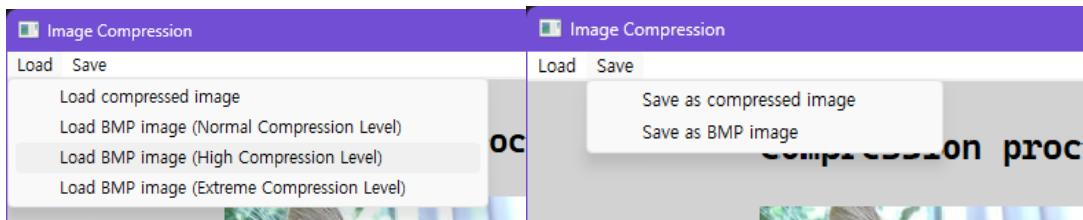


기말 프로젝트 – 이미지 압축 및 인코딩, 디코딩

2019102098 소프트웨어융합 방민수

1. 동작 설명

A. 주요 메뉴 및 기능



메뉴는 Load, Save 로 2개의 SubMenu 로 이루어져 있습니다.

Load 에서는 압축된 이미지 (*.comp) 를 불러오거나, BMP 이미지를 불러올 수 있습니다.

BMP 이미지를 불러오는 경우 Compression Level 을 결정할 수 있습니다. (Normal, High, Extreme)

Save 에서는 현재 프로그램에 보여지는 이미지를 압축된 이미지 (*.comp) 로 저장하거나, BMP 이미지로 저장할 수 있습니다. 즉, BMP 와 압축 이미지간 상호 변환이 가능합니다.

B. Compression process



위의 메뉴에서 Load BMP image (High Compression Level) 을 선택하고 BMP 이미지를 선택하면 다음과 같은 화면이 나타납니다. 실제 압축 과정을 Visualize 하기 위해 다음과 같이 화면을 구성하였습니다.

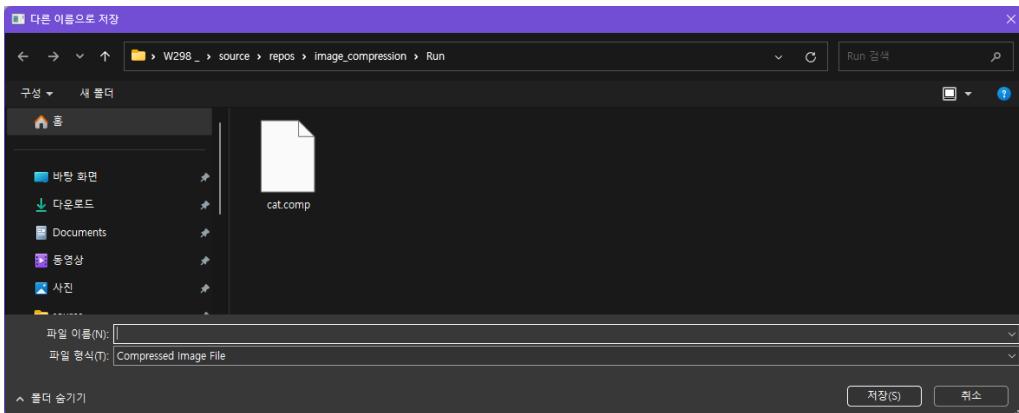
Process

- 원본 이미지
- DWT (Discrete Wavelet Transform) 을 2번 진행한 이미지
- Quantization step size 를 값이 높을수록 R 값을 높게 Visualize 한 이미지
- 인코딩된 데이터의 길이 (Bit 수) 를 한 1 Bit 당 1 Pixel 로 표시한 이미지
- 인코딩된 데이터를 이용해 실제로 디코딩을 진행한 복원 이미지

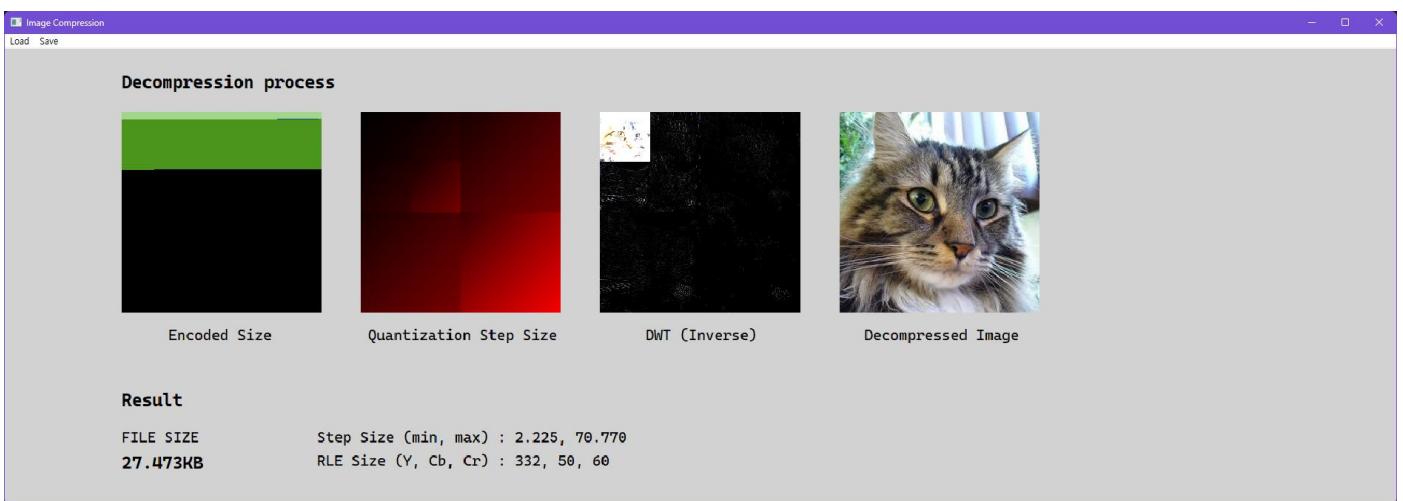
Result

- PSNR 값
- SSIM 값
- 인코딩된 파일의 크기 및 원본 이미지와의 비율
- Quantization step size 의 min, max 값
- RLE pair 의 길이 (Y, Cb, Cr)

이 상태에서 Save as compressed image 메뉴를 선택하면 다음과 같은 창이 나오고, 저장하고 싶은 위치와 이름을 지정하면 됩니다. 압축된 이미지가 Binary File로 작성됩니다.



