A Next.js Blockly-Based C++ Code Generator for a Custom Robotics Library

Leopold Kernegger
Department of Computer Science
TGM Vienna
Vienna, Austria
leopold.kernegger@tgm.ac.at

***Abstract*—This paper introduces a web-based visual programming platform that leverages Google's Blockly and the Next.js framework to generate structured C++ code for a modular robotics library. The system is designed to support rapid and reliable robotics development, especially in educational settings and competitions such as Botball at the Global Conference on Educational Robotics (GCER). Through visual abstraction and code synthesis, the platform lowers the barrier to entry for robotics programming while maintaining flexibility and performance for advanced users.**

***Index Terms*—Blockly, Next.js, C++, Robotics, Visual Programming, Code Generation, Embedded Systems**

I. Introduction

The growing popularity of robotics in education has attracted a diverse audience, including many with limited programming experience. Visual programming environments, such as Google's Blockly [1], are increasingly used to bridge this gap by offering block-based code construction. In competitive and educational contexts like Botball at GCER, participants are often tasked with rapidly prototyping robot behavior under time pressure. This demands intuitive yet powerful development tools.

We present a browser-based solution that combines Blockly with the Next.js frontend framework to generate syntactically correct and hardware-ready C++ code. The system integrates directly with a modular robotics library, enabling users to construct full robotics applications visually—without writing a single line of C++. This significantly lowers the entry barrier for beginners [2], [3] while providing a foundation that remains usable for advanced users seeking rapid prototyping capabilities.

II. System Architecture

The application is composed of three tightly integrated subsystems that together enable block-based robotics development:

A. Blockly Workspace

The Blockly workspace serves as the visual layer, providing a dynamic block editor for composing logic. We define custom blocks for robotics-specific tasks such as sensor registration, motor control, conditional logic, and task sequencing. Each block includes metadata (e.g., allowed connections, default values, color) to assist users in building valid configurations.

B. Next.js Frontend

The frontend is implemented using Next.js with TypeScript, taking advantage of server-side rendering for rapid load times and smooth user interaction. The workspace is hosted in a React component, allowing us to bind the Blockly API with custom hooks for code generation, workspace management, and exporting generated C++ files. The layout separates editing, preview, and export functions into modular panes.

C. C++ Backend Library

The robotics library provides the code target for all Blockly-generated output. It exposes high-level abstractions for motors, sensors, task scheduling, and lifecycle handling. Internally, the library is organized into subsystems: SensorRegistry, ImprovedMotor, MotorController, and LineManager. These are initialized via a central Lifecycle class, which manages setup, execution, and teardown of the robot's behavior [4].

III. Blockly Block Design and Motivation

The Blockly-based visual interface plays a critical role in enabling users—particularly those without extensive C++ knowledge—to construct robotics programs through intuitive drag-and-drop blocks. This section explains the purpose and rationale behind each major block category, including integration into the code generation pipeline and interaction with our robotics library.

A. Blockly Integration Overview

As defined in `cpprunner.tsx`, the Blockly workspace is initialized dynamically within a React component using the official Blockly API. The `toolbox` provides domain-specific blocks grouped into categories such as Program, Classes, Methods, Sensors, Tasks, Control, Logic, and Math. Every change in the workspace triggers the `cppGenerator`, which converts the visual representation into valid C++ code. Users can view and run this code directly through a backend API.

B. Block Categories and Purpose

a) Sensor Blocks: Sensor blocks like `register sensor name` and `read-digital` allow the user to register and access named sensor values. The dropdowns in the read blocks are populated dynamically based on other blocks in the workspace, ensuring consistency between defined and accessed sensors (see Fig. 1).

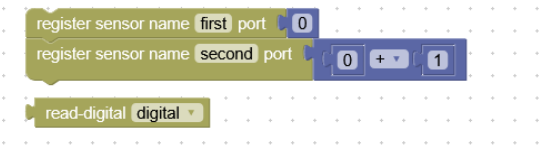

Fig. 1. Sensor registration and value reading blocks.

b) Main and Class Blocks: The `cpp_main` block (Fig. 3) wraps the program's entry point and includes an input for sequential statements. The `cpp_class` block lets users define custom classes that extend existing ones like `Lifecycle`. This visual abstraction simplifies extending base logic and adding methods such as `calibrate` or `wait` via override blocks.

Fig. 2. Main program block.

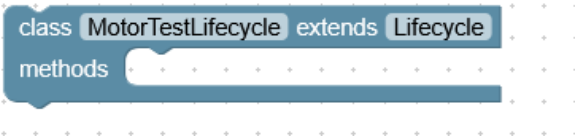

Fig. 3. Lifecycle class definition block.

c) Task Blocks: Specialized blocks such as `move-up task` or `move task speed` directly map to the lifecycle task system described earlier. They create objects like `MovingTask` or `MovingUpTask` and add them to the lifecycle's execution queue with user-defined labels and parameters.

d) Logic and Control: Logic (`if`, `and/or`, `==`) and math blocks (`+`, `modulo`, etc.) enable sensor-driven and conditional behavior. Combined with looping constructs (`repeat`), users can create sophisticated control logic graphically.

