two objective-C functions was the simplest solution. For this to work, the button should not be connected to the code as an action, but an outlet. Then, in the *viewDidLoad()*, *GestureRecognizer* can be added to the outlet as shown in Figure 6.14. These *GestureRecognizer* needs handling, which will be handled in objective-C's function (refer to Figure 6.15)

```
402             let tapGesture = UITapGestureRecognizer(target: self, action: #selector(handleTapGesture))
403             let longPressGesture = UILongPressGestureRecognizer(target: self, action: #selector(handleLongPress))
404             recordButton.addGestureRecognizer(tapGesture)
405             recordButton.addGestureRecognizer(longPressGesture)
```

Figure 6.14: Add Gesture Recognizer to Button outlet

```
199     //Tap Gesture
200   @objc func handleTapGesture(){
201     stopRecording()
202     recordButton.tintColor = UIColor(red: 0.4, green: 0.36, blue: 0.46, alpha: 1)
203     print("Tap")
204     }
205        //Long press gesture
206    @objc func handleLongPress() {
207     startRecording()
208     recordButton.tintColor = UIColor.red
209     print("Long pressed")
210    }
```

Figure 6.15: Objective-C functions to handle Gesture Recognizer

Dismiss subview and keyboard (Figure 6.16)

When user finishes reading the guidelines on *Help Screen View* or finishing input in the *textFied*, the sub-screen and the keyboard should be dismissed. For the *Help Screen View*, the solution was to use *UITouch*. This is set so that if users touches every place that is not the *Help Screen View*, the view will be hidden. For the keyboard, usually it can just be set with *textFieldShouldEndEditting* after specified *TextFieldDelegate* in the class. However, since in ProteinAR,

the whole screen are covered in other *UIGesture*, this did not work. The solution was to set the *Return* key to *Done* key in the *viewDidLoad* and then use the function of *textFieldShouldReturn* to dismiss the keyboard, as shown in Figure 6.16.

```
//--------FUNCTIONS FOR INTERACTION AND DESIGN-------
 //1. Dismiss keyboard after entering protein's name
    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        textField.resignFirstResponder()
        return true
    }
//2. Dismiss Help Screen by touching other part of the screen
    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
        let touch: UITouch? = (touches.first!)
        if touch?.view != helpView{
            self.helpView.isHidden = true
            }
    }
```

Figure 6.16: Others interactive elements

Bibliography

Argu, M. (2020). Fast, simple, student generated augmented reality approach for protein visualization in the classroom and home study. *J. Chem. Educ.*, 5.

Cai, S., Wang, X., & Chiang, F.-K. (2014). A case study of augmented reality simulation system application in a chemistry course. *Computers in Human Behavior*, *37*, 31–40.

Eriksen, K., Nielsen, B. E., & Pittelkow, M. (2020). Visualizing 3d molecular structures using an augmented reality app. *Journal of Chemical Education*, *97*(5), 1487–1490.

Goddard, T. D., Brilliant, A. A., Skillman, T. L., Vergenz, S., Tyrwhitt-Drake, J., Meng, E. C., & Ferrin, T. E. (2018). Molecular visualization on the holodeck. *Journal of Molecular Biology*, *430*(21), 3982–3996.

Introduction to ARKit - design+code. (n.d.). Retrieved September 21, 2020, from https://designcode.io/arkit-intro

Introduction to proteins and amino acids (article) — khan academy. (n.d.). Retrieved September 2, 2020, from https://www.khanacademy.org/science/biology/macromolecules/proteins-and-amino-acids/a/introduction-to-proteins-and-amino-acids

Liu, X.-H., Wang, T., Lin, J.-P., & Wu, M.-B. (2018). Using virtual reality for drug discovery: A promising new outlet for novel leads. *Expert Opinion on Drug Discovery*, *13*(12), 1103–1114.

Mittal, A., Manjunath, K., Ranjan, R. K., Kaushik, S., Kumar, S., & Verma, V. (2020). COVID-19 pandemic: Insights into structure, function, and hACE2 receptor recognition by SARS-CoV-2 (T. C. Hobman, Ed.). *PLOS Pathogens*, *16*(8), e1008762.

Norrby, M., Grebner, C., Eriksson, J., & Boström, J. (2015). Molecular rift: Virtual reality for drug designers. *Journal of Chemical Information and Modeling*, *55*(11), 2475–2484.

Rashid, M. A., Khatib, F., & Sattar, A. (n.d.). Protein preliminaries and structure prediction fundamentals for computer scientists, 24.

Ratamero, E. M., Bellini, D., Dowson, C. G., & Römer, R. A. (2018). Touching proteins with virtual bare hands: Visualizing protein–drug complexes and their dynamics in self-made virtual reality using gaming hardware. *Journal of Computer-Aided Molecular Design*, *32*(6), 703–709.

RCSB PDB: About RCSB PDB: Enabling breakthroughs in scientific and biomedical research and education. (n.d.). Retrieved September 8, 2020, from https://www.rcsb.org/pages/about-us/index

Stephenson, F. H. (2016). Protein. In *Calculations for molecular biology and biotechnology* (pp. 375–429). Elsevier.

Swift - apple developer. (n.d.). Retrieved September 8, 2020, from https://developer.apple.com/swift/

Wang, W. (2018). *Beginning ARKit for iPhone and iPad: Augmented reality app development for iOS*. Berkeley, CA, Apress.

Widłak, W. (2013). Protein structure and function [Series Title: Lecture Notes in Computer Science]. In W. Widłak (Ed.). D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, & G. Weikum (**typeredactors**), *Molecular biology* (pp. 15–29). Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg, Springer Berlin Heidelberg.

Wrobel, A. G., Benton, D. J., Xu, P., Roustan, C., Martin, S. R., Rosenthal, P. B., Skehel, J. J., & Gamblin, S. J. (2020). SARS-CoV-2 and bat RaTG13 spike glycoprotein structures inform on virus evolution and furin-cleavage effects. *Nature Structural & Molecular Biology*, *27*(8), 763–767.

Xu, K., Liu, N., Xu, J., Guo, C., Zhao, L., Wang, H.-W., & Zhang, Q. C. (2019, March 27). *VRmol: An integrative cloud-based virtual reality system to explore macromolecular structure* (preprint). Bioinformatics.

Xu, T. (2019). An interactive protein design game: Pocket peptides, 60.