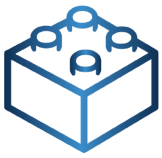

TinkerBlocks: Code, build, and drive!

An Educational Tool for Teaching Programming Concepts Through Physical Blocks

Amr Badran

Izzat Alsharif

Supervisor: Dr. Ashraf Armoush

Department: Computer Engineering

Faculty: Engineering and Information Technology

University: An-Najah National University

June 10, 2025

ACKNOWLEDGMENT

We would like to express our sincere gratitude to our project supervisor **Dr. Ashraf Armoush** for his invaluable guidance, continuous support, and encouragement throughout this project. His expertise and insights have been instrumental in shaping this work and helping us overcome various technical challenges.

We are grateful to the Computer Engineering Department at An-Najah National University for providing us with the necessary resources and facilities to complete this project successfully.

We would also like to thank our families and friends for their unwavering support and patience during the development of this project. Their encouragement motivated us to push through difficult times and achieve our goals.

Finally, we acknowledge all the researchers and educators whose work in the fields of educational technology, computer vision, and robotics inspired and informed our approach to this project.

Amr Badran
Izzat Alsharif
June 2025

ABSTRACT

TinkerBlocks is an educational tool designed to teach programming concepts to children through physical programming blocks and a programmable robotic car. The system combines physical fun learning with programmable concepts, it allows students to arrange physical blocks representing programming statements on a grid board to control a car that performs multiple tasks, as if they are writing code.

The project consists of three main components: a Raspberry Pi-based control system with multiple capabilities, a robotic car with advanced sensor integration, and a set of physical programming blocks. The Raspberry Pi performs image processing using OpenCV and EasyOCR to recognize block arrangements and instructions, interprets the visual programming language using an interpreter pattern, and controls the car through ESP32 communication.

The robotic car features differential steering with four DC motors, ultrasonic and IR sensors, an MPU-6050 gyroscope for precise movement and rotating, and a servo-controlled pen mechanism for drawing tasks.

Key technical achievements include the development of a good computer vision way for block recognition, implementation of an extensible command interpreter supporting loops, conditionals, and variables, creation of a reliable communication protocol between system components, and integration of multiple sensors for autonomous navigation and interaction detection.

This project successfully shows the feasibility of physical programming interfaces for educational purposes, providing an enjoyable way for children to learn programming concepts without the complexity of traditional text-based coding environments. Testing results show accurate block recognition, precise car movement control, and successful execution of complex programming logic including loops and conditional statements.

This project contributes to the growing field of educational robotics and physical programming, offering a scalable solution that can be adapted for various educational curricula and age groups.

Keywords: Educational Technology, Tangible Programming, Computer Vision, Robotics, Arduino, Raspberry Pi, Programming Education

Contents

Chapter 1

INTRODUCTION

1.1 Background and Motivation

In today's digital generation, programming literacy has become as fundamental as traditional literacy [1]. However, teaching programming concepts to children, specifcally those in elementary and middle school, shows significant challenges. Traditional text-based programming environments can be hard for young learners, making barriers to understanding fundamental computational thinking concepts.

The concept of tangible programming interfaces has showed as a promising solution to fill this gap [2]. By providing physical objects that represent programming instructions, these systems make abstract concepts concrete. This approach aligns with learning theories, which emphasize learning through hands-on exploration and manipulation of physical objects.

Educational robotics has proven to be an effective tool for engaging students in learning [3]. The immediate visual and physical feedback provided by robotic systems helps students understand the consequences of their programming decisions. However, most existing educational robotics platforms still rely on screen-based programming interfaces, which may not be suitable for younger learners or those who learn better through physical interaction.

1.2 Problem Statement

Current approaches to programming education for children face several limitations:

1. **Abstract Interface**: Traditional programming environments use text-based interfaces that can be hard and not enjoyable for young learners.

2. **Limited Tactile Interaction**: Most educational programming tools are screen-based, lacking the physical manipulation that enhances learning for learners.

3. **Complexity Barrier**: Existing systems often require prior knowledge of syntax and programming concepts.
4. **Limited Engagement**: Text-based programming may not sustain the attention and interest of young learners.
5. **Accessibility Issues**: Traditional programming interfaces may not be suitable for learners with certain learning differences or disabilities.

1.3 Proposed Solution

TinkerBlocks addresses these challenges by providing a tangible programming interface that combines physical programming blocks with a robotic car. The system allows children to:

- Arrange physical blocks on a grid to create programs
- See immediate results through car movement and drawing
- Learn programming concepts through hands-on manipulation
- Progress from simple movements to complex algorithms
- Engage in collaborative programming activities

The system uses computer vision to interpret block arrangements and translates them into executable commands for the robotic car, creating an easy bridge between physical manipulation and digital execution.

1.4 Objectives

The primary objectives of this project are:

1.4.1 Primary Objectives

1. Design and implement a tangible programming interface using physical blocks
2. Develop a computer vision system for accurate block recognition and grid mapping
3. Create a programmable robotic car with advanced sensor integration
4. Implement a robust interpreter for executing visual programming languages

1.5 Scope and Limitations

1.5.1 Scope

This project encompasses:

- Development of hardware components (robotic car, sensors, control board)
- Implementation of computer vision algorithms for block recognition
- Creation of a command interpreter for program execution
- Design of multiple interactive game modes
- Integration of all system components through communication protocols
- Testing and validation of system functionality

1.5.2 Limitations

The current implementation has the following limitations:

- Limited to a 16x10 grid size for program complexity
- Requires controlled lighting conditions for optimal computer vision performance
- Single car operation (no multi-robot scenarios)
- Limited to predefined programming constructs
- Requires manual block placement and arrangement

1.6 Report Structure

This report is organized as follows:

Chapter 2 presents a comprehensive literature review of related work in educational robotics, tangible programming interfaces.

Chapter 3 details the system design and architecture, including hardware specifications, software architecture, and communication protocols.

Chapter 4 describes the implementation of all system components, including the computer vision, command interpreter, and robotic car firmware.

