Hierarchical Matching Techniques for Automatic Image Mosaicing

Honours Project

Computer Science and Software Engineering
University of Canterbury

Charles Lee Begg[1]

[1]Supervisor: R Mukundan

Abstract

This paper looks at image mosaicing using hierarchical matching techniques as a way to create a novel view of a scene quickly. Several parameters including error function, region match and constraints are evaluated to find the fastest and most accurate mosaics. Constraints are shown to be very important to intensity matching. Hue matching is more accurate than intensity matching, and hue matching with the intensity correction produces overall better results.

Contents

1 Introduction **9**
- 1.1 Image Mosaicing . . . 9
- 1.2 Applications . . . 10
- 1.3 HSV Colour Space . . . 10

2 Hierarchical Image Matching **11**
- 2.1 Hierarchical Intensity Matching . . . 13
  - 2.1.1 Error Function . . . 13
  - 2.1.2 Region Matching . . . 13
  - 2.1.3 Gaussian Pyramid Depth . . . 13
  - 2.1.4 Matching Constraints . . . 14
- 2.2 Hierarchical Hue Matching . . . 14
  - 2.2.1 Error Function . . . 14
  - 2.2.2 Intensity Correction . . . 14
- 2.3 Comparison of Intensity and Hue Matching . . . 14

3 Experimental Results **15**
- 3.1 Hierarchical Intensity Matching . . . 15
  - 3.1.1 Error Function . . . 15
  - 3.1.2 Region Matching . . . 15
  - 3.1.3 Gaussian Pyramid Depth . . . 16
  - 3.1.4 Matching Constraints . . . 16
- 3.2 Hierarchical Hue Matching . . . 18
  - 3.2.1 Error Function . . . 18
  - 3.2.2 Intensity Correction . . . 18
- 3.3 Comparison of Intensity and Hue Matching . . . 18

4 Discussion and Further Work **23**
- 4.1 Discussion . . . 23
- 4.2 Future Work . . . 23

5 Conclusions **25**

6 Publication of this work **27**

Bibliography **29**

A Source Code **31**

List of Figures

2.1 Pseudo-code for hierarchical image matching 11
2.2 Typical intensity Gaussian pyramid 12

3.1 Region of 9 mismatch, typical 16
3.2 Image of error function in pyramid for the same input images as the previous image 17
3.3 Failure of hue matching error functions 19
3.4 Intensity corrected mosaic using hue matching 20
3.5 Comparison of Intensity and Hue Matching 21

Chapter 1

Introduction

Computer graphics and computer vision are two important areas in computer science. Movies are requiring higher quality graphics, as are computer games. Tele-reality [4]—remote (time or space) visualisation of a scene from reality—is becoming increasingly used in engineering, science and medicine. Image-Based Rendering [3] is a set of methods that take images from the real world and produce novel (new, but not trivially derived) views of the scene. Image-Based Rendering is a cross-over link between computer vision and computer graphics. There are a number of View Synthesis Techniques, including:

- Image Warping or Morphing [5]
- Image Mosaicing
- Stereopsis
- Three and Multiple Camera Geometry [1]

Some of these methods are used to extract depth values and structural information from an image, and can therefore recover a partial 3D model of the scene.

Of these techniques, this project focused on Image Mosaicing and areas associated with it.

1.1 Image Mosaicing

Image mosaicing creates larger images by "stitching" together many smaller images. This is often done by hand to create long panoramas using photographs. This unfortunately leaves visible joins between photos (called seams) and often the images do not line up correctly.

Computer can manipulate images quickly and easily re-project the images so they line up. Previously, a user was required to point out common points to the computer so it could do the re-projection. In the last 11 years, much research has been undertaken in how the computer can find corresponding points. Some of the methods created depend on exact measurements of the camera(s) and their positions and orientations. Other methods look for features in the images[2], and some only look at the colour values of the pixels in the images. Almost all of the algorithms assume uniform illumination and/or some epipolar geometry[6].

When the two assumptions about uniform illumination and epipolar geometry are not made, more source images are available for use in mosaicing and this decreases the need for preprocessing. It also allows images to be taken on one day and extra images taken on another day to fill in gaps in the mosaic.

The significance of the lack of epipolar geometry is that any two images that have—or appear to have—an overlap, can be stitched together. No prior knowledge of the (approximate) location or size of the overlap is known. No assumption can made as to which image is the left and right, or top and bottom, or even which of those two cases the input will be.

1.2 Applications

There are many applications for the techniques presented in this paper.

Telereality—virtual presence, increasingly used in medince and engineering—is a possible application, allowing fast setup of immersive environments without expenise equipment.

These techniques could be used by game content developers to create large textures, skyboxes or skyspheres.

The obvious use is creating cylindrical and spherical panoramas, without any assistance from the user or camera. The input images don't need to have been taken for the perpose of creating a panorama.

Another application is scene navigation through a textured background, for example a street map that shows what the sides of the road look like. A van with a camera pointing out each side is driven down the road, taking photos so that each overlaps with the previous photo taken. The techniques in this paper can take the large number of images and can create the mosaics needed offline without user intervention.

1.3 HSV Colour Space

Most of this project was performed in the HSV colour space. HSV stands for Hue, Saturation, Value. The Value component is also know as Intensity. The components are close to what the human eye perceives and gives easy access to independent components of the pixel composition.

Chapter 2

Hierarchical Image Matching

Without knowing the approximate overlap between two images, all possible overlaps need to be checked to find the best solution. To do this can take hours—up to 11 hours for two 2 megapixel images. A faster method is needed for automatic image matching and tiling.

```
create image pyramid

set min and max x and y

for each level starting at the lowest resolution
    calcuate error for each possible overlap between min and max
    find region of best match
    set new min and max x and y based to region

create new image with overlap using x and y of the middle of the
                                                              region
```

Figure 2.1: Pseudo-code for hierarchical image matching

Hierarchical Image matching is a directed search technique using Gaussian pyramids—an example of a Gaussian pyramid is shown in figure 2.2[1]. The lowest resolution is searched to find the best solution by some metric, and the position of that solution defines the area to be searched in at the next resolution higher. Pseudo-code for this algorithm is given in figure 2.1.