C. Why Blockly?

Blockly enables:

- **Semantic Constraints**: Users can't misconnect blocks that don't make sense [5].
- **Dynamic Linking**: Sensor blocks are context-aware (e.g., dropdowns show existing names only).
- **Clear Visualization**: Beginners understand flow and logic via spatial layout.
- **C++ Fidelity**: All blocks generate C++ that integrates directly with our backend.

D. Custom Block Logic

All blocks are registered and translated in `customBlocks.ts`. The system uses `defineBlocksWithJsonArray` for structural definition and attaches a C++ code generator to each block via `cppGenerator.forBlock`. The generator supports precedence and expression handling, including fallbacks for unhandled types. This design supports future extensibility [6].

E. Execution and Feedback Loop

The `Run C++` button compiles the generated code via a backend service and prints console output in real time, reinforcing the correctness of block configurations. This immediate feedback helps both in debugging and in learning C++ semantics by example [7].

F. Summary

The visual block system provides a robust layer atop our robotics backend, abstracting low-level details while preserving expressiveness. By visually guiding users through robot logic and translating it to compilable code, Blockly bridges the gap between beginner usability and expert control.

IV. Robotics Library Architecture

The library is organized into independent modules, each handling a specific concern:

A. Communication

A UDP broadcast system in heartbeat.cpp sends and receives robot state in JSON format, enabling coordination and telemetry in multi-robot systems.

B. Motor Control

Motor control modules include:

- **SolarboticMotorController**: Supports simple differential drive with PID-controlled turning.
- **MecanumMotorController**: Enables omnidirectional movement using trigonometric vector logic.

Both controllers inherit from a shared base and are accessed via a singleton MotorController.

C. Motion Abstractions

- ImprovedMotor: Threaded motor control with multiple modes (velocity, PID, positional).
- ImprovedServo: Uses degree-to-tick mapping and easing interpolation for smooth actuation.

D. Sensing and Calibration

The SensorRegistry manages dynamic sensor registration and calibration. Analog sensors are median-filtered for robustness [8]. Light sensors and gyro correction are handled through additional modules like until_light.cpp.

E. Navigation

- LineManager: Provides proportional line-following and crossing detection.
- Playground: Encodes a node graph with Dijkstra-based routing [9].
- SquareUp: Aligns the robot using wall or line-based feedback.

F. Utility Modules

Includes:

- Easing profiles (linear, exponential, cubic).
- Spatial math using Vector2D/Vector3D.
- terminate.cpp: watchdog failsafe.

G. Lifecycle Control

The Lifecycle class defines structured phases: calibrate, wait, execute_tasks, clean, reset. Each task implements an `execute()` method. Retry logic ensures fault-tolerant execution. PID control is implemented using standard approaches.

V. Conclusion

This paper presented a modular visual programming system for robotics development. Using Blockly and Next.js, students can visually compose robot behaviors that generate efficient, maintainable C++ code. The system's backend is flexible, extensible, and aligned with educational robotics challenges such as Botball.

Acknowledgment

We thank the open-source developers behind Blockly, Next.js, and related tools. Special acknowledgment goes to the TGM faculty and Botball participants for feedback during testing.

References

[1] R. Kupisch, S. März, and S. Orlowski, "Review of google blockly and its innovative use," *International Journal of Computer Science and Engineering*, vol. 8, no. 4, pp. 123–130, 2018.

[2] D. Zhang, J. Smith, A. Lee, and R. Patel, "The impact of robotics on stem education: Facilitating cognitive and interdisciplinary advancements," *International Journal of STEM Education*, vol. 10, no. 2, pp. 1–15, 2023.

[3] C. Pimmer, M. Mateescu, and U. Gröhbiel, "A systematic review of studies on educational robotics," *Journal of Peer Learning*, vol. 9, no. 2, pp. 32–54, 2020.

[4] J. Brender, M. Thémines, and P. Lucas, "Investigating the role of educational robotics in formal mathematics education: the case of geometry for 15-year-old students," *arXiv preprint arXiv:2106.10925*, 2021.

[5] O. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: Blocks and beyond," *arXiv preprint arXiv:1705.09413*, 2017.

[6] C. S. Cheah, "Factors contributing to the difficulties in teaching and learning of computer programming: A literature review," *Contemporary Educational Technology*, vol. 12, no. 2, pp. 91–114, 2020.

[7] M. Karaca and U. Yayan, "Ros based visual programming tool for mobile robot education and applications," *arXiv preprint arXiv:2011.13706*, 2020.

[8] Wikipedia contributors. (2025) Median filter. Accessed: 2025-06-15. [Online]. Available: https://en.wikipedia.org/wiki/Median$_filter$

[9] ——. (2025) Dijkstra's algorithm. Accessed: 2025-06-15. [Online]. Available: