Senior Project

Mobile application for Children with ASD

Prepared by:

Yahia Walid Mohamed Ahmed Zaky	*7137*
Nader Mohamed Mahmoud Abbas Almadawy	*7115*
Mazen Ahmed Ramzy Aly Abdelrazek Shams	*6999*
Adham Mostafa Abdulhamid Morsy	*6715*
Abdullah Gamal Mubarak	*6771*

Supervised By: Dr. Amira Youssef

Table of Contents

Introduction

Many children with neurodevelopmental disorders such as autism spectrum disorder (ASD), attention deficit hyperactivity disorder (ADHD) or developmental language disorder (DLD) have difficulty recognizing and understanding emotions. However, the reasons for these difficulties are currently not well understood.

Autism spectrum disorder (ASD) and autism are both general terms for a group of complex disorders characterized by impaired social interactions, in which children are hindered by a lack of interest in initiating conversation, atypical eye contact, and especially struggles in recognizing the emotions of others.

Figure 1

Asperger's Syndrome, while historically considered a distinct diagnosis, is now generally classified under the broader umbrella of Autism Spectrum Disorder (ASD). It is characterized by a distinctive set of symptoms that, while sharing similarities with classic autism, present in a milder form. Individuals with Asperger's Syndrome often face challenges in social interactions, experiencing difficulties in interpreting social cues, nonverbal communication, and developing peer relationships. They may exhibit repetitive behaviors or intense, focused interests, displaying a strong preference for routine and order. Communication difficulties may manifest as a literal interpretation of language, with individuals often displaying a lack of understanding of sarcasm or figurative speech.

One notable feature of Asperger's Syndrome is the preservation of average to above-average intellectual abilities. While communication and social interaction may pose challenges, individuals with Asperger's Syndrome often showcase advanced cognitive skills, excelling in specific areas of interest. This intense focus on particular subjects can lead to a depth of knowledge and expertise that sets them apart. In addition to the core symptoms, individuals with Asperger's Syndrome may demonstrate sensory

sensitivities, such as heightened responses to certain sounds, textures, or lights. These sensitivities can impact daily routines and may contribute to behavioral responses aimed at managing sensory input.

Attention Deficit Hyperactivity Disorder (ADHD) is a neurodevelopmental condition marked by persistent patterns of inattention, hyperactivity, and impulsivity. While distinct from autism spectrum disorder (ASD), children with ADHD may encounter challenges in recognizing and understanding emotions. Unlike ASD, the primary difficulties in ADHD are related to sustaining attention, organizing tasks, and regulating impulsive behavior. This disorder often manifests as an ongoing struggle with maintaining focus in various situations, impacting academic, social, and daily functioning.

Developmental Language Disorder (DLD) is characterized by significant difficulties in language development that are not attributed to other cognitive or sensory impairments. Children with DLD face obstacles in understanding and using words, forming sentences, and comprehending language, which can contribute to challenges in communication and social interaction. While there may be overlapping difficulties in recognizing emotions with other neurodevelopmental disorders, DLD specifically addresses language-related impediments that impact various aspects of a child's development.

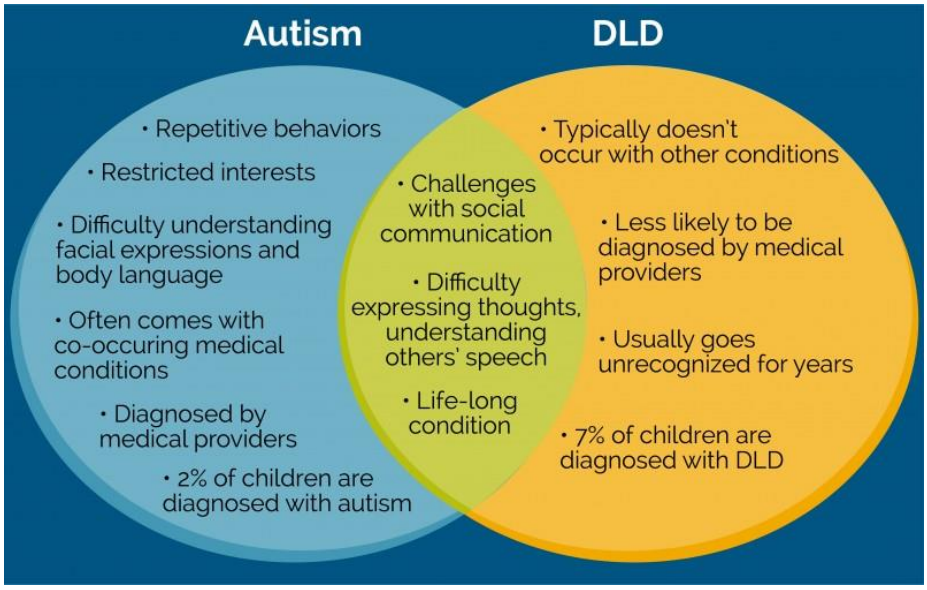


Figure 2

Despite their distinct diagnostic criteria, ASD, ADHD, and DLD can exhibit certain overlapping behaviors. Challenges in recognizing and understanding emotions are a common thread among these neurodevelopmental disorders. Difficulties in social interactions, varying degrees of impaired communication, and struggles in expressing emotions or interpreting others' emotional cues are areas where these disorders may share common ground. Understanding these shared challenges can inform more comprehensive approaches to support individuals with neurodevelopmental disorders and tailor interventions that address their unique needs.

Children with neurodevelopmental disorders like Autism Spectrum Disorder (ASD), Attention Deficit Hyperactivity Disorder (ADHD), and Developmental Language Disorder (DLD) often experience difficulties recognizing and understanding emotions. While the reasons for these difficulties are still not fully understood, research suggests that both linguistic and cognitive factors play a role.

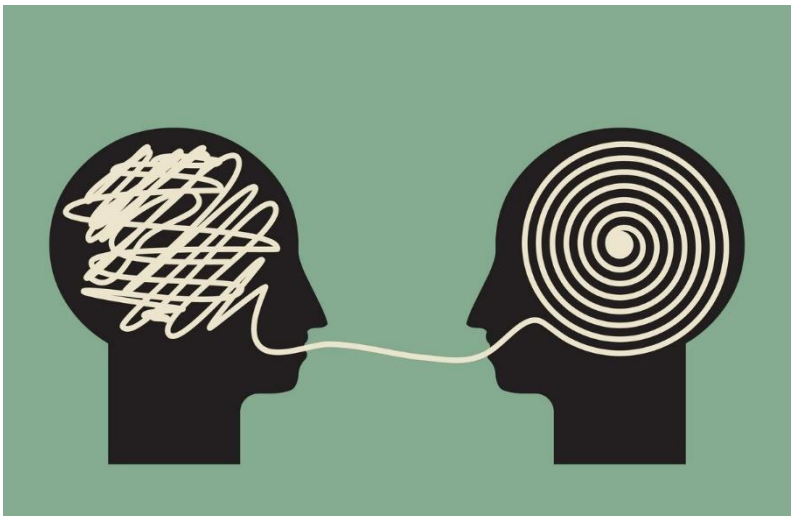


Figure 3

Linguistic Factors:

- **Expressive vocabulary**: Limited expressive vocabulary can hinder a child's ability to label and describe emotions, both in themselves and others. This can lead to difficulties understanding complex emotional expressions and nuanced communication.

- **Pragmatic skills**: Challenges with social communication and pragmatics can make it difficult for children to interpret nonverbal cues, such as facial expressions, tone of voice, and body language, which are crucial for emotion recognition.
- **Theory of Mind**: The ability to understand and predict the thoughts and feelings of others (Theory of Mind) is essential for interpreting emotional cues. Studies suggest that some children with ASD and DLD may have deficits in this area, impacting their ability to recognize emotions.

Cognitive Factors:

- **Attention**: Difficulties with attention, particularly in children with ADHD, can make it challenging to focus on and process emotional cues, leading to misinterpretations or missed signals altogether.
- **Executive function**: Executive function skills, such as planning, problem-solving, and mental flexibility, are necessary for making inferences about emotions based on context and integrating different cues. Deficits in these areas can contribute to emotion recognition difficulties.
- **Sensory processing**: Some children with ASD may have atypical sensory processing, which can make them hypersensitive or hyposensitive to certain stimuli, including emotional cues. This can lead to discomfort or avoidance of situations where emotions are displayed.

Emotion Recognition Difficulties

- **ASD**: Children with ASD often struggle with interpreting facial expressions, tone of voice, and body language. They may also have difficulty understanding the emotions of others in complex situations or identifying subtle emotional cues.
- **ADHD**: Attention difficulties in ADHD can make it hard to focus on and process emotional cues, leading to missed signals or misinterpretations. However, some evidence suggests they may not have as much difficulty recognizing basic emotions as those with ASD or DLD.
- **DLD**: Expressive and receptive language challenges in DLD can hinder understanding the vocabulary and grammar of emotion expression. They may also face difficulties with pragmatic skills, making it hard to interpret nonverbal cues and the broader context of emotional communication.

Influencing Factors

- Linguistic:
    - **ASD**: Limited expressive vocabulary and social communication skills can hinder describing and interpreting emotions. Deficits in Theory of Mind may further impact understanding the mental states of others.
    - **ADHD**: No significant differences in expressive vocabulary compared to typically developing children, but pragmatic skills may still be affected.
    - **DLD**: Core challenges involve language expression and comprehension, impacting both understanding and expressing emotions.
- Cognitive:
    - **ASD**: Difficulties with attention, executive function, and sensory processing can all contribute to emotion recognition challenges. Sensory sensitivities may lead to avoidance of situations where emotions are displayed.
    - **ADHD**: Primarily affected by attentional issues that can hinder processing emotional cues. Executive function deficits may also play a role.
    - **DLD**: Cognitive factors like memory and processing speed may be impacted, affecting the ability to integrate and interpret multiple emotional cues.

Key Differences:

- **ASD**: Typically more significant challenges with interpreting nonverbal cues and understanding the emotions of others in complex situations. Theory of Mind deficits may be more prominent.
- **ADHD**: Attention difficulties impacting focus on emotional cues are a primary concern. Emotion recognition itself may not be as significantly affected as in ASD or DLD.
- **DLD**: Primary challenges stem from language difficulties, hindering both understanding and expressing emotions. Difficulties with pragmatics may further impact nonverbal communication.

In our project, we are dedicated to facilitating the integration of children with neurodevelopmental disorders including Autism Spectrum Disorder (ASD) into society by providing them with a crucial skill set— the ability to distinguish between different emotions. Recognizing the prevalent use of

mobile technology in today's society, our approach leverages the ubiquity of mobile phones as a convenient and accessible platform for intervention. In the contemporary landscape, nearly everyone has access to a mobile phone, making this technology an inclusive and widespread tool for delivering impactful interventions. Through the development of our mobile application, we aim to harness the power of interactive activities, creating engaging experiences that cater to the unique needs of children facing neurodevelopmental challenges. By fostering emotional intelligence in a familiar and easily accessible format, our initiative strives to empower both children and their caregivers on their journey towards improved social integration and overall well-being. So we decided to develop a mobile application using Flutter framework that integrates Augmented Reality.

Discussion

1.Dart and Flutter

Figure 4

1.1 What is Dart ?

Dart is an open source language developed in Google with the aim of allowing developers to use an object-oriented language with static type analysis. Since the first stable release in 2011, Dart has changed quite a bit, both in the language itself and in its primary goals.

1.2 What is Flutter ?

Flutter is an open-source Dart framework for creating cross-platform applications with a single code. The project was officially announced in May 2017 at the Google I/O developer conference. However, its roots trace back to an earlier project known as "Sky," which was an experimental UI framework within Google.

The first stable release of Flutter came in December 2018 with Flutter 1.0. Since then, Flutter has undergone significant growth and evolution, marked by regular updates and enhancements. Google's dedication to refining Flutter is evident in its consistent efforts to address developer feedback, fix bugs, and introduce new features.

Flutter continues to advance with regular releases, addressing new use cases, expanding platform support, and enhancing its features.

1.3 Why Flutter ?

Unlike other multiplatform frameworks, the code of a Flutter application is compiled into native code, so the performance achieved is superior to applications based on web-views. Also, unlike React Native, Flutter doesn’t use native components, instead it comes with its own components, called widgets, so the same app will look the same on any device, regardless of its operating system or version. Thanks to this, the developer does not have to worry about the design of his application looking bad on older devices.

1.4 Flutter Features

1.4.1 Single Codebase for Multiple Platforms

Flutter enables the development of applications for both Android and iOS platforms using a single codebase. This "write once, run anywhere" approach reduces development time and effort.

Figure 5

1.4.2 Hot Reload

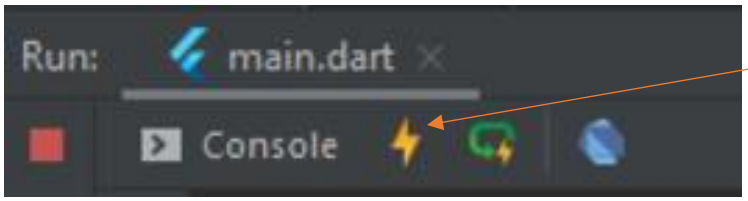


Figure 6 : Snapshot from Android Studio

It is a feature that allows developers to see changes made to their Dart code reflected in the running app almost instantly, without the need to fully restart the app.

It injects updated source code files into the running Dart Virtual Machine (VM), preserving the app's state and enabling a seamless update experience.

How it works:

1. Code Change: The developer makes a change to the code in their IDE.
2. Hot Reload Trigger: The developer saves the changes or presses the "Hot Reload" button in their IDE.
3. Code Injection: Flutter injects the updated code files into the running Dart VM.
4. Widget Tree Updates: The Flutter framework efficiently updates the widget tree based on the code changes.
5. UI Refresh: The app's UI is refreshed on the device or emulator to reflect the changes, often within a second or two.

Benefits:

- Faster Development:
    - Experiment with UI designs, try different features, and fix bugs quickly.

    - See the results of changes immediately, leading to a more iterative and productive workflow.
- Smooth Experimentation:
    - Easily try out different approaches without losing app state or context.
    - Encourages a more creative and exploratory development process.
- Improved Focus:
    - Reduces context switching between coding and app testing.
    - Allows developers to stay "in the flow" and maintain concentration.
- Better UI and UX:
    - Enables rapid refinement of UI elements and user flows.
    - Leads to better-designed and more intuitive user experiences.

Limitations:

- State Preservation: Hot Reload preserves most app state, but some complex state changes or data modifications might require a full restart.

1.4.3 Rich Widget Library

In flutter, the main two types of widgets are (stateless widgets - stateful widgets)

a)**Stateless Widgets**:

- Definition: They represent fixed parts of the UI that don't change dynamically based on user interactions or app data. Their appearance and behavior are solely determined by their input parameters.
- Key Characteristics:
    - Immutable: Their properties remain constant once created.

    - No internal state: They don't store any data that can change over time.
    - Build method: They define their UI layout within a build method that's called only once during widget creation.
- Examples: Text, Icon, Container, Padding, Center, Row, Column, etc.
- When to Use: Choose stateless widgets for UI elements that don't need to react to user input or dynamic data updates. They are efficient for static parts of your app's interface.

b)**Stateful Widgets**:

- Definition: They manage state, allowing their appearance and behavior to change dynamically based on user interactions, data updates, or other events. They maintain a mutable internal state that can be modified and trigger UI updates.
- Key Characteristics:
    - Mutable: Their properties can change over time.
    - State object: They contain a State object that holds their internal state.
    - Build and setState methods: They define their UI layout in a build method and can update the state using setState, causing a re-render of the widget and its children.
- Examples: Checkbox, Radio, Slider, TextField, Form, etc.
- When to Use: Choose stateful widgets for UI elements that need to respond to user input, display dynamic data, or handle changing information. They are essential for creating interactive and responsive UIs.

Key Differences:

- Stateless widgets are immutable and have no internal state, while stateful widgets are mutable and manage state.

- Stateless widgets' build method is called only once, while stateful widgets' build method is called whenever their state changes.
- Stateless widgets are generally more efficient for static UI elements, while stateful widgets are necessary for interactive and dynamic UI elements.

Figure 7 : Stateless vs Stateful widgets

Flutter offers a rich set of widgets that cater to various UI needs. We discuss here some important and frequently used Flutter widgets:

1. Text Widget:

- Function: Displays text with various stylistic options like font size, color, weight, and alignment.
- Use case: Essential for displaying headings, labels, paragraphs, and any other form of textual content.

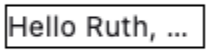


Figure 8: Text Widget

2. Container Widget:

- Function: Acts as a basic container for other widgets, defining their size, padding, margin, and background color.
- Use case: Building layouts, grouping related elements, and adding visual styling to sections of your app.

3. Column & Row Widgets:

- Function: Arrange child widgets vertically (Column) or horizontally (Row) within their container.
- Use case: Building basic layouts like lists, forms, navigation bars, and grids.

Figure 9: Container - Row - Column widgets

4. Image Widget:

- Function: Displays images from various sources like assets, network, or file system.
- Use case: Adding visuals, icons, logos, backgrounds, and any other image elements to your UI.

5. Icon Widget:

- Function: Displays icons from various icon fonts or custom image assets.
- Use case: Adding visual cues, representing actions, enhancing menus, and providing intuitive navigation.
- Flutter integrates with the Material Icons library, which is a comprehensive collection of icons created by Google for use with Material Design. There are many other icons libraries as well which offer various choices for the user to choose from.

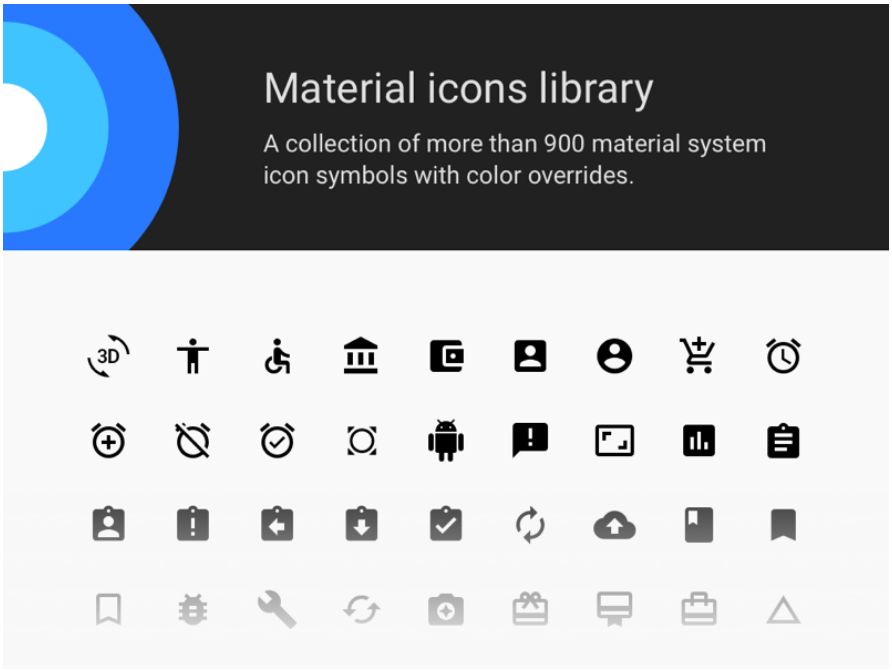


Figure 10: some icons from the Material Icons Library

6. Button Widgets:

Flutter offers a diverse toolkit of buttons to cater to any action or context within the app:

- ElevatedButton: The workhorse, a raised button perfect for primary actions like submitting or starting workflows.
- TextButton: More subtle, TextButtons are ideal for secondary options or within dialogs. Think "Cancel" or filter buttons alongside a list.
- OutlinedButton: A delicate touch, OutlinedButtons provide visual separation with their borders, suitable for toggles or secondary actions in cards and menus. Picture a "Favorite" button outlining a heart icon.
- IconButton: Small and efficient, IconButtons offer quick access to actions via icons, often seen in navigation bars or toolbars. Think of a "Play" button in a music player.
- FloatingActionButton: The attention-grabbing star, FloatingActionButtons stand out on the UI for frequent tasks like adding items or refreshing data. Think of the "+" button in a note-taking app.

Each button can be customized with color, shape, and text, adapting to your app's design and functionality. By choosing the right button for the task, you can guide users intuitively and create a delightful user experience.

