

code to parse qr code using zxing-cpp with best preprocessing and without opencv

Certainly! To parse QR codes using the ZXing-C++ library without relying on OpenCV, you typically need to:

1. Load the image.
2. Preprocess the image (adjust brightness/contrast, apply thresholds, etc.).
3. Decode the QR code.

Below is a general structure in C++ using ZXing-C++ library for QR code parsing.

First, ensure you have ZXing-C++ installed. You can find the instructions on their [GitHub page](#).

Here's a simple example code:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <zxing/common/Counted.h>
#include <zxing/Bitmap.h>
#include <zxing/DecodeHints.h>
#include <zxing/MultiFormatReader.h>
#include <zxing/common/GlobalHistogramBinarizer.h>
#include <zxing/Exception.h>
#include <zxing/ReaderException.h>

using namespace std;
using namespace zxing;
using namespace zxing::multi;

vector<char> readFile(const string& filePath) {
    ifstream file(filePath, ios::binary);
    return vector<char>((istreambuf_iterator<char>(file)),
        istreambuf_iterator<char>());
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " <path-to-qr-code-image>" << endl;
    }
}
```

```

        return 1;
    }

    string filePath = argv[1];
    vector<char> imgData = readFile(filePath);
    if (imgData.empty()) {
        cerr << "Error reading file: " << filePath << endl;
        return 1;
    }

    try {
        Ref<LuminanceSource> source(new GreyscaleLuminanceSource(
            reinterpret_cast<const unsigned char*>(&imgData[0]), imgData.size(),
            imgData.size(), 1));

        Ref<Binarizer> binarizer(new GlobalHistogramBinarizer(source));
        Ref<BinaryBitmap> binary(new BinaryBitmap(binarizer));

        DecodeHints hints(DecodeHints::DEFAULT_HINT);
        MultiFormatReader reader;

        Ref<Result> result(reader.decode(binary, hints));
        cout << "QR Code Text: " << result->getText()->getText() << endl;
    } catch (const ReaderException& e) {
        cerr << "Failed to decode QR code: " << e.what() << endl;
    } catch (const zxing::Exception& e) {
        cerr << "ZXing error: " << e.what() << endl;
    } catch (const exception& e) {
        cerr << "Error: " << e.what() << endl;
    }

    return 0;
}

```

C++

Copy code

**Key Points:**

1. **Reading Image:** The `readFile` function reads the entire image into memory. You might want to implement your own image loading mechanism suited for your needs.
2. **GreyscaleLuminanceSource:** Being used here to convert the image to greyscale. Ensure the image is in the correct format.
3. **GlobalHistogramBinarizer:** Used to binarize the greyscale image.
4. **MultiFormatReader:** Reader to decode the QR code from the binary image.
5. **Error Handling:** Basic error handling is done in the catch blocks.

This example lacks sophisticated image preprocessing. Depending on the image quality, you might need custom image operations (e.g., resizing, noise reduction), and you could add your own preprocessing steps before creating `LuminanceSource`.

Would you like to dive deeper into specific preprocessing techniques or further details on ZXing-C++? Let me know!

Copy

Good response

Bad response

Yes. please let me dive further details

Certainly! Let's go a bit deeper into specific preprocessing techniques and optimizations for better QR code parsing. I'll break it down into manageable sections.

## 1. Image Loading:

First, you'll need to load the image. Without using OpenCV, libraries like `STB Image` or `libpng` could be useful for handling image files. Here's an example using `STB Image`:

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

#include <iostream>
#include <vector>
#include <zxing/common/Counted.h>
#include <zxing/LuminanceSource.h>
#include <zxing/BinaryBitmap.h>
#include <zxing/DecodeHints.h>
#include <zxing/MultiFormatReader.h>
#include <zxing/common/GlobalHistogramBinarizer.h>
#include <zxing/Exception.h>
```

```

#include <zxing/ReaderException.h>
#include <zxing/Result.h>

using namespace std;
using namespace zxing;
using namespace zxing::multi;

vector<unsigned char> preprocessImage(const unsigned char* imageData, int width,
int height, int channels) {
    vector<unsigned char> greyData(width * height);
    for (int i = 0; i < width * height; ++i) {
        int offset = i * channels;
        // Simple grayscale conversion
        greyData[i] = static_cast<unsigned char>(0.299f * imageData[offset] +
                                                    0.587f * imageData[offset + 1] +
                                                    0.114f * imageData[offset + 2]);
    }
    return greyData;
}

class CustomLuminanceSource : public LuminanceSource {
private:
    vector<unsigned char> luminances_;
    int width_;
    int height_;

public:
    CustomLuminanceSource(const unsigned char* data, int width, int height)
        : LuminanceSource(width, height), luminances_(data, data + width*height),
width_(width), height_(height) {}

    ArrayRef<char> getRow(int y, ArrayRef<char> row) const override {
        if (row->size() != width_) {
            row.reset(ArrayRef<char>(width_));
        }
        memcpy(&row[0], &luminances_[y * width_], width_);
        return row;
    }

```

```

    }

    ArrayRef<char> getMatrix() const override {
        ArrayRef<char> matrix(width_ * height_);
        memcpy(&matrix[0], &luminances_[0], width_ * height_);
        return matrix;
    }
};

int main(int argc, char *argv[]) {
    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " <path-to-qr-code-image>" << endl;
        return 1;
    }

    int width, height, channels;
    unsigned char* imageData = stbi_load(argv[1], &width, &height, &channels, 0);
    if (imageData == nullptr) {
        cerr << "Error loading image: " << argv[1] << endl;
        return 1;
    }

    vector<unsigned char> greyData = preprocessImage(imageData, width, height,
channels);
    stbi_image_free(imageData); // Don't forget to free the image memory

    try {
        Ref<LuminanceSource> source(new CustomLuminanceSource(&greyData[0], width,
height));
        Ref<Binarizer> binarizer(new GlobalHistogramBinarizer(source));
        Ref<BinaryBitmap> binary(new BinaryBitmap(binarizer));

        DecodeHints hints(DecodeHints::DEFAULT_HINT);
        MultiFormatReader reader;

        Ref<Result> result(reader.decode(binary, hints));
        cout << "QR Code Text: " << result->getText()->getText() << endl;
    }
}

```

```

    } catch (const ReaderException& e) {
        cerr << "Failed to decode QR code: " << e.what() << endl;
    } catch (const zxing::Exception& e) {
        cerr << "ZXing error: " << e.what() << endl;
    } catch (const exception& e) {
        cerr << "Error: " << e.what() << endl;
    }

    return 0;
}

```

C++

Copy code

## 2. Preprocessing:

Preprocessing can dramatically improve QR code recognition, especially on noisy images. Here are some steps:

- **Grayscale Conversion:** Convert your image to grayscale as shown in `preprocessImage`. It simplifies further processing.
- **Thresholding/Binarization:** Convert the grayscale image to a binary image using techniques like Otsu's method or adaptive thresholding.

## 3. Custom LuminanceSource:

For better control, I've created a custom implementation of `LuminanceSource`. This allows us to use our preprocessed image data

Copy

Good response

Bad response

continue

Certainly! Let's continue from where we left off and delve into more preprocessing details and how to implement them in your C++ code.

## Further Preprocessing Techniques:

1. **Thresholding:** This can help in converting a grayscale image to a binary image, which is often required for better QR code detection. While the `GlobalHistogramBinarizer` does some of this for you, you might want to consider custom solutions if it's not performing well enough.

2. **Noise Reduction:** Apply noise reduction algorithms like median filtering to remove small artifacts in the image which might interfere with QR code detection.

