Half Size Code Test

Stefan Björnander

This report describes an application for converting a file in the TGA image file format to a file in the same format holding the image in half its size. There are four combinations of the file format depending on whether the image is *mapped* or *compressed*.

If the image is mapped, the image data is divided into two parts. The first part is the *image map*, which is a sequence of pixels. The second part is a sequence of indices referring to the pixels of the image map. If the image is not mapped, the image data is just a sequence of pixels.

If the image is compressed the indices (if the image is mapped) or the pixels (if the image is unmapped) is divided into *run-length* blocks or *raw blocks*. In the run-length case, the block starts with a byte indicating the number of pixels in the block. However, it is assumed that the block holds only one kind of pixel. Therefore, there is only one pixel, representing all the pixels in the block. However, in the raw block case the pixels are different. Therefore, they are all stored in the block. If the highest bit in the first byte of the block is one, the block is a run-length block. If the bit is zero, the block is a raw block.

When converting the image to an equivalent image of half the size, we divide the original image into groups of four pixels. For each such group, we calculate their average color. Then we build the image of half the size with the pixels of the average colors. Since both the width and height of the new half-size image is half of the width and height of the original image, its number of pixels is a quarter of the number of pixels of the original image.

When calculating an average pixel in a mapped image, the case may occur that it is not present in the image map. In that case, it need to be added to the map. Moreover, in the new half-size image there may be pixels in the image map that are not indexed in the image data. In that case, they can be removed from the image map in order to increase the file size.

To accomplish effective execution, I use the *map* and *vector* standard classes, which provide fast access to their values. I have also written the class *Matrix*, which stores the pixels and indices of the images and provide fast access to its values. To accomplish effective memory management, I dynamically allocate memory blocks, which are always deleted.

The code is divided into the following parts:

Header	Each image file starts with a *header*, holding information about the size of the image, and whether it is mapped or compressed.
Pixel	Holds a pixel of 8, 16, 24, or 32 bits. The 8-bits case is intended for black-and-white images. In the 16 bits case, each color of the pixel is stored in 5 bits.
Index	Holds the index of a pixel in the image map. The index size can be 8, 16, 24, or 32 bits.
Matrix	A template class holding the pixels or the indices of the pixels that loads and saves the image data in compressed or uncompressed form.
Block	A template class used by the matrix when saving compressed image data.
Main	The main code that loads, halves, optimizes, and saves the image file.

The *Pixel* class holds a pixel of 8, 16, 24, or 32 bits. The color of a pixel is divided into its red, green, and blue components, and there is also the alpha channel value.

Pixel.h

```
class Pixel {
  public:
    Pixel(int bitsPerPixel = 0);
    Pixel(const Pixel& pixel);
    Pixel& operator=(const Pixel& pixel);
    bool operator==(const Pixel& pixel) const;
    bool operator<(const Pixel& pixel) const;
    void load(int bitsPerPixelOrIndex, ifstream& inStream);
    void save(ofstream& outStream) const;
```

The *average* function calculates the average value of two pixels by calculating the average of its red, green, blue, and alpha components.

```
    friend Pixel average(const Pixel& leftPixel, const Pixel& rightPixel);
  public:
    int m_bitsPerPixel;
    char m_red, m_green, m_blue, m_alpha;
};
```

Pixel.cpp

```
#include <fstream>
#include <cassert>
using namespace std;
#include "Pixel.h"

Pixel::Pixel(int bitsPerPixel /* = 0 */)
 :m_bitsPerPixel(bitsPerPixel),
  m_red(0),
  m_green(0),
  m_blue(0),
  m_alpha(0) {
  // Empty.
}

Pixel::Pixel(const Pixel& pixel)
 :m_bitsPerPixel(pixel.m_bitsPerPixel),
  m_red(pixel.m_red),
  m_green(pixel.m_green),
  m_blue(pixel.m_blue),
  m_alpha(pixel.m_alpha) {
 // Empty.
}

Pixel& Pixel::operator=(const Pixel& pixel) {
  m_bitsPerPixel = pixel.m_bitsPerPixel;
  m_red = pixel.m_red;
  m_green = pixel.m_green;
  m_blue = pixel.m_blue;
  m_alpha = pixel.m_alpha;
  return *this;
}

bool Pixel::operator==(const Pixel& pixel) const {
  assert(m_bitsPerPixel == pixel.m_bitsPerPixel);
  return (m_red == pixel.m_red) &&
         (m_green == pixel.m_green) &&
         (m_blue == pixel.m_blue) &&
         (m_alpha == pixel.m_alpha);
}

bool Pixel::operator<(const Pixel& pixel) const {
  assert(m_bitsPerPixel == pixel.m_bitsPerPixel);
  int thisWord = (((int) m_alpha) << 24) |
                  (((int) m_blue) << 16) |
                  (((int) m_green) << 8) |
                   ((int) m_red),
                pixelWord = (((int) pixel.m_alpha) << 24) |
                            (((int) pixel.m_blue) << 16) |
                            (((int) pixel.m_green) << 8) |
                             ((int) pixel.m_red);
  return (thisWord < pixelWord);
}
```

