ProteinAR

AN INTERACTIVE 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

Proteins are complex molecules with various functions that are critical to any living organism which comes in all different shapes and sizes. The shape of a protein defines its function, therefore, the study of protein structure is of great importance since it can help scientists to control or modify the protein to change its function and help with various disease treatments.

In this project, ProteinAR - an iOS application was developed with the goal of aiding the field of biology by visualising protein structures and enabling interactions in Augmented Reality. While past projects looked to non-interactive digital environments for protein visualisation, with ProteinAR, users can observe on their smartphone screens simple to complex protein structures by zooming in, rotating or moving them. Users are also able to create new interactive proteins. ProteinAR was written in Swift on Xcode, a native language and IDE developed by Apple for iOS apps, and used the integrated framework ARKit for Augmented Reality functions. When a user inputs a protein ID, ProteinAR downloads protein data and displays it in real-time using RCSB PDB as the main source of protein data. Product evaluation concludes that, with further development, ProteinAR demonstrates potential for the application of augmented reality technology in protein visualisation and design, particularly for researchers, educators, and students of biology.

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

Chapter 1

Introduction

In recent years, there have been major advances in technology and molecular biology. Technology has been of great use in the field of biology aiding in research as well as making the content more accessible. This project focuses on the display and design of protein structure.

A protein is not a single substance; there are many different proteins in an organism or cell, and they come in every shape and size, each performing a unique and specific job ("Introduction to proteins and amino acids (article) — Khan Academy", n.d.). Proteins are considered the "ultimate players in the processes that allow an organism to function and reproduce" (Stephenson, 2016).

Proteins are made up of linear chains of amino acids, called polypeptides. Each protein is formed by one or more polypeptide chains, linked together in a specific order ("Introduction to proteins and amino acids (article) — Khan Academy", n.d.). Proteins are the fundamental components of all living cells (Widłak, 2013). Proteins have a countless number of functions that are extremely important in the biology of many organisms. They form enzymes to speed up reactions by break-down, link-up, or rearranging the substrates ("Introduction to proteins and amino acids (article) — Khan Academy", n.d.). They form 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 of the function of that protein much easier. By being able to manipulate the structure of a protein, scientists can create hypotheses about how to affect, how to control or how to modify protein to design mutations and change a protein's function.

The year 2020 substantiates the importance of studies in molecular biology. The impact of Severe Acute Respiratory Syndrome Coronavirus-2 (SARS-CoV-2) is worldwide, as it is "a newly emerging, highly transmissible and pathogenic coronavirus in humans that has caused the ongoing global public health emergencies and economic crises" (Mittal et al., 2020). At the time of writing, the number of infections worldwide has reached millions, while the death toll is in the

hundreds of thousands. In the efforts to develop a vaccine, much research has been conducted. Some research developed on the protein structure of SARS-CoV-2 has provided insight 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), particular to SARS-CoV-2 are the protein spikes that the virus uses to bind with and enter human cells (Wrobel et al., 2020). The spikes of SARS-CoV-2 are highly stable, and help to bind to human cells tightly. Therefore, analysing the structure of these spikes could provide clues about the virus's evolution. The study of the structure of spike proteins can aid with drug discovery and vaccine design. Understanding the new importance of implementing IT in biological research, **this project aims to aid with protein structural study as well as to raise interest in protein design.**

Due to limitations, this project only provides the first steps in bringing the visualisation of protein into AR and creating a simple protein structure in an iOS application using ARKit framework. **The main goal of this project, however, is to visualise protein structure in AR using an iOS App and allow user interaction with the structures.** There are various previous studies on protein visualisation in 3D and VR, however, studies pertaining to AR are limited, especially on iOS. This project proposed the implementation of protein structures display to serve as a trial for future study and research as it may provide more a visually appealing display than simple 3D, and reduces the side effects of VR. All protein models to be displayed are retrieved from RCSB Protein Data Bank.

The aim of this project is to visualise protein structures in two ways. The first way is directly displaying the complex protein models from RCSB PDB, and the second way is visualising the design of a simple protein structure that user created. The second function is implemented so that this project's application is not only appealing to biologists but can also be used by anyone curious about biology. The ability 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 the iOS system was developed: **ProteinAR**. This app has two main categories: **education** and **mini-game**.

The **education** category assumes that the users have prior knowledge of proteins. They can input the name (ID) of a protein and get the 3D visualisation of the protein structure in AR. Users can study the protein by zooming in, turning, and flipping the protein structure. Due to the complexity of protein structures, it is not easily observed even under advanced microscopes. Thus, the ability to interact with the structures in this way can be largely beneficial to researchers. Moreover, since this is a mobile app, users can interact and discuss the structure with other users at the same time, which can be considered a promising tool for study and research on proteins.

The **mini-game**, is user-friendly and accessible to those who are unfamiliar with proteins or biology in general. Users can combine the polypeptide chains (i.e. Flex Coil, Rig Coil, Helix, Sheet) to create a new protein. This might make the study of proteins sound more appealing to users. and motivate those who wish to enter the field. In this game, users are also able to interact with the polypeptide chains and protein models.

Last but not least, ProteinAR integrates other functions to make the app more interesting,

such as enabling photo-capture of the proteins, video-capture of the process, and providing users with additional information about protein.

This paper will elaborate on the background and research, the problems and solutions, the design and implementation, and the final evaluation of the project.Due to time constraints as well as the ongoing pandemic, there were some limitations to the project, which would also be mentioned in the paper.

Finally, this paper will discuss some critical points in dealing with the fairly new AR technology, especially use of the ARKit framework concerning: the feasibility of retrieving and displaying PDB contents, the usability of the app (AR) and room for future work.

In this paper, Chapter 2 presents findings from a literature review on AR and the usage of AR technology in education and research, especially in the biological field. In this chapter, some current apps on protein visualisation will be mentioned with personal insights. Chapter 3 introduces the technological background of software and language used in developing ProteinAR. Chapter 4 proposes the requirement functions as solutions for the current problems while Chapter 5 demonstrates the app's structural design based on these solutions. The implementations of this project can be found in Chapter 6, along with code snippets from the source code and explanations. After that, in Chapter 7, the conducted tests' results will be exemplified with an overall evaluation of the project. Finally, Chapter 8 encapsulates the project and outlines the directions for future work. Some code snippets can be found in Appendix A.

Some important technical notes about the project: **ProteinAR** was designed in Xcode 12, written in Swift 5, on MacOS version: Catalina 10.5.5. Because there is no support for AR on MacOS, the built-in simulator is unable to display AR functions 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 be unable to run ProteinAR and might also generate some unwanted errors.

Chapter 2

Literature review & Existing products research

This chapter goes through some literature on Augmented Reality (AR) and the application of AR in education, with the specific mention in the biological field. Then, a brief explanation of protein structure will be provided, follow by research on the existing solutions of protein structure display in mobile application, Virtual Reality (VR) and AR with some personal insights. Finally, the finding summary is presented.

2.1 Augmented Reality and application in education

2.1.1 Augmented Reality introduction

Augmented Reality (AR) is a technology that involves "the overlay of computer graphics on the real world" (Silva et al., 2003). AR allows users to look "at the real world and increases it with additional information generated by a computer" (Chamba-Eras & Aguilar, 2017). AR acts as a bridge, connecting physical and virtual objects, combining two worlds into one, and enables interaction between them by adding information to the real-world in real-time (Chamba-Eras & Aguilar, 2017).In AR, the user is able to stay in touch with both contexts of the real world and the virtual world.

On the other hand, even though often being mistaken for another, Virtual Reality (VR) is used to define the technology that allows the computer to generate 3D environments that users can enter and interact (Silva et al., 2003). The complete scenarios are generated by computers, creating the sensation of being physically in the generated scene for the user. In VR, users lose the context of the real-world but instead, are just aware of synthetic realism.

AR is similar to VR in the way that virtual objects are generated by the computer. However, while the goal of VR is to create an immersive experience for the user by shutting down the real physical world and replace it with a completely synthetic environment, the goal of AR is to enable the user to stay in touch with the real-world, while being able to interact with virtual objects (Chamba-Eras & Aguilar, 2017). Because of this, the defining characteristic of AR is

that it adds layers to the real-world vision. Using AR, users have more freedom of movements while projecting images. The main components of AR are scene generator, tracking system, and display. The scene generator is responsible for rendering the scene for binding real-world scenes and virtual objects. The tracking system is important because the objects in the real and virtual worlds must be "properly aligned with respect to each other, or the illusion that the two worlds coexist will be compromised" (Silva et al., 2003). As for the display, currently, there two well-known types of display for AR implementation. The first one is the implementation of AR smart-glasses such as the Microsoft HoloLens, Google Glass, Apple Glass. In contrast to VR goggles, AR smart-glasses look similar to sunglasses or normal glasses, thus, causing no discomfort to the users. The second type of implementations is on AR apps such as Pokemon Go. In this type of implementation, smartphone cameras are used to track the surrounding environment and digital models are "superimposed into the real-world" (Moro et al., 2017).

Using an AR-implemented-app does not only have the advantage of allowing the user to interact with both the real and virtual elements of their surrounding environment but also in this way, no extra equipment is required, AR app has become more and more common, especially in the field of education (Moro et al., 2017).

2.1.2 Augmented Reality in Education

There have been multiple articles promoting learning through AR. The study from the University of Cologne discussed the benefits of AR in educational environments with a conclusion that applications applying successful use of AR have been improving learning, especially in language education, mechanical skills, and spatial abilities training (Diegmann et al., 2015).The study from the University of Girona analysed 32 studies from journals about the AR trend in education. The finding results were that the number of published studies about AR in education has progressively increased year by year, while the fields of education which had the most AR applications are Science, Humanities, and Arts, in which AR has been effective for better learning performance, learning motivation, student engagement and positive attitudes because the advantage of AR is allowing interactions and collaboration (Bacca et al., 2015). The study by the University of La Laguna also concluded in the higher performance of students studying using AR applications as they can do it in their own time (Martín-Gutiérrez et al., 2015).There were also some specific proposals on how to integrate AR in learning. Brown and Gabbard proposed using AR to personalised learning for every student (Brown & Gabbard, 2015) while Huang and his team came up with a learning model based on AR, in which educational resources are discovered on the internet and be translated to AR for an educational environment to improve students' emotions and experiences (Huang et al., 2016).

Molecules studying can be confusing. In 2019, a study conducted using AR for leaning atoms and molecules reaction by students was conducted. In this study, female students, who do not show much interest in science and technology were the target of research (Ewais & Troyer, 2019). The study found that using AR technology to visualise the molecules did motivate these students to study, as it helps them understand the structure much better. Another study conducted using an augmented reality web application for high school education in biomolecular life science also discussed the fact that it is difficult to understand the spatial relationship of a

protein structure. "Proteins and protein interactions are too small to be seen by far, even under advance microscopes" (Nickels et al., 2012). However, by using this AR web-based app, students show much more interest and understanding of the field (Nickels et al., 2012). Another study from Georgia Gwimmet College in 2018 about the developing of an AR app that transforms 2D molecular representations into interactive 3D structures that user can manipulate also showed that students who have used augmented reality models found it convenient and faster than the traditional mode, and they also prefer it more as they have control over molecular manipulations (Behmke et al., 2018).

These studies showed above showed how important it is for AR to be brought into molecular biology study.

2.2 Protein Structure

In the field of biology, researchers give prioritised attention to the shape of a protein. Proteins are "the most important macromolecules in all living organisms" (Rashid et al., n.d.). Sequences of amino acids that bind 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 its tasks, thus, knowing the protein structure is very important. This task is defined by the shape of the protein, which makes the understanding of protein structure of great importance. There are four different levels of protein structures: Primary Structure, Secondary Structure, Tertiary Structure, and Quaternary Structure. A sequence of amino acids in a chain forms a *Primary structure*. These chains, then, would fold into three different shapes (Helix, Coil or Sheet) where the alpha-helix, the beta-sheet, and the random-coils are positioned, which is called the *secondary structure*. The combination of these chains of helix, coil and sheet (polypeptide chains) forms a 3D structure – the *tertiary structure* of a protein (Rashid et al., n.d.). The *quaternary structure* is a large assembly of multiple polypeptide chains (Figure 2.1).

Given the importance of protein structure, designing proteins is extremely useful as this can help to change its functions, or create new functions. In ProteinAR, users will get to design protein structure by combining different protein secondary structures of helices, coils, and sheets to form tertiary structures.

2.3 Existing solutions to protein visualisation

"Proteins are three-dimensional (3D) objects" (Ratamero et al., 2018). Computer models for protein have 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.

