Geometric Distortion Correction (GDC) Algorithm

Technical Specification Document

Version: 1.0
Date: May 28, 2025
Document Type: Algorithm Specification

---

1. Executive Summary

The Geometric Distortion Correction (GDC) Algorithm is a map-based image remapping system designed to correct lens-induced geometric distortions in digital images. The algorithm employs a two-stage interpolation process: grid interpolation in virtual domain and pixel interpolation for image warping, utilizing sparse distortion grids for memory-efficient correction.

Key Features

- Memory-efficient sparse grid representation
- Virtual domain processing for improved accuracy
- Support for multiple interpolation methods
- Real-time processing capability
- Scalable across different image resolutions

---

2. Algorithm Overview

2.1 Problem Statement

Camera lenses introduce systematic geometric distortions including:

- **Barrel Distortion**: Outward bulging effect
- **Pincushion Distortion**: Inward pinching effect
- **Fisheye Distortion**: Extreme wide-angle warping
- **Complex Distortions**: Combination of multiple distortion types

2.2 Solution Approach

The GDC algorithm transforms distorted images to geometrically correct representations through:

1. Sparse distortion grid mapping
2. Virtual domain grid interpolation
3. Image warping via pixel interpolation

---

3. Input/Output Specifications

3.1 Inputs

Parameter	Type	Description	Constraints
`InputImage`	Image Array	Distorted source image	H×W×C (C=1,3,4)
`GridX`	Float Array	X-displacement grid	M×N sparse grid
`GridY`	Float Array	Y-displacement grid	M×N sparse grid
`OutputWidth`	Integer	Target image width	> 0
`OutputHeight`	Integer	Target image height	> 0
`InterpolationMethod`	Enum	Pixel interpolation type	BILINEAR, BICUBIC, LANCZOS

3.2 Outputs

Parameter	Type	Description
`OutputImage`	Image Array	Geometrically corrected image
`ProcessingStatus`	Enum	Algorithm execution status

---

4. Detailed Algorithm Description

4.1 Stage 1: Grid Interpolation (Virtual Domain)

4.1.1 Purpose

Transform sparse distortion grid to dense pixel-level mapping coordinates.

4.1.2 Virtual Domain Setup

```
VirtualWidth = OutputWidth
VirtualHeight = OutputHeight
GridCellWidth = VirtualWidth / (GridColumns - 1)
GridCellHeight = VirtualHeight / (GridRows - 1)
```

4.1.3 Dense Grid Generation Algorithm

Input: Sparse grids `GridX[M][N]`, `GridY[M][N]`

Output: Dense grids `DenseGridX[H][W]`, `DenseGridY[H][W]`

pseudocode

```
ALGORITHM: GenerateDenseGrid()
BEGIN
    FOR each output pixel (i, j) in [0, OutputHeight) × [0, OutputWidth)
        // Map pixel to virtual grid coordinates
        virtual_x = j * (GridColumns - 1) / (OutputWidth - 1)
        virtual_y = i * (GridRows - 1) / (OutputHeight - 1)

        // Find surrounding grid cells
        grid_x_low = floor(virtual_x)
        grid_y_low = floor(virtual_y)
        grid_x_high = min(grid_x_low + 1, GridColumns - 1)
        grid_y_high = min(grid_y_low + 1, GridRows - 1)

        // Calculate interpolation weights
        weight_x = virtual_x - grid_x_low
        weight_y = virtual_y - grid_y_low

        // Bilinear interpolation
        top_left_x = GridX[grid_y_low][grid_x_low]
        top_right_x = GridX[grid_y_low][grid_x_high]
        bottom_left_x = GridX[grid_y_high][grid_x_low]
        bottom_right_x = GridX[grid_y_high][grid_x_high]

        top_interp_x = (1 - weight_x) * top_left_x + weight_x * top_right_x
        bottom_interp_x = (1 - weight_x) * bottom_left_x + weight_x * bottom_right_x
        DenseGridX[i][j] = (1 - weight_y) * top_interp_x + weight_y * bottom_interp_x

        // Repeat for Y coordinates
        top_left_y = GridY[grid_y_low][grid_x_low]
        top_right_y = GridY[grid_y_low][grid_x_high]
        bottom_left_y = GridY[grid_y_high][grid_x_low]
        bottom_right_y = GridY[grid_y_high][grid_x_high]

        top_interp_y = (1 - weight_x) * top_left_y + weight_x * top_right_y
        bottom_interp_y = (1 - weight_x) * bottom_left_y + weight_x * bottom_right_y
        DenseGridY[i][j] = (1 - weight_y) * top_interp_y + weight_y * bottom_interp_y
    END FOR
END
```

