Assignment 1: Transform Coding Extra

CS 4600 Computer Graphics Fall 2018

Rui Ying u1234364

1. Audio Coding (Haar Wavelet)
   a. Haar Transform

   According to Wikipedia[1], Haar Wavelet Transform is one transform like DCT but using different bases.

   We can use the following pattern[2] to calculate 1D Haar Transform.

      i. Treat the array as n/2 pairs called (a, b).
      ii. Calculate (a + b) / sqrt(2) for each pair, these values will be the first half of the output array.
      iii. Calculate (a - b) / sqrt(2) for each pair, these values will be the second half.
      iv. Repeat the process on the first half of the array until there's only one element.

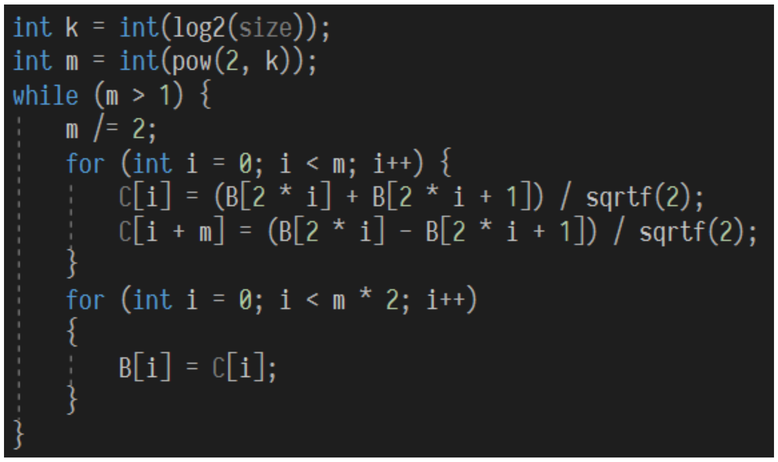
   Usually the array length is the power of 2, so we do not have to worry about that. If it isn't, we could just ignore the rest of data.

   First, get to know how many recursions we need. Variable m is calculated to keep track of the recursion.

   Then, use the pattern to calculate next recursion of array values.

   Last, use the calculated array as the new source array.

   Code snippet



```
int k = int(log2(size));
int m = int(pow(2, k));
while (m > 1) {
    m /= 2;
    for (int i = 0; i < m; i++) {
        C[i] = (B[2 * i] + B[2 * i + 1]) / sqrtf(2);
        C[i + m] = (B[2 * i] - B[2 * i + 1]) / sqrtf(2);
    }
    for (int i = 0; i < m * 2; i++)
    {
        B[i] = C[i];
    }
}
```

   b. Compress

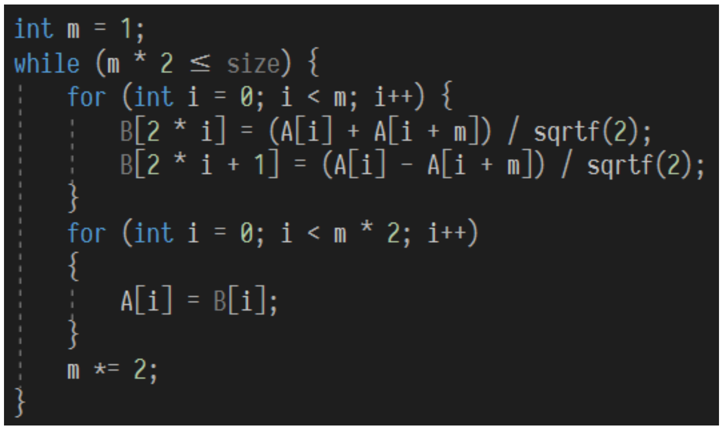
   Same as DCT

   c. Inverse Haar Transform

   While dividing m by 2 each recursion in Haar Transform, we multiply m by 2 each recursion to inverse the transform.

And the indices will be reversed too.