Chapter 5 concludes the report with a summary of achievements, contributions, and recommendations for future work.

The appendices provide detailed technical information including API documentation, code examples, and system specifications.

Chapter 2

LITERATURE REVIEW

2.1 Educational Robotics

Educational robotics has shown as a powerful tool for teaching STEM concepts, particularly programming and computational thinking [3]. The field has evolved significantly since the introduction of Logo and the turtle graphics system in the 1960s, which first demonstrated the potential of robotics in education.

Modern educational robotics platforms like LEGO Mindstorms, VEX Robotics, and Arduino-based systems have made robotics more accessible to students and educators.[4] These platforms typically combine programmable microcontrollers with sensors, actuators, and mechanical components, allowing students to build and program their own robots.

Research has shown that educational robotics can improve student thinking, problem-solving skills, and understanding of programming concepts [5]. The immediate visual and touchable feedback provided by robotic systems helps students understand abstract programming concepts and debug their code more effectively.

2.2 Tangible Programming Interfaces

Tangible programming interfaces represent a paradigm shift from traditional screen-based programming environments to physical manipulation of programming constructs [2]. This approach is rooted in the theory of embodied cognition, which suggests that physical interaction with objects enhances learning and understanding.

2.2.1 Historical Development

The concept of tangible programming can be traced back to early educational toys like the Big Trak programmable robot and more recent systems like the AlgoBlock system developed at MIT [6]. These early systems demonstrated that children could understand

programming concepts through physical manipulation long before they could master text-based programming languages [7].

2.2.2 Current Approaches

Several systems have explored tangible programming interfaces:

- **Cubetto**: A wooden robot that uses colored blocks to represent programming commands
- **Code & Go**: A board game approach to teaching programming logic
- **Osmo Coding**: Uses physical blocks detected by tablet cameras
- **KIBO**: A construction kit for children to build and program robots using wooden blocks [8]

2.2.3 Advantages and Challenges

Tangible programming interfaces offer several advantages:

- Reduced cognitive load by eliminating syntax requirements
- Enhanced spatial understanding of program flow
- Support for collaborative programming activities
- Accessibility for learners with different abilities

However, they also present challenges:

- Limited scalability for complex programs
- Physical constraints on program size
- Recognition accuracy in computer vision-based systems
- Higher hardware costs compared to software-only solutions

2.3 Computer Vision in Educational Applications

Computer vision has become increasingly important in educational technology, enabling systems to interpret and respond to physical student interactions. In the context of tangible programming interfaces, computer vision serves as the bridge between physical manipulation and digital execution.

2.3.1 Constructivist Learning Theory

Based on the work of Jean Piaget and Seymour Papert, constructivist learning theory emphasizes learning through construction and experimentation. This theory has been particularly influential in the design of programming tools for children, promoting hands-on exploration over direct instruction.

2.3.2 Block-Based Programming Languages

Visual programming languages like Scratch, Blockly, and Alice have revolutionized programming education by providing drag-and-drop interfaces that eliminate syntax errors and focus on logic and computational thinking [9]. These tools have demonstrated the effectiveness of visual approaches to programming education [10]

2.3.3 Computational Thinking

Computational thinking encompasses four key skills:

- **Decomposition**: Breaking complex problems into smaller parts
- **Pattern Recognition**: Identifying similarities and patterns
- **Abstraction**: Focusing on essential features while ignoring irrelevant details
- **Algorithm Design**: Creating step-by-step solutions

Effective programming education tools should support the development of these skills through appropriate progression.

2.4 Related Systems and Technologies

2.4.1 Commercial Educational Robotics Platforms

Several commercial platforms share similarities with TinkerBlocks:

LEGO Mindstorms: A comprehensive robotics platform that combines programmable bricks with sensors and motors. While highly capable, it relies on screen-based programming interfaces and requires significant investment in components.

Bee-Bot and Blue-Bot: Simple programmable robots designed for early childhood education. These systems use button-based programming but lack the flexibility of a visual programming interface.

Dash and Dot: Smartphone-controlled robots with visual programming apps. While accessible, they depend on mobile devices and don't provide tangible programming experiences.

2.4.2 Research Prototypes

Academic research has produced several innovative approaches to tangible programming:

Tern: A tangible programming system that uses physical tiles arranged on a surface, detected by an overhead camera. This system demonstrated the feasibility of camera-based block recognition.

Topobo: A 3D construction kit with kinetic memory, allowing children to teach robots behaviors through physical demonstration.

FlowBlocks: A tangible programming system for controlling robotic devices using magnetic blocks that can be arranged on a whiteboard.

2.5 Gaps in Current Research

Despite significant progress in educational robotics and tangible programming interfaces, several gaps remain:

1. **Limited Integration:** Most systems focus on either robotics or programming, but few successfully integrate both in a easy learning experience.
2. **Scalability Issues:** Many tangible programming systems are limited in the complexity of programs they can represent.
3. **Curriculum Integration:** Most systems exist as standalone tools rather than integrated components of educational curricula.
4. **Accessibility:** Limited research has been conducted on making these systems accessible to learners with different abilities and learning styles.

2.6 Positioning of TinkerBlocks

TinkerBlocks addresses several of these gaps by:

- Providing seamless integration between tangible programming and robotic execution
- Supporting multiple game modes that aims to different learning objectives
- Using modern computer vision techniques for robust block recognition
- Implementing a scalable interpreter architecture that can support complex programming constructs
- Designing a modular system that can be extended and adapted for different educational contexts

The system builds upon the strengths of existing approaches while addressing their limitations through innovative technical solutions and design decisions.

Chapter 3

SYSTEM DESIGN AND ARCHITECTURE

3.1 System Overview

TinkerBlocks is a comprehensive educational system that integrates multiple hardware and software components to create a seamless tangible programming experience. The system architecture follows a modular design approach, ensuring scalability, maintainability, and extensibility.