In case of 8 bits, the color is often the gray-scale average of the corresponding color. Its value is stored in the alpha field.

```
void Pixel::load(int bitsPerPixelOrIndex, ifstream& inStream) {
  switch (m_bitsPerPixel = bitsPerPixelOrIndex) {
    case 8:
      inStream.read((char*) &m_alpha, sizeof m_alpha);
      break;
```

In case of 16 bits, the word is composed by its color components, where the red, green, and blue components are stored in five bits each. The alpha component is stored in one bit.

```
    case 16: {
        short int word;
        inStream.read((char*) &word, sizeof word);

        m_red = (char) (word & 0x001F);
        m_green = (char) ((word >> 5) & 0x001F);
        m_blue = (char) ((word >> 10) & 0x001F);
        m_alpha = (char) ((word >> 15) & 0x0001);
      }
      break;

    case 24:
      inStream.read((char*) &m_red, sizeof m_red);
      inStream.read((char*) &m_green, sizeof m_blue);
      inStream.read((char*) &m_blue, sizeof m_green);
      m_alpha = 0;
      break;

    case 32:
      inStream.read((char*) &m_red, sizeof m_red);
      inStream.read((char*) &m_green, sizeof m_blue);
      inStream.read((char*) &m_blue, sizeof m_green);
      inStream.read((char*) &m_alpha, sizeof m_alpha);
      break;
  }
}

void Pixel::save(ofstream& outStream) const {
  switch (m_bitsPerPixel) {
    case 8:
      outStream.write((char*) &m_alpha, sizeof m_alpha);
      break;

    case 16: {
        short int word = (((short int) m_alpha) << 15) |
                         (((short int) m_blue) << 10) |
                         (((short int) m_green) << 5) |
                          ((short int) m_red);
        outStream.write((char*) &word, sizeof word);
      }
      break;
    case 24:
      outStream.write((char*) &m_red, sizeof m_red);
      outStream.write((char*) &m_green, sizeof m_blue);
      outStream.write((char*) &m_blue, sizeof m_green);
      break;

    case 32:
      outStream.write((char*) &m_red, sizeof m_red);
      outStream.write((char*) &m_green, sizeof m_blue);
      outStream.write((char*) &m_blue, sizeof m_green);
      outStream.write((char*) &m_alpha, sizeof m_alpha);
      break;
  }
}
```

The *average* function calculates the average of two pixels by calculating the average of its color components.

```
Pixel average(const Pixel& leftPixel, const Pixel& rightPixel) {
  assert(leftPixel.m_bitsPerPixel == rightPixel.m_bitsPerPixel);
  Pixel resultPixel;
  resultPixel.m_bitsPerPixel = leftPixel.m_bitsPerPixel;

  switch (resultPixel.m_bitsPerPixel) {
    case 8:
      resultPixel.m_alpha = (leftPixel.m_alpha + rightPixel.m_alpha) / 2;
      break;

    case 16:
    case 24:
    case 32:
      resultPixel.m_red = (leftPixel.m_red + rightPixel.m_red) / 2;
      resultPixel.m_green = (leftPixel.m_green + rightPixel.m_green) / 2;
      resultPixel.m_blue = (leftPixel.m_blue + rightPixel.m_blue) / 2;
      resultPixel.m_alpha = (leftPixel.m_alpha + rightPixel.m_alpha) / 2;
      break;
  }

  return resultPixel;
}
```

The *Index* class holds an index referring to a pixel in the image map, in case of a mapped image. Basically, it just encapsulates an integer value, which is loaded and saved.

Index.h

```
class Index {
  public:
    Index(void);
    Index(int bitsPerIndex, int index);
    Index(const Index& index);
    Index& operator=(const Index& index);

    int index() const {return m_index;}
    bool operator==(const Index& index) const;

    void load(int bitsPerIndex, ifstream& inStream);
    void save(ofstream& outStream) const;

  private:
    int m_bitsPerIndex, m_index;
};
```

Index.cpp

```
#include <fstream>
#include <cassert>
using namespace std;

#include "Index.h"

Index::Index(void)
 :m_bitsPerIndex(0),
  m_index(0) {
  // Empty.
}

Index::Index(int bitsPerIndex, int index)
 :m_bitsPerIndex(bitsPerIndex),
  m_index(index) {
  // Empty.
}

Index::Index(const Index& index)
 :m_bitsPerIndex(index.m_bitsPerIndex),
  m_index(index.m_index) {
  // Empty.
}

Index& Index::operator=(const Index& index) {
  m_bitsPerIndex = index.m_bitsPerIndex;
  m_index = index.m_index;
  return *this;
}

bool Index::operator==(const Index& index) const {
  assert(m_bitsPerIndex == index.m_bitsPerIndex);
  return (m_index == index.m_index);
}

void Index::load(int bitsPerIndex, ifstream& inStream) {
  m_index = 0;
  m_bitsPerIndex = bitsPerIndex;
  inStream.read((char*) &m_index, m_bitsPerIndex / 8);
}

void Index::save(ofstream& outStream) const {
  outStream.write((char*) &m_index, m_bitsPerIndex / 8);
}
```

Block is a template class used by the *Matrix* class below that keeps track of the raw blocks of a compressed image. It holds the number of repetitions and a list of pixels.