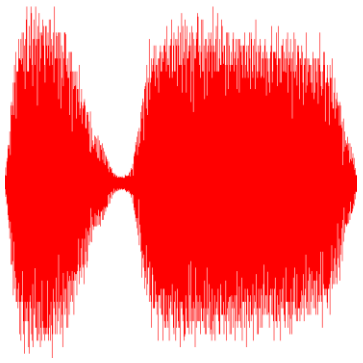
Code snippet



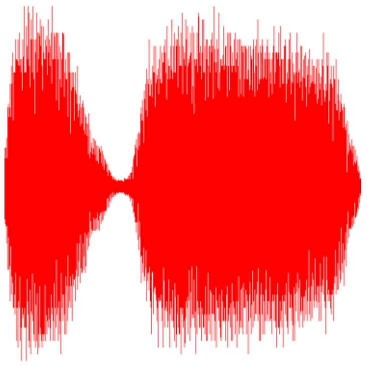
```
int m = 1;
while (m * 2 ≤ size) {
    for (int i = 0; i < m; i++) {
        B[2 * i] = (A[i] + A[i + m]) / sqrtf(2);
        B[2 * i + 1] = (A[i] - A[i + m]) / sqrtf(2);
    }
    for (int i = 0; i < m * 2; i++)
    {
        A[i] = B[i];
    }
    m *= 2;
}
```

d. result

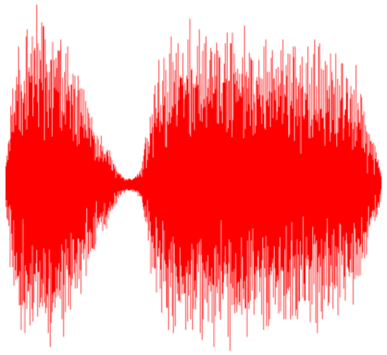
Original sound (m=0)



m=1



m=5



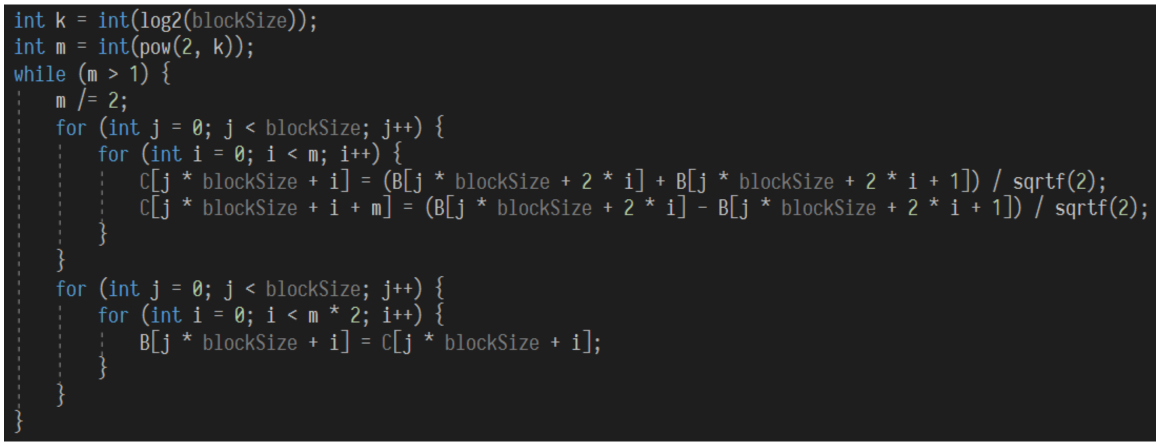
Compared to DCT compression result, there is less crispy noise.

2. Image Coding (Haar Transform)
    a. 2D Haar Transform

We can perform a 2D Haar transform by first performing a 1D Haar transform on each row and then on each column[2].

Because the input array is 1D, we still need to convert 2D indices into 1D indexing.

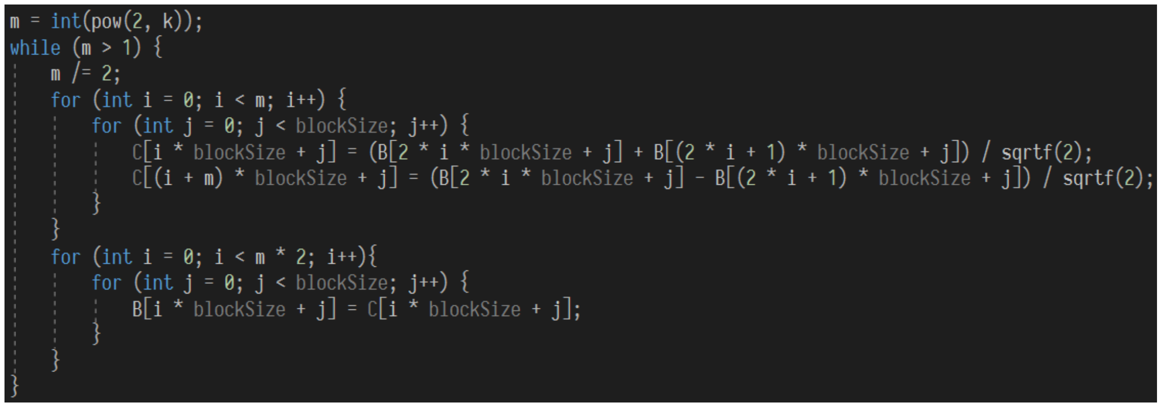
Code snippet



```
int k = int(log2(blockSize));
int m = int(pow(2, k));
while (m > 1) {
    m /= 2;
    for (int j = 0; j < blockSize; j++) {
        for (int i = 0; i < m; i++) {
            C[j * blockSize + i] = (B[j * blockSize + 2 * i] + B[j * blockSize + 2 * i + 1]) / sqrtf(2);
            C[j * blockSize + i + m] = (B[j * blockSize + 2 * i] - B[j * blockSize + 2 * i + 1]) / sqrtf(2);
        }
    }
    for (int j = 0; j < blockSize; j++) {
        for (int i = 0; i < m * 2; i++) {
            B[j * blockSize + i] = C[j * blockSize + i];
        }
    }
}
```

j is used to index the row, and i is used to index through a row, performing 1D Haar Transform on that row. After the for loop, all rows have been 1D transformed.

Code snippet



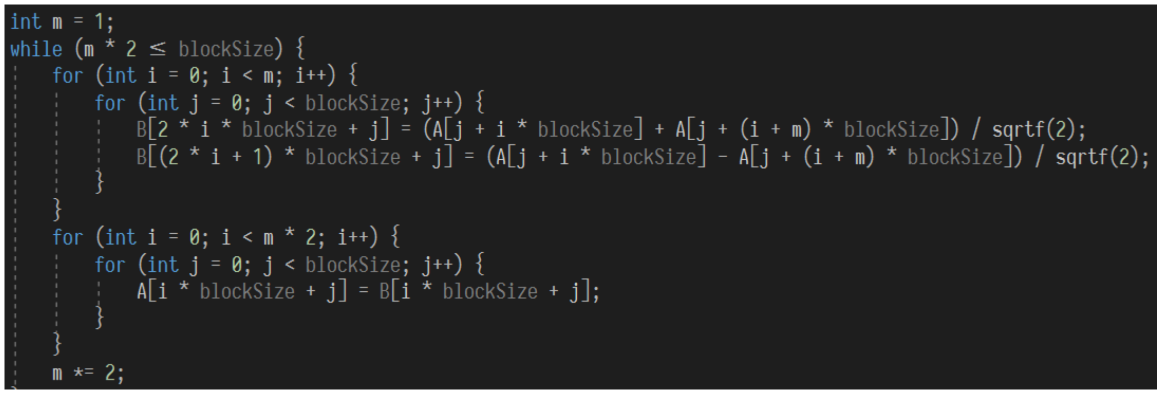
```
m = int(pow(2, k));
while (m > 1) {
    m /= 2;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < blockSize; j++) {
            C[i * blockSize + j] = (B[2 * i * blockSize + j] + B[(2 * i + 1) * blockSize + j]) / sqrtf(2);
            C[(i + m) * blockSize + j] = (B[2 * i * blockSize + j] - B[(2 * i + 1) * blockSize + j]) / sqrtf(2);
        }
    }
    for (int i = 0; i < m * 2; i++){
        for (int j = 0; j < blockSize; j++) {
            B[i * blockSize + j] = C[i * blockSize + j];
        }
    }
}
```

Now we perform 1D transform on each column. j is the column index.

b. Compress
Same as DCT

c. Inverse 2D Haar Transform
Same as inverseDCT, m should be multiplied by 2 each recursion and the indices of the source and destination array should be exchanged to perform a reversed procedure.

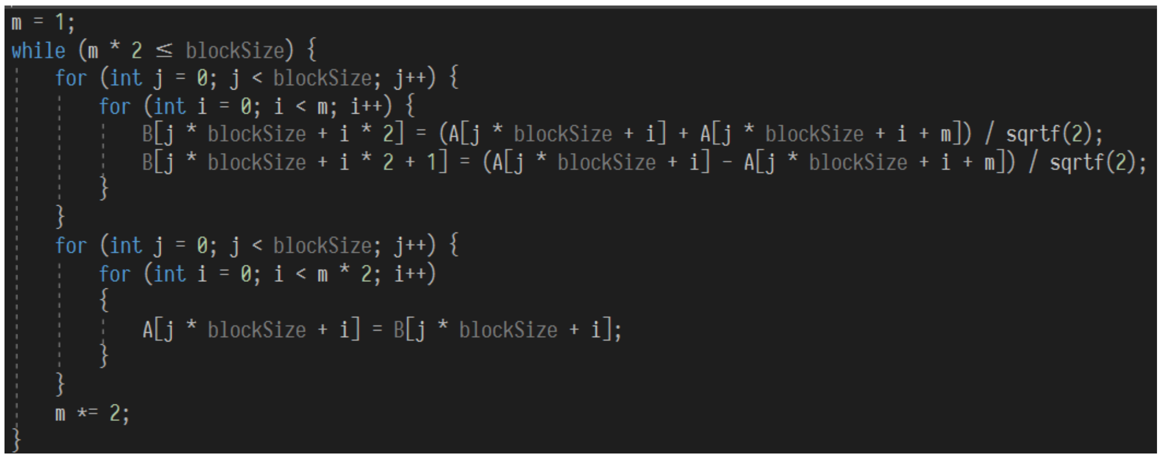
Code snippet



```
int m = 1;
while (m * 2 <= blockSize) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < blockSize; j++) {
            B[2 * i * blockSize + j] = (A[j + i * blockSize] + A[j + (i + m) * blockSize]) / sqrtf(2);
            B[(2 * i + 1) * blockSize + j] = (A[j + i * blockSize] - A[j + (i + m) * blockSize]) / sqrtf(2);
        }
    }
    for (int i = 0; i < m * 2; i++) {
        for (int j = 0; j < blockSize; j++) {
            A[i * blockSize + j] = B[i * blockSize + j];
        }
    }
    m *= 2;
```

Inverse Haar Transform on each row.

Code snippet



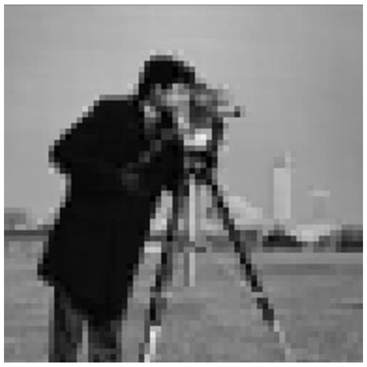
```
m = 1;
while (m * 2 ≤ blockSize) {
    for (int j = 0; j < blockSize; j++) {
        for (int i = 0; i < m; i++) {
            B[j * blockSize + i * 2] = (A[j * blockSize + i] + A[j * blockSize + i + m]) / sqrtf(2);
            B[j * blockSize + i * 2 + 1] = (A[j * blockSize + i] - A[j * blockSize + i + m]) / sqrtf(2);
        }
    }
    for (int j = 0; j < blockSize; j++) {
        for (int i = 0; i < m * 2; i++)
        {
            A[j * blockSize + i] = B[j * blockSize + i];
        }
    }
    m *= 2;
}
```

Inverse Haar Transform on each column.

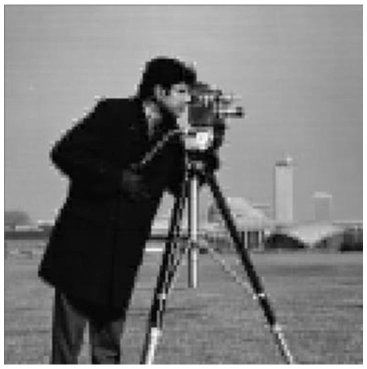
d. Result

Original photo

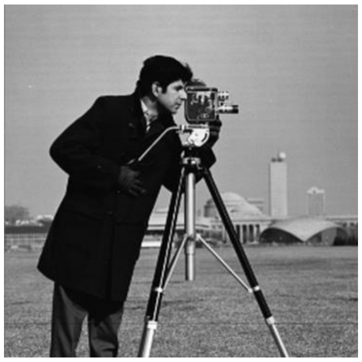
m=1



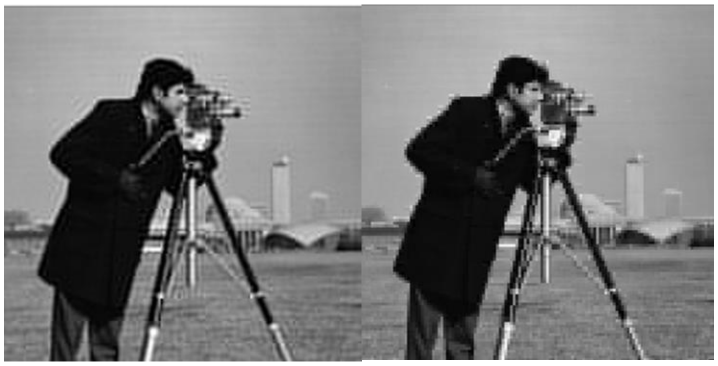
m=3



m=16

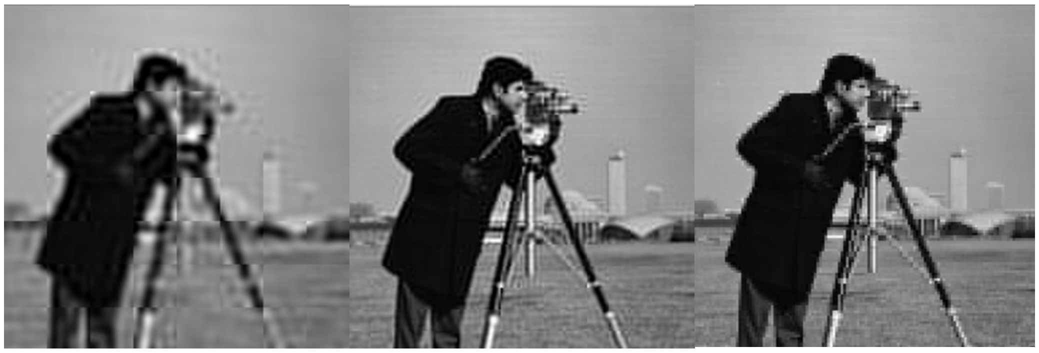


3. Larger compression block
   16 x 16 blocks with m = 7, nearly the same compression rate as m=3 with 8 x 8 blocks. Larger blocks seem to produce better quality with same compression rate. Maybe part of the reason is the photo becomes more smooth with larger blocks.



