Development and Evaluation of an Augmented Reality Application for assisted CPR Training

Henry Donnelly

Student number: N00220105

Supervisor: Joachim Pietsch

Second reader: Michael McAndrew

Abstract

QCPR stands for Quality Cardiopulmonary Resuscitation. It builds on basic CPR training by implementing technology in the manikin to track the quality of the users' compressions. “QuestQCPR” is my attempt at improving CPR training environments by instead using feedback through augmented reality to assist with compressions. It is accessible to users without a physical CPR manikin to practice along with being available for at home learning.

Functionalities of the app include a tutorial, hand gesture-controlled menus, a metronome, video assistance, real-time progression bar, different timed training sessions, and manikin alignment. As the app was developed using the 2026 Meta’s XR SDK, it is only available exclusively for Meta Quest 3.

In addition to the application, a companion website was developed to provide users with information on how to use or download the application to their device.

Acknowledgements

I would like to take a moment to earnestly thank my project supervisor, Joachim Pietsch for his guidance and insight throughout the course of the app's development.

I would also like to thank everyone who participated in the user testing in addition to submitting valuable feedback in the usability forms.

Lastly, I thank my family for putting up with me these past four years as well as pushing me to give my best effort in this final year.

Plagiarism Notice

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

DECLARATION:

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student: Henry Donnelly

Signed: Henry Donnelly

Failure to complete and submit this form may lead to an investigation into your work.

Table of Contents

Introduction and Project Context

Project Aim & Objectives

The original aim of this project was to improve existing CPR training classes with a way to provide live feedback through augmented reality, which later required a strong emphasis on user testing to ensure first-time users could learn how to navigate the application as well as performing CPR.

However, after some time developing the application in Unity, I realized I could remove the need for a physical CPR manikin and expand the user demographic to include those interested in learning from home.

Another objective was to explore how multimedia learning techniques could be integrated into the AR application to improve long term memory of core CPR concepts and practical performance. This involved combining visual and audio aids, in addition to interactive elements to create a more engaging training environment

Success Criteria & Scope

I judged the final product based on how effective the app was at teaching correct CPR principles to both experienced and inexperienced users. I had to ensure that the app had different methods of learning. I did so by researching user learning articles (Richard E. Mayer, 2001) which confirmed that people learn better through effective use of both video and audio channels, which I will discuss later.

Research and Background

Literature Review

Multimedia Learning Theory

This was a study published in 2001 which demonstrates how audio and visuals can be used together to enable positive learning outcomes. Another aspect of the publication is the two kinds of active learning which are hands on learning (behavioural) and internalizing information (cognitive). Lastly, I used both transfer (focus on memory) and retention tests (applying learned information to new scenarios) when structuring the user's feedback forms.

Figure 1
Model of Multimedia Learning Theory

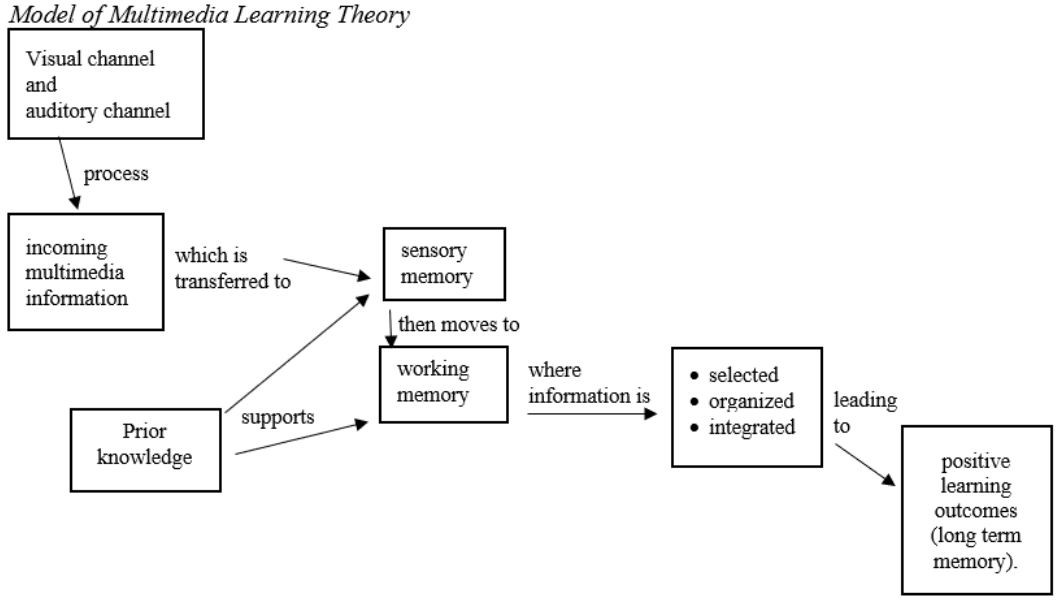


Figure 1. Model of multimedia Learning Theory

This image on MMLT explains how audio and visual channels lead to sensory memory (initial learning) and working memory (analysing learned information) when these ideas are reinforced, that leads to the strongest likelihood of retaining long term memory of the studied information.

Augmented Reality in Education

What is augmented reality?

Augmented reality is a variation of virtual reality (Azuma, 1997). In contrast to a virtual environment where users can only see a synthetic world, augmented reality instead sees the real space around them, with virtual objects overlayed on their surroundings.

A short history on augmented reality

Augmented reality was first introduced in the 1990s for the training of pilots. “NASA designed a compound synthetic vision system in 1990 for the X-38 spacecraft. This system works on AR applications that provide navigational support during the test flight” (Zulfiqar, 2023)

In 2014, Google presented its Google Glass devices. In 2016, Microsoft released its AR wearable device named Holo-lens. And now (as of September 30, 2025) Meta has introduced their new “Meta glasses”. These examples show how augmented reality, while expensive, has become increasingly commercialized in recent years.

Augmented reality has many benefits besides pilot training. The main benefit being for entertainment like interactive games or movies but include others such as sciences and healthcare. With examples including 3D anatomy models for visualization or surgery planning (Zhang, 2025).

Enhancing learning

Publications like (Dunleavy, 2009) and (Khan, 2019) both highlight how AR has the capacity for “enhancing student learning” when supplemented with classroom-based scenarios.

Khan (2019) argues that AR improves knowledge retention by enabling interactive learning, where learners actively participate in the learning process rather than passively receiving information.

These articles support my theory in using QuestQCPR for improving learning in taught CPR training courses.

Cognitive Load

Cognitive Load Theory (CLT) typically divides mental effort into three categories:

1. Intrinsic load, which is associated with the complexity of the content itself
2. Extraneous load, which is caused by how information is presented or structured
3. Germane load, which relates to the mental effort used for learning and understanding

An important way of analysing user experience is by evaluating the "cognitive load" that the app requires. Essentially, this is how much mental effort is required to use the app. There are many ways to test this, such as heart rate, fatigue, pupil dilation, and brain activity, time-to-complete durations for tasks, and questionnaires.

Technical Research

Today's Market for AR Headsets

From 2013 to 2021, Publications related to augmented reality and virtual reality technologies, along with citation counts, have almost doubled in number (Wenli Shang, 2025). This growing trend showcases the increase in popularity for both technologies. The main market leaders in the VR headset industry are currently the Meta Quest 3, Sony PSVR, Apple Vision Pro, HTC Vive and Pico 4, each having their own strengths and weaknesses.

Before building the application, I conducted a review of all available headsets to decide which one offered the most well-supported software development kits while considering other factors like ease of use, cost-effectiveness, pre-built features etc. The SDK itself was determined based on how much community support it had, documentation quality, compatibility with Unity, and support for mixed reality (MR) functionalities such as passthrough and hand tracking.

I omitted the PSVR from the list considering it doesn't offer mixed reality and is mainly only considered for PlayStation games. While the Pico 4 and HTC Vive had potential, I was very unfamiliar with both and wasn't sure how much community support they offered. This left just the Apple Vision Pro and the Meta Quest 3/Pro.

I decided on the Meta Quest over the Apple Vision Pro primarily because I found that many tutorials were tailored towards using Meta as well as Unity for MR app development. I also found that Meta forums like Reddit and YouTube comments showed much more activity over the Apple Vision Pro.