C. Decompression process

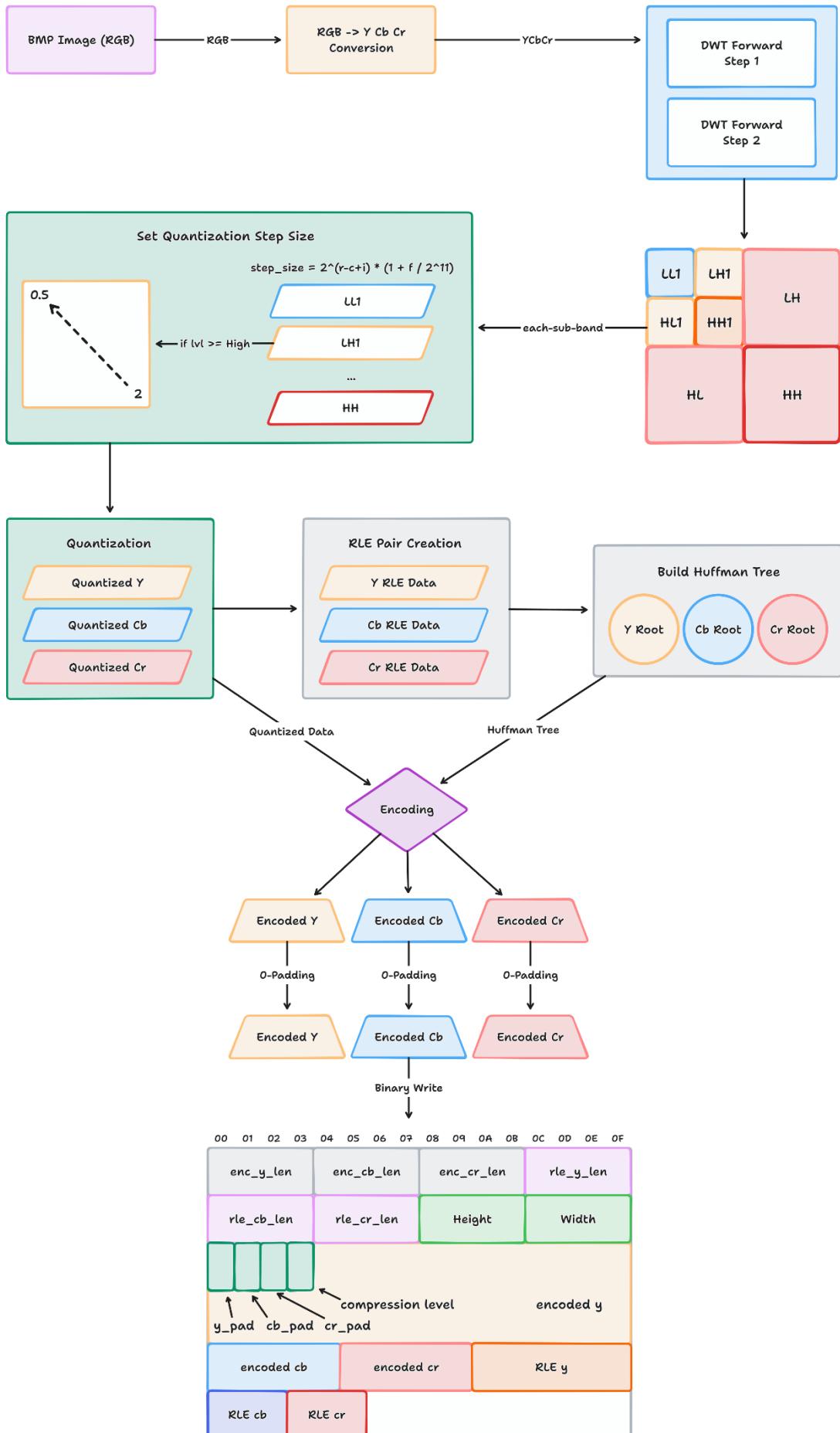


Load compressed image 메뉴를 선택하고, 저장한 이미지를 선택해 불러오면 다음과 같은 Decompression 과정이 보이게 됩니다. Decompression은 Compression의 반대로 진행하면 되기에 위의 구성과 순서만 다를 뿐 거의 같습니다.

Save as BMP image 메뉴를 이용해 압축된 이미지를 다시 BMP 이미지로 변환할 수도 있습니다.

2. 과정 요약

YCbCr 변환 -> DWT -> Quantization -> RLE Pair, Huffman Tree Build -> Encoding -> 0-Padding -> Binary Write



3. 주요 소스 코드

실제 핵심 코드 중 압축, 복원하는 과정 위주의 코드로 설명하였습니다. 파일에 Binary Data 를 쓰거나, 읽거나, Menu 를 만들거나 하는 부분은 생략하였습니다.

A. Main.cpp (Core Method)

```
class CImageProcessing : public CKhuGleWin
{
public:
    CKhuGleImageLayer* m_pImageLayer;
    std::string m_HeaderStr, m_ValueStr;
    std::string m_ResultStr, m_ResultStr2;
    std::unique_ptr<CompResult> m_TempResult;
    int m_CurrentMode = -1;

    CImageProcessing(int nW, int nH);
    ~CImageProcessing() override;
    void Update() override;
    void CleanUp();
    void LoadBMPAndCompress(const char* path, int lvl = 0);
    void LoadComp(const char* read_path);
    void SaveAsComp(const char* write_path) const;
    void SaveAsBMP(const char* write_path) const;
    void OnFileEvent(std::string path, int mode) override;
};
```

- CleanUp : 현재 상태를 Reset 합니다. 이 과정에서 m_TempResult 가 Release 됩니다.
- LoadBMPAndCompress : path 의 BMP 이미지를 로드하고, 압축과 인코딩을 진행한 후, 이를 m_TempResult 에 저장합니다. 이후 디코딩을 통해 원본 이미지와 복원 이미지와의 테스트를 진행합니다.
- LoadComp : 압축된 이미지를 로드하고 디코딩을 거쳐 복원시킵니다.
- SaveAsComp : 현재 압축, 인코딩한 데이터를 압축된 이미지 파일로 저장합니다. m_TempResult 에 있는 값을 사용하며, 값이 없을 경우 Warning MessageBox 를 호출합니다.
- SaveAsBMP : 현재 디코딩한 데이터를 BMP 이미지 파일로 저장합니다. 이는 m_pImageLayer 에 있는 값을 기반으로 저장을 진행합니다.
- OnFileEvent : 메뉴를 클릭했을 때 파일 선택 창을 통해서 path 를 지정해주면 이 함수가 실행됩니다. (KhuGleWin 에서 virtual 로 선언해 두었음)

B. Type.h

```
struct HeaderInfo
{
    int y_len, cb_len, cr_len;
    int rle_y_len, rle_cb_len, rle_cr_len;
    int h, w;
    unsigned char y_pad, cb_pad, cr_pad;
    unsigned char lvl;

    // constructor 생략
};

class CompResult
{
public:
    HeaderInfo info_;

    std::string encoded_data_y_;
    std::string encoded_data_cb_;
    std::string encoded_data_cr_;

    std::vector<std::pair<int, int>> rle_y_;
    std::vector<std::pair<int, int>> rle_cb_;
    std::vector<std::pair<int, int>> rle_cr_;

    // constructor 생략
};
```

HeaderInfo : 압축된 이미지의 Header 정보를 나타내는 구조체입니다. 동일한 순서로 Binary File 에 기록됩니다. (길이들을 저장하

는 이유는 값을 읽을 때 Offset 이 필요하기 때문)

- y_len, cb_len, cr_len: 인코딩된 이미지 데이터의 길이 (Bit 수)
- rle_y_len, rle_cb_len, rle_cr_len : RLE pair 데이터의 길이
- h, w: 이미지의 Height, Width
- y_pad, cb_pad, cr_pad : 0-Padding 의 개수
- lvl : Compression Level

CompResult : Compression 된 이미지의 정보들을 담고 있습니다.

- info_ : 위에서 정의한 HeaderInfo 의 데이터
- encoded_data_y_, encoded_data_cb_, encoded_data_cr_ : 인코딩된 이미지 데이터 (Y, Cb, Cr)
- rle_y_, rle_cb_, rle_cr_ : RLE Pair 데이터