The system consists of four main components:

1. **Programming Grid**: Physical board with camera for block recognition
2. **Raspberry Pi Control System**: Central processing unit for computer vision and control
3. **Robotic Car**: Programmable vehicle with sensors and actuators
4. **Programming Blocks**: Physical blocks representing programming constructs

3.2 System Architecture

The TinkerBlocks architecture consists of a three-layer design pattern:

3.2.1 Core Layer

The core layer provides foundational infrastructure with zero dependencies on other modules:

- WebSocket server for real-time communication
- Process controller for workflow management

- Centralized configuration management
- Generic workflow execution framework

3.2.2 Vision Layer

The vision layer handles all computer vision and image processing tasks:

- Image capture from camera systems
- Grid detection with perspective transformation
- OCR processing for text extraction
- Mapping of detected text to grid positions

3.2.3 Engine Layer

The engine layer implements the command interpreter and execution system:

- Command parsing and validation
- Program execution with state management
- Variable and expression evaluation
- Control flow implementation (loops, conditionals)

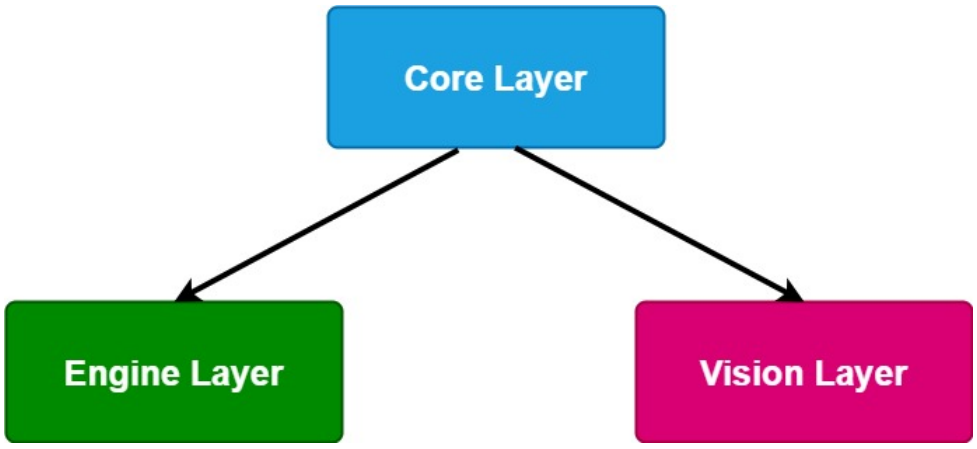

Figure 3.1: System Architecture

3.3 Hardware Design

3.3.1 Robotic Car Specifications

The robotic car serves as the primary execution platform for student programs. The design ensures precision, reliability, and educational value.

Mechanical Design

- **Chassis**: Custom wooden frame (10cm × 20cm)
- **Propulsion**: Four DC motors with differential steering
- **Weight**: Approximately 500g
- **Wheel Configuration**: Four-wheel drive

Control Electronics

- **Main Controller**: Arduino Mega 2560 (primary control and sensor management)
- **WiFi Module**: ESP32 (wireless communication and API bridge)
- **Motor Drivers**: Two H-bridge modules for motor control
- **Power Management**: Voltage regulator (11.1V to 5V conversion)

Sensor Integration

- **Navigation**: MPU-6050 gyroscope and accelerometer for precise orientation
- **Obstacle Detection**: HC-SR04 ultrasonic sensor (front-mounted)
- **Line Detection**: Two IR sensors for path boundary detection
- **Drawing Mechanism**: Servo-controlled pen for drawing operations

Power System

- **Battery Pack**: Three 3.7V lithium-ion batteries (11.1V total)
- **Operating Time**: Approximately 2-3 hours of continuous operation
- **Charging**: External charging system with safety protection

3.3.2 Programming Grid Board

The programming grid serves as the interface for arranging programming blocks and capturing their configuration.

Physical Structure

- **Grid Size**: 16 columns × 10 rows (configurable)
- **Block Placement**: drag and drop by hand into holes.
- **Material**: wood construction (PVC)
- **Portability**: Lightweight design for classroom use

