Bayer Image Processing Web Application

This Flask web application receives raw Bayer images from a smartphone, processes them, and returns the processed results. The application is designed to be modular, beautiful, and easy to understand.

Project Structure

```
bayer_image_processor/
├── app.py                # Main Flask application
├── static/               # Static assets
│   ├── css/
│   │   └── style.css     # CSS styles
│   └── js/
│       └── script.js     # Frontend JavaScript
├── templates/            # HTML templates
│   └── index.html        # Main page template
├── uploads/              # Directory for uploaded raw images
├── processed/            # Directory for processed images
├── requirements.txt      # Python dependencies
└── README.md             # Project documentation
```

Prerequisites

- Python 3.7+
- pip (Python package manager)
- Virtual environment (recommended)

Installation

1. **Clone or download the project**
2. **Create and activate a virtual environment (recommended)**

   bash

   ```
   # Create virtual environment
   python -m venv venv

   # Activate on Windows
   venv\Scripts\activate

   # Activate on macOS/Linux
   source venv/bin/activate
   ```

3. **Install dependencies**

   bash

   ```
   pip install -r requirements.txt
   ```

4. **Create required directories**

   bash

   ```
   mkdir -p uploads processed
   ```

Running the Application

1. **Start the Flask server**

   bash

   ```
   python app.py
   ```

2. **Access the application** Open your web browser and go to:

   ```
   http://localhost:5000
   ```

   If deploying on a different machine, use that IP address instead of localhost.

Connecting Your Smartphone

In your smartphone app (Kotlin), send the raw Bayer image data to:

```
http://<your-server-ip>:5000/process
```

Example Kotlin Code

kotlin

```kotlin
import okhttp3.*
import java.io.File
import java.io.IOException

fun sendBayerImage(bayerImageFile: File, imageWidth: Int, imageHeight: Int) {
    val client = OkHttpClient()

    // Create multipart request
    val requestBody = MultipartBody.Builder()
        .setType(MultipartBody.FORM)
        .addFormDataPart(
            "bayer_image",
            bayerImageFile.name,
            RequestBody.create(MediaType.parse("application/octet-stream"), bayerImage
        )
        .addFormDataPart(
            "metadata",
            "{\"width\": $imageWidth, \"height\": $imageHeight, \"bayer_pattern\": \"R
        )
        .build()

    val request = Request.Builder()
        .url("http://your-server-ip:5000/process")
        .post(requestBody)
        .build()

    client.newCall(request).enqueue(object : Callback {
        override fun onFailure(call: Call, e: IOException) {
            // Handle network failure
            e.printStackTrace()
        }

        override fun onResponse(call: Call, response: Response) {
            // Handle success
            val responseBody = response.body()?.string()
            // Parse JSON response and handle the processed image URL
        }
    })
}
```

Deep Learning Integration

To integrate your actual deep learning model, modify the `placeholder_deep_learning_process()` function in `app.py`.

```python
def placeholder_deep_learning_process(bayer_image):
    """
    Replace this function with your actual deep learning processing code.

    Args:
        bayer_image (numpy.ndarray): Raw Bayer image as numpy array

    Returns:
        numpy.ndarray: Processed image
    """
    # Your deep learning code goes here
    # For example:
    # model = load_your_model()
    # processed_image = model.predict(bayer_image)
    # return processed_image

    # Placeholder implementation
    return processed_image
```

Configuration Options

The application has several configuration options that can be modified in `app.py`:

- `UPLOAD_FOLDER`: Directory to store uploaded raw images
- `PROCESSED_FOLDER`: Directory to store processed images
- `MAX_CONTENT_LENGTH`: Maximum allowed file size (default 16MB)
- `PORT`: Port number for the Flask server (default 5000)

Customizing the Interface

You can customize the interface by modifying:

- `templates/index.html` - HTML structure
- `static/css/style.css` - Styling
- `static/js/script.js` - Frontend behavior

Deployment for Production

For production deployment, it's recommended to use a production-ready WSGI server like Gunicorn:

1. **Install Gunicorn**

   bash

   ```
   pip install gunicorn
   ```

2. **Run with Gunicorn**

   bash

   ```
   gunicorn -w 4 -b 0.0.0.0:5000 app:app
   ```

3. **Set up Nginx (recommended)** For better performance, consider setting up Nginx as a reverse proxy in front of Gunicorn.

Securing the Application

For production use, consider:

1. Adding authentication (e.g., API keys or user login)
2. Enabling HTTPS
3. Implementing rate limiting
4. Validating uploads thoroughly
5. Using environment variables for sensitive configuration

Requirements.txt

```
Flask==2.2.3
numpy==1.24.2
Pillow==9.4.0
gunicorn==20.1.0
```