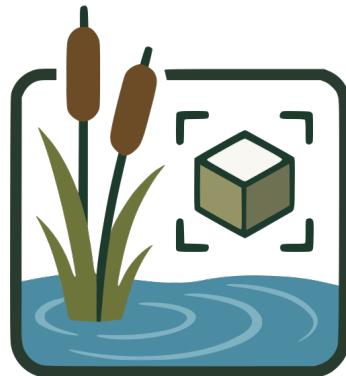


**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
THE UNIVERSITY OF TEXAS AT ARLINGTON**

**DETAILED DESIGN SPECIFICATION
CSE 4317: SENIOR DESIGN II
FALL 2025**



**AR
Wetlands**

**AR WETLAND WATCHERS
WATER POLLUTION SIMULATOR**

**ADRIAN MACIAS
MAURICIO MENDOZA-SILOS
NOORALDEEN ALSMADY
YAHIA ELSAAD
MOHAMAD NAHIB ALKHATEEB**

REVISION HISTORY

Revision	Date	Author(s)	Description
0.1	08.26.2025	AM	Document creation
1.0	09.08.2025	AM, MMS, NA, YE, MNA	Complete draft
2.0	12.05.2025	AM, MMS	Updated to align with final product

CONTENTS

1	Introduction	5
2	System Overview	5
3	Device Hardware Layer	6
3.1	Layer Hardware	6
3.2	Layer Operating System	6
3.3	Layer Software Dependencies	7
3.4	Device Camera Feed	7
3.5	Surface Detection	9
3.6	Device Touch Input	11
4	Data Manager Layer	13
4.1	Layer Hardware	13
4.2	Layer Operating System	13
4.3	Layer Software Dependencies	13
4.4	Database	13
4.5	Asset Loader	15
4.6	3D Assets	16
4.7	Audio	17
4.8	Animations	18
5	App Layer	20
5.1	Layer Hardware	20
5.2	Layer Operating System	20
5.3	Layer Software Dependencies	20
5.4	Scene Management	21
5.5	3D Object Placement	22
5.6	Interactions/Taps	24
5.7	Animation Engine	26
5.8	Marker Recognition	27
5.9	User Interface	29
6	Appendix A	31

LIST OF FIGURES

1	Full data flow architecture	6
2	Device Camera Feed Subsystem description diagram	7
3	Surface Detection Subsystem description diagram	9
4	Device Touch Input Subsystem description diagram	11
5	Database Subsystem description diagram	14
6	Asset Loader Subsystem description diagram	15
7	3D Assets Subsystem description diagram	16
8	Audio Subsystem description diagram	17
9	Animations Subsystem description diagram	18
10	Scene Management subsystem description diagram	21
11	3D Object Placement subsystem description diagram	23
12	Interactions/Taps subsystem description diagram	24
13	Animation Engine subsystem description diagram	26
14	Marker Recognition subsystem description diagram	28
15	User Interface subsystem description diagram	29

LIST OF TABLES

1 INTRODUCTION

The AR Wetlands project is an interactive AR educational system designed to raise awareness about water pollution, runoff, and environmental impacts of everyday life. Sponsored by the U.S. Army Corps of Engineers (USACE), the project combined a physical tabletop exhibit with a mobile AR application to deliver engaging, scenario based learning experiences to a broad audience. Where the main experience centers around a tabletop, serving as a visual anchor for AR content. Printed markers on the table will be recognized by the mobile device's rear camera, triggering real-time 3D simulations of environmental scenarios.

This document outlines the Detailed Design Specification (DDS) for the AR Wetlands system. It details the overall development strategy, system hardware, internal subsystems components, and interactions between components. The architecture is structured to be modular and maintainable, supporting scalability and future enhancements while ensuring robust performance during demonstrations.

2 SYSTEM OVERVIEW

Below is a detailed breakdown of the system's architecture into functional subsystems, organized across three major layers: *Device Hardware Layer*, *Data Manager Layer*, and *App Layer*. Each subsystem represents a core functional unit of the *AR Wetlands*: *Water Pollution Simulator*, operating together to deliver real-time, interactive augmented reality experiences on mobile devices.

The data flow diagram below illustrates how these subsystems interact through defined data paths, numbered for traceability. Arrows indicate the direction of data flow, and inter-layer communication is shown explicitly to reveal dependency and system cohesion.

Each subsystem shown in the diagram contributes to the real-time operation of the simulator. From user input and camera sensing to asset fetching, placement, animation, and interaction, this pipeline forms the backbone of the system. This layered, modular design ensures efficiency on mobile devices while maintaining a smooth user experience within Unity's AR Foundation framework.

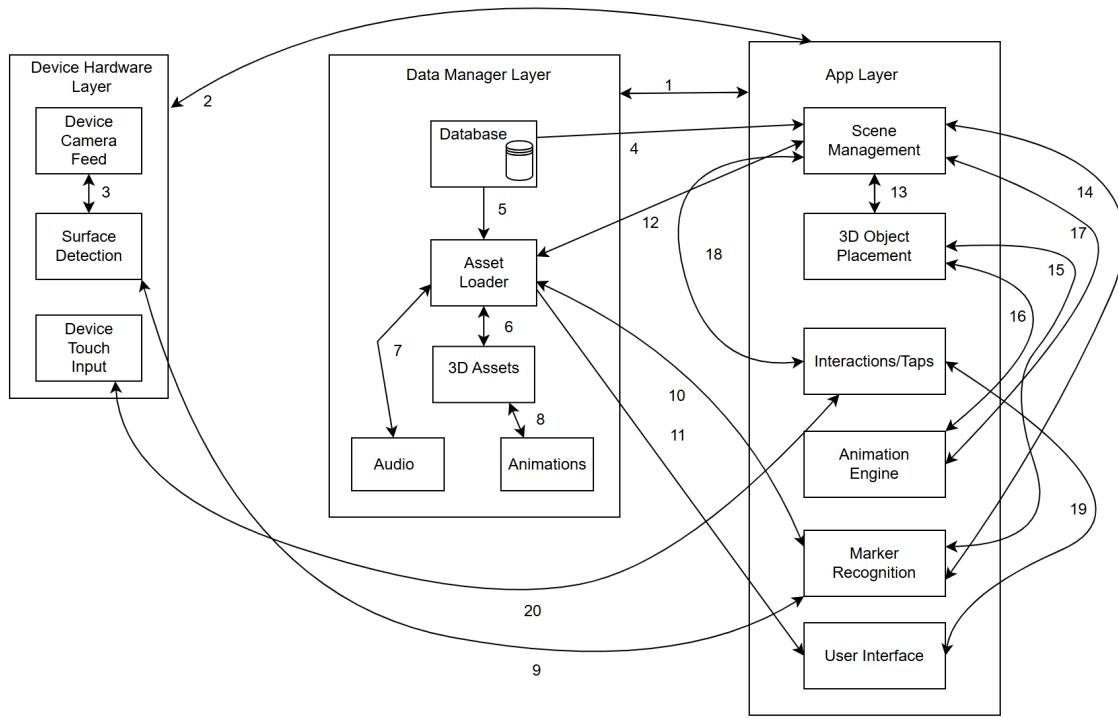


Figure 1: Full data flow architecture

3 DEVICE HARDWARE LAYER

This layer comprises the core input capabilities of the AR Wetlands: Water Pollution Simulator. These subsystems are tightly integrated with the mobile device’s physical hardware and provide the foundational data streams required for surface tracking, scene augmentation, and user interaction.