Here's an example of how you might implement thresholding and noise reduction:

```
#include <numeric>
#include <algorithm>

// Median Filter for noise reduction
void medianFilter(vector<unsigned char>& data, int width, int height) {
    vector<unsigned char> filteredData = data;
    const int windowSize = 3;
    const int kernelSize = windowSize * windowSize;
    const int edge = windowSize / 2;

    for (int y = edge; y < height - edge; ++y) {
        for (int x = edge; x < width - edge; ++x) {
            vector<unsigned char> window;
            for (int ky = 0; ky < windowSize; ++ky) {
                for (int kx = 0; kx < windowSize; ++kx) {
                    window.push_back(data[(y + ky - edge) * width + (x + kx - edge)]);
                }
            }
            std::nth_element(window.begin(), window.begin() + kernelSize / 2, window.end());
            filteredData[y * width + x] = window[kernelSize / 2];
        }
    }
    data = filteredData;
}

// Otsu's Thresholding
unsigned char otsuThreshold(const vector<unsigned char>& data, int width, int height) {
    int hist[256] = {0};
    for (unsigned char pixel : data) {
        hist[pixel]++;
    }
}
```

```

    }

    int total = width * height;
    int sumB = 0, wB = 0;
    float maxVar = 0;
    unsigned char threshold = 0;

    int sum1 = 0;
    for (int i = 0; i < 256; i++) sum1 += i * hist[i];

    for (int i = 0; i < 256; i++) {
        wB += hist[i];
        if (wB == 0) continue;

        int wF = total - wB;
        if (wF == 0) break;

        sumB += i * hist[i];
        float mB = static_cast<float>(sumB) / wB;
        float mF = static_cast<float>(sum1 - sumB) / wF;

        float varBetween = static_cast<float>(wB * wF) * (mB - mF) * (mB - mF);
        if (varBetween > maxVar) {
            maxVar = varBetween;
            threshold = i;
        }
    }
    return threshold;
}

void applyThreshold(vector<unsigned char>& data, int width, int height) {
    unsigned char threshold = otsuThreshold(data, width, height);
    for (size_t i = 0; i < data.size(); ++i) {
        data[i] = (data[i] > threshold) ? 255 : 0;
    }
}

```



C++

Copy code

## Integrating Preprocessing with QR Code Parsing:

Here's how you would integrate these preprocessing steps into the main image loading and QR code parsing flow:

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " <path-to-qr-code-image>" << endl;
        return 1;
    }

    int width, height, channels;
    unsigned char* imageData = stbi_load(argv[1], &width, &height, &channels, 0);
    if (imageData == nullptr) {
        cerr << "Error loading image: " << argv[1] << endl;
        return 1;
    }

    vector<unsigned char> greyData = preprocessImage(imageData, width, height,
channels);
    stbi_image_free(imageData);

    medianFilter(greyData, width, height); // Apply noise reduction
    applyThreshold(greyData, width, height); // Apply thresholding

    try {
        Ref<LuminanceSource> source(new CustomLuminanceSource(&greyData[0], width,
height));
        Ref<Binarizer> binarizer(new GlobalHistogramBinarizer(source));
        Ref<BinaryBitmap> binary(new BinaryBitmap(binarizer));

        DecodeHints hints
```

C++

Copy code

Copy

Good response

Bad response

continue

Certainly! Here's the continuation of the code:

```
DecodeHints hints(DecodeHints::DEFAULT_HINT);
MultiFormatReader reader;

Ref<Result> result(reader.decode(binary, hints));
cout << "QR Code Text: " << result->getText()->getText() << endl;
} catch (const ReaderException& e) {
    cerr << "Failed to decode QR code: " << e.what() << endl;
} catch (const zxing::Exception& e) {
    cerr << "ZXing error: " << e.what() << endl;
} catch (const exception& e) {
    cerr << "Error: " << e.what() << endl;
}