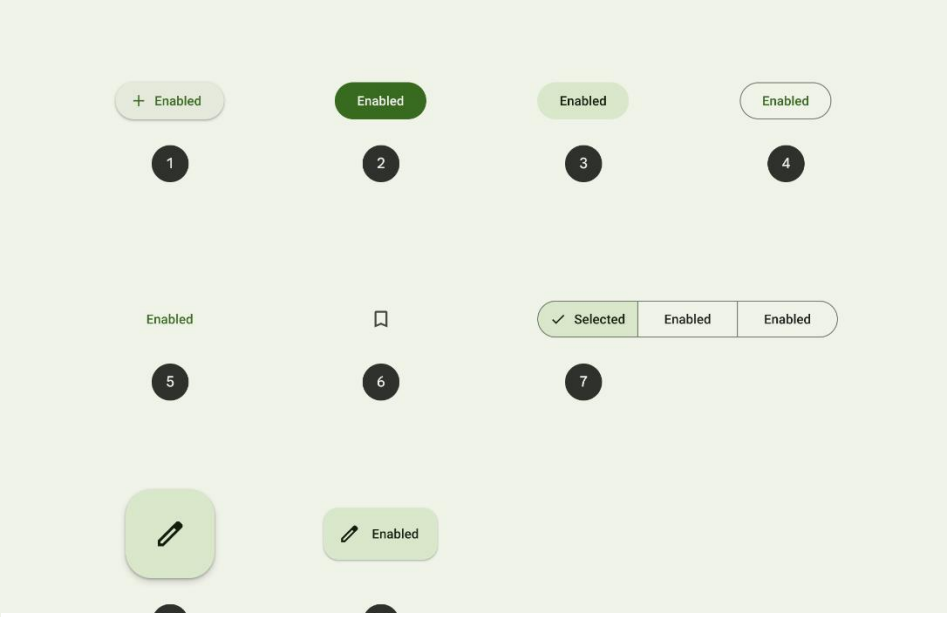


Figure 11
1.Elevated button 2.Filled Button 3.Filled Tonal Button 4.Outlined Button 5.Text Button
6.Icon Button 7.Segmented Button 8.Floating Action Button (FAB) 9.Extened FAB

1.4.4 High performance

Flutter is renowned for its high-performance capabilities, which contribute significantly to the framework's appeal among developers. Several factors contribute to Flutter's ability to deliver smooth and responsive user experiences:

1. **Compiled to Native ARM Code:** Flutter apps are compiled to native ARM code, ensuring optimal performance on both Android and iOS platforms. This compilation allows Flutter to leverage the full power of the device's hardware, delivering efficient and high-performance execution.
2. **Skia Graphics Engine:** Flutter employs the Skia graphics engine, a powerful and efficient 2D graphics library. Skia ensures fast rendering of UI elements, contributing to a smooth and visually appealing user interface. This graphics engine is particularly well-suited for creating complex and dynamic UIs.
3. **Ahead-of-Time (AOT) Compilation:** Flutter uses Ahead-of-Time compilation, which translates Dart code into native machine code before runtime. This approach enhances startup times and reduces runtime overhead, resulting in faster app launches and improved overall performance.
4. **Efficient Rendering Pipeline:** Flutter utilizes a highly efficient rendering pipeline that minimizes unnecessary repaints and layouts. The framework's reactive and declarative approach to building UIs allows for targeted updates, ensuring that only the necessary portions of the interface are recalculated and redrawn.
5. **Customizable Widgets and Flexibility:** Flutter's customizable widgets allow developers to create highly optimized UI components tailored to the specific requirements of their applications. This level of flexibility ensures that developers can optimize the performance of their apps by precisely controlling the rendering of UI elements.
6. **Native ARM Code Execution:** Flutter directly executes compiled native ARM code, bypassing the need for a JavaScript bridge. This results in faster communication between the Flutter engine and the

device's native platform, contributing to reduced latency and improved overall responsiveness.

7. **Minimalistic Runtime Overhead:** Flutter minimizes runtime overhead by avoiding the use of a traditional UI framework. The absence of a native UI framework reduces the performance bottlenecks associated with interactions between the framework and the underlying platform.
8. **Efficient State Management:** Flutter's state management mechanism is designed to be efficient and scalable. The framework's use of a reactive programming model ensures that UI components update only when their state changes, avoiding unnecessary re-renders and enhancing overall performance.
9. **Platform-Specific Code Integration:** Flutter allows developers to integrate platform-specific code directly, enabling the utilization of native features and optimizations. This approach ensures that developers can harness the full potential of each platform while maintaining a unified codebase.

In summary, Flutter's commitment to high performance is evident in its compilation strategies, efficient rendering pipeline, and optimized use of native code. This focus on performance makes Flutter well-suited for building demanding applications with smooth animations, responsiveness, and a delightful user experience.

1.4.5 Flutter's Strong Community

Flutter's strong and vibrant community is a key driving force behind the framework's success and rapid adoption. The Flutter community is a diverse and collaborative ecosystem of developers, designers, and enthusiasts who contribute to the growth and evolution of the framework. Here are some aspects that highlight the strength of Flutter's community:

1. **Active Online Presence:** The Flutter community is highly active across various online platforms, including forums, social media, and developer communities. Websites like Stack Overflow, Reddit, and the official Flutter Discord channel serve as hubs for discussions, problem-solving, and sharing knowledge.

2. **Contributions to Open Source:** Flutter is an open-source project, and the community actively contributes to its development. Developers from around the world submit code contributions, bug reports, and feature requests, fostering a collaborative environment that benefits both beginners and experienced developers.
3. **Extensive Documentation and Guides:** The community plays a crucial role in creating and maintaining extensive documentation, tutorials, and guides. These resources help developers at all levels navigate Flutter's features, troubleshoot issues, and stay updated on best practices.
4. **Package Ecosystem:** Flutter's package ecosystem, hosted on pub.dev, is a testament to the community's dedication. Developers contribute packages that extend Flutter's functionality, providing solutions for common tasks and integrations. The availability of a wide range of packages simplifies development and accelerates project timelines.
5. **Flutter Events and Conferences:** The Flutter community organizes and participates in events, meetups, and conferences worldwide. Events like Flutter Engage and local meetups provide opportunities for networking, learning, and sharing experiences. These gatherings contribute to a sense of camaraderie among Flutter enthusiasts.
6. **Online Courses and Learning Platforms:** Community members often create and share online courses and learning materials to help newcomers get started with Flutter. These resources are valuable for those looking to enhance their skills or transition into Flutter development.
7. **Supportive Social Media Presence:** Flutter has a strong presence on social media platforms like Twitter, where community members share updates, tips, and showcase their projects. The use of hashtags such as #FlutterDev fosters a sense of community and allows developers to engage with each other.

Flutter Discord Channel : https://discord-flutter.netlify.app/

Flutter Subreddit :https://www.reddit.com/r/FlutterDev/

Flutter Twitter Page :https://twitter.com/flutterdev?lang=en

8. **Flutter Challenges and Hackathons:** Community-driven challenges and hackathons encourage developers to showcase their creativity and problem-solving skills. These events not only foster healthy competition but also inspire developers to explore new features and push the boundaries of what can be achieved with Flutter.
9. **Accessibility Advocacy:** The community actively advocates for accessibility in Flutter apps. Discussions and initiatives related to creating inclusive user experiences highlight the community's commitment to making technology accessible to a broader audience.
10. **Mentorship and Support:** Flutter developers often engage in mentorship, offering guidance to those new to the framework. Support forums and chat channels provide spaces for developers to seek help, share insights, and collaborate on problem-solving.

2. ARCore

Figure 12

2.1 What is ARCore?

ARCore is Google's platform for building augmented reality applications. Using different APIs, ARCore enables your phone to sense its environment, understand the world and interact with information. ARCore is compatible with android devices unlike its counterpart, ARKit, which works with iOS devices only.

It was initially released in March 2018 for android devices version 7.0 and higher. However, its first stable version, which was 1.22.20322056, was released in January 2021.

2.2 Why ARCore?

ARCore provides SDKs for many of the most popular development environments. These SDKs provide native APIs for all of the essential AR features like motion tracking, environmental understanding, and light estimation. With these capabilities you can build entirely new AR experiences or enhance existing apps with AR features.

According to the chart in figure 13, more than 70 percent of the world's population are using phones powered by android and to allow our application to benefit as many users as possible, we will be using ARCore for our project.

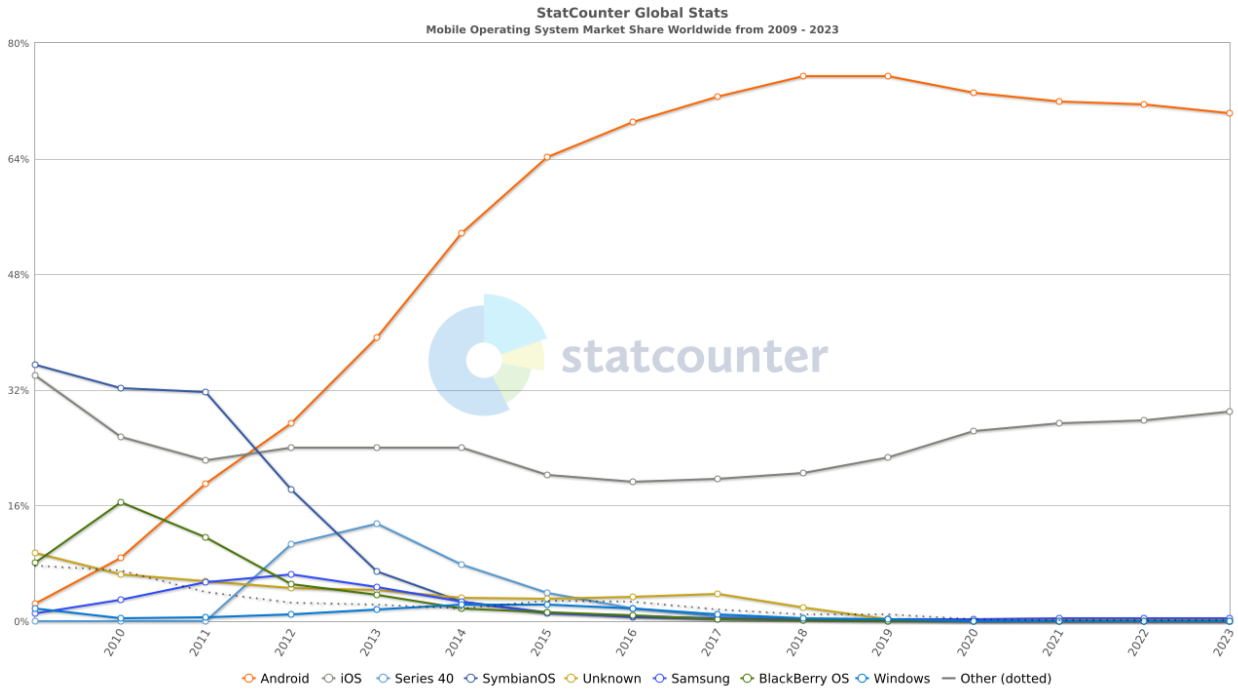


Figure 13: Orange: Android …. Grey: iOS

2.3 ARCore Features

There are three main features to ARCore that make it excel and become our choice in this application: Motion tracking, Environmental understanding and Light estimation. However, we are going to mention more of its features to show how powerful this tool is in the development of this application.

2.3.1 Motion Tracking

ARCore uses a process called simultaneous localization and mapping, short for “SLAM”, to understand where the phone is relative to its surroundings. It detects visually distinct features in the captured camera image called feature points and uses these points to compute its change in location. The visual information is combined with inertial measurements from the device's IMU (Inertial Measurement Unit) to estimate the pose (position and orientation) of the camera relative to the world over time.

We are able to render virtual content with a correct perspective but aligning both poses resulting from the virtual camera that renders the 3D content and the device’s camera from ARCore.

2.3.2 Environmental Understanding

ARCore keeps improving its understanding of the real world environment by capturing feature points and planes.

ARCore looks for clusters of feature points that appear to lie on common horizontal or vertical surfaces, like tables or walls, and makes these surfaces available to the app as geometric planes. It can also determine the boundary of each geometric plane and make that information available to the app. You can use this information to place virtual objects resting on flat surfaces as shown in the example in figure 14.

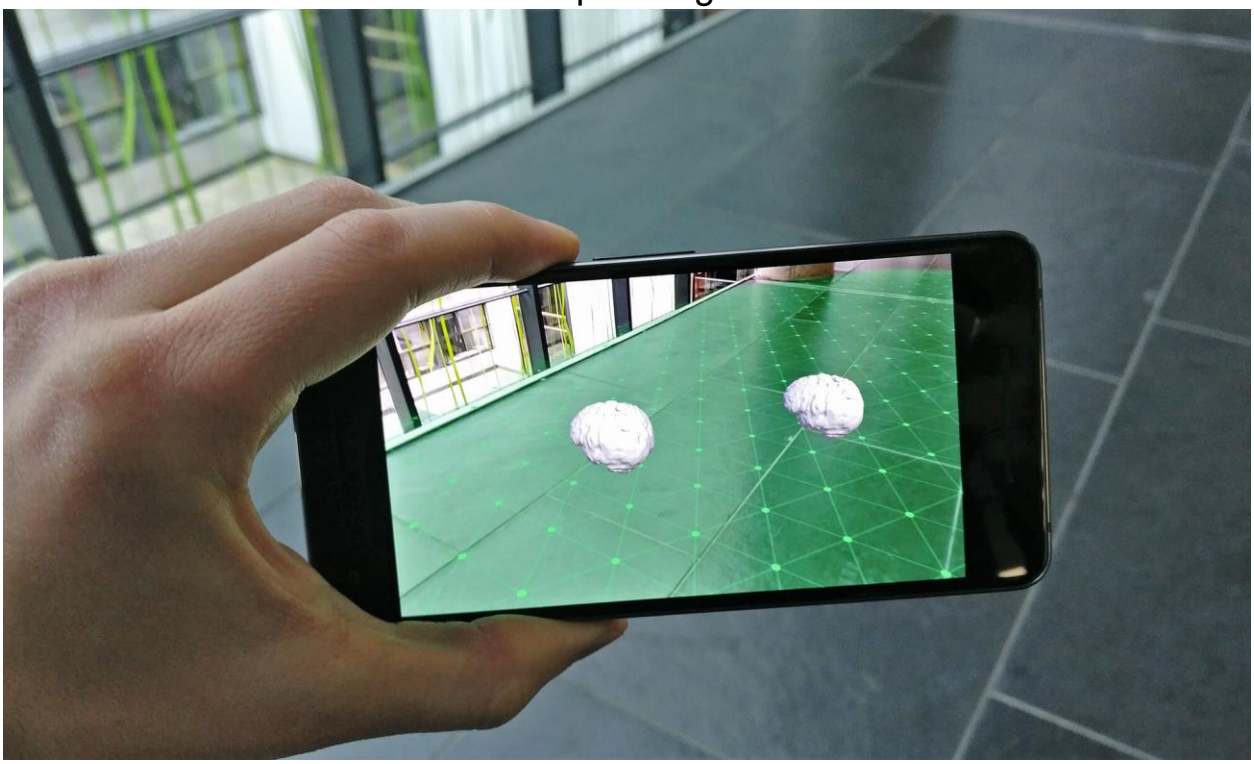


Figure 14

However, a drawback to using feature points to detect planes is that flat surfaces without texture, such as a white wall, may not be properly detected.

2.3.3 Light Estimation

ARCore can detect information about the lighting of its environment and provide the average intensity and color correction of a given camera image. This information lets you light your virtual objects under the same conditions as the environment around them, increasing the sense of realism.

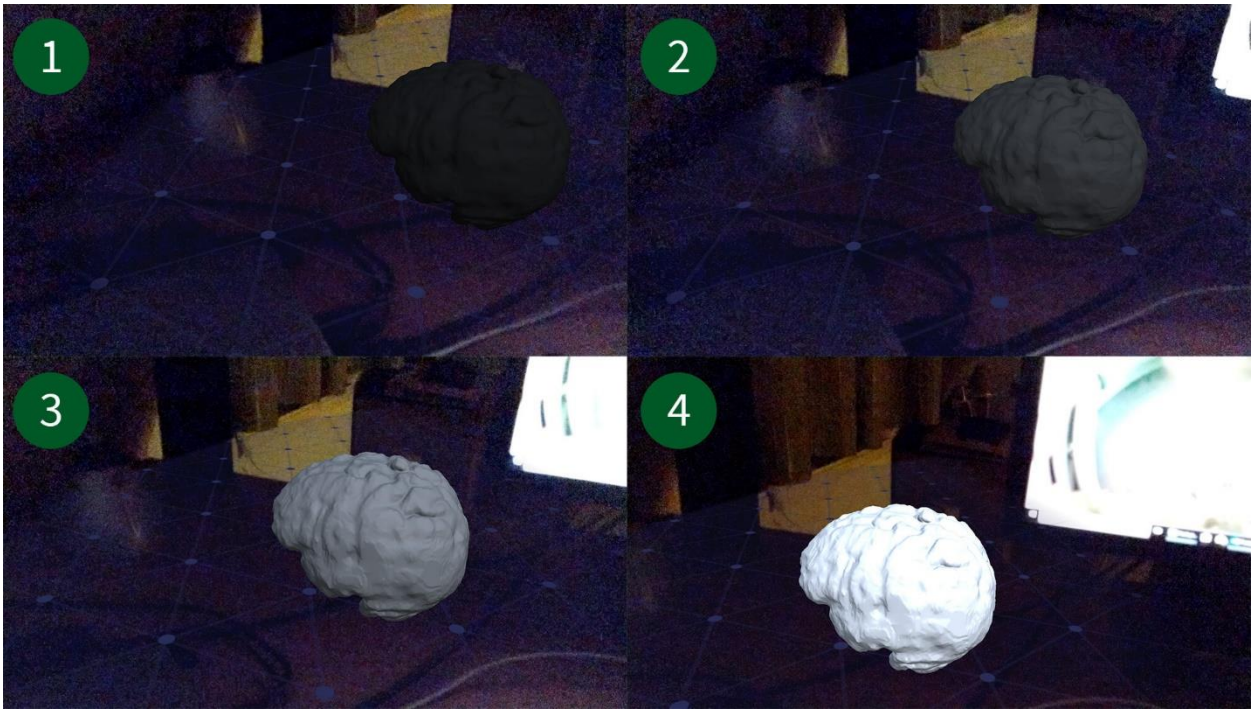


Figure 15

2.3.4 User Interaction

ARCore uses hit testing to take a 2D (x,y) coordinate corresponding to the phone's screen (provided by a tap or any other interaction with the app) and projects a ray into the camera's view of the world, returning any geometric planes or feature points that the ray intersects, along with the pose of that intersection in world space. This allows users to select or otherwise interact with objects in the environment.

2.3.5 Oriented Points

Oriented points allow us to place virtual objects on angled surfaces on a slope. When performing a hit test that returns a feature point, ARCore will look at nearby feature points and use those to attempt to estimate the angle of the surface at the given feature point. ARCore will then return a pose that takes that angle into account.

But again, due to ARCore using clusters of feature points to detect the angle of the surface, surfaces without texture, such as a white wall, may not be properly detected.

2.3.6 Anchors and Trackables

When we want to place a virtual object, we need to define an anchor to ensure that ARCore tracks the object's position over time. Most of the time we create an anchor based on the pose returned by a hit test, as described in user interaction section.

The fact that poses can change means that ARCore may update the position of environmental objects like geometric planes and feature points over time. Planes and points are a special type of object called trackables. These are objects that ARCore will track over time. You can anchor virtual objects to specific trackables to ensure that the relationship between your virtual object and the trackable remains stable even as the device moves around.

This means that if you place a virtual Android figurine on your desk, if ARCore later adjusts the pose of the geometric plane associated with the desk, the Android figurine will still appear to stay on top of the table.

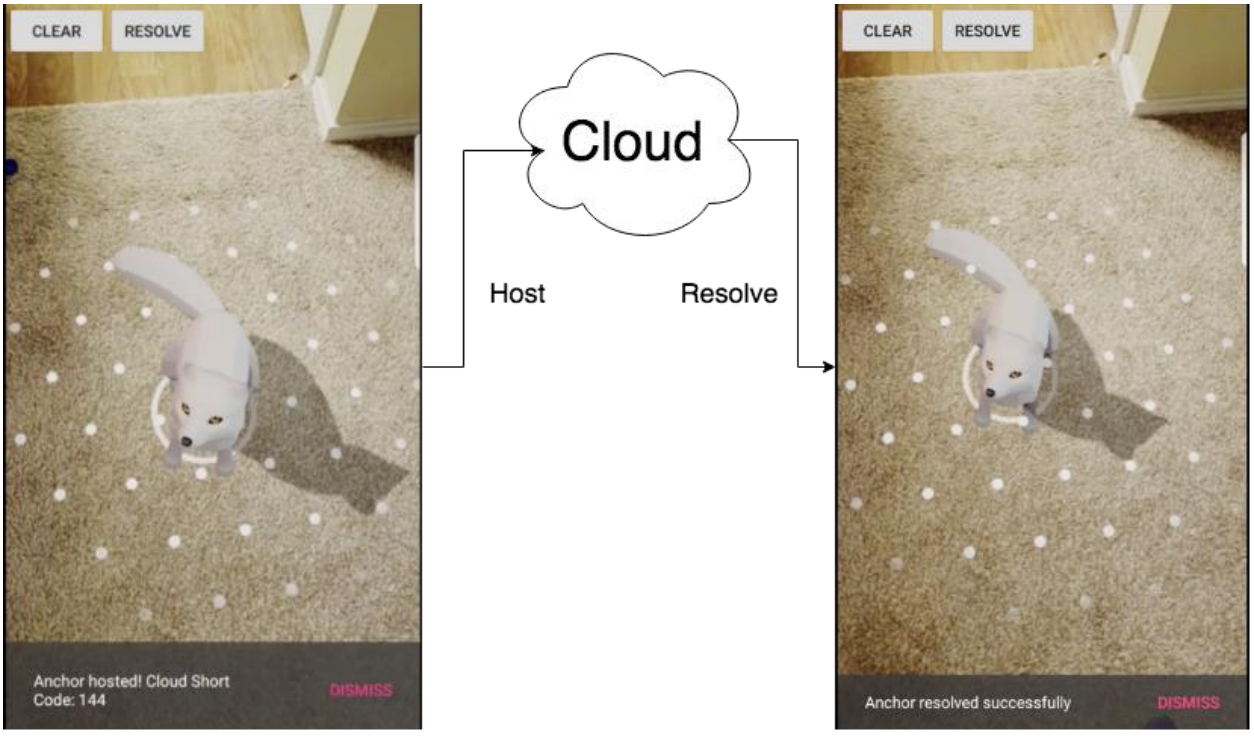


Figure 16

It is a good practice to reuse anchors if possible and detach ones that are no longer needed to reduce CPU load.