2.3.1 Protein visualisation in mobile applications

There are numerous mobile applications in which proteins are visualised in 3D. The RCSB Protein Data Bank (the single worldwide repository of protein data) also provides a mobile app allowing data access and visualisation. The protein can be downloaded directly from

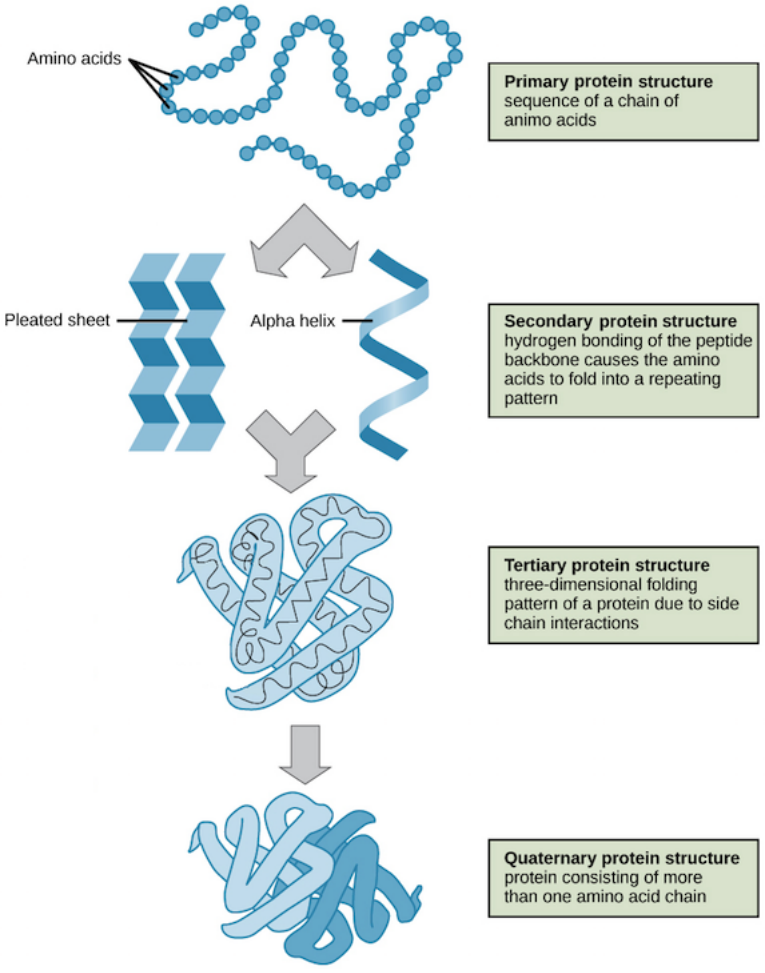

Figure 2.1: Orders of protein structure - source Khan Academy("Introduction to proteins and amino acids (article) — Khan Academy", n.d.)

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, and visualises proteins in 3D. There are some molecule viewers available on iOS. Unfortunately, most of them are no longer in use or experienced technical difficulty, and have therefore been removed from the Apple App Store. iMolview is still available, however, the interface is not very user friendly.

2.3.2 Protein Visualisation in VR

The advancement of implementing VR in Protein Display

Visualisation of proteins on the computer has been a great step, however, it lacks the immersive salience of 3D presence, and leads to limitations in the analysis of 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 users to have the freedom of hand controllers for simple manipulation and interaction with the protein instead of the conventional manipulation on 2D using a trackpad, mouse, and keyboard (Goddard et al., 2018). This makes VR's entrance into the world of molecular biology and protein visualisation more than welcome. HMDs[1] are commonly used because they are accessible, becoming increasingly more available, and are affordable. VR games have become popular, thus the tools for programming software that are compatible with HMD are effective and cheap. Projects 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 proteins (Ratamero et al., 2018).

There have been many advanced VR projects in molecular biology. In particular, MOLECULAR RIFT is an open source tool that creates a virtual reality environment steered with hand movements, and incorporates 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 movement creates a much better experience for the users than merely experiencing the 3D (Norrby et al., 2015).

Though research shows that the technology in displaying Protein in VR is advanced, the tools that need to be installed on desktop systems are often tedious (K. Xu et al., 2019). The configurations might be different for different systems and therefore cause compatibility issues. Sharing between system is also 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 they only support a few file types and cannot perform complex tasks for analytical purpose (K. Xu et al., 2019). A few solutions were proposed for an integrative cloud-based system that can directly access databases and uses VR technology to visualise and analyse macromolecular structures, such as VRMOL. This might be the new direction for protein visualisation in VR.

[1] Head Mounted Display

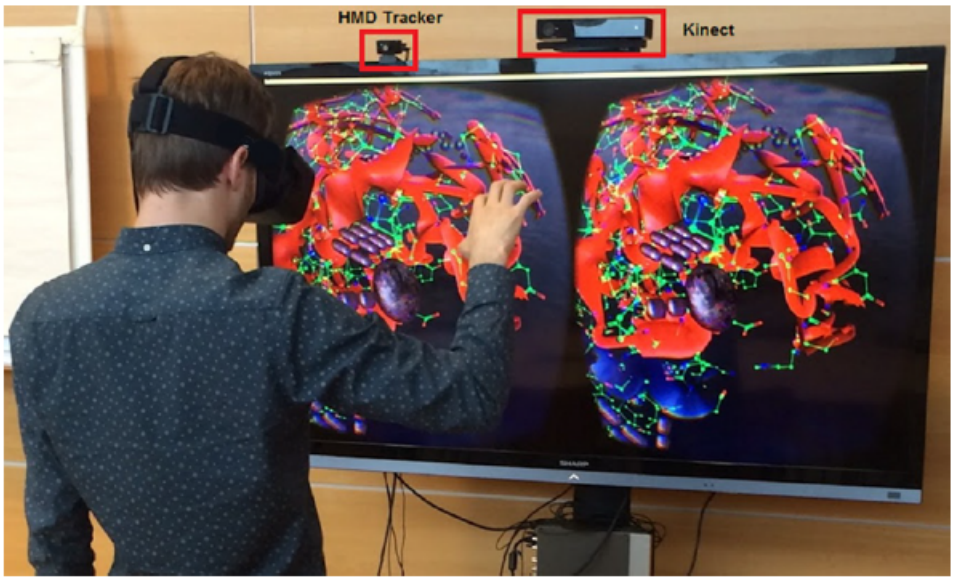

Figure 2.2: Oculus Rift (HMD) and Kinect v2 sensor placement used during Molecular Rift development

The limitations of using VR in Displaying Protein

Even though VR implementation of protein display has come far, limitations are inevitable. First, the limitations in the associated hardware/software may lead to an unsuccessful application of VR, which leads to the inaccuracy and impreciseness in the results of using the application. With the increasing development of VR techniques and the growing popularity of VR games, software and hardware are becoming more integrated with VR and therefore more compatible, however, they are still costly and need to be increased in fidelity (Liu et al., 2018). The second point concerns the unnatural feeling of using VR. Even though VR offers a realistic view, the users must wear goggles that are not transparent and thus block the vision of the real world. Furthermore, the head movements are unnatural because users will have to try to move their heads to see contents. New HMDs are better because they are much lighter but mostly VR devices are still quite bulky and are relatively difficult to use. Thirdly, most VR users claim to have motion sickness. This happens because of the disparity between what the body and the eyes of a user experiences at the same time. The actual physical actions and the actions that are carried out in VR might be different and this causes motion sickness to the users. Due to this, VR can only be used for a limited amount of time.

2.3.3 Protein Visualisation in AR

As AR gains popularity, more projects are underway, but this is limited as it is an extension of VR, and it is still very new. Some studies show that AR being used in science teaching such as displaying molecular biology in AR has yielded in good results for students, as it takes less imagination and makes things easier to understand (Cai et al., 2014).However, there are not many AR apps available to support visualising molecules.

As mentioned, there are not many projects concerning the visualisation of molecules on AR. Unlike VR, where there are various numbers 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. One such app, BiochemAR, was released in 2019 and is available on both the App Store (for iOS) and Google Play (for android). According to the developers, the idea of the app is to create a simple, easy-to-use teaching tool for both teachers and students in the classroom(Sung et al., 2020). The main function is to display protein in AR by scanning a QR code, thus the design is relatively basic. When a QR code is scanned, the app will use the smart devices' built-in camera to bring the protein structure into life through VR as shown in Figure 2.3.

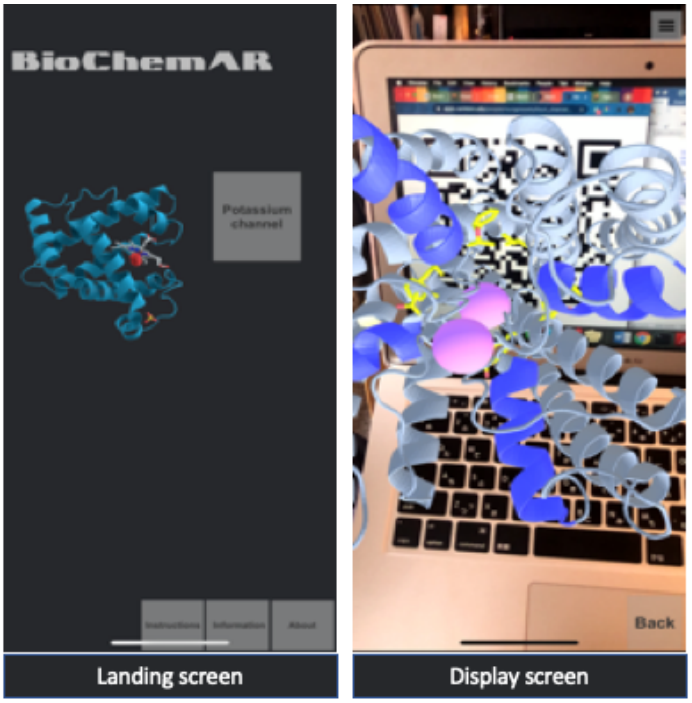

Figure 2.3: BiochemAR app screen shot

As the main purpose is to make things simple and easy to use for teachers and students, there is no other function or interaction between users and the protein. Proteins are simply visualised and users can move the phone around to look at the protein in different angles and sizes.

Having the same idea, another app called AR Assisted Visualisation was developed in 2020 to visualise proteins. 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.

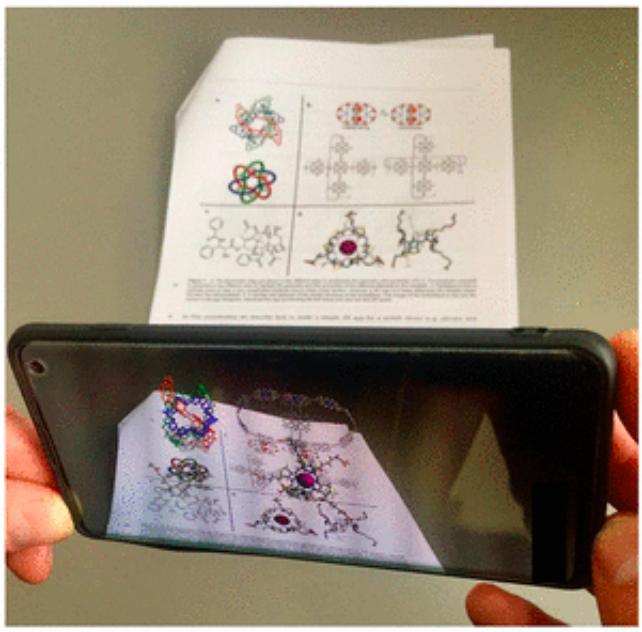

Figure 2.4: AR Assisted Visualisation App (Eriksen et al., 2020)

2.3.4 Personal insights

Integrating protein visualisation on mobile apps is a good solution because of its availability. Most students have access to a smartphone and it is handy to bring around as it is not bulky nor need specific customisation. However, the AR apps on protein visualisation are relatively new (released in 2019 and 2020). Thus, there are not many user interactions and functions to it. To use the aforementioned apps, a certain document with information of the protein, whether it is a figure of a protein or a QR code, has to be printed in order to get the AR visualisation. Moreover, the proteins can be viewed but cannot be manipulated in any way. Furthermore, these apps are one-side oriented as users can only view proteins but cannot create new ones.

2.4 Finding summary

With the advantage of allowing the user to stay in touch with the surrounding environment and not requiring any extra equipment, the use of AR in mobile applications is certainly gaining popularity. In education, the use of AR technology motivates students and establish better performance. In molecular biology, according to many studies, students are much more interested and have a greater understanding when AR technology is used in teaching. In a recent research, it showed that when undergraduate students created their own AR-protein, they were enthusiastic when performing this function, 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 offers a promising curriculum for biochemistry. Not only with education, but the integration of AR in molecular biology can also benefit researchers as it makes small interactions which are invisible under microscope visible. Currently, many

applications are being developed to visualise protein structures, however, there are still limitations. Nevertheless, in the near future, AR technology will become much more commonly used in molecular biology.

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

Based on the above findings, this project aims to provide a valuable tool for the field of biology with the experimental app ProteinAR. With further development, it can be applied as an education tool, and for researchers in need of accessible and accurate protein models.

Chapter 3

Methodology

This chapter introduces the software, language, and framework that were used to develop ProteinAR.

3.1 Software

3.1.1 Xcode

Xcode is an integrated development environment (IDE) for MacOS. It was first released in 2003, and 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 project's development process, Xcode has had a few version updates. The latest update was Xcode version 12.

Advantages of using Xcode

ProteinAR is written in Swift, a native language for iOS apps, released by Apple. Since Xcode is the native IDE of Apple, the compatibility is ideal, making the app and tests run faster and less error prone. Xcode is a highly intuitive IDE with a main storyboard interface, visualising the designs elements of an app as well as various built-in functions 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, and more ("Xcode 12 - Apple Developer", n.d.).

Disadvantages of using Xcode

ProteinAR uses the built-in ARKit framework. As this requires camera accessibility, tests cannot be run on the built-in iPhone simulators but instead, a real iPhone device. This creates a significant disadvantage due to persistent iOS updates which can cause compatibility issues between iOS and Xcode. Moreover, in some updates, the supporting packages change, causing compatibility issues that need to resolved.

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*, or *.obj*.

3.2 Swift

Swift is a powerful programming language for Apple platforms. Apple released Swift in 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 has been considered one of the most loved programming languages on Stack Overflow for many years as it is highly interactive, with concise and expressive syntax which runs fast. There are several improvements compared to other languages: there is no need for semi-colons, UTF-8 based encoding is used, strings are formatted in unicode, 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 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 being a relatively new language is in a state of regular fluctuation meaning changes to sytax and package names are relatively common. An additional difficulty in coding in such a young language is that problems might be too new to have a solution, which is the most challenging aspect of Swift.

3.3 ARKit API

The technology to develop Augmented Reality was ready for mobile devices, however, algorithms for detecting objects in real world and displaying virtual objects are highly complex. This is why Apple released ARKit in 2017 as a software framework, making developing an AR iOS app significantly 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 furniture in their own homes. Ferrari also used Metaio's technology to allow customer changing colours of cars in showrooms, and view a 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 Apple 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 creation of AR apps easier than ever (Wang, 2018).