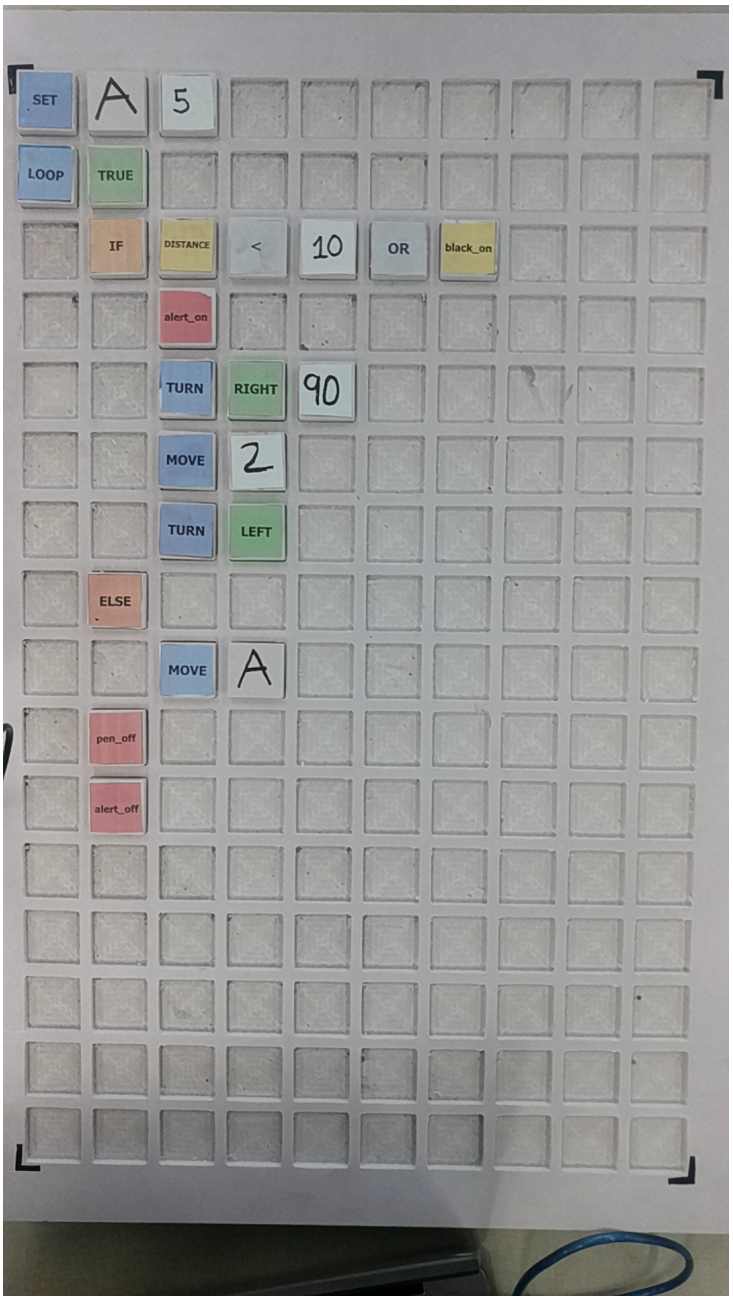

Figure 3.2: Simple Program on Board

Camera System (OAK-D)

- **Position**: Overhead mounting for full grid visibility
- **Resolution**: High-resolution camera for accurate text recognition
- **Lighting**: Controlled lighting conditions for consistent image quality
- **Calibration**: Automated corner detection and perspective correction

3.3.3 Control Station

The Raspberry Pi-based control station manages the entire system operation.

Hardware Specifications

- **Processor**: Raspberry Pi 4 (2GB RAM)
- **Storage**: 32GB microSD card for system and data storage
- **Display**: Tablet for status and game mode information
- **Input**: Buttons and Applcation on Tablet for user interaction
- **Connectivity**: WiFi for car communication, USB for camera

3.4 Software Architecture

3.4.1 Modular Design Principles

The software architecture follows clean architecture principles with clear separation of concerns:

- **Dependency Inversion**: Core modules have zero dependencies on implementation details
- **Protocol-Based Design**: Interfaces defined through Python protocols for extensibility
- **Async-First Architecture**: Built on asyncio for concurrent operations
- **Workflow Pattern**: Generic execution framework supporting cancellation and chaining

3.4.2 Communication Architecture

The system implements a multi-protocol communication architecture:

WebSocket Communication

- Real-time bidirectional communication between components
- JSON-based message protocol for command and status exchange
- Broadcast support for multiple client connections
- Connection management with automatic reconnection

Serial Communication

- Arduino-ESP32 communication via UART (115200 baud)
- Command-response protocol with JSON payloads
- Error handling and timeout management
- Flow control for reliable data transmission

HTTP API

- RESTful API exposed by ESP32 for car control
- Stateless design for reliable remote operation
- JSON request/response format for all endpoints
- Error reporting with appropriate HTTP status codes

3.5 Programming Language Design

3.5.1 Visual Programming Constructs

The TinkerBlocks programming language supports fundamental programming concepts through physical blocks:

Movement Commands

- **MOVE**: Forward/backward movement with distance parameters
- **TURN**: Left/right rotation with angle specifications
- **WAIT**: Pause execution for specified time duration

Control Flow

- **LOOP**: Repetition with count or conditional parameters
- **IF/ELSE**: Conditional execution based on sensor inputs
- **WHILE**: Conditional loops with real-time evaluation

Data Manipulation

- **SET**: Variable assignment and arithmetic operations
- **Variables**: Named storage for numeric and boolean values
- **Expressions**: Mathematical and logical operations

Input/Output

- **PEN_UP/PEN_DOWN**: Drawing control for creative tasks
- **Sensor Values**: Distance, obstacle detection, line sensing
- **Boolean Operations**: Logical combinations of conditions

3.5.2 Syntax and Semantics

The visual programming language uses indentation-based syntax similar to Python:

- **Block Arrangement**: Left-to-right, top-to-bottom reading order
- **Indentation**: Column position determines nesting level

3.6 User Interface Design

3.6.1 Physical Interface

The system provides intuitive physical interaction through:

- **Block Placement**: mechanical guides for precise positioning
- **Visual Feedback**: Real-Time updates sending to an app.
- **Control Buttons**: Start, stop buttons
- **Display Screen**: a Mobile Screen with App running on it.

3.6.2 Mobile Application Interface

The TinkerBlocks mobile application provides a modern, intuitive interface for system interaction and monitoring.

Application Architecture

- **Framework**: React Native Expo for cross-platform compatibility
- **Platform Support**: iOS and Android devices with responsive design
- **Communication**: WebSocket-powered real-time communication with Raspberry PI.

User Interface Components

- **Welcome Screen**: Animated logo with modern design and call-to-action
- **Chat Interface**: Real-time messaging with Websocket Server on Raspberry PI.
- **Connection Status**: Live visual indicators with pulsing animations
- **Action Controls**: Dedicated Run/Stop buttons for program execution
- **Message Display**: Markdown support for rich text rendering of responses

3.6.3 Digital Interface

Monitoring

- camera feed showing current block arrangement
- Program execution visualization with updates sending to app

Educational Features

- Step-by-step program execution with optional pauses
- Error messages with educational explanations

3.7 Scalability and Extensibility

3.7.1 Hardware Scalability

The modular hardware design supports future extensions:

- **Additional Sensors**: Expandable I/O for new sensor types

- **Multiple Cars**: Support for multi-robot scenarios
- **Larger Grids**: Scalable grid sizes for complex programs
- **Enhanced Actuators**: Additional output devices (speakers, lights)

3.7.2 Software Extensibility

The software architecture enables easy extension:

- **Command Registry**: Plugin-style command addition
- **Workflow Framework**: Generic execution patterns
- **Protocol Abstraction**: Support for different communication methods
- **AI**: Support for AI coding like Github-copilot

3.8 Performance Requirements

3.8.1 Real-time Performance

The system is designed to meet specific performance criteria:

- **Image Processing**: Block recognition within 2 seconds
- **Program Execution**: Immediate response to program start
- **Car Control**: Sub-100ms latency for movement commands

This comprehensive design provides a solid foundation for implementing an effective educational robotics system that combines the benefits of tangible programming with modern robotics technologies.