As of April 2026, the Meta Quest 3 is widely regarded as one of the most cost-efficient mixed reality headsets with an average price of 600 euros. I found that the headset supported all the functional requirements needed for developing an MR-assisted CPR app.

The Meta Quest was originally known as Oculus, before being bought by Meta in 2014 for two billion dollars. Another factor which determined my decision was that I had previously used an Oculus headset before.

Meta XR SDK

Introduction

The vast majority of the research phase (as well as during app development) consisted of pouring through the Meta SDK in Unity and understanding how each feature worked. In terms of developing the app, utilizing the Meta SDK was much more important than scripting with Unity C#. Key features in the All-in-one SDK include Meta XR Core SDK, Meta XR Audio, Meta XR Haptics, Meta XR Interaction SDK Essentials, and Meta XR Voice SDK.

Passthrough

Passthrough is a feature of virtual reality which allows users to see their real-world surroundings through the camera on their headset. This is also what makes my app augmented reality, as the user sees their surroundings, with digital screens overlayed.

This was my first time using the Meta “building blocks” which allow developers to streamline the setup of an app environment. I selected the passthrough block which instantly added visibility to my surroundings when starting the app.

Figure 2. Building blocks menu in Unity

Figure 3. Showcasing Unity in VR without passthrough

Sample Scenes

As someone who hasn't previously used Unity or a Meta Quest 3, learning from the pre-built assets in sample scenes was crucial in grasping how to use both. The Meta XR Interactions SDK features a sample scene which provides assets for utilizing the Meta Quest poke function to interact with objects in augmented reality.

Figure 4. Sample scene snapshot

During the early phases of the app, I used the vertical dark menu as a placeholder for the current menu. This featured the functionality for where

the user would select items such as “start CPR session” or “play tutorial video”.

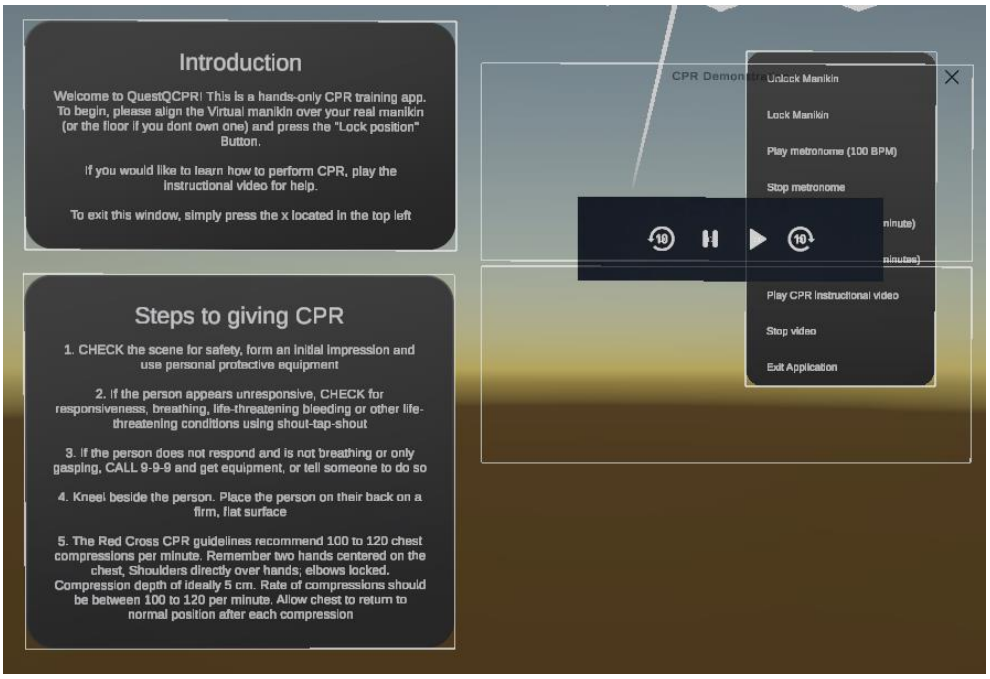


Figure 5. Early testing phase of app development (old menu visible on the right)

Poke Interaction

From the menu (and previous research), I also learned how to deconstruct and use the poke interactor class, which is now a core part of QuestQCPR. To use the class, you need an object (i.e. a button) and hand tracking (discussed later). The object requires the following components: a surface, clipped plane surface (define button poke area), Interactable Unity Event Wrapper (when the button is “poked”, play an event), and a poke interactable. This will make it so that when you poke the button, a scripted event will occur.

The areas this class is used in include compression tracking, interacting with the new menu, closing pop-ups, interacting with the tutorial screen, and lastly for using the tutorial video controls.

Figure 6. Early iteration showcasing the small (green) poke interactable button on manikin

Hand Tracking

While the poke interactable is important for object events, there needs to be a trigger for pressing the button in the first place. This is where another example of using the building blocks appears. I added the interactions rig block which adds a virtual overlay on top of your hand's which then triggers poke interactable objects. The rig creates an overlay; however, it automatically detects the user's hand and changes to fit differing sizes. It acts as a virtual hand skeleton, made up of objects for each finger segment (phalanges), palm, and wrists.

Markerless Tracking

There are two types of tracking, marker and markerless. Using marker tracking consists of using a pinpoint (i.e. a QR code) to anchor the virtual content, whereas markerless (in my case) requires the user to align the manikin and lock the position manually. I chose this method as while it requires precise alignment of the virtual manikin for performing compressions, it removes the need for a physical manikin which in turn allows for at home learning. It should be noted that while possible without, a manikin is preferred for feedback on compressions and getting more realistic resistance from compression depth.

Requirements Analysis

Requirements Gathering

User Needs Analysis

When considering what functionality the app would require, I decided to create a user interest form which provided some insight into what users would like out of the app. The form was distributed to both the testers as well as the public, being available to access from the companion website. Ten users answered the survey. Below are some questions I asked with a scale of 1-10, with 10 being very important and 1 being unimportant.

Figure 7. Overlay effectiveness question

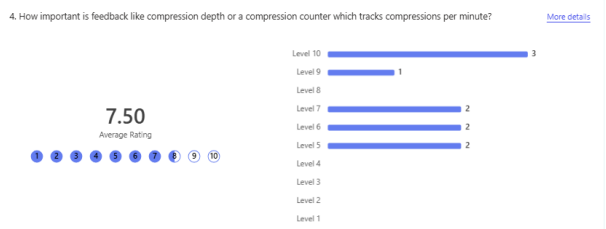


Figure 8. Compression tracker question

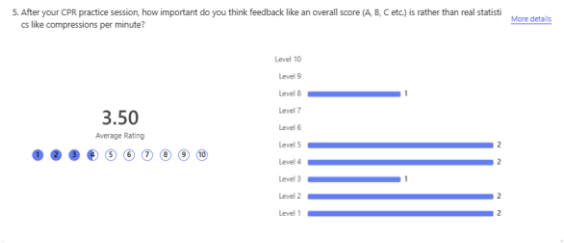


Figure 9. Compression grade question

From these results, we can see that users were more receptive to practical elements like digital screens and a compression counter over a motivational element like a graded ranking system (The grading question received a 3.5/10, while the compression counter received a 7.5/10). As a result of this, I instead opted to create a results pop-up with informational feedback based on the final compression count.

Review of Existing Systems (CPR Apps)

I studied both similar college GitHub projects on CPR training using VR as well as more general first-aid training sites to better understand what a final product should look like.

CPR Information Collection

In order to develop a first-aid CPR training app, I first needed to know the steps for performing them to ensure users aren't taught incorrect information. Thankfully, I had already previously taken a training class for CPR which meant I only needed the more minute details of training such as exact compression speed, depth, duration, types of hand placement etc.

From researching trusted sites like the red cross and irishheart, I researched the key details and included this in my app's tutorial. The specifics include a compression rate of 100-120bpm, at a depth of 5cm, for as long as it takes help to arrive. Hand placement should be directly center of the lower sternum, with hand positioning being one hand over the other, interlocking fingers or resting atop the lower hand (the latter method is preferred as interlocking fingers can cause the hand tracking to not work effectively).

Requirements Modelling

Functional Requirements

The following is a checklist of functional requirements listed in order of importance.