3.3.1 Basic understanding of the ARKit

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

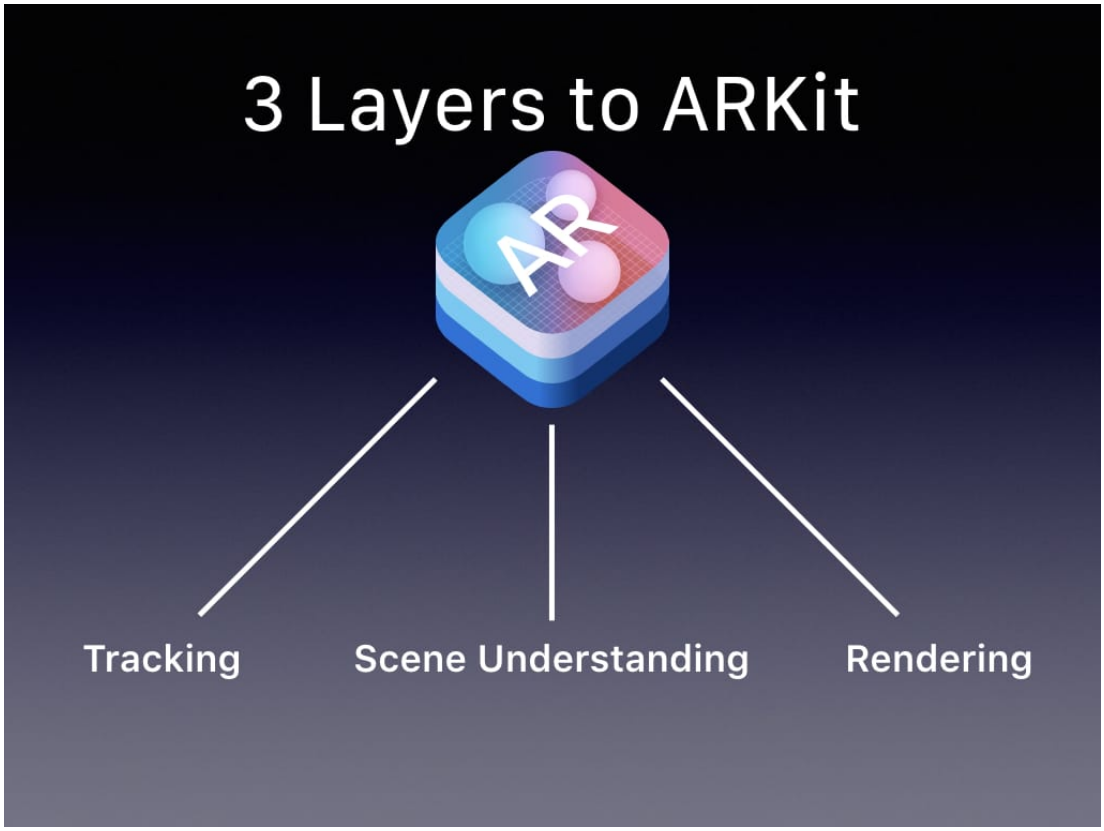

Figure 3.1: Three Layers to ARKit

Tracking is the key function of ARKit. Without ARKit, it would be very complex for developers to write algorithms 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 place a virtual object in 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 Requirements for ARKit

As previously mentioned, since augmented reality requires access to a high resolution display and camera, ARKit apps can only run on the following 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 use 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 instead has to be tested on a real iOS device from the list above via USB connection. Both Swift and Objective-C can be used to create an ARKit app. This project chose Swift as the language for its ease of use and speed. The 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 RCSB Protein Data Bank

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 ("RCSB PDB: About RCSB PDB: Enabling Breakthroughs in Scientific and Biomedical Research and Education", n.d.). RCSB was the first open access digital data resource for Protein Data Bank. It provides access to 3D structural 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 allow interaction. 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 is a basic website with only string type of data, allowing the GET method to get information from the server to the app easily. However, the PDB files containing 3D structural information of the protein were not available, so it was used as a test to discover if the POST and GET method functioned correctly in the app. I-TASSER predicts protein structure and function after users enter the sequence of amino acids. Similar to RCSB, I-TASSER allows free downloading of 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 users need to enter their emails into the server and receive the PDB files a few hours later. As the goal of ProteinAR is to visualise protein structures and display them instantly, RCSB was chosen for the database as it fits said goal.

3.5 Summary

ProteinAR is an iOS app. It was written in Swift 5 using Xcode, using ARKit API. The dataset in which protein files are downloaded from is directly connected to RCSB PDB. Other sources of protein data websites were used, such as Protein Parameter, and Protein Structure Function

and Prediction I-TASSER Server, in order to test the application during development. The app only display full functionality on an iOS device, not a built-in simulator due to the requirement to use the camera to achieve the AR function.

Chapter 4

Analysis: Goals and Functional Requirements for Solutions

This chapter identifies the goals of the project to make it predominant to the existing solutions to protein visualisation on an AR application. Then, the functional requirements as well as the non-functional requirements to help achieve these goals are discussed.

4.1 Project Goals

ProteinAR is an iOS application that visualises the three-dimensional structure of proteins. The project was set with three main goals:

(1) Provide an **educational experience**: enables download and visualisation of user-specified protein structure with data from RCSB PDB. As in Chapter 2, there are a few existing apps that use AR to visualise protein. However, these apps need to scan a code or an image to display the protein, which creates a significant limitation as the protein needed to be pre-rendered. The apps that allow direct protein structure viewing in 3D by entering proteins names also are available but not in AR. Thus, the *first goal* of this project 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 types in the name of the protein.

(2) Provide an **entertaining experience**: enables the creation of new proteins by the combining of polypeptide chains. As the existing apps on protein visualisation are more focused on simply displaying the protein, this project is set on bringing entertaining elements into the app by the addition of a mini-game function in which users can create new proteins. The *second goal* of this project 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**: enables interactions between the user and the protein models or polypeptide chains displayed on the screen. By touching the models, users are able to scale the proteins, move the proteins around, and rotate the proteins. The findings from Chapter 2 shows that little effort has been put into interactive elements in existing products.

The main function of the products is showing the protein. Therefore, the *third goal* of this project is to enable interaction with the 3D models in AR.

4.2 Functional requirements for solutions

4.2.1 Educational purpose: Visualising Protein from RCSB PDB server

There are a few problems must first be addressed in order to visualise the proteins. Firstly, the app needs to be able to send requests to the RCSB PDB server. Secondly, the app needs to be able to download the files from the server. Thirdly, the app needs to be able to track the location of the downloaded files. Finally, the app should be able to open the files 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 retrieve the files. Based on this approach, the first attempt 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 to the destination URL where the specified embedded method is *POST*. This is achieved by first accessing the website, then inspecting its element to find the action method as well as the parameters needed for 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 as with *POST*; usually the parameters can be found by inspecting the source code of the website, often under *form action*.

To test the function, ProtParam was used 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. 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 action* function. The PDB files are directly downloaded by a separate URL in which the only variable 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 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, after which it can can be moved to the *Document Directory*. ProteinAR specifies the format of the download by saving it as "proteinName.pdb". The solution to visualise the *.pdb* is to convert it to a *.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 affects app performance as the entire content of the project is loaded every time the app is run. The solution

that was used in this app is to keep all downloaded files in the *Document Directory*. The file path and file name will be specified so that whenever a model is needed, it will be identified using these attributes. For loading the file, there needs to be a converter which converts 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 technical difficulties as well as time constraints, a converter could not be made and remains the largest avenue for future work on this app. Therefore, to demonstrate the app's functionalities, a sample folder of existing "protein.dae" files is imported.

4.2.2 Entertainment purpose: Create new proteins from combination

Add polypeptide chains to screen

The app needs to be able to display user-selected individual polypeptide chain. There are four types of polypeptide chains: 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, they 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 the name of each models as a parameter, only one function is needed to add each of the four different polypeptide chains using four different buttons.

If a model is loaded on screen and the location is not specified, the model may render off-screen. An additional problem can occur when two of the same polypeptide types are loaded: in such an occurrence, the two models will appear in the exact same location with the exact same orientation appearing as if there is in fact only one model on the 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 to the screen, ProteinAR must be able to combine these chains into proteins. If such a combination exists, then the resulting protein should be displayed. 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 the protein name is created. Every time a user adds a polypeptide chain to the screen, the name of the protein is appended to the array. After a user clicks the "Try" button to combine the polypeptide chains, the names in the array are concatenated using the *array.joined()* function. The name of the models 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 Chapter 6 for further details.

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 using the touchscreen. To make this happen, *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 model.
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: What Is Core Data?", n.d.). In this app, to manage the downloaded PDB files, Core Data is used in which Protein is defined as an *Entity*, having two attributes *name* and *location*, stored as *String*. After the file is downloaded and stored in *Document Directory*, a new *Protein Entity* is saved into Core Data, with two attribute values of *name* and *location*. When a model is called, it will use the specified "proteinName" name attribute and "filePath.dae" location attribute to open the file in the app. Because there are only two attributes in a *Protein Entity*, it can easily be replaced by using String value directly in the app to call the files. However, taking the future work into consideration, where more attributes can be added to a *Protein Entity* such as molecular weights of protein, number of amino acids in a protein, etc., using Core Data can help to manage and display all these values more effectively.

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 why constraints are important in developing an iOS app. ProteinAR does not have 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 is 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 and Polypeptide chains models management

The combinations of polypeptide chains are kept in a "Combination" folder and has a naming convention that makes it easy to find each 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 combine. In order to do this, the function to add polypeptide chain is created separately.

4.4 Summary

There are three main goals the project was set to achieve. The first goal is to allow download and visualisation of protein structures from RCSB. The second goal is to allow new protein structures creation and the third goal is to enable interactions with the structures. For this to happen, several functional and non-functional requirements are needed. These include functions to download the files, display the files, loading models as individual and combination, as well as gesture recognition. Besides, non-functional requirements are also mentioned such as the use of core data, constraints, and models management.

Chapter 5

Project Design

This chapter elaborates on the design of ProteinAR by starting with the skeleton of the app, followed by the design of the solutions for each function. The overall structure of the app is illustrated by a UML diagram. After that, the UI design is explained. A brief description of the design of core data is presented at the end of the chapter.

5.1 Application Skeleton Design

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

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

(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* maintains coherency for the same content, while at the same time keep less words per screen, making it more appealing to users. Users can click on the *GET STARTED* button to go back to the first view to explore the options or simply 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. Since the main function on this screen is for the user to input the protein's name and get the pdb file back, the design is kept simple with a *textfield* and a *GET button*. To make the app more appealing, there are four more buttons with four more 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 section 5.2.