3.1 LAYER HARDWARE

In this layer, the main piece of hardware that is being interacted with is the user’s mobile device. This can be either a smartphone or tablet. The hardware components include:

- **Device Camera:** Provides the real-time video feed that forms the background for augmented content.
- **Touchscreen:** Allows the user to interact with virtual objects through taps.
- **Inertial Sensors:** Accelerometer and gyroscope sensors are used to determine device orientation and motion for alignment of AR content.
- **Depth/Environment Sensors (optional):** Android devices with depth sensors, support more accurate spatial mapping.
- **Audio Output:** Device speakers or headphones may be used to provide sound feedback during AR Scene interactions.

3.2 LAYER OPERATING SYSTEM

The Device Hardware Layer depends on the mobile device’s operating system to provide access to the hardware through standardized APIs. The supported operating systems is **Android (version 10 or higher)**.

higher). Where the OS Provides access to the device camera, touch input, and motion sensors via Android SDK APIs.

3.3 LAYER SOFTWARE DEPENDENCIES

The Device Hardware Layer requires several software dependencies that allow hardware features to be used effectively by the application:

- **AR Frameworks:** Google ARCore (Android) to manage device motion tracking, environment understanding, and camera feed integration.
- **Device SDKs:** Android SDK provide the low-level access to sensors, camera feeds, and touch input.
- **Graphics/Rendering APIs:** OpenGL ES, Metal, or Vulkan, which allow the hardware video feed to be displayed and integrated into the 3D AR scene.
- **Game Engine Integration:** Unity AR Foundation modules, which unify hardware access across platforms and simplify development.

3.4 DEVICE CAMERA FEED

The Device Camera Feed subsystem captures the live video stream from the device's rear-facing camera. It serves as the visual foundation for the AR experience by providing a continuous feed used by other subsystems. The real-time camera input is processed and forwarded to both the Surface Detection and Marker Recognition subsystems.

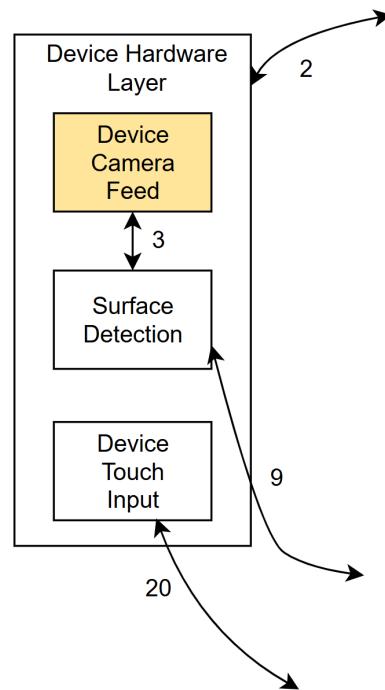


Figure 2: Device Camera Feed Subsystem description diagram

3.4.1 DEVICE CAMERA FEED SUBSYSTEM HARDWARE

The Device Camera Feed subsystem relies on the built-in mobile device camera hardware.

- The hardware supports real-time video capture at various resolutions (e.g., 720p, 1080p, or higher) and frame rates (30–60 fps).
- May also take advantage of additional sensors, such as the gyroscope and accelerometer, to align captured frames with device orientation
- Some devices may support AR-focused hardware features such as depth sensors, or dual-lens configurations for improved surface detection.

3.4.2 DEVICE CAMERA FEED SUBSYSTEM SOFTWARE DEPENDENCIES

The Device Camera Feed subsystem requires several software dependencies:

- **AR frameworks:** Google ARCore (Android) to interpret camera frames for AR tracking and environment understanding.
- **Game Engine integration:** Unity with AR Foundation with (included) AR plugins for managing the camera feed as a background texture.

3.4.3 DEVICE CAMERA FEED SUBSYSTEM PROGRAMMING LANGUAGES

The Device Camera Feed subsystem primarily uses programming languages tied to the development platform (Unity), in this case the programming language is C#.

3.4.4 DEVICE CAMERA FEED SUBSYSTEM DATA STRUCTURES

The key data structures for the Device Camera Feed subsystem include:

- **Camera Frame Buffers:** Raw pixel data stored in RGBA format, managed in memory as arrays or textures.
- **Texture Objects:** Representations of camera frames as GPU-accelerated textures, allowing efficient rendering of the live video background.
- **Pose Data Structures:** Frame metadata that includes timestamp, camera intrinsics (focal length, distortion parameters), and device orientation.
- **Marker Detection Data Packets:** Contain marker ID, 2D image coordinates, and estimated 3D pose for use in placement and interaction.

3.4.5 DEVICE CAMERA FEED SUBSYSTEM DATA PROCESSING

The Device Camera Feed subsystem applies several real-time data processing steps:

1. **Frame Capture:** Continuous acquisition of video frames from the device camera at a set frame rate.
2. **Color Space Conversion:** Converting raw YUV camera output to RGB or grayscale for downstream processing.
3. **Surface Detection Support:** Frames are analyzed in conjunction with ARKit algorithms (e.g., feature point detection, plane estimation) to detect horizontal/vertical surfaces.

4. **Marker Recognition:** Algorithms such as fiducial marker detection (e.g., ARToolkit, ArUco) or image-based recognition are applied to identify reference points in the video stream.
5. **Pose Estimation:** Visual-inertial odometry (VIO) fuses camera frames with IMU sensor data to estimate device position and orientation in real time.
6. **Texture Mapping:** The processed frames are converted into textures to serve as the AR scene's dynamic background.

3.5 SURFACE DETECTION

The Surface Detection subsystem is responsible for identifying flat, real-world surfaces (e.g., the tabletop exhibit) that serve as anchors for AR content placement. This subsystem ensures that virtual objects appear aligned and stable in the user's environment. It is not a hardware component itself but a software subsystem built on top of ARcore APIs, integrated into Unity via AR Foundation.

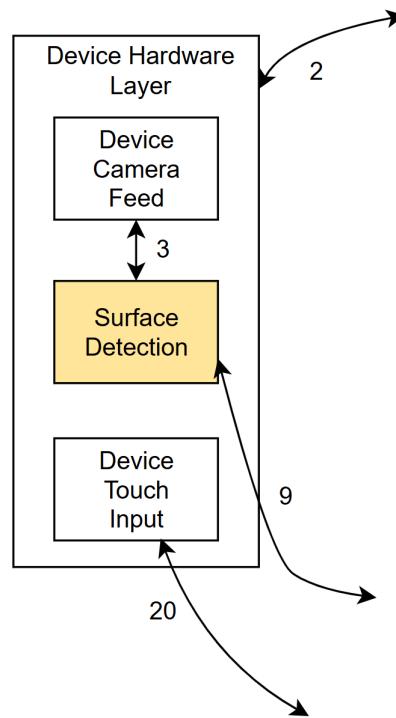


Figure 3: Surface Detection Subsystem description diagram

3.5.1 SURFACE DETECTION SUBSYSTEM HARDWARE

The hardware supporting surface detection consists of the smartphone's built-in sensors, which provide the raw data needed for ARCore's spatial understanding. The camera supplies visual information, while the IMU contributes motion and orientation data to improve tracking accuracy.

- Device rear-facing RGB camera for capturing real-time video frames.
- Uses IMU sensors (accelerometer, gyroscope) for visual-inertial odometry.