Block.h

```
template <class Type>
class Block {
  public:
    Block();
    Block(int repetitions, const Type& value);

    int getRepetitions() const { return m_repetitions; }
    void setRepetitions(int repetitions) { m_repetitions = repetitions; }

    vector<Type>& valueList() {return m_valueList;}
  private:
    int m_repetitions;
    vector<Type> m_valueList;
};

template <class Type>
Block<Type>::Block()
  :m_repetitions(0) {
  // Empty.
}

template <class Type>
Block<Type>::Block(int repetitions, const Type& value)
  : m_repetitions(repetitions) {
  m_valueList.push_back(value);
}
```

Matrix is a template class holding the pixels (in unmapped images) of indices (in mapped images) of the image. It has a width and a height, and it can be compressed or uncompressed.

Matrix.h

```
template <class Type>
class Matrix {
  public:
    Matrix(int width, int height, bool compressed);
    ~Matrix();

    int width() const { return m_width; }
    int height() const { return m_height; }

    Type get(int row, int column) const;
    void set(int row, int column, const Type& value);

    void load(int numberOfBits, ifstream& inStream);
    void save(ofstream& outStream);

  private:
    int m_width, m_height;
    bool m_compressed;
    Type* m_buffer;
};

template <class Type>
Matrix<Type>::Matrix(int width, int height, bool compressed)
 :m_width(width),
  m_height(height),
  m_compressed(compressed) {
  assert((m_width > 0) && (m_height > 0));
  m_buffer = new Type[m_width * m_height];
  assert(m_buffer != nullptr);
}

template <class Type>
Matrix<Type>::~Matrix() {
  delete [] m_buffer;
}

template <class Type>
Type Matrix<Type>::get(int row, int column) const {
  return m_buffer[(row * m_width) + column];
}
```

```
template <class Type>
void Matrix<Type>::set(int row, int column, const Type& value) {
  m_buffer[(row * m_width) + column] = value;
}
```

It is rather easy to read an uncompressed file. We just call the *load* methods of each generic value. The value will be a pixel or an index when *Matrix* becomes instantiated by a type.

```
template <class Type>
void Matrix<Type>::load(int numberOfBits, ifstream& inStream) {
  if (!m_compressed) {
    for (int row = 0; row < m_height; ++row) {
      for (int column = 0; column < m_width; ++column) {
        Type value;
        value.load(numberOfBits, inStream);
        set(row, column, value);
      }
    }
  }
```

It is more complicated to read a compressed file. The idea is that we read one run-length block or raw block at the time, until every value (pixel or index) has been read. The *index* variable keeps track of the number of pixels read so far.

```
  else {
    int size = m_width * m_height, index = 0;

    while (index < size) {
      char repetitionCount;
      inStream.read(&repetitionCount, sizeof repetitionCount);
```

For each block, we check whether it is a run-length block (the top bit is one) or a raw block (the top bit is zero).

```
      bool runLength = ((repetitionCount & 0x80) != 0);
      int repetitions = ((int) (repetitionCount & 0x7F)) + 1;
```

In case of a run-length block, we read its only pixel and add it to the matrix as many times as the start byte of the block indicates. We use the *index* variable to calculate the row and column of each value.

```
      if (runLength) {
        Type value;
        value.load(numberOfBits, inStream);

        for (int count = 0; count < repetitions; ++count) {
          int row = index / m_width, column = index % m_width;
          set(row, column, value);
          ++index;
        }
      }
```

In case of a raw block, we read and add all its pixels to the matrix.

```
      else {
        for (int count = 0; count < repetitions; ++count) {
          Type value;
          value.load(numberOfBits, inStream);
          int row = index / m_width, column = index % m_width;
          set(row, column, value);
          ++index;
        }
      }
    }
```

```
  }
}
```

In the same way, it is rather easy to save the matrix in the uncompressed case. We just call the *save* method on every value.

```
template <class Type>
void Matrix<Type>::save(ofstream& outStream) {
  if (!m_compressed) {
    for (int row = 0; row < m_height; ++row) {
      for (int column = 0; column < m_width; ++column) {
        get(row, column).save(outStream);
      }
    }
  }
```

In the compressed case, it is harder to save the matrix. According to the manual, we shall avoid saving blocks over the row limits. Therefore, we look into each row separately.

```
  else {
    for (int row = 0; row < m_height; ++row) {
      vector<Block<Type>> blockList;
      Type currValue = get(row, 0);
      int currColumn = 0;
```

First, we gather blocks of similar pixels, they shall be the run-length blocks.

```
      for (int column = 1; column < m_width; ++column) {
        Type value = get(row, column);

        if (!(value == currValue)) {
          blockList.push_back(Block<Type>(column - currColumn, currValue));
          currValue = value;
          currColumn = column;
        }
      }

      if (currColumn < m_width) {
        blockList.push_back(Block<Type>(m_width - currColumn, currValue));
      }
```