In this project, two image types were investigated for matching: Intensity and Hue.

[1] All images are available in colour at `http://llnz.dyndns.org/gallery/lmtt/`

Figure 2.2: Typical intensity Gaussian pyramid

2.1 Hierarchical Intensity Matching

Hierarchical intensity matching was the first automatic mosaicing methods attempted using the algorithm shown in figure 2.1. Only the intensity was used to create the image pyramid and used in matching.

The final image is created using the original RGB images, using colour channel-wise averages in the overlap. Using averages for the overlap does not produce a smooth transition from one image to the next.

2.1.1 Error Function

Three error functions were evaluated, average absolute difference, root-mean-square of difference, and root of the average absolute difference of squares.

For this section, v_1 is the intensity value of a pixel in the first image and v_2 is the intensity value of the corresponding pixel in the second image.

The first function is the simplest

$$error = \sum \frac{\mid v_1 - v_2 \mid}{n} \tag{2.1}$$

Root-mean-square is

$$error = \sqrt{\sum \frac{(v_1 - v_2)^2}{n}} \tag{2.2}$$

This measure is very common in computer processing and analysis.

Root of average absolute difference of squares

$$error = \sqrt{\sum \frac{\mid v_1^2 - v_2^2 \mid}{n}} \tag{2.3}$$

2.1.2 Region Matching

Three region matching algorithms where tried.

The first looks for the single lowest error.

The second looked for the region of 9 neighbouring overlaps that give the lowest average error.

The last is the same as the second, but attempts to penalise regions that have very small overlaps (typically less than 20 pixels in overlapping region). This last region match as created to try and counter act the behaviour of the previous algorithm shown in the results.

2.1.3 Gaussian Pyramid Depth

The resolution of the smallest image that matching was performed on was controlled and the speed and accuracy of the match was looked at. The size of the smallest image has a large impact on the speed, as every possible overlap is checked (within the constraints when used), and only 121 overlaps per level.

Obviously the greater the depth, the smaller the initial image and the faster the matching occurs, but it is possible for a mismatch to occur, precluding a good match.

2.1.4 Matching Constraints

By constraining the initial possible match, the matching process is sped up and the accuracy is improved by not allowing areas that shouldn't match but has low error value.

This is a more strict method of preventing very small overlaps than the third region matching function in section 2.1.2.

2.2 Hierarchical Hue Matching

Hierarchical hue matching follows the same algorithm as hierarchical intensity matching, but using the hue to match on, instead of intensity. It has been suggested that this method might be more robust than intensity matching. Because the intensity is not used in the matching process, it can be used to calculate an intensity correction that should make seams less visible.

The region matching and constraints were also added to hue matching.

2.2.1 Error Function

Hue is not a linear number, it is a cycle that has values between 0 and 360 degrees, and 0 and 360 are the same value. This complicates the error function, because it can not be a direct number comparison.

The two error functions tried were average absolute shortest difference and average distance between the hue-saturation points.

For this section, h_1 is the hue value of a pixel in the first image and h_2 is the hue value of the corresponding pixel in the second image. The values s_1 and s_2 are the corresponding saturation values in the first and second images respectfully.

The first function can be defined for each pixel as

$$error = MIN(\mid h_1 - h_2 \mid, \mid 360 - h_1 + h_2 \mid) \tag{2.4}$$

The second function for each pixel is

$$error = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2.5}$$

where

$$\begin{aligned} x_n &= s_n \times \cos h_n \\ y_n &= s_n \times \sin h_n \end{aligned} \tag{2.6}$$

2.2.2 Intensity Correction

Because the intensity is independent of the hue being matched on, it is possible to use the final overlapping region to calculate an intensity correction and apply it to the images. The correction is calculated by finding the average difference in intensity between the two images inside the overlap, and adding that value to every pixel in the second image.

2.3 Comparison of Intensity and Hue Matching

A comparison between intensity and hue matching was made using the results from the two matching methods.

Chapter 3

Experimental Results

Evaluating the performance of some of the experiments is subjective when it comes to the visibility of seams and accuracy of matching.

3.1 Hierarchical Intensity Matching

3.1.1 Error Function

There was no difference in the position matched when using any of the error functions, even in combination with some of the variation below. The first functions is the simplest, and fastest to compute, and was used for the other results.

The error values for all the overlaps was saved as an image. The difference between the functions as shown in the images was minimal. The second function has larger error values in general them the first, and all three produced the same shapes in the error images.

3.1.2 Region Matching

The region matching algorithm made a very large difference when it was changed.

The one lowest error position worked well in about 50% of cases.

The second method looking for a region of 9 neighbouring positions with the lowest average error did work well for some input image pairs, but often matched on a single row or column of pixels, as shown in figure 3.1.

The third method trying to penalise regions that have very small overlaps did improve the results from the second method, but did not completely solve the problem. This method was also much slower than the first method.

The error values for all the overlaps was saved as an image. As can be seen in figure 3.2, the region of the best match is not sharply defined—neither a group of very dark pixels, or a dark pixel surrounded by light pixels (indicating high error) define a small area of possibly accurate matches. Strong repeating vertical or horizontal lines do show clearly in the error values, and a large cross shape is often found centred at good matching positions but are too large to use beyond the first level—there is very little variance in most regions and therefore after the initial matching level will not have much impact at all due to the small size of the region searched.

Figure 3.1: Region of 9 mismatch, typical

3.1.3 Gaussian Pyramid Depth

The depth of the pyramid does make a large difference in both time and accuracy as seen in table 3.1. Each level further makes the process around much faster down to the constant overhead of 0.14s.

Depth	Time elapsed
7	0.140s
6	0.150s
5	0.200s
4	0.350s
3	2.160s
2	17.590s
1	4 min, 11.32s
0 (original size)	Hours

Table 3.1: Comparison of time taken and pyramid depth

Accuracy is not greatly affected by the depth. For 2 mega-pixel images, no change in final matching position is found until around the 7th level, where the images are 5x7 pixels.

Most pairs of related images are accurate above level 3 or 4—where images are around 100 to 200 pixels wide—and a few from as low as level 7.

Four levels seems to be the best compromise between speed and accuracy.

3.1.4 Matching Constraints

Adding constraints to the matching position made a very large difference to the mosaics.

A set of typical constraints were created that work for most image sets. The left-most point of the second image must occur between 50% and 95% of the width of the first image, and the top-most point within 20% of the height either

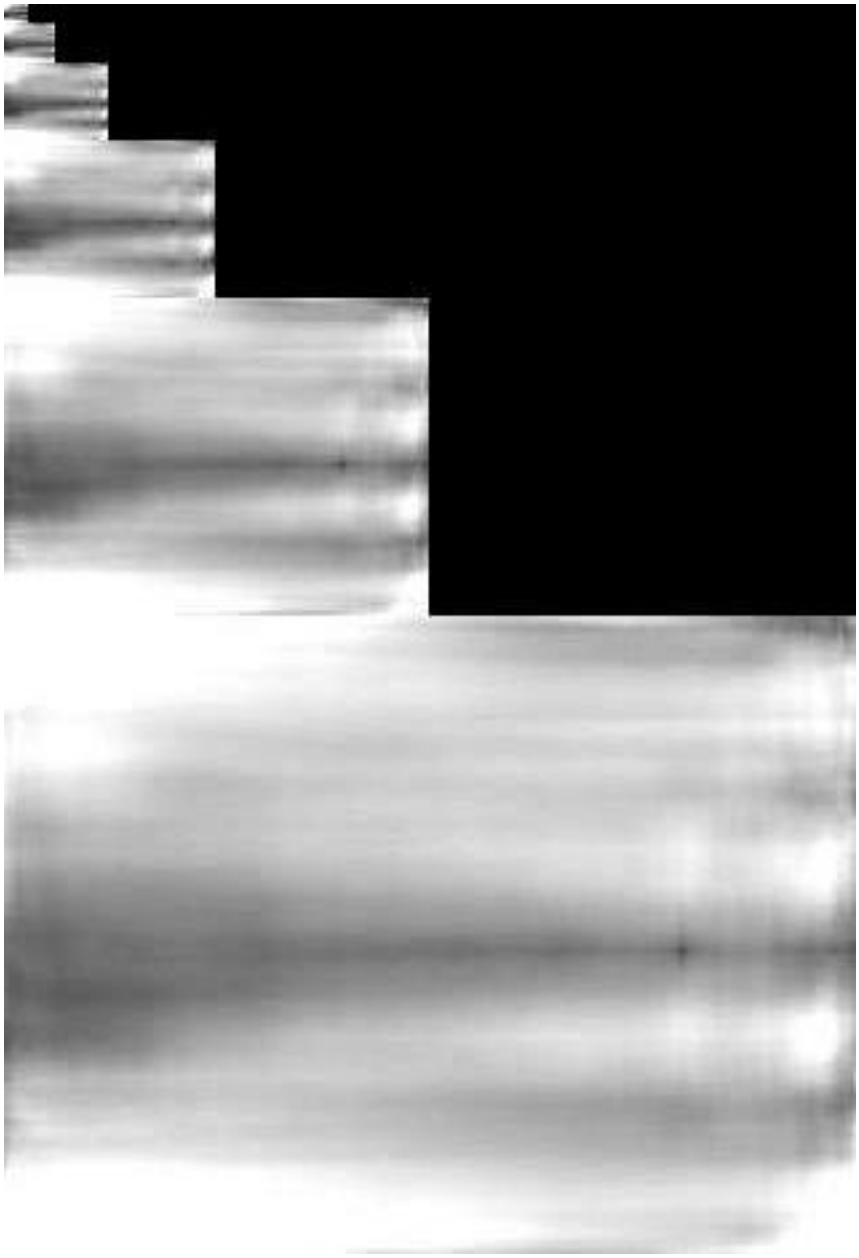

Figure 3.2: Image of error function in pyramid for the same input images as the previous image

up or down of the top of first image. The depth of the Gaussian pyramid was also limited to 4 to allow a better first match.

There is also a small reduction in time taken for matching, due to there being a smaller number of overlaps to check at the first level.

3.2 Hierarchical Hue Matching

3.2.1 Error Function

The error function for hue matching made considerable difference to the final mosaic created.

The first function is much faster to compute, but does not match well as often as the second function. Neither function matched correctly all the sample input images that were tried—for example ins figure 3.3.

3.2.2 Intensity Correction

The intensity correction does make a very large difference to the quality of the final mosaic. Most seams are completely invisible, and the rest have obvious perspective distortion as can be seen in figure 3.4.

3.3 Comparison of Intensity and Hue Matching

As shown in figure 3.5 is an example of intensity and hue matched seams. This result is typical of the comparisons done between the two methods. The person in the foreground moved towards the right of the image between the taking of the photos. The intensity matched image has matched on the person, ignoring the rest of the image—the strong contrast between the person and the scene is likely to be the main cause. Hue matching has closely matched the colours and in is much closer to seamless, only a small mount of perspective distortion is apparent.

Hue matching in most cases does provide a better match, and due to the intensity correction, a more seamless mosaic. Some exceptions were found, such as when the hue does have a large contrast of values.

(a) First error function mismatch on first seam

(b) Second error function poorly matches the central seam

(c) Second error function for end seams and first for central seam, still visibly incorrect

Figure 3.3: Failure of hue matching error functions

(a) Original hue matched image

(b) Intensity corrected hue matched image

Figure 3.4: Intensity corrected mosaic using hue matching

(a) Intensity image

(b) Hue image

Figure 3.5: Comparison of Intensity and Hue Matching

Chapter 4

Discussion and Further Work

4.1 Discussion

Constraints do add an assumption that had not been presented in previous automatic matching that was done in this project. The assumption is that there is an order to the input images. For example, the first input image is the left-most image, and the second image is the right-most. For most applications this assumption will not cause problems. It could be easily generalised into knowing which of four categories the pair of images falls into: left–right, right–left, top–bottom, bottom–top. Some quick tests revealed that constraints that just exclude the 5% closest to the edges made work well, and does not have the order/position assumption. More specific constraints could be developed for any particular application from any additional information available, speeding up matching and improving accuracy.

"Ghosts"—as in figures blah and 3.4, and more clearly in figure 3.5(b)–and other artifacts are hard to avoid and better stitching functions can not remove them completely.

Locating the overlap, even if not completely accurate could be a useful step before trying to find corresponding points in the image, greatly speeding up operations such as perspective correction.

4.2 Future Work

There are many areas that need to be looked at. Instead of hierarchical/Gaussian pyramid matching, a 2d sliding window model needs evaluation, as do feature extraction methods, which would also allow for automatic perspective corrections with less extra processing. In Hue matching, more research is needed to evaluate error functions to find ones that work more often. Another area for future work is to look at matching on hue and intensity, and matching gradients and edges instead of individual pixels.

Chapter 5

Conclusions

The error metric in intensity matching does not make a significant difference, region matching does make some difference without constraints. Constraints make a very large difference to the speed and accuracy of mosaicing, when correctly set.

Hue matching does work, with some error metrics working better than others. Intensity correction does make the seams much less visible—only perspective distortion is apparent.

Hue matching gives a better final mosaic than hierarchical intensity matching, due to being more robust to noise and a better descriptor for the scene.

Chapter 6

Publication of this work

This project was presented at the Image and Vision Computing Conference 2004. The paper and the presentation slides will be available at `http://llnz.dyndns.org/papers/`, along with source code and this paper.

Bibliography

[1] O Faugeras and Q Luong. *The Geometry of Multiple Images*. MIT Press, 2001.

[2] Y Kanazawa and K Kanatani. Image mosaicing by stratified matching. *Image And Vision Computing*, 22(2):11, Feb 2004.

[3] S B Kany. A survey of image-based rendering techniques. Technical Report CRL 97/4, Cambridge Research Lab, Digital Equipment Corp, August 1997.

[4] R Szeliski. Image mosaicing for tele-reality applications. Technical Report CRL 94/2, Cambridge Research Lab, Digital Equipment Corp, May 1994.

[5] G Wolberg. Recent advances in image morphing. *Proc. Computer Graphics International*, 1996.

[6] A Zisserman. Geometric framework for vision, 1997.

Appendix A

Source Code

The tool I created to preform image mosaicing is called `lmtt`, and it takes two images, and uses a transformation to create a third image. The final version used is 0.1.0 and the source code will be available at `http://llnz.dyndns.org/papers/`.

main.cpp

```
#include <iostream>

#include "image.h"
#include "transformfactory.h"
#include "transformer.h"

void print_usage(int argc, char** argv){
  std::cerr << "Incorrect argument" << std::endl;
  std::cout << "Usage:" << std::endl;
  std::cout << " " << argv[0] <<
    " <transform> <output_image> <input_image1> <input_image2> [options]" <<
    std::endl;
  TransformFactory::getInstance()->printTransformerNames();
}

int main(int argc, char** argv){

  if(argc < 5){
    print_usage(argc, argv);
  }else{

    Image output, input1, input2;

    if(!(input1.load(argv[3]))){
      std::cerr << "Could not load input image 1 \"" << argv[3] << "\""
<< std::endl;
      return 2;
```

```
    }

    if(!(input2.load(argv[4]))){
      std::cerr << "Could not load input image 2 \"" << argv[4] << "\""
<< std::endl;
      return 2;
    }

    // choose algorithm and do it
    Transformer * trans =
      TransformFactory::getInstance()->createTransformer(argv[1]);
    if(trans != NULL){
      // just put the pictures side by side
      if(!(trans->doTransform(output, input1, input2, argc - 5, &(argv[5])))){
std::cerr << "Could not perform transformation" << std::endl;
delete trans;
return 4;
      }

      delete trans;

    }else{
      std::cerr << "No tranformation chosen" << std::endl;
      return 1;
    }

    if(!(output.write(argv[2]))){
      std::cerr << "Unable to write output to \"" << argv[2] << "\""
<< std::endl;
      return 3;
    }

  }

  return 0;
}
```

transformfactory.cpp

```
#include <iostream>

#include "transformer.h"

#include "trivial.h"
#include "absolute.h"
#include "hierarchical.h"
#include "huematchequ.h"
```

```
#include "transformfactory.h"

TransformFactory * TransformFactory::instance = NULL;

TransformFactory::TransformFactory(){

}

TransformFactory * TransformFactory::getInstance(){
  if(instance == NULL){
    instance = new TransformFactory();
  }
  return instance;
}

Transformer * TransformFactory::createTransformer(const std::string &name){
  if(name == "trivial"){
    return new TrivialTransform();
  }else if(name == "absolute"){
    return new AbsoluteTransform();
  }else if(name == "hierarchical"){
    return new HierarchicalTransform();
  }else if(name == "huematchequ"){
    return new HueMatchEquTransform();
  }

  return NULL;
}

void TransformFactory::printTransformerNames(){
  std::cout << "Known transformers:" << std::endl;
  std::cout << " " << "trivial" << std::endl;
  std::cout << " " << "absolute" << std::endl;
  std::cout << " " << "hierarchical" << std::endl;
  std::cout << " " << "huematchequ" << std::endl;

}
```

transformer.h

```
// A transformer
//  turns two images into one


#ifndef TRANSFORMER_H
#define TRANSFORMER_H

#include <string>
```

```
#include "image.h"


class Transformer{

 public:

  virtual ~Transformer();

  virtual bool doTransform(Image &out, Image &in1, Image &in2,
   int argc, char** argv) = 0;
  virtual void printOptions() = 0;

  std::string getName();

 protected:
  std::string name;

};

#endif
```

transformer.cpp

```
#include <iostream>

#include "transformer.h"

Transformer::~Transformer(){
}

std::string Transformer::getName(){
  return name;
}
```

image.cpp

```
#include <iostream>
#include <stdio.h>
#include <jpeglib.h>
#include <setjmp.h>

#include "image.h"

Image::Image(){
  width = height = 0;
```

```
  data = NULL;
}

Image::Image(const Image &rhs){
  width = rhs.width;
  height = rhs.height;
  data = new unsigned char[width * height * 3];
  memcpy(data, rhs.data, width * height * 3);
}

Image::~Image(){
  if(data != NULL)
    delete[] data;
}

Image Image::operator=(const Image &rhs){
  width = rhs.width;
  height = rhs.height;
  if(data != NULL)
    delete[] data;
  data = new unsigned char[width * height * 3];
  memcpy(data, rhs.data, width * height * 3);
  return *this;
}

void Image::setSize(int w, int h){
  width = (w < 0) ? 0 : w;
  height = (h < 0) ? 0 : h;
  if(data != NULL)
    delete[] data;
  data = new unsigned char[w * h * 3];
}

int Image::getWidth() const{
  return width;
}

int Image::getHeight() const{
  return height;
}

unsigned char * Image::getData() const{
  return data;
}

unsigned char * Image::getLine(int line) const{

  if(line < height){
    return data + line * width * 3;
  }else{
```

```
    return NULL;
  }
}

unsigned char * Image::getPixel(int x, int y) const{
  if(x < width && y < height){
    return data + y * width * 3 + x * 3;
  }else{
    return NULL;
  }
}

Image Image::scaleDown() const{
  Image out;

  out.setSize(width / 2 + width % 2, height / 2 + height % 2);

  for(int y = 0; y < out.height - height % 2; y++){

    for(int x = 0; x < out.width - width % 2; x++){
      out.data[(y * out.width + x) * 3] =
(data[(2*y * width + 2 * x) * 3]
 + data[(2*y * width + 2 * x + 1) * 3]
 + data[((2*y + 1) * width + 2 * x) * 3]
 + data[((2*y + 1) * width + 2 * x + 1) * 3]) / 4 ;
      out.data[(y * out.width + x) * 3 + 1] =
(data[(2*y * width + 2 * x) * 3 + 1]
 + data[(2*y * width + 2 * x + 1) * 3 + 1]
 + data[((2*y + 1) * width + 2 * x) * 3 + 1]
 + data[((2*y + 1) * width + 2 * x + 1) * 3] + 1) / 4 ;
      out.data[(y * out.width + x) * 3 + 2] =
(data[(2*y * width + 2 * x) * 3 + 2]
 + data[(2*y * width + 2 * x + 1) * 3 + 2]
 + data[((2*y + 1) * width + 2 * x) * 3 + 2]
 + data[((2*y + 1) * width + 2 * x + 1) * 3] + 2) / 4 ;
    }
    if(width % 2 == 1){
      out.data[(y * out.width + out.width - 1) * 3] =
(data[(2 * y * width + width - 1) * 3]
 + data[((2 * y + 1) * width + width - 1) * 3]) / 2;
      out.data[(y * out.width + out.width - 1) * 3 + 1] =
(data[(2 * y * width + width - 1) * 3 + 1]
 + data[((2 * y + 1) * width + width - 1) * 3 + 1]) / 2;
      out.data[(y * out.width + out.width - 1) * 3 + 2] =
(data[(2 * y * width + width - 1) * 3 + 2]
 + data[((2 * y + 1) * width + width - 1) * 3 + 2]) / 2;
    }
  }
  if(height % 2 == 1){
    for(int x = 0; x < out.width - width % 2; x++){
```

```
      out.data[((out.height - 1) * out.width + x) * 3] =
(data[((height - 1) * width + 2 * x) * 3]
 + data[((height - 1) * width + 2 * x) * 3]) / 2;
      out.data[((out.height - 1) * out.width + x) * 3 + 1] =
(data[((height - 1) * width + 2 * x) * 3 + 1]
 + data[((height - 1) * width + 2 * x) * 3 + 1]) / 2;
      out.data[((out.height - 1) * out.width + x) * 3 + 2] =
(data[((height - 1) * width + 2 * x) * 3 + 2]
 + data[((height - 1) * width + 2 * x) * 3 + 2]) / 2;
    }
    if(width % 2 == 1){
      out.data[(out.height * out.width - 1) * 3] =
data[(height * width - 1) * 3];
      out.data[(out.height * out.width - 1) * 3 + 1] =
data[(height * width - 1) * 3 + 1];
      out.data[(out.height * out.width - 1) * 3 + 2] =
data[(height * width - 1) * 3 + 2];
    }
  }

  return out;
}

Image Image::scaleDownHSV() const{
  Image out;

  out.setSize(width / 2 + width % 2, height / 2 + height % 2);

  for(int y = 0; y < out.height - height % 2; y++){

    for(int x = 0; x < out.width - width % 2; x++){

      float a = ((getPixel(2*x, 2*y)[1])
 * sin(M_PI / 90.0 * (double)getPixel(2*x, 2*y)[0])
 + (getPixel(2*x + 1, 2*y)[1])
 * sin(M_PI / 90.0 * (double)getPixel(2*x + 1, 2*y)[0])
 + (getPixel(2*x + 1, 2*y)[1])
 * sin(M_PI / 90.0 * (double)getPixel(2*x + 1, 2*y)[0])
 + (getPixel(2*x + 1, 2*y + 1)[1])
 * sin(M_PI / 90.0 * (double)getPixel(2*x + 1, 2*y + 1)[0])) / 4.0;
      float b = ((getPixel(2*x, 2*y)[1])
 * cos(M_PI / 90.0 * (double)getPixel(2*x, 2*y)[0])
 + (getPixel(2*x + 1, 2*y)[1])
 * cos(M_PI / 90.0 * (double)getPixel(2*x + 1, 2*y)[0])
 + (getPixel(2*x + 1, 2*y)[1])
 * cos(M_PI / 90.0 * (double)getPixel(2*x + 1, 2*y)[0])
 + (getPixel(2*x + 1, 2*y + 1)[1])
 * cos(M_PI / 90.0 * (double)getPixel(2*x + 1, 2*y + 1)[0])) / 4.0;

      out.getPixel(x, y)[1] = (int)sqrt(a * a + b* b);
```

```
      float ang = 0.0;
      if(b != 0.0)
ang= atan(a / b);
      if(b < 0){
ang += M_PI/ 2.0;
      }
      if(ang < 0.0)
ang += 2 * M_PI;
      out.getPixel(x, y)[0] = (int)(90.0 * ang / M_PI);



      out.data[(y * out.width + x) * 3 + 2] =
(data[(2*y * width + 2 * x) * 3 + 2]
 + data[(2*y * width + 2 * x + 1) * 3 + 2]
 + data[((2*y + 1) * width + 2 * x) * 3 + 2]
 + data[((2*y + 1) * width + 2 * x + 1) * 3] + 2) / 4 ;
    }
    if(width % 2 == 1){

      out.data[(y * out.width + out.width - 1) * 3 + 1] =
(data[(2 * y * width + width - 1) * 3 + 1]
 + data[((2 * y + 1) * width + width - 1) * 3 + 1]) / 2;
      out.data[(y * out.width + out.width - 1) * 3 + 2] =
(data[(2 * y * width + width - 1) * 3 + 2]
 + data[((2 * y + 1) * width + width - 1) * 3 + 2]) / 2;
    }
  }
  if(height % 2 == 1){
    for(int x = 0; x < out.width - width % 2; x++){

      out.data[((out.height - 1) * out.width + x) * 3 + 1] =
(data[((height - 1) * width + 2 * x) * 3 + 1]
 + data[((height - 1) * width + 2 * x) * 3 + 1]) / 2;
      out.data[((out.height - 1) * out.width + x) * 3 + 2] =
(data[((height - 1) * width + 2 * x) * 3 + 2]
 + data[((height - 1) * width + 2 * x) * 3 + 2]) / 2;
    }
    if(width % 2 == 1){

      out.data[(out.height * out.width - 1) * 3 + 1] =
data[(height * width - 1) * 3 + 1];
      out.data[(out.height * out.width - 1) * 3 + 2] =
data[(height * width - 1) * 3 + 2];
    }
  }

  return out;
}
```

```
Image Image::toGray() const{
  Image out;

  out.setSize(width, height);

  for(int i = 0; i < width * height; i++){
    out.data[i * 3] = (data[i * 3] > data[i * 3 + 1])
      ? (data[i * 3])
      : ((data[i * 3 + 1] > data[i * 3 + 2])
 ? (data[i * 3 + 1])
 : (data[i * 3 + 2]));
    out.data[i * 3 + 2] = out.data[i * 3 + 1] = out.data[i * 3];
  }

  return out;
}

Image Image::toHSV() const{
  Image out;
  out.setSize(width, height);

  for(int i = 0; i < width * height; i++){
    float r = ((float)(data[i * 3])) / 255.0;
    float g = ((float)(data[i * 3 + 1])) / 255.0;
    float b = ((float)(data[i * 3 + 2])) / 255.0;
    out.data[i * 3 + 2] = (data[i * 3] > data[i * 3 + 1])
      ? ((data[i * 3] > data[i*3 + 2])
 ? data[i*3] : data[i*3 + 2])
      : ((data[i * 3 + 1] > data[i * 3 + 2])
 ? (data[i * 3 + 1]) : (data[i * 3 + 2]));
    float max = fmax(r, fmax(g, b));
    float min = fmin(r, fmin(g, b));
    float delta = max - min;

    if(max > 0.0){
      out.data[i * 3 + 1] = (int)((delta / max) * 255);
    }else{
      out.data[i * 3 + 1] = 0;
      out.data[i * 3] = 0;
      continue;
    }

    float h;

    if(r == max){
      h = (g - b) / delta;
    }else if(g == max){
      h = 2 + (b - r) / delta;
    }else{
```

```
      h = 4 + (r - g) / delta;
    }
    h *= 60.0;
    if(h < 0.0){
      h += 360.0;
    }
    out.data[i * 3] = (unsigned char)(h / 2.0);
    //std::cout << h << " " << (int)out.data[i * 3] << std::endl;
  }

  return out;
}

Image Image::fromHSV() const{
  Image out;
  out.setSize(width, height);

  for(int i = 0; i < width * height; i++){
    if(data[i * 3 + 1] == 0){
      out.data[i * 3] = out.data[i * 3 + 1]
= out.data[i * 3 + 2] = data[i * 3 + 2];
      continue;
    }
    float h = ((float)(data[i * 3])) * 2.0;
    //std::cout << (int)data[i * 3] << " " << h << std::endl;
    h /= 60.0;
    int ih = (int)floor(h);
    float v = ((float)(data[i * 3 + 2])) / 255.0;
    float s = ((float)(data[i * 3 + 1])) / 255.0;

    float f = h - ih;
    float p = v * (1 - s) * 255;
    float q = v * (1 - s * f) * 255;
    float t = v * (1 - s * (1 - f)) * 255;
    //std::cout << ih;
    switch(ih){
    case 0:
      out.data[i * 3] = data[i * 3 + 2];
      out.data[i * 3 + 1] = (unsigned char)t;
      out.data[i * 3 + 2] = (unsigned char)p;
      break;
    case 1:
      out.data[i * 3] = (unsigned char)q;
      out.data[i * 3 + 1] = data[i * 3 + 2];
      out.data[i * 3 + 2] = (unsigned char)p;
      break;
    case 2:
      out.data[i * 3] = (unsigned char)p;
      out.data[i * 3 + 1] = data[i * 3 + 2];
      out.data[i * 3 + 2] = (unsigned char)t;
```

```
      break;
    case 3:
      out.data[i * 3] = (unsigned char)p;
      out.data[i * 3 + 1] = (unsigned char)q;
      out.data[i * 3 + 2] = data[i * 3 + 2];
      break;
    case 4:
      out.data[i * 3] = (unsigned char)t;
      out.data[i * 3 + 1] = (unsigned char)p;
      out.data[i * 3 + 2] = data[i * 3 + 2];
      break;
    case 5:
    default:
      out.data[i * 3] = data[i * 3 + 2];
      out.data[i * 3 + 1] = (unsigned char)p;
      out.data[i * 3 + 2] = (unsigned char)q;
      break;
    }
  }

  return out;
}


// reading and writing the image below here
//copied from libjpeg sample

struct my_error_mgr {
    struct jpeg_error_mgr pub;    /* "public" fields */

    jmp_buf setjmp_buffer;        /* for return to caller */
};

static void my_error_exit (j_common_ptr cinfo)
{
  /* cinfo->err really points to a my_error_mgr struct, so coerce pointer */
  struct my_error_mgr * myerr = (struct my_error_mgr *) cinfo->err;

  /* Always display the message. */
  /* We could postpone this until after returning, if we chose. */
  (*cinfo->err->output_message) (cinfo);

  /* Return control to the setjmp point */
  longjmp(myerr->setjmp_buffer, 1);
}


bool Image::load(const std::string &file){

  struct jpeg_decompress_struct cinfo;
```

```
/* We use our private extension JPEG error handler.
 * Note that this struct must live as long as the main JPEG parameter
 * struct, to avoid dangling-pointer problems.
 */
struct my_error_mgr jerr;

/* More stuff */
FILE * infile;                /* source file */
JSAMPARRAY buffer;            /* Output row buffer */
int row_stride;               /* physical row width in output buffer */

/* In this example we want to open the input file before doing anything else,
 * so that the setjmp() error recovery below can assume the file is open.
 * VERY IMPORTANT: use "b" option to fopen() if you are on a machine that
 * requires it in order to read binary files.
 */

if ((infile = fopen(file.c_str(), "rb")) == NULL) {
  //std::cerr << "can't open " << file << std::endl;
  return false;
}

/* Step 1: allocate and initialize JPEG decompression object */

/* We set up the normal JPEG error routines, then override error_exit. */
cinfo.err = jpeg_std_error(&jerr.pub);
jerr.pub.error_exit = my_error_exit;
/* Establish the setjmp return context for my_error_exit to use. */
if (setjmp(jerr.setjmp_buffer)) {
  /* If we get here, the JPEG code has signaled an error.
   * We need to clean up the JPEG object, close the input file, and return.
   */
  jpeg_destroy_decompress(&cinfo);
  fclose(infile);
  return false;
}
/* Now we can initialize the JPEG decompression object. */
jpeg_create_decompress(&cinfo);

jpeg_stdio_src(&cinfo, infile);

/* Step 3: read file parameters with jpeg_read_header() */

(void) jpeg_read_header(&cinfo, TRUE);
/* We can ignore the return value from jpeg_read_header since
 *   (a) suspension is not possible with the stdio data source, and
 *   (b) we passed TRUE to reject a tables-only JPEG file as an error.
 * See libjpeg.doc for more info.
 */
```

```
/* Step 4: set parameters for decompression */

/* In this example, we don't need to change any of the defaults set by
 * jpeg_read_header(), so we do nothing here.
 */

/* Step 5: Start decompressor */

(void) jpeg_start_decompress(&cinfo);
/* We can ignore the return value since suspension is not possible
 * with the stdio data source.
 */

/* We may need to do some setup of our own at this point before reading
 * the data.  After jpeg_start_decompress() we have the correct scaled
 * output image dimensions available, as well as the output colormap
 * if we asked for color quantization.
 * In this example, we need to make an output work buffer of the right size.
 */
/* JSAMPLEs per row in output buffer */
row_stride = cinfo.output_width * cinfo.output_components;

data = new unsigned char[row_stride * cinfo.output_height];

width = cinfo.output_width;
height = cinfo.output_height;

if(cinfo.output_components != 3){
  jpeg_destroy_decompress(&cinfo);
  fclose(infile);
  std::cerr << "Umm... I think we need a full colour jpeg" << std::endl;
  return false;
}

/* Make a one-row-high sample array that will go away when done with image */
//buffer = (*cinfo.mem->alloc_sarray)
//              ((j_common_ptr) &cinfo, JPOOL_IMAGE, row_stride, 1);

/* Step 6: while (scan lines remain to be read) */
/*           jpeg_read_scanlines(...); */

/* Here we use the library's state variable cinfo.output_scanline as the
 * loop counter, so that we don't have to keep track ourselves.
 */
while (cinfo.output_scanline < cinfo.output_height) {
  /* jpeg_read_scanlines expects an array of pointers to scanlines.
   * Here the array is only one element long, but you could ask for
   * more than one scanline at a time if that's more convenient.
   */
  unsigned char* foo = & data[row_stride * cinfo.output_scanline];
```

```
    buffer = & foo;
    (void) jpeg_read_scanlines(&cinfo, buffer, 1);
    /* Assume put_scanline_someplace wants a pointer and sample count. */
    //put_scanline_someplace(buffer[0], row_stride);
  }

  /* Step 7: Finish decompression */

  (void) jpeg_finish_decompress(&cinfo);
  /* We can ignore the return value since suspension is not possible
   * with the stdio data source.
   */

  /* Step 8: Release JPEG decompression object */

  /* This is an important step since it will release a good deal of memory. */
  jpeg_destroy_decompress(&cinfo);

  /* After finish_decompress, we can close the input file.
   * Here we postpone it until after no more JPEG errors are possible,
   * so as to simplify the setjmp error logic above.  (Actually, I don't
   * think that jpeg_destroy can do an error exit, but why assume anything...)
   */
  fclose(infile);

  /* At this point you may want to check to see whether any corrupt-data
   * warnings occurred (test whether jerr.pub.num_warnings is nonzero).
   */

  return jerr.pub.num_warnings == 0;
}

bool Image::write(const std::string &file) const{
  FILE * outfile;               /* target file */
  JSAMPROW row_pointer[1];      /* pointer to JSAMPLE row[s] */
  int row_stride;

  struct jpeg_compress_struct cinfo;
  struct jpeg_error_mgr jerr;

  /* We have to set up the error handler first, in case the initialization
   * step fails.  (Unlikely, but it could happen if you are out of memory.)
   * This routine fills in the contents of struct jerr, and returns jerr's
   * address which we place into the link field in cinfo.
   */
  cinfo.err = jpeg_std_error(&jerr);
  /* Now we can initialize the JPEG compression object. */
  jpeg_create_compress(&cinfo);

  if ((outfile = fopen(file.c_str(), "wb")) == NULL) {
```

```
    //std::cerr << "can't open " << file << std::endl;
    jpeg_destroy_compress(&cinfo);
    return false;
  }
  jpeg_stdio_dest(&cinfo, outfile);

  cinfo.image_width = width;      /* image width and height, in pixels */
  cinfo.image_height = height;
  cinfo.input_components = 3;           /* # of color components per pixel */
  cinfo.in_color_space = JCS_RGB;       /* colorspace of input image */

  jpeg_set_defaults(&cinfo);

  jpeg_start_compress(&cinfo, TRUE);

  row_stride = width * 3; /* JSAMPLEs per row in image_buffer */

  while (cinfo.next_scanline < cinfo.image_height) {
    /* jpeg_write_scanlines expects an array of pointers to scanlines.
     * Here the array is only one element long, but you could pass
     * more than one scanline at a time if that's more convenient.
     */
    row_pointer[0] = & data[cinfo.next_scanline * row_stride];
    (void) jpeg_write_scanlines(&cinfo, row_pointer, 1);
  }

  jpeg_finish_compress(&cinfo);
  /* After finish_compress, we can close the output file. */
  fclose(outfile);

  /* Step 7: release JPEG compression object */

  /* This is an important step since it will release a good deal of memory. */
  jpeg_destroy_compress(&cinfo);

  return true;
}
```

