Technische Hochschule Nürnberg

EduArt

Fakultät Elektrotechnik Informationstechnik Feinwerktechnik

EdukARt: A mixed-reality racing game for physical robot teleoperation

Thesis for the degree Bachelor of Media Engineering (B. Eng.)

by

Sina Steinmüller

Student ID number: 3637368

Semester: winter 26/27

Submission date: 26.10.2026

Examiners: Prof. Dr. rer. nat. Matthias Hopf

Prof. Dr. rer. nat. Stefan May

© Sina Steinmüller, 2026

hello@sinamatic.de

This work, including its parts, is **protected by copyright.** Any use outside the narrow limits of copyright law without the consent of the author is prohibited and punishable by law. This applies in particular to reproductions, translations, microfilming and storage and processing in electronic systems.

ohm Technische Hochschule Nürnberg

Hinweis: Diese Erklärung ist mit Originalunterschrift (nicht gescannt) in das Papierexemplar der Abschlussarbeit fest einzubinden. Eine Spiralbindung ist nicht zulässig. Das digitale Exemplar enthält einen Scan der Erklärung mit Unterschrift.

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Steinmüller Vorname: Sina Matrikel-Nr.: 3637368

Fakultät: Elektrotechnik Feinwerktechnik Informations Studiengang: Media Engineering

Semester: 10

Titel der Abschlussarbeit:

EdukARt: A mixed-reality racing game for physical robot teleoperation

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe. Außerdem versichere ich, dass der Hauptteil des Papierexemplares und des digitalen Exemplares identisch sind. Das digitale Exemplar enthält, falls gefordert, lediglich weitere Anlagen.

Sofern KI-Werkzeuge in der gegenständlichen Prüfung durch die zuständigen Prüfenden zugelassen waren, ist Folgendes zu bestätigen:

☑ Hiermit versichere ich, dass ich die vorgelegte Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Ich übernehme als Autorin oder Autor beim Einsatz von IT-/KI-gestützten, insbesondere generativen KI-Systemen, die Verantwortung für die inhaltliche Richtigkeit und den rechtskonformen Einsatz (insbesondere Lizenzrecht, Urheberrecht, Datenschutz, Geheimhaltung).

Nürnberg, 22.10.26 [signature]

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung der/des Studierenden zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit ☑ genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,

☐ genehmige ich **nicht,**

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von 0 Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format an die Betreuer übermittelt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Nürnberg, 22.10.26 [signature]

Ort, Datum, Unterschrift Studierende/Studierender

Datenschutz: Die Antragstellung ist regelmäßig mit der Speicherung und Verarbeitung der von Ihnen mitgeteilten Daten durch die Technische Hochschule Nürnberg Georg Simon Ohm verbunden. Weitere Informationen zum Umgang der Technischen Hochschule Nürnberg mit Ihren personenbezogenen Daten sind unter nachfolgendem Link abrufbar: **https://www.th-nuernberg.de/datenschutz/**

Abstract

This thesis presents the development of the EdukARt prototype, a mobile mixed-reality racing game. Players steer a physical robot along a virtual race track, interacting with virtual items and obstacles in a real environment. The thesis investigates how an existing mobile robot can be integrated as an active game element into such a game without extensive modifications to its hardware or existing control software.

A prototype racing game was implemented as a native iOS application using Swift, ARKit, and RealityKit. Players control the mobile robot Eduard through an iPhone along a self-created virtual track. Items and obstacles affect its movement and lighting and generate virtual effects, which influence the race time.

Communication uses the robot's existing Wi-Fi network and ROS2 interfaces through an additional Python-based UDP-to-ROS2 bridge. Following initial experiments with LiDAR-based environmental reconstruction, an AprilTag-based spatial reference was developed, which represents all game elements, including the robot, in a shared two-dimensional map. This map serves as a basis for course creation, interaction detection, and placement of virtual content within the physical surroundings.

Testers pointed out that the concept was enjoyable and visually appealing and expressed interest in building and using their own robot for the game. The prototype demonstrates an integration approach that connects virtual game events with physical robot behavior using existing robot interfaces and a shared marker-based spatial reference. Its technical feasibility was demonstrated within the tested Eduard and iPhone setup.

Keywords

mixed reality, robot teleoperation, mobile game, iOS, Swift, ARKit, Eduard, ROS2, AprilTags, human-robot interaction, virtual race

Kurzdarstellung

Diese Arbeit beschreibt die Entwicklung von EdukARt, einem Prototyp für ein mobiles Mixed-Reality-Rennspiel. Die Spielenden steuern einen physischen Roboter entlang einer virtuellen Rennstrecke und interagieren dabei mit virtuellen Items und Hindernissen in ihrer realen Umgebung. Untersucht wird, wie sich ein vorhandener mobiler Roboter ohne größere Änderungen an seiner Hardware oder bestehenden Steuerungssoftware als aktives Spielelement in ein solches Spiel integrieren lässt.

Der Prototyp dieses Rennspiels wurde als native iOS-Anwendung mit Swift, ARKit und RealityKit umgesetzt. Spielende steuern den mobilen Roboter Eduard mit einem iPhone entlang einer selbst erstellten virtuellen Strecke. Items und Hindernisse beeinflussen seine Bewegung und Beleuchtung und erzeugen virtuelle Effekte, die sich auf die Rennzeit auswirken.

Die Kommunikation erfolgt über das vorhandene WLAN und die ROS2-Schnittstellen des Roboters mithilfe einer zusätzlichen, in Python implementierten UDP-ROS2-Brücke. Nach ersten Versuchen mit LiDAR-basierter Umgebungserfassung wurde eine AprilTag-basierte räumliche Referenz entwickelt, die alle Spielelemente wie auch den Roboter in einer gemeinsamen zweidimensionalen Karte abbildet. Diese dient als Grundlage für die Streckenerstellung, Interaktionserkennung und Platzierung virtueller Inhalte in der physischen Umgebung.

Testende hoben hervor, dass das Konzept Spaß macht und visuell ansprechend ist, und zeigten Interesse daran, einen eigenen Roboter für das Spiel zu bauen und zu nutzen. Der Prototyp zeigt einen Integrationsansatz, der virtuelle Spielereignisse über vorhandene Roboterschnittstellen und ein gemeinsames markerbasiertes Bezugssystem mit dem Verhalten des physischen Roboters verbindet. Die technische Machbarkeit wurde im getesteten Aufbau mit Eduard und einem iPhone gezeigt.

Contents

1 Introduction and Motivation **1**

2 Background and Related Work **3**
2.1 Robotics Foundations and Simulation 3
2.1.1 Requirements of Industrial and Mobile Robotic Systems 3
2.1.2 Simulation and the Sim-to-Real Gap 4
2.1.3 Mixed Reality in Robotics and Industrial Applications 5
2.2 Mixed Reality in Games and Entertainment 6
2.3 Mixed Reality Games with Physical Robots 7
2.3.1 Tabletop Frameworks for Robot Interaction 7
2.3.2 Physical Robots as Active Game Elements 8
2.3.3 PiGaming: A ROS-Based Mixed-Reality Gaming Platform 8

3 Technical Foundations and Technology Selection **10**
3.1 Mobile Mixed Reality Technologies 10
3.1.1 Game Engines for Mixed-Reality Applications 10
3.1.2 Native iOS Development 12
3.2 Robot Eduard as the Physical Platform 15
3.2.1 Company and Project Context 16
3.2.2 Technical Capabilities of Eduard 18

4 Design and Implementation of EdukARt **20**
4.1 Game Concept and Scope of the Prototype 20
4.2 Development Process and Technical Environment 21
4.2.1 Development Environment and Software Versions 21
4.2.2 Technical Environment of Eduard 22
4.2.3 AI-Assisted Development Process 23
4.3 Implementation and System Overview 24
4.3.1 Robot Remote: Connecting and Controlling Eduard 25
4.3.2 Courses: Creating a Map Using a Spatial Reference System 34
4.3.3 Virtual Game Elements: Creation and Interaction 46
4.3.4 Start Game: Running a Race 57

5 Results and Discussion **60**

6 Conclusion **65**

7 Future Work **67**

List of Figures **68**

List of Tables **70**

Bibliography **71**

Glossary **77**

Chapter 1

Introduction and Motivation

Digital devices are part of everyday life, with people from the age of six spending considerable time in front of screens [1]. Screen time is increasing even among children [2], while physical inactivity among adults is increasing worldwide [3].

Digital games and apps can, however, encourage movement and social interaction. Pokémon GO requires players to move through the real world, increasing physical activity while also bringing people together at shared locations [4, 5, 6]. Mario Kart emphasizes shared play and is commonly enjoyed as a party game [7], while fitness apps use digital elements to motivate physical activity [8]. Although these games can encourage physical activity and social interaction, they lack specific educational content.

The educational potential of games becomes particularly relevant as AI, automation and robotics play an increasingly important role in everyday life. Smart-home devices, for example, are already used in almost every second household [9]. Despite the growing market for game-based learning [10], student performance in international comparisons has declined [11], and Germany's education system faces calls for modernization and better integration of digital learning [12].

The idea behind EdukARt is to bring together the aspects outlined above in a mobile mixed reality (MR) racing game that combines physical movement, as in Pokémon GO, with shared and social play, as in Mario Kart. By using a physical robot as an active game element that players can control, EdukARt aims to introduce players to robotics and spark interest that could lead them to join school initiatives such as RobotsUnite [13], continue in university teams such as autonOhm [14], and eventually work in robotics or establish a company such as EduArt Robotik [15].

RobotsUnite provided a particular source of inspiration by bringing teams with different robots together in shared challenges. Within this initiative, groups of students work with motivated teachers outside regular school hours to build, program, and test robots before competing against one another [13]. Similar to Mario Kart, their robots must overcome obstacles and complete different tasks [13], such as reaching and bursting balloons placed in difficult-to-access locations [16].

EdukARt aims to give existing robots an additional playful use by making them active game elements, particularly those already built or operated by school teams, university teams, and robotics companies for competitions, leagues, or applications such as RoboCup [17, 18, 19]. The game could allow teams to compete against other robots at robotics events, engage visitors at trade fairs, or simply play together in a schoolyard.

This thesis investigates how an existing mobile robot can be integrated into such a game without extensive modifications to its hardware or existing control software. The resulting prototype is used to explore the technical implementation and its practical limitations. Whether EdukARt can motivate people in the long term to be more physically active, develop robotics skills, or interact more socially lies beyond the scope of this first proof of concept and requires further investigation.

The next chapter, Background and Related Work, introduces relevant robotics foundations, MR applications in robotics and entertainment, and existing games involving physical robots. Based on these foundations, Chapter 3, Technical Foundations and Technology Selection, explains the selection of the development technologies and robot platform. Chapter 4, Design and Implementation of EdukARt, presents the game concept, development process, and implementation of EdukARt.

It should be noted that large parts of the prototype's underlying software were developed using AI-assisted tools, which is discussed in more detail in Chapter 4. Chapter 5, Results and Discussion, discusses the developed integration approach in relation to the research question, drawing on testing observations and user feedback to identify its practical limitations. Chapter 6, Conclusion, summarizes the findings and contribution of the thesis, while Chapter 7, Future Work, outlines further development and research opportunities.

Chapter 2

Background and Related Work

The idea behind EdukARt brings together two fields: robotics and entertainment. Both fields use visual representations on screens, such as simulations and sensor visualizations in robotics or virtual environments in games, but their requirements and objectives differ. This chapter provides an overview of both fields and examines existing approaches that combine physical robots with MR applications and games.

2.1 Robotics Foundations and Simulation

Looking over the shoulder of a software developer working in mobile robotics, the content displayed on the screen may initially resemble a video game. It often includes an abstract map of the environment containing a simplified representation of a robot. In other cases, a colorful three-dimensional Point Cloud (Figure 3.4) of the surrounding space is shown and continuously updated. When observed for long enough, the robot may eventually begin to move or interact with its environment, sometimes followed by a celebration from the developer responsible. At first glance, such a visualization or simulation could easily be mistaken for a game. Its purpose, however, is fundamentally different.

2.1.1 Requirements of Industrial and Mobile Robotic Systems

While the track display in Mario Kart gives players a rough indication of their position and remaining laps, a robot's navigation map must provide sufficiently accurate spatial information for localization and motion planning [20]. In systems involving physical robots, inaccuracies can also affect their actual movement. Accuracy, reliability and safety are therefore essential requirements in industrial and mobile robotics [21].

Robots are complex systems that operate in the physical world and must respond to events that cannot always be predicted. Incorrect decisions or movements may damage the robot itself or endanger people in its surroundings. Furthermore, experiments with a physical robot are often expensive and time-consuming, as they require access to

the robot, a suitable environment, and personnel to conduct and supervise the tests. Simulations provide a controlled environment for testing robot behavior without risking physical damage and reduce developers' dependence on continuous access to the physical robot, which may be the only unit available.[21]

A commonly used tool for simulating robotic applications is Gazebo. It provides physics and sensor simulation and can be integrated with ROS (Robot Operating System), making it suitable for developing and testing robotic software.[22, 21] ROS is an open-source software framework that enables message-based communication between the software components of a robot, its sensors, actuators, and external computers. Its modular structure and support for robots from different manufacturers allow a wide range of robotic components to be integrated into a common system.[20]

For example, a mobile robot may use a LiDAR sensor mounted on top to scan its surroundings, while additional Time-of-Flight Sensors measure distances at individual points around the robot. These sensors may come from different manufacturers. Their respective ROS drivers translate the sensor data into defined ROS messages, which are published through topics and can be processed by other components. A controller can subscribe to these messages, evaluate the measured distances, and stop the robot when an obstacle is detected. If a sensor is replaced, large parts of the remaining software can continue to be used, provided that the new component supplies compatible messages and data [20].

Whatever tools are used, developers still need to imagine how the simulated behavior would work in the real world. They must consider limits that may not be shown or may be hard to judge in the simulation [21]. For example, an industrial robot arm might move without problems in simulation but hit a low factory ceiling in reality if the ceiling is not included in the model.

2.1.2 Simulation and the Sim-to-Real Gap

Even a detailed digital model remains an approximation of the physical system. Accurately modelling the robot is already a considerable effort, and the environment in which it operates must also be represented with sufficient detail. Moreover, factors such as wheel friction, sensor noise, lighting conditions, and communication latency are difficult to reproduce accurately. The resulting discrepancy between simulated and real-world behavior is commonly referred to as the *sim-to-real gap*.[21, 23]

Recent technologies provide new possibilities for reducing this gap. One promising approach is the use of game engines such as Unreal Engine or Unity, which can provide

visually detailed environments, advanced physics simulation, and flexible interfaces for integrating robotic systems.[23, 24]

Game engines are already used in several robotics simulation environments. Kang et al., for example, combine ROS2 with Unreal Engine to provide a visually and physically detailed environment for reinforcement learning with a TurtleBot [23]. Similarly, Li et al. use Unreal Engine to generate photorealistic data for the perception system of an agricultural robot [25]. These approaches extend conventional robot simulation by providing more detailed virtual environments, but they still depend on a virtual representation of the robot and its surroundings.

2.1.3 Mixed Reality in Robotics and Industrial Applications

The simulation environments described above represent the robot and its surroundings virtually. Virtual reality (VR) immerses users in a fully virtual environment, whereas augmented reality (AR) adds virtual content to a view of the real world. Mixed reality (MR) combines physical and virtual elements within a shared environment in which they can interact.[24, 21] Since the distinction between AR and MR is not always clear-cut, this thesis generally uses the term MR for applications that combine physical and virtual elements, including those that could also be described as AR.

Pentenrieder et al. overlay virtual planning data onto camera images of an existing production environment. This allows planners to compare proposed modifications with the actual factory, identify inconsistencies, measure distances, and detect possible collisions before changing the physical production site.[26]

A more robotics-focused application of MR is to visualize information from a physical robot directly within its operating environment and to use the virtual content for interaction with the robot. Leber and Dalm use a HoloLens application developed with Unreal Engine to display information such as the robot's position, laser measurements, map, and planned path in relation to its physical surroundings. Users can also place virtual target points and send the physical robot to the corresponding locations through ROS.[20]

Chen et al. go beyond visualization by extending a Gazebo-based simulation with a physical robot and virtual objects. Sensor data from the real robot are combined with simulated sensor data before being transmitted to the robot software. Consequently, a virtual obstacle can appear in the robot's laser measurements and cause the physical robot to avoid it as if it were part of the real environment.[21, 27]

Tae et al. apply a related approach to the testing of autonomous ground vehicles. Their system places a physical vehicle in a real outdoor environment while adding virtual

objects and obstacles to its camera images and point clouds. The combined sensor data are then processed by the vehicle's actual perception and control software, allowing potentially dangerous or difficult scenarios to be tested without physically constructing every obstacle.[28]

MR in robotics can therefore connect physical robots, real sensor measurements, and virtual components within a shared environment. The examined applications use this connection to improve visualization, create controlled test scenarios, and bring simulated conditions closer to real-world operation while reducing costs and safety risks.

2.2 Mixed Reality in Games and Entertainment

MR in games and entertainment follows different requirements from its use in robotics. Instead of reproducing reality as accurately as possible, it primarily aims to create an engaging experience by extending or deliberately changing the physical world. This is why players encounter virtual Pokémon in parks rather than ordinary squirrels.

Niantic's Pokémon GO extends real locations with virtual content and was developed in Unity. Players move between locations in the physical world and encounter Pokémon along the way. In the game's camera view, these virtual characters can be placed within the player's surroundings and viewed from different positions.[29, 30]

A less location-dependent example is The Machines by Directive Games. The game places a virtual battlefield on a physical surface. Players view it through a mobile device and move around the surface to observe the battlefield from different perspectives. Their physical viewing position thereby becomes part of the game strategy.[31]

The Game AR Robot by PlaySlide combines parts of both concepts by placing virtual fighting robots within the player's physical surroundings. Players can train and customize their robot before competing against other robots locally or online.[32]

In these examples, however, the physical environment primarily provides a spatial setting for the virtual content. Apart from moving through the environment or selecting a surface on which the content is placed, the physical surroundings have little influence on the game state.

Mario Kart Live: Home Circuit by Nintendo, extends this concept by making the physical environment part of the interaction instead of using it as a setting for virtual content. Players construct a course using physical gates and drive a camera-equipped kart through the real environment. The camera transmits a live image to the Nintendo Switch, where virtual opponents, items, and track elements are added. Events within the game also

affect the physical kart: it accelerates when receiving a boost and stops temporarily when hit by an item.[33]

BattleKart brings this principle to a larger and more equipment-intensive setting. Players sit in real electric karts and drive through a darkened hall in which tracks and game elements are projected onto the floor using multiple projectors. Sensors and electronic components installed throughout the hall and in the karts track the vehicles and allow players to collect virtual items and interact with other karts and projected game elements.[34]

Mario Kart Live and BattleKart go beyond placing virtual content within a physical setting. In both games, a real vehicle becomes part of the game state and reacts to virtual events. However, this interaction relies on hardware and environments designed specifically for the respective game. The following section reviews existing mixed-reality games and interaction frameworks involving physical robots.

2.3 Mixed Reality Games with Physical Robots

Early examples of such games can be found in interactive tabletop systems, which combine a horizontal physical play area with projected or displayed virtual content and physical robots.