1. Must allow virtual manikin to be aligned custom to the user's preference
2. Must track hand movements and interactions with objects
3. Must calculate compression count as well as rate per minute
4. Must feature a menu for accessing other parts of the app
5. Must offer assistance through visual and audio guides
6. Must feature exit button to close app

Non-Functional Requirements

The following is a checklist of non-functional requirements listed in order of importance.

1. Should be intuitive and easy to use for new users (low cognitive load)
2. Hand tracking should be accurate to avoid errors
3. The app should feature additional functionality in addition to core features
4. The app should be usable without the need for a physical manikin
5. The application shall be exclusive to the Meta Quest 3

System Constraints

While I found that there were no performance issues with the Meta Quest, there was a more general constraint absence of physical haptic feedback with augmented reality. While the application simulates CPR practice visually, it cannot fully replicate the resistance of compressing a physical manikin.

Design

Understanding Unity

Before explaining how the project is structured, I will briefly explain how Unity scenes work.

Figure 10. Unity scene virtual manikin in view

The center image is called the scene view. This is where you can rotate game objects, change sizes and align them (since the manikin object is repositioned to the center of the headsets starting coordinates, it doesn't matter where it begins).

The hierarchy window “SampleScene” is where you can create new game objects and adjust existing ones into child relationships of other objects. Greyed out text represents inactive objects (usually activatable by triggers like button presses).

Below the scene view, you can see the project view. This is where all the files, like assets and scripts, are stored. With the installed Meta SDK, the prebuilt assets and scenes are stored in their sample scene folders.

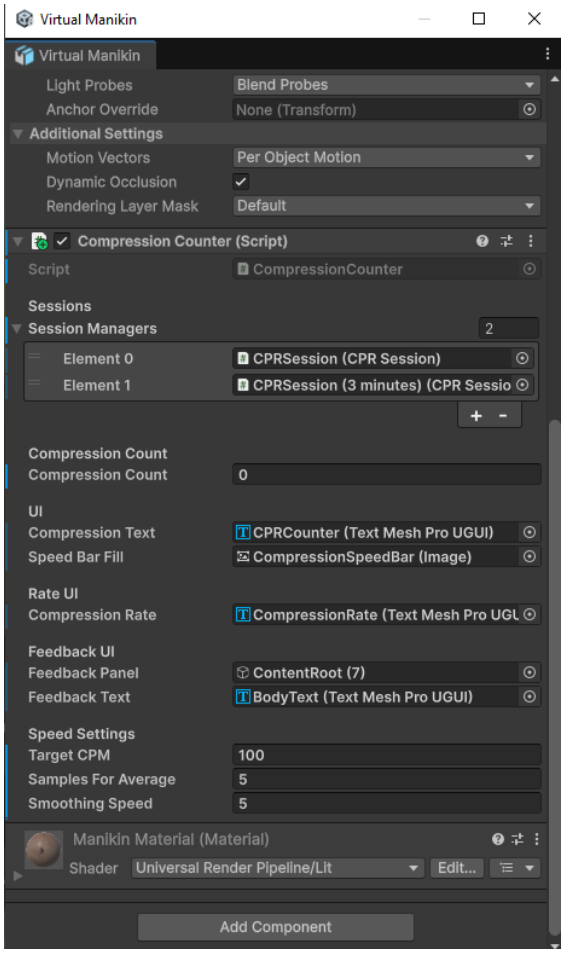


Figure 11. Inspector window showcasing the Virtual manikin with counter script

The inspector window is where users can interact with selected objects by adding components, adjusting size, position, and rotation values, as well as configuring scripts and properties that define how the object behaves within the scene. This inspector shows the compression counter which is a script with inputs for other variables and scripts like CPRSession, which tracks the duration of the 1- and 3-minute sessions.

Interface Design

Virtual Manikin model

For the most part, AI was not involved with the development of the app. However, when trying to find (.fbx) manikin models, I could only find expensive models which had designs I wasn't looking for (for example full body models or too detailed). Fortunately, I found a free model available for download from Meshy, an AI prompt site where you tell it what you want to render. I was able to find another user's downloadable model (which while free to download, required a paid subscription to access).

General interface design inspiration

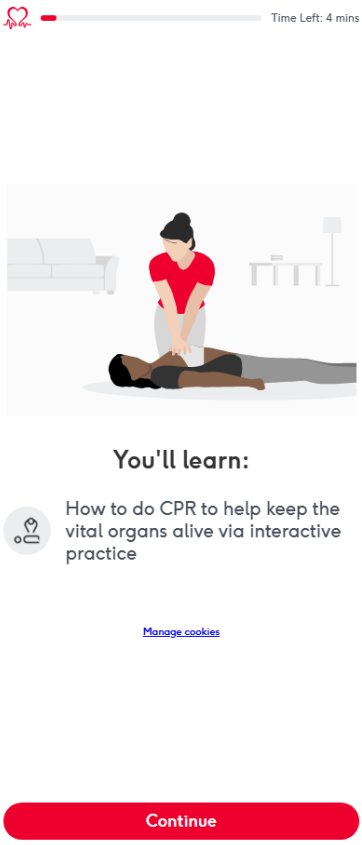


Figure 12. RevivR tutorial screen

I took most of my design inspiration for the app from RevivR, another cpr training app (built for phone compression assistance). I used a white palette with red secondary colours to highlight buttons for my tutorial.

As for my left-hand gesture-activated menu, I decided to keep the white background and instead used grey for the secondary colours. The reason for this change to grey was for the fact that the menu has a colour toggle between dark/light mode.

Figure 13. New menu screen (dark mode)

Lastly, I redesigned the progress bar for counting the compression rate to appear less pixelated and fully fill the containing white box. I also chose a tamer, more forest green to appear less distracting to the user.

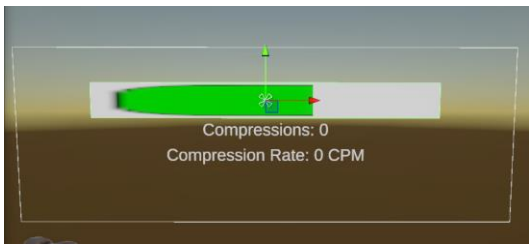


Figure 14. Old progress bar

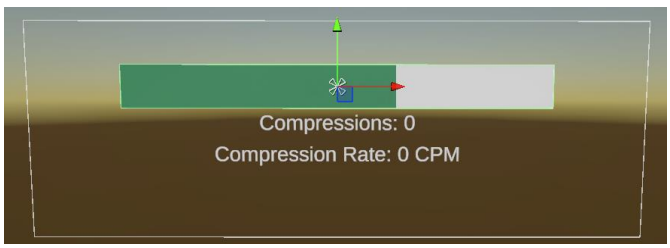


Figure 15. New progress bar

Process Design

User Flow Design

Upon starting the application, the user is greeted by the tutorial menu, where they can read or skip to move on to the next step. After the tutorial, the user can use their left-hand to gesture the CPR menu to appear in front of them, where they can access the main content of the app. Here they can choose whether to watch a training video, start a metronome to help with compression timing, or move straight to aligning the manikin and beginning CPR. After locking the manikin in position and the user is ready to perform CPR, they can press the CPR submenu button and select a 1 minute or 3 minute session. After completing this session, they will be greeted with a results panel based on how many compressions they did within the timeframe. Lastly, they can either choose to exit the app or improve their practice.

Interaction Logic

The gesture menu with the left hand is another asset available with the Meta XR SDK, which I modified to show the relevant submenus. There is a pre-existing Meta Quest gesture for bringing up the settings page for the device; however, this asset replaces that when imported. It also requires the Interactions building block to be imported for hand tracking.

The logic for the compression tracking consists of a poke interactable button on the virtual manikin in the center of the lower sternum, which has a Unity C# script for incrementing the counter on value changed (button pressed).