Then we store the pixels of blocks holding only one pixels in raw blocks. When we encounter a block holding more than one pixel (a run-length block) we insert the current raw block (if it is not empty) to the list of blocks.

```
      vector<pair<int,int>> pairList;
      int currIndex = ((int) blockList.size()) - 1;
      int lastOneIndex = -1;

      vector<Block<Type>> newBlockList;
      Block<Type> rawBlock;

      for (Block<Type>& block : blockList) {
        if (block.getRepetitions() == 1) {
          rawBlock.setRepetitions(rawBlock.getRepetitions() + 1);
          rawBlock.valueList().push_back(block.valueList().front());
        }
        else {
          if (rawBlock.getRepetitions() > 0) {
            newBlockList.push_back(rawBlock);
            rawBlock = Block<Type>();
          }
          newBlockList.push_back(block);
        }
      }

      if (rawBlock.getRepetitions() > 0) {
        newBlockList.push_back(rawBlock);
      }
```

When all the run-length and raw blocks have been gathered, we write them to the file stream. Remember that in the run-length case we just store one copy of the pixels in the block, while in the raw block case we store every pixel of the block.

```
      for (Block<Type>& block : newBlockList) {
        if ((block.getRepetitions() > 1) &&
            (block.valueList().size() == 1)) {
          char repetitionCount =
            0x80 | (0x7F & (block.getRepetitions() - 1));
          outStream.write((char*) &repetitionCount,
                          sizeof repetitionCount);
          Type value = block.valueList().front();
          value.save(outStream);
        }
        else {
          assert(block.getRepetitions() == block.valueList().size());
          char repetitionCount =
            0x7F & (block.getRepetitions() - 1);
          outStream.write(&repetitionCount, sizeof repetitionCount);

          for (const Type& value : block.valueList()) {
            value.save(outStream);
          }
        }
      }
    }
  }
}
```

The *Header* struct holds the information of the image file header, which consists of twelve fields describing the file.

Offset	Size	Name	Description
0	1	idLength	The length of an optional area following the header, range from 0 to 255 bytes.
1	1	colorMapType	1 if the image file holds an image map, 0 otherwise
2	1	dataTypeCode	The date type, see the table below.
3	2	colorMapOrigin	The offset of the first entry in the image map.
5	2	colorMapLength	The size of the color map (number of entries).
7	1	colorMapDepth	The size of each entry, in bits.
8	2	xOrigin	The origin of the image in the x direction.
10	2	yOrigin	The origin of the image in the y direction.
12	2	width	The width of the image, in pixels.
14	2	height	The height of the image, in pixels.
16	1	bitsPerPixelOrIndex	In the unmapped case: the size of each pixel, in bits. In the mapped case, the size of each index, in bits.
17	1	imageDescriptor	An optional description byte. Ignored in this application.

The *dataTypeCode* field can hold the following value:

Value	Description
0	No image data include, the file is empty
1	Uncompressed mapped color image
2	Uncompressed unmapped color image
3	Uncompressed unmapped black-and-white image
9	Compressed mapped color image
10	Compressed unmapped color image
11	Compressed unmapped black-and-white image

Header.h

```
struct Header {
  char idLength;
  char colorMapType;
  char dataTypeCode;
  short int colorMapOrigin;
  short int colorMapLength;
  char colorMapDepth;
  short int xOrigin;
  short int yOrigin;
  short int width;
  short int height;
  char bitsPerPixelOrIndex;
  char imageDescriptor;
```

The *load* and *save* methods in the *Header* struct reads and writes the fields of the header from and to a file stream.

```
  void load(ifstream& inStream);
  void save(ofstream& outStream) const;
};
```

Header.cpp

```
#include <fstream>
using namespace std;

#include "Header.h"

void Header::load(ifstream& inStream) {
  inStream.read((char*) &idLength, sizeof idLength);
  inStream.read((char*) &colorMapType, sizeof colorMapType);
  inStream.read((char*) &dataTypeCode, sizeof dataTypeCode);
  inStream.read((char*) &colorMapOrigin, sizeof colorMapOrigin);
  inStream.read((char*) &colorMapLength, sizeof colorMapLength);
  inStream.read((char*) &colorMapDepth, sizeof colorMapDepth);
  inStream.read((char*) &xOrigin, sizeof xOrigin);
  inStream.read((char*) &yOrigin, sizeof yOrigin);
  inStream.read((char*) &width, sizeof width);
  inStream.read((char*) &height, sizeof height);
  inStream.read((char*) &bitsPerPixelOrIndex, sizeof bitsPerPixelOrIndex);
  inStream.read((char*) &imageDescriptor, sizeof imageDescriptor);
}

void Header::save(ofstream& outStream) const {
  outStream.write((char*) &idLength, sizeof idLength);
  outStream.write((char*) &colorMapType, sizeof colorMapType);
  outStream.write((char*) &dataTypeCode, sizeof dataTypeCode);
  outStream.write((char*) &colorMapOrigin, sizeof colorMapOrigin);
  outStream.write((char*) &colorMapLength, sizeof colorMapLength);
  outStream.write((char*) &colorMapDepth, sizeof colorMapDepth);
  outStream.write((char*) &xOrigin, sizeof xOrigin);
  outStream.write((char*) &yOrigin, sizeof yOrigin);
  outStream.write((char*) &width, sizeof width);
  outStream.write((char*) &height, sizeof height);
  outStream.write((char*) &bitsPerPixelOrIndex,sizeof bitsPerPixelOrIndex);
  outStream.write((char*) &imageDescriptor, sizeof imageDescriptor);
}
```

The file Main.cpp holds the *main* function of the application, together with functions for reading and writing the files, for converting the images to half its size in both the mapped and unmapped cases, and optimizing the image map in the mapped case.