2.3.1 Tabletop Frameworks for Robot Interaction

An example of such a tabletop system is IncreTable. It combines small physical robots with virtual game elements projected onto a table. The robots move along projected markers and can influence both physical and virtual objects.[35]

RoboTable extends this approach into a framework for interacting with robots on a touch-sensitive tabletop. In addition to the robots, the system detects touch gestures and physical objects equipped with visual markers. These inputs can be used to control the robots or influence virtual content, which could also be used to teach concepts related to robotics and programming.[36]

While IncreTable and RoboTable primarily investigate technical frameworks for interaction between physical robots and virtual content, other works place greater emphasis on the robot's role within a specific game.

2.3.2 Physical Robots as Active Game Elements

MRMaze uses a Sphero, a small spherical robot that moves by rolling and can be controlled through a smartphone application, in a cooperative game for two players. One player controls the robot through a maze projected onto the floor, while the second player receives additional information about obstacles and communicates it to the robot operator. An overhead camera tracks the robot by its color, while Unity processes virtual obstacles, collectable objects, and collisions. The imperfect control of the robot is deliberately incorporated into the gameplay. The robot is therefore not merely a physical representation of a virtual game character; its actual movement characteristics directly influence the game.[37]

RobotQuest transfers a similar concept to a larger floor-projected environment. A physical robot follows a virtual path and encounters projected opponents along the way. Players influence the game by placing physical Bluetooth beacons near the robot to select opponents and abilities. The projected path determines the robot's direction, while the robot's position and the placement of the beacons influence the virtual game state. ROS connects the robot to the remaining system, while the Unity game engine manages the game logic and projected content.[38]

2.3.3 PiGaming: A ROS-Based Mixed-Reality Gaming Platform

A broader platform-based approach is presented by PiGaming. Instead of implementing a single game, the system supports several MR games in which two, three, or four physical robots interact with virtual game elements. PiGaming includes PiTanks, a time-based combat game; Robot Factory, an industrially inspired delivery game; and Robot Race, a lap-based racing game.[39]

Virtual walls, projectiles, paths, and other game elements are projected onto a two-dimensional mat on which the physical robots move. A camera captures the play area, while visual markers attached to the robots are used to determine their positions. The players control the robots using joysticks, and ROS connects the robots, tracking system, and game logic. This enables movements of the physical robots and events in the virtual game to be processed within a shared game state.[39]

The work also describes practical limitations of the physical setup. Its size and limited availability made it difficult to test changes or develop new game functionality whenever required. Therefore, an additional simulation was developed in which all three games can be executed without access to the physical setup.[39]

Taken together, the reviewed fields provide different requirements and approaches for connecting physical robots with virtual content. Robotics contributes established communication interfaces and simulation practices, while industrial MR focuses on accurate and safe interaction between physical and virtual components. Entertainment applications demonstrate how physical robots can become active game elements whose positions, movements, and technical characteristics influence gameplay, while virtual events define their tasks or affect their behavior. The following chapter builds on these foundations to explain the choice of development technologies and introduce Eduard as the physical robot platform for EdukARt.

Chapter 3

Technical Foundations and Technology Selection

As discussed in Background and Related Work, mobile robotics applications often use ROS for communication between software components and tools such as Gazebo for simulation. Depending on the application, these simulations aim to reproduce relevant aspects of the physical robot and its environment with sufficient accuracy. Game engines such as Unity and Unreal Engine can extend conventional robotics simulations by providing more detailed visualization, rendering, and interaction capabilities.

3.1 Mobile Mixed Reality Technologies

3.1.1 Game Engines for Mixed-Reality Applications

Unity and Unreal Engine both support cross-platform game development and mobile mixed reality applications, but they emphasize different development workflows. Unity is generally considered more accessible and well suited to rapid prototyping, mobile applications, and 2D games, whereas Unreal Engine places greater emphasis on high-fidelity 3D environments and real-time graphics.[40]

For the initial EdukARt prototype, an Apple Silicon MacBook Pro was used as the development system, while an iPhone 13 Pro served as the target and test device. Compatibility with this setup was therefore relevant when evaluating the development tools. The following Table 3.1 provides a direct comparison of both engines. General characteristics are based on [40], while the comparison of macOS support is based on [41].

Criterion	Unity	Unreal Engine
Primary use cases	Indie, 2D, and mobile games as well as rapid prototyping	High-end 3D games, cinematic visualization, and graphically demanding multiplayer applications

Criterion	Unity	Unreal Engine
Primary programming language	C#	C++
Graphics	High-quality rendering; lower photorealistic ceiling than Unreal Engine	High-end real-time rendering, photorealistic visualization
Source-code access	Closed source	Source available under Epic's proprietary license
Learning curve	Easier than Unreal Engine	More difficult than Unity
Asset ecosystem	Large and diverse asset store	Smaller asset store, generally focused on high-quality assets
Mobile AR	Supported through AR Foundation with platform providers such as ARKit (Apple) and ARCore (Google)	Supported through the engine's handheld AR framework with platform providers such as ARKit (Apple) and ARCore (Google)
macOS support	Strong native Apple-Silicon and iOS/Xcode support	Supported on Apple Silicon, but some advanced rendering features are unavailable

Table 3.1: Comparison of Unity and Unreal Engine.

Godot was also considered because it had already been used successfully in a previous project. However, its ecosystem for mobile mixed-reality applications is less established than those of Unity and Unreal Engine, so it was not evaluated further.

The reviewed literature includes robotics applications that combine ROS or ROS2 with both Unity and Unreal Engine. Compatibility with ROS was therefore not considered a distinguishing criterion between the two engines. Unity was evaluated first because it was also used in several of the reviewed mixed-reality games involving physical robots.

To become familiar with its mobile MR workflow, a small Unity test application was created by following an introductory tutorial. However, the recommended Unity 6.3 LTS release caused several package installation errors on the development system, consisting of macOS 26.0.1 running on an Apple Silicon MacBook Pro. Clearing the package cache, reinstalling Unity and Unity Hub, disabling network-related services, and updating macOS did not resolve the problem. A functional project could only be created after installing an earlier Unity version. Although Unity generally provides strong support for Apple Silicon and iOS development, this demonstrated the additional dependencies and configuration effort introduced by the engine.

Unreal Engine was subsequently evaluated as an alternative. Its initial setup also resulted in problems involving Xcode, the command-line tools, and the Metal toolchain. Further compatibility and packaging problems prevented its standard Handheld AR template from being deployed to the iPhone. The available documentation did not resolve the encountered compatibility and packaging problems.

Both evaluations required substantial configuration and troubleshooting before any functionality relevant to the prototype could be implemented. Moreover, using either engine introduces an additional abstraction layer between the application and the platform's native spatial framework. In return, this abstraction enables a largely device-independent and cross-platform implementation: depending on the target platform, the engines access ARKit on Apple devices or ARCore on Android devices.

This trade-off raised the question of how important cross-platform compatibility was for the initial EdukARt prototype. A cross-platform implementation would provide a stronger foundation for continued development and make the application accessible to a broader range of users. However, it would also increase the initial development effort.

3.1.2 Native iOS Development

Given the intended use of an iPhone and the game-engine difficulties described in Game Engines for Mixed-Reality Applications, native iOS development without a game engine was considered as an alternative.

Previous work in university and robotics projects had repeatedly involved compatibility and configuration difficulties with ROS, virtual machines, and Gazebo across several MacBook generations. Development in these projects are primarily based on Linux or Windows, while Macs are rarely used. Since macOS was already central to the established workflow in graphics and media production, these experiences motivated a personal interest in exploring whether Apple's native tools could also provide a practical and more integrated workflow for developing a robotics-related MR application.

However, limiting the application to iOS would exclude Android users and considerably reduce its potential audience. As an indication of the global platform distribution, iOS accounted for approximately 31% of the worldwide mobile web usage measured by Statcounter in September 2026 [42]. A developer-oriented comparison presents ARCore as the preferable option for broader global reach, while favoring ARKit for more consistent AR performance within Apple's hardware ecosystem [43]. This suggests a trade-off between potential audience size and the consistency of the targeted hardware and software environment.

An initial technical argument in favor of focusing on Apple devices was the LiDAR sensor integrated into the available iPhone 13 Pro, as this technology is used in robotics to measure distances and support mapping and localization, as discussed in Requirements of Industrial and Mobile Robotic Systems. The smartphone's integrated sensor therefore appeared promising for capturing the spatial game environment. If the game depended on this hardware, restricting the implementation to LiDAR-equipped iPhones could appear technically justified. However, this would further reduce its potential audience by excluding other iPhone users as well.

Before accepting these restrictions, it was necessary to examine whether the required spatial information could also be obtained without dedicated LiDAR hardware and whether its advantages justified the platform choice. The following section therefore compares Apple's ARKit with Google's ARCore, focusing on spatial tracking, depth sensing, and alternatives to LiDAR, while considering the additional development effort that supporting both platforms would involve.

3.1.2.1 Mobile Depth Sensing: LiDAR and ARCore

On the iPhone 13 Pro, ARKit can combine the measurements of the LiDAR sensor with camera and motion data. The resulting depth information can be used to estimate physical surfaces and reconstruct the surrounding environment as a polygonal mesh. This supports the spatial placement and occlusion of virtual content as well as interactions between virtual objects and the reconstructed physical environment [44].

Android devices can provide comparable information through the ARCore Depth API, which primarily estimates depth by comparing camera images captured from different positions. Its results therefore generally improve as the device is moved through the environment. If a device additionally provides a hardware depth sensor, such as a Time-of-Flight sensor, ARCore can combine its measurements with the camera-based depth estimate [45]. Table 3.2 provides a direct comparison of these two approaches to depth sensing.

Criterion	iPhone	Android
Primary depth source	Integrated LiDAR scanner	Depth estimation from camera images and device movement
Additional hardware	Defined hardware sensor on supported iPhone Pro models	No dedicated depth sensor required; some devices additionally provide a ToF sensor
Dependence on movement	LiDAR measurements are not primarily dependent on movement of the device	Depth estimation generally improves as the device is moved through the environment

Criterion	iPhone	Android
Environmental representation	ARKit can generate a polygonal mesh of the physical environment	ARCore provides depth images describing the estimated distance of visible surfaces
Device consistency	Known sensor configuration on the available test device	Available depth functions and hardware support vary between devices

Table 3.2: Depth sensing on LiDAR-equipped iPhones [44] and Android devices[45].

The movement required by camera-based depth estimation does not conflict with EdukARt, since physical movement is already intended to be part of the game. ARCore provides an alternative approach to depth sensing, and a future version of EdukARt could support both iOS and Android. However, supporting the wider range of hardware configurations found across Android devices would increase the development and validation effort. Differences in cameras, motion sensors, processing capabilities, and optional depth sensors would require additional capability checks, adaptations, and testing on multiple devices. For the initial prototype, development could instead focus on the known configuration of the available iPhone 13 Pro.

Restricting the application to iOS nevertheless reduces its potential audience, as indicated by the platform distribution discussed above. For the initial development and testing context, this limitation was considered manageable: Apple devices were common in the immediate university and media-production environment, and the first tests were planned to take place on the available device within the development environment. In potential school-based applications, iPhones already owned by participants could also be shared for successive game sessions, so each participant would not necessarily need a compatible device.

Native iOS development was therefore selected for this prototype. The decision combined the personal interest in Apple's native development environment with the practical aim of demonstrating the game's technical feasibility within a manageable implementation and testing scope. Following this decision, ARKit and RealityKit were used to process the iPhone's spatial information and manage the virtual game content. Their respective responsibilities are introduced in the following section.

3.1.2.2 ARKit and RealityKit

ARKit and RealityKit provide different but complementary parts of Apple's MR development environment. ARKit processes camera and motion-sensor data to estimate the

position and orientation of the iPhone within its surroundings. It establishes the world coordinate system, manages spatial anchors, detects surfaces, and provides access to camera and depth information.[44].

RealityKit uses this spatial information to manage the virtual content of the application. Three-dimensional models are represented as entities that can be positioned relative to anchors in the physical environment. RealityKit additionally provides functionality for rendering, animation, collision detection, and physics-based interaction [46].

In simplified terms, ARKit determines where the device is and what exists in its physical surroundings, while RealityKit determines which virtual objects are displayed there and how they behave. Native development provides direct access to both frameworks without requiring the intermediate abstraction layer of a game engine.

Together, ARKit and RealityKit provide the tracking, environmental understanding, rendering, and interaction capabilities required by the mobile application. However, these technologies represent only the virtual side of the prototype. The concept also requires a physical robot whose movement can become part of the shared game environment.

3.2 Robot Eduard as the Physical Platform

For the first prototype of EdukARt, the mobile robot Eduard by EduArt Robotik GmbH[1] was used as the physical platform. EduArt originated from the robotics environment at Technische Hochschule Nürnberg Georg Simon Ohm and has close connections to the university's RoboCup team autonOhm[2]. The company develops robotic platforms that transfer concepts and components from industrial mobile robotics into compact systems for education, research and prototyping.

Eduard is designed as a flexible research and development platform. It incorporates products and components from different industrial manufacturers and is itself used by industry partners for prototyping, testing and demonstration purposes. This makes it suitable as a representative platform for developing and evaluating new hardware and software concepts. EdukARt follows the same idea by using Eduard as the physical platform for a new proof of concept.

[1] Website of EduArt Robotik GmbH
[2] autonOhm Team of Technische Hochschule Nürnberg Georg Simon Ohm

3.2.1 Company and Project Context

EduArt Robotik GmbH was founded in 2021 by Prof. Dr. Stefan May and Markus Fenn. The original objective was to make mobile robotics accessible for education while using concepts and components that are also relevant in industrial robotic systems. This background is still reflected in the company's work, which combines educational robotics with research, software development, robot prototyping and industrial automation.

Eduard was the first central robot platform developed around this idea. It can therefore be used for teaching mobile robotics as well as for testing sensors, actuators, algorithms or other hardware and software concepts[3]. This concept also makes Eduard suitable as a smaller representative platform for developments that would otherwise require considerably larger mobile robots.

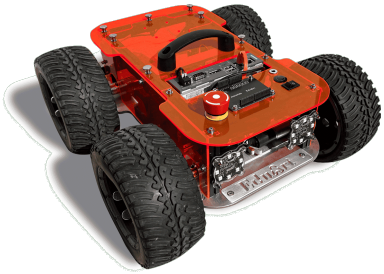


Fig. 3.1: Eduard in fluorescent red color with Off-Road wheels

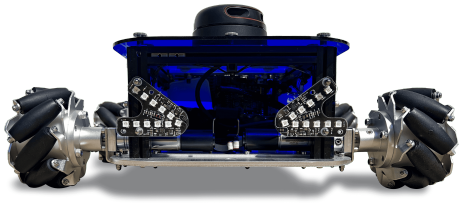


Fig. 3.2: Blue Eduard with mecanum wheels and LiDAR on top

The product portfolio has expanded beyond Eduard. The mobile platform Arthur[4] focuses on mobile manipulation and can combine an omnidirectional mobile base with a robotic arm such as the igus ReBeL. It is intended for applications such as autonomous transport, machine operation, object manipulation and the RoboCup Smart Manufacturing League (former @Work). A further product is the sensor ring Vision360[5], a modular arrangement of 3D Time-of-Flight sensors. Multiple sensor modules can be distributed around a robot and combined to provide up to 360° environment coverage. The system is intended for obstacle detection and collision avoidance and can be adapted to robots of different shapes and sizes.

[3] EduArt Robotik: Product page of Eduard
[4] EduArt Robotik: Product page of Arthur
[5] EduArt Robotik: Product page of Sensor Ring Vision360

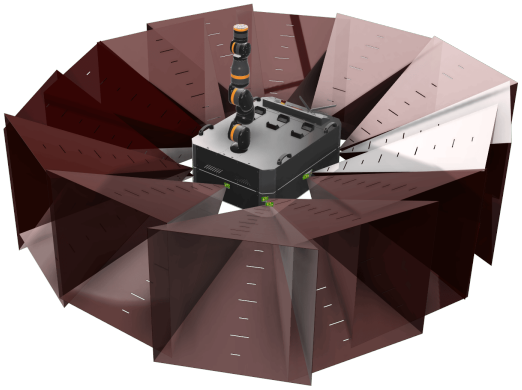


Fig. 3.3: Robot Arthur with red boxes indicating the fields of view of its surrounding sensor ring Vision360

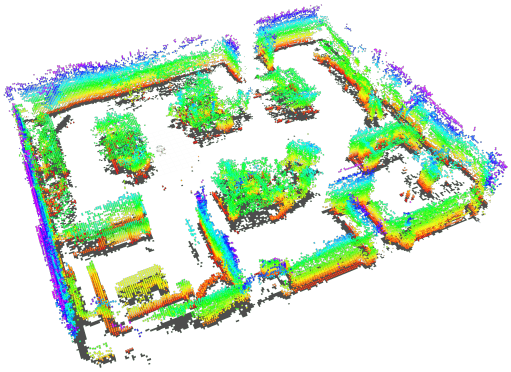


Fig. 3.4: Resulting Vision360 point cloud of a room

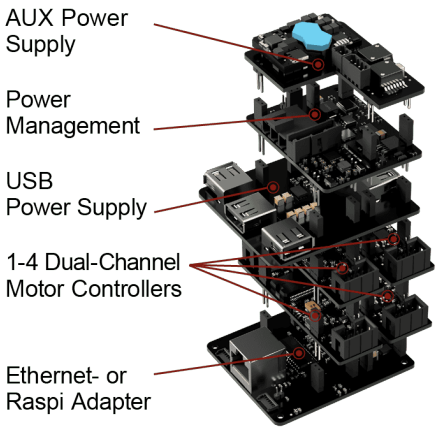


Fig. 3.5: Kinematics Kit Kim

Instead of providing a complete robot, the Kinematics Kit Kim follows another approach. It provides modular control, motor and power electronics from which users can build their own robot around a Raspberry Pi or an external computer. Depending on the configuration, up to eight DC motors can be controlled. The kit is used, for example, in RobotsUnite projects, where students design and build their own mobile robots[6].

In addition to these products, EduArt develops customized robotic systems and software, automation prototypes and autonomous drone applications. PCB design and prototype assembly are also carried out in-house, allowing electronics developed for individual projects to be transferred into later robot prototypes.

This combination of education and industrial development is also visible in collaborations with other companies. Different Eduard configurations integrate industrial components such as Siemens SIMATIC computing systems or drive components from FAULHABER. Eduard has also repeatedly been used to demonstrate technologies from industrial partners. At Hannover Messe, for example, Eduard was presented at the FAULHABER booth[7]. Other exhibition applications included a functional-safety demonstration with

[6] EduArt Robotik: Product page of Kinematics Kit Kim
[7] EduArt Robotik on LinkedIn: Eduard at the FAULHABER booth

Neuron Automation[8] and demonstrations with SICK at SPS[9]. The relationship with igus is similarly reflected in the hardware: Arthur can be equipped for example with an igus ReBeL robot arm, while Eduard has also been demonstrated with ReBeL systems at trade fairs [10].