2.3.7 Augmented Images

Augmented Images is a feature that allows you to build AR apps that can respond to specific 2D images such as product packaging or movie posters. Users can trigger AR experiences when they point their phone's camera at specific images. For example, they could point their phone's camera at a movie poster and have a character pop out. ARCore can also track moving images such as, a billboard on the side of a moving bus.

Images can be compiled offline to create an image database, or individual images can be added in real time from the device. Once registered, ARCore will detect these images, the images' boundaries, and return a corresponding pose.

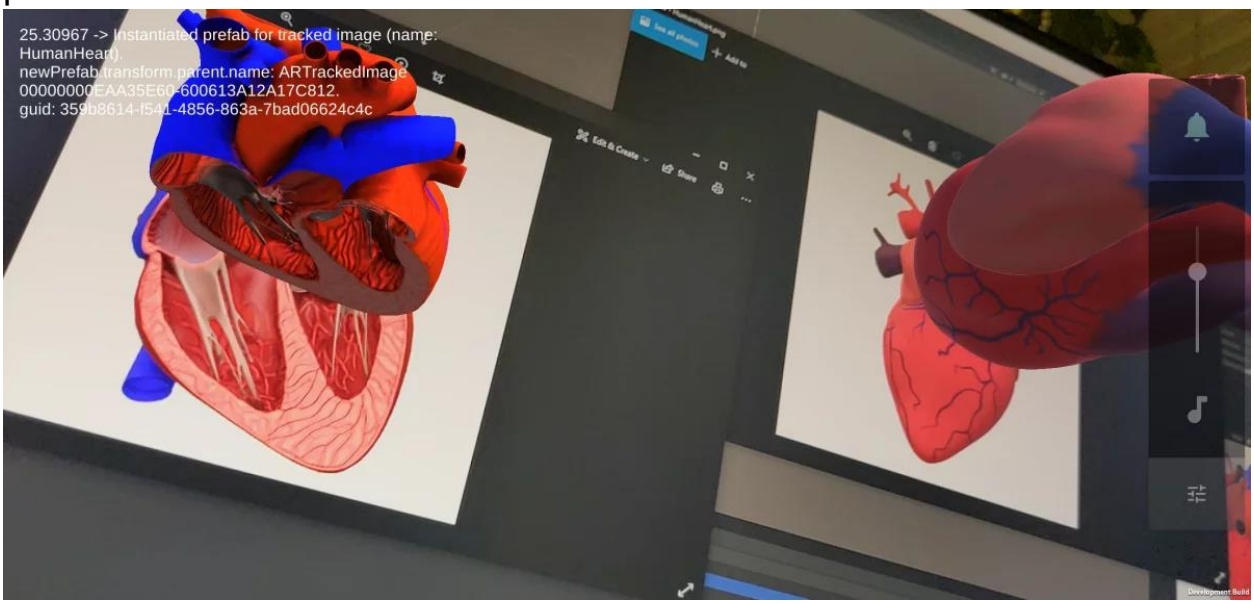


Figure 17

3. Flutter plugins:

2 plugins are used:

- arcore_flutter_plugin
- ar_flutter_plugin

4.1 arcore_flutter_plugin:

4.1.1 Introduction:

"arcore_flutter_plugin" is a Flutter Plugin for Augmented Reality - Supports ARKit for iOS and ARCore for Android devices.

4.1.2 Origin:

The repository on github is owned by Gian Marco Di Francescoand it was built on the basis of arkit_flutter_plugin by Oleksandr Leuschenko.

It was created to streamline the AR app development for both IOS and Android devices.

4.1.3 Featured classes:

ArCoreView:

- Displays the AR scene and camera feed.
- Handles user interactions with virtual objects.
- Manages rendering and frame updates.

ArCoreController:

- Manages ARCore session initialization and configuration.
- Tracks device movement and orientation.
- Detects planes and feature points in the environment.

- Handles light estimation and shadow rendering.

ArCoreFaceView:

- Specifically designed for facial AR experiences.
- Detects and tracks faces in the camera view.
- Renders virtual objects anchored to facial features.

ArCoreFaceContrIler:

- Manages face detection and tracking algorithms.
- Provides access to facial landmarks and expressions.

ArCoreSphere, ArCoreCylinder, ArCoreCube:

- 3D geometric primitives for creating virtual objects.
- Set their position, rotation, scale, and material properties.

ArCoreNode:

- Base class for representing objects in the AR scene.
- Contains a transform (position, rotation, scale).
- Can be parented to other nodes for hierarchical structures.

ArCoeMaterial:

- Defines surface appearance properties like color, texture, and lighting.
- Applied to 3D objects to create visual variety.

ArCoreHitTestResult:

- Contains information about raycast collisions with virtual objects or planes.
- Used for user interaction and object placement.

ArCoreRotatingNode:

- Rotates continuously around a specified axis.
- Used for creating dynamic animations or rotating objects.

ArCorePlane:

- Represents detected planar surfaces in the real world.
- Used for placing virtual objects on detected planes.

ArCoreReferenceNode:

- Represents a pre-configured model or object from a reference file.
- Efficiently loads and renders complex 3D assets.

4.2.5 Why was this plugin chosen?

Compatibility:

The plugin is compatible with our flutter version and it helps in developing products we desire (android app in our case).

Functionality:

It offers desirable features that can be used in the development of our android application.

The other plugin we use uses it as it’s basis.

Documentation:

The plugin is well documented and every library included in it is also well documented.

Support:

The last commit was done on Feb 20, 2022 totalling to 165 commits.

Ease of use:

It facilitates the process of making apps by providing building blocks to common features instead of writing it from scratch.

Community support:

There is already formus that acknowledges this plugin in developing android applications with frequently asked questions we may encounter answered.

4.2 ar_flutter_plugin:

4.2.1 Introduction:

"ar_flutter_plugin" is a Flutter Plugin for Augmented Reality - Supports ARKit for iOS and ARCore for Android devices.

4.2.2 Origin:

The repository on github is owned by Lars Carius and it was built on the basis of arkit_flutter_plugin by Oleksandr Leuschenko and Gian Marco Di Francesco's arcore_flutter_plugin.

It was created to streamline the AR app development for both IOS and Android devices.

4.2.3 Features:

- Serve as a bridge between Flutter's cross-platform framework and the native AR frameworks on each device (ARCore for Android, ARKit for iOS)
- Provide a simplified way to leverage those native capabilities within a Flutter app.
- Camera access and control for AR experiences.
- Scene rendering and management.
- Placement and manipulation of 3D objects in the real world.

- Handling user interactions within the AR environment.

4.2.4 Examples:

Example Name	Description	Link to Code
Debug Options	Simple AR scene with toggles to visualize the world origin, feature points and tracked planes	Debug Options Code
Local & Online Objets	AR scene with buttons to place GLTF objects from the flutter asset folders, GLB objects from the internet, or a GLB object from the app's Documents directory at a given position, rotation and scale. Additional buttons allow to modify scale, position and orientation with regard to the world origin after objects have been placed.	Local & Online Objects Code
Objects & Anchors on Planes	AR Scene in which tapping on a plane creates an anchor with a 3D model attached to it	Objects & Anchors on Planes Code
Object Transformation Gestures	Same as Objects & Anchors on Planes example, but objects can be panned and rotated using gestures after being placed	Objects & Anchors on Planes Code
Screenshots	Same as Objects & Anchors on Planes Example, but the snapshot function is used to take screenshots of the AR Scene	Screenshots Code
Cloud Anchors	AR Scene in which objects can be placed, uploaded and downloaded, thus creating an interactive AR experience that can be shared	Cloud Anchors Code

Example Name	Description	Link to Code
	between multiple devices. Currently, the example allows to upload the last placed object along with its anchor and download all anchors within a radius of 100m along with all the attached objects (independent of which device originally placed the objects). As sharing the objects is done by using the Google Cloud Anchor Service and Firebase, this requires some additional setup, please read Getting Started with cloud anchors	
External Object Management	Similar to the Cloud Anchors example, but contains UI to choose between different models. Rather than being hard-coded, an external database (Firestore) is used to manage the available models. As sharing the objects is done by using the Google Cloud Anchor Service and Firebase, this requires some additional setup, please read Getting Started with cloud anchors. Also make sure that in your Firestore database, the collection "models" contains some entries with the fields "name", "image", and "uri", where "uri" points to the raw file of a model in GLB format	External Model Management Code

4.2.5 Why was this plugin chosen?

Compatibility:

The plugin is compatible with our flutter version and it helps in developing products we desire (android app in our case).

Functionality:

It offers desirable features that can be used in the development of out android application.

Documentation:

The plugin is well documented and every library included in it is also well documented.

Support:

The last commit was done on Dec 23, 2022 totalling to 180 commits.

Ease of use:

It facilitates the process of making apps by providing building blocks to common features instead of writing it from scratch.

Community support:

There is already formus that acknowledges this plugin in developing android applications with frequently asked questions we may encounter answered.

4. Choices we had to reconsider:

Some options we had to develop were reconsidered for the usage of flutter and arcore because their cons usually offset their pros.

1. Native Development:

Pros:

- High performance and control: Granular access to device hardware and capabilities.
- Platform-specific features: Leverage native AR frameworks like ARKit (iOS) and ARCore (Android) for optimal performance and features.
- Offline functionality: Works without internet connection in some cases.

Cons:

- Higher development cost and complexity: Requires separate codebases for each platform (iOS and Android).
- Steeper learning curve: Requires knowledge of specific native programming languages and AR frameworks.
- Limited cross-platform compatibility: Difficult to share code across platforms.

2. Web-based AR with WebGL:

Figure 21

Pros:

- Platform-agnostic: Works on any device with a web browser, no app installation required.
- Easy to share and update: No app store deployment needed, updates reach users instantly.
- Lower development cost: Potentially cheaper to develop and maintain compared to native or cross-platform apps.

Cons:

- Performance limitations: WebGL can be less performant than native AR frameworks.
- Limited device capabilities: Web browsers might not have access to all device features.
- Offline functionality: Difficult to achieve true offline functionality due to browser limitations.

3. Cloud-based AR:

Figure 22

Pros:

- Offload processing: Heavy processing can be done on the cloud, reducing device requirements.
- Scalability: Can handle large numbers of users and complex AR experiences.
- Centralized data management: Easier to manage data and updates across different devices.

Cons:

- Network dependency: Requires a stable internet connection for proper functioning.
- Latency issues: Network latency can impact responsiveness and user experience.
- Security concerns: Data privacy and security considerations need to be addressed.
- Costs: Hefty Processing Bills.
- Vendor Lock-in: Choosing a specific cloud AR platform often leads to vendor lock-in, making it difficult and expensive to switch to another provider later.

4.Cross-Platform Frameworks:

1. React Native:

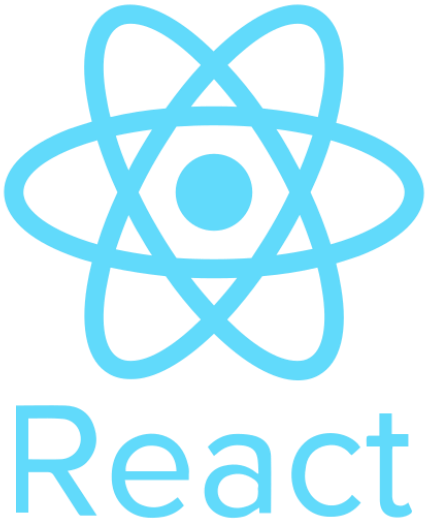


Figure 23

Pros:

- Large community and abundant resources: Easy to find solutions and support.
- Hot reloading and live-preview: Efficient development iteration with instant updates.
- Wide range of existing third-party libraries: Can integrate AR libraries like React Native ARKit or react-native-arcore.
- Familiar JavaScript development: Suitable for developers already familiar with React.

Cons:

- Performance limitations compared to native: Might not be ideal for complex or graphics-intensive AR experiences.
- Potential compatibility issues: Integrating various AR libraries can present challenges.
- Learning curve for native AR frameworks: Need to understand basic concepts of ARKit or ARCore for optimal implementation.

2. Xamarin:

Figure 24

Pros:

- Mature and stable framework: Backed by Microsoft with years of development and a large user base.
- Excellent native integration: Provides access to full native capabilities of each platform.
- Good performance and reliability: Suitable for enterprise-grade applications.
- Xamarin.Forms for cross-platform UI: Can build a shared UI codebase for both iOS and Android.

Cons:

- Steeper learning curve compared to React Native and Flutter: Requires knowledge of C# and platform-specific concepts.
- Limited AR development resources: Not as many dedicated AR libraries or community support compared to other frameworks.
- Less focus on UI and hot reloading: Development process might be less iterative and dynamic.

3. NativeScript:

Figure 25

Pros:

- Truly native app development: Leverages native UI components and APIs for each platform.
- Performance and stability: Delivers native-like experience and responsiveness.
- TypeScript support: Familiar language for developers with JavaScript experience.
- Open-source and free to use: Cost-effective option for individual developers or small teams.

Cons:

- Higher development complexity: Requires separate codebases for each platform.
- Steeper learning curve: Need to understand platform-specific development approaches and frameworks.
- Limited AR libraries and resources: Not as many readily available options compared to other frameworks.

5. AR SDKs:

1. Vuforia:

Figure 26

Pros:

- Cross-platform support for iOS, Android, Unity, and web.
- Strong image and object recognition capabilities.
- Cloud recognition for large-scale databases of images or objects.
- Extensive documentation and tutorials.

Cons:

- Can be less performant than native frameworks in some cases.
- Free plan has limitations, paid plans can be expensive for large-scale projects.

2. Wikitude:

Figure 27

Pros:

- Cross-platform support for iOS, Android, Unity, and web.
- Good tracking and image recognition capabilities.
- Cloud-based recognition and storage options.
- Supports geolocation-based AR experiences.

Cons:

- Free plan has limitations, paid plans can be costly for commercial projects.
- Documentation and support could be more comprehensive.

3. AR.js:

Figure 28

Pros:

- Web-based AR for easy deployment and accessibility.
- No app installation required.
- Works on most modern browsers that support WebGL.
- Open-source and free to use.

Cons:

- Limited performance compared to native or cross-platform SDKs.
- Relies on marker-based AR, which can be less immersive.
- Not suitable for complex AR experiences.

4. Unity MARS:

Figure 29

Pros:

- Integrated into Unity game engine for streamlined development.
- Powerful authoring tools for complex AR experiences.
- Supports rapid prototyping and iteration.
- Good for gaming and simulation-based AR applications.

Cons:

- Requires knowledge of Unity development.
- Can be resource-intensive for large-scale projects.
- Licensing costs for commercial use.

5. MAXST:

Figure 30

Pros:

- Cross-platform support for iOS, Android, Unity, and web.
- Strong focus on industrial and enterprise AR applications.
- Features for object recognition, spatial mapping, and remote assistance.

Cons:

- Pricing structure can be complex, especially for enterprise use.
- Documentation and support could be more comprehensive.

5. App Features:

5.1 MCQ Game Feature

5.1.1 Feature Description:

The feature is a Multiple Choice Question (MCQ) game designed to aid autistic children in recognizing and familiarizing themselves with various emotions. The game consists of four distinct levels, each presenting a unique set of visual stimuli to engage the child's understanding of emotional expressions.

Level 1 employs images of children portraying different emotions, allowing the child to associate facial expressions with the corresponding emotional states.

Level 2 builds upon this concept by utilizing a consistent character, a young girl, across multiple images, enabling the child to recognize emotional nuances within a familiar context.

The third level, the Emoji Level, introduces a modern and widely used form of emotional representation, emojis. By presenting emojis as visual cues, the child can develop an understanding of the emotions conveyed through these digital icons, which may be particularly relevant in the context of social media and digital communication.

Finally, Level 4 (Cartoon Level) introduces a cartoon character, a young boy, as the subject of emotional expression. This level aims to bridge the gap between realistic and stylized representations, preparing the child to recognize and interpret emotional states across various visual mediums, including animated content.

Throughout each level, the child is presented with multiple-choice questions, prompting them to identify the emotion depicted in the given visual stimulus. This interactive approach reinforces the child's ability to recognize and associate emotional expressions with their corresponding labels, fostering a deeper understanding of emotions and their manifestations.

5.1.2 Materials used in the feature:

In developing this Multiple Choice Question (MCQ) game, it was crucial to ensure that all visual materials, especially those depicting children, were obtained from legitimate sources and could be used legally and ethically within the context of this application. We carefully reviewed the policies and usage rights associated with each website and source to confirm that the images could be appropriately utilized for this project. The sources and usage rights of these materials are outlined below:

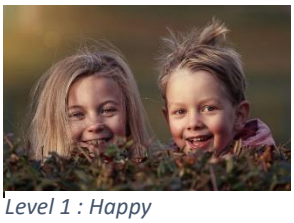
unsplash.com : The photos on Unsplash are free to use and can be used for most commercial, personal projects, and for editorial use. You do not need to ask permission from or provide credit to the photographer or Unsplash, although it is appreciated when possible.

pexels.com : All photos and videos on Pexels are free for commercial use. You can use them on your commercial website, blog, product, or anywhere else.

pixabay.com : All content is released under the Pixabay license, which makes the content safe to use without asking permission or giving credit to the artist, even for commercial purposes

freepik.com : Freepik allows you to use all free resources for personal and commercial projects. However, if you are a free user, you must give credit to the author by using the line “Designed by Freepik”.

The following are the images used for **Level 1** :



Level 1 : Happy

Level 1 : Angry (designed by freepik)

Level 1 : Sad

Level 1 : Scared

Level 1 : Surprised (designed by freepik)

For **Level 2** , some images were chosen from this collection :

Level 2 : collection of different emotions for the same woman

For the **Emoji Level** (third level) , some of the most used digital emotions (emojis) were chosen from the following collection :

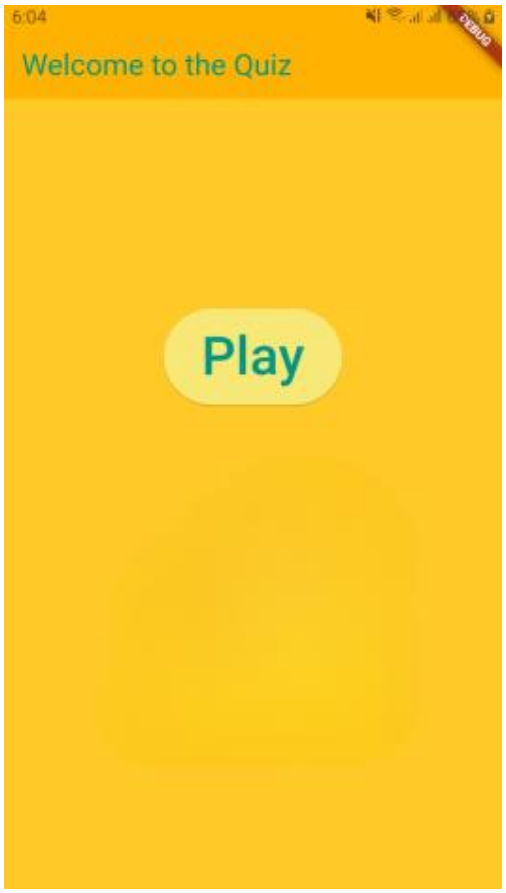
Emoji Level : Collection of different emojis *(designed by freepik)*

For the **Cartoon Level** (fourth level) , some of following emotions were chosen from the following collection:

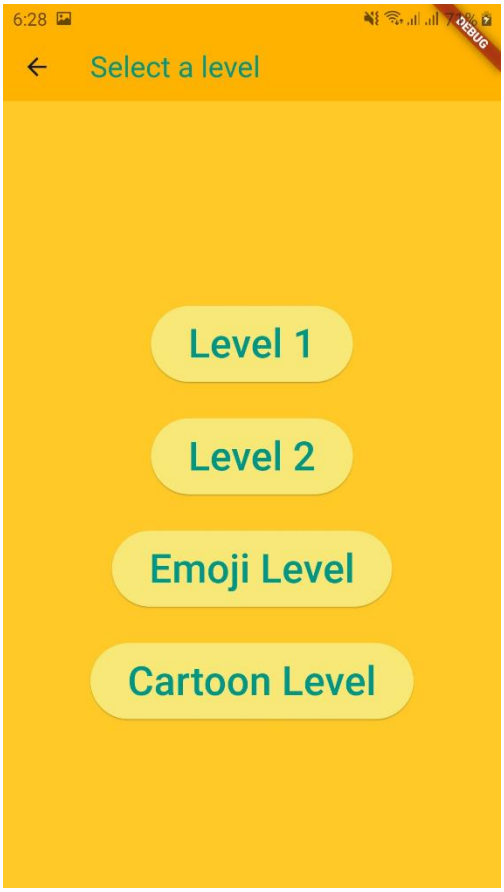
Cartoon Level : Collection of different emotions for the same cartoon boy