4.2 Stage 2: Pixel Interpolation (Image Warping)

4.2.1 Purpose

Generate corrected image by sampling from distorted input using dense mapping grid.

4.2.2 Image Warping Algorithm

Input: `InputImage`, `DenseGridX`, `DenseGridY`

Output: `OutputImage`

```pseudocode
ALGORITHM: ImageWarping()
BEGIN
    FOR each output pixel (i, j) in [0, OutputHeight) × [0, OutputWidth)
        // Get mapped coordinates in input image
        src_x = DenseGridX[i][j]
        src_y = DenseGridY[i][j]

        // Boundary check
        IF (src_x < 0 OR src_x >= InputWidth OR src_y < 0 OR src_y >= InputHeight)
            OutputImage[i][j] = DefaultColor  // Usually black or transparent
            CONTINUE
        END IF

        // Perform pixel interpolation
        pixel_value = InterpolatePixel(InputImage, src_x, src_y, InterpolationMethod)
        OutputImage[i][j] = pixel_value
    END FOR
END
```

4.2.3 Interpolation Methods

Bilinear Interpolation

pseudocode

```
FUNCTION BilinearInterpolation(Image, x, y)
BEGIN
    x1 = floor(x)
    y1 = floor(y)
    x2 = min(x1 + 1, ImageWidth - 1)
    y2 = min(y1 + 1, ImageHeight - 1)

    dx = x - x1
    dy = y - y1

    pixel_value = (1-dx)*(1-dy)*Image[y1][x1] +
                  dx*(1-dy)*Image[y1][x2] +
                  (1-dx)*dy*Image[y2][x1] +
                  dx*dy*Image[y2][x2]

    RETURN pixel_value
END
```

Bicubic Interpolation

pseudocode

```
FUNCTION BicubicInterpolation(Image, x, y)
BEGIN
    x_int = floor(x)
    y_int = floor(y)
    dx = x - x_int
    dy = y - y_int

    // Extract 4x4 neighborhood
    FOR i = 0 to 3
        FOR j = 0 to 3
            px = clamp(x_int - 1 + i, 0, ImageWidth - 1)
            py = clamp(y_int - 1 + j, 0, ImageHeight - 1)
            neighborhood[i][j] = Image[py][px]
        END FOR
    END FOR

    // Apply bicubic kernel
    pixel_value = 0
    FOR i = 0 to 3
        FOR j = 0 to 3
            weight = CubicKernel(i - 1 - dx) * CubicKernel(j - 1 - dy)
            pixel_value += weight * neighborhood[i][j]
        END FOR
    END FOR

    RETURN clamp(pixel_value, 0, 255)
END

FUNCTION CubicKernel(t)
BEGIN
    t = abs(t)
    IF t <= 1
        RETURN (1.5*t - 2.5)*t*t + 1
    ELSE IF t <= 2
        RETURN ((-0.5*t + 2.5)*t - 4)*t + 2
    ELSE
        RETURN 0
    END IF
END
```

---

5. Mathematical Foundation

5.1 Coordinate Transformation

The complete transformation is represented as:

```
I_output(x, y) = I_input(T_x(x, y), T_y(x, y))
```

Where:

- `T_x(x, y) = DenseGridX[y][x]`
- `T_y(x, y) = DenseGridY[y][x]`

5.2 Grid Interpolation Mathematics

For bilinear interpolation at point (u, v) within grid cell:

```
f(u, v) = (1-α)(1-β)f(0,0) + α(1-β)f(1,0) + (1-α)βf(0,1) + αβf(1,1)
```

Where α and β are the fractional parts of u and v respectively.