Eduard itself regularly attracts considerable attention at exhibitions. Both children and adults are interested in seeing the robot move and often want to control it themselves. This makes the platform particularly suitable for applications in which visitors should not only observe a robotic system but directly interact with it.

Eduard therefore provides a suitable foundation for EdukARt. Its mobility, accessible controls, and existing appeal to exhibition visitors offer a starting point for turning robot interaction into a mixed-reality game. Instead of using Eduard only for free driving, the robot becomes part of a game with a defined objective. EdukARt is therefore intended to make the platform's existing exhibition use more engaging and provide a reusable experience in which visitors explore the robot's capabilities through direct interaction.

To support reuse at different events, the setup should not depend on one fixed exhibition environment. Eduard may be presented at EduArt's own booth, integrated into a partner's exhibition space, or used in considerably smaller demonstration areas. The game concept should therefore support a modular setup that can be assembled quickly and adapted to the available space without requiring a permanently installed tracking or projection infrastructure.

3.2.2 Technical Capabilities of Eduard

Table 3.3 summarizes Eduard's technical specifications and available configurations, based on EduArt's datasheet[11].

Category	Specification
Dimensions ($W \times H \times D$)	$380 \times 210 \times 420$ mm
Weight	7.6 kg
Kinematics *omnidirectional Mecanum steering*	Skid steering with Off-Road wheels;
Maximum speed	Off-Road: 0.56 m/s; Mecanum: 0.32 m/s
Payload	Off-Road: 5 kg; Mecanum: 10 kg

[8] Neuron Automation on LinkedIn: functional-safety demonstration with Eduard
[9] EduArt Robotik on LinkedIn: demonstration with SICK at SPS
[10] EduArt Robotik on LinkedIn: demonstration with igus ReBeL
[11] EduArt Robotik: Eduard datasheet.

Category	Specification
Battery runtime	Up to 4 h of driving time
Charging time	Approximately 3 h
Battery	4.5 Ah nickel-metal hydride battery (NiMH)
Available voltage outputs	19 V DC; 12 V DC; 5 V DC; // battery voltage of approximately 17.5–30 V
Integrated sensors	Four Time-of-Flight distance sensors; six-degree-of-freedom IMU; voltage and temperature monitoring; high-precision incremental encoders
Connectivity	WiFi; Ethernet; USB 3.0 and 2.0; two CAN interfaces; Bluetooth 4.0; Arduino-Uno-compatible GPIOs including I/O, I^2C, SPI, and UART
Bluetooth controllers	PlayStation 5 controller; optionally Steam Deck
Onboard computer	IOT: Siemens SIMATIC IOT2050 Advanced; Debian 10 Buster IPC: Siemens IPC127E, IPC227E, or IPC227G; Debian 10 Buster or Edge Runtime Raspi: Raspberry Pi 5; current Raspberry Pi OS
Programming	ROS 1 or ROS2 nodes using Python or C++; Node-RED

Table 3.3: Technical specifications and configurations of Eduard

In addition to its physical capabilities, Eduard's existing software provides a basis for integration into EdukARt. EduArt defines a common ROS2 interface across its robot platforms, through which external software can control movement, activate or deactivate the robot, and change its lighting[12]. These functions allow the game application to influence Eduard's physical behavior and provide visible feedback, while the robot's existing software continues to handle drive control. EdukARt can therefore build on an already operational robot without requiring modifications to its drive hardware.

For the initial prototype, Eduard is used in a basic configuration with Mecanum wheels. These enable lateral movement in addition to forward and backward motion, allowing the robot to maneuver along freely arranged indoor tracks.

In this setup, Eduard provides the physical movement and feedback, while the iPhone captures the spatial game environment and displays the virtual content. Connecting these components requires communication with the robot, determination of its position and orientation, and coordination between physical movement and virtual game events. The following Chapter, Design and Implementation of EdukARt, builds on these requirements to develop the game concept and explain its implementation.

[12] EduArt Robotik: ROS2 interface documentation

Chapter 4

Design and Implementation of EdukARt

This Chapter presents the game concept and derives the functional, software, and hardware requirements of the prototype. It then describes the development environment and the implementation of EdukARt. The implementation covers robot communication and control, spatial localization, track creation, and gameplay.

4.1 Game Concept and Scope of the Prototype



Fig. 4.1: EdukARt Keyvisual

EdukARt is conceived as a mixed-reality racing game inspired by traditional racing games such as Mario Kart and location-based games such as Pokémon Go. It combines virtual game elements with the movement of a physical robot in a real environment. In the prototype, Eduard serves as the physical game element, while other or self-built robots could be supported in the future. The player's objective is to steer the robot along a freely created race track, avoid obstacles, collect items, and achieve the fastest possible time. The player follows the robot through the physical environment to keep it within the iPhone's camera view. This movement supports the interaction with the robot but is not itself an active element of the gameplay. A video demonstration of the prototype and its gameplay is available on YouTube[a].

[a] EdukARt: Prototype and Gameplay Demonstration

The current prototype implements selected aspects of this game concept, with a focus on connecting physical robot behavior with virtual game events. It primarily serves as a collection of functions and interaction mechanisms rather than a fully developed racing game. The implemented items, obstacles, and timing effects demonstrate possible forms of interaction, while their combination into a coherent game experience still requires further

development and evaluation. In particular, balanced effects, meaningful challenges, and a consistent scoring system remain open aspects of the game design.

4.2 Development Process and Technical Environment

This section introduces the development environment, the hardware and software used, and the technical configuration of the robot. It also describes the transition from the exploratory first prototype to the codebase used for the final prototype.

4.2.1 Development Environment and Software Versions

Developing EdukARt involved implementing and testing the mobile application, integrating it with the physical robot, creating the visual game elements, and documenting the development process. Development and testing were carried out within Apple's native development environment using the hardware and software summarized in Table 4.1. Eduard's own computing and software environment is described separately in the following section.

Area of Use	**Tool or Device**	**Version or Specification**
Development system	MacBook Pro with Apple M4 Max	macOS Tahoe 26.4.1, 64 GB RAM
Mobile test device	iPhone 13 Pro	iOS 26.6.1
Application development	Xcode	Version 26.6
AI-assisted development	ChatGPT Pro, GPT-5.6 Sol	Reasoning effort: medium
Images	Adobe Photoshop	Version 27.8.0
Vector graphics	Adobe Illustrator	Version 30.6
AR preparation	Reality Composer Pro	Version 2.0 (494.100.6)
3D modeling	Blender	Version 5.0.1
CAD modeling	Autodesk Fusion 360	Version 2705.1.15, ARM64 (native)
Notes for documentation	Obsidian	Version 1.13.7
LaTeX documentation	Visual Studio Code	Version 1.139.1

Table 4.1: Development, documentation, and design tools used for the prototype

Adobe Photoshop and Illustrator supported the creation of graphics, icons, and figures. Autodesk Fusion 360, Blender, and Reality Composer Pro were used to create, adapt, and prepare 3D assets for the application, as described in Virtual Game Elements: Creation and Interaction.

Development notes recorded in Obsidian supported writing the thesis in Visual Studio Code using LaTeX. DeepL assisted with translation, while DeepL Write and ChatGPT supported linguistic revision of existing drafts. All suggestions were reviewed and adapted, with responsibility for the final wording and accuracy remaining with the author.

4.2.2 Technical Environment of Eduard

The prototype was developed using Eduard_Blue3, equipped with a Siemens SIMATIC IOT2000 running Debian, while its ROS2-based control software operates within an Ubuntu Docker container. The installed software, running containers, and interfaces were examined to determine the robot's configuration and behavior.

The software information was retrieved directly from Eduard using the terminal command in Listing 4.1. Table 4.2 summarizes the relevant hardware, software, and communication parameters of this robot, whose configuration may differ from other Eduard installations or the publicly available GitHub documentation.

Listing 4.1: Inspection of Eduard's software environment

```
ssh root@192.168.0.100 'bash -s' <<'CHECK'
cat /etc/os-release
uname -r
docker version
docker compose version
docker ps -a --format 'table {{.Names}}\t{{.Image}}'
docker exec eduard-robot-control-iotbot-0.5.0 \
    bash -c 'cat /etc/os-release
python3 --version
ls /opt/ros
pwd'
CHECK
```

Component	Specification
Onboard computer	Siemens SIMATIC IOT2000
Host operating system	Debian GNU/Linux 11 (Bullseye)
Linux kernel	Version 5.10.64

Component	Specification
Containerization	Docker 27.2.1 and Docker Compose 2.29.2
Robot-control Docker image	`eduartrobotik/eduart-robot-control:0.5.0`
Robot-control container OS	Ubuntu 24.04.1 LTS
Robotics framework	ROS2 Jazzy
Python version	Python 3.12.3
ROS workspace	`/home/user/ros`
Robot ROS namespace	`/eduard/blue3`
Robot WiFi network	`Eduard_Blue3`
Robot IP address	`192.168.0.100`
UDP port	`50505`

Table 4.2: Technical environment of the EduArt robot used for the prototype

4.2.3 AI-Assisted Development Process

Following the initial technology evaluation, a first prototype was developed with AI assistance in Xcode to explore the suitability of native iOS development. Returning to a game engine for a cross-platform implementation remained an option if the native approach proved unsuitable. Codex supported experiments with ARKit and RealityKit, initially using a joystick-controlled virtual robot. The resulting functionality soon provided a practical basis for the game, supporting the decision to continue with native iOS development. However, as the physical Eduard and further features were integrated, insufficient control over generated changes led to duplicated functionality, unintended effects on existing components, and an increasingly difficult-to-maintain codebase. Once the physical robot successfully reacted to a virtual game element, development restarted with a more structured implementation. The exploratory prototype is not documented separately – its relevant concepts and functions are addressed in the implementation described below.

The second prototype started from a new codebase managed with Git. Useful functions from the first prototype were reviewed and reintroduced step by step, with clear responsibilities for user-interface code, robot control, communication, and spatial tracking. Shared control logic was separated from functions specific to the physical or virtual robot. Selected virtual-robot functions were retained for testing and comparison, while development focused on the physical robot. ChatGPT supported the discussion of implementation approaches and generated Swift code proposals, which were manually reviewed, integrated, and tested. Particular attention was paid to avoiding duplicated functionality and unnecessary complexity. Git commits preserved functional intermediate states and

made changes traceable through comparisons between versions. Xcode's integrated AI tools continued to support user-interface development, followed by testing and refinement on the target iPhone.

4.3 Implementation and System Overview

The difficulties encountered in the first prototype made the separation of components a central consideration for EdukARt. Integrating physical robot control into the existing AR functionality had introduced duplicated code and changes that affected other parts of the application. The second prototype therefore needed a structure in which both robot representations could share common functions, while new features could be added without repeatedly changing unrelated components.

The software structure of EdukARt therefore separates user interaction, robot control, communication, spatial tracking, and gameplay into components with distinct responsibilities. Shared functionality is reused across the application, while robot-specific behavior is handled separately. This organization aims to keep changes manageable and reduce duplicated code, particularly where the physical and virtual robots use the same control functions. The separation between the AR Eduard and the physical Eduard also provides a basis for integrating other virtual or physical robots through robot-specific implementations without restructuring the entire application. An overview of the prototype's functionality is shown in Fig. 4.2.

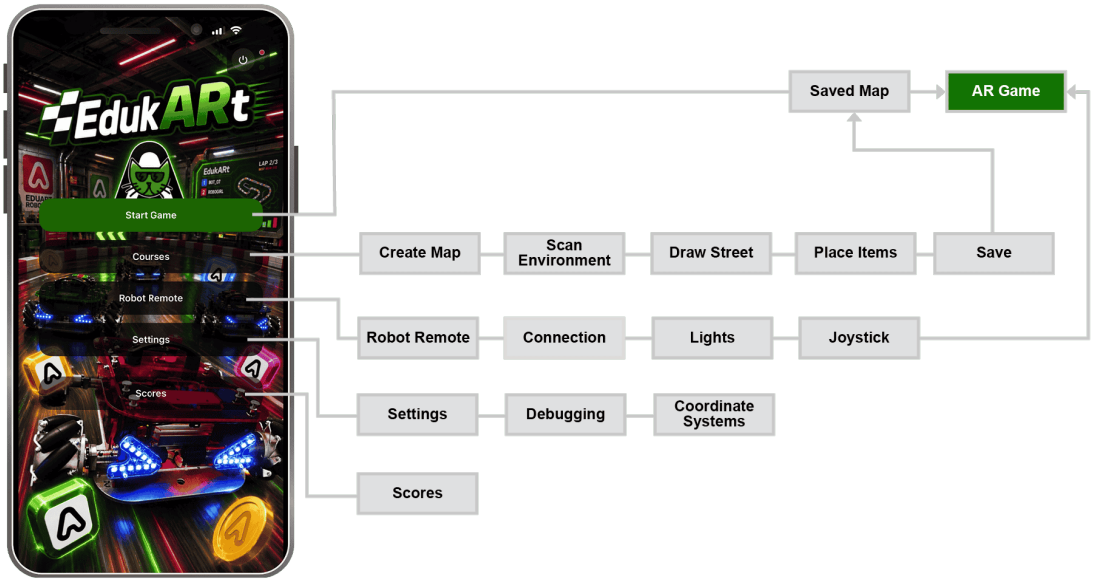


Fig. 4.2: System architecture of the EdukARt prototype

The following sections describe its implementation in the order of the intended workflow for first-time use. `Robot Remote` (Section 4.3.1) provides the functions required to connect and control the physical robot, including joystick-based movement and lighting control. `Courses` (Section 4.3.2) allows users to create, edit, and delete race tracks containing items and obstacles (4.3.3). From `Start Game` (Section 4.3.4), users can either open `Courses` or load an existing course and begin a race. Results can be saved and compared through `Scores`, while `Settings` provides supporting and debugging functions. The menu itself places `Start Game` first to provide quick access to existing courses. Alternative approaches and the reasons for discarding them are discussed where relevant. The accompanying tables summarize the responsibilities of the main software components.

4.3.1 Robot Remote: Connecting and Controlling Eduard

The `Robot Remote` menu provides access to Eduard's movement and lighting controls independently of a race. Its interface is implemented in `RobotRemoteView.swift`, which combines connection and activation controls, joystick input, and lighting controls as described in the following sections.

4.3.1.1 Communication: Establishing a Connection to the Physical Robot

To control Eduard, the iPhone must be connected to the robot and the motors must be enabled through the application. These two steps are distinguished in the `Settings` panel within `Robot Remote`, which provides separate controls for checking the connection and enabling or disabling the motors.

To establish this connection, both Bluetooth and Wi-Fi were considered as communication channels between the iPhone and Eduard. Bluetooth was initially preferred because Eduard's Wi-Fi network does not provide Internet access, causing iOS to display warnings and suggest switching to mobile data during testing. During development, Internet access was also needed to verify the developer certificate after deployment from Xcode. This required repeatedly disconnecting from Eduard's network, deploying and verifying the application, and reconnecting for testing. Since Eduard already supports a Bluetooth-connected PS4 controller, the initial idea was to reproduce this control approach within the application. However, research into custom Bluetooth communication on iOS indicated greater implementation effort than a network-based solution.

Wi-Fi was therefore selected, using Eduard's existing network, which the iPhone can join through the iOS Wi-Fi settings. EduArt's browser-based Virtual Joy interface already demonstrates robot control over this network.

The three states shown in Figs. 4.3–4.5 illustrate the connection and activation sequence. A red `WiFi` control indicates that Eduard is not connected, while the yellow `Enable` control indicates that the motors have not been enabled through the application. Once Eduard is connected, the `WiFi` control turns green. Enabling the motors also turns the activation control green and changes its label to `Disable`, allowing the user to disable the motors again. The control overlay in the upper-right corner of most screens, implemented in *RobotControlOverlay.swift*, displays the same status information and allows users to enable or disable the motors when the iPhone is connected to Eduard's Wi-Fi network.

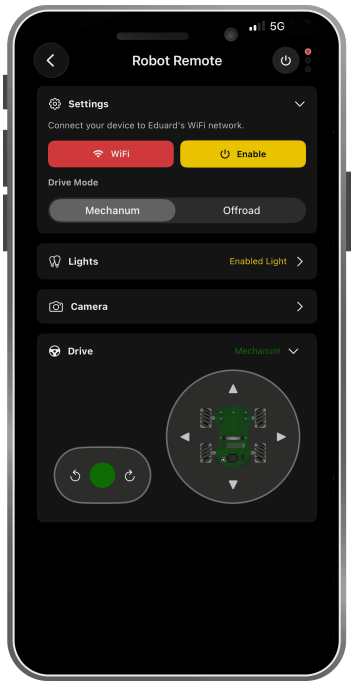


Fig. 4.3: Not Connected

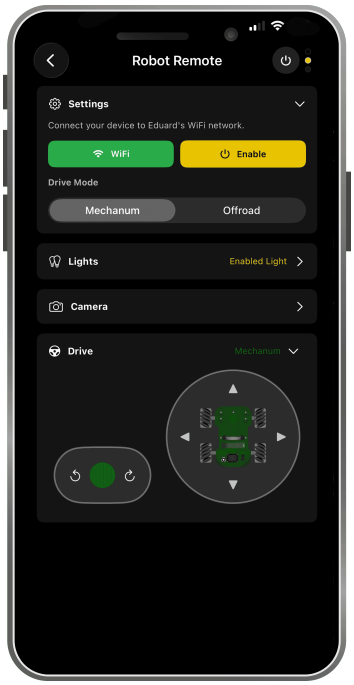


Fig. 4.4: Not Enabled

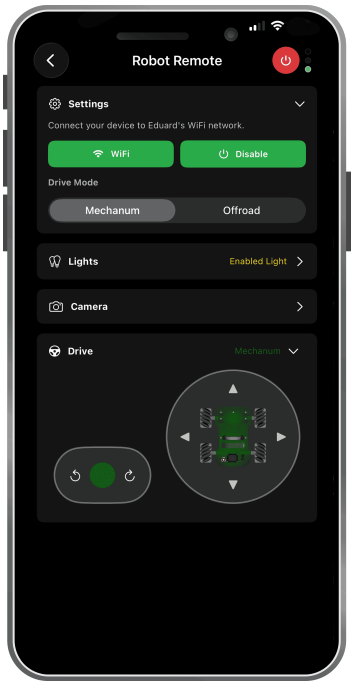


Fig. 4.5: Enabled + Connected

As described in Background and Related Work, ROS is widely used in robotics. The communication approach builds on Eduard's existing ROS2 interfaces introduced in Technical Foundations and Technology Selection. The task was therefore to connect the native iOS application to these interfaces while keeping the existing robot-control software unchanged. ROS2 remained on the robot, avoiding an additional installation on the development computer, while game-specific behavior was implemented within the application.

Given the previously encountered compatibility and configuration difficulties with ROS on macOS described in Native iOS Development, an initial approach was to connect the application to Eduard through a Python receiver running on the robot, avoiding a native ROS2 installation on the MacBook. Initially, it translated fixed application commands, such as driving or enabling the motors, into predefined ROS2 messages and service

calls. However, each additional function required changes to the script and renewed deployment on the robot. It was therefore redesigned as a generic UDP-to-ROS2 bridge: the application specifies the topic or service, its message type, and the required data, while the receiver forwards the corresponding message or request to ROS2. Exposing these parameters through the application's settings could later allow users to configure other ROS2-based robots without modifying the bridge.