5.1.3 Sample run:

1. When the child chooses to play the MCQ game , they are greeted with a welcome page.



2. Then the child is prompted to choose one of the four levels



3. After Choosing a level, the child chooses the answers

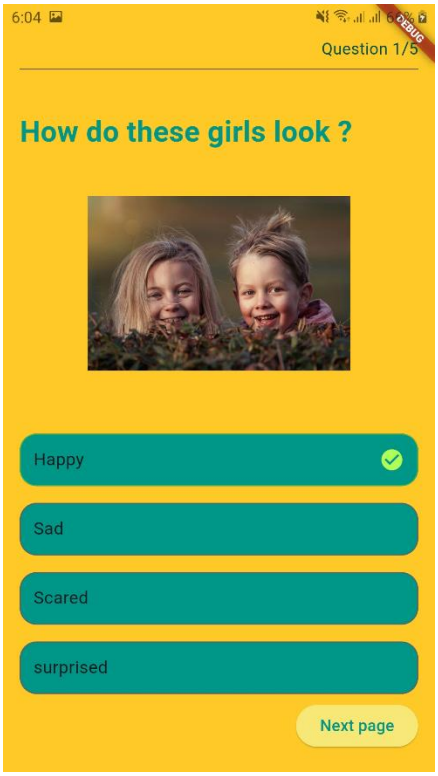


Figure 19 : Screenshot from Level 1

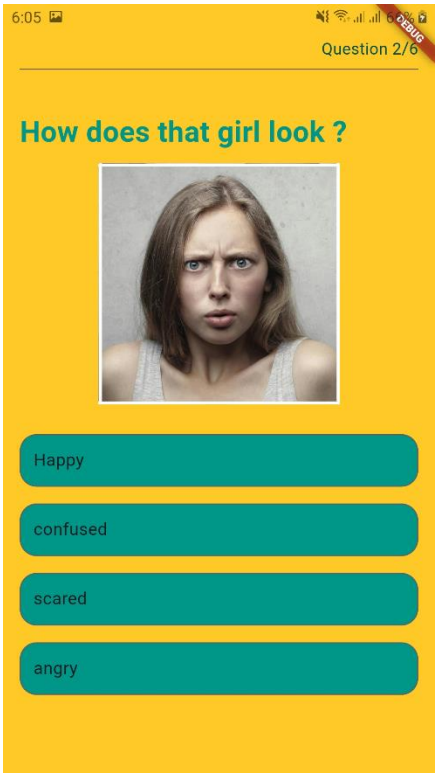


Figure 18: Screenshot from Level 2

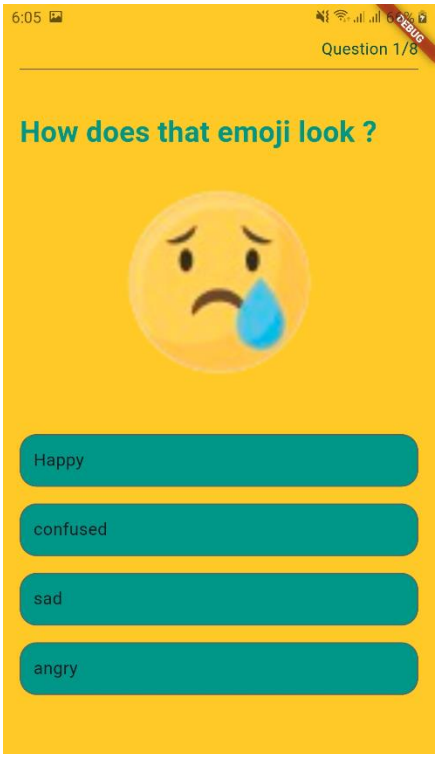


Figure 3: Screenshot from the Emoji Level

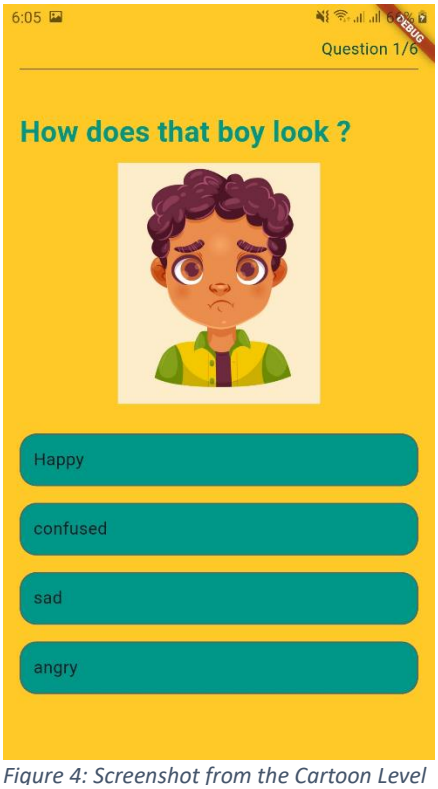
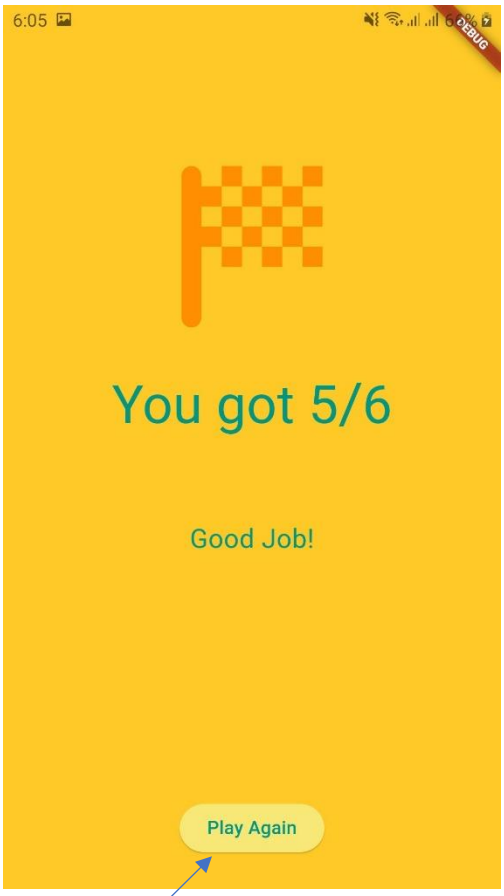


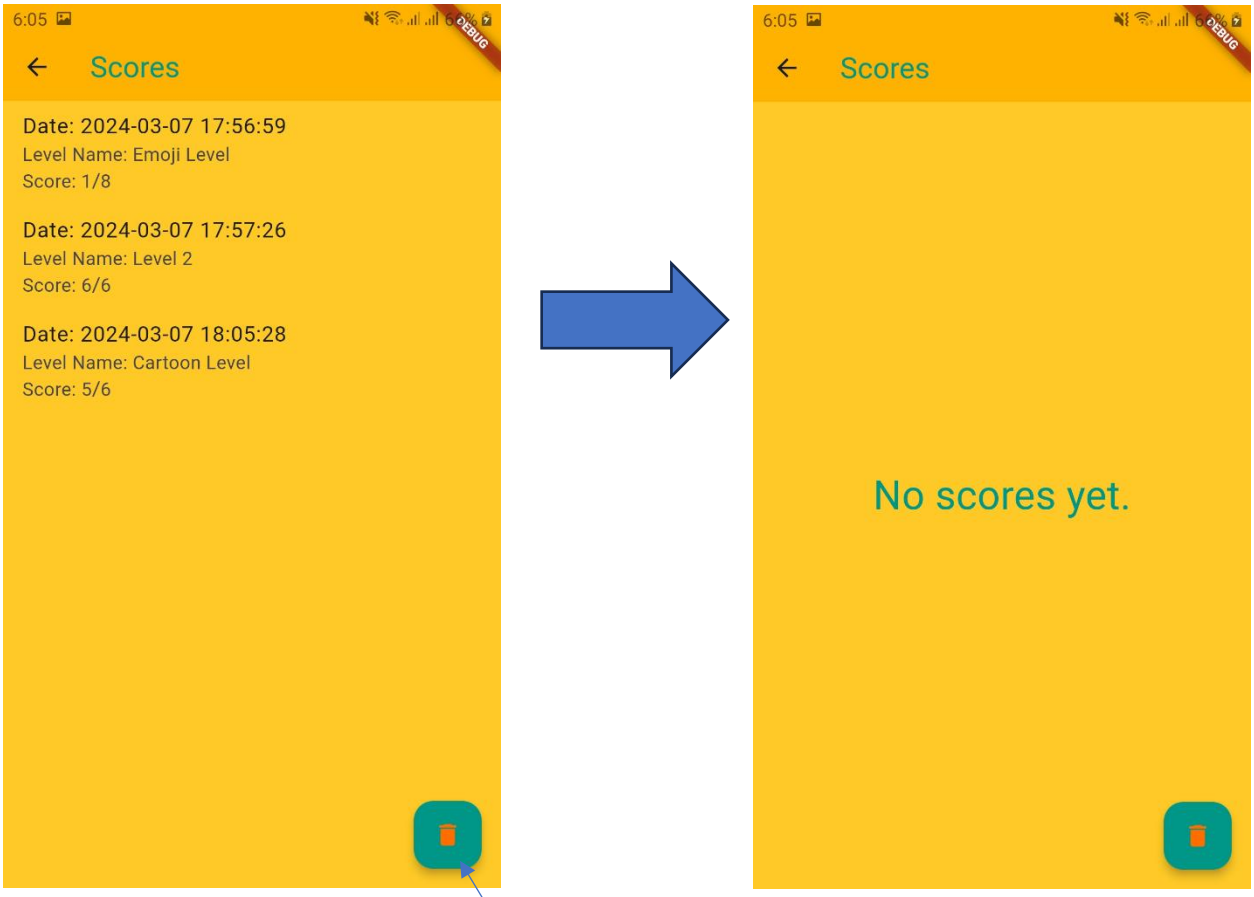
Figure 4: Screenshot from the Cartoon Level

4. The score appears to the child at the end of the game



Clicking this button will return you back to step 2 where the child chooses the desired level

5. Through the Parent Mode , the scores can be viewed/deleted



Click this button to delete previous data

5.1.4 Code Overview:

First Page and Pack Selection Page These are the initial screens of the application. The FirstPage displays a "Play" button that navigates to the PackSelectionPage when clicked. The PackSelectionPage presents four buttons representing the different levels (Level 1, Level 2, Emoji Level, and Cartoon Level). When a level is selected, the corresponding set of questions is loaded into the question_set variable, and the user is navigated to the QuestionWidget.

These pages serve as the entry point for the application and allow the user to choose the desired level of questions. The PackSelectionPage also calls the resetQuiz() function to reset the quiz state before starting a new level.

Question Widget This is the main widget responsible for displaying the questions and handling user interactions. It uses a PageView to present one question at a time and a PageController to navigate through the questions. The buildQuestion method renders the question text, image, and options. The OptionsWidget handles the display and interaction with the answer options.

The QuestionWidget is the core component that manages the quiz flow. It displays the current question number, question text, and image . The user can select an option, and the widget keeps track of the user's score and whether the question is locked (answered). It also provides a "Next" button to move to the subsequent question or navigate to the result page if it's the last question.

Options Widget This widget is responsible for rendering the answer options for a given question. It uses a SingleChildScrollView to ensure that the options are scrollable if they don't fit on the screen. The buildOption method creates a container for each option, displaying the option text and an icon (if applicable). The getColorForOption and getIconForOption methods determine the color and icon to display based on whether the option is selected and correct.

The OptionsWidget presents the answer choices for the current question. It handles the user's option selection and updates the question state accordingly. The color and icon for each option change based on whether the

user has selected it and whether it is the correct answer, providing visual feedback to the user.

Result Page This page is displayed after the user has completed all the questions in the level. It shows the final score and a message based on the user's performance. It also provides a "Play Again" button that resets the quiz and navigates back to the FirstPage. It also saves the score to a file using the saveScore function.

Question Packs The code includes four lists of questions, each representing a different level (question_pack_1, question_pack_2, question_pack_3, and question_pack_4). Each question consists of a text prompt, an image location, and a list of answer options. These questions are loaded into the question_set variable based on the selected level.

```
final question_pack_1 = [

  Question(
    text: "How do these girls look ?",
    image_location: "assets/mcq_emotions/pack1/happy.jpg",
    options:[
      const Option(text: "Happy", isCorrect: true),
      const Option(text: "Sad", isCorrect: false),
      const Option(text: "Scared", isCorrect: false),
      const Option(text: "surprised", isCorrect: false),
    ] ,
  ),
```

Example : the representation of the first question in level 1

The question packs contain the actual questions and answer options for each level. The questions are defined as instances of the Question class, which encapsulates the question text, image location, and a list of Option objects representing the answer choices.

For storing the questions , the options and solutions , the main two classes used are the *Option* and *Question* classes

1. Option Class

```dart
class Option {
  final String text;
  final bool isCorrect;

  const Option({
    required this.text,
    required this.isCorrect,
  });
}
```

The Option class represents an answer choice for a question. It has two properties: text (a string representing the option text) and isCorrect (a boolean indicating whether the option is the correct answer or not). This class is used to create instances of answer options for each question.

2. Question Class

```dart
class Question {
  final String text;
  final String image_location ;
  final List<Option> options;
  bool isLocked;
  Option? selectedOption;

  Question({
    required this.text,
    required this.options,
    required this.image_location,
    this.isLocked = false,
    this.selectedOption,
  });
}
```

The Question class represents a single question in the quiz. It has the following properties:

- text: A string representing the question text.
- image_location: A string representing the file path or asset location of the image associated with the question (if any).
- options: A list of Option objects representing the answer choices for the question.
- isLocked: A boolean indicating whether the user has answered the question or not.
- selectedOption: An optional Option object representing the answer choice selected by the user for this question.

The Question class is used to create instances of questions, each with its own text, image location, and answer options. The isLocked and selectedOption properties are used to keep track of the user's interaction with the question.

Utility Functions The code includes several utility functions, such as *getScoreMessage*, *getLevelName*, *saveScore*, and *resetQuiz*. These functions handle tasks like generating a score message based on the user's performance, retrieving the level name, saving the user's score to a file, and resetting the quiz state, respectively.

1. *getScoreMessage function*

```
String getScoreMessage(int score, int questionLength) {
  final double scorePercentage = score / questionLength;

  if (scorePercentage == 1.0) {
    return "Well Done!";
  } else if (scorePercentage >= 0.8) {
    return "Good Job!";
  } else if (scorePercentage >= 0.6) {
    return "Great!";
  } else if (scorePercentage >= 0.5) {
    return "Good!";
  } else if (scorePercentage >= 0.2) {
    return "We can do better!";
  } else {
    return "Lets Try again";
  }
}
```

This function takes the user's score and the total number of questions as input and returns a motivational message based on the user's performance. It calculates the percentage of correct answers and returns a corresponding message based on different score ranges. For example, if the user gets all questions correct, it returns "Well Done!". If the user scores between 80% and 100%, it returns "Good Job!", and so on.

2. *getLevelName* function

```
String getLevelName() {

  if (question_set == question_pack_1) {
    return "Level 1";
  } else if (question_set == question_pack_2) {
    return "Level 2";
  } else if (question_set == question_pack_3) {
    return "Emoji Level";
  } else if (question_set == question_pack_4) {
    return "Cartoon Level";
  }else{
    return "";
  }
}
```

This function returns the name of the currently selected level based on the question_set variable. It compares the question_set with the different question packs (question_pack_1, question_pack_2, question_pack_3, and question_pack_4) and returns the corresponding level name as a string. If the question_set doesn't match any of the packs, it returns an empty string.

3. saveScore function:

```dart
void saveScore(int score) async {

  // Path to a directory where the app may place data that is user-generated
  //final directory = await getApplicationDocumentsDirectory();

  try {
    final directory = await getApplicationDocumentsDirectory();
    final file = File('${directory.path}/mcq_scores.txt');

    final now = DateTime.now();
    final formattedDate = '${now.hour}:${now.minute}:${now.second} ' '' '' ' ${now.day}/${now.month}/${now.year}'; // Time followed by date
    final text = '$formattedDate%${getLevelName()}%$score/${question_set.length}\n';


    if (!await file.exists()) {
      await file.writeAsString(text);
    } else {
      await file.writeAsString(text, mode: FileMode.append);
    }
    print("\n$text\n");
    print("\nsaved to ${directory.path}\n");

  } on PlatformException catch (e) {
      print(e);
  }
}
```

This function saves the user's score to a file named mcq_scores.txt. It first retrieves the application's documents directory using getApplicationDocumentsDirectory. It then creates a File object with the file path ${directory.path}/mcq_scores.txt. The function constructs a formatted string containing the current date, time, level name, and the user's score. If the file doesn't exist, it creates a new file and writes the formatted string to it. If the file already exists, it appends the formatted string to the file. The function also prints the saved data and the file path to the console for debugging purposes.

4. resetQuiz function

```
void resetQuiz() {

  for (var question in question_set) {
    //print("resetting question");
    question.isLocked = false;
    question.selectedOption = null;
    question.selectedOption = null;
  }
}
```

The resetQuiz function is used to reset the state of the quiz when the user starts a new level or decides to play again after completing the current level.

The function iterates over the question_set list, which contains all the questions for the current level.

For each Question object in the question_set, it performs the following operations:

question.isLocked = false: This line sets the isLocked property of the Question object to false. As explained earlier, the isLocked property indicates whether the user has already answered the question or not. By setting it to false, the function ensures that all questions are unlocked and can be answered again.

5.2 Routing

Routing in `main.dart`

The `main.dart` file for the Flutter application is responsible for setting up the initial routing configuration. This setup determines how the application navigates between different screens. Below is a detailed explanation of the routing configuration in the `main.dart` file.

Imports

The file starts with importing necessary packages and files. These imports include Flutter's material package and several custom Dart files that represent different screens or features of the application.

```
import 'package:flutter/material.dart';
import 'PRorCH.dart';
import 'child/index.dart';
import 'child/ChatBot/home_page.dart';
import 'child/EDgoogleML/EDmain.dart';
import 'child/EDgoogleML/home.dart';
import 'child/mcq/mcq_model.dart';
import 'child/AR/ar_index.dart';
import 'child/AR/ar_panda.dart';
import 'child/AR/objectgesturesexample.dart';
import 'parent/index.dart';
import 'parent/mcq_history/mcq_history.dart';
import 'parent/chat_history/chat_history.dart';
```

Main Function

The `main` function is the entry point of the application. It runs the `runApp` function, which takes a `MaterialApp` widget as an argument.

```
void main() => runApp(MaterialApp(
  initialRoute: '/',
  routes: {
        '/': (context) => PRorCh(),
        '/ch_index': (context) => Chfeatures(),
        '/vchome': (context) => VChomepage(),
        '/EDchild': (context) => EDchild(),
        '/childED': (context) => childED(),
        '/mcq': (context) => FirstPage(),
        '/ar_index': (context) => ar_index(),
        '/ar_panda': (context) => ar_panda(),
        '/ar_monkey': (context) => ObjectGesturesWidget(),
        '/parent_index': (context) => PRfeatures(),
        '/chat_history': (context) => chat_history(),
        '/mcq_history': (context) => mcq_history(),
  },
));
```

MaterialApp Widget

The `MaterialApp` widget is the root of the application and is responsible for managing routes. Here are the key properties used:

- **`initialRoute`: This defines the initial route that the application loads, which is set to `'/'`.**
- **`routes`: A map that defines the available routes in the application. Each key is a string representing the route's name, and the value is a function that returns the widget associated with that route.**

Defined Routes

Here is a breakdown of each defined route:

1. **`'/': (context) => PRorCh()`**
    - **This is the initial route, loading the `PRorCh` widget.**
2. **`'/ch_index': (context) => Chfeatures()`**
    - **Loads the `Chfeatures` widget, which is located inside the `child` folder.**
3. **`'/vchome': (context) => VChomepage()`**
    - **Loads the `VChomepage` widget, which is the home page for the chatbot.**
4. **`'/EDchild': (context) => EDchild()`**
    - **Loads the `EDchild` widget, which is the main screen for the child emotion detector feature.**
5. **`'/childED': (context) => childED()`**
    - **Loads the `childED` widget, another screen related to the child emotion detector.**
6. **`'/mcq': (context) => FirstPage()`**
    - **Loads the `FirstPage` widget, which is the starting screen for the MCQ (Multiple Choice Question) test.**
7. **`'/ar_index': (context) => ar_index()`**
    - **Loads the `ar_index` widget, an index screen for AR (Augmented Reality) features.**

8. `'/ar_panda': (context) => ar_panda()`
    - Loads the `ar_panda` widget, an AR feature focused on a panda model.
9. `'/ar_monkey': (context) => ObjectGesturesWidget()`
    - Loads the `ObjectGesturesWidget` widget, another AR feature likely involving object gestures.
10. `'/parent_index': (context) => PRfeatures()`
    - Loads the `PRfeatures` widget, the index screen for parent-related features.
11. `'/chat_history': (context) => chat_history()`
    - Loads the `chat_history` widget, displaying the chat history.
12. `'/mcq_history': (context) => mcq_history()`
    - Loads the `mcq_history` widget, displaying the MCQ history.