Main.cpp

```
#include <Map>
#include <Vector>
#include <String>
#include <FStream>
#include <IOStream>
#include <CAssert>
using namespace std;

#include "Header.h"
#include "Pixel.h"
#include "Index.h"
#include "Block.h"
#include "Matrix.h"
```

There are seven different formats of the image file, given in the file header. *NoImageDataIncluded* indicates that the file is empty. *UncompressedColorMappedImage* indicates that the image is mapped and uncompressed, *RunLengthEncodedColorMappedImage* that the image is mapped and compressed. *UncompressedColorImage* and *UncompressedGrayScaleImage* are actually treated as the same format, they are unmapped and uncompressed. The only difference is the size of its pixels, given in the header. In the same way, *RunLengthEncodedColorImage* and *RunLengthEncodedGrayScaleImage* are treated as the same format. They are unmapped and compressed.

```
enum {NoImageDataIncluded = 0,
      UncompressedColorMappedImage = 1,
      UncompressedColorImage = 2,
      UncompressedGrayScaleImage = 3,
      RunLengthEncodedColorMappedImage = 9,
      RunLengthEncodedColorImage = 10,
      RunLengthEncodedGrayScaleImage = 11};

void loadEmptyImage(Header& header, ifstream& inStream,
                    const string& outName);
void loadUnmappedImage(Header& header, ifstream& inStream,
                       const string& outName, bool compressed);
void loadMappedImage(Header& header, ifstream& inStream,
                     const string& outName, bool compressed);
void generateHalfPixelMatrix(const Matrix<Pixel> &fullPixelMatrix,
                             Matrix<Pixel>& halfPixelMatrix);
void generateHalfIndexMatrix(Header& header,
                             const Matrix<Index> &fullIndexMatrix,
                             Matrix<Index>& halfIndexMatrix,
                             vector<Pixel>& pixelVector,
                             map<Pixel,Index>& pixelToIndexMap);
void optimizeHalfIndexMatrix(Header& header, Matrix<Index>& halfIndexMatrix,
                             vector<Pixel>& pixelVector);
void check(bool test, const string& message);
void main(int argc, char* argv[]) {
  check(argc == 3, "usage: halfsize <inputfile> <outputfile>");
  string inName = argv[1], outName = argv[2];

  ifstream inStream(inName, ios::in | ios::binary);
  check((bool) inStream,
        "Error when opening \"" + inName + "\" for reading.");

  cout << "Reading \"" << inName << "\" ..." << endl;
  Header header;
  header.load(inStream);
```

In this application, we only accept bit sizes of 8, 16, 24, or 32.

```
  check((header.bitsPerPixelOrIndex == 8) ||
        (header.bitsPerPixelOrIndex == 16) ||
        (header.bitsPerPixelOrIndex == 24) ||
        (header.bitsPerPixelOrIndex == 32),
        "Pixels or indices must be 8, 16, 24, or 32 bits.");

  switch (header.dataTypeCode) {
    case NoImageDataIncluded:
      loadEmptyImage(header, inStream, outName);
      break;
```

We do not really distinguish between the color and gray-scales cases. The only difference is that the gray-scale case usually requires a pixel size of 8 bits, while the color case usually requires a pixel size of 16, 24, or 32 bits.

```
    case UncompressedColorImage:
    case UncompressedGrayScaleImage:
      loadUnmappedImage(header, inStream, outName, false);
      break;
```

However, the mapped case is difference, why we call a different function.

```
    case UncompressedColorMappedImage:
      loadMappedImage(header, inStream, outName, false);
      break;
```

In the case of run-length compression, we call the same functions as in the uncompressed cases. However, we send true instead of *false* as the last *compression* parameter.

```
    case RunLengthEncodedColorImage:
    case RunLengthEncodedGrayScaleImage:
      loadUnmappedImage(header, inStream, outName, true);
      break;

    case RunLengthEncodedColorMappedImage:
      loadMappedImage(header, inStream, outName, true);
      break;

    default:
      check(false, string("Invalid Data Type Code: ") +
                   header.dataTypeCode);
      break;
  }
}
```

Since there actually is a code for an empty image, we need to deal with it. In that case, we just save the header and a potential id area.

```
void loadEmptyImage(Header& header, ifstream& inStream,
                    const string& outName) {
  char* idBuffer = new char[header.idLength];
  assert((header.idLength == 0) || (idBuffer != nullptr));
  inStream.read(idBuffer, header.idLength);
  cout << "Done: the image contains no data." << endl;

  ofstream outStream(outName, ios::out | ios::binary);
  check((bool) outStream,
        "Error when opening \"" + outName + "\" for writing.");

  cout << "Saving \"" << outName << "\" ..." << endl;
  header.save(outStream);
  outStream.write(idBuffer, header.idLength);
  cout << "Done.";
}
```