On the application side, this communication is implemented in *Eduard.swift*. It translates requests such as activating the robot, driving, or changing its lighting into the ROS2 operations used by Eduard. These operations are transmitted as JSON messages over UDP to `192.168.0.100` on port `50505`.

Installing and testing the receiver required access to Eduard's onboard computer. After connecting the development computer to the robot's Wi-Fi network, an SSH connection can be established using the following command.

```
ssh root@192.168.0.100
```

On the Eduard used for this prototype, ROS2 runs inside a Docker container. This is important when accessing the robot through SSH: ROS2 commands are not available in the host shell and must instead be executed within the container. The following command opens an interactive shell in one of Eduard's existing ROS2 containers:

```
docker exec -it eduard-iotbot-1.0.2 bash
```

The receiver also required access to the container's ROS2 environment. Initially, the Python script was stored on the robot's host system and copied into the existing robot-control container, where it was started manually. This manual procedure was cumbersome and had to be repeated after robot restarts, as the copied script was no longer available inside the container.

An initial workaround used a shell script and a systemd service to copy the receiver into the robot-control container and start it automatically. However, since Eduard already had other containers that started automatically, including Node-RED, their startup configuration was examined to apply the same approach to the receiver.

Based on these findings, the receiver was integrated into a separate Docker Compose container with access to the Python script stored permanently on the robot. Its startup command loaded the required ROS2 environment and launched the receiver, while `restart: always` ensured automatic startup when Docker restarted.

The configuration and receiver script were stored under `/home/user/edukart_ros2_bridge/docker/iot2050`. The file `docker-compose.yaml`

defined the bridge service as `edukart-ros2-receiver`. After this initial configuration, the container was created and started using the following commands:

```
cd /home/user/edukart_ros2_bridge/docker/iot2050
docker compose up -d
```

The bridge supports both publishing ROS2 messages and calling ROS2 services. In each case, the application specifies the interface name, its type, and the required data. Robot activation provides an example of a service call: selecting `Enable` causes the application to send a request to Eduard's `set_mode` service of type `edu_robot/srv/SetMode`. The JSON command in Listing 4.2 activates remote control with Mecanum kinematics.

Listing 4.2: JSON command for activating remote control with Mecanum kinematics

```
{
  "kind": "service",
  "service": "set_mode",
  "serviceType": "edu_robot/srv/SetMode",
  "request": {
    "mode": {
      "mode": 2.0,
      "drive_kinematic": 2.0,
      "feature_mode": 0.0
    },
    "disable_feature": 0.0
  }
}
```

The receiver creates the specified service request, fills its fields with the supplied values, and calls the service within the configured robot namespace, `/eduard/blue3`. Here, `mode = 2.0` selects remote control and `drive_kinematic = 2.0` selects Mecanum kinematics. Selecting `Disable` uses the same request structure with `mode = 0.0` and `disable_feature = 1.0`. Movement commands use topic messages instead and are explained in the following section on robot movement.

The application accesses these robot-specific communication functions through *RobotController.swift*, which coordinates robot control across the application. It forwards control requests to *Eduard.swift* and maintains the connection and activation state displayed in the interface. To monitor the connection independently of command transmission, it also requests reachability checks through *Eduard.swift*, which uses the Pingx library. This helped distinguish interruptions of the network connection from problems with the robot's control functions.

Command transmission through the UDP bridge is unidirectional, which means that no status updates, service results, or error messages are sent back to the application. For the prototype, however, feedback about the robot's behavior was provided visually through Eduard's integrated lights during testing and operation instead of implementing an additional return channel. More detailed feedback within the application remains a possible extension for future versions. Table 4.3 summarizes the responsibilities of the components involved.

Component	Responsibility
RobotController.swift	Coordinates robot control and maintains the connection and activation state used throughout the application.
Eduard.swift	Provides the interface to the physical Eduard and translates application requests into commands understood by the robot.
edukart_ros2_receiver.py	Runs on the robot and passes commands received from the application to the robot's ROS2 system.

Table 4.3: Software components for robot communication and connection control

4.3.1.2 Drive Modes: Controlling Eduard's Movement

The `Drive` panel within `Robot Remote` provides the joystick controls. During development, different control types, including joysticks and buttons, were explored in various arrangements and sizes. Informal testing and feedback from testers led to the joystick layout used in the current prototype. This layout assumes right-handed use. An option to exchange the positions of the controls could be evaluated in future versions to adapt the layout for left-handed users.

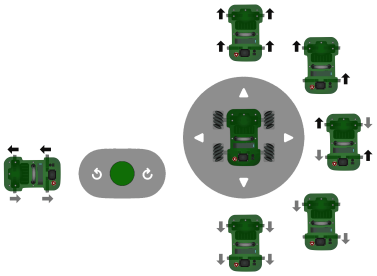


Fig. 4.6: Joystick layout and wheel directions for Mecanum movement

The resulting layout uses a larger joystick on the right for forward, backward, lateral, and diagonal movement, and a smaller joystick on the left for rotation. The interface is visible in Figs. 4.3–4.5. Figure 4.6 shows the corresponding wheel directions for Eduard's Mecanum kinematics. The surrounding robot illustrations use arrows in place of the wheels to indicate their direction of rotation for each movement. For forward movement, shown above the larger joystick, all four arrows point forward. Lateral and diagonal movement require different combinations of wheel directions, while the illustration beside the smaller control shows the opposing wheel directions used to rotate the robot.

The joystick controls are implemented in *JoystickView.swift*, using the SwiftUIJoystick library [47] as a basis and adapting its input handling to EdukARt's movement and rotation controls. Finger movement determines the joystick input within a limited range, and releasing the control returns it to its neutral position with zero input. The same view is reused during gameplay, where active item and obstacle effects can modify how joystick input affects the robot's movement.

RobotController.swift uses separate control modes to determine whether joystick input controls the physical Eduard, its virtual representation, or both. If the physical robot is not connected or enabled, movement commands are routed to *EduardSimulation.swift*. Otherwise, the configured mode determines how commands are handled: the default real mode forwards them through *Eduard.swift* exclusively to the physical robot, while simulation mode controls only the virtual representation.

To allow the robots to be controlled and tested independently or together without disconnecting Eduard or interrupting its visual detection, the implementation also includes synchronization options. Synchronized mode sends movement commands to the physical robot and animates the virtual model's wheels. Live Sync can additionally align the virtual model with the localized position and orientation of the physical robot.

To send the joystick commands to the physical robot, *Eduard.swift* maps the shared drive command to a `geometry_msgs/msg/Twist` message. Forward and lateral movement are represented by `linear.x` and `linear.y`, while rotation uses `angular.z`. The JSON message in Listing 4.3 instructs the bridge to publish a command for forward movement to `/eduard/blue3/cmd_vel`.

Listing 4.3: JSON command for forward movement

```
{
  "kind": "publish",
  "topic": "cmd_vel",
  "messageType": "geometry_msgs/msg/Twist",
  "message": {
    "linear": {
      "x": 0.1,
      "y": 0.0,
      "z": 0.0
    },
    "angular": {
      "x": 0.0,
      "y": 0.0,
      "z": 0.0
    }
   }
}
```

During development, the physical robot was observed to become disabled when command transmission stopped. Its documentation describes a timeout of 200 ms without a new command and demonstrates repeated transmission at 10 Hz, including zero-velocity commands when the robot should remain stationary.[1]

To maintain continuous transmission, *RobotController.swift* starts a command loop after enabling the robot with an interval of 0.05 seconds keeping the robot stationary and enabled. Input changes are also processed immediately. Table 4.4 summarizes the components used for this shared control structure.

Component	Responsibility
JoystickView.swift	Provides the movement and rotation controls, drag handling, and return to the neutral position.
JoystickMonitor (SwiftUIJoystick)	Stores joystick positions and makes input changes available to the application.
RobotTypes.swift	Defines shared drive commands, control and drive modes, and robot poses.
RobotController.swift	Interprets joystick input and directs the resulting movement commands to the physical robot, the virtual robot, or both.

[1] EduArt Robotik: edu_simulation documentation, After the launch.

Component	Responsibility
Eduard.swift	Sends movement commands to the physical Eduard through its ROS2 interface.
EduardSimulation.swift	Applies movement commands to the virtual Eduard and animates its wheels.

Table 4.4: Software components for controlling the physical and virtual robot

4.3.1.3 Lights: Configuring the Robot's Visual Feedback

The `Lights` panel within `Robot Remote` allows Eduard's lighting functions to be tested. Eduard has four integrated lighting modules, with two positioned at the front and two at the rear. These already indicate robot states during normal operation. EdukARt extends their use to provide physical feedback about interactions with virtual game elements. During development, visible changes to the lights also helped verify that commands were received and applied by the robot.

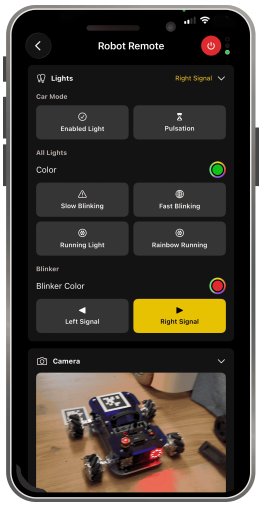


Fig. 4.7: Light control

Figure 4.7 shows the lighting controls in `Robot Remote`, implemented in *RobotRemoteView.swift*. `Car Mode` sets white lights at the front and red lights at the rear, resembling a car's lighting. `All Lights` provides blinking and running-light patterns for all four modules using a selected color. Its `Rainbow Running` option creates automatic color changes by repeatedly sending new RGB values to Eduard's existing running-light. `Blinker` activates the left or right lighting group using a separately selected signal color. Table 4.5 summarizes the mapping between the application controls and Eduard's ROS2 lighting modes.

The expandable `Camera` panel, also shown in Fig. 4.7, displays the iPhone's live camera image through *CameraPreviewView.swift*. This allows the controls and the robot's response to be captured together in a screen recording.

EdukARt UI	ROS2 mode	Target group	Lighting behavior
Enabled Light	DIM (1)	`all`	Predefined white front lights and red rear lights.
Pulsation	PULSATION (3)	`all`	Predefined pulsating lighting pattern.
Slow Blinking	FLASH (2)	`all`	Slow blinking using the selected all-lights color.

EdukARt UI	ROS2 mode	Target group	Lighting behavior
Fast Blinking	ROTATION (4)	`all`	Fast blinking using the selected all-lights color.
Running Light	RUNNING (5)	`all`	Running-light pattern using the selected all-lights color.
Rainbow Running	RUNNING (5)	`all`	Running-light pattern with RGB values continuously changed by the application.
Left / Right	FLASH (2)	`left_side` `right_side`	Blinking on the selected side using the separately selected signal color.

Table 4.5: Mapping of EdukARt's lighting controls to Eduard

The ROS2 mode `ROTATION (4)` is labelled `Fast Blinking` in EdukARt to describe the lighting behavior observed on the tested robot. The selected lighting mode is passed to *RobotController.swift*, while *Eduard.swift* translates it into the corresponding robot-specific lighting command. The command is transmitted through the UDP-to-ROS2 bridge described previously and published to `/eduard/blue3/set_lighting_color` using the message type `edu_robot/msg/SetLightingColor`. Each message specifies RGB values, a lighting mode, and the target group. The groups `all`, `left_side`, and `right_side` address either all four modules or the two modules on one side of the robot.

The same lighting functions are reused during gameplay. RobotController determines when an interaction should trigger visual feedback and requests the corresponding color and pattern from Eduard. Temporary effects can restore the previous lighting state after they end. This separates the purpose and timing of a game effect from the robot-specific command required to display it and avoids implementing a separate communication path for each item or obstacle. The individual lighting effects are described together with the corresponding game elements. Table 4.6 summarizes the responsibilities of the main lighting-control components.

Component	Responsibility
RobotController.swift	Forwards lighting-mode selections and coordinates visual feedback for game events, including temporary effects and restoration of previous lighting states.
Eduard.swift	Stores selected colors, maps lighting requests to Eduard and generates the rainbow effect.

Table 4.6: Software components responsible for lighting control and visual game feedback

4.3.2 Courses: Creating a Map Using a Spatial Reference System

Building on the robot-control functions described above, a spatial game environment had to be created in which Eduard could interact with virtual tracks, items, and obstacles. These elements needed to be positioned within the physical surroundings, and their arrangement had to be saved and reused across multiple game sessions. The interface for creating and managing these courses is implemented in *CoursesView.swift*.

As discussed in Native iOS Development, the LiDAR scanner integrated into the available iPhone 13 Pro was an initial technical argument for focusing on Apple devices. Development therefore explored ARKit's capabilities for capturing the environment using this sensor. The initial approach was to derive a map from the detected physical surfaces and use it as a basis for placing virtual game elements.

The following sections explain the practical limitations encountered and the approaches explored to address them, leading to the development of a dedicated coordinate system as the spatial foundation of the game. From Section 4.3.2.4 onward, the description follows the application's menu areas, explaining how users record a map, draw a race track and place items and obstacles.

4.3.2.1 LiDAR-Based Mapping and ARKit's Coordinate System

To create a reusable game environment from the physical surroundings, it was necessary to investigate how the iPhone's LiDAR scanner could capture the room. ARKit's scene reconstruction provided a starting point by generating a polygonal mesh of detected surfaces and classifying parts of the geometry as floors, walls, or other objects [44].

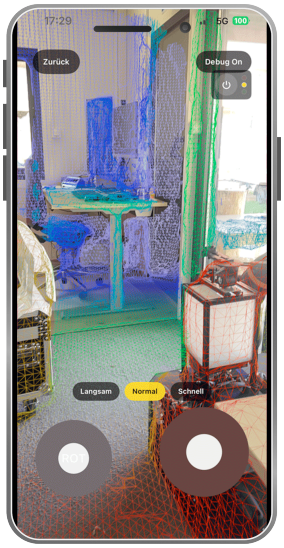


Figure 4.8 shows the mesh visualization used during these experiments. Its colors represent distance from the iPhone: nearby surfaces appear red, intermediate areas green, and more distant areas blue. Furniture and other potential obstacles were clearly represented, although inconsistencies were observed around windows and more distant surfaces. This suggested that the reconstruction could serve directly as a map of the room. The initial idea was to save the mesh, reopen it later, and place virtual tracks, items, and obstacles within the reconstructed environment. Assuming that the physical arrangement remained unchanged, the same map could then be reused across multiple game sessions.

Fig. 4.8

However, the iPhone became increasingly hot during scanning, and rendering became less smooth as more of the environment was captured. Furthermore, the complete geometry of objects above the floor appeared unnecessary for course creation: the relevant information was where the robot could move, while areas occupied by physical obstacles should remain unavailable for track placement. A simpler approach was therefore explored in which only the detected floor surfaces would be stored. These surfaces were converted into a grid of square tiles, making it possible to visually examine how reliably the floor was recognized and how closely the resulting map matched the physical surroundings.

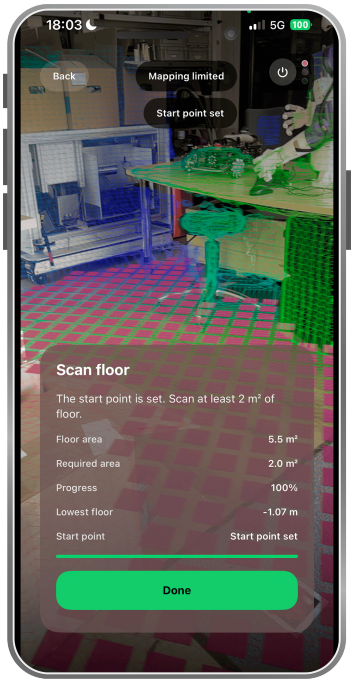


Fig. 4.9

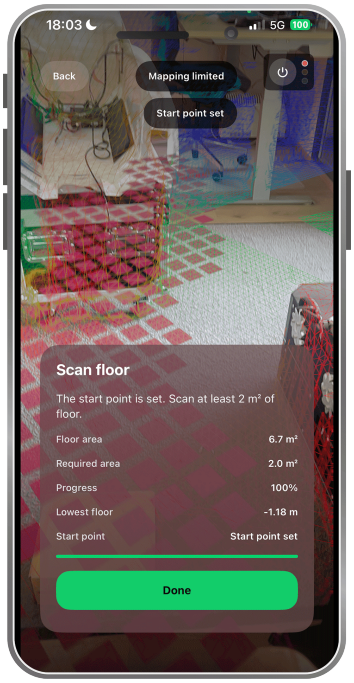


Fig. 4.10

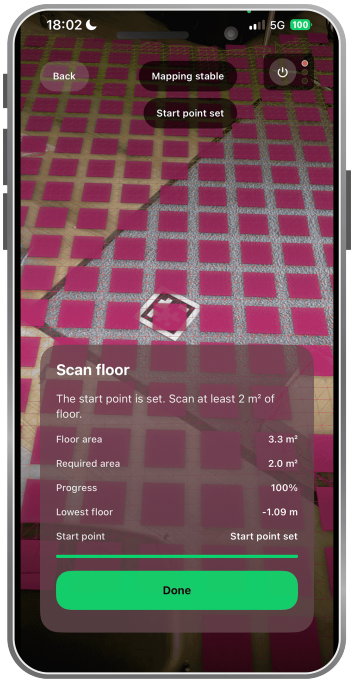


Fig. 4.11

Figure 4.9 shows the tiles closely aligned with the physical floor, including the area beneath a desk. Although occasional shifts occurred as the scan was updated, as illustrated in Fig. 4.10, the results suggested that the detected floor could provide a useful basis for course creation. The next step was therefore to save the tile arrangement and reopen it for editing and reuse.

Saving and loading preserved the arrangement of the tiles, but did not automatically restore their original position within the room. Attempts to restore their placement in a new ARKit session resulted in the map appearing rotated, displaced, or above or below the physical floor. Adjusting the map's vertical position using the detected floor height, allowed it to be aligned with it.

This resolved the vertical offset and placed the tiles on the floor. However, the tests revealed that the map's horizontal position and its rotation depended on where and how the new session was started. The initial solution therefore relied on users visually checking

whether the loaded tiles matched the physical surroundings and repeating the placement when necessary. This proved imprecise because the iPhone could not be positioned and oriented consistently enough to align the entire map. Even when nearby tiles appeared correctly placed, small angular errors caused increasingly visible displacement at greater distances.

Further testing showed that the spatial reference depended on the ARKit session: its origin was established at the iPhone's position when the session started, while its horizontal orientation depended on the device's initial orientation [48]. Figure 4.12 visualizes this reference using an axis marker at the session origin. The iPhone was moved slightly from its initial position to make the axes visible in the camera image.

A printed AprilTag, visible in Fig. 4.11, was introduced to guide the iPhone's position and orientation when defining the map reference.



