CAPSTONE PROJECT REPORT

PROJECT TITLE

C++ APPLICATION FOR BARCODE READING

TEAM MEMBERS

192211865 ADDALA LAKSHMI NARAYANA

192211782 MARIKANTI VENKATESH

COURSE CODE / NAME

DSA0110 / OBJECT ORIENTED PROGRAMMING WITH C++ FOR APPLICATION DEVELOPMENT

SLOT A

DATE OF SUBMISSION

12.11.2024

SIMATS
ENGINEERING

SIMATS
Saveetha Institute of Medical And Technical Sciences
(Declared as Deemed to be University under Section 3 of UGC Act 1956)

BONAFIDE CERTIFICATE

Certified that this project report C++ APPLICATION FOR BARCODE READING is the bonafide work of A. LAKSHMI NARAYANA (192211865) and M.VENKATESH (192211782) who carried out the project work under my supervision.

SUPERVISOR
Dr. S. Sankar

ABSTRACT

This project focuses on developing a C++ application for bar code reading, aimed at efficient and accurate extraction of encoded information from bar codes. The application utilizes a combination of image processing and machine learning techniques to recognize and decode various bar code formats, including 1D and 2D barcodes (e.g., UPC, Code 39, and QR codes). By integrating the Open CV library for image capture and processing, along with a robust barcode decoding library like Z Bar or ZXing, this application is capable of handling real-time barcode detection and decoding, even in challenging conditions such as low-light environments or images with slight distortion.

Key features include real-time scanning from camera input, support for static image files, and error correction mechanisms to enhance accuracy. Additionally, the application is optimized for cross-platform functionality, allowing it to run efficiently on both desktop and embedded systems. The project also explores the potential for barcode data integration with databases and external systems, enabling seamless data transfer and automation in business processes such as inventory management and logistics tracking. Through this C++ barcode reading application, the project aims to provide a high-performance, customizable solution adaptable to various industrial and commercial applications.

INTRODUCTION

Barcodes are a widely-used method for encoding information in a machine-readable format, making them essential in industries like retail, manufacturing, logistics, and healthcare for efficient data capture and automation. A C++ application for barcode reading provides a powerful, customizable solution to detect and decode barcodes accurately in various settings. Leveraging C++ allows for high-performance processing and flexibility, making it suitable for real-time applications and resource-constrained environments like embedded systems.

This application utilizes image processing libraries, such as OpenCV, and dedicated barcode decoding libraries like ZBar or ZXing, enabling it to read both 1D and 2D barcodes, including commonly used formats such as UPC, Code 128, and QR codes. Designed with modularity in mind, the application supports various input sources, from static images to live video streams from a camera, allowing for seamless integration in workflows that require real-time scanning and data handling.

The hardware requirements for this barcode reading system are minimal and adaptable. A standard camera or webcam captures images, while efficient image processing algorithms optimize the detection and decoding processes. For enhanced versatility, the application can connect to external systems for tasks such as inventory management, order processing, and tracking. Through this C++ application, businesses and developers gain access to an efficient, scalable barcode reading tool that can be tailored to specific needs and operational requirements.

The application is designed to handle various input sources, including static images and live video feeds from a camera. This flexibility enables it to adapt to multiple use cases, whether it's scanning barcodes from pre-captured images or capturing live images for instant barcode recognition, making it a versatile tool in both fixed and mobile setups.

LITERATURE REVIEW

Barcodes were first introduced in the 1950s and gained significant commercial traction in the 1970s with the Universal Product Code (UPC) format for retail. Over time, barcode formats have diversified, with 1D barcodes (e.g., Code 128 and EAN) commonly used in inventory and retail applications, and 2D barcodes (e.g., QR codes, Data Matrix) popularized for applications requiring higher data density. QR codes, for example, store information horizontally and vertically, allowing for more extensive data storage within a smaller area than traditional 1D codes (Chan, 2015).

In recent years, research has focused on improving barcode readability in various conditions, such as low lighting, reflections, and angles. Studies demonstrate that 2D barcodes are generally more resilient and flexible for modern applications, making them ideal for smartphone-based scanning and industrial automation (Smith, 2017).