trivial.cpp

```
#include <iostream>

#include "trivial.h"

TrivialTransform::TrivialTransform(){
  name = "trivial";
}
```

```
TrivialTransform::~TrivialTransform(){

}

bool TrivialTransform::doTransform(Image &out, Image &in1, Image &in2,
   int argc, char** argv){


  bool vert = false;
  //check params

   for(int i = 0; i < argc; i++){
    std::cout << argv[i] << std::endl;
    if(strncmp(argv[i], "-v", 2) == 0){
      vert = true;
    }
  }

  if(vert){
    out.setSize((in1.getWidth() < in2.getWidth())
? in2.getWidth()
: in1.getWidth(), in1.getHeight() + in2.getHeight());
    for(int l = 0; l < in1.getHeight(); l++){
      memcpy(out.getLine(l), in1.getLine(l), in1.getWidth() * 3);
      if(in1.getWidth() < in2.getWidth()){
memset(out.getPixel(in1.getWidth(), l), '\0',
       (in2.getWidth() - in1.getWidth()) * 3);
      }
    }
    for(int l = 0; l < in2.getHeight(); l++){
      memcpy(out.getLine(l + in1.getHeight()), in2.getLine(l), in2.getWidth() * 3);
      if(in2.getWidth() < in1.getWidth()){
memset(out.getPixel(in2.getWidth(), l + in1.getHeight()), '\0',
       (in1.getWidth() - in2.getWidth()) * 3);
      }
    }
  }else{
    out.setSize(in1.getWidth() + in2.getWidth(),
(in1.getHeight() < in2.getHeight())
? in2.getHeight() : in1.getHeight());

    for(int l = 0; l < out.getHeight(); l++){
      if(l < in1.getHeight()){
memcpy(out.getData() + (l * out.getWidth() * 3),
       in1.getData() + (l * in1.getWidth() * 3), in1.getWidth() * 3);
      }else{
memset(out.getData() + (l * out.getWidth() * 3), '\0', in1.getWidth() * 3);
      }
      if(l < in2.getHeight()){
memcpy(out.getData() + (l * out.getWidth() + in1.getWidth()) * 3,
```

```
       in2.getData() + (l * in2.getWidth() * 3), in2.getWidth() * 3);
      }else{
memset(out.getData() + (l * out.getWidth() + in1.getWidth()) * 3,
       '\0', in2.getWidth() * 3);
      }
    }

  }

  return true;
}

void TrivialTransform::printOptions(){
  std::cout << "-v\tVertical Mosic" << std::endl;
  std::cout << "-h\tHozional Mosic (default)" << std::endl;
}
```