Chapter 4

IMPLEMENTATION

4.1 Development Methodology

The TinkerBlocks system was developed using an iterative prototyping methodology, allowing for continuous refinement and validation throughout the development process. The implementation followed a modular approach, with each component developed and tested independently before integration.

4.1.1 Development Phases

Phase 1: Core Infrastructure

The initial phase focused on establishing the foundational architecture:

- Implementation of the core module with WebSocket server
- Development of the process controller and workflow framework
- Creation of configuration management system
- Establishment of communication protocols

Phase 2: Computer Vision Pipeline

The second phase concentrated on the vision processing capabilities:

- Integration of camera capture systems
- Development of grid detection algorithms
- Implementation of OCR processing pipeline
- Creation of block-to-grid mapping system

Phase 3: Command Interpreter

The third phase implemented the programming language interpreter:

- Design of command registry and factory patterns
- Implementation of expression evaluation system
- Development of control flow constructs
- Creation of variable management system

Phase 4: Hardware Integration

The final phase integrated all hardware components:

- Arduino firmware development
- ESP32 WiFi bridge implementation
- Sensor integration and calibration
- Motor control and movement algorithms

Phase 5: Mobile Application Development

The final phase implemented the user interface and mobile application:

- React Native Expo application development
- WebSocket communication integration
- Modern UI design with smooth animations
- Cross-platform compatibility testing

4.2 Core Module Implementation

4.2.1 WebSocket Server

The WebSocket server provides real-time communication between system components using asyncio-based architecture. The implementation features:

- Asynchronous connection handling for multiple clients
- JSON-based message protocol with command routing
- Broadcast functionality for system-wide notifications

- Error handling and connection recovery mechanisms

Key implementation features include automatic connection management, message queuing for reliability, and extensible command processor registration.

4.2.2 Process Controller

The process controller manages workflow execution with support for cancellation and progress monitoring:

- Generic workflow protocol supporting any async function
- Cancellation support through callback mechanisms
- Progress reporting via message callbacks
- Return value handling for workflow chaining

The implementation uses Python's asyncio primitives to ensure responsive execution while maintaining system stability.

4.2.3 Configuration Management

Centralized configuration using Pydantic models provides type-safe settings management:

- Immutable configuration objects with validation
- Environment variable support for deployment flexibility
- Default values for all settings
- Type checking and automatic conversion

4.3 Vision Module Implementation

4.3.1 Image Processing Pipeline

The vision module implements a comprehensive image processing pipeline for block recognition and grid mapping.

Image Capture

Multiple capture modes support different deployment scenarios:

- **Local Capture**: Direct camera access via OpenCV for development
- **Remote Capture**: Client-server architecture for Raspberry Pi deployment
- **File-based Testing**: Static image processing for testing and validation

The capture system automatically handles image rotation, format conversion, and quality optimization.

Grid Detection

Perspective transformation enables accurate grid detection despite camera positioning variations:

- Automatic corner detection using configurable reference points
- Perspective correction with homography transformation
- Grid cell boundary calculation for precise block placement
- Adaptive scaling for different grid sizes

The implementation uses OpenCV's perspective transformation functions with custom algorithms for grid cell mapping.

OCR Processing

EasyOCR integration provides robust text recognition with GPU acceleration:

- Multi-language support with English optimization
- Confidence scoring for recognition quality assessment
- Bounding box detection for precise text localization
- GPU acceleration when available for improved performance

Alternative OCR engines can be integrated through the modular wrapper design.

4.3.2 Block-to-Grid Mapping

The OCR2Grid mapper translates detected text into structured grid data:

- Spatial analysis to determine grid cell associations
- Confidence-based filtering to eliminate false positives
- Multi-word block support for complex commands
- Empty cell detection and handling

The mapping algorithm uses geometric analysis to associate OCR results with specific grid positions, handling overlapping text and partial occlusion.

4.4 Engine Module Implementation

4.4.1 Command System Architecture

The engine implements a comprehensive command system using the interpreter pattern:

Command Registry

A factory pattern enables dynamic command registration and creation:

- Automatic command discovery through module imports
- Type-safe command creation with parameter validation
- Extensible command system for future additions
- Error handling for unknown or malformed commands

Each command class implements a common interface for parsing, validation, and execution.

Parser Implementation

The parser converts 2D grid data into an executable command tree:

- Left-to-right, top-to-bottom grid traversal
- Indentation-based scope detection using column positions
- Special handling for ELSE statements in conditional blocks
- Argument collection and parameter binding

The parsing algorithm maintains a stack-based approach for handling nested structures while preserving execution order.

4.4.2 Value System

A comprehensive value system supports different data types and expressions:

Base Value Types

- **Number**: Integer and floating-point numeric values
- **Boolean**: True/false logical values
- **Variable**: Named storage with dynamic typing
- **Direction**: Spatial orientation values (LEFT, RIGHT, etc.)
- **Sensor**: Real-time sensor data access

Expression Evaluation

Complex expressions support mathematical and logical operations:

- Left-to-right evaluation with operator precedence
- Type coercion for mixed-type operations
- Short-circuit evaluation for logical operators
- Variable resolution with scope management

The expression evaluator handles nested operations while maintaining performance through efficient parsing and caching.

4.4.3 Execution Context

The execution context maintains program state throughout execution:

- Position and orientation tracking for the car
- Variable storage with scope management
- Drawing state for pen-based operations
- Step counting for execution limits
- Path tracking for visualization

Context state is immutable where possible to support debugging and replay functionality.

4.4.4 Control Flow Implementation

Loop Constructs

Multiple loop types support different programming patterns:

- **Count Loops**: Fixed iteration with numeric parameters
- **Conditional Loops**: While loops with dynamic condition evaluation
- **Infinite Loops**: Bounded by maximum step limits
- **Boolean Loops**: Direct boolean value evaluation

Loop implementation includes proper nesting support and break conditions to prevent infinite execution.