C. LoadBMPAndCompress (Definition)

```
void CImageProcessing::LoadBMPAndCompress(const char* path, int lvl)
{
    // 먼저 헤더를 읽어 이미지의 사이즈를 받아옴
    int image_size[2];
    m_pImageLayer->m_ImageVec[0].ReadBmp(path, image_size);

    const int bmp_height = image_size[0];
    const int bmp_width = image_size[1];

    // RGB, YCbCr Allocation 진행
    double** original_r = dmatrix(bmp_height, bmp_width);
    double** original_g = dmatrix(bmp_height, bmp_width);
    double** original_b = dmatrix(bmp_height, bmp_width);

    double** original_y = dmatrix(bmp_height, bmp_width);
    double** original_cb = dmatrix(bmp_height / 2, bmp_width / 2);
    double** original_cr = dmatrix(bmp_height / 2, bmp_width / 2);

    // ReadBMP 로 Load 한 데이터가 m_pImageLayer->m_ImageVec[0] 에 있기 때문에 이를 로드
    for (int y = 0; y < bmp_height; ++y)
    {
        for (int x = 0; x < bmp_width; ++x)
        {
            original_r[y][x] = m_pImageLayer->m_ImageVec[0].m_Red[y][x];
            original_g[y][x] = m_pImageLayer->m_ImageVec[0].m_Green[y][x];
            original_b[y][x] = m_pImageLayer->m_ImageVec[0].m_Blue[y][x];
        }
    }

    // RGB 를 YCbCr 로 변환 후, Compression 진행
    // m_pImageLayer->m_ImageVec 을 Reference 로 주었고, Compression 과정을 여기에 그림
    // 결과는 CompResult Type 으로 m_TempResult 에 저장
    RGB2YCbCr(original_r, original_g, original_b, original_y, original_cb, original_cr, bmp_height,
    bmp_width);
    CompressImage(m_TempResult, original_y, original_cb, original_cr, bmp_height, bmp_width, lvl,
    m_pImageLayer->m_ImageVec, m_ResultStr, m_ResultStr2);

    m_pImageLayer->m_ImageVec[4].m_Red = cmatrix(m_nH, m_nW);
    m_pImageLayer->m_ImageVec[4].m_Green = cmatrix(m_nH, m_nW);
    m_pImageLayer->m_ImageVec[4].m_Blue = cmatrix(m_nH, m_nW);

    m_pImageLayer->m_ImageVec[4].m_nH = bmp_height;
    m_pImageLayer->m_ImageVec[4].m_nW = bmp_width;

    // 복원 이미지 Memory Allocation
    double** test_r = dmatrix(bmp_height, bmp_width);
    double** test_g = dmatrix(bmp_height, bmp_width);
    double** test_b = dmatrix(bmp_height, bmp_width);

    double** test_y = dmatrix(bmp_height, bmp_width);
    double** test_cb = dmatrix(bmp_height / 2, bmp_width / 2);
    double** test_cr = dmatrix(bmp_height / 2, bmp_width / 2);

    // Compression 결과로 받은 m_TempResult 의 값을 이용해 이를 Decompress 진행 및
    // test_y, test_cb, test_cr 에 저장
    // 이를 RGB 로 저장
```

```

DecompressImage(m_TempResult.get(), test_y, test_cb, test_cr);
YCbCr2RGB(test_y, test_cb, test_cr, test_r, test_g, test_b, bmp_height, bmp_width);

// 원본 파일 사이즈와 압축된 파일 사이즈를 측정
const int original_file_size = MeasureFileSize(path);
const int file_size = MeasureFileSize(m_TempResult.get());

// 복원된 이미지를 Render
for (int y = 0; y < bmp_height; ++y)
{
    for (int x = 0; x < bmp_width; ++x)
    {
        m_pImageLayer->m_ImageVec[4].m_Red[y][x] = test_r[y][x];
        m_pImageLayer->m_ImageVec[4].m_Green[y][x] = test_g[y][x];
        m_pImageLayer->m_ImageVec[4].m_Blue[y][x] = test_b[y][x];
    }
}

// PSNR, SSIM 계산
double psnr = GetPsnr(
    m_pImageLayer->m_ImageVec[0].m_Red,
    m_pImageLayer->m_ImageVec[0].m_Green,
    m_pImageLayer->m_ImageVec[0].m_Blue,
    m_pImageLayer->m_ImageVec[4].m_Red,
    m_pImageLayer->m_ImageVec[4].m_Green,
    m_pImageLayer->m_ImageVec[4].m_Blue,
    bmp_width, bmp_height);

double ssim = ComputeSSIM(
    m_pImageLayer->m_ImageVec[0].m_Red,
    m_pImageLayer->m_ImageVec[4].m_Red,
    bmp_width, bmp_height);

m_HeaderStr = "PSNR      SSIM      FILE SIZE";

char buffer[100];
sprintf(buffer, "%6.3f %4.3f %6.3fKB (%.3f %% of Original)", psnr, ssim, (double)file_size /
1024,
        (double)file_size / (double)original_file_size * 100);
m_ValueStr = std::string(buffer);

// Release
free_dmatrix(original_r, bmp_height, bmp_width);
free_dmatrix(original_g, bmp_height, bmp_width);
free_dmatrix(original_b, bmp_height, bmp_width);

free_dmatrix(test_r, bmp_height, bmp_width);
free_dmatrix(test_g, bmp_height, bmp_width);
free_dmatrix(test_b, bmp_height, bmp_width);

free_dmatrix(original_y, bmp_height, bmp_width);
free_dmatrix(original_cb, bmp_height / 2, bmp_width / 2);
free_dmatrix(original_cr, bmp_height / 2, bmp_width / 2);

free_dmatrix(test_y, bmp_height, bmp_width);
free_dmatrix(test_cb, bmp_height / 2, bmp_width / 2);
free_dmatrix(test_cr, bmp_height / 2, bmp_width / 2);

m_pImageLayer->DrawBackgroundImage();
}

```

D. LoadComp (Definition)

```

void CImageProcessing::LoadComp(const char* read_path)
{
    // 먼저 헤더를 읽어 이미지의 사이즈를 받아옴
    const std::pair<int, int> image_size = MeasureImageSize(read_path);
    const int comp_height = image_size.first;
    const int comp_width = image_size.second;

    // RGB, YCbCr Allocation 진행
    double** decompressed_y = dmatrix(comp_height, comp_width);
    double** decompressed_cb = dmatrix(comp_height / 2, comp_width / 2);
    double** decompressed_cr = dmatrix(comp_height / 2, comp_width / 2);

    double** decompressed_r = dmatrix(comp_height, comp_width);
    double** decompressed_g = dmatrix(comp_height, comp_width);
    double** decompressed_b = dmatrix(comp_height, comp_width);
}

```

```

// 디코딩 및 이미지 복원 진행
// m_pImageLayer->m_ImageVec 를 Reference 로 전달하여 과정을 기록
// decompressed_y, decompressed_cb, decompressed_cr 를 값 리턴
// RGB 로 변환
DecompressImage(read_path, decompressed_y, decompressed_cb, decompressed_cr, m_pImageLayer-
>m_ImageVec, m_ResultStr,
                 m_ResultStr2);
YCbCr2RGB(decompressed_y, decompressed_cb, decompressed_cr, decompressed_r, decompressed_g,
decompressed_b,
            comp_height, comp_width);

m_pImageLayer->m_ImageVec[3].m_Red = cmatrix(m_nH, m_nW);
m_pImageLayer->m_ImageVec[3].m_Green = cmatrix(m_nH, m_nW);
m_pImageLayer->m_ImageVec[3].m_Blue = cmatrix(m_nH, m_nW);

m_pImageLayer->m_ImageVec[3].m_nH = comp_height;
m_pImageLayer->m_ImageVec[3].m_nW = comp_width;

// 복원된 이미지를 Render
for (int y = 0; y < comp_height; y++)
{
    for (int x = 0; x < comp_width; x++)
    {
        m_pImageLayer->m_ImageVec[3].m_Red[y][x] = decompressed_r[y][x];
        m_pImageLayer->m_ImageVec[3].m_Green[y][x] = decompressed_g[y][x];
        m_pImageLayer->m_ImageVec[3].m_Blue[y][x] = decompressed_b[y][x];
    }
}

m_HeaderStr = "FILE SIZE";

// 파일 사이즈 측정
const int file_size = MeasureFileSize(read_path);

char buffer[100];
sprintf(buffer, "%6.3fKB", (double)file_size / 1024);
m_ValueStr = std::string(buffer);

// Release
free_dmatrix(decompressed_y, comp_height, comp_width);
free_dmatrix(decompressed_cb, comp_height / 2, comp_width / 2);
free_dmatrix(decompressed_cr, comp_height / 2, comp_width / 2);

free_dmatrix(decompressed_r, comp_height, comp_width);
free_dmatrix(decompressed_g, comp_height, comp_width);
free_dmatrix(decompressed_b, comp_height, comp_width);

m_pImageLayer->DrawBackgroundImage();
}