Fig. 4.12: ARKit World Coordinate System



Fig. 4.13: AprilTag Coordinate System

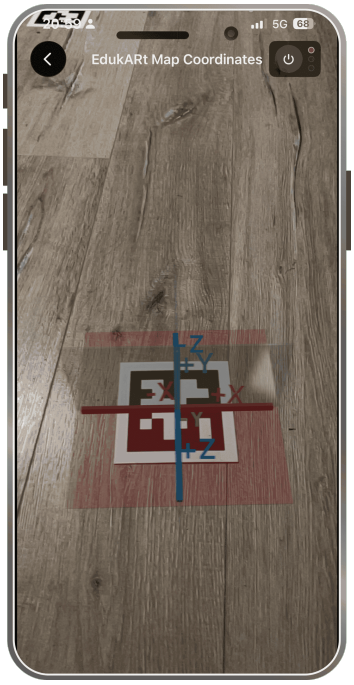


Fig. 4.14: EdukARt Map Coordinate System

4.3.2.2 AprilTags and their Coordinate system

AprilTags are visual markers consisting of a black-and-white coded pattern with a white outer border for high contrast, used in robotics to identify reference points and estimate their position and orientation. For EdukARt, the `tag36h11` family was selected because it was already familiar from its use within the RoboCup@Work team (now SML).

Existing paper markers were reused for the initial experiments, retaining the familiar outer dimension of 40 mm [49].

The marker was first used as a visual guide for defining the floor map's reference point. Its edges were aligned with the edges of the iPhone's camera view, and it was made to appear as large as possible while remaining fully visible. Once the alignment looked correct, the reference point was set for saving the map. Repeating this alignment when loading it was intended to restore its position and orientation within the room. A smaller on-screen frame was then introduced to guide the marker's placement more consistently. In this second approach, ARKit's image detection estimated the marker's position and orientation, and measurements from multiple frames were averaged to improve the reference. The floor tiles were stored relative to it and repositioned when the marker was detected again. Although this improved the alignment, small orientation differences still caused visible displacement at greater distances. Dedicated AprilTag detection was therefore investigated as an alternative to ARKit's image detection.

Initial tests with the AprilTag iPhone app [50] showed that these small markers were only reliably detected when the iPhone was brought within approximately 1 m. The markers were therefore enlarged to improve detection at greater distances and replaced with more durable 3D prints.

Exploring AprilTags for environmental mapping also introduced the idea of using a marker to locate Eduard within the same environment. Their potential use on the robot was therefore considered when choosing their size, with the available mounting area limiting their outer dimension to 13 cm. This was subsequently reduced to 12 cm, allowing four markers to be printed simultaneously on 3D printers with a print area of 25 cm by 25 cm or larger. The corresponding black detection boundary measured 9.6 cm.

The marker designs were scaled in Adobe Illustrator, exported as DXF files, and extruded to a thickness of 1 mm in Autodesk Fusion. They were then prepared in Bambu Studio and printed on a Bambu Lab P2S using matte PLA. This material was chosen to provide a low-gloss surface for camera detection and sufficient durability for indoor use. The base was printed in white, with the black pattern confined to the upper two layers to reduce material changes and filament waste.

With the enlarged, 3D-printed markers available, initial tests with the SwiftAprilTag library[51] examined detection at greater distances and from different viewing positions. Several markers were arranged in a line on the floor and observed while sitting and standing. In the first comparison, the furthest marker was not detected from the seated position, as shown in Fig. 4.15, but was detected while standing at an estimated camera-to-marker distance of approximately 3.2 m, as shown in Fig. 4.16.

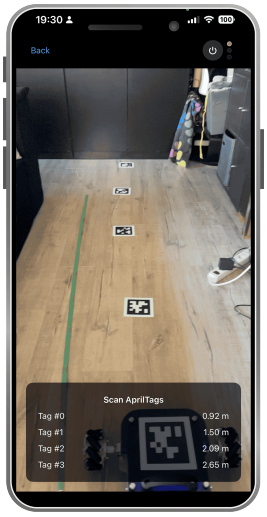


Fig. 4.15: Camera Distances While Sitting

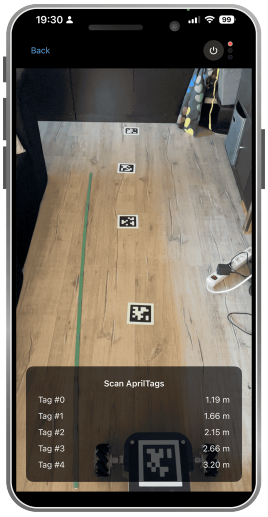


Fig. 4.16: Camera Distances While Standing

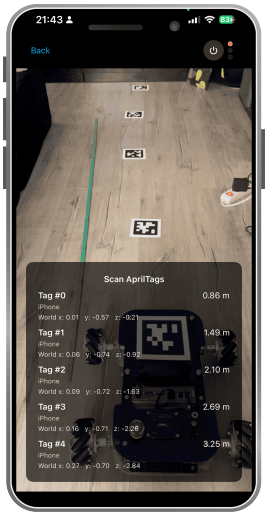


Fig. 4.17: World Coordinates While Sitting

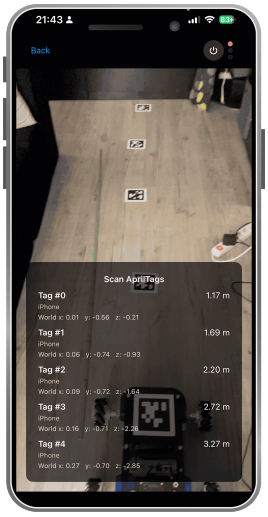
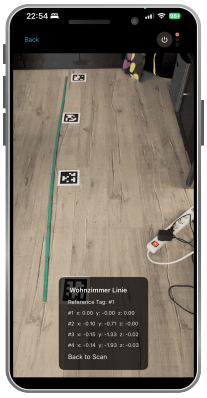


Fig. 4.18: World Coordinates While Standing

The displayed camera-to-marker distances and absolute ARKit world coordinates made it difficult to assess how accurately the detected positions reproduced this arrangement, because they complicated a direct comparison of the markers' relative positions with their physical arrangement. Figures 4.17 and 4.18 show the world-coordinate readouts, which were expressed relative to the session origin and the iPhone's initial orientation. The next approach was therefore to use the nearest floor marker as a common reference for measuring the positions of the other tags. To implement this approach, the AprilTag coordinate system had to be examined.

Comparing Fig. 4.13 with Fig. 4.12 shows differences in both axis orientation and the reference used to define each coordinate system. Whereas ARKit initially establishes its world coordinate system from the iPhone's position and orientation, the AprilTag coordinate system is defined relative to the physical marker. It therefore offered a recognizable reference for comparing marker positions across sessions, provided that the reference tag remained in the same physical position.

A subsequent test used tag 1, the nearest floor marker at the bottom of the image, as the reference origin. Figure 4.19 shows the linear arrangement compared with a measuring ruler. The estimated longitudinal distances showed approximate agreement with the ruler measurements, while the lateral offsets were consistent with the slightly staggered placement of the markers. However, the estimated height increasingly deviated from the reference plane for more distant markers, although all tags lay on the same physical floor.





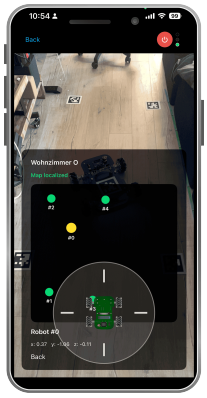


Fig. 4.19 *Fig. 4.20* *Fig. 4.21* *Fig. 4.22* *Fig. 4.23*

The relative marker positions were then plotted in a two-dimensional view to make their arrangement easier to assess. Figures 4.20 and 4.21 show tests with trapezoidal and L-shaped arrangements. The plots broadly reproduced both layouts and allowed a direct visual comparison with the physical markers. The robot's position was then added to the same map and displayed as a yellow point. Figure 4.22 shows Eduard viewed from the direction of tag 1 at the bottom of the map, while Fig. 4.23 shows it from the direction of tag 2 at the top. Despite the changed viewpoint, the yellow point followed the robot's movement in the correct map direction.

Although the estimated horizontal positions of the AprilTags broadly matched their physical arrangement, their estimated heights still did not consistently match the physical floor. The following section describes how the map coordinate system was adapted to address these deviations.

4.3.2.3 EdukARt's Coordinate System and 2D Minimap

The next approach combined the physical reference provided by an AprilTag with ARKit's gravity-aligned vertical direction. At the beginning of scanning, a detected floor marker was selected as the reference, defining the map's origin at $(0, 0)$ and its horizontal orientation. ARKit provided the upward direction. Together, they defined a horizontal map plane at the height of the reference marker in which marker positions, the robot, and virtual game elements could be represented independently of the iPhone's initial position. Figure 4.14 shows the resulting coordinate system.

The same map was then visualized in the AR environment by transforming its two-dimensional positions into ARKit's three-dimensional world coordinates. Figure 4.24 shows virtual cubes placed at the recorded marker positions to check their alignment with the physical AprilTags. The three images in Fig. 4.25 show the resulting grid, marker positions, and distances on the physical floor, making the spatial relationships

between the mapped markers visible. Map coordinates were expressed in metres, while the minimap was scaled proportionally to fit the iPhone's display.

AprilTag 0 mounted on Eduard was handled separately from the floor markers. It was excluded from selecting the map reference and recording the floor map because it moved with the robot and was positioned above the floor. Its detected position and orientation were instead transferred into the shared coordinate system to locate Eduard within the map.

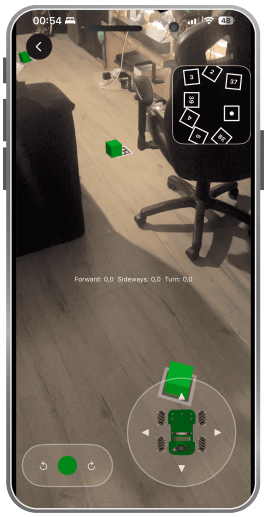


Fig. 4.24: Virtual cubes



Fig. 4.25: Spatial relationships between mapped AprilTags

During gameplay, the observed positions of known floor markers are compared with their stored map positions. Corrections are based on multiple measurements and applied gradually, helping maintain the map's horizontal alignment without noticeable jumps in the AR view.

Further integration of LiDAR-based environmental capture was postponed so that development could first focus on placing a track within the room and testing the interaction mechanisms. Consequently, the marker-based map did not automatically describe physical obstacles or determine which areas were suitable for driving. Incorporating detected floor surfaces and obstacles into this map remains a possible extension for future development.

With this spatial foundation established, the next Section returns to the menu-based description and explains how the application records a map for course creation.

4.3.2.4 Scanning AprilTags to Create a Map

Selecting `Create Course` within `Courses` opens the course-creation interface implemented in *CreateMapView.swift*. Its first stage, `Create Map`, records the floor markers within the coordinate system described in Section EdukARt's Coordinate System and 2D Minimap.

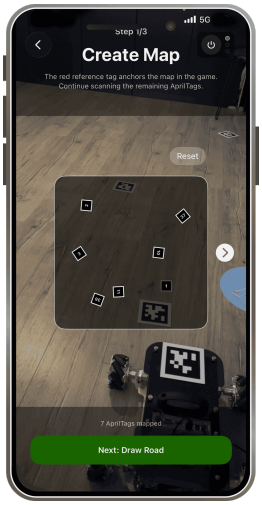


Fig. 4.26: Map creation

Figure 4.26 shows the scanning interface. The camera image helps users locate the physical markers, while the minimap displays their recorded positions and orientations. Users move through the environment to capture further markers and can restart the scan using `Reset`.

Camera frames are processed through *CameraARView.swift*, which uses the SwiftAprilTag library to detect visible markers. *AprilTagLocalization.swift* estimates their poses and expresses them within the map coordinate system. At the beginning of scanning, the smallest currently detected ID is selected as the reference.

The reference is established from 20 valid pose estimates. Their positions and horizontal orientations are averaged before the marker is entered into the map at $(0, 0)$. Its horizontal orientation also determines the orientation of the minimap: rotating the reference marker by 90° before scanning makes the same physical marker arrangement appear rotated in the two-dimensional view. Stricter quality checks are applied because this reference determines the alignment of the entire map. Reference measurements are accepted only within a camera-to-marker distance of 1.2 m.

Once the reference has been established, *AprilTagMapBuilder.swift* collects ten valid measurements for each additional marker and averages its horizontal position and orientation. During testing, pose estimates appeared less reliable at viewing distances of around 4 m. To avoid widely separated map markers, the implementation applies a separate neighbourhood rule: a new marker must lie within 3 m of at least one already mapped marker. This allows the map to grow beyond the immediate area around the reference while encouraging a denser marker arrangement.

The live map is displayed through *AprilTagMapView.swift*, which passes the recorded positions to the shared renderer in *MapCanvasView.swift*. Reusing this renderer keeps the representation consistent with the maps used for course editing and gameplay.

Selecting `Next: Draw Road` fixes the recorded marker geometry for the remaining creation stages.

The implementation separates camera handling, marker localization, map construction, and visualization into dedicated components, as outlined in Table 4.7.

Component	Responsibility
CameraARView.swift	Provides the camera and AR session, detects AprilTags through SwiftAprilTag, and passes their detections to the localization and mapping components.
AprilTagLocalization.swift	Selects and stabilizes the reference marker, checks pose quality, and determines marker positions and orientations within the map coordinate system.
AprilTagMapBuilder.swift	Collects and averages measurements, checks the distance to known markers, and assembles the recorded map points.
AprilTagMapView.swift	Prepares the live map data for display through the shared map renderer.
MapCanvasView.swift	Provides the reusable two-dimensional representation of marker positions, IDs, and orientations.

Table 4.7: Software components responsible for recording and displaying the AprilTags

4.3.2.5 Draw Road: Creating the Virtual Race Track

After recording the AprilTags for the minimap, selecting `Next: Draw Road` opens the second stage of *CreateMapView.swift*. The first approach was to generate a track by connecting the recorded markers. However, this would have tied the route to their physical placement. The markers were intended to support localization, while users should be able to create different tracks within the same environment.

The prototype therefore allows users to draw the track freely on the recorded map, as shown in Fig. 4.29. The markers provide spatial orientation without prescribing the route. Drawing takes place in one continuous stroke, with its beginning defining the start, shown in yellow, and its end defining the finish, shown in green. Selecting `Reset` clears the track so that it can be drawn again. The prototype supports a single run from start to finish instead of multiple laps.

Drawing is handled through *GameMapView.swift* and the shared *MapCanvasView.swift*. Finger positions are converted from display coordinates into horizontal map coordinates and passed to *Course.swift*. *ARTrackRenderer.swift* uses these positions to construct the track's representation within the shared AR reference.

The initial idea was to display a dashed centerline resembling road markings. The recorded points were used directly to construct it, but their spacing varied with drawing speed and small finger movements. This produced inconsistent dash lengths and spacing and also made the construction of road boundaries difficult. Figures 4.27 and 4.28 show early AR visualizations using line segments and virtual cubes to examine the resulting geometry.

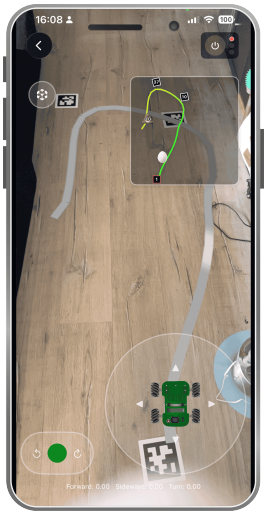


Fig. 4.27

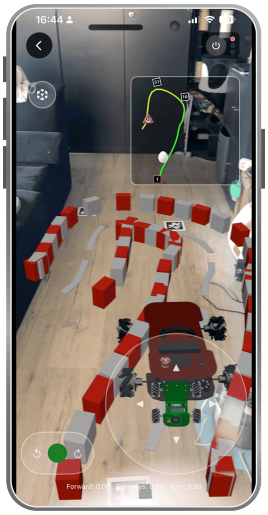


Fig. 4.28



Fig. 4.29

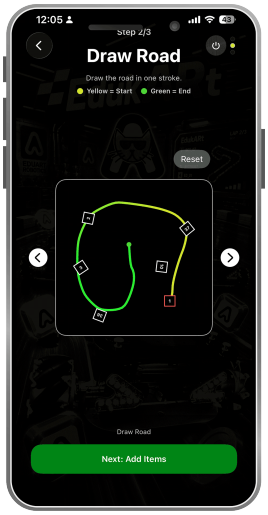


Fig. 4.30

The recorded input was therefore processed before being used for AR rendering. *Course.swift* removes redundant points, smooths the line, and generates new points at approximately 10 cm intervals. This produces a more regular centerline whose point distribution no longer depends directly on drawing speed. Figure 4.29 shows the raw input during drawing, while Fig. 4.30 shows the processed result after the finger is released.

A separate problem concerned the centerline appearing over the physical robot in the AR view. An occlusion model aligned with Eduard was tested to hide the line behind its geometry, while road boundaries were added on either side. Once this representation appeared satisfactory, tests with the moving robot showed that the occlusion model lagged behind Eduard and did not consistently conceal the line. The dashed centerline was therefore removed, while occlusion was retained for virtual game elements.

After several experiments with their appearance, the road boundaries were implemented as continuous white lines positioned 30 cm on either side of the centerline. The resulting width of 60 cm is defined in *TrackRules.swift*. Making this width configurable remains a possible extension to accommodate robots of different sizes.

Selecting `Next: Add Items` continues to the placement of interactive game elements, described in Section Add Items and Obstacles: Placing Game Elements. Table 4.8 summarizes the components responsible for track input, processing, and rendering.

Component	Responsibility
GameMapView.swift	Combines the recorded marker map with the editable track and forwards drawing input to the track-processing component.
MapCanvasView.swift	Displays the map and track, captures finger movements, and converts display positions into map coordinates.
Course.swift	Collects drawing input and simplifies, smooths, and resamples it into the processed track centerline.
ARTrackRenderer.swift	Constructs the three-dimensional road surface, boundaries, and start and finish markings from the track data.
TrackRules.swift	Defines the shared physical road width used for the track representation.

Table 4.8: Software components responsible for track creation and representation

4.3.2.6 Add Items and Obstacles: Placing Game Elements

The final stage of course creation allows users to place items and obstacles on the recorded map. Their positions use the same coordinate system as the track, allowing them to be displayed within the physical surroundings during gameplay.

Early experiments reused the map's touch interaction for both drawing the track and placing objects. An editing mode selected whether input created track points, coins, or itemboxes. Although this allowed objects to be positioned, switching between drawing and placement modes was considered less intuitive during development. Object placement was therefore separated from track drawing. A horizontal palette grouped the available elements into items and obstacles, and drag-and-drop was introduced to combine object selection and positioning in a single gesture.



Fig. 4.31: Placing items and obstacles

Figure 4.31 shows the `Items / Obstacles` stage in *CreateMapView.swift*. The horizontally scrollable palette contains four manually placeable items and four obstacles, represented by symbols and names. Users hold an element and drag it onto the map, where its symbol appears.

`Undo` removes the most recent placement, while `Reset` clears all placed objects without removing the recorded markers or track. Existing objects cannot currently be moved directly, so correcting their positions requires undoing placements or resetting the arrangement. When an element is dropped, *MapCanvasView.swift* converts its display position into map coordinates and passes these to *GameMapView.swift*, which creates the placed object.