5.3 Error Analysis

Interpolation Error Bound: For bilinear interpolation with grid spacing h:

```
|E| ≤ (h²/8) * max|∂²f/∂x²| + max|∂²f/∂y²| + 2*max|∂²f/∂x∂y|)
```

---

6. Performance Characteristics

6.1 Computational Complexity

Operation	Time Complexity	Space Complexity
Grid Interpolation	O(H × W)	O(H × W)
Image Warping	O(H × W)	O(1) additional
Overall	O(H × W)	O(H × W)

Where H = OutputHeight, W = OutputWidth

6.2 Memory Requirements

- **Dense Grid Storage:** 2 × H × W × sizeof(float)
- **Input Image:** H_in × W_in × C × sizeof(pixel)
- **Output Image:** H × W × C × sizeof(pixel)
- **Total:** ~(2 + C) × H × W × sizeof(data_type)

6.3 Performance Optimization Strategies

Cache Optimization

pseudocode

```
// Process in tiles to improve cache locality
ALGORITHM: TiledProcessing()
BEGIN
    TileSize = 64  // Optimize based on cache size
    FOR tile_y = 0 to OutputHeight STEP TileSize
        FOR tile_x = 0 to OutputWidth STEP TileSize
            ProcessTile(tile_x, tile_y, min(TileSize, OutputWidth - tile_x),
                       min(TileSize, OutputHeight - tile_y))
        END FOR
    END FOR
END
```

SIMD Optimization

- Process multiple pixels simultaneously using vector instructions
- Typical speedup: 2-4x for bilinear interpolation
- Hardware-dependent implementation

---

7. Implementation Guidelines

7.1 Error Handling

pseudocode

```
ALGORITHM: RobustGDC()
BEGIN
    // Input validation
    IF InputImage is NULL OR GridX is NULL OR GridY is NULL
        RETURN ERROR_INVALID_INPUT
    END IF

    IF OutputWidth <= 0 OR OutputHeight <= 0
        RETURN ERROR_INVALID_DIMENSIONS
    END IF

    // Grid size validation
    IF GridX.rows != GridY.rows OR GridX.cols != GridY.cols
        RETURN ERROR_GRID_MISMATCH
    END IF

    // Memory allocation with error checking
    TRY
        AllocateDenseGrids()
        AllocateOutputImage()
    CATCH OutOfMemoryException
        RETURN ERROR_INSUFFICIENT_MEMORY
    END TRY

    // Execute algorithm
    status = GenerateDenseGrid()
    IF status != SUCCESS
        RETURN status
    END IF

    status = ImageWarping()
    RETURN status
END
```

7.2 Boundary Conditions

1. **Grid Extrapolation:** Use nearest neighbor for coordinates outside sparse grid
2. **Image Boundaries:** Apply padding or use default color for out-of-bounds sampling
3. **Numerical Stability:** Implement proper floating-point comparisons

7.3 Multi-threading Support

pseudocode

```
ALGORITHM: ParallelImageWarping()
BEGIN
    num_threads = GetOptimalThreadCount()
    rows_per_thread = OutputHeight / num_threads

    FOR thread_id = 0 to num_threads - 1
        start_row = thread_id * rows_per_thread
        end_row = (thread_id == num_threads - 1) ?
                  OutputHeight : (thread_id + 1) * rows_per_thread

        CreateThread(ProcessImageRows, start_row, end_row)
    END FOR

    WaitForAllThreads()
END
```