Also has functionality for updating the text for the compression count as well as storing time between each compression interval to calculate compressions per minute in another script.

```
public void AddCompression()
{
    bool anySessionActive = false;

    foreach (CPRSession session in sessionManagers)
    {
        if (session != null && session.IsSessionActive())
        {
            anySessionActive = true;
            break;
        }
    }

    if (!anySessionActive)
        return;

    compressionCount++;
    UpdateText();

    float now = Time.time;

    if (lastCompressionTime > 0f)
    {
        float interval = now - lastCompressionTime;

        compressionIntervals.Add(interval);

        if (compressionIntervals.Count > samplesForAverage)
            compressionIntervals.RemoveAt(0);
    }

    lastCompressionTime = now;
}
```

Figure 16. Add compression logic

Website Design

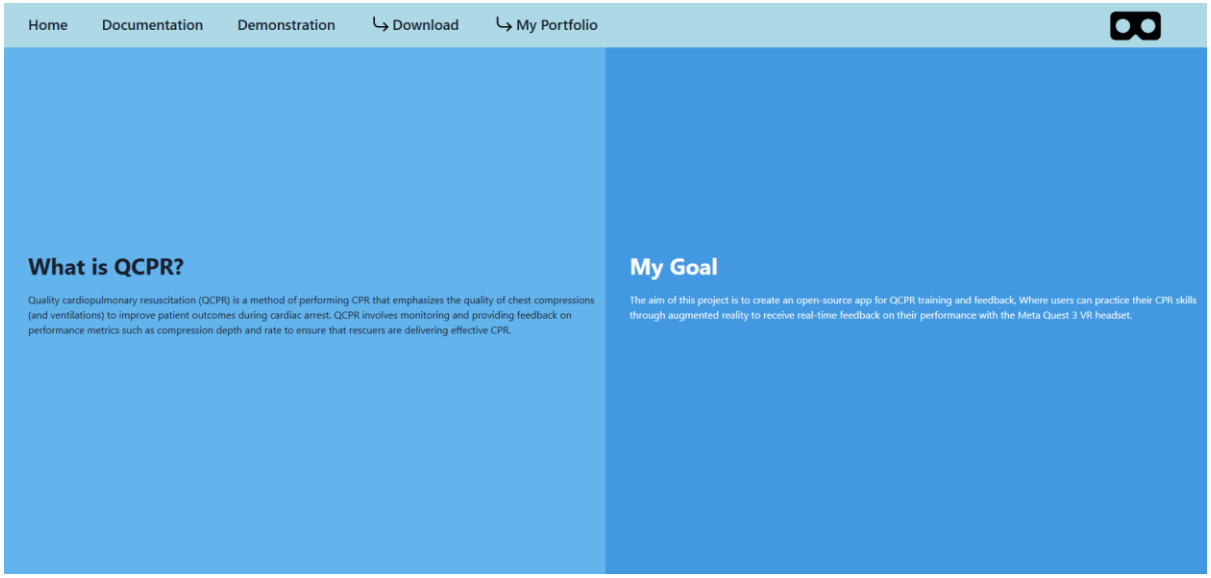


Figure 17. Homepage for QuestQCPR site

The companion site was developed with React and hosted for free using Vercel. When deciding on what to do for the website, I started by creating the navbar and designing the website as I coded it. This was great for saving valuable time and I'm impressed with the final look. In essence, there's 3 pages for home, documentation, and demonstration followed by a download link (to an APK on google drive) and another link to my portfolio.

The home page features content on what QCPR is and what the end goal of my app is. The documentation page features info on how to use as well as installing the application on a Meta Quest 3 device. Finally, the demonstration link leads to a page with a centered YouTube video showcasing me going through the user flow.

The colour palette is mainly blue with dark/light blue accents followed by a majority of black colour for text and the headset icon (available from svgfree.com, located on the top right).

Implementation

Development Methodology & Process

Methodology

The development process was built in iterative stages, spanning the course of three months. This allowed me to build, test, and fix any errors in structured phases. I chose this method because of my inexperience with previously using Unity and this allowed me to learn while developing. Another reason for doing this was because I was constantly thinking of ideas the app could benefit from which I then combined with previous research.

Development Process

Initialization

I started by initializing the project and getting familiar with the utility Meta provides, such as building blocks, components, and sample assets. I made sure to import extra functionality even if I wasn't currently using it in case I wanted to utilize it later in development. One example was the Voice SDK, which while I spent around three days trying to implement it, I wasn't able to add voice commands to my project, which I think really would have benefited it.

Manikin adjustment Phase

The second phase involved importing the manikin model asset from meshy.com and applying a mesh to give it colour, I decided on making it appear faded and tuned down the alpha on the mesh to make it see-through, which in turn made it easier to align over a physical manikin. Then I had to consider how the user would manipulate the position of the manikin before performing CPR. From previous research, I realized that if I utilized the Headsets “CenterEyeAnchor” object, I could have the virtual model follow the Meta Quest camera, emulating where the user looked. However, this came with a caveat.

“How do I stop the model from constantly following the camera?”

I found that with a C# script, I could design it so that when the app starts up, the manikin model is a child element of the anchor object. Then when the user activates a trigger (like a button), the manikin would no longer be a child element and instead return to the root of the Unity scene, while remaining at the desired position.

```
public class ManikinAligner : MonoBehaviour
{
    [Header("Alignment")]
    public Transform head;
    public Vector3 headOffset = new Vector3(0f, -0.6f, -0.6f);

    private bool isLocked = false;

    void Start()
    {
        if (head == null)
        {
            Camera cam = Camera.main;
            if (cam != null)
                head = cam.transform;
        }

        if (head != null)
        {
            transform.SetParent(head);
            transform.localPosition = headOffset;
            transform.localRotation = Quaternion.identity;
        }
    }
```

Figure 18. Aligning manikin to (camera) head on startup

For reference, the app understands that the head is referring to the headset anchor because when the script is linked to an object, I can specify the correlation by dragging the anchor object into the scripts input field (it's coded to expect a value).

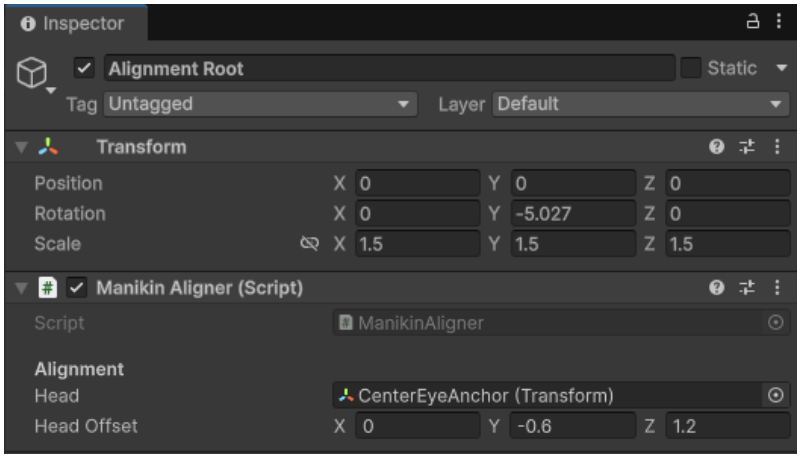


Figure 19. Manikin root object with head variable linking to CenterEyeAnchor

Compression scripting Phase

Following this, I then had to create core user functionalities such as chest compression logic. This was a simple script which had a counter variable which was incremented when the sternum was pushed on.

```
public void AddCompression()
{
    bool anySessionActive = false;

    foreach (CPRSession session in sessionManagers)
    {
        if (session != null && session.IsSessionActive())
        {
            anySessionActive = true;
            break;
        }
    }

    if (!anySessionActive)
        return;

    compressionCount++;
    UpdateText();

    float now = Time.time;

    if (lastCompressionTime > 0f)
    {
        float interval = now - lastCompressionTime;

        compressionIntervals.Add(interval);

        if (compressionIntervals.Count > samplesForAverage)
            compressionIntervals.RemoveAt(0);
    }

    lastCompressionTime = now;
}
```

Figure 20. Addcompression() code

Firstly, the app does not allow for starting compressions without a CPR 1- or 3-minute session being active. This is to make sure sessions are timed and there aren't accidental presses during the manikin alignment phase. This is checked with the foreach function at the beginning of the AddCompression() function.

After this, each time the function runs (where a user performs one compression) and the session is active, the counter increases by 1 and records the timing between each one to be stored by the “compressionIntervals.Add(interval)” for tracking the compression rate later.

The line “compressionIntervals.RemoveAt(0);” was added to ensure the rate of compressions is calculated on a rolling basis (rather than storing all values), removing the first counted rate in the list. This behaviour is triggered by a conditional if statement, which checks whether the number of stored intervals has exceeded the predefined sample limit before removing the oldest entry.