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

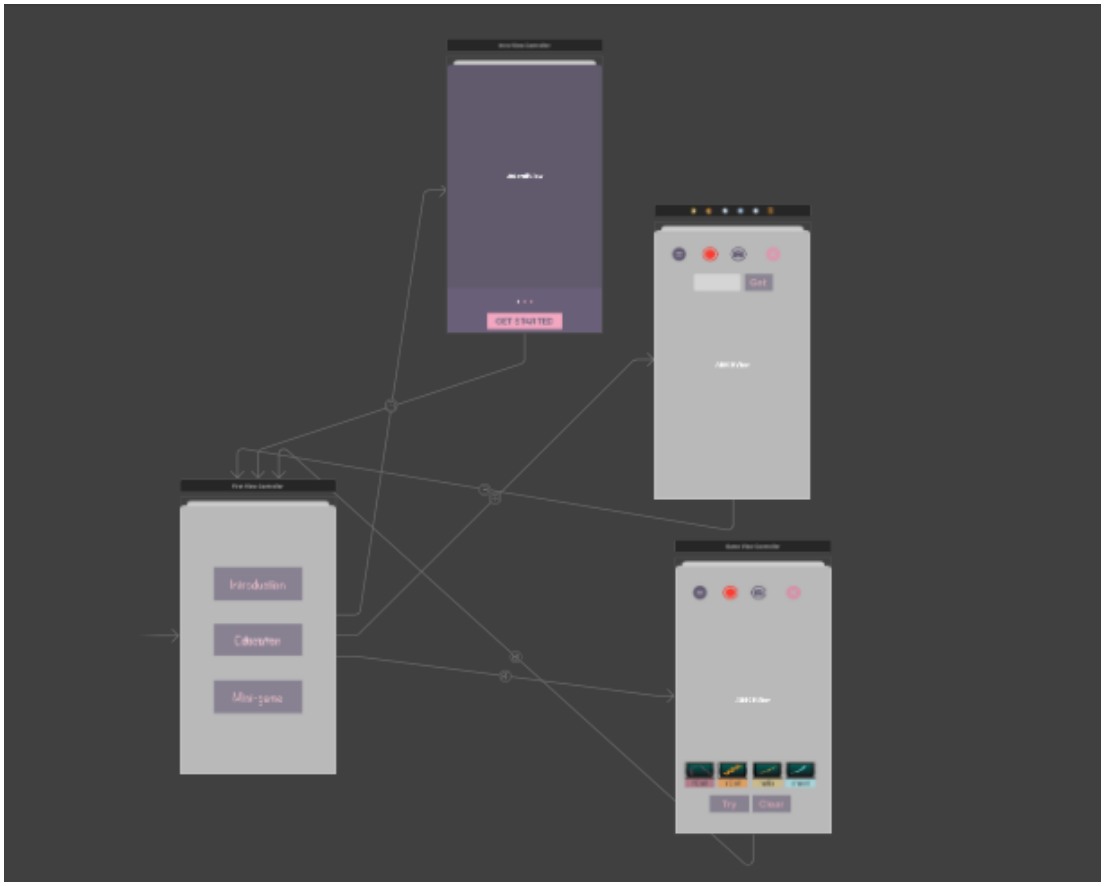

Figure 5.1: The skeleton of the app

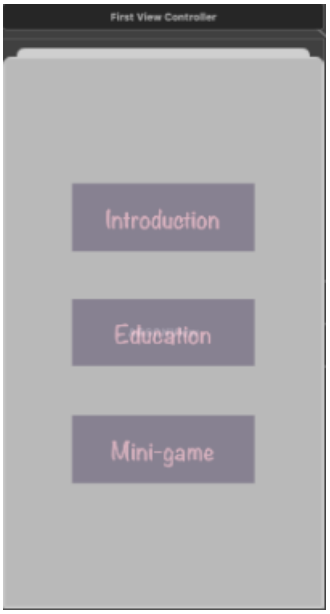

Figure 5.2: The First Screen View – Landing view after launch screen

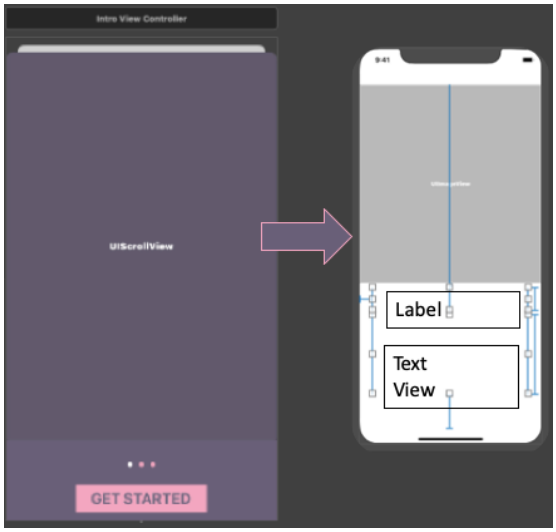

Figure 5.3: Introduction View Controller and Page Controller

Figure 5.4: Education View Controller

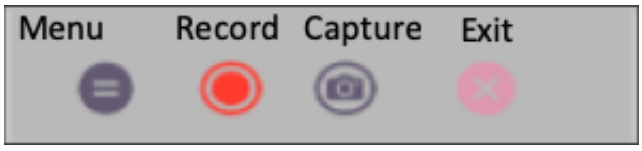

Figure 5.5: Four buttons on top of Education View Controller and Game View Controller

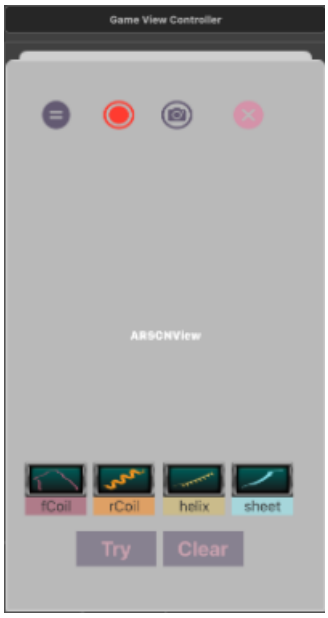

Figure 5.6: Game View Controller

5.2 Solution Design

Figure 5.7 shows the overall structure of the app. In this Figure, the four frames represent the four screens of the app. Different colours are used to indicate different components of the app.

- **Purple** represents the buttons displayed on the app's screen.
- **Yellow** represents user action (swipe, press, tap, long press, type).
- **Green** represents the options displayed on the screen after an action was taken.
- **White** represents the actions that the app executes after a button was pressed or an option was chosen.
- **Pink** represents the destination screen or URL the app opens after a button was pressed or an option was chosen.

The app starts from "First View Controller" and depends on the chosen buttons, the suitable "View Controller" will be on display. From any "View Controller", the user can always go back to "First View Controller" by pressing on the "Exit" button. The details of each function design will be elaborated in the following part of this section.

5.2.1 Utility buttons

The utility buttons are the same on both Education View Controller and Game View Controller. This creates coherency throughout 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.

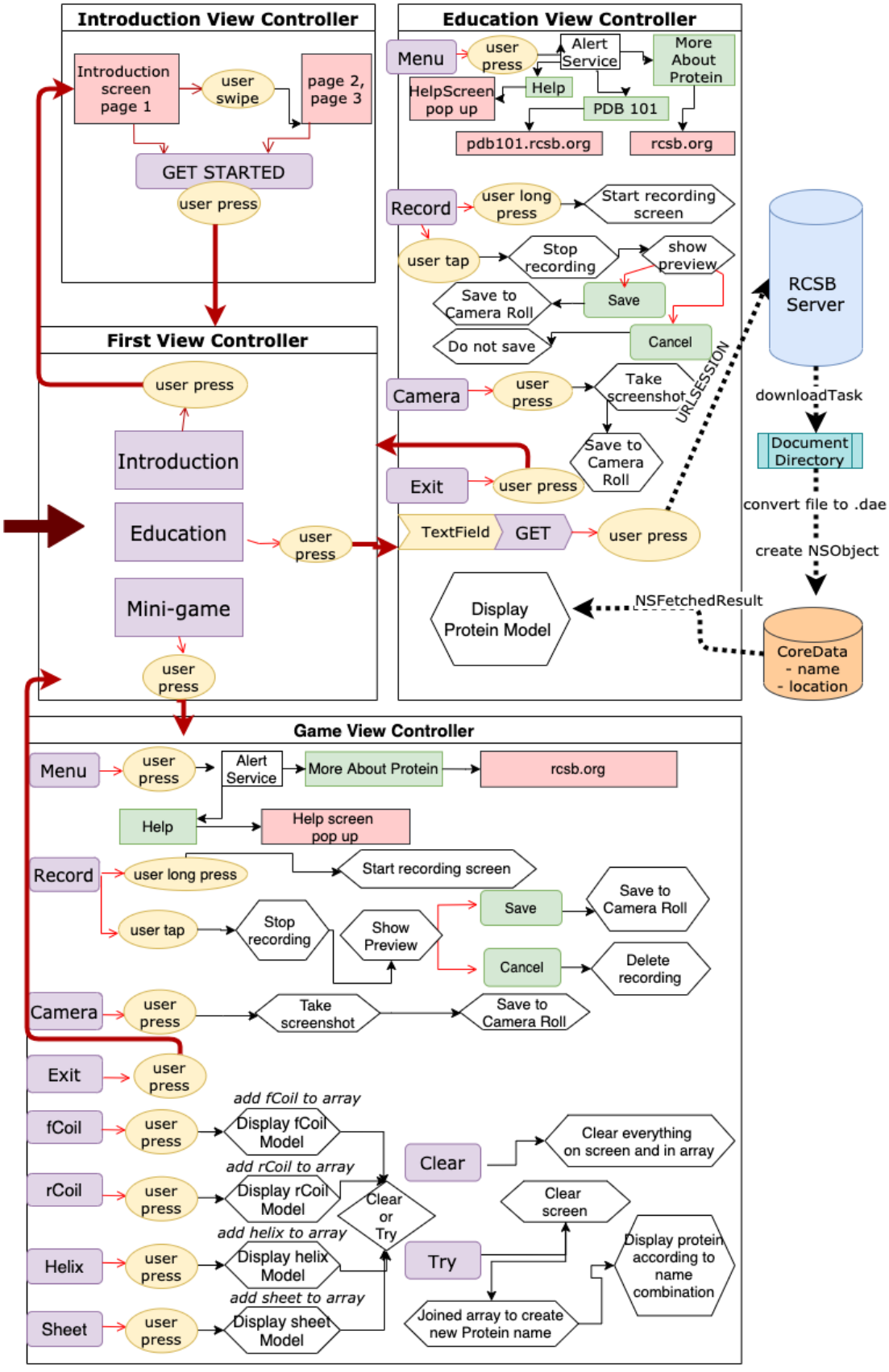

Figure 5.7: Application Solution Design

- *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 the user four options: *More About Protein*, *Help*, and *PDB 101* and *Cancel*. 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 guidelines on how to use and navigate around the *Education View* and *Game View* respectively. Since this is more akin to a demo-app, the options only directly open link to the RCSB website, however, in future development, more in-depth options can be integrated to create a more scientific experience for the user.

- *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 creating 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 users to watch the recorded video before deciding to save the video or not. The two actions of *long-pressing* and *tapping* are enabled using the *UIGestureRecognizer* of the ARKit.

- *Camera* is the function to capture the AR screen and save the photo to the phone's *Camera Roll*. When the user taps the *Camera* button, the screen will be captured and saved immediately. The UIButton flashes colours to indicate that the shot has been taken. Even though iOS already has the screen-capture function, by using that, all the buttons on the screen will also be captured, which is not desirable. With this *Camera* function, users can save a photo of just the protein they want.

5.2.2 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 much trial and error as mentioned in Chapter 4 of using the *HTTP Request POST* and *HTTP Request GET* method. In the process of making the task possible, Alamofire was also considered as an option. Alamofire is a Swift-based HTTP networking library 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 attribute types are of *String* type. 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 visualised in Figure 5.8.

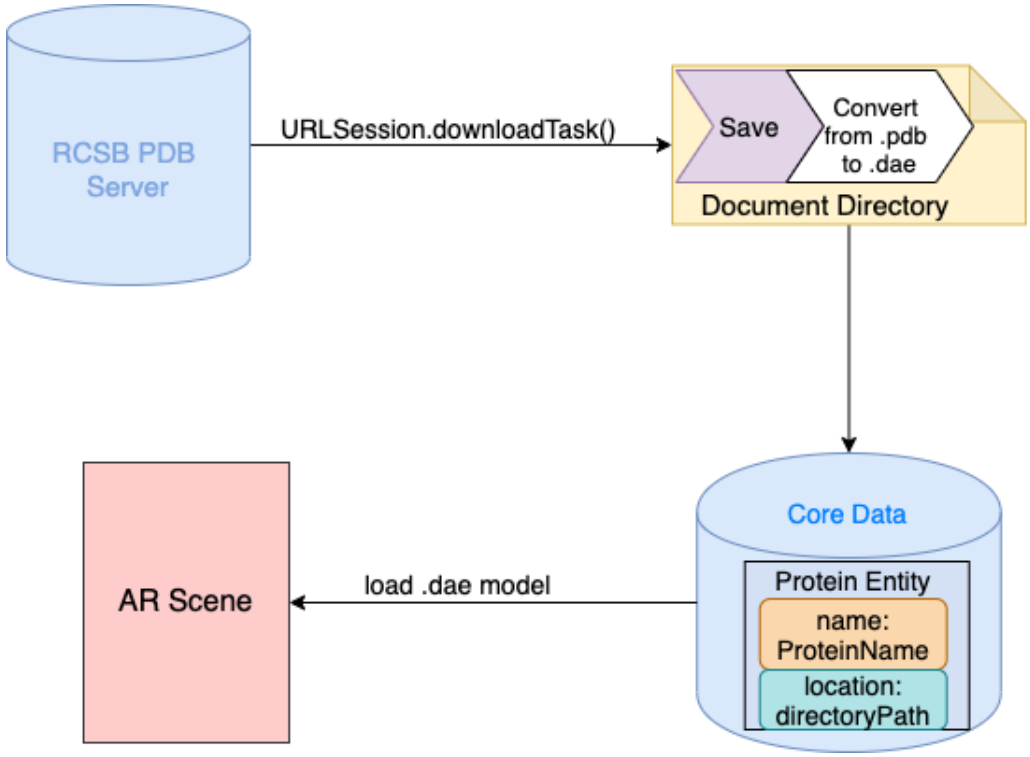

Figure 5.8: Process of downloading, saving and displaying downloaded protein model

5.2.3 Visualisation of protein models from PDB files

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 and therefore could not be implemented into the app.

5.2.4 Combining polypeptide chains into a protein

Each polypeptide chain is designed to be referred to as a value in an array. Every time a user presses on a polypeptide chain's button, that model of protein is displayed, while at the same time, that model's name is added as a value in the array. After these actions, if the *Clear* button is pressed, not only are the models on the screen 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 invalid, no models will be displayed. The errors will be caught and, on the console, "This model does

not exist" will be printed. On the user's end, 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.

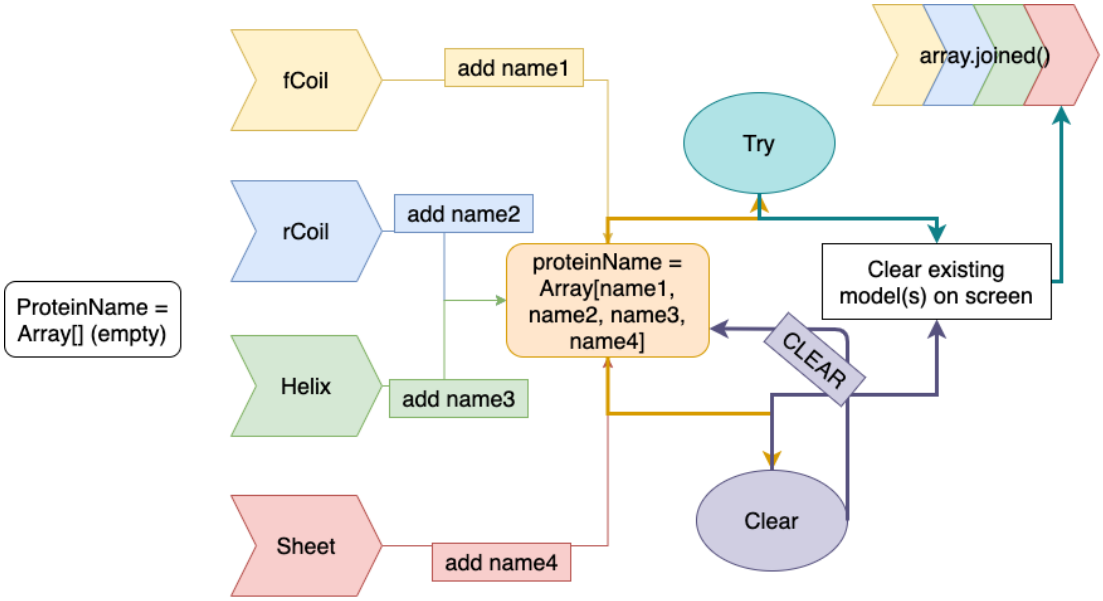

Figure 5.9: Create a new protein name from existing ones

5.3 User Interface (UI) Design

5.3.1 Introduction to user interface

In order to appropriate the tools of computers and smart devices, users need to communicate with them. The way users 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 commands and receiving feedback in a chain to complete a task. The user interface of any computer-based product does not only create first impressions which convinces users to continue to use that product, it also plays an important role in maintaining the interest of the users. With a complicated or inefficient UI, users would not want to keep using the product because it requires too much cognitive effort. Therefore, a UI should be *intuitive* – be kept simple where no training should be needed to operate, and be *efficient* – functions are precise, on point, and *user-friendly* ("What is a User Interface?", n.d.). Currently, there are three formats of user interfaces ("What is a User Interface?", n.d.):

- **Graphical User Interfaces (GUIs)** – interactions happen through visual representations on digital control panels such as a computer desktop, or a website interface.
- **Voice-controlled interfaces(VUIs)** – interactions happen through voice representation such as Siri, Google Home or Alexa.