Routing Explanation in `PRorCh` Widget

The `PRorCh` widget in the provided code serves as the mode selection screen of the Flutter application. It allows users to navigate to either the "Play Time" or "Parent Mode" sections of the application. Here's a detailed explanation of how the routing works in this widget:

Widget Structure

1. **Imports and Class Definition:**
   - The file begins with importing the necessary Flutter package.
   - The `PRorCh` class is defined as a stateful widget, which will handle the state changes when users interact with it.
2. **State Class:**
   - The `_PRorChState` class manages the state of the `PRorCh` widget.
   - It overrides the `build` method to create the UI.
3. **UI Elements:**
   - **Scaffold:**
     - The `Scaffold` widget provides the basic structure, including an `AppBar` and a `body`.
   - **AppBar:**
     - The `AppBar` has a title "Mode Selection" and a blue background color.
   - **Body:**
     - The body of the `Scaffold` is a `Container` with a background image (`bgImg`).
     - Inside the `Container`, a `Column` is used to arrange the elements vertically.
     - The `Column` contains a `Spacer` and a `Row` with two buttons for navigation.

4. **Navigation Buttons:**
    - **Play Time Button:**

```
ElevatedButton(
  onPressed: () async{
        Navigator.pushNamed(context, '/ch_index');
  },
  style: ElevatedButton.styleFrom(backgroundColor:
Colors.blueAccent,),
  child: const Padding(
        padding: EdgeInsets.all(10.0),
        child: Text("Play Time", style: TextStyle(color: Colors.black,
fontSize: 24), ),
  ),
),
```

- **When pressed, this button triggers a navigation event using `Navigator.pushNamed`.**
- **It navigates to the route defined by `'/ch_index'`, which corresponds to the `Chfeatures` widget.**

Parent Mode Button:

```
ElevatedButton(
  onPressed: () async{
        Navigator.pushNamed(context, '/parent_index');
  },
  style: ElevatedButton.styleFrom(backgroundColor: Colors.blueAccent,),
  child: const Padding(
        padding: EdgeInsets.all(10.0),
        child: Text("Parent Mode", style: TextStyle(color: Colors.black, fontSize:
24), ),
  ),
),
```

- **When pressed, this button also triggers a navigation event using `Navigator.pushNamed`.**
- **It navigates to the route defined by `'/parent_index'`, which corresponds to the `PRfeatures` widget.**

Routing Explanation in `Chfeatures` Widget

The `Chfeatures` widget provides a central hub for navigating to various child-related features within the application. This widget allows users to choose between different functionalities such as a Chat Bot, Emotion Detector, MCQ Quiz, and Augmented Reality (AR). Here's a detailed explanation of how the routing works in this widget:

Widget Structure

1. **Imports and Class Definition:**
   - The file begins with importing the necessary Flutter package.
   - The `Chfeatures` class is defined as a stateful widget, which will handle state changes when users interact with it.
2. **State Class:**
   - The `_ChfeaturesState` class manages the state of the `Chfeatures` widget.
   - It overrides the `build` method to create the UI.
3. **UI Elements:**
   - **Scaffold:**
     - The `Scaffold` widget provides the basic structure of the app, including an `AppBar` and a `body`.
   - **AppBar:**
     - The `AppBar` has a blue background color, giving a consistent look and feel with the rest of the application.
   - **Body:**
     - The body of the `Scaffold` contains an `Align` widget to center its child elements.
     - Inside the `Align` widget, a `Column` widget is used to arrange the buttons vertically.

4. **Navigation Buttons:**
    - **Each button is designed to navigate to a different feature within the child mode. The buttons use the `Navigator.pushNamed` method to handle the navigation.**
    - **Chat Bot Button:**

```
ElevatedButton(
  onPressed: () {
        Navigator.pushNamed(context, '/vchome');
  },
  style: ElevatedButton.styleFrom(backgroundColor:
Colors.blueAccent,),
  child: const Padding(
        padding: EdgeInsets.all(10),
        child: Text("Chat Bot", style: TextStyle(color: Colors.black,
fontSize: 24,),)
  ),
),
```

- **When pressed, this button navigates to the `'/vchome'` route, which corresponds to the `VChomepage` widget.**

Emotion Detector Button:

```
ElevatedButton(
  onPressed: () {
        Navigator.pushNamed(context, '/EDchild');
  },
  style: ElevatedButton.styleFrom(backgroundColor: Colors.blueAccent,),
  child: const Padding(
        padding: EdgeInsets.all(10),
        child: Text("Emotion Detector", style: TextStyle(color: Colors.black,
fontSize: 24,),)
  ),
),
```

When pressed, this button navigates to the `'/EDchild'` route, which corresponds to the `EDchild` widget.

Summary of Routing

- **Navigating to Chat Bot (`/vchome`):**
  - **The "Chat Bot" button navigates the user to the `VChomepage` screen by pushing the `'/vchome'` route onto the navigation stack.**
- **Navigating to Emotion Detector (`/EDchild`):**
  - **The "Emotion Detector" button navigates the user to the `EDchild` screen by pushing the `'/EDchild'` route onto the navigation stack.**
- **Navigating to MCQ Quiz (`/mcq`):**
  - **The "MCQ Quiz" button navigates the user to the `FirstPage` screen by pushing the `'/mcq'` route onto the navigation stack.**
- **Navigating to AR (`/ar_index`):**
  - **The "AR" button navigates the user to the `ar_index` screen by pushing the `'/ar_index'` route onto the navigation stack.**

5.3 Chat bot implementations

(openai_service.dart + home_page.dart)

Feature Description of openai_service.dart

Overview

The openai_service.dart file defines a service class, OpenAIService, designed to interact with the OpenAI API. This service processes user prompts to determine the type of response required (text or image) and handles the communication with the API accordingly. The service is tailored to assist autistic children by providing friendly and helpful interactions.

Key Features

1. **Initialization of Messages:**
    - The service initializes with a predefined system message that establishes the assistant's role as friendly helpers named Fuzzy and Fizzle. These assistants are designed to help children with autism learn emotions and answer questions.
2. **API Interaction:**
    - **isArtPromptAPI(String prompt):**
        - This method takes a user prompt and sends it to OpenAI's Chat API to determine if the prompt requires generating an image.
        - The response from the API is analyzed to check if the prompt starts with keywords like "draw" or "image" or if the API response indicates an image generation request.
        - Depending on the API's response, the method either calls dallEAPI(prompt) for image generation or chatGPTAPI(prompt) for text responses.
        - If there is an error or the response status code is not 200, it returns an error message.
    - **chatGPTAPI(String prompt):**
        - This method adds the user prompt to the list of messages and sends it to OpenAI's Chat API.
        - It processes the API response and extracts the assistant's reply, which is then returned as a string.
    - **dallEAPI(String prompt):**
        - Similar to chatGPTAPI, this method prepares a request to OpenAI's DALL-E API for image generation.
        - It sends the prompt to the DALL-E API, retrieves the image URL from the response, and returns it.
3. **Error Handling:**
    - The methods within the class include error handling mechanisms to catch exceptions and return relevant error messages.

4. **Context Maintenance:**

    - The service maintains a list of messages to keep track of the conversation context. This is essential for generating coherent and contextually appropriate responses.

Feature Description of home_page.dart

Overview

The home_page.dart file defines the user interface for interacting with the OpenAIService. It includes the implementation of a voice-controlled input system using speech-to-text and text-to-speech services, making it accessible and user-friendly, especially for children.

Key Features

1. **UI Structure:**
    - The main UI is built using Flutter's Scaffold, AppBar, and Body widgets, creating a visually structured and navigable interface.
2. **Voice-Controlled Interaction:**
    - **Speech-to-Text Integration:**
        - The app uses a speech-to-text service to allow users to input prompts via voice commands.
        - The startListening and initSpeechToText methods handle the initialization and listening states of the speech-to-text service.
        - When the service is listening, the speechtoGPT method is triggered to process the spoken input.
    - **Text-to-Speech Integration:**
        - After generating a response (text or image URL), the app uses text-to-speech to read the response aloud.
        - This enhances accessibility and engagement for users, particularly children with autism.
3. **Interactive Text Input:**
    - In addition to voice commands, the app provides a text input field with a TextEditingController to capture user prompts.
    - The input field is equipped with a microphone button to toggle speech recognition and an arrow button to manually submit text input.
4. **State Management:**
    - The app uses Flutter's StatefulWidget to manage the state of the input field, speech recognition status, and response display.
    - State updates trigger visual changes in the UI, providing real-time feedback to users.
5. **Prompt Handling and Response Display:**
    - The user's voice or text input is processed and sent to the OpenAIService.
    - Responses (text or image URLs) are displayed in the UI, and images are rendered using Flutter's Image.network widget.
6. **Local Data Storage:**
    - The saveQuestion function saves the interaction history (questions and answers) to a local file, allowing users to review past interactions.
    - The method retrieves the application's document directory, formats the data, and appends it to a text file.

7. **Error Handling and User Feedback:**
    - The app provides visual and auditory feedback for different states (e.g., listening, processing, and displaying responses).
    - Error handling mechanisms ensure that any issues during speech recognition or API interaction are gracefully managed and communicated to the user.

Materials used in the feature:

2. OpenAI API

- **GPT-3.5 Turbo Model:**
  - The `isArtPromptAPI`, `chatGPTAPI`, and `dallEAPI` methods in `openai_service.dart` utilize OpenAI's GPT-3.5 turbo model for natural language processing. This model is capable of understanding and generating human-like text, making it suitable for creating conversational agents.
- **DALL-E API:**
  - The `dallEAPI` method uses the DALL-E API for image generation based on text prompts. This API allows the application to generate and display images as responses, enhancing the interactivity and engagement of the app.

3. HTTP Package

- **Package: `http`**
  - The `http` package is used to make HTTP requests to the OpenAI API endpoints. It handles the network communication required to send user prompts and receive responses.

4. JSON Encoding/Decoding

- **Dart `dart:convert` Library:**
  - This library is used for encoding and decoding JSON data. It is crucial for formatting the data sent to and received from the OpenAI API, ensuring proper communication and data integrity.

5. Speech-to-Text and Text-to-Speech Services

- **Speech Recognition:**
  - The app integrates speech recognition services to convert spoken language into text. This feature is implemented using methods like `startListening` and `initSpeechToText` in `home_page.dart`.
- **Text-to-Speech:**
  - The app uses text-to-speech services to read responses aloud to the user. This is particularly beneficial for young users or those with reading difficulties, providing an auditory feedback loop.

6. State Management

- **StatefulWidget:**
  - Flutter's `StatefulWidget` is employed to manage the state of the application, including the status of speech recognition, user input, and response display. This ensures that the UI updates in real-time based on user interactions and received data.

7. Local Storage

- **File I/O:**
  - The `saveQuestion` method demonstrates the use of file I/O to store chat history locally. This involves writing data to a text file in the device's document directory, allowing users to retain and review their interaction history.

8. Secret Management

- **Secrets File (`secrets.dart`):**
  - API keys and other sensitive information are stored in a separate `secrets.dart` file. This practice ensures that such data is kept secure and is not hard-coded into the main application files.

9. Flutter Widgets

- **UI Widgets:**

A variety of Flutter widgets are used to build the user interface, including `Scaffold`, `AppBar`, `TextField`, `IconButton`, and `Image.network`. These widgets provide the structural and interactive elements needed for a functional and engaging UI.

Code Overview of `openai_service.dart`

The `openai_service.dart` file contains the implementation of the `OpenAIService` class, which is responsible for interacting with the OpenAI API. It processes user prompts, determines the appropriate response type (text or image), and handles communication with the OpenAI API to generate responses.

Key Components

1. **Imports:**

```
import 'dart:convert';
import 'secrets.dart';
import 'package:http/http.dart' as http;
```

- `dart:convert` for JSON encoding/decoding.
- `secrets.dart` for API key storage.
- `http` package for making HTTP requests.

2. **Class: OpenAIService**

Properties:

```
final List<Map<String, String>> messages = [
{
'role': 'system',
'content': 'You are friendly helpers to an autistic child, you are named Fuzzy and Fizzle, you can draw any image and help children with autism learn emotions and answer any question asked.'
}
];
```

`messages`: A list to store conversation history with a predefined system message setting the assistant's context.

Methods:

- **isArtPromptAPI(String prompt):**

```
Future<String> isArtPromptAPI(String prompt) async {
 try {
  final res = await http.post(
   Uri.parse('https://api.openai.com/v1/chat/completions'),
   headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer $openAIAPIKey',
   },
   body: jsonEncode({
    "model": "gpt-3.5-turbo",
    "messages": [
     {
      'role': 'system',
      'content': 'read this request ( $prompt ). Does this request want you to draw or generate an image ? answer with yes or no.',
     }
    ],
   }),
  );
  print(res.body);
  if (res.statusCode == 200) {
   String content = jsonDecode(res.body)['choices'][0]['message']['content'];
   content = content.trim();

   if (prompt.startsWith("draw") || prompt.startsWith("image")) {
    final res = await dallEAPI(prompt);
    return res;
   }

   switch (content) {
    case 'Yes':
    case 'yes':
    case 'Yes.':
    case 'yes.':
     final res = await dallEAPI(prompt);
     return res;
    default:
     final res = await chatGPTAPI(prompt);
     return res;
   }
  }
  return 'An internal error occurred in the open AI';
 } catch (e) {
  return e.toString();
 }
}
```

Determines if the prompt is asking for an image and calls the appropriate API method (`dallEAPI` or `chatGPTAPI`).

- **chatGPTAPI(String prompt):**

```
Future<String> chatGPTAPI(String prompt) async {
  messages.add({
    'role': 'user',
    'content': prompt
  });

  try {
    final res = await http.post(
      Uri.parse('https://api.openai.com/v1/chat/completions'),
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer $openAIAPIKey',
      },
      body: jsonEncode({
        "model": "gpt-3.5-turbo",
        "messages": messages,
      }),
    );
    if (res.statusCode == 200) {
      String content = jsonDecode(res.body)['choices'][0]['message']['content'];
      messages.add({
        'role': 'assistant',
        'content': content
      });
      return content;
    }
    return 'An internal error occurred';
  } catch (e) {
    return e.toString();
  }
}
```

Sends a user prompt to OpenAI's Chat API and returns the assistant's response.

- **dallEAPI(String prompt):**

```
Future<String> dallEAPI(String prompt) async {
  messages.add({
    'role': 'user',
    'content': prompt
  });

  try {
    final res = await http.post(
      Uri.parse('https://api.openai.com/v1/images/generations'),
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer $openAIAPIKey',
      },
      body: jsonEncode({
        "model": "image-alpha-001",
        "prompt": prompt,
      }),
    );
    if (res.statusCode == 200) {
      String imageUrl = jsonDecode(res.body)['data'][0]['url'];
      messages.add({
        'role': 'assistant',
        'content': imageUrl
      });
      return imageUrl;
    }
    return 'An internal error occurred';
  } catch (e) {
    return e.toString();
  }
}
```

Sends a prompt to OpenAI's DALL-E API for image generation and returns the image URL.

Code Overview of `home_page.dart`

The `home_page.dart` file defines the user interface and functionality for interacting with the `OpenAIService`. It includes voice-controlled input using speech-to-text services, text-to-speech responses, and text input options for user prompts.

Key Components

1. **Imports:**

```
import 'package:flutter/material.dart';
import 'package:your_project_name/openai_service.dart'; // Import the OpenAI service
import 'package:speech_to_text/speech_to_text.dart' as stt;
import 'package:flutter_tts/flutter_tts.dart';
import 'dart:async';
import 'package:path_provider/path_provider.dart';
import 'dart:io';
```

- Flutter material package for UI components.
- OpenAI service for API interactions.
- `speech_to_text` and `flutter_tts` for speech recognition and text-to-speech functionality.
- `path_provider` and `dart:io` for local file storage.

2. **Class: HomePage**

```
class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}
```

Stateful widget to manage UI state.

- **Initialization:**

```
class _HomePageState extends State<HomePage> {
  stt.SpeechToText _speech;
  bool _isListening = false;
  String _text = "Press the button and start speaking";
  double _confidence = 1.0;
  FlutterTts _flutterTts;
  OpenAIService openAIService = OpenAIService();
}
```

Initializes speech-to-text, text-to-speech, and OpenAI service instances.

- **Speech-to-Text Methods:**

```
void _listen() async {
  if (!_isListening) {
    bool available = await _speech.initialize(
      onStatus: (val) => print('onStatus: $val'),
      onError: (val) => print('onError: $val'),
    );
    if (available) {
      setState(() => _isListening = true);
      _speech.listen(
        onResult: (val) => setState(() {
          _text = val.recognizedWords;
          if (val.hasConfidenceRating && val.confidence > 0) {
            _confidence = val.confidence;
          }
        }),
      );
    }
  } else {
    setState(() => _isListening = false);
    _speech.stop();
  }
}
```

Handles speech recognition and updates the UI with recognized words.

- **Text-to-Speech Methods:**

```
void _speak(String text) async {
  await _flutterTts.speak(text);
}
```

Converts text responses to speech for auditory feedback.

- **UI Build Method:**

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Voice Assistant'),
    ),
    body: Column(
      children: <Widget>[
        Text(
          'Confidence: ${(_confidence * 100.0).toStringAsFixed(1)}%',
        ),
        Expanded(
          child: Container(
            padding: const EdgeInsets.all(16),
            child: Text(
              _text,
              style: const TextStyle(
                fontSize: 32.0,
                color: Colors.black,
                fontWeight: FontWeight.w400,
              ),
            ),
          ),
        ),
        FloatingActionButton(
          onPressed: _listen,
          child: Icon(_isListening ? Icons.mic : Icons.mic_none),
        ),
      ],
    ),
  );
}
```

Builds the UI with a floating action button to toggle listening, a text display for recognized words, and a confidence indicator.

- **Local Storage:**

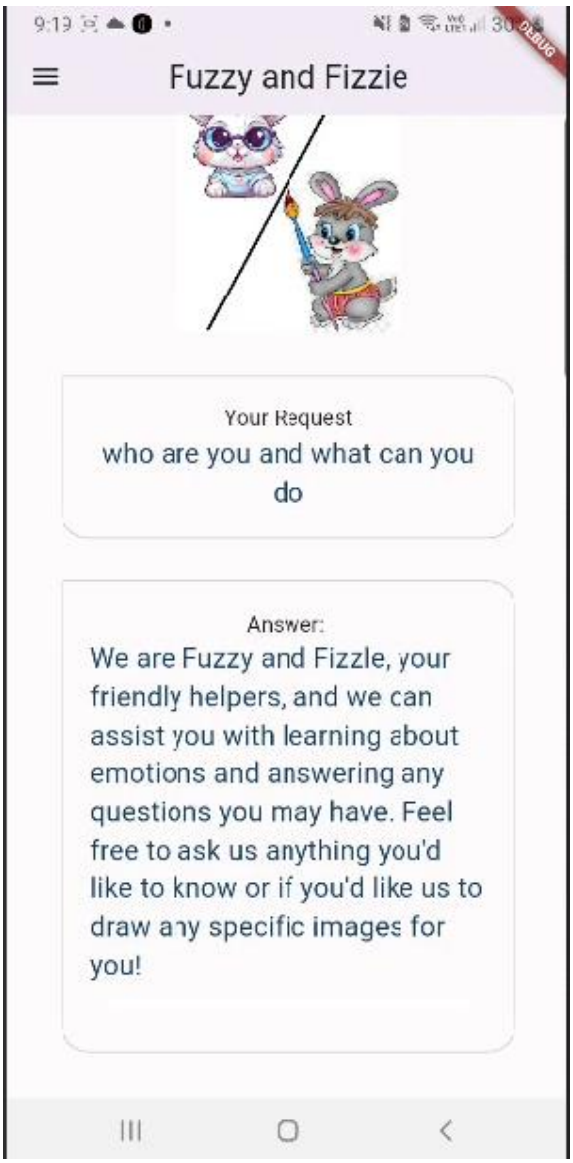
```
void saveQuestion(String question, String answer) async {
  try {
    final directory = await getApplicationDocumentsDirectory();
    final file = File('${directory.path}/chat_history.txt');

    final now = DateTime.now();
    final formattedDate = '${now.hour}:${now.minute}:${now.second}
${now.day}/${now.month}/${now.year}';
    final text = '$formattedDate*$question*$answer\n';