3.5.2 SURFACE DETECTION SUBSYSTEM SOFTWARE DEPENDENCIES

Surface detection is powered by ARCore and exposed to Unity through AR Foundation. Unity's physics engine is also used to generate colliders so virtual objects can interact naturally with detected surfaces.

- ARCore (Android): Provides plane detection and point cloud APIs.
- Unity AR Foundation: Wraps ARCore functionality for cross-platform use.
- Unity Physics Engine: Used to create plane colliders for interaction.

3.5.3 SURFACE DETECTION SUBSYSTEM PROGRAMMING LANGUAGES

The logic controlling detected surfaces, anchors, and related AR behaviors is implemented in C#. Developers use AR Foundation's APIs to query ARCore's detected planes and update the scene accordingly.

- Implemented in C# scripts within Unity.
- Uses AR Foundation APIs to query detected planes and update anchors.

3.5.4 SURFACE DETECTION SUBSYSTEM DATA STRUCTURES

To represent surface information and maintain tracking stability, the subsystem uses a set of Unity and ARCore data structures. These store plane geometry, anchor points, and the raw point cloud used for surface estimation.

- ARPlane class: Represents detected planes, including boundary polygon, orientation, and pose.
- ARAnchor objects: Store stable reference points tied to real-world positions.
- PointCloud data: Collection of feature points used by ARCore to estimate surfaces.

3.5.5 SURFACE DETECTION SUBSYSTEM DATA PROCESSING

Surface detection involves real-time processing of camera frames and sensor data to estimate planes accurately. ARCore continuously updates its understanding of the environment, refining plane boundaries and anchor stability as the user moves.

- Capture frames from the camera and extract feature points.
- Fuse camera and IMU data using Visual-Inertial Odometry (VIO).
- Run ARCore's Plane Detection Algorithm to estimate horizontal/vertical planes.
- Generate ARPlane objects in Unity with mesh boundaries and colliders.
- Surface anchors are created for placement of AR objects.
- Update planes dynamically as the user moves, refining geometry over time.

3.6 DEVICE TOUCH INPUT

The Device Touch Input subsystem allows users to interact with AR objects through the touchscreen. It interprets gestures such as taps, drags, and pinches, mapping them to actions within the AR scene. This subsystem acts as the bridge between user interaction and Unity object control.

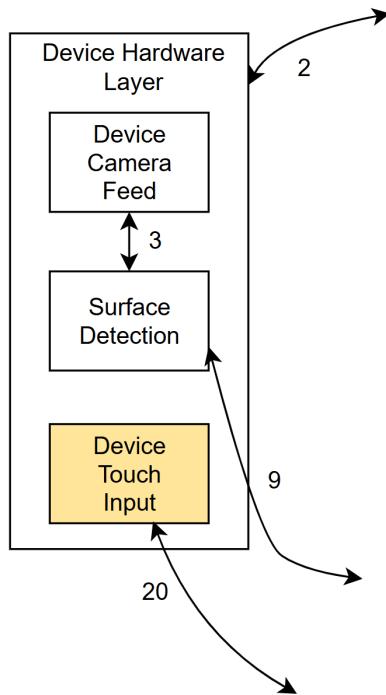


Figure 4: Device Touch Input Subsystem description diagram

3.6.1 DEVICE TOUCH INPUT SUBSYSTEM HARDWARE

This subsystem relies on the built-in touchscreen hardware of the mobile device, which captures detailed multi-touch input. The device's display provides immediate visual feedback, ensuring interactions feel responsive and intuitive.

- Mobile device touchscreen digitizer for capturing multi-touch input.
- Display hardware for immediate visual feedback.

3.6.2 DEVICE TOUCH INPUT SUBSYSTEM OPERATING SYSTEM

The Android operating system plays a key role in processing raw touch signals and packaging them into standardized gesture events. These events are then passed into Unity through the Android input framework, allowing the AR application to react consistently across devices.

- Android OS translates raw touch events into structured inputs (e.g., tap, swipe, pinch).
- Events are passed to Unity through the Android input framework.

3.6.3 DEVICE TOUCH INPUT SUBSYSTEM SOFTWARE DEPENDENCIES

Touch input functionality is built using Unity's Input System or legacy Touch API. When interacting with AR content, AR Foundation components such as the `ARRaycastManager` are used to convert touch positions into world-space intersections with detected surfaces.

- Unity Input System or Touch API for reading touch events.
- Unity AR Foundation for mapping touch positions to AR planes/surfaces via integration with ARRaycastManager.

3.6.4 DEVICE TOUCH INPUT SUBSYSTEM PROGRAMMING LANGUAGES

Interaction behaviors, gesture logic, and AR raycasting operations are implemented using Unity's C# scripting environment.

- Implemented in C# scripts using Unity Input API.

3.6.5 DEVICE TOUCH INPUT SUBSYSTEM DATA STRUCTURES

The subsystem uses several built-in Unity data structures to manage touch events and AR interactions. Custom structures may also be used to track higher-level interaction events based on gesture activity.

- Touch struct (Unity): Contains finger ID, position (x,y), and phase (began, moved, ended).
- RaycastHit / ARRaycastHit: Store intersection results between screen-space touch rays and AR-detected surfaces or objects.
- InteractionEvent objects: Custom data packets linking a user gesture to a specific AR object action.

3.6.6 DEVICE TOUCH INPUT SUBSYSTEM DATA PROCESSING

Touch input processing occurs continuously as users interact with the screen. Gestures are translated into AR actions such as selecting objects, triggering animations, or interacting with scene elements. Feedback (visual, audio, or haptic) is provided to make interactions feel natural.

1. Capture raw touch input from Unity Input System.
2. If touch begins, cast a ray from the screen point into the AR scene.
3. If the ray intersects a detected ARPlane or AR object:
 - Tap: Select or trigger interaction (e.g., start narration).
4. Update object state and provide haptic/audio feedback if enabled.
5. Pass results to Scene Management for synchronization with animations or audio.

4 DATA MANAGER LAYER

The Data Manager Layer is responsible for providing all structured content used by the AR Wetlands system during runtime. It includes internal data management subsystems that collectively support asset delivery, visual rendering, and scenario construction. These subsystems operate offline and serve as the foundation for dynamic scene generation and user interaction. This layer ensures that 3D models, animations, audio, and educational content are preloaded and retrievable within the Unity environment. Each subsystem in this layer contributes to resource availability, playback synchronization, and environmental simulation accuracy. The components described below form the internal backbone of the application's runtime behavior, supplying the App Layer with critical scene elements on demand.

4.1 LAYER HARDWARE

This layer does not introduce new hardware. Instead, it relies on the mobile device's general-purpose hardware components, including:

- **Central Processing Unit (CPU):** For executing data retrieval and management logic. The minimum requirement is a Qualcomm Snapdragon 835 (or equivalent 64-bit ARM with NEON) or higher.
- **Random Access Memory (RAM):** For caching assets and holding runtime data structures. The minimum requirement is 4GB RAM or higher.
- **Internal Storage (Flash/SSD):** For the persistent storage of the application and all its bundled assets, including 3D models, audio files, and the database configurations. While there is no strict minimum, at least 1GB of free space is recommended.
- **Graphics Processing Unit (GPU):** For rendering 3D assets efficiently in real time. The minimum requirement is a Qualcomm Adreno 540 or higher.