Selecting `Save Course` completes course creation. New courses are assigned a name, and *GameMap.swift* combines the reference marker ID, recorded marker positions and orientations, processed track, and placed objects. *GameMapStore.swift* saves these data locally in `game-maps.json`, allowing courses to be reopened after the application has been closed.

After saving, the application returns to `Courses`, where stored courses can be edited or deleted. Editing begins at `Draw Road`, allowing the track and object arrangement to be changed while retaining the recorded marker map. Creating another course currently requires a new marker scan.

The following section describes the creation of the virtual game elements and their interaction with the robot. Table 4.9 summarizes the components responsible for placement and course storage.

Component	Responsibility
MapCanvasView.swift	Displays placed objects, receives drag-and-drop input, and converts drop positions into map coordinates.
GameMapView.swift	Creates placed objects from map input and applies object-specific placement rules.
GameMap.swift	Defines object types and placement data and combines them with the recorded markers and track in a complete course.
GameMapStore.swift	Saves and loads courses locally and manages updates and deletion of stored courses.

Table 4.9: Software components responsible for object placement and course storage

4.3.3 Virtual Game Elements: Creation and Interaction

The selected game elements connect virtual events with robot behavior through collection and transport tasks, temporary changes to the controls, and obstacles that restrict movement. The elements were developed as examples of possible interactions, with gameplay balance left for further development.

The prototype includes four placeable item types, four obstacles, and coins. In general, items disappear after being used or activated, while obstacles remain active in the game environment throughout the race. All of them trigger lighting feedback on the robot and haptic feedback on the iPhone, as described in the following sections.

4.3.3.1 Creation and Preparation of 3D Assets

The creation of the game assets followed an iterative workflow. For the coin and itembox, two-dimensional designs were prepared in Adobe Illustrator and exported as DXF files. These designs were imported into Autodesk Fusion, where the shapes were extruded and their edges rounded. Parts requiring different colors or materials were kept as separate bodies so that they could be edited independently during subsequent preparation.

The models were exported as OBJ files and imported into Blender for material assignment and geometry optimization. Figures 4.32, 4.33, and 4.34 illustrate the preparation of the itembox across the three applications.

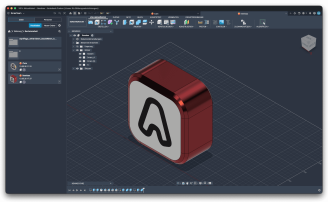


Fig. 4.32: Itembox in Autodesk Fusion



Fig. 4.33: Itembox in Blender



Fig. 4.34: Itembox in Reality Composer Pro

Other elements were adapted from existing models. The tree, stone, egg, oil, and water lily assets were obtained through Envato [52], while the egg cup and poop models were obtained from MakerWorld [53, 54].

Initial versions contained more geometry and texture data than were needed for their representation in the mobile application. The models were therefore modified and optimized in Blender according to their individual structure and the requirements of the application. Different approaches were tested, including the Decimate modifier, merging components, removing duplicate geometry, and adjusting materials and textures. The aim was to retain recognizable shapes while reducing the amount of data loaded and rendered by the application.

Some materials created in Blender did not appear as expected in the application. Reality Composer Pro was therefore used to adjust their appearance and prepare the final assets. This also allowed simpler material configurations to replace parts of the original texture setup. The final models were included in EdukARt as USDZ files.

Table 4.10 shows the reductions achieved for selected assets during a documented optimization step. The same general workflow was also applied to the virtual Eduard model, whose detailed CAD geometry included many components unnecessary for its on-screen representation.

Asset	**Before**	**After**
Coin	1.51 MB	72.1 kB
Egg	6.27 MB	34.1 kB
Itembox	1.80 MB	301.6 kB
Oil	2.40 MB	194.3 kB
Tree	11.74 MB	254.2 kB
Water	8.86 MB	221.3 kB

Table 4.10: File sizes of selected USDZ assets before and after optimization

4.3.3.2 Spatial Representation and Interaction

CameraARView.swift loads the corresponding models and places them relative to the shared map reference. Their two-dimensional map positions become horizontal positions within the AR scene, while height offsets and animations determine their visual presentation. The same placement data are used for both views, with three-dimensional models in the AR environment and simplified icons on the minimap. Changes during gameplay, such as collecting an item or revealing a hidden obstacle, update the AR representation, while the minimap retains the original object arrangement.

Interaction detection uses the shared two-dimensional map coordinates described in Section EdukARt's Coordinate System and 2D Minimap instead of collisions between models in the AR scene. *CollissionManager.swift* checks whether the robot overlaps an object's interaction area within this shared coordinate system. An overlap triggers the corresponding interaction, such as removing a collected item from the AR view or applying an effect to the robot.

The implementation distinguishes between entering and leaving an interaction area, preventing a single overlap from continuously generating new entry events. Blocking obstacles additionally restrict movement commands that would lead into their blocking area.

GameController.swift maintains the active game elements and processes the resulting events. Depending on the element, this can remove an object, change its state, update the recorded result, or request a temporary movement effect. *GameView.swift* connects these requests to the shared *RobotController.swift*, which coordinates movement and lighting feedback using the robot-control functions described previously. New lighting effects can replace ongoing ones without restoring them afterwards. Haptic feedback on the iPhone also helps players notice interactions. During a race, the robot's speed is normally limited to 80 % of the configured maximum. Items and obstacles can change this limit.

Visual overlap with the physical robot is handled separately through a simplified Eduard model with an occlusion material. This model follows the detected robot pose and hides virtual content behind its geometry. As discussed in Draw Road: Creating the Virtual Race Track, delays in its alignment were noticeable during movement.

The responsibilities for representation, interaction detection, game state, and robot feedback are divided between the components listed in Table 4.11.

Component	Responsibility
GameMap.swift	Defines placed-object data and object-specific properties, including models, scales, and interaction radii.
CameraARView.swift	Loads, positions, and animates virtual elements and updates their representation when the game state changes.
CollissionManager.swift	Detects entry and exit events within two-dimensional interaction areas and checks movement against blocking obstacles.
GameController.swift	Maintains the active game state, processes interactions, and coordinates their gameplay effects and haptic feedback.
RobotController.swift	Applies movement restrictions, temporary control effects, speed changes, and lighting feedback through the shared robot controls.
EduardSimulation.swift	Includes the simplified robot representation and the occlusion component used to hide virtual content behind Eduard.

Table 4.11: Software components responsible for representing game elements and processing their interactions

4.3.3.3 Itembox: From Item Generator to Bonus Item

The itembox was the first item implemented and was originally intended to reveal a random usable item after collection. This would have required collected items to be used in some way, which only seemed meaningful in the intended concept if there were opponents or other players. This functionality was therefore initially set aside. However, as one of only two EduArt-specific items, the itembox was still intended to remain in the game and was given the role of a bonus item that provides a 20 s time bonus. It therefore acts as a counterpart to the coin, which is described in more detail later.

Ultimately, developing the itembox led to the realization that items should be directly visible on the track, making their effects immediately apparent. Driving over an item should feel appealing and trigger an effect that players would want to experience. This led to the development of the poop item described next.

Figure 4.35 shows the itembox representation in the game environment together with its main properties (Table 4.12).

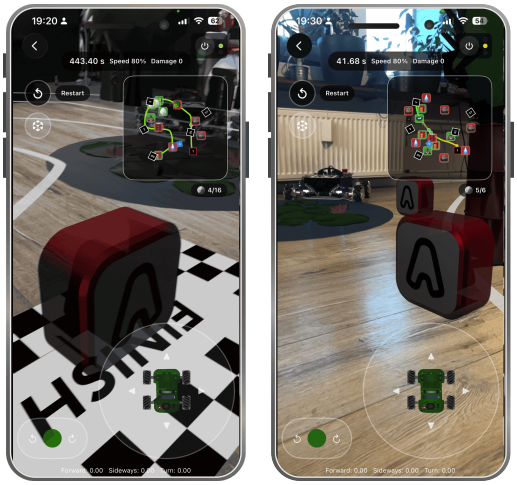


Fig. 4.35: Itembox

Property	Specification
Interaction radius	12 cm (object radius)
Lighting	Duration: 0.8 s Mode: Fast blinking Color: Purple
Effect	Collected and removed from the AR view
Impact	Time bonus of 20 s per collected itembox

Table 4.12: Properties of the itembox

4.3.3.4 Poop: Triggering a Zigzag Movement

The poop item was the first attempt to introduce a humorous, emoji-inspired game element that players would want to activate. The initial appeal was the idea of having a track filled with wiggling poop emojis in the living room. Its function followed naturally from the imagined consequences of stepping in something similar: slipping, moving awkwardly, and leaving a trail.

When the robot collides with the item, it automatically moves in a zigzag pattern at reduced speed for 5 s. During this effect, the player cannot influence its movement through the joysticks. The disable control described in Communication: Establishing a Connection to the Physical Robot remains available if the robot needs to be stopped to prevent a collision. The movement also suggests an attempt to wipe dirt off a shoe, while the virtual brown trail left behind visually contaminates the track. The item disappears from the AR view when triggered.

Figure 4.36 shows the poop item and its virtual trail in the game environment together with its main properties (Table 4.13).



Fig. 4.36: Poop item and its virtual trail

Property	Specification
Interaction radius	12 cm (object radius)
Lighting	Duration: 5 s Mode: Fast blinking Color: Orange
Effect	Removed from the AR view when triggered
Impact	Slow zigzag movement for 5 s; virtual brown trail

Table 4.13: Properties of the poop item

4.3.3.5 Egg and Egg Cup: Transporting Virtual Objects

After the poop item introduced a direct effect on robot control, the next idea was to give items a purpose beyond changing movement behavior. Inspired by transport tasks in the RoboCup SML league, objects were to be collected and delivered to a designated location. Eggs and an egg cup were chosen as a playful reference to contemporary humor, with the eggs serving as transport objects and the egg cup as their destination.

The initial idea was to carry up to three eggs simultaneously on the robot's back. However, animating and positioning three separate eggs was considered too complex for the prototype. Instead, the number of carried eggs is unlimited, with a single visible egg moving with the robot to represent the entire collection.

Future versions could develop this approach into a separate task-based mode with manual or autonomous collection and delivery. Towards the end of development, the interaction was given a function within the racing prototype: delivering eggs repairs the simulated damage accumulated through obstacle interactions and restores the robot's normal gameplay speed.

Figure 4.37 shows the egg and egg cup representations in the game environment together with their main properties (Table 4.14).



Fig. 4.37: Eggs and Egg Cup

Property	Specification
Interaction radius	10 cm for both objects
Lighting	Duration: 0.55 s (Egg), 1.5 s (Egg Cup) Mode: Fast blinking Color: Green
Effect	Egg: Collected and transported by Eduard Egg Cup: Receives the carried eggs
Impact	Successful egg delivery restores normal gameplay speed to 80%; can be repeated

Table 4.14: Properties of eggs and egg cup

4.3.3.6 Coins: A Visible Guide Along the Track

Together with the itembox, the coin was among the first game elements developed for EdukARt. It was initially intended to be placed manually along the track, with collected coins counted and displayed at the end of the race. However, this did not yet provide a clear reason to collect them. The other items raised a similar question: if players could see which effect awaited them, why would they interact with it instead of driving around it and reach the finish more quickly? Coins were therefore placed automatically along the track to guide players, as originally intended for the centerline described in Draw Road: Creating the Virtual Race Track. Their collection was initially evaluated through a point-based scoring system that also deducted points for collisions with obstacles, but this approach was discarded due to its complexity. For the final prototype, performance was evaluated using the race time instead, with a time penalty added for each uncollected coin to encourage players to follow the track.

Unlike the manually placed items, coins are generated automatically by *GameController.swift*. Their positions are calculated along the processed track, with the first coin placed 50 cm from the start and subsequent coins at 50 cm intervals. The `ARCoinRenderer` contained in *CameraARView.swift* displays them rotating around the vertical axis and moving slightly up and down. When a coin is collected, Eduard provides yellow lighting feedback and the coin disappears from the AR view.

To encourage collection, coins also temporarily increase the gameplay speed multiplier to 90 % for 2 s. Further collection events can extend the boost duration.

Figure 4.38 shows the coin representation in the game environment together with its main properties (Table 4.15).

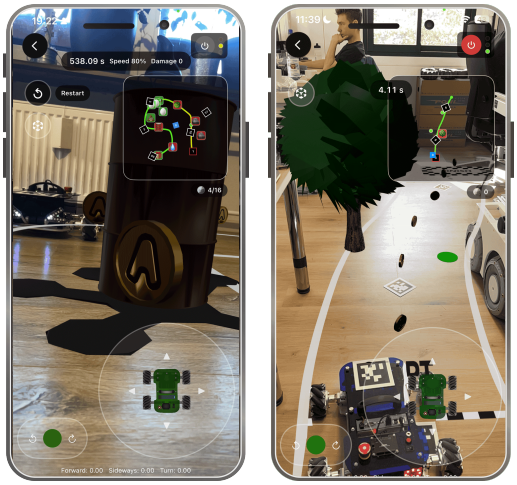


Fig. 4.38: Coin representations during development and gameplay

Property	Specification
Interaction radius	3.5 cm (object radius)
Lighting	Duration: 0.55 s Mode: Fast blinking Color: Yellow
Effect	Collected and removed from the AR view
Impact	Speed increased to 90 % for 2 s; +5 s for each uncollected coin

Table 4.15: Properties of coins

4.3.3.7 Oil: A Persistent Obstacle with a Slipping Effect

Oil was the first obstacle implemented and is represented by a barrel standing in an oil spill. Like the poop item, oil temporarily takes control of the robot's movement. Instead of moving in a zigzag pattern, Eduard rotates on the spot for 5 s, which results in an approximately 180° turn. Unlike poop, the oil obstacle remains in place after the interaction, but a cooldown of 20 s begins when the effect is triggered, preventing immediate repeated activation and giving the player time to move away once control is restored.

Joystick input is disabled during this effect, so the player must wait before regaining control. The barrel also acts as a blocking obstacle: movement commands that would lead into its blocking area are prevented and can cause simulated damage by reducing the normal gameplay speed by ten percentage points, while movement away from it remains possible. A separate 10 s damage cooldown prevents further speed reductions from the same barrel during this interval.

Figure 4.39 shows the oil obstacle in the game environment together with its main properties (Table 4.16).



Fig. 4.39: Oil

Property	Specification
Interaction radius	Trigger: 18 cm Blocking: 7 cm
Lighting	Duration: 5 s Mode: Slow blinking Color: Light red
Effect	Automatic rotation for 5 s
Impact	Blocks movement into the barrel; -10 percentage points of speed per collision; 20 s rotation cooldown; 10 s damage cooldown

Table 4.16: Properties of the oil obstacle

4.3.3.8 Stone: Blocking Eduard's Path

The stone follows the blocking principle introduced with the oil barrel, but without an additional automatic movement effect. The initial idea was to draw continuous virtual walls on the map, represented in AR by boxes similar to those used in the early road-boundary experiments. However, the robot sometimes became trapped and could no longer move, so individual blocking objects were used instead. Since simple grey cubes appeared visually uninteresting in AR, an adapted Stonehenge model was used for the prototype. Different stone models could be introduced in future versions and combined to form walls or other barriers. Registered blocking collisions reduce the normal gameplay speed by ten percentage points, with a 10 s damage cooldown per stone.

Figure 4.40 shows the stone obstacle in the game environment together with its main properties (Table 4.17).

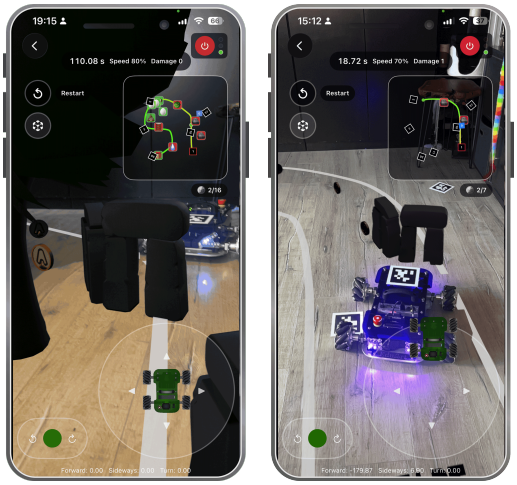


Fig. 4.40: Stone

Property	Specification
Interaction radius	Blocking: 15 cm (object radius)
Lighting	Duration: 0.8 s Mode: Slow blinking Color: Dark red
Effect	Persistent obstacle; movement into its blocking area is prevented
Impact	Normal gameplay speed reduced by ten percentage points per collision; 10 s damage cooldown per stone

Table 4.17: Properties of the stone obstacle

4.3.3.9 Tree: An Unexpected Obstacle

While the stone was intended primarily as a visible track boundary, the tree was designed as a less obvious obstacle that surprises the player. It was inspired by *Trees Hate You* by Tykenn, in which trees confront players and interfere with their progress [55].

During course creation, the player places the tree's trigger on the minimap instead of positioning the tree directly. Both the trigger, shown as a green circle, and the tree are aligned with the track's centerline to obstruct the robot's path. When the trigger is activated, the tree quickly grows out of the ground and reduces the robot's normal gameplay speed by ten percentage points if the robot collides with it.

Figure 4.41 shows the tree trigger and the appearing obstacle in the game environment together with their main properties (Table 4.18).



Fig. 4.41: Tree trigger and tree

Property	Specification
Interaction radius	Trigger: 18 cm Tree blocking: 10 cm
Lighting on appearance	Duration: 3 s Mode: Fast blinking Color: Green
Lighting on damage	Duration: 0.8 s Mode: Slow blinking Color: Dark red
Effect	Grows out of the ground when triggered
Impact	Normal gameplay speed reduced by ten percentage points; 10 s damage cooldown per tree

Table 4.18: Properties of the tree obstacle

4.3.3.10 Water: Switching to Swimming Controls

The final obstacle represents a large puddle of water on the track. While the robot remains within the water's interaction area, it responds differently to joystick input. Instead of moving continuously while the joystick is held, Eduard moves approximately 5 cm forward per stroke, while lateral strokes cover considerably less distance. This difference was deliberately retained as a playful analogy to sideways swimming being less efficient than swimming forward. The joystick must return to its neutral position before another movement can be triggered. Repeated joystick movements therefore resemble swimming strokes and require more active input to cross the puddle.

Figure 4.42 shows the water obstacle in the game environment together with its main properties (Table 4.19).



Fig. 4.42: Water

Property	Specification
Interaction radius	Trigger: 40 cm (object radius)
Lighting	Duration: While water mode is active Mode: Fast blinking Color: Blue
Effect	Short swimming strokes of approximately 5 cm; joystick must return to neutral between strokes
Impact	Repeated input required while inside the puddle

Table 4.19: Properties of the water obstacle

Attempts were also made to reproduce the described item and obstacle effects for the virtual AR robot. Most effects, including lighting feedback and movements such as rotation, could be implemented with approximately comparable behavior, while adapting the swimming controls used for water remained difficult and requires further refinement.

Overall, the implemented items and obstacles show some of the possibilities for connecting virtual game events with physical robot behavior. With the course and its interactive elements defined, the following section concludes the implementation chapter by describing the gameplay accessed through the `Start Game` menu button and revisiting individual implementation components in the context of a race.