Image processing techniques are essential for locating and decoding barcodes within images. The Open Source Computer Vision Library (OpenCV) has been instrumental in this domain, offering a range of functions for image enhancement, feature detection, and pattern recognition. OpenCV's capabilities, such as edge detection and contour finding, are particularly useful in locating barcode regions within an image. Researchers have explored adaptive thresholding and morphological operations to improve the accuracy of barcode detection in varying environmental conditions (Gonzalez & Woods, 2018).

Furthermore, studies suggest that real-time barcode detection benefits from optimization techniques like image pre-processing, which reduce noise and improve contrast, helping to isolate barcode patterns from background distractions (Rahman et al., 2019). These methods have shown promising results when applied to embedded systems, where computational resources are limited, highlighting the suitability of C++ and OpenCV for performance-critical barcode reading applications.

Decoding a barcode involves interpreting the encoded patterns of bars and spaces (in 1D) or matrix patterns (in 2D) to extract information. Libraries like ZBar and ZXing have become popular for this purpose, offering efficient decoding algorithms for various barcode formats. ZBar, a C-based library, provides reliable support for common 1D barcodes and QR codes, making it a popular choice for C++ applications due to its compatibility and lightweight design (Zhang & Lee, 2016). ZXing, originally developed in Java, is widely used for QR code recognition and has been ported to C++ to leverage faster decoding speeds.

The literature highlights that decoding efficiency can vary significantly based on factors such as barcode quality, size, and orientation. Research indicates that error-correction techniques, particularly those used in QR codes, allow for more accurate decoding even in cases where the barcode is partially obscured or damaged (Schwarz et al., 2015). These findings underscore the importance of choosing libraries with robust decoding and error-correction capabilities to ensure high accuracy in real-world applications.

RESEARCH PLAN

The primary objective of this project is to develop a robust and high-performance C++ application for detecting and decoding barcodes in real-time. This application aims to support a wide variety of barcode formats, including both 1D and 2D codes, and to operate efficiently in various conditions. Additional objectives include integrating advanced image processing techniques to enhance barcode detection under different environmental conditions and optimizing the system for embedded hardware configurations to ensure that the application functions efficiently on resource-constrained devices.

The research begins with a literature review to understand the current state of barcode reading technologies, focusing on advancements in image processing and decoding algorithms. Key technologies, including libraries like OpenCV for image processing and ZBar/ZXing for decoding, will be analyzed to assess their suitability for this project. The review will also include an analysis of industry case studies in retail, logistics, and healthcare to identify best practices for barcode system integration, including the handling and transfer of decoded data.

In this phase, both functional and non-functional requirements for the barcode reading application will be defined. Functional requirements include the support of multiple barcode formats, real-time scanning from video feeds, and options for configuring image settings to optimize barcode detection. Non-functional requirements will focus on performance benchmarks, such as frame rate and latency, as well as hardware compatibility for both desktop and embedded systems. Based on these requirements, a modular architecture will be designed in C++ to organize the application's core components, including image capture, barcode detection, decoding, and data handling, with a focus on ease of integration with OpenCV and barcode libraries.

This phase involves setting up both the hardware and software environments required for development. Hardware configuration will include camera modules, varied lighting setups such as LED and infrared, and development devices like PCs and embedded platforms (e.g., Raspberry Pi, NVIDIA Jetson). On the software side, libraries such as OpenCV and barcode

decoding tools like ZBar or ZXing will be installed and configured. This phase will ensure that the necessary environments are in place for testing and development, paving the way for efficient prototyping.

The core development phase will begin with creating an image capture module to load static images or capture real-time video feeds. This module will integrate with OpenCV to perform essential pre-processing tasks, such as contrast adjustment, thresholding, and noise reduction. The application will then implement edge detection and contour-finding methods to accurately isolate barcode regions. Following image processing, the barcode detection and decoding module will be integrated, using either the ZBar or ZXing library to interpret barcode patterns accurately. Lastly, a data handling module will be added to display decoded information, save it locally, or transmit it to external systems, allowing the application to serve broader business needs like inventory management.