```

E. CompressImage

- Visualize 관련 코드는 생략하였습니다. (핵심 코드가 아니기 때문에…)

```

inline void CompressImage(std::unique_ptr<CompResult>& res, double** InputY, double** InputCb, double** 
InputCr, int img_height, int img_width,
                         int lvl,
                         std::vector<CKhuGleSignal>& out_image_vec, std::string& str, std::string& str2)
{
    // DWT Forward 진행
    FWT2D(InputY, img_height);
    FWT2D(InputCb, img_height / 2);
    FWT2D(InputCr, img_height / 2);

    FWT2D(InputY, img_height / 2);
    FWT2D(InputCb, img_height / 4);
    FWT2D(InputCr, img_height / 4);

    // DWT Forward 를 진행한 이미지를 Visualize
    // 생략

    // Sub-Band 로 각 구역을 분할하기 위해 Factor 지정
    std::vector<std::string> map_name = {"LL1", "LH1", "HL1", "HH1", "LH", "HL", "HH"};

```

```

std::vector<std::pair<std::pair<int, int>, std::pair<int, int>>> map;
std::vector<double> factor = {2, 8, 8, 32, 32, 128, 512};

map.push_back({{0, h_qut}, {0, w_qut}});
map.push_back({{0, h_qut}, {w_qut, w_half}});
map.push_back({{h_qut, h_half}, {0, w_qut}});
map.push_back({{h_qut, h_half}, {w_qut, w_half}});
map.push_back({{0, h_half}, {w_half, w}});
map.push_back({{h_half, h}, {0, w_half}});
map.push_back({{h_half, h}, {w_half, w}});

std::vector<double> step_size_vec;
double** step_size_2d = dmatrix(h, w);

// Quantization 을 진행한 YCbCr Memory Allocation
std::vector<std::vector<int>> quantized_y(img_height, std::vector<int>(img_width));
std::vector<std::vector<int>> quantized_cb(img_height / 2, std::vector<int>(img_width / 2));
std::vector<std::vector<int>> quantized_cr(img_height / 2, std::vector<int>(img_width / 2));

// 각 Sub-Band 에 대해서 Quantization 진행
int index = 0;
for (auto sub_band : map)
{
    // Compression Level 에 따라서, Sub-Band 의 Type 에 따라서 Step size 차등 설정
    double r = lvl == 0 ? 8 : lvl == 1 ? 10 : 12;
    double i = 2;
    double c = 8;
    double f = lvl == 0 ? 23 : lvl == 1 ? 230 : 2300;

    double tau = pow(2, r - c + i) * (1 + f / pow(2, 11));

    double step_size;
    if (index == 0)
    {
        step_size = tau / pow(2, i);
    }
    else if (index == 1 || index == 2)
    {
        step_size = tau / pow(2, 2 - 1);
    }
    else if (index == 3)
    {
        step_size = tau / pow(2, 2 - 2);
    }
    else if (index == 4 || index == 5)
    {
        step_size = tau / pow(2, 1 - 1);
    }
    else if (index == 6)
    {
        step_size = tau / pow(2, 1 - 2);
    }

    step_size_vec.push_back(step_size);

    // Compression Level 이 High 이상일 경우, Quantization Level 이 Sub-Band
    // 안에서 Uniform 하지 않고, Dynamic 하게 설정되도록 함
    // Left Top = 0.5 * step_size, Right Bottom = 2 * step_size
    auto h_range = sub_band.first;
    auto w_range = sub_band.second;

    int alpha = h_range.first + w_range.first;
    int beta = h_range.second + w_range.second;
    double a = 1.5 / double(beta - alpha);
    double b = 0.5 - double(a * alpha);

    for (int y = h_range.first; y < h_range.second; y++)
    {
        for (int x = w_range.first; x < w_range.second; x++)
        {
            double mul = a * (x + y) + b;
            step_size_2d[y][x] = step_size * (lvl >= 1 ? mul : 1);

            int q = (int)std::round(InputY[y][x] / (step_size * (lvl >= 1 ? mul : 1)));
            quantized_y[y][x] = q;
        }
    }

    alpha = h_range.first / 2 + w_range.first / 2;
    beta = h_range.second / 2 + w_range.second / 2;
}

```

```

        a = 1.5 / double(beta - alpha);
        b = 0.5 - double(a * alpha);

        for (int y = h_range.first / 2; y < h_range.second / 2; y++)
        {
            for (int x = w_range.first / 2; x < w_range.second / 2; x++)
            {
                double mul = a * (x + y) + b;

                int q = (int)std::round(InputCb[y][x] / (step_size * (lvl >= 1 ? mul : 1)));
                quantized_cb[y][x] = q;

                q = (int)std::round(InputCr[y][x] / (step_size * (lvl >= 1 ? mul : 1)));
                quantized_cr[y][x] = q;
            }
        }

        index++;
    }

    // Quantization Step size 를 Visualize
    // 생략

    // Quantization 을 진행한 YCbCr 값으로 RLE pair 를 Create
    std::vector<std::pair<int, int>> rle_y = RunLengthEncoding(quantized_y);
    std::vector<std::pair<int, int>> rle_cb = RunLengthEncoding(quantized_cb);
    std::vector<std::pair<int, int>> rle_cr = RunLengthEncoding(quantized_cr);

    char buffer[100];
    sprintf(buffer, "Step Size (min, max) : %.3f, %.3f", lvl >= 1 ? min_step_size_2d : min_step_size,
           lvl >= 1 ? max_step_size_2d : max_step_size);
    str = std::string(buffer);

    sprintf(buffer, "RLE Size (Y, Cb, Cr) : %d, %d, %d", rle_y.size(), rle_cb.size(), rle_cr.size());
    str2 = std::string(buffer);

    // Huffman Tree Build
    HuffmanNode* root_y = BuildHuffmanTree(rle_y);
    HuffmanNode* root_cb = BuildHuffmanTree(rle_cb);
    HuffmanNode* root_cr = BuildHuffmanTree(rle_cr);

    // Huffman Tree 로 Quantization 된 YCbCr 값을 Encoding
    std::string encoded_data_y = EncodeWithHuffman(root_y, quantized_y);
    std::string encoded_data_cb = EncodeWithHuffman(root_cb, quantized_cb);
    std::string encoded_data_cr = EncodeWithHuffman(root_cr, quantized_cr);

    // 0-Padding 처리 진행
    int y_pad = 0;
    int cb_pad = 0;
    int cr_pad = 0;

    while (encoded_data_y.length() % 8 != 0)
    {
        encoded_data_y += '0';
        y_pad++;
    }
    while (encoded_data_cb.length() % 8 != 0)
    {
        encoded_data_cb += '0';
        cb_pad++;
    }
    while (encoded_data_cr.length() % 8 != 0)
    {
        encoded_data_cr += '0';
        cr_pad++;
    }

    // Encoded Size Visualize 진행
    // 생략

    // Header 정보 작성
    HeaderInfo info = HeaderInfo(
        encoded_data_y.length(),
        encoded_data_cb.length(),
        encoded_data_cr.length(),
        rle_y.size(),
        rle_cb.size(),

```

```

        rle_cr.size(),
        img_height, img_width,
        y_pad, cb_pad, cr_pad,
        lvl
    );
    delete step_size_2d;

    // 결과 리턴
    res = std::make_unique<CompResult>(info, encoded_data_y, encoded_data_cb, encoded_data_cr, rle_y,
rle_cb, rle_cr);
}

```

F. DecompressImage

- 마찬가지로 Visualize 하는 부분은 생략했습니다.

```

inline void DecompressImage(const CompResult* result, double** OutY, double** OutCb, double** OutCr,
                           bool visualize = false,
                           std::vector<CKhuGleSignal>& out_image_vec = std::vector<CKhuGleSignal>(),
                           std::string& str = std::string(""), std::string& str2 = std::string(""))
{
    const HeaderInfo info = result->info_;

    int h = info.h;
    int w = info.w;
    int h_half = info.h / 2;
    int w_half = info.w / 2;
    int h_qut = info.h / 4;
    int w_qut = info.w / 4;

    // Sub-Band 로 각 구역을 분할하기 위해 Factor 지정
    std::vector<std::string> map_name = {"LL1", "LH1", "HL1", "HH1", "LH", "HL", "HH"};
    std::vector<std::pair<std::pair<int, int>, std::pair<int, int>>> map;
    std::vector<double> factor = {2, 8, 8, 32, 32, 128, 512};

    map.push_back({{0, h_qut}, {0, w_qut}});
    map.push_back({{0, h_qut}, {w_qut, w_half}});
    map.push_back({{h_qut, h_half}, {0, w_qut}});
    map.push_back({{h_qut, h_half}, {w_qut, w_half}});
    map.push_back({{0, h_half}, {w_half, w}});
    map.push_back({{h_half, h}, {0, w_half}});
    map.push_back({{h_half, h}, {w_half, w}});

    std::string r_encoded_data_y = result->encoded_data_y_;
    std::string r_encoded_data_cb = result->encoded_data_cb_;
    std::string r_encoded_data_cr = result->encoded_data_cr_;

    // 인코딩 데이터의 0-Padding 제거
    r_encoded_data_y.erase(r_encoded_data_y.length() - info.y_pad, info.y_pad);
    r_encoded_data_cb.erase(r_encoded_data_cb.length() - info.cb_pad, info.cb_pad);
    r_encoded_data_cr.erase(r_encoded_data_cr.length() - info.cr_pad, info.cr_pad);

    std::vector<std::pair<int, int>> r_rle_y = result->rle_y_;
    std::vector<std::pair<int, int>> r_rle_cb = result->rle_cb_;
    std::vector<std::pair<int, int>> r_rle_cr = result->rle_cr_;

    // RLE pair 데이터를 기반으로 Huffman Tree Re-Build
    HuffmanNode* r_root_y = BuildHuffmanTree(r_rle_y);
    HuffmanNode* r_root_cb = BuildHuffmanTree(r_rle_cb);
    HuffmanNode* r_root_cr = BuildHuffmanTree(r_rle_cr);

    // Re-Build 한 Huffman Tree 를 바탕으로 Decoding 진행
    std::vector<std::vector<int>> decoded_data_y = DecodeWithHuffman(r_encoded_data_y, r_root_y,
info.w);
    std::vector<std::vector<int>> decoded_data_cb = DecodeWithHuffman(r_encoded_data_cb, r_root_cb,
info.w / 2);
    std::vector<std::vector<int>> decoded_data_cr = DecodeWithHuffman(r_encoded_data_cr, r_root_cr,
info.w / 2);

    std::vector<double> step_size_vec;
    double** step_size_2d = dmatrix(h, w);

    // 각 Sub-Band 에 대해서 De-Quantization 진행
    int index = 0;
    for (auto sub_band : map)