4.3.4 Start Game: Running a Race

After creating the map and saving the course as described in Add Items and Obstacles: Placing Game Elements, users can enter the game mode through `Start Game` in the main menu. They first select an existing course, which is then opened in a new ARKit session.

To reduce loading delays when opening the AR view, *ContentView.swift* loads the virtual Eduard model asynchronously through *EduardModelStore.swift* while the application's opening logo is displayed. Unlike the other assets, Eduard's many individual components limited how much its model could be reduced. The remaining approximately 2.5 MB had caused a delay of several seconds when loaded directly upon opening the camera view. The model is therefore retained for reuse, allowing *CameraARView.swift* to create a copy without loading the asset again.

The application initially displays a preview of the selected course's minimap, as shown in Fig. 4.43, and asks users to scan its reference AprilTag, which is highlighted in red on the minimap. The reference is established from 20 valid pose estimates and used to align the saved map with the current session, following the approach described in Scanning AprilTags to Create a Map.

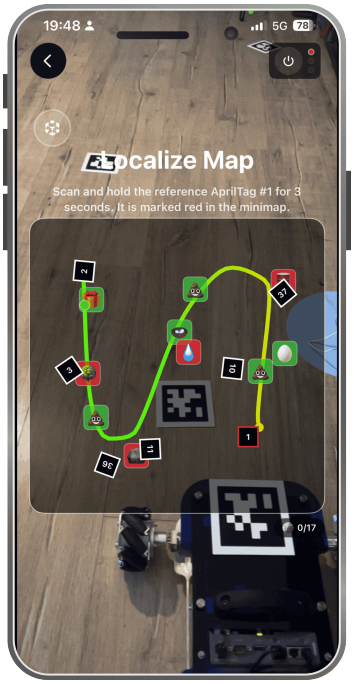


Fig. 4.43: Course Localization

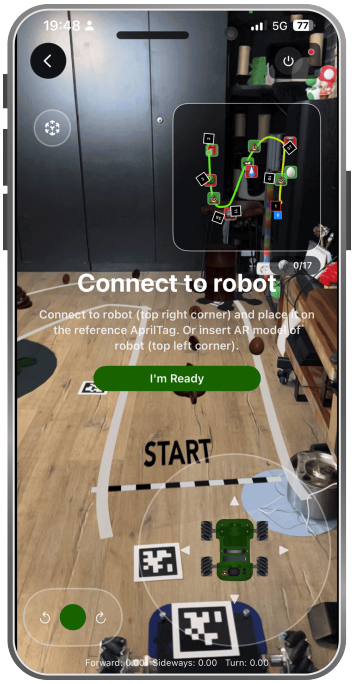


Fig. 4.44: Game Scene

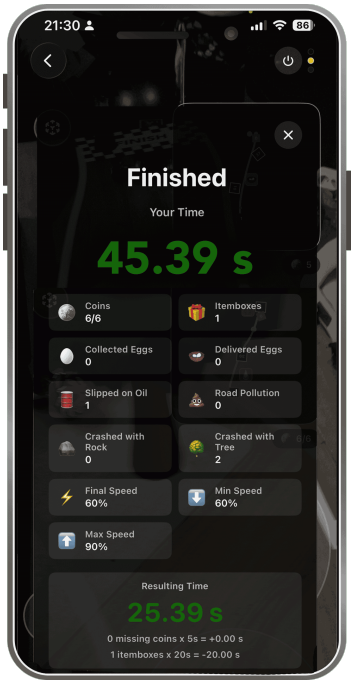


Fig. 4.45: Race Results and Leaderboard

Once this reference has been established, the iPhone's AR view displays the track and the virtual representations of the items and obstacles described in Virtual Game Elements: Creation and Interaction, as shown in Fig. 4.44. Start and finish markings are also displayed automatically by *ARTrackRenderer.swift*, using image textures positioned near the beginning and at the end of the drawn track and aligned with its direction. The minimap remains visible in the upper-right corner and shows the robot's current estimated position in blue, using the localization approach described in EdukARt's Coordinate System and 2D Minimap.

Before starting the race, players can activate and position Eduard or insert the virtual robot using the available controls. This allows the course and robot localization to be checked while the stopwatch and item collection remain inactive. These preparations are coordinated by *GameView.swift*, which also manages the countdown and result display. The `I'm Ready` button becomes enabled once the robot is positioned behind the beginning of the track. Selecting it starts a countdown during which robot control is locked, preventing an early start.

Once the countdown ends, robot control is released and *GameController.swift* starts measuring the elapsed time and processing interactions with the items and obstacles described in Virtual Game Elements: Creation and Interaction. The objective is to achieve the best final time by reaching the finish quickly, for which players should collect all coins along the route to avoid time penalties. The prototype does not check whether the robot remains within the road boundaries or follows the complete route. Coin penalties therefore provide an incentive to follow the track, while explicit out-of-bounds detection remains a possible extension.

After the robot has left the start area, reaching the area around the final track point ends the race and stops the stopwatch. This is determined from the robot's position within the shared map coordinate system. Eduard is automatically disabled and signals completion through rainbow lighting. The application displays the elapsed time together with the final time after applying all bonuses and penalties, as shown in Fig. 4.45. Players can enter their name and save the result to the course's leaderboard, which ranks runs by their final time after bonuses and penalties. Results are stored locally by *GameController.swift* and can be accessed through `Scores` in the main menu.

After closing the result display, the game scene remains available and users can inspect the remaining objects. They can also reactivate Eduard to continue free driving or return to the start. Gameplay changes affect a temporary copy of the placed objects, leaving the saved course unchanged. Once the robot is behind the start again, selecting `Start New Game` restores the original object arrangement, resets the race statistics and temporary effects, and starts another countdown. The course's alignment within the active AR session is retained, so no additional reference-marker scan is required.

The following chapter, Results and Discussion, presents the results achieved with the EdukARt prototype and discusses how its technical implementation and initial player feedback support the intended game concept.

Chapter 5

Results and Discussion

This chapter presents and discusses the results of the thesis. Following a brief summary of the implemented game, it examines the individual components described in the preceding chapters. The discussion draws on observations made during development and combines verbal feedback and online survey responses from approximately 15 test users who tested different parts of the application under varying conditions, without following a standardized study procedure. This informal mixed-methods approach to user experience research was applied during development and an interim presentation of EdukARt, attended by school students, professors, and other adults. Most participants provided verbal feedback, while five completed the online survey [56].



Fig. 5.1: EdukARt Keyvisual

In this thesis, EdukARt was implemented as a prototype of a mixed-reality racing game in which the physical robot Eduard serves as an active game element. Players can use an iPhone to steer Eduard along a self-created track displayed within the physical surroundings and interact with virtual items and obstacles. Figure 5.1 illustrates the game concept. These interactions affect the robot's lighting and movement and can also produce virtual effects. The prototype addresses the central research question by demonstrating how Eduard can be integrated into an MR game through its existing interfaces, without modifying its drive hardware or replacing its control software. The game also prompted participants to move through the environment to follow the robot, while their feedback indicated motivation to build and use their own robot for EdukARt.

Building on Eduard's existing Wi-Fi network and ROS2 interfaces made it possible to integrate the robot into the game with only limited changes. The application provides the connection controls shown in Fig. 4.4. An additional Python receiver acts as a UDP-to-ROS2 bridge, forwarding application commands to the corresponding topics and services, as described in Communication: Establishing a Connection to the Physical Robot. The existing drive-control software and hardware remained unchanged, while game-specific behavior was implemented primarily within the application.

The receiver provides a generic communication bridge that, after a one-time installation and configuration for the robot's ROS2 environment, allows the application to send messages and service requests through the robot's available interfaces. This separation provides a basis for supporting other ROS2 robots, although integration was tested only with Eduard.

The combination of Wi-Fi and the Python receiver provided a suitable communication interface for the prototype, with some minor inconveniences associated with using Eduard's network. Its lack of Internet access required switching networks for developer-certificate verification during development and caused iOS to display warnings during use. Although occasional connection interruptions did not noticeably interfere with gameplay during testing, they were made visible through ping-based reachability checks and the persistent connection-status indicator described in Communication: Establishing a Connection to the Physical Robot.

Integrating the robot into the EdukARt game required a shared spatial reference for its position and the virtual game elements. AprilTags were used both to map the environment and to locate Eduard within it.

The resulting map coordinate system, represented by the minimap, combined the origin and horizontal orientation of a reference AprilTag with ARKit's gravity-aligned Y-axis, providing consistent floor alignment.

The shared map connected two-dimensional course editing with the placement of virtual elements on the physical floor, as described in EdukARt's Coordinate System and 2D Minimap. Repeated comparisons between visible AprilTags and their expected map positions helped maintain this alignment as players moved through the room, reducing drift over time and distance. Figure 5.2 shows a spatial visualization of the mapped marker relationships.

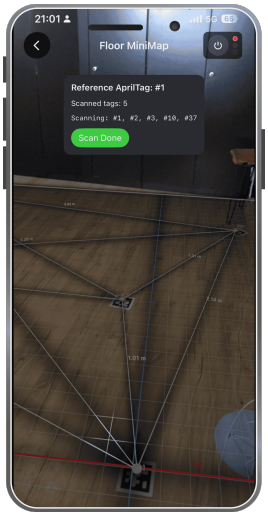


Fig. 5.2: Shared spatial reference



Fig. 5.3: Course creation

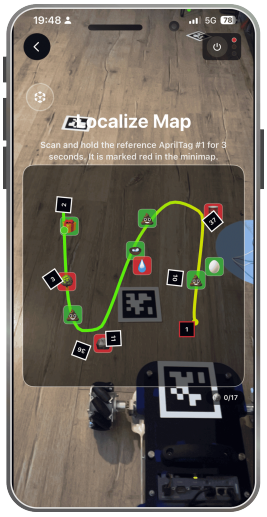


Fig. 5.4: Course relocalization

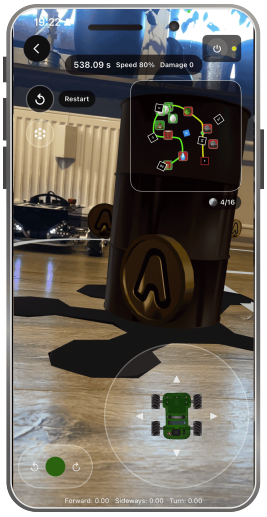


Fig. 5.5: Coin representation in AR

Figure 5.4 shows the relocalization of a saved course within the physical environment. Small offsets of a few centimetres at greater distances did not noticeably interfere with gameplay during testing. Both the survey responses and verbal feedback described the course placement as stable, without noticeable jitter, and praised the correspondence between the two-dimensional map and its placement on the physical floor. This stability depended on the floor markers remaining in their recorded positions so that they could support relocalization and ongoing map correction.

Although LiDAR initially motivated the choice of native iOS development, the implemented spatial reference ultimately relied on AprilTags and ARKit's world tracking without environmental reconstruction. This shows that the demonstrated game interactions did not require a complete scan of the room. However, physical obstacles were consequently not represented automatically in the map, and support for other devices or platforms was not evaluated.

The shared map also provided the basis for course creation. Users could draw a track and place items and obstacles through the two-dimensional editor, as shown in Fig. 5.3. Saved courses could then be reused in later sessions. Detecting the recorded reference marker restored their placement within the room, with the minimap preview in Fig. 5.4 helping identify the required marker.

Feedback on the robot's localization within this map was mixed. Using the iPhone for tracking and visualization avoids the fixed external cameras and projectors used in several systems discussed in Background and Related Work. However, players must keep the robot and its AprilTag visible in the camera view, which requires some practice.

Detection was also affected by lighting conditions and reflections, particularly from direct sunlight.

Different approaches were observed during testing: Some participants found it intuitive to use Eduard's Mecanum movement without rotating the robot. This sometimes led them to follow it less closely, increasing the viewing distance and reducing localization reliability. Others remained in one position but rotated the robot while driving, causing Eduard's carrying handle to sometimes obscure the AprilTag and interrupt its detection. Participants who followed the robot and remained behind it generally experienced fewer difficulties interpreting the controls and maintained more consistent localization. These observations suggest that a short guided introduction could help players understand the relationship between joystick input, viewing position, and marker visibility.

The implemented game elements described in Virtual Game Elements: Creation and Interaction demonstrate several ways of connecting virtual events with physical robot behavior. Coins, shown in AR in Fig. 5.5, and item boxes influence the final race time, while poop and oil trigger automatic movement effects. Poop, shown with its virtual trail in Fig. 5.6, emerged as the most popular element, and water's alternative joystick controls were also positively received. Stones and trees restrict movement and eggs and egg cups demonstrate the collection, transport, and delivery of virtual objects using the physical robot.



Fig. 5.6: Poop item



Fig. 5.7: Water item

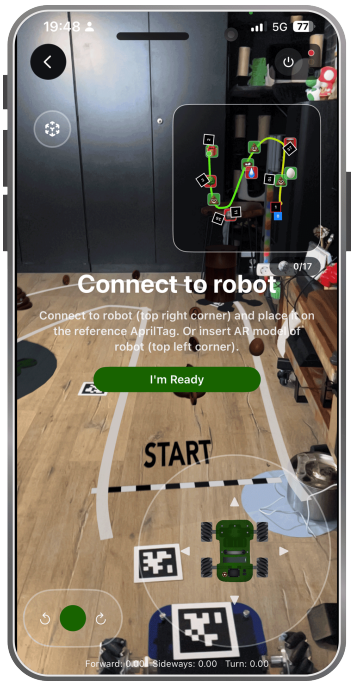


Fig. 5.8: Race preparation

Lighting and haptic feedback supported the perception of these interactions. Figure 5.7 shows Eduard's blue lighting while the water effect is active. Overlapping lighting effects and the absence of a priority system still limited the consistency of this feedback, as described in Spatial Representation and Interaction.

The recorded course and its game elements could be brought together in a playable race, as shown in Fig. 5.8. Courses could be replayed and results compared through course-specific leaderboards, providing a basis for successive players to compete on the same track.

Feedback was divided on the fairness of the time bonuses and penalties. For example, penalizing missed coins was described as unfair, although this mechanism was intended to encourage players to follow the track. Respondents also indicated that they would take shortcuts if possible, highlighting the need to detect whether the robot remains on the track. These responses show that the implemented effects still need to be combined into a more convincing game concept, with clearer incentives to follow the route and interact with its elements.

Despite the remaining need to refine the gameplay mechanics, the prototype was well received overall, from course creation to its technical operation. No application crashes were reported during the informal playtests, and participants found the game visually appealing and enjoyable. Their feedback expressed interest in building their own robot for EdukARt, playing it at exhibitions or at home, and recommending it to others. These findings support the potential of the implemented integration approach, while showing that its technical functionality is more developed than its overall gameplay design.

Chapter 6

Conclusion

EdukARt combines the idea of physical movement found in Pokémon GO with the shared and competitive play of Mario Kart, using a physical robot as an active game element. Inspired by initiatives such as RobotsUnite, the aim was to give existing robots an additional playful application that could encourage engagement with robotics.

This concept was implemented using Eduard and an iPhone. Integration was achieved through the robot's existing control interfaces, an additional communication bridge, and a shared marker-based spatial reference connecting its position with virtual game elements. Game-specific behavior was implemented primarily within the application, allowing the robot's existing hardware and drive-control software to be retained. The prototype therefore demonstrates how an already operational robot can become part of a mobile mixed-reality game without extensive modifications.

Playing EdukARt encourages players to move through the environment to keep the robot within the iPhone's camera view, making physical movement part of the experience. Testing also showed that following Eduard closely helped maintain reliable localization and made its controls easier to interpret. The movement envisioned in the introduction thus became a practical part of gameplay. Recorded race results provide a basis for comparing performance on the same course, while shared challenges involving multiple robots remain a possible extension of the social game concept.

For school initiatives and university robotics teams, the use of an existing robot could provide an opportunity to apply their systems outside the competitions or projects for which they were originally developed. Adapting the robot-specific communication interfaces could extend the approach to further platforms, including self-built robots. The implemented game elements also explore possibilities beyond racing: collecting, transporting, and delivering virtual eggs introduces task-based interaction. Such mechanics could be relevant to RoboCup SML teams and provide a starting point for game modes based on completing robotic tasks.

Initial testing and feedback from approximately 15 participants indicated that the prototype was appealing and enjoyable. Participants expressed interest in building and using

their own robot for EdukARt. Students and professors alike responded enthusiastically in informal discussions and contributed ideas for how they could use the application. This interest supports the envisaged use across different educational and professional contexts, including robotics projects and demonstrations at exhibitions.

The EdukARt prototype connects physical movement, playful competition, and practical engagement with robotics in a mobile mixed-reality game. The enthusiasm expressed by students and professors provides a strong motivation to continue exploring its possibilities. The project will be continued within the Master of Applied Research (MAPR) programme, building on the prototype to explore further robot platforms, game modes, and educational applications. The following chapter outlines these opportunities for future development and research.

Chapter 7

Future Work

EdukARt will be continued and refined within the Master of Applied Research programme. Further development will address the technical and usability issues identified throughout Design and Implementation of EdukARt, with the aim of turning the prototype into a more reliable and accessible application.

A third prototype will focus on use at trade fairs and exhibitions, building on Eduard's existing demonstration role described in Company and Project Context. Visitors should be able to try the game, while the live AR view and race information are transmitted to a larger display so that spectators can follow the interaction between the robot and virtual elements. This would make EdukARt both a playable activity and a visually engaging demonstration that attracts attention to the booth. The setup should be adaptable to different exhibition spaces and support repeated short game sessions.

Broader use will require examining where and how EdukARt would be played and which functions are relevant in these settings. The prototype currently targets iPhones and uses marker-based mapping without LiDAR-based environmental reconstruction. Building on Native iOS Development, further work should determine whether LiDAR and more precise spatial information provide meaningful advantages for the intended applications and whether a cross-platform implementation would be appropriate.

Robot independence and gameplay will also be developed further. Building on the interface described in Communication: Establishing a Connection to the Physical Robot, documentation is planned for constructing a compatible robot and integrating existing self-built or industrial robot platforms. Further testing will guide the revision of the game mechanics and the development of different modes, including track-based racing and task-based interactions such as collecting and transporting objects. Further studies should examine whether the initial interest leads to sustained use, physical activity, and engagement with robotics.

List of Figures

The EdukARt keyvisual combines AI-generated content with other image material assembled and edited by the author in Adobe Photoshop. Sources of externally created assets are identified in the corresponding sections.