    if (!await file.exists()) {
      await file.writeAsString(text);
    } else {
      await file.writeAsString(text, mode: FileMode.append);
    }
    print("\n$text\n");
    print("\nsaved to ${directory.path}\n");
  } on PlatformException catch (e) {
    print(e);
  }
}
```

Saves questions and answers to a local file for interaction history.

Sample runs:


Fuzzy and Fizzie
Your Request
who are you and what can you do
Answer:
We are Fuzzy and Fizzle, your friendly helpers, and we can assist you with learning about emotions and answering any questions you may have. Feel free to ask us anything you'd like to know or if you'd like us to draw any specific images for you!


Fuzzy Fizzie
Your Request
image of a person smiling
Answer:

5.4 AR Feature

Feature Description

`objectgesturesexample.dart`

The `objectgesturesexample.dart` file showcases an AR (Augmented Reality) application feature that enables users to interact with and manipulate 3D objects using touch gestures. This feature is implemented through a Flutter widget called `ObjectGesturesWidget`, which integrates with AR session, object, and anchor managers to provide a seamless AR experience. Key functionalities include:

- **Mood Management:**
  - The application allows users to switch between different predefined moods. Each mood is represented by a `Mood` class containing a value and display text. A dropdown menu facilitates easy selection of these moods.
- **AR Object Placement and Manipulation:**
  - Users can place AR objects into the scene and interact with them using touch gestures. The `ARView` widget is central to rendering the AR content and capturing user interactions.
- **Object Removal:**
  - The feature includes functionality to remove all AR objects from the scene with a single button click, providing an easy way to reset the AR environment.
- **Session Management:**
  - The application manages AR sessions through `arSessionManager`, `arObjectManager`, and `arAnchorManager`, ensuring proper initialization and disposal of AR resources.
- **User Interface:**
  - The interface consists of a scaffold with an app bar and a stack layout containing the AR view and interactive buttons for object manipulation and mood selection.

`ar_panda.dart`

The `ar_panda.dart` file describes an AR feature focused on displaying and managing 3D panda models representing various moods. This functionality is encapsulated within a stateful widget, providing the following key features:

- **Panda Mood Models:**
  - The application supports multiple panda models, each corresponding to a different mood (e.g., excited, laughing, sad, crying, mad). These models are loaded and displayed in the AR scene based on user interaction.

- **Dynamic Node Management:**
    - The feature includes methods for adding and removing panda models dynamically. Each mood-specific method checks for the presence of the corresponding node and either adds or removes it as necessary.
- **Complete Node Removal:**
    - A method `_removeAllNodes` is provided to remove all panda nodes from the AR session, allowing for a complete reset of the AR environment.
- **User Interaction:**
    - Users can switch between different panda moods using the provided interface, which updates the AR scene accordingly. This interaction is managed through buttons and event handlers that modify the AR content.
- **Session and Object Handling:**

Similar to the previous feature, this application manages AR sessions and objects using `arSessionManager` and `arObjectManager`. Proper handling of AR resources ensures smooth operation and clean disposal of objects.

Materials Used in the Feature

`objectgesturesexample.dart`

The `objectgesturesexample.dart` file incorporates several key materials to implement its AR features. These materials are crucial in providing a seamless and interactive user experience. Here are the primary materials used:

1. **Flutter Framework:**
    - **Widgets:**
        - `StatefulWidget` and `State`: Fundamental to creating interactive UI components.
        - `Scaffold`: Provides the basic structure for the application's UI, including an app bar and a body for content display.
        - `Stack`: Allows layering of UI elements, enabling the overlay of AR content and control buttons.
        - `Align`, `Row`, `ElevatedButton`, `DropdownButton`: Used for positioning and creating interactive controls within the UI.
2. **AR Flutter Plugin:**
    - **ARView**: Core widget for rendering AR content and capturing user interactions within the AR scene.
    - **ARSessionManager**: Manages the AR session, handling initialization and disposal of AR resources.
    - **ARObjectManager**: Handles the addition, removal, and management of AR objects within the scene.
    - **ARAnchorManager**: Manages anchors, which are reference points for placing AR objects in the real world.
3. **3D Models and Assets:**
    - While the specific 3D models and assets are not detailed in this file, it is implied that various 3D objects are used to represent different moods. These models are likely stored in local assets and referenced within the application.
4. **Vector Math Library:**
    - **Vector3**: Used for defining the position and scale of AR objects.

Vector4: Used for defining the rotation of AR objects, ensuring proper orientation within the scene.

`ar_panda.dart`

The `ar_panda.dart` file utilizes similar materials but focuses specifically on managing different 3D panda models. Here are the primary materials used:

1. **Flutter Framework:**
    - **Widgets:**
        - `StatefulWidget` and `State`: Core to managing the state of the application and AR interactions.
2. **AR Flutter Plugin:**
    - **ARView**: Central widget for displaying AR content and handling user input.
    - **ARSessionManager**: Manages the lifecycle of the AR session.
    - **ARObjectManager**: Responsible for adding, removing, and managing AR objects within the scene.
3. **3D Panda Models:**
    - **Local GLTF2 Models:**
        - The application uses GLTF2 models stored locally to represent different panda moods (e.g., excited, laughing, sad, crying, mad).
        - These models are referenced using URIs pointing to the local assets directory (e.g., `"assets/Models/panda_excited/panda_excited.gltf"`).
    - **NodeType.localGLTF2**: Specifies the type of 3D model being used, ensuring compatibility with the AR plugin.
4. **Vector Math Library:**
    - **Vector3**: Utilized for defining the scale and position of the panda models within the AR scene.
    - **Vector4**: Utilized for defining the rotation of the panda models, ensuring correct orientation.
5. **Asset Management:**
    - The application uses a structured approach to manage and reference local 3D models, ensuring that each mood is represented by a corresponding 3D panda model.

Code Overview

`objectgesturesexample.dart`

The `objectgesturesexample.dart` file is structured to create an interactive AR experience where users can manipulate 3D objects using gestures. Here's a detailed overview of its sections:

1. **Imports:**
    - Essential libraries and packages are imported at the beginning, including `package:flutter/material.dart` for Flutter components, `package:ar_flutter_plugin` for AR functionalities, and `package:vector_math/vector_math_64.dart` for vector math operations.
2. **Mood Class and List:**
    - **Mood Class:**

```
class Mood {
  final int value;
  final String displayText;

  Mood(this.value, this.displayText);
}
```

Defines a class to represent different moods with a value and display • text.

- **List of Moods:**

```
List<Mood> moods = [
  Mood(0, 'Happy'),
  Mood(1, 'Laughing'),
  Mood(2, 'Afraid'),
  Mood(3, 'Cry'),
  Mood(4, 'Mad'),
];
```

Initializes a list of `Mood` objects representing various emotional states

- **ObjectGesturesWidget:**
    - **StatefulWidget:**

```
class ObjectGesturesWidget extends StatefulWidget {
  ObjectGesturesWidget({Key? key}) : super(key: key);
  @override
  _ObjectGesturesWidgetState createState() => _ObjectGesturesWidgetState();
}
```

A stateful widget that serves as the main interface for AR object manipulation.

- **State Management:**

```
class _ObjectGesturesWidgetState extends State<ObjectGesturesWidget> {
  ARSessionManager? arSessionManager;
  ARObjectManager? arObjectManager;
  ARAnchorManager? arAnchorManager;

  int moodindex = 0;
  List<ARNode> nodes = [];
  List<ARAnchor> anchors = [];
```

Manages the state of the AR session, object, and anchor managers, along with a list of nodes and anchors

- **Dispose Method:**

```
@override
void dispose() {
  super.dispose();
  arSessionManager!.dispose();
}
```

Ensures proper disposal of AR session resources when the widget is removed from the tree.

Widget Build Method:

- **Build Method:**

```
@override
Widget build(BuildContext context) {
  return Scaffold(
      appBar: AppBar(
        title: const Text('Object Transformation Gestures'),
      ),
      body: Container(
          child: Stack(children: [
            ARView(
              onARViewCreated: onARViewCreated,
              planeDetectionConfig: PlaneDetectionConfig.horizontalAndVertical,
            ),
            Align(
              alignment: FractionalOffset.bottomCenter,
              child: Row(
                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
                  children: [
                    ElevatedButton(
                        onPressed: onRemoveEverything,
                        child: Text("Remove Everything")),
                    DropdownButton<int>(
                        value: moodindex,
                        items: moods.map((Mood mood) {
                          return DropdownMenuItem<int>(
                            value: mood.value,
                            child: Text(mood.displayText),
                          );
                        }).toList(),
                        onChanged: (int? newValue) {
                          setState(() {
                            moodindex = newValue!;
                          });
                        }),
                  ]),
            )
          ])));
}
```

Builds the UI, including an AR view, buttons for removing objects, and a dropdown for selecting moods.

AR View Initialization:

- **onARViewCreated Method**

```
void onARViewCreated(ARSessionManager arSessionManager, ARObjectManager arObjectManager,
ARAnchorManager arAnchorManager) {
  this.arSessionManager = arSessionManager;
  this.arObjectManager = arObjectManager;
  this.arAnchorManager = arAnchorManager;
}
```

Initializes AR session, object, and anchor managers when the AR view is created.

Object Removal Method:

- **onRemoveEverything Method:**

```
void onRemoveEverything() {
  arObjectManager!.removeAllNodes();
  arAnchorManager!.removeAllAnchors();
}
```

Removes all AR objects and anchors from the scene.

ar_panda.dart

The `ar_panda.dart` file is focused on managing different 3D panda models representing various moods within an AR environment. Here's a detailed overview of its sections:

1. **Imports:**
    - Essential libraries and packages are imported at the beginning, including `package:flutter/material.dart` for Flutter components and `package:ar_flutter_plugin` for AR functionalities.
2. **AR View Widget:**
    - **StatefulWidget:**

```
class ARViewWidget extends StatefulWidget {
  ARViewWidget({Key? key}) : super(key: key);
  @override
  _ARViewWidgetState createState() => _ARViewWidgetState();
}
```

A stateful widget that serves as the main interface for displaying and managing 3D panda models.

State Management:

- **State Class:**

```
class _ARViewWidgetState extends State<ARViewWidget> {
  ARSessionManager? arSessionManager;
  ARObjectManager? arObjectManager;
  ARAnchorManager? arAnchorManager;

  ARNode? localObjectNodepanda_excited;
  ARNode? localObjectNodepanda_laugh;
  ARNode? localObjectNodepanda_sad;
  ARNode? localObjectNodepanda_cry;
  ARNode? localObjectNodepanda_mad;
```

Manages the state of the AR session and various panda model nodes.

Widget Build Method:

- **Build Method:**

```
@override
Widget build(BuildContext context) {
 return Scaffold(
    appBar: AppBar(
     title: const Text('AR Panda'),
    ),
    body: Container(
       child: Stack(children: [
        ARView(
         onARViewCreated: onARViewCreated,
         planeDetectionConfig: PlaneDetectionConfig.horizontalAndVertical,
        ),
        Align(
         alignment: FractionalOffset.bottomCenter,
         child: Column(
            mainAxisAlignment: MainAxisAlignment.end,
            children: [
             ElevatedButton(
                onPressed: _removeAllNodes,
                child: Text("Remove All Pandas")),
             ElevatedButton(
                onPressed: panda_excited,
                child: Text("Show Excited Panda")),
             ElevatedButton(
                onPressed: panda_laugh,
                child: Text("Show Laughing Panda")),
             ElevatedButton(
                onPressed: panda_sad,
                child: Text("Show Sad Panda")),
             ElevatedButton(
                onPressed: panda_cry,
                child: Text("Show Crying Panda")),
             ElevatedButton(
                onPressed: panda_mad,
                child: Text("Show Mad Panda")),
            ]),
        )
       ])));
}
```

Builds the UI, including an AR view and buttons for displaying and removing various panda models.

AR View Initialization:

- **onARViewCreated Method:**

```
void onARViewCreated(ARSessionManager arSessionManager, ARObjectManager arObjectManager,
ARAnchorManager arAnchorManager) {
  this.arSessionManager = arSessionManager;
  this.arObjectManager = arObjectManager;
  this.arAnchorManager = arAnchorManager;
}
```

Initializes AR session, object, and anchor managers when the AR view is created.

Panda Mood Methods:

- Methods for managing different panda moods, such as `panda_excited`, `panda_laugh`, `panda_sad`, `panda_cry`, and `panda_mad`. Each method checks if the corresponding node exists, removes it if it does, or creates and adds a new node if it doesn't.

```
Future<void> panda_excited() async {
  if (this.localObjectNodepanda_excited != null) {
    this.arObjectManager!.removeNode(this.localObjectNodepanda_excited!);
    this.localObjectNodepanda_excited = null;
  } else {
    var newNode = ARNode(
        type: NodeType.localGLTF2,
        uri: "assets/Models/panda_excited/panda_excited.gltf",
        scale: Vector3(0.2, 0.2, 0.2),
        position: Vector3(0.0, 0.0, 0.0),
        rotation: Vector4(1.0, 0.0, 0.0, 0.0));
    bool? didAddLocalNodepanda_excited =
        await this.arObjectManager!.addNode(newNode);
    this.localObjectNodepanda_excited =
        (didAddLocalNodepanda_excited!) ? newNode : null;
  }
}
```

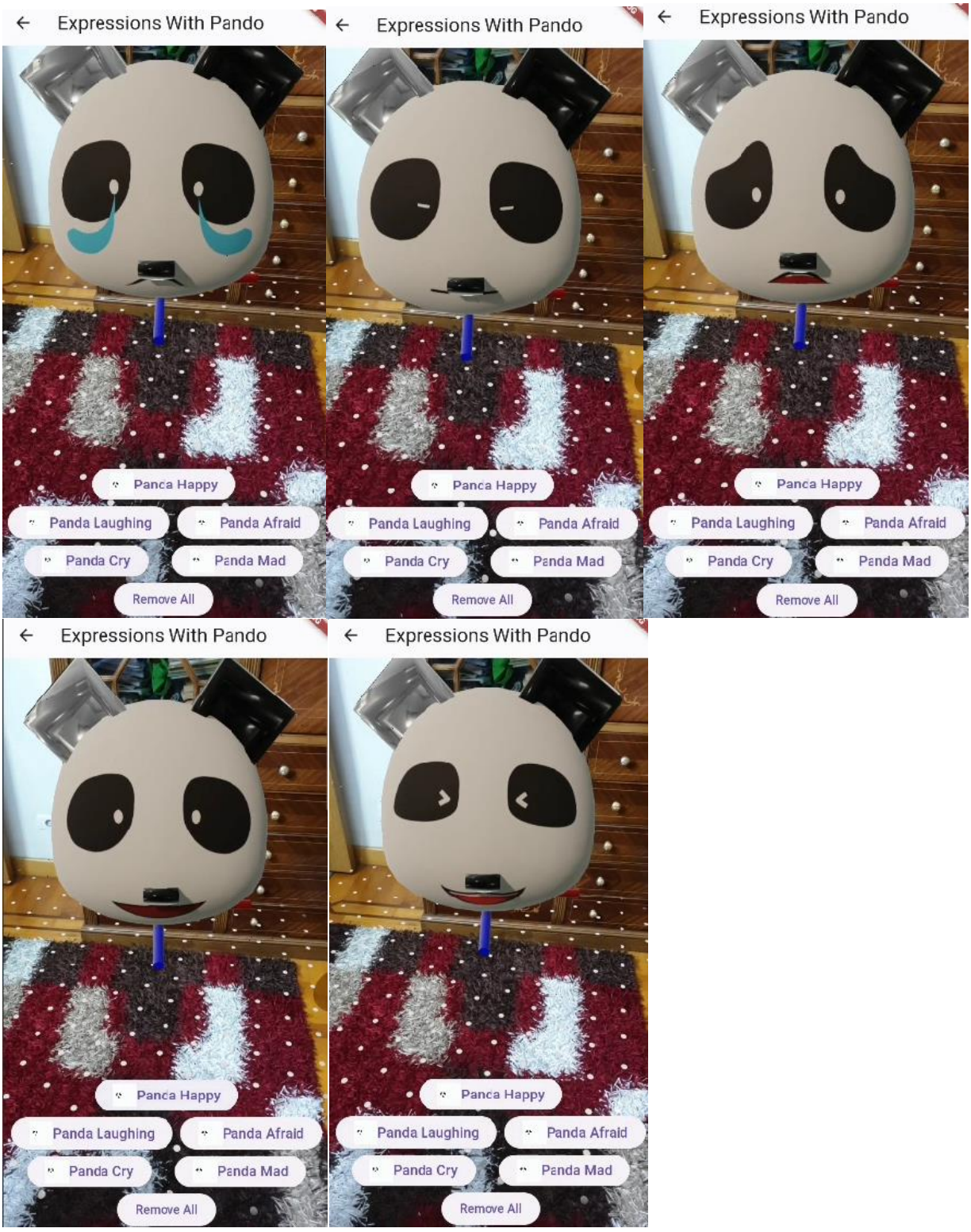
Remove All Nodes Method:

- **_removeAllNodes Method:**