In the unmapped case, we read the optional id area and the pixels of the image, which we convert to half its size and save. Note that we do not really care whether the image is compressed, the *Matrix* class takes care of that.

```
void loadUnmappedImage(Header& header, ifstream& inStream,
                       const string& outName, bool compressed) {
  check(header.colorMapType == 0,
        "Invalid Color Map Type: " + header.colorMapType);
```

The id area may be empty (isLength == 0). However, if it is not empty, we read it and save it in *idBuffer*.

```
  char* idBuffer = new char[header.idLength];
  assert((header.idLength == 0) || (idBuffer != nullptr));
  inStream.read(idBuffer, header.idLength);
```

The original pixel data is loaded from the file with the *Matrix* class.

```
  Matrix<Pixel> originalPixelMatrix(header.width, header.height,
                                    compressed);
  originalPixelMatrix.load(header.bitsPerPixelOrIndex, inStream);
  cout << "Done: the image is unmapped and "
       << (compressed ? "compressed" : "uncompressed") << "." << endl;

  cout << "Original size: " << header.width << "x" << header.height << endl;
  header.width /= 2;
  header.height /= 2;
```

In order to create a half-size image, we generate a new matrix.

```
  Matrix<Pixel> halfPixelMatrix(header.width, header.height, compressed);
```

We call the *generateHalfPixelMatrix* function to fill the half-size image matrix with appropriate pixels.

```
  generateHalfPixelMatrix(originalPixelMatrix, halfPixelMatrix);
```

When saving the half-size image file, we open an output stream.

```
  ofstream outStream(outName, ios::out | ios::binary);
  check((bool) outStream,
        "Error when opening \"" + outName + "\" for writing.");
```

We save the header, id area, and pixel matrix to the output stream.

```
  cout << "Saving \"" << outName << "\" ..." << endl;
  header.save(outStream);
  outStream.write(idBuffer, header.idLength);
  halfPixelMatrix.save(outStream);
  cout << "Done.";
}
```

When generating the half-size image, we traverse every second pixel in the x and y direction (rows and columns). Each pixel in the half-size image is the average of four pixels in the original image.

```
void generateHalfPixelMatrix(const Matrix<Pixel> &originalPixelMatrix,
                             Matrix<Pixel>& halfPixelMatrix) {
  cout << "Resizing to: " << halfPixelMatrix.width() << "x"
       << halfPixelMatrix.height() << " ..." << endl;
```

For each pixel in the half-size image matrix, we select a group of four pixels in the original image matrix.

```
  for (int row = 0; row < halfPixelMatrix.height(); ++row) {
    for (int column = 0; column < halfPixelMatrix.width(); ++column) {
      Pixel pixel1 = originalPixelMatrix.get(2 * row, 2 * column),
            pixel2 = originalPixelMatrix.get(2 * row, (2 * column) + 1),
            pixel3 = originalPixelMatrix.get((2 * row) + 1, 2 * column),
            pixel4 = originalPixelMatrix.get((2 * row) + 1, (2 * column)+1);
```

We add the average pixel to the half-size image matrix.

```
      Pixel averagePixel =
        average(pixel1, average(pixel2, average(pixel3, pixel4)));
      halfPixelMatrix.set(row, column, averagePixel);
    }
  }

  cout << "Done." << endl;
}
```

It is a little bit more complicated to load a mapped file. Before we load the image data, we need to load the image map.

```
void loadMappedImage(Header& header, ifstream& inStream,
                     const string& outName, bool compressed) {
  check(header.colorMapType == 1,
        "Invalid Color Map Type: " + header.colorMapType);

  char* idBuffer = new char[header.idLength];
  assert((header.idLength == 0) || (idBuffer != nullptr));
  inStream.read(idBuffer, header.idLength);
```

Each pixel in the image map must hold a size of 8, 16, 24, or 32 bits.

```
  check((header.colorMapDepth == 8) ||
        (header.colorMapDepth == 16) ||
        (header.colorMapDepth == 24) ||
        (header.colorMapDepth == 32),
        "Pixels must be 8, 16, 24, or 32 bits");
```

We load the image map into *pixelVector*. We also load the vector index of each pixel into *pixelToIndex-Map*.

```
  vector<Pixel> pixelVector;
  map<Pixel,Index> pixelToIndexMap;
  for (int count = 0; count < header.colorMapLength; ++count) {
    Pixel pixel;
    pixel.load(header.colorMapDepth, inStream);
    pixelToIndexMap[pixel] =
      Index(header.bitsPerPixelOrIndex, (int) pixelVector.size());
    pixelVector.push_back(pixel);
  }
```

We read the original image matrix in the same way as the unmapped case above.

```
  Matrix<Index> originalIndexMatrix(header.width, header.height,
                                    compressed);
  originalIndexMatrix.load(header.bitsPerPixelOrIndex, inStream);
  cout << "Done: the image is mapped and "
       << (compressed ? "compressed" : "uncompressed") << "." << endl;

  cout << "Original size: " << header.width << "x" << header.height << endl;
  header.width /= 2;
  header.height /= 2;
```

Unlike the unmapped case above, where we defined a matrix of pixels, we now define a matrix of indices.

```
  Matrix<Index> halfIndexMatrix(header.width, header.height, compressed);
```

We call *generateHalfIndexMatrix* to add appropriate values to the half-size image matrix, and *optimizeHalfIndexMatrix* to minimize the size of the image map.

```
  generateHalfIndexMatrix(header, originalIndexMatrix, halfIndexMatrix,
                          pixelVector, pixelToIndexMap);
  optimizeHalfIndexMatrix(header, halfIndexMatrix, pixelVector);

  ofstream outStream(outName, ios::out | ios::binary);
  check((bool) outStream,
        "Error when opening \"" + outName + "\" for writing.");
  cout << "Saving \"" << outName << "\" ..." << endl;
  header.save(outStream);
  outStream.write(idBuffer, header.idLength);
```

Unlike to unmapped case above, we need to save the vector holding the image map.

```
  for (const Pixel& pixel : pixelVector) {
    pixel.save(outStream);
  }

  halfIndexMatrix.save(outStream);
  cout << "Done.";
}
```

When calculating the average pixel in the mapped case, we first need to use the indices in the matrix to look up the actual pixels in the pixel vector.

```
void generateHalfIndexMatrix(Header& header,
                             const Matrix<Index> &originalIndexMatrix,
                             Matrix<Index>& halfIndexMatrix,
                             vector<Pixel>& pixelVector,
                             map<Pixel, Index>& pixelToIndexMap) {
  cout << "Resizing to: " <<header.width << "x"
       << header.height << " ..." << endl;

  for (int row = 0; row < halfIndexMatrix.height(); ++row) {
    for (int column = 0; column < halfIndexMatrix.width(); ++column) {
      Index index1 = originalIndexMatrix.get(2 * row, 2 * column),
            index2 = originalIndexMatrix.get(2 * row, (2 * column) + 1),
            index3 = originalIndexMatrix.get((2 * row) + 1, 2 * column),
            index4 = originalIndexMatrix.get((2 * row) + 1, (2 * column)+1);
      Pixel pixel1 = pixelVector[index1.index()],
            pixel2 = pixelVector[index2.index()],
            pixel3 = pixelVector[index3.index()],
            pixel4 = pixelVector[index4.index()];
      Pixel averagePixel =
        average(pixel1, average(pixel2, average(pixel3, pixel4)));
```

If the average pixel is already stored in the image map, we just add its index to the matrix.

```
      if (pixelToIndexMap.count(averagePixel) > 0) {
        halfIndexMatrix.set(row, column, pixelToIndexMap[averagePixel]);
      }
```

However, there is a possibility that the average pixel is not stored in the image map. In that case we need to add it and increase the value of the *colorMapLength* field in the header.

```
      else {
        pixelToIndexMap[averagePixel] =
          Index(header.bitsPerPixelOrIndex, (int) pixelVector.size());
        pixelVector.push_back(averagePixel);
        ++header.colorMapLength;
        halfIndexMatrix.set(row, column, pixelToIndexMap[averagePixel]);
      }
    }
  }

  cout << "Done." << endl;
}
```

Finally, we need to look into the image map. There is a possibility that there are pixels in the image map that are not referred to in the image data. In that case, we should remove them. When doing so, we must also change the indices in the image data matrix.

```
void optimizeHalfIndexMatrix(Header& header, Matrix<Index>& halfIndexMatrix,
                             vector<Pixel>& pixelVector) {
  cout << "Optimizing image map ..." << endl;
```

We start by defining the array *markArray* of the same size as the image map, holding of Boolean values. We initialize each value to *false*.

```
  bool* markArray = new bool[pixelVector.size()];
  assert(markArray != nullptr);
  for (int index = 0; index < ((int) pixelVector.size()); ++index) {
    markArray[index] = false;
  }
```

We traverse the image data matrix, and mark each referred index in the image map.

```
  for (int row = 0; row < header.height; ++row) {
    for (int column = 0; column < header.width; ++column) {
      markArray[halfIndexMatrix.get(row, column).index()] = true;
    }
  }
```

Then we define the new image map *newPixelVector*, we store only the pixels actually referred. We also store the index of each pixel in *newPixelToIndexMap*.

```
  vector<Pixel> newPixelVector;
  map<Pixel, Index> newPixelToIndexMap;

  for (int index = 0; index < ((int) pixelVector.size()); ++index) {
    if (markArray[index]) {
      Pixel pixel = pixelVector[index];
```

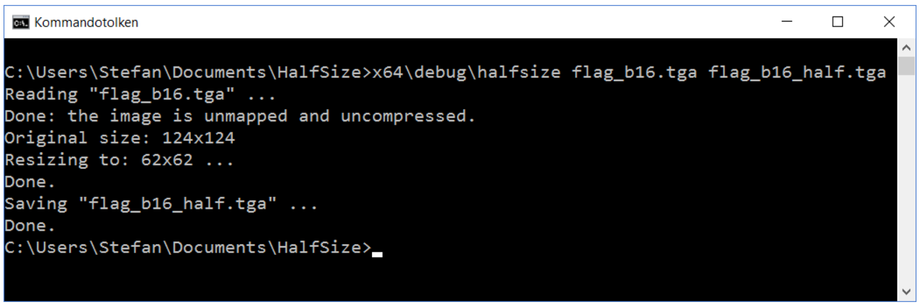
```
        newPixelToIndexMap[pixel] =
          Index(header.bitsPerPixelOrIndex, (int) newPixelVector.size());
        newPixelVector.push_back(pixel);
      }
    }
```

Finally, we traverse the image data matrix and change each index. We look up the pixel of each index and use *newPixelToIndexMap* to find the new index.