4.2 LAYER OPERATING SYSTEM

The Data Manager Layer runs on Android 10 or higher, as supported by Unity with AR Foundation and ARCore. The operating system ensures access to local file storage, in-memory caching and real-time audio/visual asset handling. No external servers or cloud-based OS dependencies are required since the application is designed for full offline functionality.

4.3 LAYER SOFTWARE DEPENDENCIES

The primary software dependency for this layer is the Unity Engine. Within it are the Unity built-in systems for asset serialization, management, and retrieval, such as the APIs for handling Prefabs, ScriptableObjects, AudioClips, and Animation Clips.

4.4 DATABASE

The Database subsystem manages the application's structured data, including scene configurations and the relationships between AR objects and markers. It is not a traditional relational database but a software component implemented within Unity to serve as a local, offline storage system. It provides the configuration data needed by the Scene Management and Asset Loader subsystems.

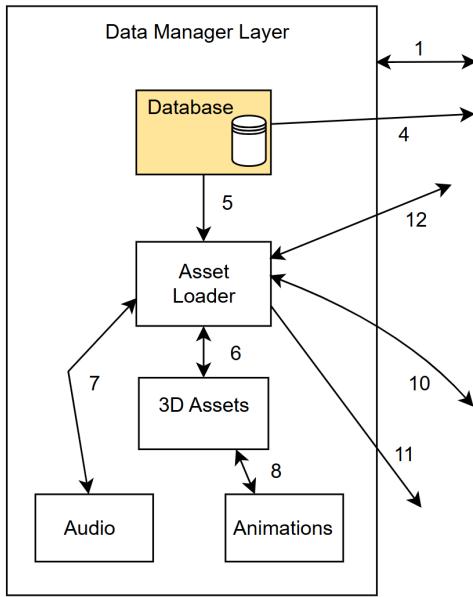


Figure 5: Database Subsystem description diagram

4.4.1 DATABASE SUBSYSTEM HARDWARE

No external hardware, runs entirely in device memory/storage.

4.4.2 DATABASE SUBSYSTEM OPERATING SYSTEM

Android handles file I/O permissions and local persistence.

4.4.3 DATABASE SUBSYSTEM SOFTWARE DEPENDENCIES

The core dependency is the Unity Engine, specifically its serialization system and support for ScriptableObjects.

4.4.4 DATABASE SUBSYSTEM PROGRAMMING LANGUAGES

The logic for creating, accessing, and managing the database structures is written in C#, which is the primary scripting language used in Unity.

4.4.5 DATABASE SUBSYSTEM DATA STRUCTURES

The primary data structures will be implemented using Unity's ScriptableObjects. The most noteworthy being:

- **SceneConfig class:** Which will have a scene ID, asset list, and interaction triggers.
- **MarkerConfig class:** Which will have marker ID and model placement.
- **AssetReference objects:** Which will have paths to 3D, audio, or animation files.

4.4.6 DATABASE SUBSYSTEM DATA PROCESSING

Data processing is limited to read-only operations at runtime. The primary algorithm is a simple look-up. So it will:

1. Load scene parameters on initialization.

2. Resolve marker IDs to assets.
3. Pass asset references to the Asset Loader.

4.5 ASSET LOADER

The Asset Loader is a software class that acts as the central hub for loading all necessary assets from storage into memory at runtime. It ensures assets are retrieved efficiently to maintain a responsive user experience, based on requests from other subsystems like Scene Management or Marker Recognition.

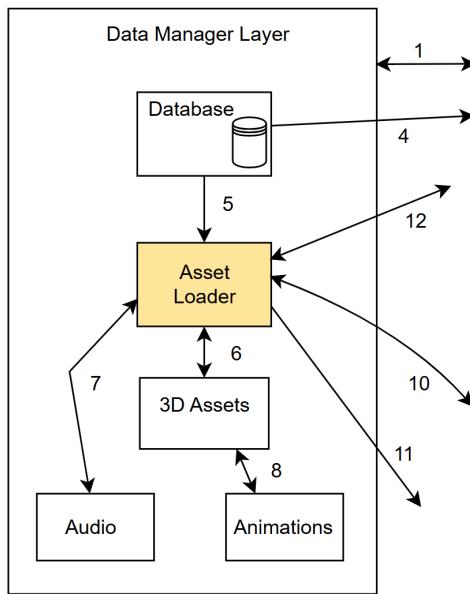


Figure 6: Asset Loader Subsystem description diagram

4.5.1 ASSET LOADER SUBSYSTEM HARDWARE

No additional hardware beyond device CPU/GPU.

4.5.2 ASSET LOADER SUBSYSTEM OPERATING SYSTEM

Android file system provides access to packaged asset bundles.

4.5.3 ASSET LOADER SUBSYSTEM SOFTWARE DEPENDENCIES

The implementation will likely use Unity's Resources folder system for simplicity or the more advanced Addressable Asset System for better memory management.

4.5.4 ASSET LOADER SUBSYSTEM PROGRAMMING LANGUAGES

The logic for accessing and loading the assets is written in C#.

4.5.5 ASSET LOADER SUBSYSTEM DATA STRUCTURES

The most noteworthy data structures would be:

- **LoadedAssetCache class:** Which will have a dictionary of asset IDs that lead to prefabs, audios, and animations.
- **AssetBundleReference objects:** Which will have bundleID and file path.

4.5.6 ASSET LOADER SUBSYSTEM DATA PROCESSING

The system uses a simple cache algorithm. So it will:

1. Request asset reference from Database.
2. Load asynchronously (to avoid frame drops).
3. Cache loaded object for reuse.
4. Return asset to Scene Manager.

4.6 3D ASSETS

This subsystem is not a piece of software but a collection of the 3D models used in the AR scenes, such as environmental props, buildings, and infrastructure. These assets are designed to be optimized for real-time rendering on mobile devices.

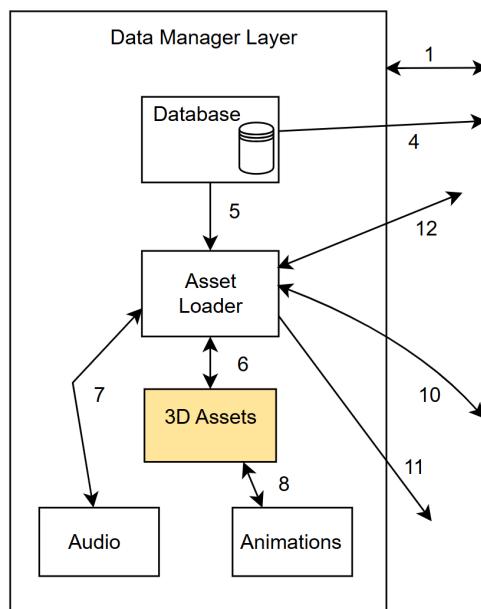


Figure 7: 3D Assets Subsystem description diagram

4.6.1 3D ASSETS SUBSYSTEM HARDWARE

The assets are stored on the device's internal storage. During runtime, they are loaded into RAM and processed by the device's GPU for rendering.

4.6.2 3D ASSETS SUBSYSTEM SOFTWARE DEPENDENCIES

The assets are created in external 3D modeling software and are imported and managed by the Unity Engine. They are typically saved in file formats like .FBX or .OBJ.

4.6.3 3D ASSETS SUBSYSTEM PROGRAMMING LANGUAGES

No custom programming language is used for these assets, though playback and instantiation are controlled by Unity C# scripts.