```
Future<void> _removeAllNodes() async {
  if (this.localObjectNodepanda_excited != null) {
    await this.arObjectManager!.removeNode(this.localObjectNodepanda_excited!);
    this.localObjectNodepanda_excited = null;
  }
  if (this.localObjectNodepanda_laugh != null) {
    await this.arObjectManager!.removeNode(this.localObjectNodepanda_laugh!);
    this.localObjectNodepanda_laugh = null;
  }
  if (this.localObjectNodepanda_sad != null) {
    await this.arObjectManager!.removeNode(this.localObjectNodepanda_sad!);
    this.localObjectNodepanda_sad = null;
  }
  if (this.localObjectNodepanda_cry != null) {
    await this.arObjectManager!.removeNode(this.localObjectNodepanda_cry!);
    this.localObjectNodepanda_cry = null;
  }
  if (this.localObjectNodepanda_mad != null) {
    await this.arObjectManager!.removeNode(this.localObjectNodepanda_mad!);
    this.localObjectNodepanda_mad = null;
  }
  print('All nodes removed successfully');
}
```

Removes all panda nodes from the AR session, ensuring a clean slate.

Sample runs:


Expressions With Pando
Panca Happy
Panda Laughing
Panda Afraid
Panda Cry
Panda Mad
Remove All

5.5 Emotion Detection Feature

5.5.1 Feature Description

This feature interacts with the mobile's camera app with some functionalities for object recognition using TensorFlow Lite. \

The feature is used to take the input of a question using the user's face to help them recognise emotions

Camera and Image Capture:

- The app utilizes the `camera` package to access the device's camera.
- A `ScanController` class manages the camera functionalities.
- The controller initializes the camera and captures images at a set interval.
- Users can likely trigger a capture action using a button (not shown in the provided snippets).

Object Recognition with TensorFlow Lite:

- The app integrates TensorFlow Lite, a mobile-friendly machine learning framework.
- A pre-trained TensorFlow Lite model (likely stored in the assets folder) is loaded for object recognition.
- The captured images are converted to a suitable format for the model.
- The model performs inference on the images to identify objects.

Image and Label Display:

- Captured images are stored in a list (`imageList`) within the `ScanController`.
- Recognized labels (text strings) are stored in separate lists (`resultList` and `answersList`) within the controller.
- The `TopImageViewer` widget displays a scrollable list of captured images with their corresponding labels.
- The label color indicates a match between the emotion and the requested emotion

5.5.2 Packages and Apps used in this feature

Core Development Framework:

- **Flutter:** An open-source framework by Google used for building beautiful and performant mobile applications for iOS and Android (and other platforms) using a single codebase. It provides a rich set of widgets for building user interfaces and interacts with native platform functionalities through plugins.

Camera Access:

- **camera** package: A commonly used Flutter package that provides access to the device's camera functionalities. It allows capturing images, handling camera orientation, and potentially accessing other camera features.

State Management:

- **GetX** library: A popular state management library for Flutter applications. It simplifies managing application state and provides reactive updates to the UI whenever the state changes. In this app, `ScanController` likely extends `GetxController` to manage the camera state, captured images, and recognized labels.

Machine Learning:

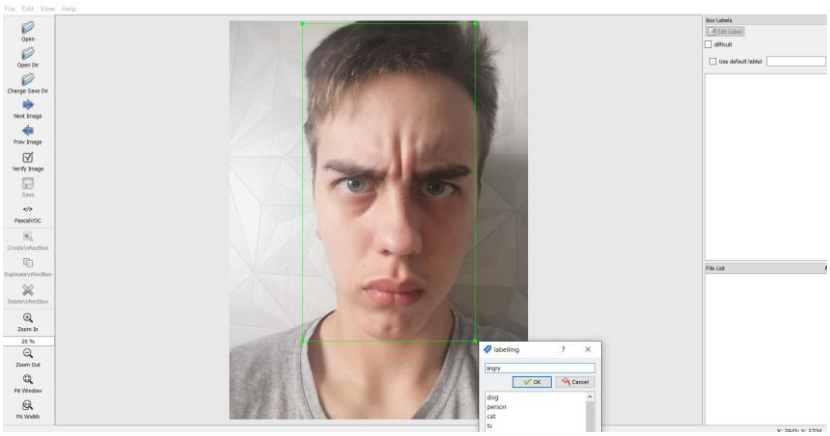
- **TensorFlow Lite:** A lightweight machine learning framework by Google optimized for mobile and embedded devices. This app uses a pre-trained TensorFlow Lite model stored in the assets folder for object recognition on captured images.

Additional Libraries:

- **image** package: Likely used for image manipulation tasks like converting captured images to a format suitable for the TensorFlow Lite model.
- **path_provider** package: Potentially used for accessing the application directory to store captured images (although this functionality isn't entirely clear from the provided snippets).

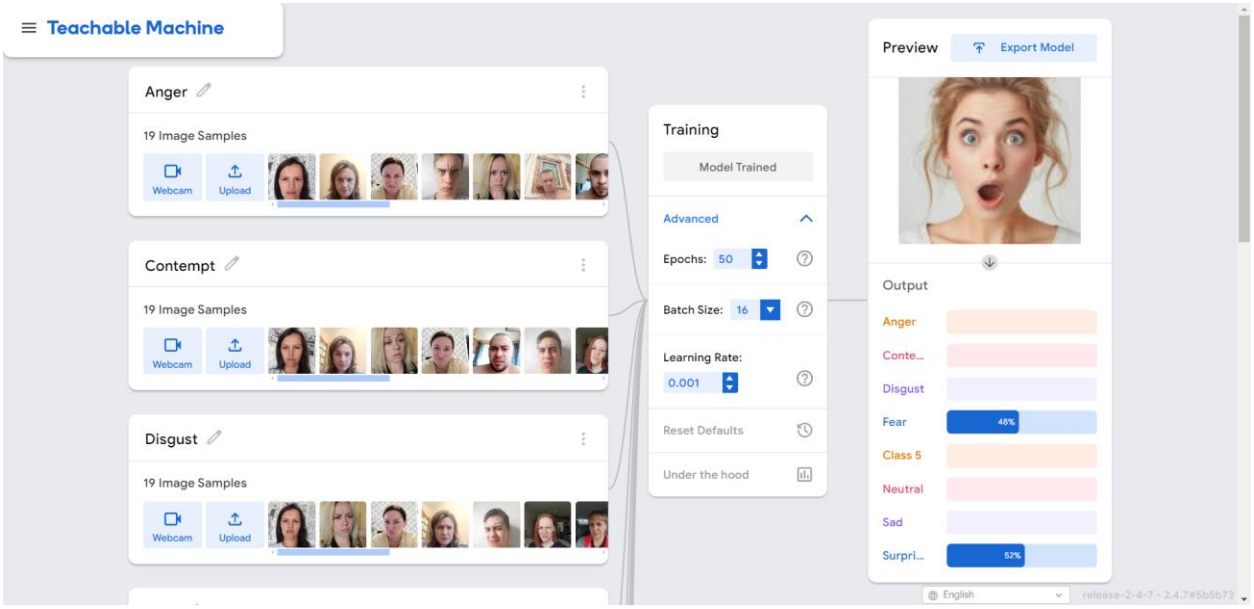
Labeling:

LabelImg is a graphical image annotation tool [1]. It allows you to label images with shapes like bounding boxes and polygons. These annotations are saved in formats that are compatible with popular datasets like PASCAL VOC and ImageNet, which can be useful for machine learning tasks like object detection and image classification.



5.5.3 Creation of the model:

Method 1:



Google's Teachable Machine is a web-based tool designed to make creating machine learning models accessible to everyone, without the need for coding. It allows you to train a computer to recognize things like images, sounds, and poses.

Here's how it can be used for this feature:

1. **Image Classification:** You can set up a project focused on image classification.
2. **Define Emotions:** Create different categories for facial expressions, like happy, sad, angry, etc.
3. **Train the Model:** Use your webcam to capture images of yourself displaying each emotion. The more data you provide, the better your model will learn.
4. **Test and Refine:** Once trained, you can test the model with new images and see how well it recognizes emotions. You can then refine it by adding more data or adjusting categories.

Pros of using Teachable Machine for face emotion detection:

- **Easy to Use:** No coding knowledge required, making it beginner-friendly.
- **Visual Interface:** Simple interface for capturing images and training the model.
- **Accessible:** Works directly in your web browser, no software installation needed.

Cons of using Teachable Machine for face emotion detection:

- **Limited Accuracy:** May not be as accurate as more complex machine learning models.

- **Small Dataset:** Relies on user-provided data, which might be limited for comprehensive emotion recognition.
- **Basic Functionality:** Lacks advanced features found in professional emotion detection tools.

Method 2:

Pros of Using Colab for Face Emotion Detection App Development:

- **Free Powerful Resources:** Colab offers access to powerful GPUs for training complex models without upfront hardware costs. This makes it ideal for experimentation and learning.
- **Easy Collaboration:** Colab notebooks can be shared and edited collaboratively, making it suitable for team projects.
- **Pre-built Libraries:** Essential libraries like TensorFlow, Keras, and OpenCV are readily available, saving setup time.
- **Scalability (to an extent):** While Colab has resource limitations, you can connect to external cloud platforms (like Google Cloud Platform) for larger-scale training needs.
- **Reproducibility:** Colab notebooks provide a documented record of your development process, making it easier to replicate and share your work.

Cons of Using Colab for Face Emotion Detection App Development:

- **Limited Resources:** Colab sessions run for a maximum of 12 hours and resources are not guaranteed. This can be disruptive for long training sessions.
- **Security Concerns:** Uploading sensitive data (e.g., facial images) to a cloud platform might raise security concerns, depending on the application.
- **Not Production-Ready:** Colab is not a production environment. Deploying a trained model for real-world use requires packaging it for specific app frameworks and hosting it on a reliable platform.
- **Limited Control:** You have limited control over the underlying hardware and software environment, which can be a drawback for fine-tuning performance.

- **Dependency on Internet:** Continuous internet connectivity is essential for using Colab, which can be an issue for users with unreliable connections.

5.5.4 Code overview:

main.dart

1. MyApp Class:

- Extends `StatelessWidget`: This means the UI for this widget won't change dynamically based on state.
- Constructor (`MyApp`): A simple constructor with an optional key parameter.
- `build` Method (commented out):
    - Takes `BuildContext` as input, which provides information about the widget location in the widget tree.
    - Returns a `Widget`: This defines the root widget of your application.
    - Uses a conditional statement to choose between two options:
        - `GetMaterialApp`: This widget is from the GetX library and configures the MaterialApp for a GetX application.
            - `initialBinding: GlobalBindings()`: Sets up the `GlobalBindings` class from the beginning, making the `ScanController` available throughout the app.
            - Other properties configure the app title, theme, and the initial screen (`CameraScreen`).
        - `MaterialApp`: This is the standard Flutter widget for building material design apps. (Commented out in this case)
            - Configures app title and theme.
            - Sets the initial screen (`CameraViewer`).

2. main Function:

- The entry point of the application.
- Calls `runApp` to launch the widgets defined in the `MyApp` class.

camera_screen.dart

1. CameraScreen Class:

- Extends `StatelessWidget`: This means the UI for this screen won't change dynamically based on state.
- Constructor (`CameraScreen`): A simple constructor with an optional key parameter.

2. build Method:

- Takes `BuildContext` as input, which provides information about the widget location in the widget tree.
- Returns a `Widget`: This defines the UI for the camera screen.

- Creates a `Stack` widget to stack multiple widgets on top of each other with specified alignment (center in this case).
    - Uses four child widgets within the `Stack`:
        - `Positioned` widget: Positions a `Center` widget at the top with adjusted values (`top: 175.0`).
            - The `Center` widget vertically centers the wrapped `Obx` widget.
            - `Obx` widget (from GetX): Wraps a `Text` widget to rebuild the text whenever the `latestLabel` in `scanController` changes.
                - The `Text` widget displays a message based on the content of `scanController.latestLabel`:
                    - If empty, it shows "Press the Camera button to get started".
                    - Otherwise, it displays a message suggesting trying an emotion based on the recognized label.
        - `CameraViewer` widget: Likely displays the camera feed preview (implementation not shown here).
        - `CaptureButton` widget: Likely displays a button to capture an image (implementation not shown here).
        - `TopImageViewer` widget: Probably displays a scrollable list of captured images with recognized labels (implementation not shown in this file).

camera_viewer.dart

1. CameraViewer Class:

- Extends `GetView<ScanController>`: This signifies it's a GetView associated with the `ScanController` class.
- Constructor (`CameraViewer`): A simple constructor with an optional key parameter.

2. build Method:

- Takes `BuildContext` as input.
- Returns a `Widget`: This defines the UI for the camera preview area.
- Uses `GetX<ScanController>` from GetX:
    - Takes a builder function that receives the `ScanController` instance (`controller`).
    - Checks if the camera is initialized using `controller.isInitialized`.
        - If not initialized, returns an empty `Container` (likely showing a placeholder).
    - If initialized, it uses a `ClipPath` widget:
        - `clipper` property sets a custom clipper (`MyCustomClipper`) to define the shape of the camera preview.
        - `child` property holds a `SizedBox` that expands to fill the screen (`Get.height` and `Get.width`).
            - Inside the `SizedBox`, a `CameraPreview` widget from the `camera` package displays the camera feed. This widget requires a

`CameraController` instance, which is accessed from `controller.cameraController`.

3. MyCustomClipper Class:

- Extends `CustomClipper<Path>`: This class defines a custom shape for clipping the camera preview.
- Constructor (`MyCustomClipper`): Defines an optional key parameter and sets the cutout height for the custom shape (`cutoutHeight`).

4. getClip Method:

- This method is required by `CustomClipper`.
- Takes `Size` as input, representing the dimensions of the area to be clipped.
- Returns a `Path` object that defines the clipping shape.
    - In this case, the path creates a rectangle with a cutout at the bottom:
        - Starts from the bottom left corner of the cutout (`moveTo(0.0, cutoutHeight)`)
        - Draws a line to the bottom right corner of the cutout (`lineTo(size.width, cutoutHeight)`)
        - Draws a line to the bottom right corner of the screen (`lineTo(size.width, size.height)`)
        - Draws a line to the top left corner of the screen (`lineTo(0.0, size.height)`)
        - Closes the path explicitly (`close()`) to ensure a complete shape.

5. shouldReclip Method:

- This method is required by `CustomClipper`.
- Takes a `CustomClipper<Path>` as input, representing the previous instance of the clipper.
- Returns a `bool` indicating if the clipper needs to be redrawn.
- In this case, it always returns `false`, meaning the clip path won't be recalculated unless a new instance of `MyCustomClipper` is created.

capture_button.dart

1. CaptureButton Class:

- Extends `GetView<ScanController>`: This signifies it's a GetView associated with the `ScanController` class.
- Constructor (`CaptureButton`): A simple constructor with an optional key parameter.

2. build Method:

- Takes `BuildContext` as input.
- Returns a `Widget`: This defines the UI for the capture button.
- Uses a `Positioned` widget to place the button at the bottom of the screen (`bottom: 30`).
- Inside the `Positioned` widget, a `GestureDetector` detects user taps on the button.
    - The `onTap` callback calls the `controller.capture()` method from the `ScanController` when the button is tapped.
- The `GestureDetector` child is a `Container` that defines the button's appearance:
    - The container has a height and width of 100 to create a square button.
    - It has padding of 5 around the edges for aesthetics.
    - A `BoxDecoration` defines the button's style:
        - `shape: BoxShape.circle` makes it a circular button.
        - `color: Colors.transparent` makes the inner area transparent.
        - `border: Border.all` creates a white border around the button with a width of 5 pixels.
    - Inside the first container, another `Container` creates the white circle inside the button:
        - `color: Colors.white` fills the inner area with white.
        - `shape: BoxShape.circle` maintains the circular shape.
    - A `Center` widget positions the camera icon within the button.
    - An `Icon` widget displays a camera icon (`Icons.camera`) with a size of 60 pixels.

global_bindings.dart

1. GlobalBindings Class:

- Extends `Bindings`: This signifies it's a class used for dependency injection in GetX.

2. dependencies Method:

- This method is required by the `Bindings` class.
- It doesn't take any arguments.
- Its purpose is to define dependencies that can be accessed throughout the application using GetX.
- In this case, it uses `Get.lazyPut`:
    - The first argument is a key (`ScanController`): This key is used to identify the dependency.
    - The second argument is a function that creates the dependency:
        - It creates a new instance of `ScanController` using `() => ScanController()`.
- By calling `Get.lazyPut`, this dependency is registered with GetX and can be accessed using `Get.find<ScanController>()` from anywhere in the application.

scan_controller.dart

1. ScanController Class:

- Extends `GetxController`: This signifies it's a GetX controller class managing camera, image capture, and recognition functionalities.

2. Member Variables:

- `_cameras`: List to store available cameras on the device.
- `_cameraController`: CameraController instance for controlling the camera.
- `_isInitialized`: RxBool to track camera initialization state (initialized or not).
- `_cameraImage`: CameraImage object to hold the latest captured image frame.
- `_imageList`: RxList to store captured image data (as Uint8List).
- `_resultList`: Private RxList to store recognized labels (String). Public getter (`resultList`) provides access.
- `_answersList`: Private RxList to store suggested answers based on recognized labels (String). Public getter (`answersList`) provides access.
- `_latestLabel`: RxString to store the latest recognized label. Public getter (`latestLabel`) provides access.
- `_myString`: RxString (not directly related to camera or recognition).
- `_imageCount`: Counter variable to track captured image frames (likely used for rate limiting object recognition).
- `_labelflag`: Flag variable (purpose unclear without context).
- `_randomLabels`: List to store randomly chosen labels (likely for suggesting answers).

3. Public Getters and Setters:

- `cameraController`: Getter to access the CameraController instance.
- `isInitialized`: Getter to check camera initialization state.
- `imageList`: Getter to access the list of captured image data.
- `resultList`: Public getter for the private `_resultList` to access recognized labels.
- `answersList`: Public getter for the private `_answersList` to access suggested answers.
- `latestLabel`: Public getter for the RxString holding the latest recognized label.
- `myString`: Public getter and setter for the `_myString` variable.

4. dispose Method:

- Called when the controller is disposed of.
- Sets `_isInitialized` to false, disposes of `_cameraController`, and closes the TensorFlow session.

5. getRandomString Method:

- Takes a list of strings as input.
- Throws an exception if the list is empty.
- Uses a random number generator to select and return a random string from the input list.

6. _initTensorFlow Method (Async):

- Loads the TensorFlow model ("converted_model.tflite") and labels ("labels.txt") from the assets folder.
- This method sets up the TensorFlow environment for object recognition.

7. initCamera Method (Async):

- Retrieves available cameras (`_cameras`).
- Initializes `_cameraController` with the second camera (index 1) using high resolution and YUV420 image format group.
- Starts the image stream and captures images periodically (every 30 frames based on `_imageCount`).
- Sets `_isInitialized` to true upon successful initialization.

8. onInit Method:

- Calls `initCamera` to initialize the camera.
- Calls `_initTensorFlow` to set up TensorFlow.
- Inherited from `GetxController`, likely called when the controller is initialized.

9. _objectRecognition Method (Async):

- Takes a `CameraImage` object as input (the captured image frame).
- If `cameraImage` is not null:
    - Runs object recognition on the image using TensorFlow's `runModelOnFrame` method.
    - Extracts the label with the highest confidence score.
    - Implements logic for choosing a random label from `_randomLabels` if the `_labelflag` is set:
        - Fills `_randomLabels` with labels from the first recognition pass (when `_labelflag` is 0).
        - In subsequent calls (when `_labelflag` is 1), it picks the label with the highest confidence score.
    - Updates `_latestLabel` with the chosen label and refreshes it.
    - Adds the chosen label to `_answersList` and refreshes it.
    - Adds the image data (as Uint8List) to `_imageList` and refreshes it.
- Returns the chosen label (or null if no recognition was successful).

10. _convertYUV420toImageColor Method:

- Takes a `CameraImage` object as input.
- Converts the YUV420 image format to an `imglib.Image` object in RGB format.
- This method prepares the captured image for processing by TensorFlow which expects RGB format.

11. rotateImage180Degrees Method (continued):

- Takes an `imglib.Image` object as input.
- Creates a new `imglib.Image` object with the same dimensions.
- Iterates through each pixel of the original image and copies it to the new image, but flipped horizontally (180 degrees rotation).
- Returns the rotated image.

12. capture Method (Async):

- Called when the user taps the capture button.
- Checks if `_cameraImage` is not null (meaning an image has been captured).
- If an image is available:
    - Attempts to convert the captured YUV420 image to an RGB `imglib.Image` using `_convertYUV420toImageColor`.
    - Rotates the converted image by 180 degrees using `rotateImage180Degrees`.
    - Converts the rotated RGB image into a byte list (`Uint8List`) using `imglib.encodePng`.
    - Calls `_objectRecognition` to perform object recognition on the captured image.
    - Generates a random string from `_randomLabels` and assigns it to a variable `randomString`.
        - This suggests the answer based on potentially recognized objects.
    - Updates `_latestLabel` with `randomString` and refreshes it.
    - Adds `randomString` to `_answersList` and refreshes it.
    - Adds the image byte list to `_imageList` and refreshes it.
- Catches any errors during image conversion or saving and logs them.

This breakdown highlights the functionalities of the `ScanController`:

- It manages camera initialization, image capture, and object recognition using TensorFlow.
- It converts captured YUV420 images to RGB format for TensorFlow processing.
- It rotates captured images by 180 degrees (likely due to camera sensor orientation).
- It stores captured image data and recognized labels in RxLists for access from the UI.
- It offers functionalities to suggest answers based on recognized objects.

top_image_viewer.dart

1. TopImageViewer Class:

- Extends `StatelessWidget`: This signifies it's a static UI widget that doesn't change dynamically based on state.
- Constructor (`TopImageViewer`): A simple constructor with an optional key parameter.

2. build Method:

- Takes `BuildContext` as input.
- Returns a `Widget`: This defines the UI for the top image viewer.

- Uses `GetX<ScanController>` from GetX:
    - Takes a builder function that receives the `ScanController` instance (`controller`).
    - Builds a `Positioned` widget to place the image viewer at the top of the screen (`top: 50`).
    - Inside the `Positioned` widget, a `SizedBox` defines the overall size of the viewer (`height: 100` and `width: Get.width`).
        - A `ListView.builder` widget creates a horizontal scrolling list to display captured images:
            - `scrollDirection: Axis.horizontal` ensures horizontal scrolling.
            - `itemCount` sets the number of items (images) based on `controller.imageList.length`.
            - `itemBuilder` builds each list item (image thumbnail):
                - Extracts the recognized label (`resultText`) and suggested answer (`answerText`) from the controller's lists, handling potential index out-of-range issues.
                - Sets the text color (`textColor`) based on whether the label and answer match (green for match, red otherwise).
                - A `SizedBox` defines the size of each image thumbnail (`height: 100`, `width: 75`).
                - A `Stack` widget layers the image and text within the thumbnail:
                    - A `ClipRRect` with rounded corners clips the image (`controller.imageList[index]`).
                    - A `RepaintBoundary` ensures the image is correctly repainted when scrolled.
                    - Inside `RepaintBoundary`, a `Container` holds the image using a `DecorationImage` with `MemoryImage`.
                    - A `Positioned` widget places the recognized label text (`resultText`) at the bottom right corner (`bottom: 5.0`, `right: 5.0`).
                        - Text style is set using `color` (based on match) and `fontSize`.

5.5.5 Model Generation Overview:

1: Gather and Label Training Images:

Building a dataset for emotion recognition involves gathering images of faces and labeling them with corresponding emotions. We can use our webcam to collect images and store them. We can browse Kaggle for datasets featuring different emotions.

Using Labelimg we can label each image we find and generate a corresponding file that can be used to guide the model of what emotions it can detect.

2: Install TensorFlow Object Detection Dependencies

```
# Clone the tensorflow models repository from GitHub
!pip uninstall Cython -y # Temporary fix for "No module named
'object_detection'" error
!git clone --depth 1 https://github.com/tensorflow/models
# Copy setup files into models/research folder
%%bash
cd models/research/
protoc object_detection/protos/*.proto --python_out=.
#cp object_detection/packages/tf2/setup.py .
# Modify setup.py file to install the tf-models-official repository
targeted at TF v2.8.0
import re
with
open('/content/models/research/object_detection/packages/tf2/setup.py') as
f:
    s = f.read()
```

```
with open('/content/models/research/setup.py', 'w') as f:
    # Set fine_tune_checkpoint path
    s = re.sub('tf-models-official>=2.5.1',
               'tf-models-official==2.8.0', s)
    f.write(s)
# Install the Object Detection API (NOTE: This block takes about 10
minutes to finish executing)