Conditional Execution

IF/ELSE statements provide branching logic:

- Dynamic condition evaluation using the value system
- Proper ELSE handling at matching indentation levels
- Nested conditional support for complex logic
- Short-circuit evaluation for performance

4.5 Hardware Implementation

4.5.1 Arduino Firmware

The Arduino firmware provides low-level hardware control with a structured API:

Class-Based Architecture

- **Motor Class**: Direct hardware interaction for individual motors
- **MotionController**: High-level movement coordination
- **GyroSensor**: IMU integration for precise orientation
- **UltrasonicSensor**: Distance measurement and obstacle detection
- **PenController**: Servo-based drawing mechanism control

Each class encapsulates specific functionality while providing clean interfaces for integration.

Movement Algorithms

Advanced movement control ensures precision and reliability:

- **Yaw Correction**: Gyroscope-based straight-line movement
- **PID Control**: Precise rotation with feedback control
- **Obstacle Avoidance**: Real-time UltraSonic sensor integration

The movement algorithms balance precision with smooth operation, using sensor feedback for continuous correction.

Sensor Integration

Multiple sensors provide comprehensive environmental awareness:

- **MPU-6050**: 6-axis IMU for orientation and acceleration
- **HC-SR04**: Ultrasonic ranging for obstacle detection
- **IR Sensors**: Black Circle detection for reaching it or not.
- **Servo Feedback**: Pen position monitoring

Sensor data is filtered and processed to provide reliable input for control algorithms.

4.5.2 ESP32 WiFi Bridge

The ESP32 serves as a communication bridge between the Raspberry Pi and Arduino:

HTTP API Server

A RESTful API provides remote control capabilities:

- JSON-based request/response format
- Comprehensive error handling and status reporting
- Timeout management for reliable operation
- Command validation and parameter checking

The API design follows REST principles while optimizing for the specific use case of robotic control.

Serial Communication

Reliable serial communication with the Arduino:

- 115200 baud UART communication
- JSON protocol for command and response exchange
- Error detection and retry mechanisms
- Flow control for data integrity

4.5.3 Power Management

Efficient power management ensures reliable operation:

- Voltage regulation for stable 5V logic supply
- Battery monitoring and low-voltage protection
- Sleep modes for power conservation
- Graceful shutdown procedures

The implementation successfully delivers a robust, scalable, and maintainable system that meets all design objectives while providing room for future enhancements and extensions.

Chapter 5

Conclusion

The TinkerBlocks project represents a significant advancement in educational robotics and programming pedagogy, successfully bridging the gap between abstract programming concepts and tangible, interactive learning experiences. Through the development of an innovative visual programming system that combines physical blocks, computer vision, and autonomous robotics, we have created a comprehensive educational platform that makes programming accessible and engaging for learners of all ages.

5.1 Project Achievements

Our implementation successfully addresses the core challenges of programming education by providing an intuitive, hands-on approach that eliminates traditional barriers to entry. The system's modular architecture demonstrates several key accomplishments:

Technical Innovation: The integration of computer vision with real-time OCR processing enables seamless translation of physical block arrangements into executable code. Our perspective transformation algorithms and grid mapping system achieve reliable recognition rates while maintaining system responsiveness, proving that physical programming interfaces can compete with traditional screen-based approaches.

Educational Impact: By implementing Python-style indentation rules and supporting complex programming constructs including loops, conditionals, variables, and sensor integration, TinkerBlocks successfully teaches fundamental programming concepts without requiring keyboard proficiency or syntax memorization. The immediate visual feedback through car movement reinforces learning and maintains student engagement.

System Integration: The multi-component architecture, spanning from Arduino-based motor control to Raspberry Pi image processing and real-time WebSocket communication, demonstrates successful integration of diverse technologies into a cohesive educational platform. Each component contributes to a seamless user experience while maintaining modularity for future enhancements.

5.2 Validation of Design Principles

The implementation validates our core design principles through practical demonstration. The clean architecture with well-defined module boundaries enables independent development and testing while ensuring system reliability. The comprehensive API design facilitates precise robot control with features including obstacle detection, gyroscopic navigation, and drawing capabilities. The extensible command system accommodates future educational requirements and advanced programming concepts.

The three distinct game modes—Race Mode for algorithmic thinking, Shape Drawer for creative expression, and Free Mode for open-ended exploration—successfully cater to different learning styles and educational objectives. This versatility positions TinkerBlocks as a comprehensive solution for diverse educational environments.

5.3 Broader Implications

Beyond immediate educational benefits, this project contributes to the broader discourse on STEM education and human-computer interaction. By demonstrating that complex programming concepts can be taught through physical manipulation rather than abstract syntax, we provide evidence for the effectiveness of embodied learning approaches in technical education.

The open-source nature of our implementation, with comprehensive documentation and modular design, enables broader adoption and community-driven enhancement. Educational institutions can adapt and extend the system according to their specific curricula and student needs, while researchers can build upon our foundation to explore advanced educational robotics concepts.

5.4 Future Trajectory

The foundation established by TinkerBlocks opens numerous avenues for future development. The modular architecture supports integration of additional sensors, more sophisticated AI-based learning assessment, and expanded programming constructs. Cloud-based deployment could enable remote learning scenarios, while IoT integration could connect multiple robots for collaborative programming exercises.

Advanced computer vision techniques could enhance block recognition accuracy and support more complex visual programming languages. Machine learning integration could provide personalized learning paths and intelligent tutoring capabilities, adapting to individual student progress and learning patterns.

5.5 Impact and Significance

TinkerBlocks demonstrates that educational technology can successfully combine pedagogical effectiveness with technical sophistication. By making programming concepts tangible and immediately rewarding, we contribute to addressing the growing need for computational literacy in modern education. The project provides a practical template for developing educational robotics systems that prioritize learning outcomes while maintaining technical rigor.

The comprehensive documentation, test coverage, and modular design ensure that TinkerBlocks serves not only as an educational tool but also as a reference implementation for similar projects. Our approach to integrating computer vision, robotics, and educational theory provides valuable insights for the broader educational technology community.