4.6.4 3D ASSETS SUBSYSTEM DATA STRUCTURES

The primary data structures for this subsystem would be the:

- **Unity Prefab:** Which will contain mesh, texture, and collider metadata.
- **AssetMetadata objects:** Which will have scaling factors, anchor points, and physics properties.

4.6.5 3D ASSETS SUBSYSTEM DATA PROCESSING

The data processing occurs as follows:

1. Instantiate prefabs from Asset Loader.
2. Apply scaling/orientation.
3. Attach animation or interaction scripts.

4.7 AUDIO

The Audio subsystem is the repository of all sound files used in the application, including voice narration and environmental sound effects. All audio content is packaged with the application to ensure it functions offline.

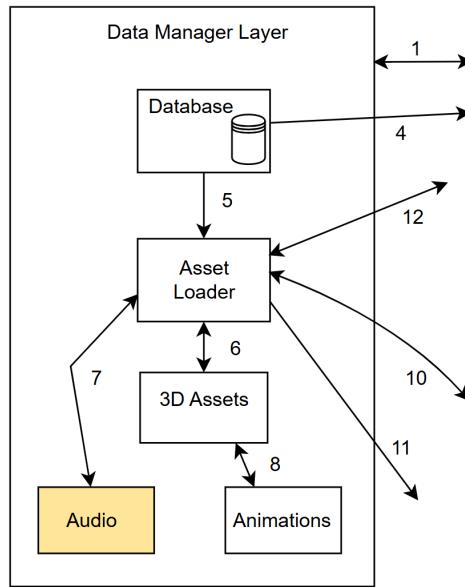


Figure 8: Audio Subsystem description diagram

4.7.1 AUDIO SUBSYSTEM HARDWARE

Audio files are stored on the device's internal storage. Playback utilizes the device's CPU for decoding and the audio output hardware (speakers or headphones).

4.7.2 AUDIO SUBSYSTEM OPERATING SYSTEM

Android Audio API manages low-level audio output.

4.7.3 AUDIO SUBSYSTEM SOFTWARE DEPENDENCIES

The subsystem relies on the Unity Engine's audio system (AudioListener components). Audio files may be created in external software like Audacity and are typically in .MP3 or .WAV format.

4.7.4 AUDIO SUBSYSTEM PROGRAMMING LANGUAGES

No custom programming language is used for these assets, though playback and instantiation are controlled by Unity C# scripts.

4.7.5 AUDIO SUBSYSTEM DATA STRUCTURES

The primary data structures are the AudioClip object and the AudioEvent object in Unity.

- **AudioClip object:** Which will contain clip ID and file path.
- **AudioEvent dictionary:** Which will have trigger ID and leads to clip ID.

4.7.6 AUDIO SUBSYSTEM DATA PROCESSING

Audio files are processed upon import into Unity. Typically following this process:

1. Retrieve clip ID from Database.
2. Load clip via Asset Loader.
3. Sync playback with animations/events.

4.8 ANIMATIONS

The Animations subsystem is a collection of pre-designed animation data used to create dynamic visual effects, such as water runoff or object transformations. These animations are created within Unity and linked to their corresponding 3D assets.

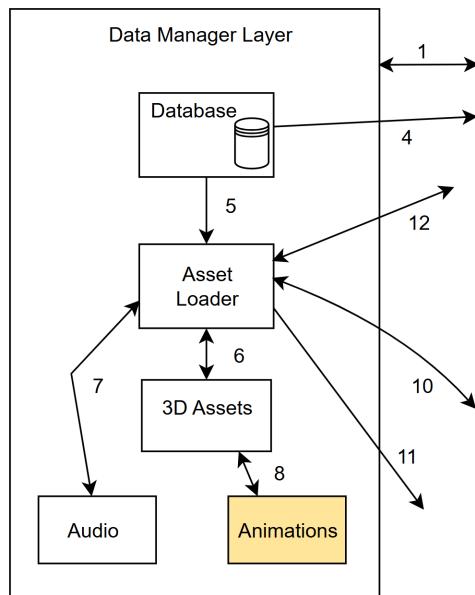


Figure 9: Animations Subsystem description diagram

4.8.1 ANIMATIONS SUBSYSTEM HARDWARE

Animation data is stored on the device's internal storage. At runtime, the CPU and GPU execute the animations, updating object properties each frame.

4.8.2 ANIMATIONS SUBSYSTEM SOFTWARE DEPENDENCIES

This subsystem is entirely dependent on Unity's animation system, known as Mecanim. This includes the Animation, Animator, and Timeline tools.

4.8.3 ANIMATIONS SUBSYSTEM PROGRAMMING LANGUAGES

The animation data itself is not code, but its playback is controlled via triggers sent from C# scripts to the Animator component.

4.8.4 ANIMATIONS SUBSYSTEM DATA STRUCTURES

The two main data structures are:

- **AnimationClip data asset:** Which will store keyframe information for properties of a GameObject over time.
- **AnimatorController state machine:** Which will have trigger ID and leads to clip ID.

4.8.5 ANIMATIONS SUBSYSTEM DATA PROCESSING

Runtime data processing is handled by Unity's Animator engine. The engine:

1. Receives trigger from Interactions or Scene Manager.
2. Matches event to correct clip.
3. Plays animation with synchronized audio if needed.

5 APP LAYER

The App Layer is responsible for everything the user directly sees and interacts with in the AR application. It serves as the main interface between the system's internal data and the user, ensuring that the experience is intuitive, responsive, and educational. This layer is designed with user-friendliness as a core priority, making it easy for users to understand and engage with the AR scenes. The App Layer will allow users to view an interactive AR scene projected onto a physical table or surface. Users can tap on objects within the scene to receive visual or animated responses, such as polluted water becoming clean or trash disappearing. This interaction supports the educational goals of the project by helping users understand the impact of pollution in a visual and engaging way.

5.1 LAYER HARDWARE

In this layer, the main hardware components of the mobile device that allow the user to interact with the application include:

- **Touchscreen:** Captures user input through taps and gestures.
- **Display Screen:** Renders the augmented reality content and user interface elements.
- **Audio Output:** Built-in speakers or headphones provide narration and sound effects to support the educational experience.

5.2 LAYER OPERATING SYSTEM

The App Layer runs on the Android operating system, which provides the environment for executing the augmented reality application:

- Manages access to the device display, touchscreen, and audio hardware.
- Handles rendering support, memory management, and input events for Unity.
- Development and testing are performed on Windows laptops, with deployment to Android devices.

5.3 LAYER SOFTWARE DEPENDENCIES

The App Layer depends on the following software tools and frameworks:

- **Unity Game Engine (2022.3.1):** Used to build and manage visual, interactive, and audio components.
- **AR Foundation and ARCore:** Provide augmented reality features such as surface tracking and marker recognition.
- **Android SDK:** Required for building and deploying the app to mobile devices.
- **Programming Language:** Implemented in C# within the Unity environment.

5.4 SCENE MANAGEMENT

The Scene Management subsystem is responsible for controlling the flow of the augmented reality experience. It manages which scenario is active, handles transitions between different educational scenes, and ensures that the correct assets, animations, and audio are displayed at the right time. This subsystem acts as the central controller for the App Layer and communicates with the Database and Asset Loader in the Data Manager Layer to retrieve the resources needed for each scene.