# Need to do a temporary fix with PyYAML because Colab isn't able to
install PyYAML v5.4.1
!pip install pyyaml==5.3
```

3: Upload Image Dataset and Prepare Training Data

Upload the file full of images and labels as a zip file and unzip it
split the images accordingly to traina dn test data

4: Set Up Training Configuration:

Browse theTF2 obejection detection zoo for a mobilessd model to be able to be used in out application
choose a model and assign parameters to it

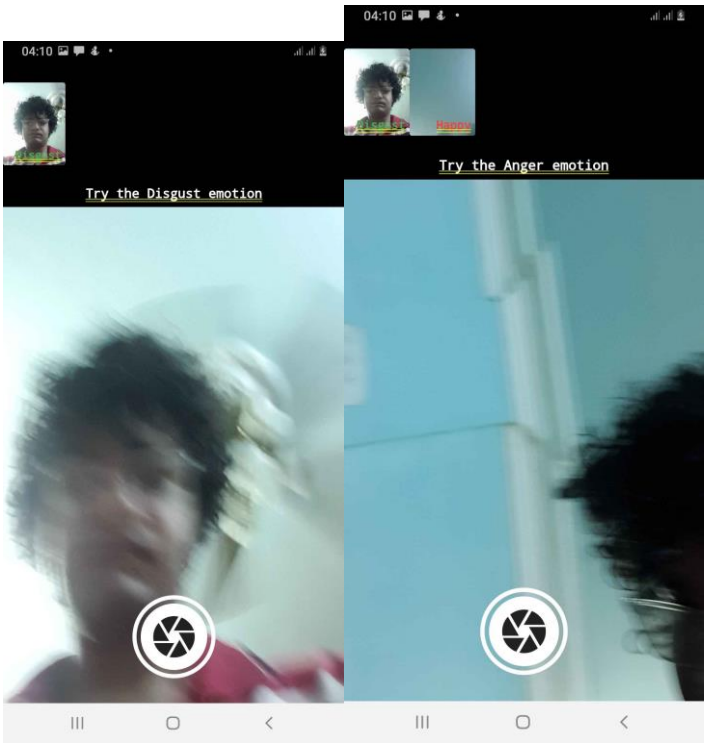
5: Train Custom TFLite Detection Model

run the chosen model on our data
then convert it to a tflitemodel and a txt file containing every class

5.5.6 Screenshots:

The app asks u to start it using the camera button

The app then looks for the labels.txt file and selects a random class (emotion) to use as a test



The app stores the input in a sliding list above and displays the captured emotion if its correct labels it green if not red

6. Problems encountered:

ARCore Integration

1. Integration Issues

a. Platform-Specific Code

- **Problem:** Flutter relies on platform channels to communicate between Dart and the native code (Java/Kotlin for Android, Swift/Objective-C for iOS). This can be complex, especially for handling AR functionalities which often require fine-grained control and real-time processing.
- **Solution:** Familiarize yourself with writing platform-specific code and use Flutter's `MethodChannel` to bridge Dart and native code.

b. ARCore SDK Compatibility

- **Problem:** Ensuring the ARCore SDK is compatible with the Flutter environment and managing dependencies can be tricky.
- **Solution:** Use existing packages like `ar_flutter_plugin` that simplify this process, but be prepared to dive into native code for advanced functionalities.

2. Performance Optimization

a. Real-Time Processing

- **Problem:** AR applications require real-time data processing and rendering, which can strain the device's CPU and GPU.
- **Solution:** Optimize your code for performance, minimize the Dart-Native interaction overhead, and ensure efficient memory management.

b. Rendering Performance

- **Problem:** Flutter's rendering pipeline might not be as optimized for AR as native solutions.
- **Solution:** Leverage native views for AR rendering when necessary, using `PlatformView` in Flutter to embed native AR views within the Flutter app.

3. Device Compatibility

a. Hardware Variations

- **Problem:** ARCore requires specific hardware features, and not all Android devices are ARCore-compatible.
- **Solution:** Implement runtime checks to ensure the device supports ARCore and handle cases where it does not gracefully.

b. iOS Compatibility

- **Problem:** ARCore is specific to Android, so you need to handle ARKit for iOS separately.
- **Solution:** Use platform-specific code to integrate ARKit on iOS and ensure feature parity between Android (ARCore) and iOS (ARKit).

4. User Experience (UX)

a. UI/UX Design for AR

- **Problem:** Designing a seamless and intuitive user interface for AR experiences can be challenging.
- **Solution:** Follow AR design best practices, ensure that the AR elements are interactively designed, and provide clear instructions and feedback to users.

b. Gesture Handling

- **Problem:** AR interactions often rely on gestures, which can conflict with Flutter's gesture system.
- **Solution:** Carefully design and implement gesture recognizers to ensure they work seamlessly with both Flutter widgets and AR elements.

5. Testing and Debugging

a. Testing AR Features

- **Problem:** Testing AR functionalities requires physical devices with ARCore support and cannot be fully tested on emulators.

- **Solution:** Regularly test on a range of physical devices to ensure compatibility and performance.

b. Debugging

- **Problem:** Debugging AR interactions and performance issues can be more complex due to the real-time nature of AR.
- **Solution:** Utilize debugging tools provided by both Flutter and native platforms, and consider logging detailed AR session data for analysis.

6. Learning Curve

a. Familiarity with AR Concepts

- **Problem:** AR development involves understanding 3D geometry, sensors, and spatial computing, which might be new to many Flutter developers.
- **Solution:** Invest time in learning AR concepts, and consider leveraging existing AR tutorials and resources.

b. Hybrid Development Skills

- **Problem:** You need to be proficient in both Flutter and native development (Java/Kotlin for Android, Swift/Objective-C for iOS).
- **Solution:** Continuously improve your skills in both areas and collaborate with developers who have complementary expertise.

Chat Bot Integration

1. Integration with Backend Services

a. API Integration

- **Problem:** Integrating the chatbot with a backend service, whether it's a custom server or a third-party chatbot API, requires managing network requests, handling responses, and ensuring real-time communication.
- **Solution:** Use packages like `http` or `dio` for API calls and `web_socket_channel` for real-time updates. Ensure robust error handling and retries for network operations.

b. Authentication and Security

- **Problem:** Managing secure communication between the app and the backend, especially when dealing with sensitive user data.
- **Solution:** Implement proper authentication mechanisms like OAuth, JWT, or token-based authentication. Ensure all data transmitted is encrypted using HTTPS.

2. State Management

a. Maintaining Chat State

- **Problem:** Managing the state of the conversation, especially in a dynamic and interactive environment like a chatbot.
- **Solution:** Use state management solutions like Provider, Riverpod, Bloc, or Redux to manage and persist chat states effectively.

3. UI/UX Challenges

a. Dynamic UI Updates

- **Problem:** Updating the UI dynamically based on the conversation flow, which can include text messages, quick replies, images, and other media.
- **Solution:** Design a flexible chat UI that can handle various message types. Use Flutter widgets like `ListView.builder` for efficient list rendering and `StreamBuilder` for real-time updates.

b. User Interactions

- **Problem:** Ensuring smooth and intuitive user interactions, including sending messages, displaying typing indicators, and handling different types of user inputs.
- **Solution:** Implement input widgets that support text, voice, and other forms of input. Provide visual feedback like typing indicators and message delivery statuses.

4. Performance Optimization

a. Smooth Scrolling and Rendering

- **Problem:** Ensuring smooth scrolling and rendering of the chat interface, especially with a large number of messages.
- **Solution:** Optimize the chat list with efficient scrolling using `ListView` with proper caching mechanisms and lazy loading.

b. Network Efficiency

- **Problem:** Minimizing network latency and efficiently handling data transfer to provide a real-time chat experience.
- **Solution:** Use WebSockets for real-time communication and optimize the payload size of messages to reduce latency.

5. Weaker hardware

While deploying emotion recognition models on mobile devices offers convenience and real-time processing, it comes with limitations:

1. **Computational Constraints:** Mobile devices often have limited processing power and memory compared to desktop computers. Running complex emotion detection models can drain battery life and cause performance issues.

Solution: Utilize lightweight, mobile-optimized emotion detector models. These models are pre-trained on powerful machines but pruned and compressed for efficient mobile deployment. Tools like TensorFlow Lite offer solutions for model conversion and deployment on mobile platforms.

2. **Limited Data Acquisition:** Capturing high-quality images with a mobile camera can be challenging due to factors like poor lighting, varying angles, and occlusions (e.g., glasses, hairstyles). A limited dataset for training the on-device model can lead to reduced accuracy.

Solution: Combine on-device data collection with pre-trained models from cloud services. This leverages the benefits of both approaches: on-device processing for speed and pre-trained models for handling variations in facial data. Techniques like federated learning can also be explored, where multiple devices contribute to model training without sharing the actual image data.

3. **Privacy Concerns:** Storing and processing facial images on the device raise privacy issues. Users might be hesitant to use an app that analyzes their emotions without transparency or control.

Solution: Implement clear data handling practices. Inform users about how their data is collected, stored, and used. Offer options to opt-out of emotion recognition or anonymize the facial data before processing.

Cloud Processing: A Double-Edged Sword

Offloading emotion detection to a cloud server can address computational limitations and potentially offer access to more comprehensive pre-trained models. However, this approach raises significant privacy concerns:

- **Data Security:** Sending facial images to a cloud server introduces a risk of data breaches or unauthorized access. Implementing robust security measures like encryption and access controls is crucial.
- **Latency and Network Dependence:** Cloud processing introduces latency, causing a delay between image capture and receiving the emotion recognition result. This might be unacceptable for real-time applications. Additionally, unreliable internet connectivity can disrupt the functionality.

Face Detection

1. Integration with Face Detection API

a. Platform-Specific Implementation

- **Problem:** Google's ML Kit for face detection needs to be integrated differently for Android and iOS.
- **Solution:** Use platform channels to bridge Flutter with native code (Java/Kotlin for Android and Swift/Objective-C for iOS). Packages like `firebase_ml_vision` (now part of `google_ml_kit`) can simplify this integration.

b. SDK Compatibility and Updates

- **Problem:** Keeping up with updates to Google's ML Kit and ensuring compatibility with the Flutter framework.
- **Solution:** Regularly check for updates to the ML Kit and test compatibility with the latest Flutter versions.

2. Performance Optimization

a. Real-Time Processing

- **Problem:** Face detection requires real-time image processing, which can be computationally intensive.
- **Solution:** Optimize image processing pipelines and use efficient data handling techniques. Utilize device hardware acceleration (e.g., GPU) when possible.

b. Battery Consumption

- **Problem:** Continuous image processing can drain the battery quickly.
- **Solution:** Optimize the app to process images only when necessary, such as when detecting faces or when the camera is active. Implement efficient resource management techniques.

3. Handling Camera Functionality

a. Camera Integration

- **Problem:** Managing camera functionality in Flutter and ensuring smooth performance.
- **Solution:** Use Flutter plugins like `camera` to access the device's camera. Ensure proper handling of camera permissions and lifecycle.

b. Frame Rate and Quality

- **Problem:** Balancing between high frame rates and image quality for accurate face detection.
- **Solution:** Adjust camera settings to find an optimal balance between frame rate and image quality. Experiment with different resolutions and processing rates.

4. User Experience (UX) Design

a. Intuitive Interface

- **Problem:** Designing a user-friendly interface that clearly indicates the face detection process.
- **Solution:** Provide visual feedback such as bounding boxes around detected faces, and ensure the interface is responsive and intuitive.

b. Error Handling and Feedback

- **Problem:** Handling cases where faces are not detected or detection fails.
- **Solution:** Implement robust error handling and provide clear feedback to users. Indicate possible issues (e.g., poor lighting) and suggest corrective actions.

5. Privacy and Security

a. Data Privacy

- **Problem:** Handling sensitive data like facial images requires strict privacy measures.
- **Solution:** Ensure compliance with data protection regulations (e.g., GDPR). Avoid storing facial images unless absolutely necessary and use encryption for any stored data.

b. Permissions Management

- **Problem:** Properly managing permissions for camera and storage access.
- **Solution:** Clearly request and explain permissions to users, ensuring they understand why access is needed. Handle permission denials gracefully.

6. Testing and Debugging

a. Cross-Platform Testing

- **Problem:** Ensuring consistent performance across different devices and platforms (Android and iOS).
- **Solution:** Regularly test on a variety of devices and screen sizes. Use emulators and real devices to cover different scenarios.

b. Debugging Real-Time Issues

- **Problem:** Debugging real-time face detection issues can be complex.
- **Solution:** Use logging and debugging tools to track issues in real-time processing. Monitor performance metrics and identify bottlenecks.

7. Scalability and Maintenance

a. Scalability

- **Problem:** Ensuring the app can handle an increasing number of users without performance degradation.
- **Solution:** Design the app to be scalable from the start. Use efficient data handling and processing techniques to manage load.

b. Maintenance

- **Problem:** Keeping the app up-to-date with the latest Flutter and ML Kit versions.
- **Solution:** Allocate resources for regular maintenance and updates. Continuously monitor for new releases and test compatibility.

MCQ APP

1. Data Management

a. Question Storage

- **Problem:** Storing and managing a large set of questions, potentially with images, explanations, and different formats.
- **Solution:** Use a structured database like SQLite for local storage or a backend service like Firebase Firestore for cloud storage. Ensure the database schema is well-designed to handle various question types.

b. Dynamic Content

- **Problem:** Loading and updating questions dynamically from a server.
- **Solution:** Implement efficient API communication using packages like `http` or `dio`. Ensure proper data caching and handling for offline access.

2. User Interface (UI) Design

a. Responsive Design

- **Problem:** Ensuring the app looks good and functions well on different screen sizes and orientations.
- **Solution:** Use Flutter's responsive design features and media queries to adapt the layout. Implement flexible widgets like `ListView` and `GridView` to accommodate various screen sizes.

b. Interactive Elements

- **Problem:** Designing interactive elements such as selectable options, timers, and progress indicators.
- **Solution:** Use Flutter's rich set of widgets to create interactive UI elements. Ensure touch-friendly design and provide clear feedback on interactions.

3. State Management

a. Maintaining User Progress

- **Problem:** Tracking user progress through quizzes, including answers given, time taken, and scores.
- **Solution:** Use state management solutions like Provider, Riverpod, or Bloc to manage app state effectively. Ensure data persistence for user progress using local storage.

b. Real-time Updates

- **Problem:** Implementing real-time updates for features like live quizzes or leaderboards.
- **Solution:** Use real-time databases like Firebase Realtime Database or Firestore and leverage Flutter's real-time capabilities with streams and listeners.

4. User Experience (UX)

a. Feedback and Validation

- **Problem:** Providing immediate feedback on answers and explaining correct answers.
- **Solution:** Design clear and informative feedback mechanisms. Use animations and visual cues to enhance the user experience.

b. Accessibility

- **Problem:** Ensuring the app is accessible to users with disabilities.
- **Solution:** Implement accessibility features like screen reader support, large text options, and high contrast modes using Flutter's accessibility tools.

5. Performance Optimization

a. Smooth Navigation

- **Problem:** Ensuring smooth navigation between questions and sections.
- **Solution:** Optimize navigation using Flutter's navigation stack and ensure efficient state management. Preload questions to minimize loading times.

b. Efficient Rendering

- **Problem:** Handling rich content, including images and multimedia, without compromising performance.
- **Solution:** Use efficient image loading techniques, like caching with `cached_network_image`, and ensure media content is optimized for mobile devices.

6. Testing and Debugging

a. Automated Testing

- **Problem:** Ensuring the app functions correctly across different scenarios.
- **Solution:** Implement unit tests, widget tests, and integration tests using Flutter's testing framework. Mock dependencies to isolate test cases.

b. User Feedback

- **Problem:** Collecting and incorporating user feedback to improve the app.
- **Solution:** Implement in-app feedback mechanisms and use analytics tools like Firebase Analytics to monitor user behavior and gather insights.

7. Security and Privacy

a. Data Protection

- **Problem:** Protecting user data, including quiz results and personal information.
- **Solution:** Implement secure data storage practices, use encryption, and ensure compliance with data protection regulations like GDPR and CCPA.

b. Authentication

- **Problem:** Managing user authentication and authorization for personalized experiences.
- **Solution:** Use authentication services like Firebase Authentication or OAuth providers. Ensure secure handling of login credentials and user sessions.

8. Scalability and Maintenance

a. Scalability

- **Problem:** Ensuring the app can handle a growing number of users and data.
- **Solution:** Design the app architecture to be scalable from the start. Use cloud services for scalable backend infrastructure.

b. Maintenance

- **Problem:** Keeping the app up-to-date with new features, bug fixes, and security updates.
- **Solution:** Implement a continuous integration/continuous deployment (CI/CD) pipeline for regular updates. Monitor app performance and user feedback to prioritize maintenance tasks.

Resources

1. Intl J Lang Comm Disor - 2019 - L yt m ki - The role of linguistic and cognitive factors in emotion recognition

2. The Nature of Facial Emotion Recognition Impairments in Children on the Autism Spectrum Nathaniel A. Shanok1 · Nancy Aaron Jones1 · Nikola N. Lucas

3. Augmented Reality Adapted Book (AREmotion) Design as Emotional Expression Recognition Media for Children with Autistic Spectrum Disorders (ASD) Tika Miningrum1, Herman Tolle2, Fitra A Bachtiar3 ,Faculty of Computer Science, Brawijaya University Malang, Indonesia1,2,3

4. Löytömäki J, Ohtonen P, Laakso ML, Huttunen K. The role of linguistic and cognitive factors in emotion recognition difficulties in children with ASD, ADHD or DLD. Int J Lang Commun Disord. 2020 Mar;55(2):231-242. doi: 10.1111/1460-6984.12514. Epub 2019 Dec 3. PMID: 31797474.

5. Marton K, Abramoff B, Rosenzweig S. Social cognition and language in children with specific language impairment (SLI). J Commun Disord. 2005 Mar-Apr;38(2):143-62. doi: 10.1016/j.jcomdis.2004.06.003. PMID: 15571714.

6. Flutter documentation https://docs.flutter.dev/

7. ARCore documentation https://developers.google.com/ar/develop

8.Nielsen, J., & Norman, D. "Usability for Kids." Nielsen Norman Group.

9. Miningrum, T., Tolle, H., & Bachtiar, F. A. (*2021*). Augmented Reality Adapted Book (AREmotion) Design as Emotional Expression Recognition Media for Children with Autistic Spectrum Disorders (ASD).

10. Banerjee, A., Washington, P., Mutlu, C., Kline, A., & Wall, D. P. (Year). Training and Profiling a Pediatric Emotion Recognition Classifier on Mobile Devices.

11. Schwarze, A., Freude, H., & Niehaves, B. (2019). Advantages and Propositions of Learning Emotion Recognition in Virtual Reality for People with Autism.

12. Voss, Catalin, Peter Washington, Nick Haber, Aaron Kline, Jena Daniels,

Azar Fazel, Titas De et al. "Superpower glass: delivering unobtrusive

real-time social cues in wearable systems." In Proceedings of the 2016

ACM International Joint Conference on Pervasive and Ubiquitous

Computing: Adjunct, pp. 1218-1226. 2016.

13. Zhang, M., Ding, H., Naumceska, M., & Zhang, Y. (Year). Virtual Reality Technology as an Educational and Intervention Tool for Children with Autism Spectrum Disorder: Current Perspectives and Future Directions

14. arcore_flutter_plugin documentation: https://pub.dev/documentation/arcore_flutter_plugin/0.1.0/

15. ar_flutter_plugin documentation: https://pub.dev/documentation/ar_flutter_plugin/0.7.3/

16. WebGl documentation: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

17. AWS website: https://aws.amazon.com/

18. React native documentation: https://reactnative.dev/docs/getting-started

19. Xamarin documentation: https://learn.microsoft.com/en-us/xamarin/

20. NativeScript documentation: https://docs.nativescript.org/

21. Vuforia documentation: https://developer.vuforia.com/library/

22. Wikitude documentation: https://www.wikitude.com/external/doc/documentation/

23. AR.js documentation: https://ar-js-org.github.io/AR.js-Docs/

24. Unity MARS documentation: https://docs.unity3d.com/Packages/com.unity.mars@1.0/manual/index.html

25. Maxst documentation: https://developer.maxst.com/MD/doc/6_2_x/intro

Figures:

Figure1 : ID 260255139 © Joebite | Dreamstime.com

Figure2: https://dldandme.org/differences-between-asd-and-dld/

Figure3: https://academiamag.com/scope-of-psycholinguistics-in-pakistan/

Figure4: https://venturebeat.com/business/google-launches-dart-2-5-with-intelligent-code-completion-flutter-1-9-with-ios-13-and-macos-catalina-support/

Figure5: https://www.macworld.com/article/673487/iphone-vs-android-market-share.html

Figure7: https://itnext.io/stateless-vs-stateful-cde9d178084f

Figure8: https://api.flutter.dev/flutter/widgets/Text-class.html

Figure9: https://medium.com/flutter-community/breaking-layouts-in-rows-and-columns-in-flutter-8ea1ce4c1316

Figure10: Zsolt Szilvai's Material Icons Sketch Library

Figure11: https://m3.material.io/components/all-buttons

Figure12: https://developers.google.com/ar

Figure13: https://gs.statcounter.com/os-market-share/mobile/worldwide/#yearly-2009-2023

Figure14: https://www.andreasjakl.com/getting-started-with-google-arcore-part-2-visualizing-planes-placing-objects/

Figure15: https://www.andreasjakl.com/real-time-light-estimation-with-google-arcore/

Figure16: https://medium.com/@ardeploy/build-shared-augmented-reality-experience-for-android-using-sceneform-and-arcore-cloud-anchors-29ae1c55bea7

Figure17: https://www.andreasjakl.com/2d-image-tracking-with-ar-foundation-part-4/

Figure18: https://mobile-ar.reality.news/news/office-depot-updates-elf-yourself-app-with-tiniest-bit-ar-0181461/

Figure19: https://pocketofpreschool.com/monster-feelings-cards/

Figure20: https://theconversation.com/new-autism-early-detection-technique-analyzes-how-children-scan-faces-120820

Figure21: https://www.khronos.org/legal/trademarks/

Figure22: https://logos-world.net/amazon-web-services-logo/

Figure23: https://www.iconfinder.com/icons/7423887/react_react_native_icon

Figure24: https://www.clarity-ventures.com/resources/xamarin/xamarin-mobile-app-development

Figure25: https://seeklogo.com/vector-logo/273738/nativescript

Figure26: https://github.com/PTCInc/vuforia-engine

Figure27: https://www.wikitude.com/media-resources/

Figure28: https://github.com/AR-js-org

Figure29: https://unity.com/products/unity-mars

Figure30: https://www.crunchbase.com/organization/maxst

MCQ App :
Level 1 :
Sad : https://unsplash.com/photos/boy-leaning-on-brown-wooden-railings-wZAQJLjDWNY

Scared : https://www.pexels.com/photo/scared-kid-looking-at-camera-4959221/

Happy : https://pixabay.com/photos/children-siblings-happy-hide-play-1879907/

Surprised : https://www.freepik.es/foto-gratis/guau-retrato-frontal-medio-cuerpo-masculino-atractivo-sobre-fondo-color-rosa-joven-adolescente-sorprendido-emocional-pie-boca-abierta_12698165.htm

30. Angry : https://www.freepik.com/free-photo/angry-displeased-distressed-hateful-blond-little-girl-child-staring-furious-upset-complaining-grimacing-bothered-intense-fight-look-pissed-gesturing-clench-fist-stand-white-wall_18040533.htm

Level 2 : https://www.pexels.com/photo/collage-photo-of-woman-3812743/

Emoji Level : https://www.freepik.com/free-vector/mixed-emoji-set_4159931.htm

Cartoon Level : https://www.pexels.com/photo/collage-photo-of-woman-3812743/