This script also featured calculating the compression rate by averaging the time between compressions and dividing 60 seconds to get a rate per minute. After testing, I decided to implement a smooth function to make the progress bar seem more accurate.

```
void UpdateLiveSpeed()
{
    if (compressionIntervals.Count == 0)
    {
        SmoothSpeed(0f);

        if (compressionRate != null)
            compressionRate.text = "Compression Rate: 0 CPM";

        return;
    }

    // Calculate average interval
    float avgInterval = 0f;
    foreach (float t in compressionIntervals)
        avgInterval += t;

    avgInterval /= compressionIntervals.Count;

    // Convert to CPM
    float rawCPM = 60f / avgInterval;

    // Clamp to realistic CPR range
    rawCPM = Mathf.Clamp(rawCPM, 0f, 160f);

    // Update UI TEXT (live number)
    if (compressionRate != null)
        compressionRate.text = "Compression Rate: " + Mathf.RoundToInt(rawCPM) + " CPM";

    SmoothSpeed(rawCPM);
}
```

Figure 21. UpdateLiveSpeed() function with smoothing calculations

Starts by checking if there's a value for compression intervals, else exits function early. If there is, the function calculates the average time between compressions by summing all stored interval values and dividing the total number of samples. This average interval is then converted into compressions per minute by dividing 60 seconds by the average time.

To prevent unrealistic values, I added logic to restrict the compression counter to be between 0-160 by using Unity's built-in Mathf.Clamp() function.

It's worth mentioning that the visible max value for the progress bar is 100 compressions, which means what while the compression standard is between 100-120, only 100 is visible (the results screen still accounts for the ideal range to be 100-120).

```
void SmoothSpeed(float targetCPMValue)
{
    // Smoothly interpolate speed
    currentDisplayedCPM = Mathf.Lerp(
        currentDisplayedCPM,
        targetCPMValue,
        Time.deltaTime * smoothingSpeed
    );

    float normalized = Mathf.Clamp01(currentDisplayedCPM / targetCPM);
    UpdateSpeedBar(normalized);
}

void UpdateText()
{
    if (compressionText != null)
        compressionText.text = $"Compressions: {compressionCount}";
}

void UpdateSpeedBar(float value)
{
    if (speedBarFill != null)
        speedBarFill.fillAmount = value;
}
```

Figure 22. UpdateLiveSpeed() function with SmoothSpeed and updating text & bar

The SmoothSpeed() function is used to create a gradual transition when updating the displayed compression speed. Instead of instantly jumping to the new value, it uses Unity's Mathf.Lerp() function to insert between the current displayed CPM and the target CPM.

The UpdateText function updates the compression text dynamically with the {compressionCount} value, While the SpeedBar gets updated by adjusting the fillAmount property, it represents the user's current CPR speed as a percentage of the target rate.

```
    public void ShowFeedback()
    {
        if (feedbackPanel != null)
            feedbackPanel.SetActive(true);

        if (feedbackText == null) return;

        if (compressionCount < 80)
        {
            feedbackText.text = "Too few compressions.\nPush faster and maintain rhythm.";
        }
        else if (compressionCount < 100)
        {
            feedbackText.text = "Good effort.\nTry to reach at least 100 compressions.";
        }
        else if (compressionCount <= 120)
        {
            feedbackText.text = "Excellent!\nPerfect compression rate.";
        }
        else
        {
            feedbackText.text = "Too fast.\nSlow down slightly for optimal CPR.";
        }
    }
}
```

Figure 23. ShowFeedback() changing based on final compression count

Lastly, I added this dynamic results screen which shows itself after completing the CPR session. It features text which changes based on the compression score being lower than 80, being less than 100 but more than 80, being more than 100 but less than 120, and lastly being more than 120.

Session scripting Phase

Due to compression counting being dependent on session tracking and vice-versa (circular dependency), these were developed at the same time, despite requiring different sections to discuss. There are two session objects, one for a 1-minute session and another for the 3-minute session which both share one script. Considering they are identical besides the session length; I will only explain the 1-minute object.

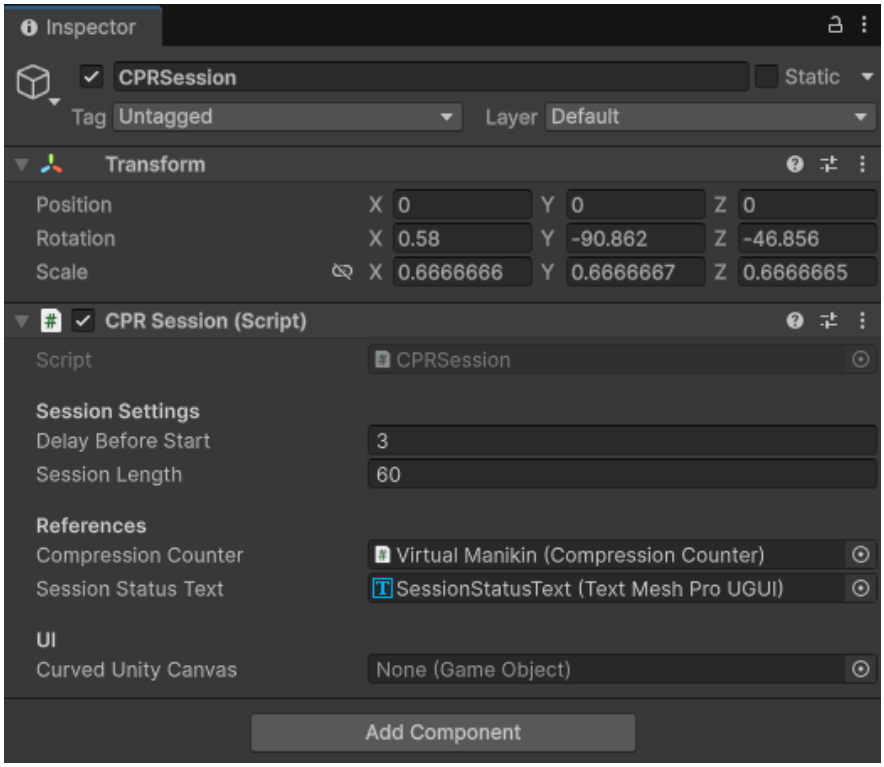


Figure 24. Showcasing the 1-minute session inspector

Here you can see how each section like the delay and session length is flexible in case of changes in the futures like requiring more session times. The counter and text both need to be referenced to ensure the script updates the correct fields.

```
IEnumerator SessionRoutine()
{
    sessionRunning = true;
    sessionActive = false;

    // 3 second delay countdown
    float countdown = delayBeforeStart;

    while (countdown > 0)
    {
        if (sessionStatusText != null)
            sessionStatusText.text = "Starting in: " + Mathf.Ceil(countdown);

        countdown -= Time.deltaTime;
        yield return null;
    }
```

Figure 25. Showcasing the 1-minute session inspector#

This is the coroutine of the start session function. A normal function in Unity executes from start to finish in a single frame. If the countdown and session timer were written as a standard function, the entire process would be completed immediately, leaving no visible delay for the user.

A coroutine (defined by using IEnumerator) allows execution to pause and resume across multiple frames using yield statements. This makes it ideal for handling features such as countdowns, timers, and delayed transitions.

This coroutine starts by initiating a 3-second countdown before setting the session to active.

```
    // Start session
    sessionActive = true;

    if (curvedUnityCanvas != null)
        curvedUnityCanvas.SetActive(false);

    if (sessionStatusText != null)
        sessionStatusText.text = "CPR IN PROGRESS";

    float timer = sessionLength;

    while (timer > 0)
    {
        timer -= Time.deltaTime;

        if (sessionStatusText != null)
            sessionStatusText.text = "Time Left: " + Mathf.Ceil(timer);

        yield return null;
    }
```

Figure 26. Start session logic

This is when the session is active (after 3-second countdown). Now, the session counts down from 60-seconds and displays the time remaining.

```
    // End session
    sessionActive = false;
    sessionRunning = false;

    compressionCounter.ShowFeedback();

    if (curvedUnityCanvas != null)
        curvedUnityCanvas.SetActive(true);

    if (sessionStatusText != null)
        sessionStatusText.text = "SESSION COMPLETE";
}
```

Figure 27. End session logic

Finally, the session completes, and the final compression count and rate remain on screen until a new session is started.