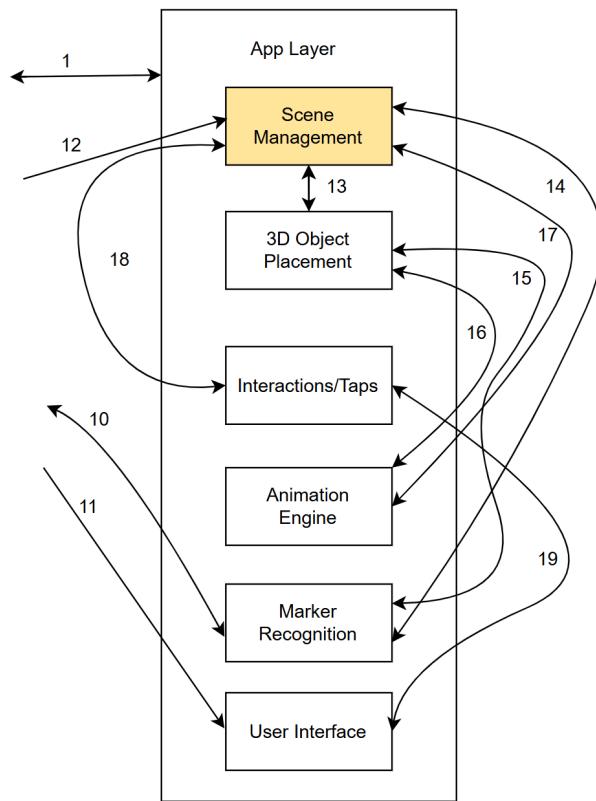


Figure 10: Scene Management subsystem description diagram

5.4.1 SCENE MANAGEMENT SUBSYSTEM HARDWARE

- Uses the device CPU and GPU for processing scene logic and rendering.
- No unique or external hardware is required beyond the mobile device.

5.4.2 SCENE MANAGEMENT SUBSYSTEM OPERATING SYSTEM

Runs on the Android operating system, which provides memory management, rendering support, and event handling that Unity uses to keep scene transitions and updates stable.

5.4.3 SCENE MANAGEMENT SUBSYSTEM SOFTWARE DEPENDENCIES

- **Unity Engine:** Provides the core environment for scene control.
- **SceneManager API:** Used to create, load, and transition between scenes.
- **Unity AR Foundation:** Connects scene logic with AR features such as plane detection and marker tracking.

- **Database and Asset Loader:** Supplies scene data and resources.

5.4.4 SCENE MANAGEMENT SUBSYSTEM PROGRAMMING LANGUAGES

The Device Camera Feed subsystem primarily uses programming languages tied to the development platform (Unity), in this case the programming language is C#.

5.4.5 SCENE MANAGEMENT SUBSYSTEM DATA STRUCTURES

- **SceneConfig:** Stores scene ID, required assets, and trigger events.
- **SceneState:** Tracks the current status of the application, including whether a scene is loading, active, or completed.

5.4.6 SCENE MANAGEMENT SUBSYSTEM DATA PROCESSING

The subsystem coordinates scene flow based on user actions and system events.

1. Read scene metadata from the Database.
2. Request required assets from the Asset Loader.
3. Fetch audio files such as narration and sound effects for the scene.
4. Activate objects and UI elements for the current scene.
5. Handle transitions using Unity asynchronous loading to reduce lag.

5.5 3D OBJECT PLACEMENT

The 3D Object Placement subsystem is responsible for positioning augmented reality objects in the physical environment. It takes the surface information and marker detection results provided by lower layers and uses them to anchor virtual objects to real-world locations. This subsystem ensures that objects appear stable and properly aligned with the table surface, giving the user a realistic and consistent AR experience.

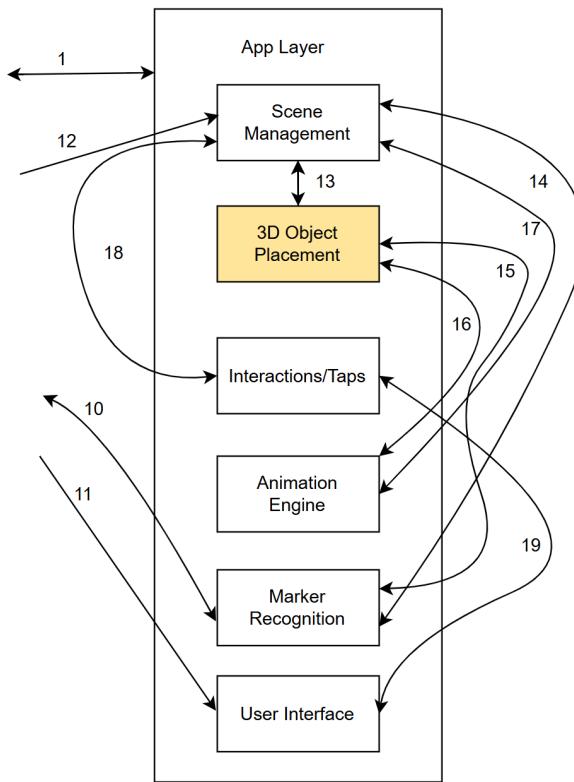


Figure 11: 3D Object Placement subsystem description diagram

5.5.1 3D OBJECT PLACEMENT SUBSYSTEM HARDWARE

The 3D Object Placement subsystem relies entirely on the mobile device's internal processing components. All placement logic and rendering operations run locally on the Android device without requiring any additional equipment.

- Uses only the mobile device's built-in CPU and GPU for placement calculations and rendering.
- No external or specialized hardware is required.

5.5.2 3D OBJECT PLACEMENT SUBSYSTEM OPERATING SYSTEM

This subsystem operates on the Android operating system, which provides essential APIs for rendering, input detection, accelerometer data, and ARCore integration. Unity relies on these OS-level services to translate user touch input and marker positions into world-space coordinates for object placement.

5.5.3 3D OBJECT PLACEMENT SUBSYSTEM SOFTWARE DEPENDENCIES

- **Unity AR Foundation and ARCore:** Handle raycasting, anchor creation, and surface alignment.
- **Unity Transform and GameObject classes:** Manage positional data, rotation, and scaling for virtual objects in the scene.

5.5.4 3D OBJECT PLACEMENT SUBSYSTEM PROGRAMMING LANGUAGES

This subsystem is implemented using the programming language supported by the Unity development environment, which is C#.

5.5.5 3D OBJECT PLACEMENT SUBSYSTEM DATA STRUCTURES

- PlacementConfig: Stores position, rotation, and scale data for placed objects.
- PlacementAnchor: Binds virtual objects to world coordinates supplied by ARCore.

5.5.6 3D OBJECT PLACEMENT SUBSYSTEM DATA PROCESSING

The subsystem processes user interactions and surface detections to place and maintain virtual objects on real surfaces.

1. Perform raycasting to convert screen taps into 3D world coordinates.
2. Validate that the raycast hit lies on a detected plane or tracked marker.
3. Create an anchor and instantiate the virtual object at the detected location.
4. Apply orientation and alignment adjustments to match the surface geometry.
5. Continuously update object transforms to ensure stable placement as the device moves.

5.6 INTERACTIONS/TAPS

The Interactions/Taps subsystem is responsible for handling all user touch inputs that occur during the augmented reality experience. It interprets taps made on the device's touchscreen and maps them to virtual objects in the scene. This subsystem allows users to access educational narration. It is designed to provide intuitive control so that the learning experience feels natural and responsive.