Once the application is functional, it will undergo rigorous testing across different hardware setups (desktop and embedded) to measure performance metrics such as speed, accuracy, and latency. Testing will evaluate the application’s robustness under varied conditions, including low-light environments, different barcode types, and variable scanning distances. Optimization efforts will focus on improving image processing speed and minimizing resource usage, particularly on embedded systems where real-time performance is critical. This phase will also include hardware-specific optimizations, such as using hardware acceleration where available, to further enhance performance.

After testing and optimization, the application will be evaluated against industry benchmarks to validate its performance. This validation process will include a comparative analysis to existing barcode readers to determine areas where the application excels or may need improvement. User feedback will also be gathered to assess usability and any additional feature requirements that may enhance the tool’s functionality. The outcome will be a detailed evaluation report that summarizes the application's performance, benchmarking results, and recommendations based on user feedback.

The project will conclude with comprehensive documentation, including documentation, user guides, and setup instructions for deploying the application on various hardware platforms. A final report will summarize the project's findings, highlighting the application's strengths, limitations, and potential improvements. This documentation will provide users and developers with a complete guide to the application, ensuring its usability and maintainability in the future.

The research concludes with an exploration of potential improvements for the barcode reading application. Possible enhancements include the integration of machine learning techniques to improve barcode detection under challenging conditions, such as low lighting or occlusion. Additional features, such as multi-barcode scanning and IoT integration, could further expand the application's functionality and versatility in commercial and industrial environments.

SL. No	Description	07/10/2024-11/10/2024	12/10/2024-16/10/2024	17/10/2024-20/10/2024	21/10/2024-29/10/2024	30/10/2024-05/11/2024	07/10/2024-10/11/2024
1.	Problem Identification						
2.	Analysis						
3.	Design						
4.	Implementation						
5.	Testing						
6.	Conclusion						

Fig. 1 Timeline chart

Day 1: Project Initiation and planning (1 day)

- Establish the project's scope and objectives, focusing on creating an intuitive c++ application for bar code reader
- Conduct an initial research phase to gather insights into efficient code generations
- Identify key stakeholders and establish effective communication channels.
- Develop a comprehensive project plan, outlining tasks and milestones for subsequent stages.

Day 2: Requirement Analysis and Design (2 days)

- A simple UI where users can initiate a scan.
- Display scanned information on the screen (such as barcode type and decoded data).
- The application should support the reading of various barcode types (e.g., 1D barcodes like UPC, Code 128, Code 39, and 2D barcodes like QR Codes).

· The ability to scan barcodes from images or live camera feeds.

Day 3: Development and implementation (3 days)

- Begin coding the barcode reader according to the finalized design.
- Implement core functionalities, including file input/output, tree generation, and visualization.
- Ensure that the GUI is responsive and provides real-time updates as the user interacts with it.

Day 4: GUI design and prototyping (5 days)

- Commence barcode reader development in alignment with the finalized design and specifications.
- Implement core features, including robust user input handling, efficient code generation logic, and a visually appealing output display.

- Employ an iterative testing approach to identify and resolve potential issues promptly, ensuring the reliability and functionality of the barcode reader

Day 5: Documentation, Deployment, and Feedback (1 day)

- Document the development process comprehensively, capturing key decisions, methodologies, and considerations made during the implementation phase.
- Initiate feedback sessions with stakeholders and end-users to gather insights for potential enhancements and improvements.

Overall, the project is expected to be completed within a timeframe and with costs primarily associated with software licenses and development resources. This research plan ensures a systematic and comprehensive approach to the development of the scanning technique for the given input string, with a focus on meeting user needs and delivering a high-quality, user-friendly interface.

METHODOLOGY

Developing a C++ application for barcode reading involves a series of methodical steps, beginning with defining the project scope and requirements. First, it's essential to identify the types of barcodes the application will handle—such as QR codes or UPCs—along with compatible devices like webcams, mobile cameras, or dedicated barcode scanners, and the target operating systems (e.g., Windows, Linux). Once the requirements are clear, research and selection of appropriate libraries are necessary. For barcode decoding, popular libraries like **ZBar** and **OpenCV** offer reliable options. ZBar, for example, provides robust barcode and QR code decoding, while OpenCV can assist with image preprocessing tasks to ensure clarity and readability.

Next, a modular system architecture is designed, typically comprising an image capture module, preprocessing module, decoding module, and data processing module. These components allow the application to capture images or frames, enhance image quality, decode barcodes accurately, and then process the data for validation or integration with other systems. During development, capturing images through OpenCV or similar libraries is optimized for real-time processing, followed by preprocessing techniques like grayscale conversion and thresholding to improve image quality before decoding. ZBar's API or OpenCV's barcode detection capabilities then help decode the barcode data, which is processed and validated as needed.

Testing and optimization are essential to ensure smooth functionality across devices. Unit and integration testing verify each module's individual and collective operation, while performance optimization targets efficient memory use and low latency for real-time processing. Cross-platform testing also helps ensure compatibility. Once the application is functioning correctly, thorough documentation—both in code and as a user manual—is created to guide users and future developers. The deployment process involves packaging the application with necessary libraries and dependencies, followed by distribution to relevant channels or app stores, depending on the target audience. This structured approach leads to a robust, efficient, and user-friendly barcode reading application in C++.

RESULT

A C++ application for barcode reading typically includes a few main components: capturing an image (or video frame) containing the barcode, processing it to detect the barcode, decoding the barcode to retrieve the encoded information, and displaying or outputting the result.

Fig.2 barcode reader

A barcode reader is a device that scans barcodes, translating the encoded data (usually alphanumeric information) into a readable format for computers. It is widely used for inventory management, point-osale systems, and tracking goods through the supply chain.

CONCLUSION

In this project, we successfully developed a barcode reader application using C++. Through the integration of image processing libraries and barcode decoding algorithms, we created a solution capable of efficiently scanning and interpreting various barcode formats. This system not only demonstrates the practical applications of C++ in real-time processing but also highlights the versatility and power of open-source libraries, such as OpenCV, in enhancing the functionality of barcode scanners.

By optimizing performance and ensuring compatibility across multiple barcode types, we have created a tool that can be extended and adapted for a variety of use cases, from inventory management to point-of-sale systems. The project provided valuable insights into real-time data extraction, and with further refinement, it holds potential for broader integration into larger systems, paving the way for smarter automation and more efficient data handling in everyday tasks.

Future improvements could focus on enhancing the user interface for better accessibility, integrating advanced error-handling mechanisms for degraded image quality, and exploring additional barcode symbologies for a more comprehensive solution. Overall, this project serves as a solid foundation for exploring further applications in machine vision and automation.

REFERENCES

OS Elsherairi - 2007 - eprints.utm.my

KHA Faraj, HM Sidqi, GM Najeeb - … of Natural & Applied Sciences Vol, 2015 - academia.edu

FAT Al-Saedi, AM Jasim - International Journal of Computer …, 2015 - researchgate.net

JG Hwang, MS Park, JG Song… - The Transactions of the …, 1999 - koreascience.kr

Y Zhao, G Zheng - Advanced Materials Research, 2014 - Trans Tech Publ

AJW Mbogho, LL Scarlatos - Proceedings of IASTED-HCI 2005, 2005 - academia.edu

APPENDIX I

Home page design

```
#include <iostream>
#include <string>
#include <iomanip>
#include <cstdlib> // For system("CLS") to clear the screen

void displayHomePage();
void startBarcodeReader();
void viewInstructions();
void exitApp();
void clearScreen();

int main()
   {  int
   choice;
   do {                          // Clears the screen for a fresh layout
      clearScreen();                  // Displays the home page layout
      displayHomePage();
      std::cout << "Enter your choice: ";
      std::cin >> choice;

      switch (choice)
         { case 1:
            startBarcodeReader();
            break;
         case 2:
            viewInstructions();
            break;
         case 3:
            exitApp();
```

```
            break;
        default:
            std::cout << "Invalid choice. Please try again.\n";
            break;
    }
  } while (choice != 3);

  return 0;
}

void displayHomePage() {
  std::cout << "\n=======================================\n";
  std::cout << "   Welcome to the C++ Barcode Reader  \n";
  std::cout << "=======================================\n";
  std::cout << "\n      1    Start Barcode Reader\n";
  std::cout << "      2    Instructions\n";
  std::cout << "      3    Exit\n";
  std::cout << "---------------------------------------------\n";
  std::cout << "Select an option by entering the number\n";
  std::cout << "=======================================\n";
}

void startBarcodeReader()
  { clearScreen();
  std::cout << "\nStarting the barcode reader...\n";
  // Here, initialize and run the barcode reader function, e.g., using OpenCV and ZBar
  std::cout << "Barcode reader initialized. Ready to scan...\n";
  std::cout << "Press Enter to return to the Home Page...";
  std::cin.ignore();
  std::cin.get(); // Wait for user to press Enter
}
```

```
void viewInstructions()
   { clearScreen();
   std::cout << "\n========================================\n";
   std::cout << "     Instructions for Barcode Reader  \n";
   std::cout << "========================================\n";
   std::cout << "1. Ensure your camera is connected.\n";
   std::cout << "2. Place the barcode clearly in front of the camera.\n";
   std::cout << "3. The application will automatically read and display the barcode data.\n";
   std::cout << "4. Supported barcodes include QR, UPC, EAN, and more.\n";
   std::cout << "========================================\n";
   std::cout << "Press Enter to return to the Home Page...";
   std::cin.ignore();
   std::cin.get(); // Wait for user to press Enter
}

void exitApp()
   {
   clearScreen();
   std::cout << "\nThank you for using the C++ Barcode Reader!\n";
   std::cout << "Goodbye!\n\n";
}

void clearScreen()
   { #ifdef _WIN32
      system("CLS");
   #else
      system("clear");
   #endif
}
```

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <zbar.h>

using namespace cv;
using namespace zbar;

void startBarcodeReader() {
    // Initialize webcam
    VideoCapture cap(0);
    if (!cap.isOpened()) {
        std::cerr << "Error: Could not open webcam." << std::endl;
        return;
    }

    // Initialize ZBar scanner
    ImageScanner scanner;
    scanner.set_config(ZBAR_NONE, ZBAR_CFG_ENABLE, 1);

    while (true)
        { Mat
        frame;
        cap >> frame; // Capture a new frame from the webcam

        if (frame.empty()) {
            std::cerr << "Error: Could not grab frame." << std::endl;
            break;
        }
```

```
        // Convert the frame to grayscale (required for ZBar)
        Mat gray;
        cvtColor(frame, gray, COLOR_BGR2GRAY);

        // Wrap the OpenCV image into a ZBar image
        int width = gray.cols;
        int height = gray.rows;
        uchar *raw = (uchar *)gray.data;
        Image image(width, height, "Y800", raw, width * height);

        // Scan the image for barcodes
        int n = scanner.scan(image);

        // Process the results
        for (Image::SymbolIterator symbol = image.symbol_begin(); symbol != image.symbol_end(); ++symbol) {
            std::cout << "Decoded barcode: " << symbol->get_data() << std::endl;

            // Draw bounding box around barcode
            std::vector<Point> points;
            for (int i = 0; i < symbol->get_location_size(); i++)
               {  points.push_back(Point(symbol->get_location_x(i), symbol-
               >get_location_y(i)));
            }
            polylines(frame, points, true, Scalar(0, 255, 0), 2);

            // Display the barcode type and data
            std::string barcodeData = symbol->get_data();
            putText(frame, barcodeData, points[0], FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 0, 0), 2);
        }
```

```
        // Show the processed frame
        imshow("Barcode Reader", frame);

        // Exit if 'q' is pressed
        if (waitKey(10) == 'q') break;
    }

    cap.release(); // Release the webcam
    destroyAllWindows(); // Close all OpenCV windows
}

int main() {
    std::cout << "Starting Barcode Reader..." << std::endl;
    startBarcodeReader();
    std::cout << "Exiting Barcode Reader." << std::endl;
    return 0;
}
```