Design Phase

As a result of developing and learning at the same time, this meant not much design research went into the pre-planning phases of the app. This led to the last stage taking the longest as I had to design the entire app flow as well as making the UI seem easy-to-understand.

For the app flow, I structured the experience so that the user is greeted with an optional tutorial, followed by the menus for navigating the app, lastly leading to performing CPR.

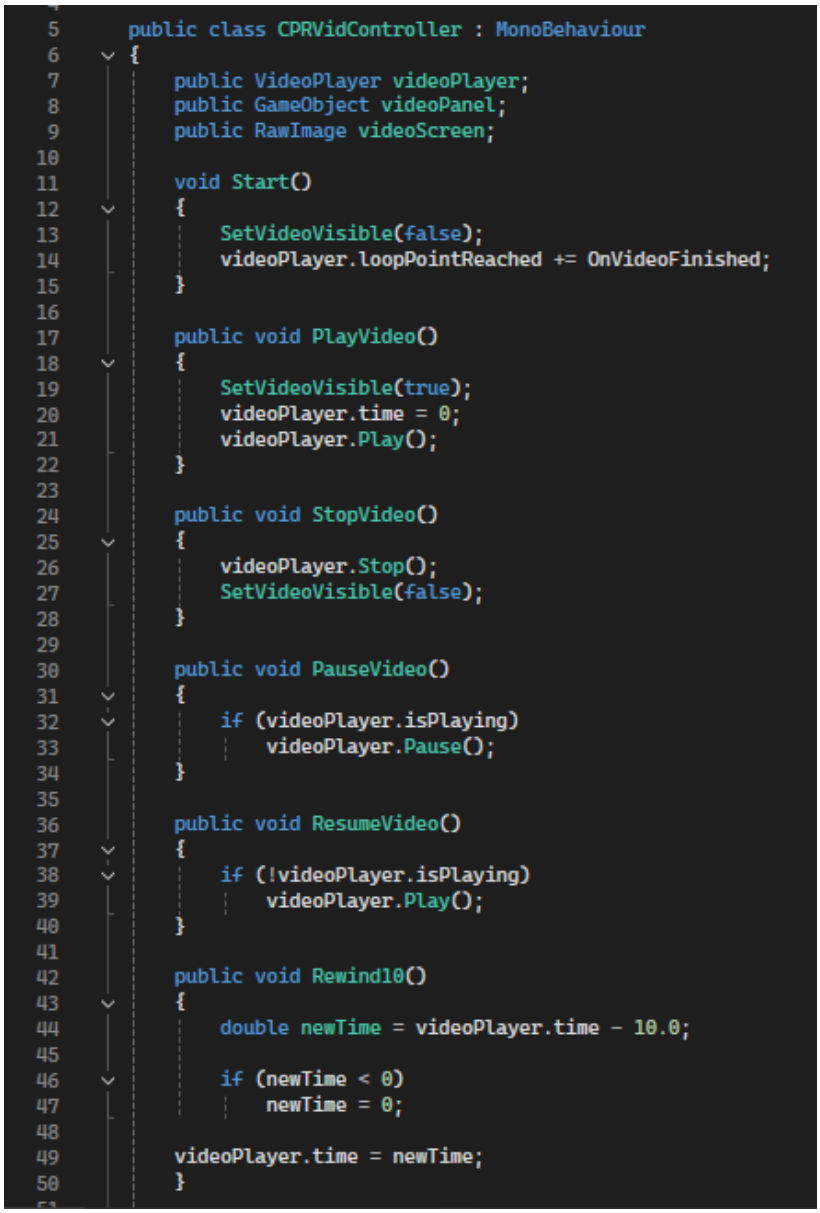
As for the user interface, I took inspiration from the RevivR first-aid site. I made the menu backgrounds plain, with large buttons with clear, visible text. I used red accents for the tutorial buttons which I felt reinforced the idea that the app is for medical training (emulating the white background for red cross symbol).

Figure 28. Each tutorial screen either active/inactive

The logic for the tutorial follows the simple use of inactive objects. When The user presses the next button, the current screen becomes inactive, and the next screen becomes active. This process was repeated for all 6 tutorial screens.

On a small side note, I found that the gesture-activated menu came with ray interactions, simply meaning the user can perform a pinching gesture on the menus outline to drag it closer or push it away from themselves.

This helps when having to align the manikin and for when the user has to lower themselves to perform CPR.



```
public class CPRVidController : MonoBehaviour
{
    public VideoPlayer videoPlayer;
    public GameObject videoPanel;
    public RawImage videoScreen;

    void Start()
    {
        SetVideoVisible(false);
        videoPlayer.loopPointReached += OnVideoFinished;
    }

    public void PlayVideo()
    {
        SetVideoVisible(true);
        videoPlayer.time = 0;
        videoPlayer.Play();
    }

    public void StopVideo()
    {
        videoPlayer.Stop();
        SetVideoVisible(false);
    }

    public void PauseVideo()
    {
        if (videoPlayer.isPlaying)
            videoPlayer.Pause();
    }

    public void ResumeVideo()
    {
        if (!videoPlayer.isPlaying)
            videoPlayer.Play();
    }

    public void Rewind10()
    {
        double newTime = videoPlayer.time - 10.0;

        if (newTime < 0)
            newTime = 0;

    videoPlayer.time = newTime;
    }
```

Figure 29. Controller for video functions

The video controller is initially set to not be visible on app startup. After pressing a button with the PlayVideo function. The event will happen and the video will begin.

Fortunately, Unity C# comes with a built-in properties for the VideoPlayer component. The only real coding I had to do was creating the fast-forward and rewind functionality which was as simple as making a variable for reducing the current video time by 10 (while checking for if the video has progressed less than 10 seconds, which instead reverts it to 0) and linking this function to the button event handler.

Overall, I believe the “learning while implementing” method allowed the app to refine itself after each stage, with the final product being a clear example of what I hoped it would become.

Major challenges & Technical Solutions

The first error I received was that the native way of linking the Unity project to the Meta Quest for testing (Meta Horizon Link App) wasn't working. I received a permanent loading screen when trying to use this method. Fortunately, I wasn't the only one with this problem, and forums as well as YouTube tutorials showcased that building the app for android via an APK was another practical way to test Unity projects. One thing to note was that as the project advanced and file sizes grew, each time I had to test new features, The APK took longer and longer to build and run, with the final APK taking around 8 minutes to compile.

One unavoidable error which was preventing me from building the APK was a duplicate TCP error where the TCP port for the Meta Quest 3 was being read as “already in use”, which didn't make sense as I only had one Unity application open and only one device plugged in. I attempted manually opening terminal to terminate the listening TCP port, unplugging my computer and Meta Quest, but the error persisted. The solution was to completely uninstall Unity and the project folder while also pulling the previous git push (which fortunately was just before the error occurred, which meant minimal lost progress). One thing to note was that this required me to git push –force (as this pull request meant I had to create a new project folder).

A third error I encountered was that the Unity version 6.0 was out of date (problems with SDK web sockets). Thankfully, this was a quick fix, and updating the Unity version didn't cause any problems.

Another problem I had was that the Voice SDK that came with the Meta all-in-one SDK wasn't working, for this problem I wasn't able to come up with a solution despite closely following video guides as well as documentation from Meta themselves. I set up the building block, created a Wix AI profile and linked it to the project, and still even trying to create even simple voice commands didn't work. I ended up not including it in the project and removed the components that were added by the building block.

The last difficulty was less of a problem and more of a limitation. When designing how the user would interact with the manikin, I aligned the manikin to where the user looked with the headset. This was functional, but I was advised by both users and the project supervisor to make the alignment more user-friendly. Unfortunately, when trying to apply grab interactions to the manikin or the hand tracking (from the interaction rig), I wasn't able to get it to function, which meant I had to leave the headset tracking in the app.

Development Environment & Tools

Hardware

The reason I chose the Meta Quest 3 is because it is the best mainstream headset which also features a beginner-friendly software developer kit. Asides from not being able to use the Meta Horizon Link app, the device worked without fault and was straightforward to use. While the app is hands-only, the controllers that came with the headset were simple to use but were only important in setting up the device.