3.1 Eduard in fluorescent red color with Off-Road wheels . . . 16
3.2 Blue Eduard with mecanum wheels and LiDAR on top . . . 16
3.3 Robot Arthur with red boxes indicating the fields of view of its surrounding sensor ring Vision360 . . . 17
3.4 Resulting Vision360 point cloud of a room . . . 17
3.5 Kinematics Kit Kim . . . 17

4.1 EdukARt Keyvisual . . . 20
4.2 System architecture of the EdukARt prototype . . . 24
4.3 Not Connected . . . 26
4.4 Not Enabled . . . 26
4.5 Enabled + Connected . . . 26
4.6 Joystick layout and wheel directions for Mecanum movement . . . 30
4.7 Light control . . . 32
4.8 LiDAR mesh visualization of the environment . . . 34
4.9 Floor tiles aligned beneath a desk . . . 35
4.10 Displacement of the mesh and floor tiles . . . 35
4.11 Floor tiles with a physical reference marker . . . 35
4.12 ARKit World Coordinate System . . . 36
4.13 AprilTag Coordinate System . . . 36
4.14 EdukARt Map Coordinate System . . . 36
4.15 Camera Distances While Sitting . . . 38
4.16 Camera Distances While Standing . . . 38
4.17 World Coordinates While Sitting . . . 38
4.18 World Coordinates While Standing . . . 38
4.19 AprilTag distances in a linear arrangement . . . 39
4.20 AprilTags in a trapezoidal arrangement . . . 39
4.21 AprilTags in an L-shaped arrangement . . . 39
4.22 Robot localization viewed from tag 1 . . . 39

4.23 Robot localization viewed from tag 2 . . . 39
4.24 Virtual cubes . . . 40
4.25 Spatial relationships between mapped AprilTags . . . 40
4.26 Map creation . . . 41
4.27 Early AR track visualization using line segments . . . 43
4.28 Early AR track visualization using virtual cubes . . . 43
4.29 Track geometry during drawing . . . 43
4.30 Processed track geometry after drawing . . . 43
4.31 Placing items and obstacles . . . 45
4.32 Itembox in Autodesk Fusion . . . 47
4.33 Itembox in Blender . . . 47
4.34 Itembox in Reality Composer Pro . . . 47
4.35 Itembox . . . 50
4.36 Poop item and its virtual trail . . . 51
4.37 Eggs and Egg Cup . . . 52
4.38 Coin representations during development and gameplay . . . 53
4.39 Oil . . . 54
4.40 Stone . . . 55
4.41 Tree trigger and tree . . . 56
4.42 Water . . . 57
4.43 Course Localization . . . 58
4.44 Game Scene . . . 58
4.45 Race Results and Leaderboard . . . 58

5.1 EdukARt Keyvisual . . . 60
5.2 Shared spatial reference . . . 62
5.3 Course creation . . . 62
5.4 Course relocalization . . . 62
5.5 Coin representation in AR . . . 62
5.6 Poop item . . . 63
5.7 Water item . . . 63
5.8 Race preparation . . . 63

List of Tables

3.1 Comparison of Unity and Unreal Engine. 11
3.2 Depth sensing on LiDAR-equipped iPhones [44] and Android devices[45]. . . 14
3.3 Technical specifications and configurations of Eduard 19

4.1 Development, documentation, and design tools used for the prototype 21
4.2 Technical environment of the EduArt robot used for the prototype 23
4.3 Software components for robot communication and connection control 29
4.4 Software components for controlling the physical and virtual robot 32
4.5 Mapping of EdukARt's lighting controls to Eduard 33
4.6 Software components responsible for lighting control and visual game feedback 33
4.7 Software components responsible for recording and displaying the AprilTags . 42
4.8 Software components responsible for track creation and representation 44
4.9 Software components responsible for object placement and course storage . . 46
4.10 File sizes of selected USDZ assets before and after optimization 47
4.11 Software components responsible for representing game elements and processing their interactions 49
4.12 Properties of the itembox 50
4.13 Properties of the poop item 51
4.14 Properties of eggs and egg cup 52
4.15 Properties of coins 53
4.16 Properties of the oil obstacle 54
4.17 Properties of the stone obstacle 55
4.18 Properties of the tree obstacle 56
4.19 Properties of the water obstacle 57

Bibliography

[1] Sabine Feierabend, Thomas Rathgeb, Yvonne Gerigk, and Stephan Glöckler. KIM-Studie 2024: Kindheit, Internet, Medien. Basisuntersuchung zum Medienumgang 6- bis 13-Jähriger. Technical report, Medienpädagogischer Forschungsverbund Südwest (mpfs), Stuttgart, June 2025. 2. aktualisierte Auflage. Accessed October 21, 2026. URL: https://mpfs.de/app/uploads/2025/05/KIM-Studie-2024.pdf. 1

[2] OECD. *How's Life for Children in the Digital Age?* OECD Publishing, Paris, 2025. Accessed October 21, 2026. URL: https://www.oecd.org/en/publications/how-s-life-for-children-in-the-digital-age_0854b900-en.html, doi:10.1787/0854b900-en. 1

[3] Tessa Strain, Seth Flaxman, Regina Guthold, Elizaveta Semenova, Melanie Cowan, Leanne M. Riley, Fiona C. Bull, Gretchen A. Stevens, and Country Data Author Group. National, regional, and global trends in insufficient physical activity among adults from 2000 to 2022: a pooled analysis of 507 population-based surveys with 5.7 million participants. *The Lancet Global Health*, 12(8):e1232–e1243, 2024. URL: https://www.thelancet.com/journals/langlo/article/PIIS2214-109X(24)00150-5/fulltext, doi:10.1016/S2214-109X(24)00150-5. 1

[4] Domenic Angelino. Did Pokémon GO really make people healthier? what the data says, 2026. Accessed September 14, 2026. URL: https://getfitcraft.com/de/blog/pokemon-go-fitness. 1

[5] Helmut Schatz. Health-promoting effects of Pokémon GO through increased physical activity? new smartphone games should encourage even more activity!, 2017. Accessed September 14, 2026. URL: https://blog.endokrinologie.net/pokemen-go-3181/. 1

[6] YouTube. Pokémon GO player gathering. Video; accessed September 22, 2026. URL: https://www.youtube.com/watch?v=gokcItN99-0. 1

[7] Johannes Gehrling. Best Nintendo games: Top 10 Switch games that run on the Switch 2 thanks to backward compatibility, 2025. Accessed September 22, 2026. URL: https://www.nordbayern.de/service/gaming/nintendo-switch-2-top-10-spiele-von-der-alten-switch-abwartskompatibel-mario-kart-zelda-evgr-rep-1-1.14628335. 1

[8] Miriam Held. Fitness apps: Six apps that make workouts more fun, 2022. Accessed September 14, 2026. URL: https://www.gq-magazin.de/leben-als-mann/fitness/gamification-fitness-apps. 1

[9] COMPUTER BILD. Almost half of german households use smart technology, 2025. Accessed September 14, 2026. URL: https://www.computerbild.de/artikel/News-Smart-Home-Smart-Home-boomt-Fast-jeder-zweite-deutsche-Haushalt-ist-schon-vernetzt-40690317.html. 1

[10] Fortune Business Insights. Game based learning market size, share and industry analysis, 2026–2034, 2026. Updated September 7, 2026; accessed September 22, 2026. URL: https://www.fortunebusinessinsights.com/game-based-learning-market-116405. 1

[11] Anna Küch. German students perform worse than ever in the PISA study, 2026. Accessed September 14, 2026. URL: https://www.tagesschau.de/inland/gesellschaft/pisa-deutschland-schueler-100.html. 1

[12] WELT. Majority considers the german school system outdated and obsolete, 2021. Accessed September 14, 2026. URL: https://www.welt.de/politik/deutschland/article234261428/Mehrheit-haelt-das-deutsche-Schulsystem-fuer-ueberholt-und-veraltet.html. 1

[13] TeacherW. Robots unite!, 2023. Accessed September 14, 2026. URL: https://www.printables.com/education/516309-robots-unite. 1

[14] Nuremberg Institute of Technology Georg Simon Ohm. autonOhm: RoboCup@work team. Accessed October 13, 2026. URL: https://www.th-nuernberg.de/fakultaeten/efi/forschung-an-der-fakultaet-efi/forschungsaktive-labore/mobile-robotik/robocup-work/team/. 1

[15] EduArt Robotik GmbH. EduArt: Developing robotics solutions. Accessed October 13, 2026. URL: https://www.eduart-robotik.com/. 1

[16] Martin Bauernschmitt. RoboCup rapidly manufactured robot challenge 2023, 2023. Accessed September 15, 2026. URL: https://www.francor.de/2023/11/03/robocup-rapidly-manufactured-robot-challenge-2023/. 1

[17] RoboCup Federation. RoboCup Federation official website, 2026. Accessed September 14, 2026. URL: https://www.robocup.org/. 1

[18] RoboCup Federation. RoboCup 2026 in seoul, 2026. Accessed September 14, 2026. URL: https://www.robocup.org/robocup-2026-in-seoul/. 1

[19] RoboCup Germany. Rapidly manufactured robot competition (RMRC), 2026. Accessed September 14, 2026. URL: https://robocup.de/ligen/rmrc. 1

[20] Patrick Leber and Kris Dalm. Using MR to interact with a mobile robot based on ROS. In *Proceedings of the 12th Conference on Learning Factories (CLF 2022)*, April 2022. Available at SSRN. URL: https://ssrn.com/abstract=4074341, doi:10.2139/ssrn.4074341. 2.1.1, 2.1.3

[21] Ian Chen, Bruce Macdonald, and Burkhard Wünsche. Mixed reality simulation for mobile robots. pages 232–237, June 2009. doi:10.1109/ROBOT.2009.5152325. 2.1.1, 2.1.2, 2.1.3

[22] Open Robotics. Gazebo Sim: A robotic simulator, 2026. Accessed September 18, 2026. URL: https://gazebosim.org/libs/sim/. 2.1.1

[23] Hyunseok Kang, Uiseok Lee, Yechan Park, Dugki Min, Tuan Anh Nguyen, and Eunmi Choi. Metaverse-ready simulation: Advancing sim-to-real by reinforcement learning for TurtleBot in ROS2 and Unreal Engine. In *2025 International Conference on Metaverse Computing, Networking and Applications (MetaCom)*, pages 7–12, 2025. doi:10.1109/MetaCom65502.2025.00008. 2.1.2

[24] Prakash Aryan, Sujala Deepak Shetty, V. Kalaichelvi, and R. Karthikeyan. SimNav-XR: An extended reality platform for mobile robot simulation using ROS2 and Unity3D. *Frontiers in Robotics and AI*, 13, 2026. URL: https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2026.1708161, doi:10.3389/frobt.2026.1708161. 2.1.2, 2.1.3

[25] Xingjian Li, Jeremy Park, Chris Reberg-Horton, Steven Mirsky, Edgar Lobaton, and Lirong Xiang. Photorealistic arm robot simulation for 3d plant reconstruction and automatic annotation using Unreal Engine 5. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 5480–5488, 2024. doi:10.1109/CVPRW63382.2024.00557. 2.1.2

[26] Katharina Pentenrieder, Christian Bade, Fabian Doil, and Peter Meier. Augmented reality-based factory planning: An application tailored to industrial needs. pages 31–42, December 2007. doi:10.1109/ISMAR.2007.4538822. 2.1.3

[27] Ian Chen, Bruce Macdonald, and Burkhard Wünsche. Designing a mixed reality framework for enriching interactions in robot simulation. pages 331–338, January 2010. 2.1.3

[28] Hyunsung Tae, Saedong Yeo, Sukhyun Hwang, Seogu Park, and Gyuhwan Hwang. System-in-the-loop test system with mixed reality for autonomous ground vehicle

(AGV) and military applications. *IEEE Access*, 13:46383–46394, 2025. doi:10.1109/ACCESS.2025.3549372. 2.1.3

[29] Niantic. Catching Pokémon in AR mode, 2026. Accessed September 18, 2026. URL: https://niantic.helpshift.com/hc/en/6-pokemon-go/faq/28-catching-pokemon-in-ar-mode-1712012768/. 2.2

[30] Unity Technologies. Unity Technologies supports iOS 11 and ARKit, September 2017. Accessed September 22, 2026. URL: https://unity.com/news/unity-technologies-supports-ios-11-and-arkit. 2.2

[31] Directive Games. The machines, 2017. Apple App Store; accessed September 18, 2026. URL: https://apps.apple.com/us/app/the-machines/id1280682965. 2.2

[32] AR Critic. AR Robot – game review (iOS), September 2018. Accessed September 22, 2026. URL: https://arcritic.com/2464/ar-robot-game-review-ios/. 2.2

[33] Nintendo. How does Mario Kart Live: Home Circuit work?, 2026. Accessed September 18, 2026. URL: https://www.nintendo.com/au/support/articles/how-does-mario-kart-live-home-circuit-work/. 2.2

[34] BattleKart Entertainment. BattleKart: Frequently asked questions, 2026. Accessed September 22, 2026. URL: https://www.battlekart.com/de/faq/. 2.2

[35] Jakob Leitner, Michael Haller, Kyungdahm Yun, Woontack Woo, Maki Sugimoto, and Masahiko Inami. IncreTable: A mixed reality tabletop game experience. In *Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, pages 9–16, New York, NY, USA, 2008. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/1501750.1501753, doi:10.1145/1501750.1501753. 2.3.1

[36] Aleksander Krzywinski, Haipeng Mi, Weiqin Chen, and Masanori Sugimoto. RoboTable: A tabletop framework for tangible interaction with robots in a mixed reality. In *Proceedings of the International Conference on Advances in Computer Entertainment Technology*, pages 107–114, New York, NY, USA, 2009. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/1690388.1690407, doi:10.1145/1690388.1690407. 2.3.1

[37] Adam Sosa, Richard Stanton, Stepheny Perez, Christian Keyes-Garcia, Sara Gonzalez, and Phoebe O. Toups Dugas. Imperfect robot control in a mixed reality game to teach hybrid human-robot team coordination. In *Proceedings of the 2015 Annual Symposium on Computer-Human Interaction in Play*, pages 697–702, New York, NY, USA, 2015. Association for Computing Machinery. URL: https:

//dl.acm.org/doi/10.1145/2793107.2810288, doi:10.1145/2793107.2810288. 2.3.2

[38] Fabrizio Lamberti, Davide Calandra, Federica Bazzano, Filippo G. Pratticò, and Davide M. Destefanis. RobotQuest: A robotic game based on projected mixed reality and proximity interaction. In *2018 IEEE Games, Entertainment, Media Conference (GEM)*, pages 196–199. IEEE, 2018. URL: https://ieeexplore.ieee.org/document/8516501, doi:10.1109/GEM.2018.8516501. 2.3.2

[39] José Nuno Amaro Freixo. Simulating and upgrading PiGaming mixed reality setup with artificial intelligence. Master's thesis, Faculty of Engineering, University of Porto, July 2020. Integrated Master's dissertation. URL: https://repositorio-aberto.up.pt/handle/10216/128963. 2.3.3

[40] Hackr.io. Unity vs. Unreal Engine: Which game engine should you choose?, 2026. Accessed September 22, 2026. URL: https://hackr.io/blog/unity-vs-unreal-engine. 3.1.1

[41] LaptopLegend. Mac studio for game development: Unity and Unreal Engine on macOS, May 2026. Accessed September 23, 2026. URL: https://evezone.evetech.co.za/daily-drop/mac-studio-for-game-development-unity-and-unreal-engine-on-macos. 3.1.1

[42] Statcounter Global Stats. Mobile operating system market share worldwide – september 2026, September 2026. Based on mobile web usage rather than counts of smartphone users; accessed October 5, 2026. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-202609-202609-bar. 3.1.2

[43] ATNO For AR / VR / MR / XR Developer. ARCore vs ARKit in 2026: The definitive comparison every AR developer needs, March 2026. Medium; accessed October 5, 2026. URL: https://medium.com/@atnoforarvrdeveloper/arcore-vs-arkit-in-2026-the-definitive-comparison-every-ar-developer-needs-5f42b93c9034. 3.1.2

[44] Apple Inc. Visualizing and interacting with a reconstructed scene, n.d. Apple Developer Documentation; accessed September 24, 2026. URL: https://developer.apple.com/documentation/arkit/visualizing-and-interacting-with-a-reconstructed-scene. 3.1.2.1, 3.2, 3.1.2.2, 4.3.2.1, 7

[45] Google LLC. Depth adds realism, n.d. ARCore Developer Documentation; accessed September 24, 2026. URL: https://developers.google.com/ar/develop/depth. 3.1.2.1, 3.2, 7

[46] Apple Inc. RealityKit, n.d. Apple Developer Documentation; accessed September 24, 2026. URL: https://developer.apple.com/documentation/realitykit. 3.1.2.2

[47] Michael Ellis. SwiftUIJoystick: A customizable joystick made with SwiftUI. GitHub repository. Accessed October 14, 2026. URL: https://github.com/michael94ellis/SwiftUIJoystick. 4.3.1.2

[48] Apple Inc. ARConfiguration.WorldAlignment.gravity, n.d. Apple Developer Documentation; accessed October 14, 2026. URL: https://developer.apple.com/documentation/arkit/arconfiguration/worldalignment-swift.enum/gravity. 4.3.2.1

[49] RoboCup@Work. RoboCup@Work rulebook 2024, 2024. Section 3.4.3: April Tagged Object Set. URL: https://atwork.robocup.org/wp-content/uploads/2024/07/Rulebook2024.pdf. 4.3.2.2

[50] Edwin Olson. AprilTag. Apple App Store. iOS application; accessed October 14, 2026. URL: https://apps.apple.com/de/app/apriltag/id736108128. 4.3.2.2

[51] KeyQ, Inc. SwiftAprilTag. GitHub repository. Accessed October 14, 2026. URL: https://github.com/keyqcloud/SwiftAprilTag. 4.3.2.2

[52] Envato: 3D models used for EdukARt. Envato model collection. Individual model sources: Tree; Stone; Egg; Oil barrel; Oil spill; Water lilies. 4.3.3.1

[53] Egg cup model Friedrich. MakerWorld, 3D model. URL: https://makerworld.com/de/models/2943880-egg-cup-model-friedrich#profileId-3297411. 4.3.3.1

[54] Mini poop. MakerWorld, 3D model. URL: https://makerworld.com/de/models/1862625-mini-poop#profileId-2769629. 4.3.3.1

[55] Tykenn. Trees Hate You. Steam, official game description. Accessed October 10, 2026. URL: https://store.steampowered.com/app/4171850/Trees_Hate_You/. 4.3.3.9

[56] EdukARt: Online user survey. Submitted digitally as supplementary material accompanying this thesis. 5

Glossary

Bluetooth beacon A small wireless transmitter that broadcasts Bluetooth signals, allowing nearby devices to detect its presence and estimate its proximity. 8

Indie Short for independent; describes games or development teams that operate independently of major publishers and typically work with smaller teams and budgets. 10

JSON JavaScript Object Notation, a text-based format for storing and exchanging structured data using key-value pairs and arrays. 27

LiDAR Light Detection and Ranging, an active sensing method that emits laser pulses and measures their return time to determine distances and represent the surrounding environment. 4

Mixed Reality (MR) Systems in which physical and virtual elements share a spatial context and influence the same interaction or system state. 1, 2

Point Cloud A collection of points in three-dimensional space, typically generated by sensors such as LiDAR or depth cameras to represent the surfaces and geometry of an environment. 3

ROS (Robot Operating System) An open-source software framework for developing robotic applications, providing tools and message-based communication between components such as sensors, actuators, and controllers. 4

Time-of-Flight Sensor A distance sensor that measures the time required for an emitted signal, typically light, to travel to an object and return to the sensor. 4

User Datagram Protocol (UDP) A communication protocol that allows the transmission of data packets over a network. 23, 27