- **Gesture-based interfaces** – interactions happen through physical motions in 3D spaces, such as 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. This was created in GIMP and then exported to various sizes 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, making it easy to remember for users. This is not a new approach. Simple logo design with only one letter can be found amongst popular apps such as Facebook app or Google app.

Figure 5.10: App's logo in different sizes.

Colour scheme

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

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.

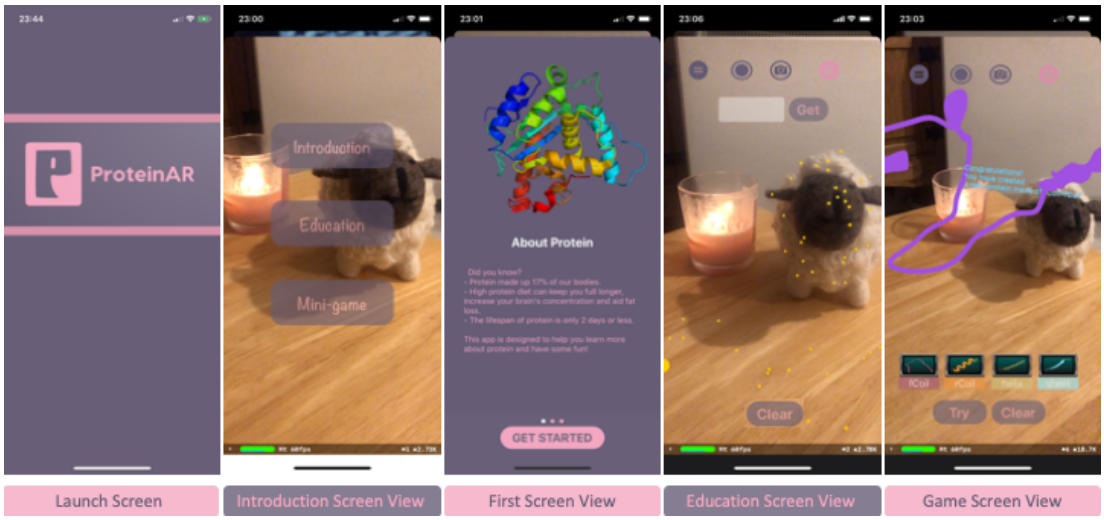

Figure 5.11: UI design of ProteinAR

Figure 5.12: Analogous Colour Scheme

Button design

As for the utility buttons of the Education and Mini-game screen, the main colour scheme is maintained. As the first three buttons generate actions, they are in the same mauves colour

and the exit button is in pink, which creates the slight distinction of the function. The two main function buttons of *Try* and *Clear* also follow the main colour scheme.

In the utility buttons, button icons are used instead of button labels. These icons are familiar to mobile app users, this 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 generates a type of menu display. For example, the *hamburger icon* opens a navigation drawer to more actions; the *kebab icon*, commonly seen on Android operating system, normally opens a smaller inline menu. In this project, the chosen icon for the *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 it symbolises generating more actions but less actions than a *hamburger icon*.

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

(2) Record and Camera button: The record button accepts two types of actions: long press and tap. Long press generates the action of recording the screen and tap ends it. The long press action also changes the colour of the button to red, which is commonly associated with recording. Tap brings it back to its original colour, which symbolises the end of the recording action. As for the camera button, the colour only flashes, implying the act of picture taking has been done.

As mentioned, the buttons in ProteinAR mainly follow the colour scheme of Pinks-Mauves. However, the four buttons to add polypeptide chains are the exceptions. These four buttons

use images as the buttons and to increase usability for users, they are labelled with their names. The designs of the buttons are inherited from Tianshu Xu's 2019 Master project (T. Xu, 2019) and the label colours were chosen to be matched with the colours of the polypeptides. Since this is a mini-game, the colourful elements are chosen for visual appeal and clarity.

Figure 5.14: Polypeptide chains button

5.4 Core Data Design

In ProteinAR, Core Data is simply designed with only one *Entity* called Protein. This *Entity* has two *attributes*: *name* and *location* of type *String*. In the current design of the app, Core Data only stores two information in a Protein *Entity*: the name of downloaded protein and its file path in the *Document Directory*. When the data is called, information stored in the attribute *location* of Core Data simply act as a *String* value to concatenate with the file's name and extension to fetch the file that are stored in the *Document Directory*. Figure 5.15 shows the *Entity* and *attributes* in the data model in Core Data.

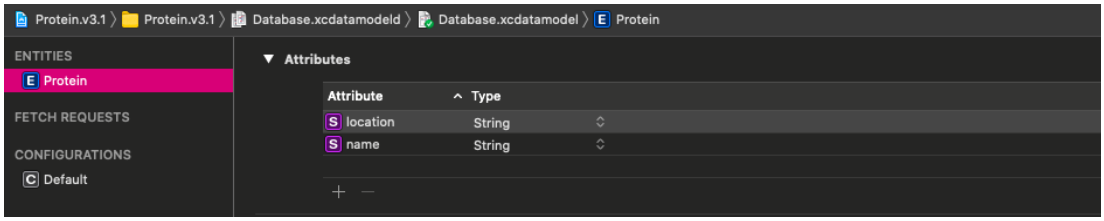

Figure 5.15: Core Data Entity and attributes

The details on implementation of Core Data can be found in Figure 6.7 with explanation in Chapter 6. As explain in Chapter 4, with the current implementations of ProteinAR, integrating Core Data is unnecessary. Document path can be directly called into the displaying function using *File Manager*. The reason Core Data was used is for future work, when more protein data may need to be stored.

5.5 Summary

Based on the proposed solutions, the app was designed to have four screens, with different functions on each to achieve the set goals. The two main screens designed to acquire educational and entertaining purposes are "Education" screen and "Mini-game" screen. Each screen has a

different set of buttons to execute different required functions. The app was designed to use Core Data as the database to store protein information after they were downloaded. As the existing products have not been focusing on the UI design, ProteinAR took UI design into careful consideration in order to bring a better experience to the user.

Chapter 6

Project Implementation

In this chapter, the implementation will be demonstrated with code snippets from the source code of the project. Firstly, the implementation to enable download and visualisation or protein models will be explained step-by-step. Secondly, the method for creating new proteins will be discussed. Last but not least, the chapter will elaborate on the implementation of interactive elements in the app.

6.1 Download and Visualisation of 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

First, Core Data is set up by adding a new *Data model* from the *Core Data* section. In this app, the database has only one *Entity* Protein which has two *attributes*: *name* and *location*, defined in type *String* as in Figure 5.15. Core Data Stack and Core Data Saving Support need to be added to the *App Delegate* if not automatically generated by Xcode as shown in Figure **??**. After that, two subclasses must be created where Protein is defined as a public class in *NSManagedObjectSubclass* and function *fetchRequest* is defined as a public class in the extension of Protein. In these subclasses, the attributes of *name* and *location* are also declared as public variables (refer to Figure 6.2).

6.1.2 Step 2: Download from RCSB PDB and save the PDB file locally

Download PDB file from RCSB PDB

Donwload is the critical function in this process. The function is split into two functions: *getDownloadURL* and *download*.

In the function *getDownloadURL* (Figure 6.3), the URL to the source file is created. After observing how files are downloaded from RCSB, a URL pattern was found. Instead of using the

```swift
    // MARK: - Core Data stack

    lazy var persistentContainer: NSPersistentContainer = {

        let container = NSPersistentContainer(name: "Database")
        container.loadPersistentStores(completionHandler: { (storeDescription, error) in
            if let error = error as NSError? {
                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
        })
        return container
    }()

    // MARK: - Core Data Saving support

    func saveContext () {
        let context = persistentContainer.viewContext
        if context.hasChanges {
            do {
                try context.save()
            } catch {
                let nserror = error as NSError
                fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
            }
        }
    }
```

Figure 6.1: Core Data Stack and Core Data Saving Support

```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

GET method in *action form*, RCSB allows downloading the PDB files directly from a URL. The structure of the URLs are the same for all of the different PDB files, starting with the same path. The file name is the only part that needs to be changed. The file name is set as 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. After creating the URL to the download files, the download function is called with two arguments of *URL* and *parameters* where URL is the to-be constructed URL and parameter is the user's inputted protein name.

```
380     func getDownloadURL() {
381         //Create URL to the source file to download
382         let parameter = textField.text
383         let domain = "https://files.rcsb.org/download/"
384         let fileExt = ".pdb"
385         let fileURL = URL(string: "\(domain)" + parameter! + "\(fileExt)")!
386         print(fileURL)
387         download(fileURL: fileURL, parameter: parameter!)
388     }
```

Figure 6.3: Function getDownloadURL

The code snippet in Figure 6.4 shows the download function that was called by *getDownload-URL*. In this function, the code uses *URLSession* and *downloadTask()* to generate the download task. *URLSession* provides an API for downloading and uploading data to specified URLs. This API helps perform background downloads. In this code, *default* type for *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, *downloadTask* is used as it retrieves data in the form of a file and supports background downloads. It is important to take note of the status code returned by *HTTPURLResponse* as it can allow us to address the different possible errors appropriately. The two most common errors are: the file does not exist (status code 404), and the connection to the server was interrupted (status code 500).

Save the file in Document Directory

When the code performs its download task, the file is stored in a temporary location, as called in the code *temporaryURL* (Figure 6.4, line 402). To save the file locally, the file should be moved to a permanent location in the *Document Directory*, using an *absolute path*. To achieve this, *File Manager* was used, in which the destination folder was allocated to *.documentDirectory*, under *userDomainMask* (Figure 6.4, line 408). Stating the path to the *Document Directory* is not enough to make *an absolute path* as the whole URL to the file must be indicated. Therefore, the format of the file that will be downloaded is specified in Figure 6.4, line 409 as *destinationURL* by adding the path components including the inputted protein's name as the file's name and *.pdb* as the file's extension. For example, if the user domain name is abc123, the proteinID is "6MK1", the path to the downloaded PDB files would be "abc123/Documents/6MK1.pdb". After this absolute path was created, the downloaded file in the temporary location can be moved to the permanent location in *Document Directory* using *FileManager.default.moveItem*.

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

```
390     func download(fileURL:URL, parameter: String){
391         //Use URLSession and downloadTask
392         let sessionConfig = URLSessionConfiguration.default
393         let session = URLSession(configuration: sessionConfig)
394         let request = URLRequest(url: fileURL)
395         let task = session.downloadTask(with: request) { temporaryURL, response, error in
396                 //Get the httpresonse code to make sure file exists
397                 if let statusCode = (response as? HTTPURLResponse)?.statusCode{
398                     print("Successfully downloaded. Status code:\(statusCode)")
399                 }else {
400                     return
401                 }
402                 guard let temporaryURL = temporaryURL, error == nil else {
403                     print(error ?? "Unknown error")
404                     return
405                 }
406                     do {
407                     //download file and save as pre-defined format
408                     let documentsUrl =  try FileManager.default.url(for: .documentDirectory, in: .userDomainMask, appropriateFor: nil, create: false)
409                     let destinationURL = documentsUrl.appendingPathComponent(parameter + ".pdb")
410                     //manage the downloaded files
411                     let manager = FileManager.default
412                     try? manager.removeItem(at: destinationURL)// remove the old one, if there is any
413                     try manager.moveItem(at: temporaryURL, to: destinationURL)// move the new one to destinationURL
414                     print(destinationURL)
415
416                     //Save Files information to Core Data
417                     self.saveContext(name:parameter, location: String(describing: documentsUrl))
418
419                 }
420                 catch let moveError {
421                     print("\(moveError)")
422                 }
423         }
424         task.resume()
425     }
```

Figure 6.4: Function download

same *moveItem()* method. In the source code, this alternative way is disabled. 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 every time the files are 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

Firstly, the *context* is declared by *persistentContainer* and the *proteinManagedObject* is declared and initialised as *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 *name* and *location* are saved into the Protein *Entity* as a *String* type. 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

After the *.pdb* file is downloaded and saved to *Document Directory*, it must be converted to a *.dae* file because ARScene only allows displaying Collada models to its AR Scene. A few solutions were used to solve this problem. One of those is to borrow the 'PDB to Collada conversion' from UCSF Chimera. Since Chimera was written in Python, its scripts could be run in Swift since Python has a C interface API. However, the challenge was that Chimera is not compatible with iOS, so this solution could not be used.

OpenBabel was another solution that was investigated . Unfortunately, OpenBabel is only compatible with Android and MacOS, not iOS and in effect 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.

This, therefore remains an outstanding issue of the app.

6.1.5 Step 5: Fetch and Visualise PDB files

Data saved into the Core Data can be fetched using *NSFetchRequestResult* and can be displayed easily using the attributes assigned in Core Data. Once the PDB file is converted, the resulting Collada file can be displayed using the function shown in Figure 6.7. After some trials, all the models after being converted to Collada files are quite large which cannot fit on the screen and need to be scaled down. Each of these models consist of many nodes, however, *node_1* is always the main node for the whole model. Therefore, in the function (Figure 6.7, line 201 - 202), *node_1* is set as a variable and is scaled down. This assures that in the future, after being ownloaded and converted, the models can always be display completely on screen.

As the app does not yet have a functioning "PDB to Collada' converter, already converted protein files in Collada format are made available to demonstrate the visualisation step.