Another feature which I found was how simple it was to record videos for tutorials or for analysing what the user saw after testing the app. It was as simple as pressing the record video button when selecting the app from the Meta library.

Software

For deciding on what software to use, I decided on Unity as it is easier to learn C# over Unreal Engines C++. While Unreal Engine has improved graphics, it is more important to consider how Unity is more lightweight and is quicker to test for. I also found that when researching tutorials for new developers, there were many more demonstrations for Unity.

As for scripting, I used Visual Studio 2022 since it was the default software when installing Unity. Visual studio code was used for coding with the website because it's been my choice of IDE for the past four years. Hosting was done with vercel as it's free to use and linking it to your GitHub repository is incredibly simple. The version control for the QuestQCPR app and the website was managed by GitHub. GitHub was vital in providing access to previous project files when errors occurred and allowed me to keep track of work with commit titles.

Implementation of UI Design

The implementation of the user interface focused on creating a clear, accessible, and intuitive user experience for augmented reality interaction. Since the application was designed for hands-only use, the interface elements had to be large enough for readability as well as interaction (poke and ray interaction) while remaining unobtrusive within the user's field of view.

Menus were implemented from Metas asset scenes and repurposed to feature CPR related content. I decided to use the gesture-activated menu over the previous iteration of having a menu on screen at all times (using scripts to deactivate the menu when performing CPR). I chose this because it allows the user to control the experience and was more intuitive than a static menu.

Feedback like the results screen as well as the progress bar were introduced to provide the user with a method of tracking their progress and a way to improve their practice over different sessions.

I decided on keeping the app "hands-only" because having to navigate menus with controllers and then putting them down to perform CPR seemed counter-intuitive and not user-friendly. This meant that more work was required to allow for hand-to-hand menu interactions.

Testing and Evaluation

Test plan

While the results of the forms were from after the app was completed, I did have a few users to test the earlier iterations of the app. From the testing I found that users were impressed with the idea of the app but voiced that the onscreen menu was distracting and the exit buttons for the panels were sometimes not appearing. Before redesigning the UI, I fixed these issues by creating a script for hiding the menu when performing compressions as well as adding new buttons without visibility issues.

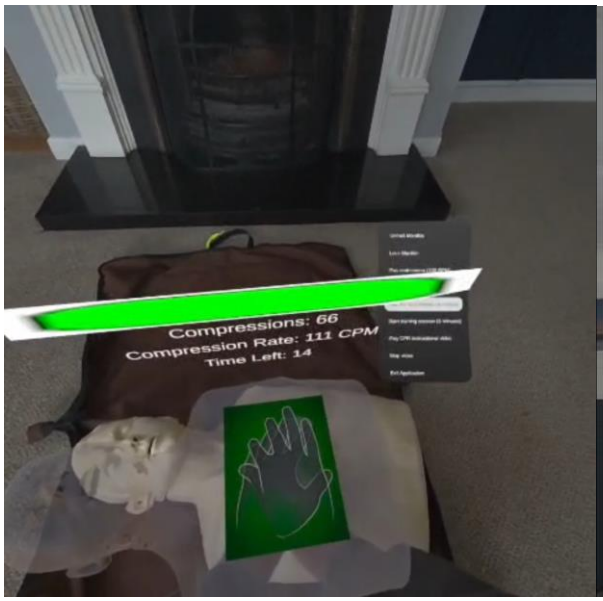


Figure 30. compressions (with physical manikin)

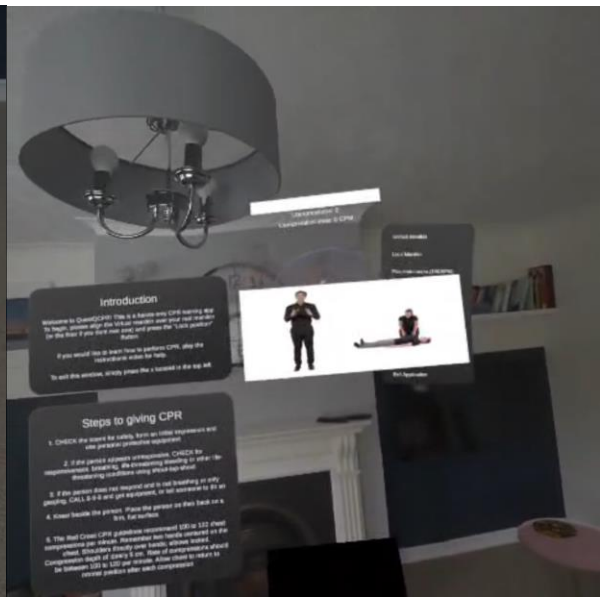


Figure 31. Early design with information panels

While it wasn't said by users during early testing, I found that the previous progress bar was unsightly and did not match the colour palette of the manikin compression area. I added a material to fill the box and changed the colour to fix this.

Usability & User Testing

Apart from creating a user interest form to gauge interest in the app, I also decided to create a usability form to assess how users felt after trying out the application for the first time.

This form featured a series of 9 questions gauging how the user experience felt to use. The form was distributed to the users to test after trying QuestQCPR, and out of the ten testers, nine answered.

Figure 32. Overlay effectiveness question

I wanted to highlight this question because it shows that despite mostly positive feedback across the board, there were concerns with virtual manikin alignment. I think that an improved system of aligning it with your hands rather than with looking with the headset cameras would have solved this. However, after returning post-testing to fix this, I found that I was unable to find a way to implement this with the hand interaction rig.

Figure 33. Overlay effectiveness question

Figure 34. Compression tracker question

I wrote these questions to understand user interest in the apps idea as well as how they felt about the idea of not needing a manikin for practicing CPR. Both questions received positive responses with an average of 7/10.

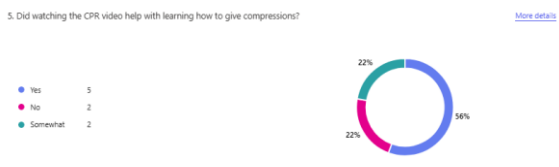


Figure 35. Overlay effectiveness question

Figure 36. Compression tracker question

The first question received backlash with two thirds of the interviewees finding the manikin difficult to align. A solution to this issue should have been to add more methods to interact with the virtual manikin (i.e. allowing the manikin to be grabbable with the user's hand tracking).

The second question received generally positive results, with at least three quarters of the users finding the instructional video at least somewhat beneficial.

Figure 37. Overlay effectiveness question

Figure 38. Compression tracker question

These questions received expected results with users approving the compression tracking and finding the app not overwhelming to use. The responses to the overwhelming question highlight that users generally did not experience a high cognitive load when using the app. It should be noted that one user did experience motion sickness despite the ability of seeing their surroundings through the headset cameras.

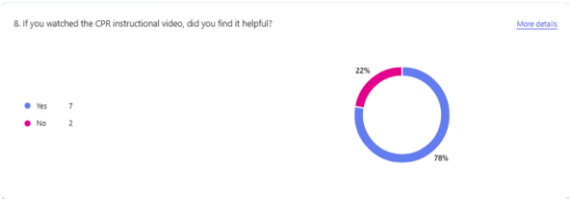


Figure 39. Overlay effectiveness question

As expected, most users found that the educational videos were helpful in learning how to perform CPR. In hindsight, I should have clarified another question asking if the audio was helpful either. This means that the question of whether audio in addition to video was beneficial remained ambiguous.

9. Were there any improvements or changes you would like to see to the app?

7 Responses

ID ↑	Name	Responses
1	anonymous	No, it was very helpful and easy to use.
2	anonymous	Everything was very well laid out.
3	anonymous	took multiple attempts to align manikin, otherwise perfect
4	anonymous	Other than manikin alignment. No.
5	anonymous	No
6	anonymous	n/a
7	anonymous	please fix manikin positioning

Figure 40. Overlay effectiveness question

From these results, I can conclude that aside from manikin alignment issues, users were receptive of the apps usability experience. From personal experience, I am able to align the manikin without difficulty. However, I have much more experience with the app, whereas each user only had one attempt to perform a run-through.

Figure 41. Final product featuring manikin and menus

Project Management

Methodology

Since the project was developed by learning while implementing, I kept general dates for deadlines while deciding how long it would take to work on each stage on a case-by-case basis.

This adaptive structure was necessary because many aspects of the project involved coding with tools and assets I was unfamiliar with, meaning times to complete tasks could not always be accurately predicted in advance.