5.6 Final Reflection

The successful completion of TinkerBlocks validates the potential of physical programming interfaces to transform computer science education. By removing traditional barriers while maintaining conceptual depth, we have created a system that makes programming accessible, engaging, and immediately rewarding. The project stands as evidence that innovative approaches to educational technology can successfully bridge the gap between theoretical concepts and practical application, preparing students for an increasingly digital future while fostering creativity, logical thinking, and problem-solving skills.

Through TinkerBlocks, we have not merely built an educational tool, but demonstrated a pathway toward more inclusive, engaging, and effective programming education that honors both the complexity of computational thinking and the fundamental human need for tangible, meaningful learning experiences.

Appendix A

API DOCUMENTATION AND COMMAND REFERENCE

A.1 Arduino Serial API

The Arduino firmware exposes a comprehensive serial API for controlling the robotic car. All commands follow the format: `command:{"param1":"value1","param2":value2}`

A.1.1 Movement Commands

Move Command

```
// Move forward 20cm at speed 100
move:{"speed":100,"distance":20}

// Move backward for 1 second at speed 150
move:{"speed":-150,"timeMs":1000}

// Move 30cm in 2 seconds (speed calculated automatically)
move:{"distance":30,"timeMs":2000}

// Move without obstacle checking
move:{"speed":100,"distance":20,"checkUltrasonic":false}
```

Listing A.1: Move Command Examples

Parameters:

- `speed` (int): Motor speed (-255 to 255, negative for backward)
- `distance` (float): Distance in centimeters

- `timeMs` (unsigned long): Time in milliseconds
- `checkUltrasonic` (bool): Enable obstacle detection (default: true)
- `enableYawCorrection` (bool): Use gyro correction (default: true)

Success Response:

```
{
  "success": true,
  "success_result": "{\"distance_traveled\":20.00,\"
   time_taken\":784,\"final_yaw\":0.32}"
}
```

Failure Response:

```
{
  "success": false,
  "failure_reason": "Obstacle detected at 12.45cm"
}
```

Rotation Command

```
// Turn left 90 degrees
rotate:{"angle":90,"speed":100}

// Turn right 45 degrees
rotate:{"angle":-45,"speed":80}

// Rotate to absolute heading (North = 0 degrees)
rotate:{"angle":0,"speed":100,"absolute":true}
```

Listing A.2: Rotation Command Examples

Parameters:

- `angle` (float): Rotation angle in degrees (positive = left/CCW, negative = right/CW)
- `speed` (int): Rotation speed (1-255, default: 100)
- `absolute` (bool): Absolute vs relative rotation (default: false)

A.1.2 Sensor Commands

Ultrasonic Sensor

```
// Get distance reading
sensor:{"action":"distance"}

// Check for obstacles within 15cm
sensor:{"action":"obstacle","threshold":15}
```

Listing A.3: Sensor Command Examples

Gyroscope Commands

```
// Calibrate gyroscope
gyro:{"action":"calibrate"}

// Get current sensor data
gyro:{"action":"data"}

// Get current yaw angle
gyro:{"action":"yaw"}

// Set reference orientation
gyro:{"action":"reference"}
```

Listing A.4: Gyroscope Command Examples

A.1.3 Pen Control Commands

```
// Lift pen up
pen:{"action":"up"}

// Put pen down
pen:{"action":"down"}

// Set custom pen position
pen:{"action":"position","position":45}
```

Listing A.5: Pen Control Examples

A.2 ESP32 HTTP API

The ESP32 serves as a WiFi bridge, exposing HTTP endpoints that translate to Arduino serial commands.

A.2.1 HTTP Endpoints

```
# Movement control
POST /api/move
Content-Type: application/json
{"speed": 100, "distance": 20}

# Rotation control
POST /api/rotate
Content-Type: application/json
{"angle": 90, "speed": 100}

# Pen control
POST /api/pen
Content-Type: application/json
{"action": "up"}

# Sensor readings
POST /api/sensor
Content-Type: application/json
{"action": "distance"}

# Gyroscope control
POST /api/gyro
Content-Type: application/json
{"action": "calibrate"}

# IR sensor readings
POST /api/ir
Content-Type: application/json
{"action": "black_obstacle"}
```

Listing A.6: HTTP API Endpoints

A.3 WebSocket Communication Protocol

The Raspberry Pi control system uses WebSocket communication for real-time control and monitoring.

A.3.1 Client Commands

```
// Run complete OCR to engine pipeline
{
  "command": "run",
  "params": {
    "workflow": "full"
  }
}

// Run OCR only
{
  "command": "run",
  "params": {
    "workflow": "ocr_grid"
  }
}

// Run OCR with automatic engine execution
{
  "command": "run",
  "params": {
    "workflow": "ocr_grid",
    "chain_engine": true
  }
}

// Run engine with custom grid
{
  "command": "run",
  "params": {
    "workflow": "engine",
    "grid": [
      ["MOVE", "5"],
      ["TURN", "RIGHT"],
      ["LOOP", "3"],
      ["", "MOVE", "2"],
      ["", "TURN", "LEFT"]
    ]
```

```
  }
}

// Stop current process
{
  "command": "stop"
}
```

Listing A.7: WebSocket Client Commands

A.3.2 Server Responses

```
// Status update
{
  "type": "status",
  "message": "Processing image..."
}

// Progress update
{
  "type": "progress",
  "percentage": 45,
  "message": "Running OCR..."
}

// Completion notification
{
  "type": "complete",
  "result": "Success",
  "data": {
    "execution_time": 2.3,
    "commands_executed": 15
  }
}

// Error notification
{
  "type": "error",
  "message": "Block recognition failed",
```

```
  "details": "Insufficient lighting detected"
}
```