```
184     //idea of the function that should be able to display the protein if there is a converter
185     func displayProtein(name: String) {
186         let proteinScene = SCNScene(named: proteinManagedObject.location! + "/"  + name + ".dae")
187         if proteinScene == nil {
188             print("Model does not exist")
189             displayText(name: name)
190         } else {
191
192             let cameraNode = SCNNode()
193             cameraNode.camera = SCNCamera()
194             cameraNode.position = SCNVector3(x: 0, y: 0, z: 0)
195             scene.rootNode.addChildNode(cameraNode)
196
197             let node = proteinScene!.rootNode
198             node.scale = SCNVector3(x: 0.0008, y: 0.0008, z: 0.0008)
199
200
201             let node1 = proteinScene?.rootNode.childNode(withName: "node_1", recursively: true)!
202             node1!.scale = SCNVector3(x: 0.0008, y: 0.0008, z: 0.0008)
203
204             let centerConstraint = SCNLookAtConstraint(target: node)
205             cameraNode.constraints = [centerConstraint]
206             secondSceneView.scene = proteinScene!
207         }
208     }
209
```

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

ProteinAR uses ARKit and SceneKit to load the models. For this to happen, models need to be imported. Firstly, a new directory *Scene Catalogue* must be made. In this project, the directory is named "Combinations". Imported models are in *.dae* format, however, to improve loading times for SceneKit, they are converted to *.scn* type. The files are named according to the polypeptide chains that make them in the order that they make them. See Figure 6.8 for more details as well as some examples. This naming convention makes it easy to pass arguments

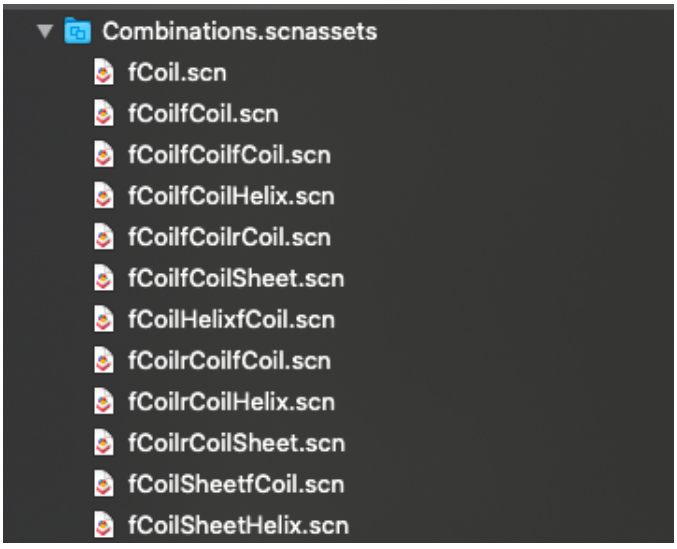

Figure 6.8: Combinations of polypeptide chains stored in a folder

and load models in the upcoming functions. To make the models clearer, Phong shading is used and the colour of each model is assigned randomly.

6.2.2 Step 2: Add polypeptide function

Figure 6.9 shows the function used to add polypeptides to the screen. In this function, the argument is pre-defined as a *String* type and has a name matching the *protein's name*. In *ARKit*, *SCNScene* is used to load the 3D models. Since all the models have the same format (inside the "Combinations.scnassets" directory with ".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. This function also adds a camera node to the screen at the position of (0, 0, 0) as well as fixes model position using *SCNVector3* to ensure that the models will always appear in front of the camera. One problem that users might encounter with using AR technology is that the space is infinite so models may be loaded off-screen. It is therefore important to ensure the position of the loaded models are visible. This function also uses *eulerAngles* with a random *SCNVector3* to ensure that every time new models are loaded they are at the same position but with different orientations so they do not perfectly overlap each other. This makes it clear to user that a new model has been added.

```
// 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

6.2.3 Step 3: Create new protein

To simplify the process, the protein combinations are not generated by the code 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 minor steps.

```
52      var proteinArray = [String]()
53
54      @IBAction func flexCoil(_ sender: UIButton) {
55          addProtein(name: "fCoil")
56          proteinArray.append("fCoil")
57      }
58
59
60      @IBAction func rigCoil(_ sender: UIButton) {
61          addProtein(name: "rCoil")
62          proteinArray.append("rCoil")
63      }
64
65
66      @IBAction func helix(_ sender: UIButton) {
67          addProtein(name: "Helix")
68          proteinArray.append("Helix")
69
70      }
71
72
73      @IBAction func sheet(_ sender: UIButton) {
74          addProtein(name: "Sheet")
75          proteinArray.append("Sheet")
76      }
```

Figure 6.10: Actions of each of the polypeptide button

Clear everything off the screen

Clearing the screen will make the transition to a new model more natural.

```
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 to the screen as nodes (model node, camera node, light node), simply removing all the nodes from *ParentNode()* will clear the screen entirely.

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 is declared in the beginning and every time a button is pressed, a value is added to the array. For example, when the *fCoil button* is pressed, "fCoil" is appended to the array. The values in the array will then be concatenated using *array.joined()* to create the combination name as shown in Figure 6.12. If the combination exits, the model will be displayed on screen. Similar to the *addProtein* function, to ensure the models appear in front of the camera, camera and protein are added at a fixed position using *cameraNode* and *proteinNode*. Since the models generated from PDB file are large and cannot fit on the screen, they are scaled down when loaded.

```
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

Display 3D text

These functions are called inside of the function *createProtein*. If a user-generated combination is valid, together with the model, the 3D text "Congratulations" will be loaded, followed by the names of the polypeptides in order of input. If the combination is invalid, no model will be loaded and instead, only the text "Sorry"! The combination of (*user-pressed-buttons*) cannot be made" will appear. As the function to load text is quite simple and similar to loading models, the code is not shown here. See Appendix A for the full code of this function.

6.3 Interactive elements

6.3.1 Interacting with Protein Models using three gestures

ARKit is a very powerful framework as it enables gesture interaction. To do this, the gestures must be dragged onto the *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*. In ProteinAR, the goal for setting interactions is that the interactions cover the whole screen. This is why the area of enabling gesture is set as *SCNView* while *hitTest* is used to run the gesture.

The *Pinch Gesture* function shown in figure 6.13 uses *SCNVector3* to change the float values of x, y, and z. Using the two fingertips, users can zoom in and out on the models. The other two functions for rotating and panning respectively are similar to pinch gesture, with the defining characteristics of *.state* being initialised by *.changed*. These other two functions will not be

```
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

displayed here. The codes can be found in the Appendix A .

6.3.2 Other interactive elements

Although it is not a requirement of the app, a more interactive display makes for a presentable UI, so, additional gestures and touches were added to the app. While this may be counterintuitive, the additions improve overall user experience.

Gesture Recognizer button

For the *Record* button, the two gestures of *Tap* and *Long press* were added. Among several alternative methods, 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 as an outlet. Then, in the *viewDidLoad()*, the *GestureRecognizer* can be added to the outlet as shown in Figure 6.14. The *GestureRecognizer* function is handled in objective C code (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

When a user finishes reading the guidelines on *Help Screen View* or finishes inputting 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 a user touches any place that is not the *Help Screen View*, the view will be hidden.

For the keyboard, it can usually be set with *textFieldShouldEndEditting* after specifying *TextFieldDelegate* in the class. However, in ProteinAR, since the whole screen is covered by *UIGesture*, this did not work. The solution was to set the *Return* key as a *Done* key in *viewDidLoad* and then use the function of *textFieldShouldReturn* to dismiss the keyboard, as shown in Figure 6.16.

```
424 //---------FUNCTIONS FOR INTERACTION AND DESIGN-------
425  //1. Dismiss keyboard after entering protein's name
426     func textFieldShouldReturn(_ textField: UITextField) -> Bool {
427         textField.resignFirstResponder()
428         return true
429     }
430 //2. Dismiss Help Screen by touching other part of the screen
431     override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
432         let touch: UITouch? = (touches.first!)
433         if touch?.view != helpView{
434             self.helpView.isHidden = true
435             }
436     }
```

Figure 6.16: Others interactive elements

6.4 Summary

Based on the design, coding solutions were implemented to achieve the three main goals: download and visualise protein models, create new protein models, and add more interactive elements. To enable downloading and visualising protein models, the process was divided into five steps of implementation. Although the direction of the five steps was planned, step four (convert PDB file to Collada file) was not successfully implemented, resulting in the incompletion of the function. For future development, this step should be the main focus. The implementation to achieve creating new proteins and adding more interactive elements were successfully conducted. Although there are plenty of rooms for further development, these functions lay a solid foundation for the app.

Chapter 7

Testing and Evaluation

In this chapter, the conducted test results will be reported. These tests include function testing, unit testing, performance testing, and usability testing. The result of each test, as well as the analysis, and limitations of the tests will be provided. Finally, an overall evaluation of the project is summarised in table 7.2.

7.1 Function Testing

7.1.1 Education Screen

When the protein ID is inputted, the model of the protein is displayed. In Figure 7.1 on the left, the protein 6K01 is displayed on the AR screen as 6K01 is a valid protein ID. On the right, when the protein is invalid, the screen shows only 3D text informing the user that protein does not exist.

7.1.2 Mini-game screen

On the Mini-game screen, when a polypeptide chain button is pressed, the model of that chain will be displayed. Similarly, when another polypeptide chain button is pressed, the second model appears on top of the first one as shown in Figure 7.2. If the "Try" button is then pressed and the combination exists, it will be displayed with text showing the names of its elements. If the combination does not exist, only text will appear as shown in Figure 7.3.

7.2 Unit Testing

By running the app, only functions that can be displayed on the screen can be tested. Functions to download the PDB files cannot be tested in this way. Therefore, in this project a unit test was built to test the download function. In Xcode, the *XCTest* is a built-in framework for writing unit tests. *XCTest* asserts that during code execution, certain conditions are satisfied and if not, the errors messages will be shown together with the test failure result. The full code for the unit test can be found in Appendix A. In Figure 7.4, the main part of the code is shown.

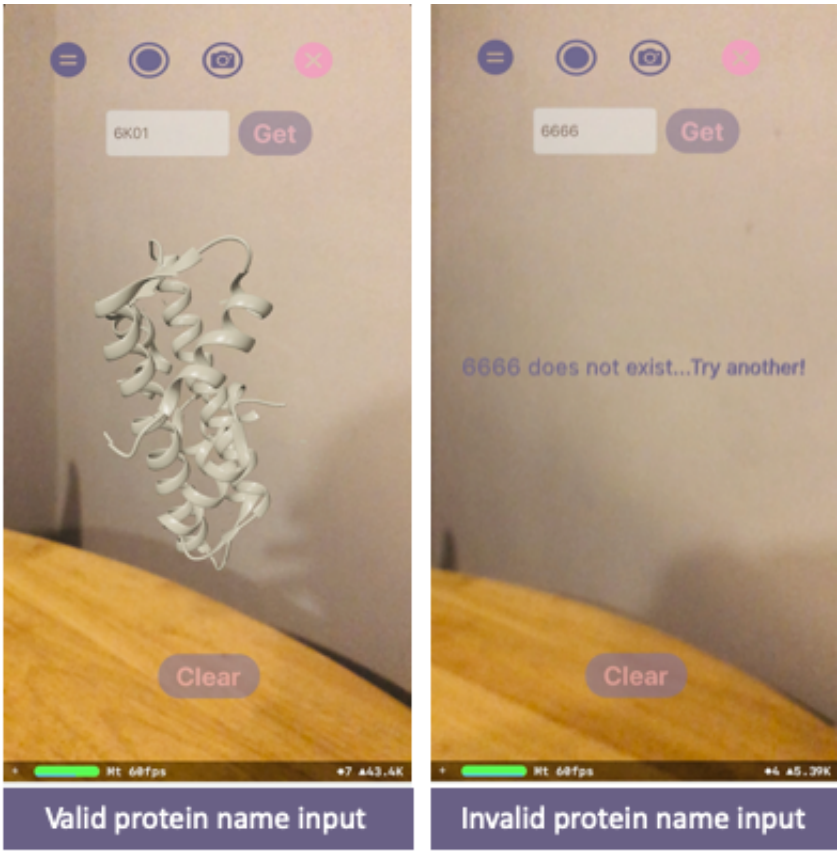

Figure 7.1: Education App Screen

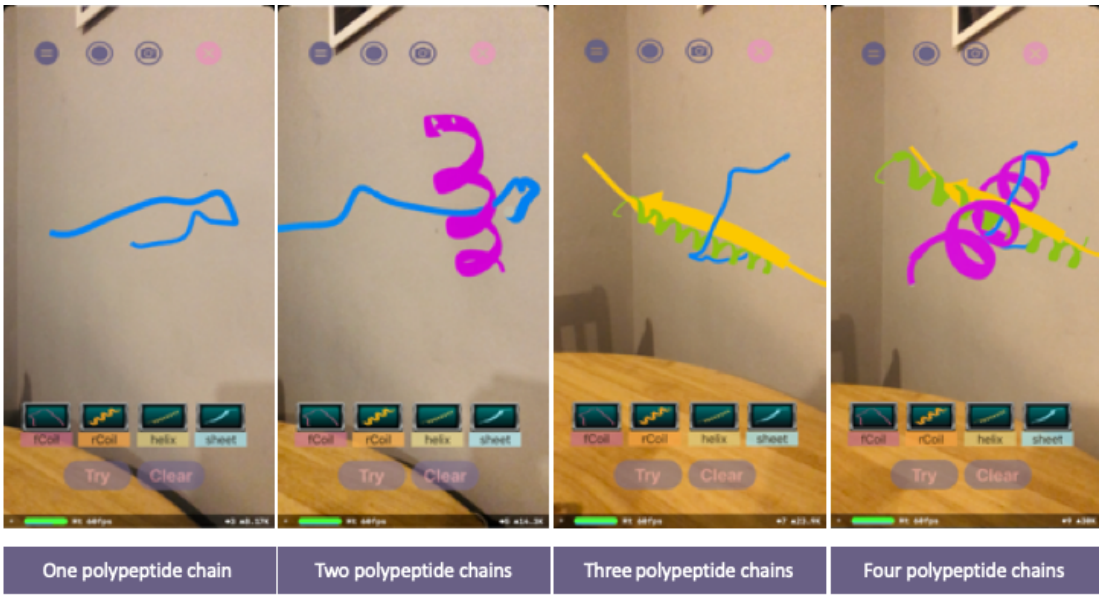

Figure 7.2: Mini-game App Screen (1)

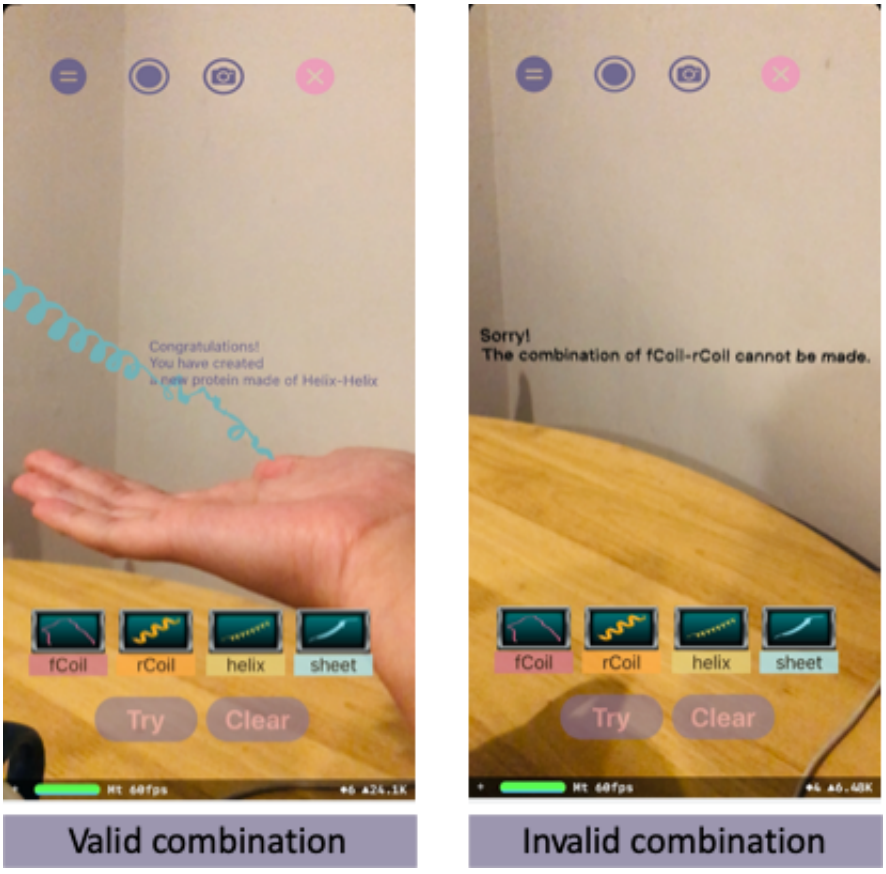

Figure 7.3: Mini-game App Screen (2)

In this test function, the URL is given with a valid protein ID (6K03) and the code checks if the file 6K03.pdb exists in the *Document directory* after the download function completes. The green tick on the function shows that the tested function (download) works.