```

```

{
    // Compression Level 에 따라서, Sub-Band 의 Type 에 따라서 Step size 차등 설정
    // 아래 과정은 Compression 과 동일한 값 사용
    double r = info.lvl == 0 ? 8 : info.lvl == 1 ? 10 : 12;
    double i = 2;
    double c = 8;
    double f = info.lvl == 0 ? 23 : info.lvl == 1 ? 230 : 2300;

    double tau = pow(2, r - c + i) * (1 + f / pow(2, 11));

    double step_size;
    if (index == 0)
    {
        step_size = tau / pow(2, i);
    }
    else if (index == 1 || index == 2)
    {
        step_size = tau / pow(2, 2 - 1);
    }
    else if (index == 3)
    {
        step_size = tau / pow(2, 2 - 2);
    }
    else if (index == 4 || index == 5)
    {
        step_size = tau / pow(2, 1 - 1);
    }
    else if (index == 6)
    {
        step_size = tau / pow(2, 1 - 2);
    }

    step_size_vec.push_back(step_size);

    // Compression Level 이 High 이상일 경우, Quantization Level 이 Sub-Band
    // 안에서 Uniform 하지 않고, Dynamic하게 설정되도록 함
    // Left Top = 0.5 * step_size, Right Bottom = 2 * step_size
    auto h_range = sub_band.first;
    auto w_range = sub_band.second;

    int alpha = h_range.first + w_range.first;
    int beta = h_range.second + w_range.second;
    double a = 1.5 / double(beta - alpha);
    double b = 0.5 - double(a * alpha);

    for (int y = h_range.first; y < h_range.second; y++)
    {
        for (int x = w_range.first; x < w_range.second; x++)
        {
            double mul = a * (x + y) + b;
            step_size_2d[y][x] = step_size * (info.lvl >= 1 ? mul : 1);

            OutY[y][x] = decoded_data_y[y][x] * (step_size * (info.lvl >= 1 ? mul : 1));
        }
    }

    alpha = h_range.first / 2 + w_range.first / 2;
    beta = h_range.second / 2 + w_range.second / 2;
    a = 1.5 / double(beta - alpha);
    b = 0.5 - double(a * alpha);

    for (int y = h_range.first / 2; y < h_range.second / 2; y++)
    {
        for (int x = w_range.first / 2; x < w_range.second / 2; x++)
        {
            double mul = a * (x + y) + b;
            OutCb[y][x] = decoded_data_cb[y][x] * (step_size * (info.lvl >= 1 ? mul : 1));
            OutCr[y][x] = decoded_data_cr[y][x] * (step_size * (info.lvl >= 1 ? mul : 1));
        }
    }

    index++;
}

// Inverse Discrete Wavelet Transform 진행
IWT2D(OutY, h_half);

```

```
IWT2D(OutCb, h_qut);
IWT2D(OutCr, h_qut);

IWT2D(OutY, h);
IWT2D(OutCb, h_half);
IWT2D(OutCr, h_half);

} // delete step_size_2d;
```

4. 결과

A. Cat.bmp

Compression process (Normal Compression Level)

Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	
40.751	0.994	53.167KB (27.598 % of Original)	Step Size (min, max) : 1.011, 8.090 RLE Size (Y, Cb, Cr) : 1068, 182, 182

Compression process (High Compression Level)

Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	
32.817	0.961	27.473KB (14.261 % of Original)	Step Size (min, max) : 2.225, 70.770 RLE Size (Y, Cb, Cr) : 332, 50, 60

Compression process (Extreme Compression Level)

Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	
25.179	0.798	15.961KB (8.285 % of Original)	Step Size (min, max) : 16.984, 540.315 RLE Size (Y, Cb, Cr) : 51, 11, 9

	PSNR	SSIM	File Size	% of Original
Normal	40.751	0.994	53.167 KB	27.598 %
High	32.817	0.961	27.473 KB	14.261 %
Extreme	25.179	0.798	15.961 KB	8.285 %

High Level 까지는 SSIM 이 0.95 이상이므로 원본과의 차이가 거의 없었으나, Extreme Level 에서는 0.798 로 복원된 이미지가 조금 깨지는 것을 확인할 수 있습니다.

B. Couple.bmp

Compression process (Normal Compression Level)

Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	
33.737	0.924	41.927KB (21.831 % of Original)	Step Size (min, max) : 1.011, 8.090 RLE Size (Y, Cb, Cr) : 559, 88, 126

Compression process (High Compression Level)

Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	
31.823	0.865	21.579KB (11.236 % of Original)	Step Size (min, max) : 2.225, 70.770 RLE Size (Y, Cb, Cr) : 189, 25, 35

Compression process (Extreme Compression Level)

Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

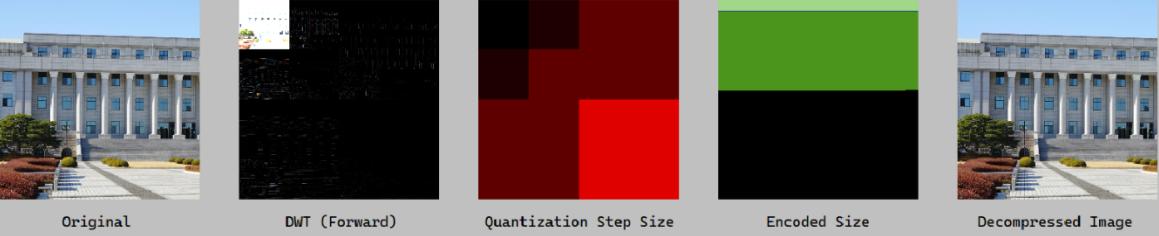
PSNR	SSIM	FILE SIZE	
27.730	0.702	14.184KB (7.385 % of Original)	Step Size (min, max) : 16.984, 540.315 RLE Size (Y, Cb, Cr) : 30, 5, 7

	PSNR	SSIM	File Size	% of Original
Normal	33.737	0.924	41.927 KB	21.831 %
High	31.823	0.865	21.579 KB	11.236 %
Extreme	27.730	0.702	14.184 KB	7.385 %

위의 Cat.bmp 와는 다르게 Extreme Level에서 SSIM이 0.702 까지 내려갔기 때문에 이미지가 깨지는 것을 좀 더 명확히 확인할 수 있습니다. Cat.bmp 에서의 RLE Size 는 (51, 11, 9) 였던 것에 반해, 이 이미지에서는 (30, 5, 7)로 Quantization 이 더 가파르게 일어났다는 것을 확인할 수 있습니다.

C. Library.bmp (512 x 512)

Compression process (Normal Compression Level)

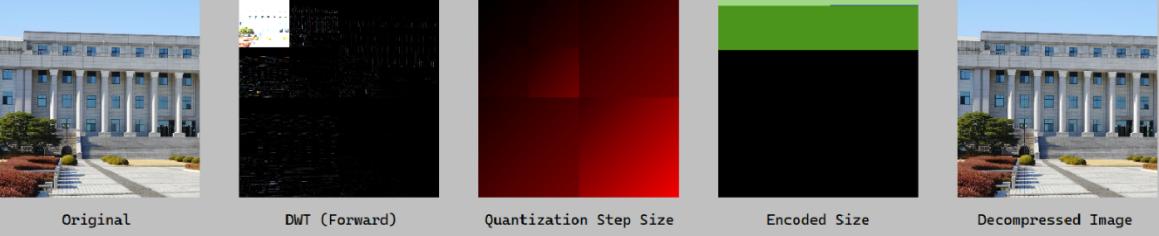


Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	Step Size (min, max)	RLE Size (Y, Cb, Cr)
39.628	0.991	160.818KB (20.938 % of Original)	1.011, 8.090	1072, 267, 261

Compression process (High Compression Level)



Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	Step Size (min, max)	RLE Size (Y, Cb, Cr)
33.759	0.953	94.679KB (12.327 % of Original)	2.225, 70.979	374, 88, 73

Compression process (Extreme Compression Level)



Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	Step Size (min, max)	RLE Size (Y, Cb, Cr)
25.874	0.754	62.127KB (8.089 % of Original)	16.984, 541.908	51, 13, 11

	PSNR	SSIM	File Size	% of Original
Normal	39.628	0.991	160.818 KB	20.938 %
High	33.759	0.953	94.679 KB	12.327 %
Extreme	25.874	0.754	62.127 KB	8.089 %

512 x 512 이미지로도 진행해 보았습니다. 위의 결과와 크게 다르지 않은 것을 확인할 수 있습니다.

D. Camera.bmp (Grayscale)

Compression process (Normal Compression Level)

Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	
45.232	0.988	34.679KB (18.057 % of Original)	Step Size (min, max) : 1.011, 8.090 RLE Size (Y, Cb, Cr) : 802, 1, 1

Compression process (High Compression Level)

Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	
34.780	0.947	18.711KB (9.743 % of Original)	Step Size (min, max) : 2.225, 70.770 RLE Size (Y, Cb, Cr) : 291, 1, 1

Compression process (Extreme Compression Level)

Original DWT (Forward) Quantization Step Size Encoded Size Decompressed Image

Result

PSNR	SSIM	FILE SIZE	
26.299	0.785	11.232KB (5.849 % of Original)	Step Size (min, max) : 16.984, 540.315 RLE Size (Y, Cb, Cr) : 41, 1, 1

	PSNR	SSIM	File Size	% of Original
Normal	45.232	0.988	34.679 KB	18.057 %
High	34.780	0.947	18.711 KB	9.743 %
Extreme	26.299	0.785	11.232 KB	5.849 %

이 이미지는 Grayscale 이미지로 확실히 위 두 이미지보다 압축률이 높은 것을 확인할 수 있습니다. 다만 위 두 이미지와 비슷하게 High Level 까지는 원본과의 차이를 찾아보기 힘드나, Extreme Level 에서는 배경에 Stripe 가 생기는 것을 확인할 수 있습니다.

E. 결론

세 이미지 모두 High Level 까지는 SSIM 값이 0.9 ~ 0.95 정도 되기 때문에 인간의 눈으로 구분하기 어려운 정도의 손실을 보였으나, Extreme Level 에서는 0.7 ~ 0.8 정도로 이미지가 깨져 보이거나 줄무늬가 생기는 등의 손실이 확실히 보였습니다.

다만 그렇기에 이미지의 사이즈는 원본 이미지의 5 ~ 8 % 정도로 원본 이미지의 품질을 낮추어서라도 이미지의 크기를 줄여야 할 때 Extreme Level 을 사용하면 유용할 것으로 보입니다.

이외의 경우, 즉 일반적인 경우에는 High Level 도 9 ~ 14 % 정도의 압축률을 보이기 때문에 High Level 를 사용하는 것이 좋을 것으로 보입니다.