Jairo Adolfo Céspedes Plata

Functionalities:

- Keyboard Detection: The app uses machine learning to detect "keyboard" objects in the environment.

- 3D Cube Overlay: When a keyboard is detected, a 3D cube is displayed on top of the detected object. Only one cube can be present at a time.

- Random User Data: On startup, the app fetches a random name and gender from the [Random User API](https://randomuser.me/).

- Gyroscope Interaction: The cube can be rotated by selecting it and using the device's gyroscope.

- Color Customization: Users can change the color of the cube through the UI.

- Scene Reset: Clicking on an empty area removes all cubes from the interface.

Design Decisions and Technologies Used in ARApp

Architecture and Code Organization

I structured ARApp using the Model-View-ViewModel (MVVM) architecture to promote a clear separation of concerns, enhancing both testability and maintainability. This approach allows for independent development, testing, and maintenance of the application's components.

Project Structure

The project is organized into the following directories, following a modular and scalable structure to ensure clarity and maintainability:

1. config: Contains configuration-related files and utilities, such as global settings and constants, ensuring consistent behavior across the application.
2. data:
    - api: Defines Retrofit services and API interfaces to manage network communication.
    - repository: Implements the repository pattern to act as a single source of truth for data, abstracting data retrieval from the API or local sources.
3. domain.model: Holds the core data models used throughout the application, defining the structure of the data and ensuring consistency.
4. shared.ui: Contains reusable UI components and utilities that are shared across multiple parts of the application, promoting code reusability and consistency.
5. ui:
    - ar: Handles augmented reality-specific features and logic, such as object detection and ARCore integrations.
    - components: Houses modular, reusable UI components that enhance the UI consistency and reduce code duplication.

- scenes: Manages specific AR scenes and their associated UI and logic, encapsulating scene-related behavior.
- theme: Defines the application's visual design elements, such as colors, typography, and styles, ensuring a cohesive look and feel.

6. util:
   - graphics: Provides utilities for managing and rendering graphical elements in the AR environment.
   - ml: Handles machine learning-related logic, including loading and running TensorFlow Lite models for object detection.
   - sensors: Manages sensor integrations, such as the gyroscope, enabling user interactions like 3D model rotation.

This modular organization aligns with best practices, making the codebase scalable, maintainable, and easier to understand. Each directory has a well-defined responsibility, ensuring a clear separation of concerns.

Design Patterns

I implemented several design patterns to enhance code reusability and maintainability:

- Singleton Pattern: Ensures a single instance of specific classes, such as RetrofitInstance for API communication and Settings for global configuration, providing centralized and controlled access across the app.
- Factory Pattern: Utilized in the ObjectDetectionModel class to encapsulate the creation of the TensorFlow Lite interpreter, labels, and related resources. This ensures flexibility and abstraction, allowing changes in the object creation process without impacting dependent code.
- Observer Pattern: Leveraged through Android's LiveData architecture component to enable reactive updates in the user interface. Changes in the ViewModel (such as updates from API responses or object detection events) are observed and reflected in the UI seamlessly.
- Builder Pattern: Applied to configure complex objects, such as AR-related configurations (Config objects in ARCore). This pattern simplifies the construction of objects with many optional parameters and ensures immutability once created.
- Repository Pattern: Centralizes data handling logic by abstracting the sources of data (e.g., API, cache). For instance, RandomUserRepository handles API data retrieval, allowing the app to remain modular and testable.
- Strategy Pattern: Used for handling different AR node behaviors and interactions. For example, user actions (like cube color change or rotation) are dynamically handled by defining strategies for each type of interaction, making the system extensible for future interactions.
- Dependency Injection: While not using a framework like Dagger or Hilt explicitly, dependencies (e.g., ObjectDetectionModel and RandomUserRepository) are injected manually into ViewModels and other components to promote testability and reduce coupling.Concurrency

I utilized Kotlin Coroutines for asynchronous operations, such as network requests and data processing, providing a concise and efficient way to handle concurrency.

Camera Integration

The application integrates the device's camera using ARCore, enabling real-time object detection and augmented reality features. ARCore provides the necessary tools to access the camera feed and overlay 3D models onto the real-world environment. This integration allows the app to detect objects and render 3D cubes in the correct position and orientation.

Object Detection

The application employs a TensorFlow Lite model (ssd_mobilenet_v1_1_metadata_1.tflite) for object detection, specifically identifying **keyboards**. The model is accompanied by a labels.txt file containing the corresponding labels. The ObjectDetectionModel class manages the loading and execution of this model, processing camera frames to detect objects in real-time.

User Interaction

Users can interact with the 3D cube overlaid on detected keyboards by:

- **Rotation**: Utilizing the device's gyroscope, managed by the GyroscopeController class, to rotate the cube.
- **Color Change**: Selecting different colors for the cube through the UI.
- **Removal**: Tapping on an empty area removes all cubes from the scene.

External API Integration

Upon startup, the app fetches a random name and gender from the Random User API, demonstrating external API integration. This is managed by the RandomUserRepository class, which utilizes Retrofit for network operations.

Error Handling

I implemented comprehensive error handling strategies to ensure robustness:

- **Network Issues**: Detects lack of internet connectivity and provides user feedback.
- **Object Detection Failures**: Informs users when the target object (keyboard) cannot be detected.
- **API Errors**: Handles API failures gracefully, ensuring the app remains responsive.

Unit Testing

The application includes unit tests covering over 80% of the codebase, focusing on:

- **Business Logic**: Ensuring correct data processing and transformations.
- **API Interactions**: Verifying successful data retrieval and error handling from the Random User API.

Testing is facilitated using JUnit and Mockito frameworks.

Dependencies

I leveraged several libraries to enhance functionality and development efficiency:

- **AndroidX Libraries**: For backward compatibility and modern Android components.
- **Google ARCore and Sceneform**: To build augmented reality experiences.
- **TensorFlow Lite**: For on-device machine learning and object detection.
- **ML Kit**: Provides machine learning functionalities, including object detection.
- **Retrofit**: For network operations and API consumption.
- **Moshi**: For JSON serialization and deserialization.
- **JUnit and Mockito**: For unit testing and mocking dependencies.