return 0;
}
```

C++

Copy code

## Explanation of the Code:

### 1. Image Loading:

2. `unsigned char* imageData = stbi_load(argv[1], &width, &height, &channels, 0);`
3. `if (imageData == nullptr) {`
4.  `cerr << "Error loading image: " << argv[1] << endl;`
5.  `return 1;`
6. `}`

C++

Copy code

### 7. Preprocessing:

- o Convert image data to grayscale:
- o `vector<unsigned char> greyData = preprocessImage(imageData, width, height, channels);`

C++

Copy code

- Free the original image data since it's now in `greyData` and we no longer need the raw image data:
- `stbi_image_free(imageData);`

C++

Copy code

- Apply noise reduction using a median filter:
- `medianFilter(greyData, width, height);`

C++

Copy code

- Apply Otsu's thresholding to binarize the image:
- `applyThreshold(greyData, width, height);`

C++

Copy code

## 8. Custom LuminanceSource:

- Use `CustomLuminanceSource` to wrap the preprocessed image data:
- `Ref<LuminanceSource> source(new CustomLuminanceSource(&greyData[0], width, height));`

C++

Copy code

## 9. Binarization and QR Code Decoding:

- Create a `GlobalHistogramBinarizer` from the `LuminanceSource`:
- `Ref<Binarizer> binarizer(new GlobalHistogramBinarizer(source));`

C++

Copy code

- Wrap the binarizer result into a `BinaryBitmap`:
- `Ref<BinaryBitmap> binary(new BinaryBitmap(binarizer));`

C++

Copy code

- Set decode hints, if any. Here, default hints are set:
- DecodeHints `hints`(DecodeHints::DEFAULT\_HINT);

C++

Copy code

- Initialize a `MultiFormatReader` and decode the QR code:
- `MultiFormatReader reader;`
- `Ref<Result> result(reader.decode(binary, hints));`

C++

Copy code

## 10. Error Handling:

- Handle different potential exceptions to ensure the program exits gracefully if something goes wrong:
- `catch (const ReaderException& e) { cerr << "Failed to decode QR code: " << e.what() << endl; }`
- `catch (const zxing::Exception& e) { cerr << "ZXing error: " << e.what() << endl; }`
- `catch (const exception& e) { cerr << "Error: " << e.what() << endl; }`

C++

Copy code

## Testing and Optimization:

- **Test with Different Images:** Ensure to test with a variety of QR code images to validate robustness.
- **Fine-Tuning:** Depending on the quality and nature of your QR code images, you might need to tweak the preprocessing parameters such as filter window size and thresholding technique.
- **Logging and Debugging:** Add logging or debugging statements to better understand where the process might be failing, especially with images that do not get decoded correctly.

This complete example should give you a solid foundation for reading and preprocessing QR code images without relying on OpenCV, but leveraging the power of ZXing-C++.

If you have any other questions about specific parts of the process or need further assistance, feel free to ask!

## preprocessing for distorted qr image

Preprocessing distorted QR codes can be challenging but feasible with a series of image transformations aimed at correcting distortions and enhancing QR readability. Here, we can focus on common distortions like perspective distortions and apply necessary preprocessing steps.

Here's an extended version of the previous code that includes corrections for potential distortions, leveraging additional techniques such as perspective transformation. Note that we are still not using OpenCV to keep it more aligned with your constraints, but this approach might not cover all use cases perfectly compared to more specialized libraries like OpenCV.

## Handling Perspective Distortion:

Let's consider a basic approach for perspective correction. For more sophisticated corrections, integration with libraries like OpenCV would typically be necessary, but here we'll continue in a manner similar to previous methods:

## Complete Code with Basic Perspective Correction:

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <zxing/common/Counted.h>
#include <zxing/LuminanceSource.h>
#include <zxing/BarcodeFormat.h>
#include <zxing/DecodeHints.h>
```

```

#include <zxing/MultiFormatReader.h>
#include <zxing/common/GlobalHistogramBinarizer.h>
#include <zxing/Exception.h>
#include <zxing/ReaderException.h>
#include <zxing/Result.h>

using namespace std;
using namespace zxing;
using namespace zxing::multi;

vector<unsigned char> preprocessImage(const unsigned char* imageData, int width,
int height, int channels) {
    vector<unsigned char> greyData(width * height);
    for (int i = 0; i < width * height; ++i) {
        int offset = i * channels;
        greyData[i] = static_cast<unsigned char>(0.299f * imageData[offset]
                                                    + 0.587f * imageData[offset + 1]
                                                    + 0.114f * imageData[offset +
2]);
    }
    return greyData;
}

void medianFilter(vector<unsigned char>& data, int width, int height) {
    vector<unsigned char> filteredData = data;
    const int windowSize = 3;
    const int kernelSize = windowSize * windowSize;
    const int edge = windowSize / 2;

    for (int y = edge; y < height - edge; ++y) {
        for (int x = edge; x < width - edge; ++x) {
            vector<unsigned char> window;
            for (int ky = 0; ky < windowSize; ++ky) {
                for (int kx = 0; kx < windowSize; ++kx) {
                    window.push_back(data[(y + ky - edge) * width + (x + kx -
edge)]);
                }
            }
        }
    }
}