```
func testDownload() {
    // Use XCTAssert and related functions to verify your tests produce the correct results.
    //1. given
    let fileURL = URL(string: "https://files.rcsb.org/download/6K03.pdb")!
    let parameter = "6K03"
    //2. when
    sut.download(fileURL: fileURL, parameter: parameter)
    //3.then
    let fm = FileManager.default
    do {
        let documentsUrl =  try FileManager.default.url(for: .documentDirectory, in: .userDomainMask,
            appropriateFor: nil, create: false)
        let destinationURL = documentsUrl.appendingPathComponent(parameter + ".pdb")
        try? fm.removeItem(at: destinationURL)// remove the old one, if there is any
        try fm.moveItem(at: destinationURL, to: destinationURL)
        print(destinationURL)
        print("Hello")

        XCTAssert(fm.fileExists(atPath: (String(describing: destinationURL))), "File does not exist")
    } catch {
        print("Error")
    }
}
```

Figure 7.4: Unit Test - Download function

7.3 Performance Testing

Since ProteinAR is an iPhone app, evaluating how the app performs on an iPhone is important. Xcode has a built-in debug navigator to show how the app performs on the device. In this navigator, there are reports to visualise how the application impacts the running of a simulator device. Figure 7.5 shows the impact on iPhone's CPU while the app is running. The CPU percentages fluctuates frequently, though it maintains a range between 80 and 120 percent. The testing device is iPhone X with six cores, bringing the maximum CPU capacity to 600 percent. Since the app performs many tasks at the same time (recognising the real-world's surfaces, putting layers on, downloading from the web, displaying models, etc.,), this is considered acceptable. In Figure 7.5, the percentage used is shown to remain in the green zone.

As for memory, the app has low memory consumption. In future work, when a conversion function is made and the app can display models from downloaded PDB files, memory will not be a problem as the data will be stored in *Document directory*. A visualisation of memory usage is shown in Figure 7.6.

An AR app should have a frame rate of at least 30 FPS to allow the app to run smoothly and save CPU and GPU usage. In ProteinAR, the frame rate is relatively high, at 60 FPS. This creates smooth movement, but also requires substantial GPU usage (Figure 7.7). This might also lead to extra energy usage, negatively affecting the energy impact of the app (Figure 7.8).

Through observation, when the app starts running, the energy impact is already stated in the report as "Very High". The thermal state starts at "Fair" and in only a few minutes changes to "Serious". Sometimes, the thermal state increases to "Critical" if the app is left running. This negatively affects app performance, and drains the device's battery faster than normal.

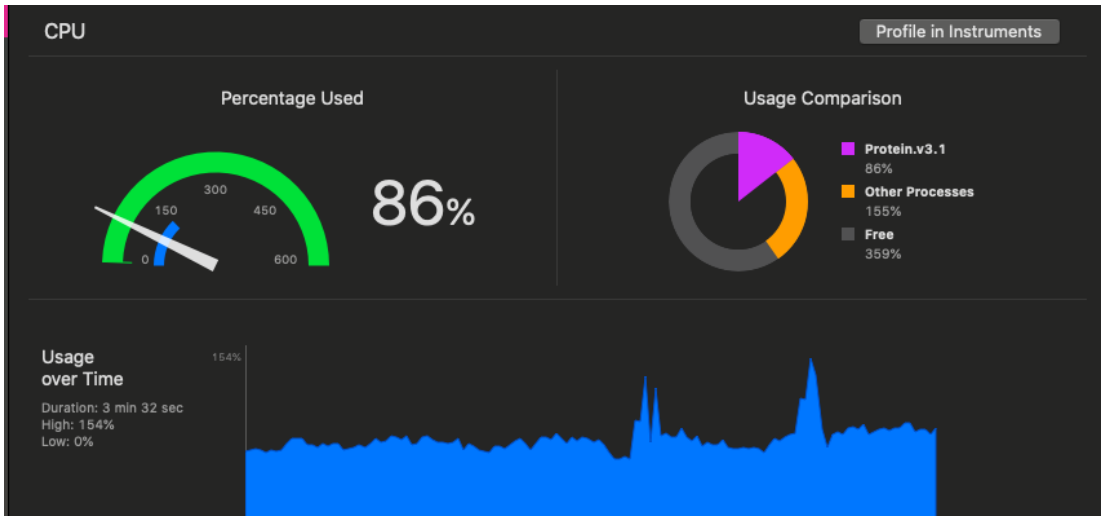

Figure 7.5: CPU usage

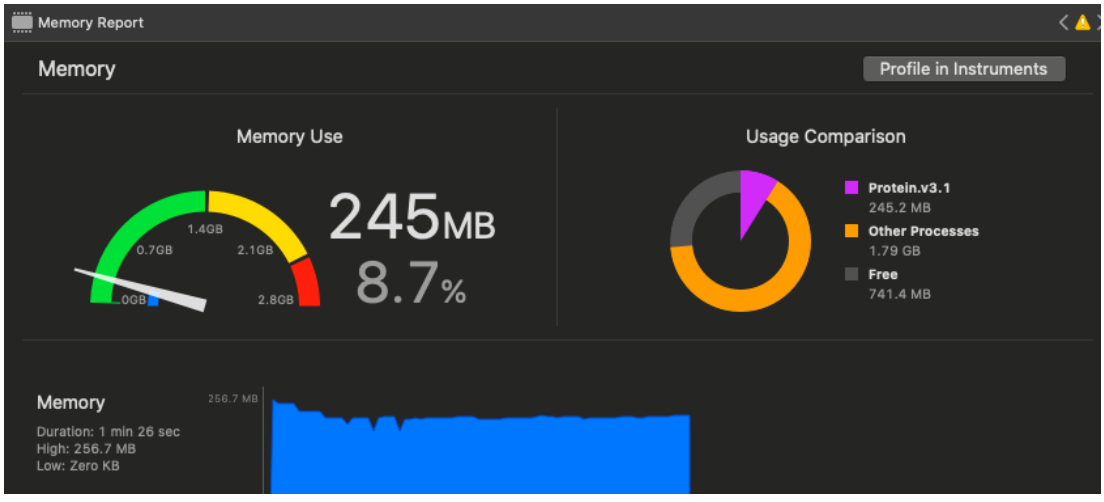

Figure 7.6: Memory usage

Figure 7.7: GPU Usage

Figure 7.8: Energy impact

If the device overheats, loaded models lag, and extra screens (i.e. the "Help Screen") fail to appear when clicked due to the excessive heat generated by the UI elements. The performance evaluation is summarised in table 7.1.

Category	Device impact	Advantages	Disadvantages
CPU/GPU	High	Multi-tasking, smooth transition	Increase energy impact
Memory	Normal	No extra workload on the device	N/A
FPS	High	Smooth display of models and AR layers	Increase CPU and GPU usage
Energy	Very High	N/A	Freeze the app and Drain battery

Table 7.1: Performance Evaluation

To help lower the energy impact, more testing needs to be implemented in future work.

7.4 Application Usability Testing

Usability evaluation is an important evaluation that any system must take before product release. This helps ensure the app meets the requirements of a business and secures customer satisfaction. By doing usability testing, the app can be improved based on user feedback.

In this project, the usability evaluation was conducted based on the post-testing usability questions. There are ten questions, adapted from the SUS (System Usability Scale) questionnaire. This means the questions are asked after users have had experienced using the app. However, the project ran into two problems while conducting usability testing.

- Limitation of testing subjects:

  For users to remotely download and use the app, ProteinAR has to be available on the App store. After an app is submitted to the App store, it moves through a very strict review process and may not be available for a period of time. This would require more work to complete the app since an app with remaining issues would not make it through the review process. Furthermore, uploading an app on the App store categorises it as a commercial product which may go against UCC policy. Therefore, it was not possible for other users to remotely test the app. Alternatively, there are two methods for the app to be tested by other users which require in-person meeting:

  - Users directly use the app on developer's device or downloads from the developer's computer. With the ongoing complications of Covid-19, this was not advisable.
  - User can install the app by downloading the project on Github. However, this requires target users to have access to a MacOS system and an iPhone 6 or onwards with the compatible MacOS and iOS version.

    Since the second method was difficult to conduct, all the tests were carried out in the first method, Due to the previous mentioned difficulty of Covid-19, the number of users who conducted the test was limited to five people.

- Lack of objectivity in questionnaire result:

  For all tests, the users and developers were in the same room. This removed the anonymity of the test introducing bias to the results.

The tests were conducted despite the above limitations, and the questionnaire results are shown in Figure 7.9. Prior to the test, users were provided protein names to start with. Since there are limited buttons on the screen, it did not take much time for users to learn how to navigate and use the app. The element of using Augmented Reality impressed users as it is new and considered "exciting" and "cool", which was a major factor in increasing user motivation for continued use. However, the app is about proteins, which might not be the subject of choice for most users. In effect, users indicated that they would not recommend the app to others, suggesting that the entertainment element does not meet with user expectations.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1. I thought the app was easy to use			1	1	3
2. I found the app unnecessarily complex	1	4			
3. I think I would need the support of a technical person to use the app		4	1		
4. I found the various functions in this app were integrated			1	4	
5. I thought there was too much inconsistency in this app	1	4			
6. I would imagine most people would learn to use this app very quickly			1	3	1
7. I felt very confident using the app			1	3	1
8. I found the interface appealing				3	2
9. I would like to play again				5	
10. I would recommend it to others		2	3		

Figure 7.9: System Usability Scale (SUS) Questionnaire and Feedbacks

The extra functions linking the app to RCSB website or PDB 101 were not intentionally chosen. Users indicated that reading information on another website required undesired effort. This could be replaced by simpler but educational screens or websites. The AR triggered excitement in using the app, and therefore, should be the main focus for future development. The mini-game would be more interesting if the new protein created could be linked to daily life such as "found in human skin, found in sheep wool, etc.".

7.5 Overall Evaluation

Based on the tests conducted, the overall project evaluation is summarised in table 7.2.

Although much future work is desired for the completion of the app, ProteinAR succeeded in creating the first step to bringing protein visualisation to AR. Without the need to print materials, or use extra devices such as VR goggles, ProteinAR can be developed to become a useful tool for students, teachers, and scientists in the study and research of proteins.

GOALS	IMPLEMENTATION	ACHIEVED RESULT	EVALUATION
Download Protein from RCSB PDB	• Download using *downloadTask()* using constructed URL • Save files to Document directory • Assign attributes to Core Data	Based on the result from Unit Testing, these functions work without errors.	**Completed**
Visualise Protein models on AR	• Convert PDB file to Collada files • Visualise by loading 3D models on AR screen using function *displayProtein* (Figure: 6.7)	• Conversion function was not completed • Can visualise predownloaded sample models	**Partially completed**
Create new Protein on AR	• Display individual polypeptide chains with function *addProtein()* (Figure: 6.9) • Display combinations of polypeptide chains according to user input using function *createProtein* (Figure: 6.12)	Displaying individual polypeptide chains and clearing them to display the combination creates a smooth transition and interactive experience for the user.	**Completed**
Interact with Protein models	Use UIGestureRecognizer for • Pinch Gesture (allow zooming in and out) (Figure:6.13) • Rotation Gesture (allow rotating models) (see the Appendix A) • Pan Gesture (allow moving models) (see the Appendix A)	Based on the user testing questionnaire, the use of interactions is satisfactory.	**Completed**
Appealing UI	• Use two different screens for two different purposes (education and mini-game) • Easy to navigate • Analogous colour theme • Interactive elements (buttons, display, model-interactions, create new proteins) (Chapter: 5)	Based on the user testing questionnaire, the UI is easy to use and appealing.	**Completed**
Good performance	• CPU Usage: High • GPU Usage: High • Memory Usage: Normal • Energy Impact: Very High	The thermal state reaches "Serious" after a short period of use, which is not ideal for the app. Further tests are needed to solve this problem.	**Partially completed**

Table 7.2: Overall Evaluation

Chapter 8

Final Conclusion

In this chapter, project summary with achieved results will be concluded, followed by the ideas of direction for future work.

8.1 Conclusion

This project aimed to bring Augmented Reality technology into biological research by visualising protein structures in AR and allow for user interactions. To achieve this, the iOS app ProteinAR was developed to download PDB files from the RCSB Protein Data Bank and display them in AR environment. ProteinAR also allows interactions, in which users can pinch, rotate, move, or create new proteins from polypeptide chains. This app was designed using Xcode and written in Swift using ARKit, Apple's relatively new framework for developing app-based AR technology. Three main goals were set for the project: enabling the display of protein models from inputted protein IDs, enabling the creation of new protein models, and enabling user interactions with these models. To do this, the process was divided into minor steps, in which minor functions were written to achieve the goals. Additionally, design principles were integrated with extra functions to ensure the usability of the app. ProteinAR succeeded in downloading models from RCSB PDB. Any models that were downloaded and converted (using UCSF Chimera) can be displayed on the AR screen. ProteinAR also made it possible for users to combine polypeptide chains into new protein structures. Whether the models are downloaded or created, users can interact with the protein structure to learn more about them. Compared to existing products on visualising protein models in AR, ProteinAR allows more user interactions, as manipulation of protein structures is enabled, and the function to create new protein models from polypeptide chains is a predominant feature.