Listing A.8: WebSocket Server Responses

A.4 Programming Language Reference

A.4.1 Movement Commands

```
// Basic movement
MOVE                    // Move forward 1 unit
MOVE | 5                // Move forward 5 units
MOVE | -3               // Move backward 3 units
MOVE | X                // Move forward X units (variable)

// Conditional movement
MOVE | WHILE | X < 10  // Move while X is less than 10
MOVE | WHILE | DISTANCE > 30  // Move while distance > 30cm

// Turn commands
TURN | LEFT                      // Turn left 90 degrees
TURN | RIGHT                     // Turn right 90 degrees
TURN | 45                        // Turn right 45 degrees
TURN | LEFT | 30                 // Turn left 30 degrees
TURN | RIGHT | WHILE | X < 5     // Turn right while X < 5
```

Listing A.9: Movement Command Examples

A.4.2 Control Flow Examples

```
// Simple loop
LOOP | 5
    MOVE | 2
    TURN | RIGHT

// Conditional loop
LOOP | WHILE | X < 10
    MOVE | 1
    SET | X | X + 1

// Infinite loop with condition
LOOP | TRUE
    MOVE | 1
    IF | OBSTACLE
        TURN | RIGHT
```

```
16
17 // Conditional statements
18 IF | DISTANCE < 30
19     TURN | LEFT
20 ELSE
21     MOVE | 5
22
23 // Nested conditions
24 IF | X > 10
25     IF | Y < 5
26         MOVE | X
27     ELSE
28         TURN | RIGHT
29 ELSE
30     SET | X | 0
```

Listing A.10: Control Flow Examples

A.4.3 Variable Operations

```
1  // Basic assignment
2  SET | X | 5
3  SET | SPEED | 10
4  SET | Y | DISTANCE
5  SET | FLAG | TRUE
6
7  // Arithmetic operations
8  SET | Z | X + 2
9  SET | COUNT | COUNT + 1
10 SET | RESULT | X * Y / 2
11 SET | AVERAGE | (X + Y) / 2
12
13 // Boolean operations
14 SET | FOUND | DISTANCE < 30
15 SET | SAFE | NOT OBSTACLE
16 SET | READY | X > 0 AND Y > 0
17 SET | CONTINUE | DISTANCE > 10 OR COUNT < 5
```

Listing A.11: Variable Operation Examples

A.4.4 Sensor Integration

```
1  // Using sensor values in conditions
2  IF | DISTANCE < 20
3      TURN | RIGHT
4
```

```
IF | OBSTACLE
    MOVE | -2
    TURN | 180

IF | BLACK_DETECTED
    MOVE | 1
ELSE
    TURN | LEFT

// Sensor values in expressions
SET | SAFE_DISTANCE | DISTANCE + 5
SET | TOO_CLOSE | DISTANCE < 15

// Waiting for sensor conditions
WAIT | WHILE | DISTANCE < 20
WAIT | WHILE | BLACK_DETECTED
```

Listing A.12: Sensor Integration Examples

A.4.5 Drawing Commands

```
// Draw a square
SET | SIDE | 3
PEN_DOWN
LOOP | 4
    MOVE | SIDE
    TURN | RIGHT
PEN_UP

// Draw with conditional pen control
LOOP | 10
    IF | X % 2 = 0
        PEN_DOWN
    ELSE
        PEN_UP
    MOVE | 2
    SET | X | X + 1
```

Listing A.13: Drawing Command Examples

A.5 Error Codes and Troubleshooting

A.5.1 Common Error Codes

```
// Arduino Serial API Errors
```

```
"Invalid speed. Must be between 1 and 255."
"Obstacle detected at 12.45cm"
"Missing or invalid 'position' parameter for pen position"
"Unknown gyro action: invalid_action"
"Unknown sensor action: invalid_action"

// Vision Processing Errors
"Block recognition failed - insufficient lighting"
"Grid detection failed - corners not found"
"OCR processing timeout"
"Invalid grid dimensions"

// Engine Execution Errors
"Unknown command: INVALID_COMMAND"
"Maximum steps exceeded (1000)"
"Variable 'X' not defined"
"Division by zero in expression"
"ELSE without matching IF"
```

Listing A.14: Error Code Reference

A.5.2 Troubleshooting Guide

```
// Poor block recognition
Problem: Low recognition accuracy
Solution:
- Ensure lighting > 300 lux
- Check camera focus and positioning
- Verify block text clarity
- Clean camera lens

// Communication timeouts
Problem: API timeouts
Solution:
- Check WiFi connection strength
- Verify ESP32 power supply
- Restart system components
- Check serial cable connections

// Movement inaccuracy
Problem: Car movement imprecise
Solution:
- Calibrate gyroscope sensor
- Check battery voltage level
- Verify wheel alignment
- Test on smooth surface

```

```
// Program execution errors
Problem: Commands not executing
Solution:
- Verify grid syntax correctness
- Check indentation alignment
- Validate parameter ranges
- Review variable definitions
```

Listing A.15: Common Solutions

References

[1] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[2] H. Ishii, "Tangible user interfaces: past, present, and future directions," *Communications of the ACM*, vol. 51, no. 6, pp. 32–36, 2008.

[3] F. B. V. Benitti, "Exploring the educational potential of robotics in schools: A systematic review," *Computers & Education*, vol. 58, no. 3, pp. 978–988, 2012.

[4] S. Papert, "Mindstorms: Children, computers, and powerful ideas," 1980.

[5] A. Sullivan, E. R. Kazakoff, and M. U. Bers, "Robots in the classroom: Teaching young children to program with robotics," *Childhood Education*, vol. 89, no. 6, pp. 380–389, 2013.

[6] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Alphablocks: a tangible programming language for children," in *Proceedings of the 8th International Conference on Interaction Design and Children*, 2009, pp. 170–177.

[7] H. Suzuki and H. Kato, "Interaction design and children," *Communications of the ACM*, vol. 42, no. 5, pp. 39–41, 1999.

[8] E. R. Kazakoff, A. Sullivan, and M. U. Bers, "Putcode: Creating tangible programming blocks for young children," in *Proceedings of the 10th International Conference on Interaction Design and Children*, 2011, pp. 75–84.

[9] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: Programming for all," in *Communications of the ACM*, vol. 52, no. 11. ACM, 2009, pp. 60–67.

[10] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, pp. 1–15, 2010.