```

```

        }
        nth_element(window.begin(), window.begin() + kernelSize / 2,
window.end());
        filteredData[y * width + x] = window[kernelSize / 2];
    }
}
data = filteredData;
}

```

```

unsigned char otsuThreshold(const vector<unsigned char>& data, int width, int
height) {
    int hist[256] = {0};
    for (unsigned char pixel : data) {
        hist[pixel]++;
    }

    int total = width * height;
    int sumB = 0, wB = 0;
    float maxVar = 0;
    unsigned char threshold = 0;

    int sum1 = 0;
    for (int i = 0; i < 256; i++) sum1 += i * hist[i];

    for (int i = 0; i < 256; i++) {
        wB += hist[i];
        if (wB == 0) continue;

        int wF = total - wB;
        if (wF == 0) break;

        sumB += i * hist[i];
        float mB = static_cast<float>(sumB) / wB;
        float mF = static_cast<float>(sum1 - sumB) / wF;

        float varBetween = static_cast<float>(wB) * wF * (mB - mF) * (mB - mF);
    }
}

```

```

        if (varBetween > maxVar) {
            maxVar = varBetween;
            threshold = i;
        }
    }
    return threshold;
}

void applyThreshold(vector<unsigned char>& data, int width, int height) {
    unsigned char threshold = otsuThreshold(data, width, height);
    for (size_t i = 0; i < data.size(); ++i) {
        data[i] = (data[i] > threshold) ? 255 : 0;
    }
}

void rotateImage90Degrees(vector<unsigned char>& data, int& width, int& height) {
    vector<unsigned char> rotated

```

C++

Copy code

Copy

Good response

Bad response

continue

Sure, let's continue implementing the `rotateImage90Degrees` function and then proceed with adding the basic perspective correction functionality:

```

    vector<unsigned char> rotatedData(width * height);
    int newWidth = height;
    int newHeight = width;

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            rotatedData[(newWidth * x) + (newWidth - y - 1)] = data[(width * y) +
x];
        }
    }
}

```



```

    data = rotatedData;
    width = newWidth;
    height = newHeight;
}

// Simple function to simulate perspective correction using linear interpolation
// Note: For real-world usage, you may need more sophisticated tools like OpenCV
void correctPerspective(vector<unsigned char>& data, int& width, int& height) {
    // Define four corner points of the input quadrilateral (assumed distorted QR
    code's corners)

    int x1 = 0;           // top-left corner
    int y1 = 0;
    int x2 = width - 1;   // top-right corner
    int y2 = 0;
    int x3 = width - 1;   // bottom-right corner
    int y3 = height - 1;
    int x4 = 0;           // bottom-left corner
    int y4 = height - 1;

    // Here, we simplistically assume these points. For real-world use, you'd
    need a method to find these points dynamically

    // Calculate the dimensions of the corrected image
    int newWidth = width;
    int newHeight = height;

    vector<unsigned char> correctedData(newWidth * newHeight);

    for (int newY = 0; newY < newHeight; ++newY) {
        for (int newX = 0; newX < newWidth; ++newX) {
            float origX = newX * (x2 - x1) / static_cast<float>(newWidth) + y1 *
(x4 - x1) / static_cast<float>(newHeight);
            float origY = newY * (y3 - y1) / static_cast<float>(newHeight) + x1 *
(y4 - y1) / static_cast<float>(newWidth);
            if (origX < 0) origX = 0;
            if (origX >= width) origX = width - 1;
            if (origY < 0) origY = 0;
            if (origY >= height) origY = height - 1;