However, the project has an outstanding issue that needs to be addressed before the app can reach completion. In displaying the protein models downloaded from RCSB PDB, a conversion function from PDB to Collada file is necessary as the ARKit only allows display of the the Collada file type. Due to time constraint of the project, this remains undeveloped, thus ProteinAR can only display protein models from previously downloaded and converted models. Nevertheless, ProteinAR proved that retrieving and displaying data from the RCSB

PDB is achievable. The advancement of AR technology signals the potential for ProteinAR to be further developed into an invaluable tool for academics of biology, and for anyone with a curious itch to see the unseeable.

8.2 Future Work

Given the limitations of the project, there is plenty of rooms for improvement with ProteinAR. Furthermore, Augmented Reality technology is relatively new and Apple has been acquiring new companies specialising in AR to update the ARKit framework rapidly, creating further possibilities for the development of the app in the future.

First and foremost, the **protein real-time visualisation** remains unfinished due to it missing a function to convert the PDB file type to the Collada file type. With this function completed, ProteinAR could become extremely useful in biology class as it displays any protein structure in real time. This should be the main focus of future work.

Secondly, **new protein creation** can be improved in the following directions:

- Currently, the combinations of polypeptide chains are pre-loaded into a folder. In the future, if these combinations can be generated in real time, using a server such as I-TASSER, more combinations can be created not limited only to tertiary but quaternary structures.
- The newly created protein, if pre-existing, should link to some information such as its parameter, its function, etc. This can be achieved using POST and GET REQUEST to the available source of information.
- The newly created protein should be exportable as a PDB file. This way, it could be used for research purposes.

Thirdly, more **interactive elements** can be added. ProteinAR can integrate Machine Learning and CoreML to control ARKit. There are many ways to implement this. One popular way is to combine image classification and AR to create new experiences by experimenting with hand gesture recognition. Photos of hand poses can be taken then used in training models such as TensorFlow, Keras, Custom Vision. Xcode also has a training interface. The models can be trained so that users can use hand poses to control the protein models. Additionally, the protein models could be made more realistic by allowing the user to bend and twist the nodes of the structure. Currently, ProteinAR only allow users to pinch, rotate and pan the models.

Last but not least, **more functions** can be integrated into the app. Currently, the menu function only links the app to the RCSB home page, with no specific information. These functions can be further customised to be more appropriated and interesting.

Bibliography

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

Bacca, J., Baldiris, S., Fabregat, R., Graf, S., & Kinshuk. (2015). Augmented reality trends in education: A systematic review of research and applications. *Education Technology & Society*, 17.

Behmke, D., Kerven, D., Lutz, R., Paredes, J., Pennington, R., Brannock, E., Deiters, M., Rose, J., & Stevens, K. (2018). Augmented reality chemistry: Transforming 2-d molecular representations into interactive 3-d structures. *Proceedings of the Interdisciplinary STEM Teaching and Learning Conference*, *2*(1).

Brown, T. M., & Gabbard, J. L. (2015, September). Interactive learning methods: Leveraging persoalized learning and augmented reality, In *2015 international conference on interactive collaborative learning (ICL)*. 2015 International Conference on Interactive Collaborative Learning (ICL).

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.

Chamba-Eras, L., & Aguilar, J. (2017). Augmented reality in a smart classroom—case study: SaCI [Conference Name: IEEE Revista Iberoamericana de Tecnologias del Aprendizaje]. *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, *12*(4), 165–172.

Core data programming guide: What is core data? (n.d.). Retrieved September 17, 2020, from https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/index.html

Diegmann, P., Schmidt-Kraepelin, M., Eynden, S., & Basten, D. (2015). Benefits of augmented reality in educational environments - a systematic literature review, 16.

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.

Ewais, A., & Troyer, O. D. (2019). A usability and acceptance evaluation of the use of augmented reality for learning atoms and molecules reaction by primary school female students in palestine [Publisher: SAGE Publications Inc]. *Journal of Educational Computing Research*, *57*(7), 1643–1670.

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.

Huang, T.-C., Chen, C.-C., & Chou, Y.-W. (2016). Animating eco-education: To see, feel, and discover in an augmented reality-based experiential learning environment. *Computers & Education*, *96*, 72–82.

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.

Martín-Gutiérrez, J., Fabiani, P., Benesova, W., Meneses, M. D., & Mora, C. E. (2015). Augmented reality to promote collaborative and autonomous learning in higher education. *Computers in Human Behavior*, *51*, 752–761.

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.

Moro, C., Štromberga, Z., Raikos, A., & Stirling, A. (2017). The effectiveness of virtual and augmented reality in health sciences and medical anatomy [_eprint: https://anatomypubs.onlinelibrary.wiley *Anatomical Sciences Education*, *10*(6), 549–559.

Nickels, S., Sminia, H., Mueller, S. C., Kools, B., Dehof, A. K., Lenhof, H.-P., & Hildebrandt, A. (2012, July). ProteinScanAR - an augmented reality web application for high school education in biomolecular life sciences [ISSN: 2375-0138], In *2012 16th international conference on information visualisation*. 2012 16th International Conference on Information Visualisation. ISSN: 2375-0138.

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

Silva, R., Oliveira, J. C., & Giraldi, G. A. (2003). Introduction to augmented reality, 11.

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

Sung, R.-J., Wilson, A. T., Lo, S. M., Crowl, L. M., Nardi, J., St. Clair, K., & Liu, J. M. (2020). BiochemAR: An augmented reality educational tool for teaching macromolecular structure and function. *Journal of Chemical Education*, *97*(1), 147–153.

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.

What is a user interface? — definition — every interaction explain [Every interaction]. (n.d.). Retrieved September 11, 2020, from https://www.everyinteraction.com/definition/user-interface/

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.

Xcode 12 - apple developer. (n.d.). Retrieved September 8, 2020, from https://developer.apple.com/xcode/

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, October). *An interactive protein design game: Pocket peptides* (Doctoral dissertation). University College Cork.

Appendix A

Code snippets

```
344     //7. Function to display text
345     //When successfully create a new protein
346     func displayText1(){
347         let newElements = proteinArray.joined(separator: "-")
348         let text = SCNText(string: "Congratulations!\nYou have created \na new protein made of \(newElements)", extrusionDepth: 2)
349         let material = SCNMaterial()
350         material.diffuse.contents = UIColor(red: 0.4, green: 0.36, blue: 0.46, alpha: 1)
351         text.materials = [material]
352         let node = SCNNode()
353         node.position = SCNVector3(x: -0.005, y: -0.005, z: -0.01)
354         node.scale = SCNVector3(0.0005, 0.0005, 0.0005)
355         node.geometry = text
356         node.name = "shape"
357         thirdSceneView.scene.rootNode.addChildNode(node)
358
359     }
360      //When unsuccessfully create a new protein
361     func displayText2(){
362           let newElements = proteinArray.joined(separator: "-")
363           let text = SCNText(string: "Sorry!\nThe combination of \(newElements) cannot be made.", extrusionDepth: 2)
364           let material = SCNMaterial()
365           material.diffuse.contents = UIColor.black
366           text.materials = [material]
367           let node = SCNNode()
368           node.position = SCNVector3(x: -0.007,y: -0.005, z: -0.01)
369           node.scale = SCNVector3(0.0005, 0.0005, 0.0005)
370           node.geometry = text
371           node.name = "shape2"
372           thirdSceneView.scene.rootNode.addChildNode(node)
373       }
```

Figure A.1: Function to display 3D Text

```swift
// Rotation Gesture
@IBAction func rotationGesture(_ sender: UIRotationGestureRecognizer) {
    if sender.state == .changed {
            let areaTouched = sender.view as? SCNView
            let location = sender.location(in: areaTouched)

            let hitTestResults = thirdSceneView.hitTest(location, options: nil)

            if let hitTest = hitTestResults.first {
                let plane = hitTest.node
                newAngleZ = Float(-sender.rotation)
                newAngleZ += currentAngleZ
                plane.eulerAngles.z = newAngleZ
            }
        } else if sender.state == .ended {
                currentAngleZ = newAngleZ
        }
}
```

Figure A.2: Rotation Gesture

```swift
//Pan Gesture
@IBAction func panGesture(_ sender: UIPanGestureRecognizer) {
    let areaPanned = sender.view as? SCNView
    let location = sender.location(in: areaPanned)
    let hitTestResults = areaPanned?.hitTest(location, options: nil)
    print("\(String(describing: areaPanned))")
   if let hitTest = hitTestResults?.first {
       if let plane = hitTest.node.parent {
           if sender.state == .changed {
               let translate = sender.translation(in: areaPanned)
               plane.localTranslate(by: SCNVector3(translate.x/10000,-translate.y/10000,0.0))
            }
       }
   }
}
```

Figure A.3: Pan Gesture

```swift
import XCTest                                                                    testDownload()
@testable import Protein_v3_1

class Protein_v3_1Tests: XCTestCase {
    var sut: EduViewController!
    override func setUp(){
        // Put setup code here. This method is called before the invocation of each test method in the class.
        super.setUp()
        sut = EduViewController()
    }
    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each test method in the class.
        sut = nil
        super.tearDown()
    }

    func testDownload() {
        // Use XCTAssert and related functions to verify your tests produce the correct results.
        //1. given
        let fileURL = URL(string: "https://files.rcsb.org/download/6K03.pdb")!
        let parameter = "6K03"
        //2. when
        sut.download(fileURL: fileURL, parameter: parameter)
        //3.then
        let fm = FileManager.default
        do {
            let documentsUrl =  try FileManager.default.url(for: .documentDirectory, in: .userDomainMask,
                appropriateFor: nil, create: false)
            let destinationURL = documentsUrl.appendingPathComponent(parameter + ".pdb")
            try? fm.removeItem(at: destinationURL)// remove the old one, if there is any
            try fm.moveItem(at: destinationURL, to: destinationURL)
            print(destinationURL)
            print("Hello")

            XCTAssert(fm.fileExists(atPath: (String(describing: destinationURL))), "File does not exist")
        } catch {
            print("Error")
        }
    }
}
```

Figure A.4: Full test script for unit test