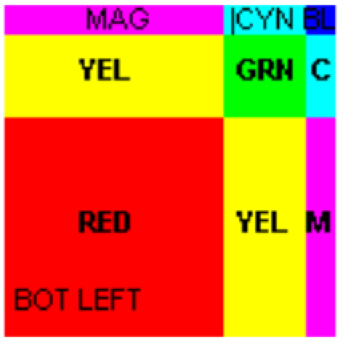
```
    for (int row = 0; row < header.height; ++row) {
      for (int column = 0; column < header.width; ++column) {
        Index oldIndex = halfIndexMatrix.get(row, column);
        Pixel pixel = pixelVector[oldIndex.index()];
        Index newIndex = newPixelToIndexMap[pixel];
        halfIndexMatrix.set(row, column, newIndex);
      }
    }

    header.colorMapLength = (short int) newPixelVector.size();
    cout << "Done: reduced " << pixelVector.size() << " entries to "
         << newPixelVector.size() << " entries." << endl;
    pixelVector = newPixelVector;
  }

  void check(bool test, const string& message) {
    if (!test) {
      cout << message << endl;
      exit(-1);
    }
  }
```

In order to test the application, I have downloaded the following TGA image files from *http://www.fileformat.info/format/tga/sample/*. I have executed the application on the image files with successful result.

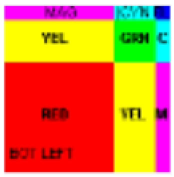
Name	**Mapped**	**Compressed**
cbw8.tga	No	Yes
ccm8.tga	Yes	Yes
ctc16.tga	No	Yes
ctc24.tga	No	Yes
ctc32.tga	No	Yes
flag_b16.tga	No	No
flag_b24.tga	No	No
flag_b32.tga	No	No
flag_t16.tga	No	No
flag_t32.tga	No	No
marbles.tga	No	No
ubw8.tga	No	No
ucm8.tga	Yes	No
utc16.tga	No	No
utc24.tga	No	No
utc32.tga	No	No
xing_b16.tga	No	No
xing_b24.tga	No	No
xing_b32.tga	No	No
xing_t16.tga	No	No
xing_t24.tga	No	No
xing_t32.tga	No	No



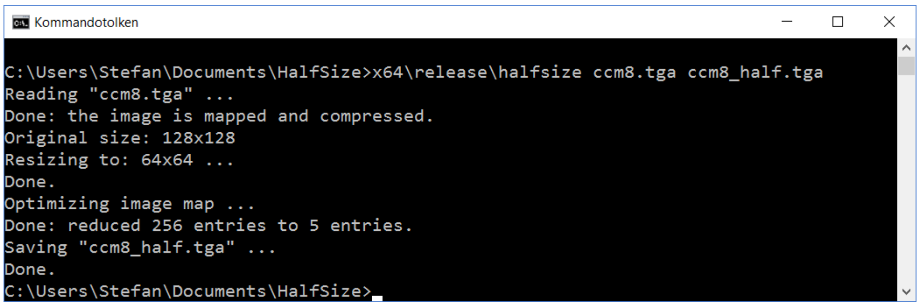
```
C:\Users\Stefan\Documents\HalfSize>x64\debug\halfsize flag_b16.tga flag_b16_half.tga
Reading "flag_b16.tga" ...
Done: the image is unmapped and uncompressed.
Original size: 124x124
Resizing to: 62x62 ...
Done.
Saving "flag_b16_half.tga" ...
Done.
C:\Users\Stefan\Documents\HalfSize>_
```



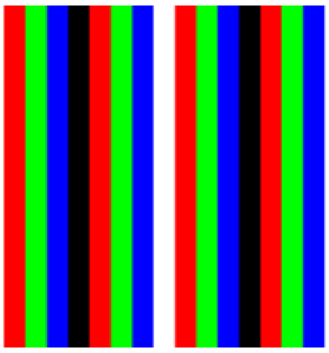
flag_b16.tga



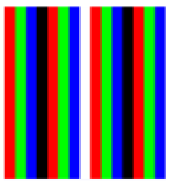
flag_b16_half.tga



```
C:\Users\Stefan\Documents\HalfSize>x64\release\halfsize ccm8.tga ccm8_half.tga
Reading "ccm8.tga" ...
Done: the image is mapped and compressed.
Original size: 128x128
Resizing to: 64x64 ...
Done.
Optimizing image map ...
Done: reduced 256 entries to 5 entries.
Saving "ccm8_half.tga" ...
Done.
C:\Users\Stefan\Documents\HalfSize>_
```



ccm8.tga



ccm8_half.tga