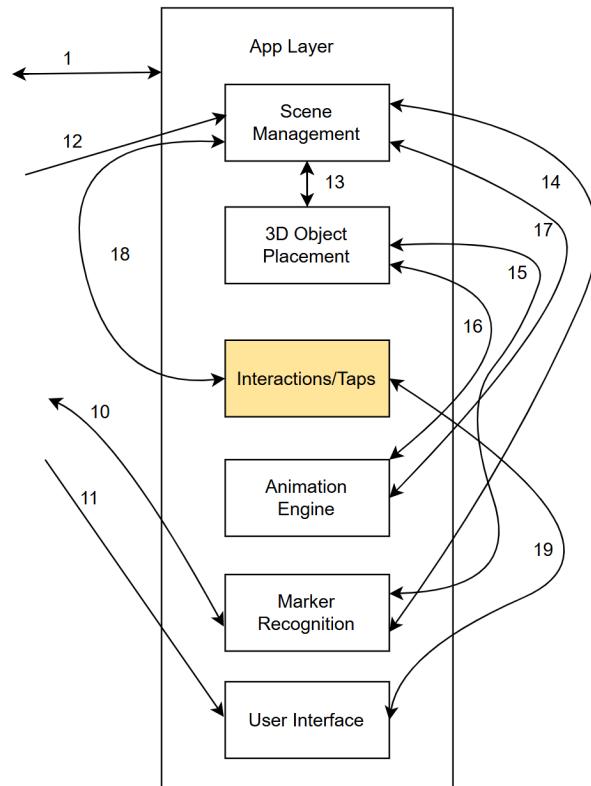


Figure 12: Interactions/Taps subsystem description diagram

5.6.1 INTERACTIONS/TAPS SUBSYSTEM HARDWARE

This subsystem relies entirely on the mobile device's built-in touchscreen to detect user taps and touch gestures. All inputs are processed through the phone's CPU, allowing the AR application to respond quickly without requiring any external devices or peripheral hardware.

- Uses the device touchscreen to capture taps and gestures.
- Relies on the CPU for processing touch inputs.
- No extra hardware is required beyond standard mobile device components.

5.6.2 INTERACTIONS/TAPS SUBSYSTEM SOFTWARE DEPENDENCIES

This subsystem builds on Unity's input and event frameworks as well as AR Foundation's raycasting capabilities. Together, these systems translate user taps into interactions with AR objects or UI elements in the scene.

- **Unity Input System:** Detects tap events.
- **AR Foundation:** Provides raycasting tools to connect gestures to AR objects.
- **Unity EventSystem:** Manages interactions with user interface elements.

5.6.3 INTERACTIONS/TAPS SUBSYSTEM PROGRAMMING LANGUAGES

All interaction logic is written in C#, Unity's primary scripting language. These scripts define how taps are detected, interpreted, and converted into meaningful responses within the AR environment.

- Implemented in C# within the Unity development platform.

5.6.4 INTERACTIONS/TAPS SUBSYSTEM DATA STRUCTURES

To support consistent tap processing, the subsystem uses structured data representations that track gesture properties and define how specific AR objects should respond during user interaction.

- **GestureEvent:** Stores data such as touch position, gesture type, and timestamp.
- **ObjectInteractionConfig:** Defines what action occurs when a user interacts with a specific AR object.

5.6.5 INTERACTIONS/TAPS SUBSYSTEM DATA PROCESSING

Touch gestures go through a clear processing pipeline that transforms raw touch input into AR actions. The system determines what object (if any) was tapped and triggers appropriate visual, auditory, or animated responses to reinforce interactivity.

1. Capture screen touch coordinates from the Input System.
2. Perform a raycast to translate screen space to world space.
3. Determine if a valid AR object or UI element was hit.
4. Read the object's interaction configuration and execute the action (for example, play an animation, show a pop-up, or trigger audio).
5. Provide immediate visual or audio feedback to confirm the interaction.

5.7 ANIMATION ENGINE

The Animation Engine is a software subsystem responsible for rendering visual animations that represent water flow, pollution dispersion, and user interactions within the AR environment. It is built using Unity and deployed on Android devices, serving as a visual layer that enhances user understanding through real-time animations. This subsystem directly communicates with simulation logic and user interface components to present dynamic environmental changes on mobile screens.

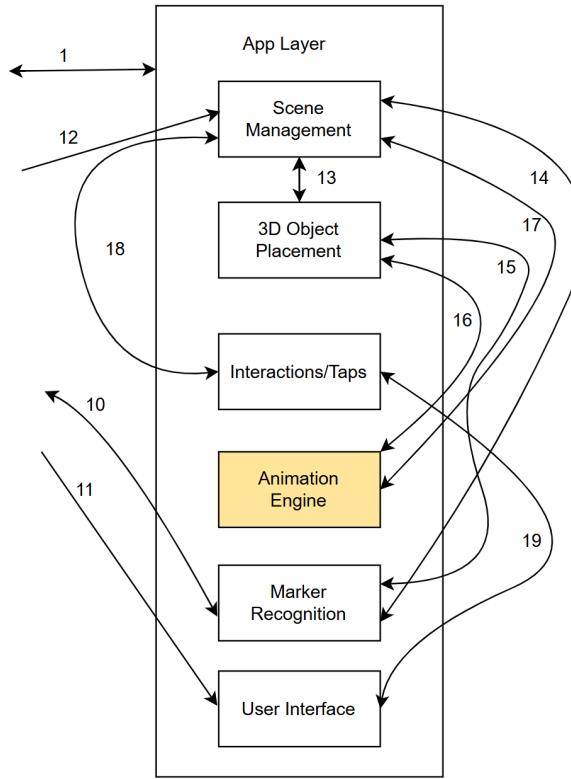


Figure 13: Animation Engine subsystem description diagram

5.7.1 ANIMATION ENGINE SUBSYSTEM HARDWARE

The Animation Engine subsystem operates entirely on the mobile device, relying on the smartphone's built-in processing capabilities to render smooth visual effects. Unity's rendering engine handles all animation playback without the need for external equipment. The subsystem is optimized to take advantage of the device's GPU, ensuring stable performance even during complex AR scenes.

- No external hardware components are required for the Animation Engine.
- All animations are processed on the user's Android device using Unity's built-in rendering engine.
- The subsystem leverages the phone's integrated GPU for optimal performance.

5.7.2 ANIMATION ENGINE SUBSYSTEM SOFTWARE DEPENDENCIES

The subsystem uses Unity 2022.3.1 as the main development environment. AR Foundation and ARCore XR Plugin are required to support AR marker tracking and surface rendering. Unity's Animation Rigging package is also used to support character animation and object movement.

5.7.3 ANIMATION ENGINE SUBSYSTEM PROGRAMMING LANGUAGES

The subsystem is developed using C# , Unity's default scripting language. This language is used for all animation control scripts, event triggers, and state transitions.

5.7.4 ANIMATION ENGINE SUBSYSTEM DATA STRUCTURES

- Custom classes and structs are used in the animation engine.
- These represent animation states, environmental triggers, and object attributes.

5.7.5 ANIMATION ENGINE SUBSYSTEM DATA PROCESSING

Animations are triggered and handled through Unity's animation system as follows:

1. Animations are triggered based on marker recognition and scene input.
2. Unity's Animator Controller and State Machine Behaviors manage the animations.
3. Events are processed in real-time based on marker visibility and proximity.
4. These events enable simulations of pollution flow and water dynamics.
5. Unity's Update loop and coroutine system manage asynchronous animation handling.