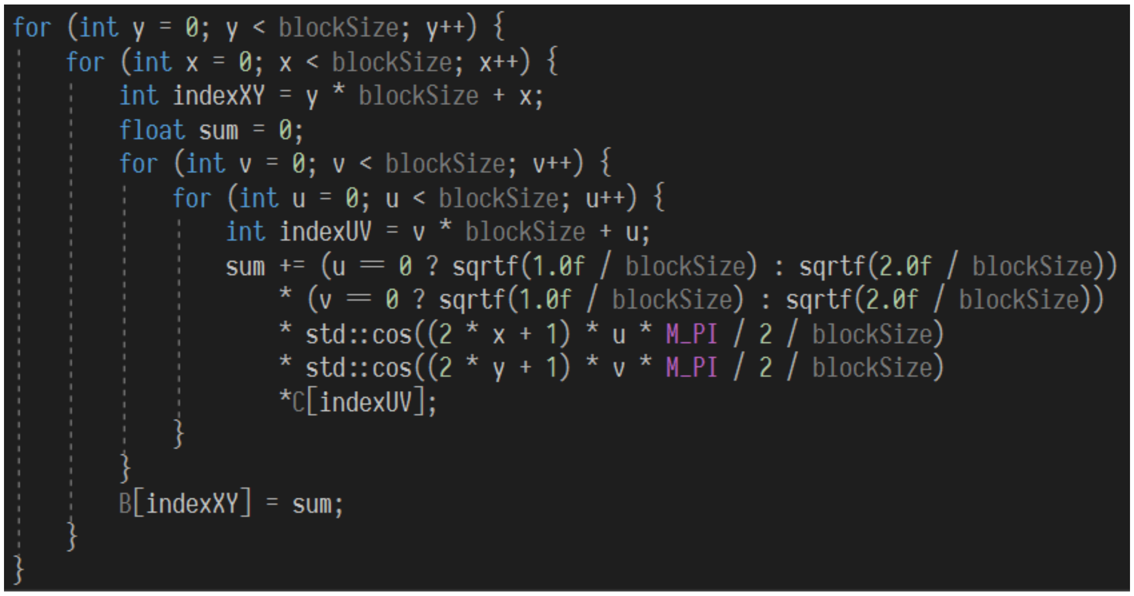
Left: 16x16   Right: 8x8

32 x 32 blocks with m = 16, nearly the same compression rate as m=3 with 8 x 8 blocks.


Left: 32x32 middle: 16x16 Right: 8x8

However, when blocks grow bigger, the photo becomes blurred and can easily find these blocks existing.
In general, the photo looks better with 16x16 blocks.
Larger blocks may seem helpful but the final result really depends on how big the block is.

Bug solved
At first, when I try to alter 8x8 to 16x16, the output becomes somewhat whitening. It turned out that the coefficients of DCT is not fixed but determined by the block size.

Code snippet

```
for (int y = 0; y < blockSize; y++) {
    for (int x = 0; x < blockSize; x++) {
        int indexXY = y * blockSize + x;
        float sum = 0;
        for (int v = 0; v < blockSize; v++) {
            for (int u = 0; u < blockSize; u++) {
                int indexUV = v * blockSize + u;
                sum += (u == 0 ? sqrtf(1.0f / blockSize) : sqrtf(2.0f / blockSize))
                    * (v == 0 ? sqrtf(1.0f / blockSize) : sqrtf(2.0f / blockSize))
                    * std::cos((2 * x + 1) * u * M_PI / 2 / blockSize)
                    * std::cos((2 * y + 1) * v * M_PI / 2 / blockSize)
                    *C[indexUV];
            }
        }
        B[indexXY] = sum;
    }
}
```

4. References

[1] Haar Wavelet Transform (https://unix4lyfe.org/haar/)
[2] Haar Wavelet (https://en.wikipedia.org/wiki/Haar_wavelet#Haar_transform)