---

8. Quality Metrics and Validation

8.1 Correction Accuracy Metrics

1. **Straight Line Preservation:** Measure deviation of corrected lines from ideal straight lines
2. **Grid Distortion:** Analyze rectangular grid correction quality
3. **Corner Detection:** Evaluate 90-degree angle preservation

8.2 Image Quality Metrics

1. **PSNR (Peak Signal-to-Noise Ratio):** Compare with reference undistorted image
2. **SSIM (Structural Similarity Index):** Perceptual quality assessment
3. **Edge Preservation:** Measure edge sharpness retention

8.3 Performance Benchmarks

```
Standard Test Configuration:
- Input: 1920×1080 RGB image
- Grid: 20×15 sparse grid
- Hardware: Intel i7-10700K, 32GB RAM
- Expected Performance: <50ms processing time
```

---

9. Applications and Use Cases

9.1 Computer Vision

- **Stereo Vision:** Camera rectification for depth estimation
- **Object Detection:** Preprocessing for improved feature extraction
- **Augmented Reality:** Real-time lens correction

9.2 Image Processing Pipelines

- **Camera ISP:** Built-in lens correction
- **Video Processing:** Real-time distortion correction
- **Photography:** Post-processing correction tools

9.3 Industrial Applications

- **Machine Vision:** Quality control and inspection
- **Autonomous Vehicles:** Camera calibration for perception
- **Medical Imaging:** Endoscopy and microscopy correction

---

10. Limitations and Considerations

10.1 Algorithm Limitations

- Requires accurate calibration data
- Assumes static distortion patterns
- May introduce slight image blur due to interpolation
- Processing overhead for real-time applications

10.2 Quality Trade-offs

Interpolation Method	Quality	Speed	Memory
Bilinear	Good	Fast	Low
Bicubic	Better	Medium	Medium
Lanczos	Best	Slow	High

10.3 Hardware Requirements

- **Minimum:** 4GB RAM, dual-core processor
- **Recommended:** 16GB RAM, quad-core processor with SIMD support
- **Optimal:** GPU acceleration for real-time processing

---

11. Future Enhancements

11.1 Advanced Interpolation

- Implement edge-preserving interpolation methods
- Adaptive interpolation based on local image content
- Machine learning-based super-resolution integration

11.2 Dynamic Distortion Handling

- Support for zoom lens distortion variation
- Real-time calibration updates
- Multi-grid interpolation for complex distortion patterns

11.3 Hardware Acceleration

- GPU shader implementations
- FPGA acceleration for embedded systems
- Neural processing unit (NPU) optimization

---

12. References and Standards

12.1 Related Standards

- ISO 17850: Camera calibration standards
- OpenCV camera calibration framework
- MATLAB Camera Calibration Toolbox methodologies

12.2 Mathematical References

- Digital Image Processing (Gonzalez & Woods)
- Computer Vision: Algorithms and Applications (Szeliski)
- Numerical Recipes in C (Press et al.)

Appendix A: Code Examples

A.1 Basic Implementation Structure

```c
class GDCProcessor {
private:
    float** denseGridX;
    float** denseGridY;
    int outputWidth, outputHeight;
    int gridRows, gridCols;

public:
    bool Initialize(float** gridX, float** gridY,
                   int rows, int cols, int outW, int outH);
    bool ProcessImage(const Image& input, Image& output);
    void SetInterpolationMethod(InterpolationMethod method);

private:
    bool GenerateDenseGrid(float** sparseGridX, float** sparseGridY);
    bool WarpImage(const Image& input, Image& output);
    float InterpolatePixel(const Image& img, float x, float y);
};
```

A.2 Performance Profiling Template

```c
struct GDCMetrics {
    double gridInterpolationTime;
    double imageWarpingTime;
    double totalProcessingTime;
    size_t memoryUsage;
    double psnr;
    double ssim;
};
```

Document Control

- **Author:** Algorithm Development Team
- **Reviewers:** Computer Vision Team, Performance Engineering
- **Approval:** Technical Architecture Board
- **Next Review Date:** November 28, 2025