```

```

        correctedData[newY * newWidth + newX] = data[static_cast<int>(origY)
* width + static_cast<int>(origX)];
    }
}

data = correctedData;
width = newWidth;
height = newHeight;
}

class CustomLuminanceSource : public LuminanceSource {
private:
    vector<unsigned char> luminances_;
    int width_;
    int height_;

public:
    CustomLuminanceSource(const unsigned char* data, int width, int height)
        : LuminanceSource(width, height), luminances_(data, data + width *
height), width_(width), height_(height) {}

    ArrayRef<char> getRow(int y, ArrayRef<char> row) const override {
        if (row->size() != width_) {
            row.reset(ArrayRef<char>(width_));
        }
        memcpy(&row[0], &luminances_[y * width_], width_);
        return row;
    }

    ArrayRef<char> getMatrix() const override {
        ArrayRef<char> matrix(width_ * height_);
        memcpy(&matrix[0], &luminances_[0], width_ * height_);
        return matrix;
    }
};

```

```

int main(int argc, char *argv[]) {
    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " <path-to-qr-code-image>" << endl;
        return 1;
    }

    int width, height, channels;
    unsigned char* imageData = stbi_load(argv[1], &width, &height, &channels, 0);
    if (imageData == nullptr) {
        cerr << "Error loading image: " << argv[1] << endl;
        return 1;
    }

    vector<unsigned char> greyData = preprocessImage(imageData, width, height,
channels);
    stbi_image_free(imageData);

    medianFilter(greyData, width, height); // Apply noise reduction
    applyThreshold(greyData, width, height); // Apply thresholding

    // Optional: Correct perspective if required
    correctPerspective(greyData, width,

```

C++

Copy code

Copy

Good response

Bad response

continue

Certainly, let's continue from where we left off with the `main` function, incorporating the perspective correction step and finishing the QR code decoding process:

```

    correctPerspective(greyData, width, height);

    // Optional: Rotate the image if required
    // Uncomment to rotate 90 degrees
    // rotateImage90Degrees(greyData, width, height);
    // Uncomment and repeat rotation if 180 degrees is required

```

```

// rotateImage90Degrees(greyData, width, height);

try {
    Ref<LuminanceSource> source(new CustomLuminanceSource(&greyData[0], width,
height));
    Ref<Binarizer> binarizer(new GlobalHistogramBinarizer(source));
    Ref<BinaryBitmap> binary(new BinaryBitmap(binarizer));

    DecodeHints hints(DecodeHints::DEFAULT_HINT);
    MultiFormatReader reader;

    Ref<Result> result(reader.decode(binary, hints));
    cout << "QR Code Text: " << result->getText()->getText() << endl;
} catch (const ReaderException& e) {
    cerr << "Failed to decode QR code: " << e.what() << endl;
} catch (const zxing::Exception& e) {
    cerr << "ZXing error: " << e.what() << endl;
} catch (const exception& e) {
    cerr << "Error: " << e.what() << endl;
}

return 0;
}

```

C++

Copy code

## Explanation of Perspective Correction:

- **Perspective Transform:** The `correctPerspective` function applies a basic perspective correction using linear interpolation. It calculates the new coordinates (`origX`, `origY`) for each pixel in the corrected image (`correctedData`) based on the original distorted image (`data`).

## Complete Processing Steps:

1. **Load Image:** Load the image using the `stb_image` library.
2. **Convert to Grayscale:** Convert the image to grayscale.
3. **Noise Reduction:** Apply a median filter to reduce noise.
4. **Thresholding:** Apply Otsu's threshold to binarize the image.

5. **Perspective Correction:** Apply a perspective correction to handle any distortion.
6. **Rotate Image** (optional): Rotate the image if needed.
7. **Decode QR Code:** Use the ZXing library to decode the QR code from the preprocessed image.

**Notes:**

- The perspective correction implemented here is rudimentary and assumes fixed points. In a real-world scenario, it's often more complex and requires detecting corner points dynamically.
- For more advanced perspective correction and handling more complex distortions, using libraries such as OpenCV is highly recommended.