Tools Used

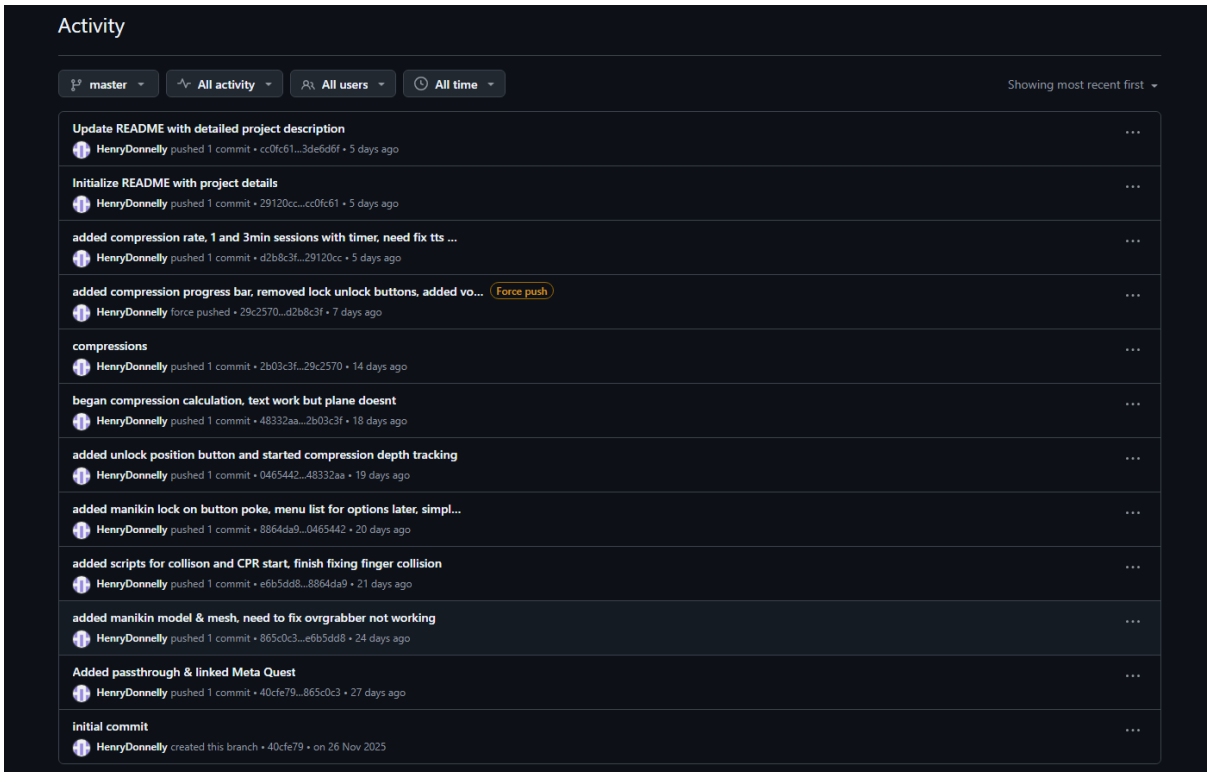


Figure 42. Git commit history

The Git commit history shows the development timeline and iterative stages of the project throughout its lifecycle. Version control was managed using Git to track changes, document progress, and ensure that all stages of development were recorded. I used these commit titles to understand what work had to be done and what was completed. The two force pushes were results of hardware errors that were quick fixes.

In addition to tracking progress using git, I also used an online notepad for recording important deadlines like app due dates, rough timelines on allotted time for feature implementation, etc. I found that this setup was superior to scrum/agile methods as it was simple to understand and supported my constantly changing app development cycle.

Development began in November and ended in late March. Early stages focused on basics like headset setup and finding a manikin model, whereas final stages were dedicated to final deadlines and redesigning the solid foundation I created to be more user-friendly.

Project Supervision

Weekly meetings with Joachim discussing development work helped ensure that I always remained on track with a plan to work towards. Discussing my original app idea and improving it to ensure I could create a feasible project was a great source of help. In addition, being recommended an article Joachim co-developed was great for researching user testing.

Another aspect which was important (mainly for the literature research phase) was that we were asked to develop an extended outline on our project idea. This had me read research papers on augmented reality much earlier before development started.

Conclusion and Future Work

Summary of findings

To summarise, my findings indicate that when users are presented with both visual and audio instructional aids, these multimodal elements can enhance the learning process in CPR training contexts. The addition of features such as guided tutorials, metronome sound cues, and responsive feedback supported app testers in learning correct compression techniques.

Critical Reflection

Developing an augmented reality app for the meta quest greatly improved my skills in coding with Unity and C#, as well as learning how to work with new technologies like the Meta SDK.

With previous projects improving my project planning experience, I felt that this deepened my personal skills in time management and designing user experiences. The nature of the flexible design stages helped me understand how long different tasks realistically take to complete.

Limitations

While I'm happy with the final product, I felt that there were quite a few limitations during development. Which is unfortunate because these drawbacks would have improved the learning experience for the user.

The first limitation was mentioned earlier when discussing the all-in-one SDK. I was unable to get the voice commands to work, which would have added another avenue for the user to interact with the app elements. For example, if the user knew all the commands for interacting with the menu, they could navigate the app quicker.

Another limitation was the inability for the user to manually align the manikin with their hands. This resulted in some feedback from the forms where users felt that taking numerous attempts to align the manikin with the Quest camera was not ideal.

Yet another aspect which limited the app was the lack of progress tracking or long-term data storage. In hindsight, I think creating a scoreboard which stored previous attempts in a JSON file could allow the user to compare previous attempts and drive them to improve, rather than relying on memory for previous compression scores.

Future work

While I enjoyed working with AR as well as Unity, I think for my next project I would like to develop an application for Unreal Engine. Coding with C# was fun, but I feel that learning how to code with C++ would be more of a challenge, which is what I'm looking for. As to whether I would develop for augmented reality, I'm not sure. I think I would be more interested in taking advantage of Unreal Engines superior graphics, focusing more on AI systems. I don't have any specific ideas yet, but a personal game developed as a small passion project would be something I'm interested in.

Overall conclusion

Overall, I believe that the QuestQCPR final product completed its original objective to provide users with an intuitive experience for practicing CPR from home / supplementing training courses with new technology for learning.

Despite the limitations, I'm proud of the result and am glad I learned coding with Unity C# and improved my understanding of augmented reality devices and its potential applications.

References

User Testing Methodologies for immersive Shipwreck Virtual Reality experience (Griffin, R. Pietsch, J, 2026)

A Survey of Augmented Reality (Azuma, 1997)

Affordances and Limitations of Immersive Participatory Augmented Reality Simulations for Teaching and Learning. J Sci Educ Technol 18, 7–22 (Dunleavy, M., Dede, C. & Mitchell, R, 2009).

Multimedia Learning Theory, (Richard E. Mayer, 2001)

The Impact of an Augmented Reality Application on Learning Motivation of Students. (Tasneem Khan, Kevin Johnston, Jacques Ophoff, 2019)

The effectiveness of augmented reality/mixed reality in medical education: a meta-analysis (Zhang R, Jin X, Liu M, Tong HY, 2025)

Advancements in Virtual Reality Technology: A Systematic Review of User Experience and Application Trends (Wenli Shang, Kazlova Alena, 2025)

Multimedia Learning Theory, Wayan K. Yana

Zulfiqar, F., Raza, R., Khan, M. O., Arif, M., Alvi, A., & Alam, T. (2023). Augmented reality and its applications in education: A systematic survey

Marker-less mobile augmented reality application for massive 3D point clouds and semantics, (A.Kharroubi, R.Billen, F.Poux, 2020)

The Challenges of Evaluating the Usability of Augmented Reality (AR), (Jessyca L. Derby, Barbara S. Chaparro, 2021)

Documents in Your Hands: Exploring Interaction Techniques for Spatial Arrangement of Augmented Reality Documents, (Weizhou Luo, Mats Ole Ellenberg, Marc Satkowski, Raimund Dachselt, 2025)

Modern Augmented Reality: Applications, Trends, and Future Directions, (Shervin Minaee, Xiaodan Liang, Shuicheng Yan, 2022)