5.8 MARKER RECOGNITION

The Marker Recognition subsystem enables the mobile app to detect and track physical image markers using the device camera. It is implemented in Unity and deployed to Android using Unity's Android build tools. Upon marker detection, C# scripts trigger AR content overlays for interactive visualization.

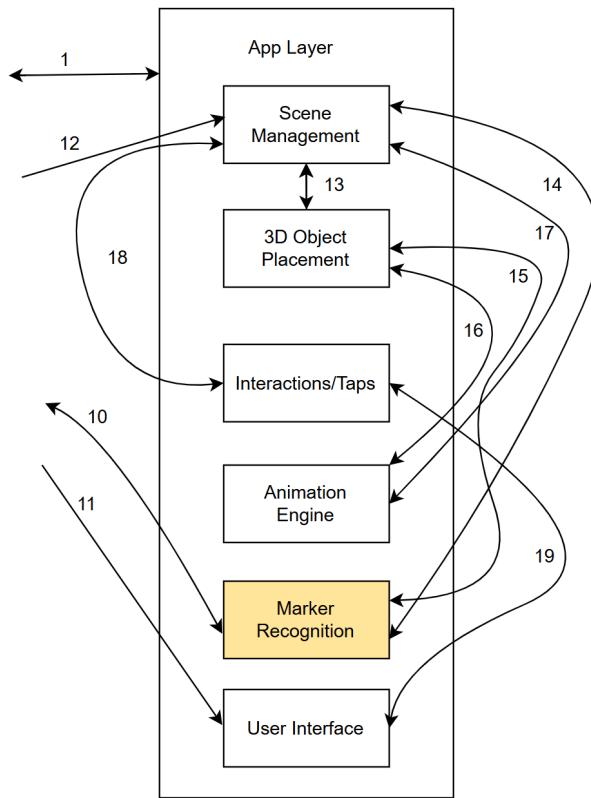


Figure 14: Marker Recognition subsystem description diagram

5.8.1 MARKER RECOGNITION SUBSYSTEM HARDWARE

This subsystem relies entirely on the smartphone's existing hardware. No additional equipment is needed, making the system accessible and easy to deploy.

- Uses the smartphone's built-in camera.
- No external hardware is required.
- AR markers are detected using the phone camera.

5.8.2 MARKER RECOGNITION SUBSYSTEM OPERATING SYSTEM

The Marker Recognition component runs on Android, relying on Unity's Android build pipeline to interface with ARCore. The operating system and associated build tools ensure compatibility and hardware access.

- Requires the **Android OS**.
- Unity project is exported as an Android build using **Unity Android Build Support**.
- Build process includes **Gradle** and **Android SDK tools**.

5.8.3 MARKER RECOGNITION SUBSYSTEM PROGRAMMING LANGUAGES

The subsystem is implemented using C#, which controls marker events, object spawning, and scene interaction logic.

5.8.4 MARKER RECOGNITION SUBSYSTEM DATA STRUCTURES

The system uses lightweight, efficient data structures to associate detected markers with AR content. These structures support fast lookups during real-time detection.

- Marker data stored as a list of `TrackableId` objects linked to marker metadata.
- Dictionaries map marker IDs to labels or scene actions for fast lookup.

5.8.5 MARKER RECOGNITION SUBSYSTEM DATA PROCESSING

Once ARCore detects a marker, the system processes its pose and identity to determine what virtual content to display. This processing drives all subsequent AR interactions.

- Data is filtered, parsed, and mapped to a virtual object.
- The object is rendered in real time relative to the marker pose.

5.9 USER INTERFACE

The User Interface (UI) subsystem handles all user interactions within the AR application. It includes the UI menus, buttons, and on-screen information that guide the user through the AR experience. Developed in Unity, the UI is designed to work seamlessly on Android devices while integrating with AR features in real time.

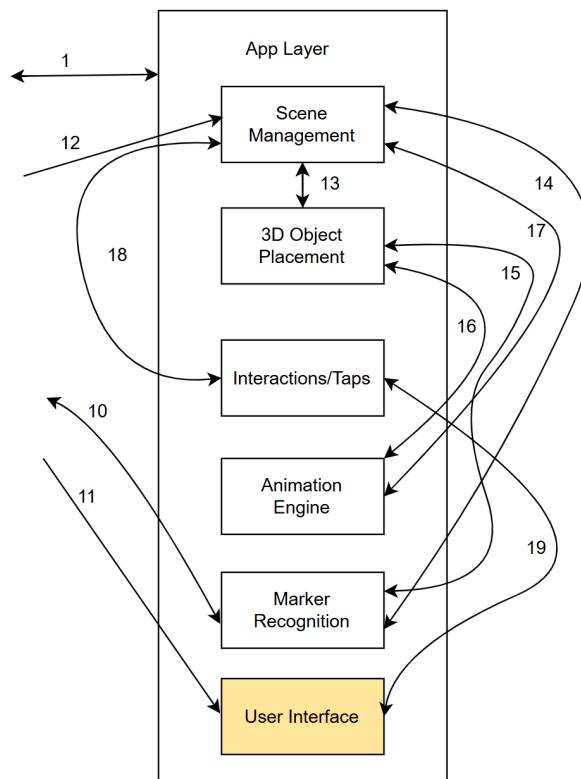


Figure 15: User Interface subsystem description diagram

5.9.1 USER INTERFACE SUBSYSTEM HARDWARE

The UI depends on the capabilities of modern smartphones, particularly their displays and input mechanisms. All interactions occur directly on the mobile device.

- **Phone Device:** Targeted for phones.
- **Built-In Phone Camera:** Used for real-time marker detection and AR rendering.
- **Touchscreen Input:** Taps, swipes, and gestures.

5.9.2 USER INTERFACE SUBSYSTEM SOFTWARE DEPENDENCIES

The UI relies on several Unity and Android components to operate correctly, including ARCore support and mobile build tools.

- **Unity 3D Engine**
- **ARCore via AR Foundation**
- **Android SDK and NDK**
- **Gradle Build System**

5.9.3 USER INTERFACE SUBSYSTEM PROGRAMMING LANGUAGES

The UI logic is implemented in C#, which controls button behavior, animations, and dynamic updates to the screen.

5.9.4 USER INTERFACE SUBSYSTEM DATA STRUCTURES

To organize UI elements and AR objects, the subsystem uses several C# data structures and Unity-specific assets. These structures support organizing gameplay states, UI layouts, and AR interactions.

- **C# Classes / Structs:** e.g., Pollutant, Stakeholder, EventState.
- **Lists / Dictionaries:** For AR objects, UI elements, and animations.
- **ScriptableObjects:** Data-driven configuration of scenarios.

5.9.5 USER INTERFACE SUBSYSTEM DATA PROCESSING

The UI updates dynamically as markers are detected and AR content changes. Various Unity systems help ensure that rendering and animations remain smooth during AR interactions.

- **Marker Anchoring:** Via ARCore through AR Foundation.
- **Visual Effects:** Unity Animation and Particle Systems.
- **Optimized Rendering:** URP to maintain 60 FPS.

6 APPENDIX A

Include any additional documents (CAD design, circuit schematics, etc) as an appendix as necessary.

REFERENCES