absolute.cpp

```
#include <iostream>
#include <string.h>

#include "absolute.h"

AbsoluteTransform::AbsoluteTransform(){
  name = "absolute";
  ishue = false;
}

AbsoluteTransform::~AbsoluteTransform(){

}

bool AbsoluteTransform::doTransform(Image &out, Image &in1, Image &in2,
    int argc, char** argv){

  int xoffset = 0;
  int yoffset = 0;
  bool maxcommonstrip = false;

  for(int i = 0; i < argc; i++){
    std::cout << argv[i] << std::endl;

    if(strncmp(argv[i], "-x", 2) == 0){
      xoffset = atoi(argv[i] + 2);
    }else if(strncmp(argv[i], "-y", 2) == 0){
      yoffset = atoi(argv[i] + 2);
    }else if(strncmp(argv[i], "-s", 2) == 0){
```

```
      maxcommonstrip = true;
    }

  }


  std::cout << "Offsets, x: " << xoffset << " y: " << yoffset << std::endl;

  return doTransform(out, in1, in2, xoffset, yoffset, maxcommonstrip);

}


bool AbsoluteTransform::doTransform(Image  &out, Image &in1, Image &in2,
    int xoffset, int yoffset, bool maxcommonstrip){

  int xsize, ysize;

  if(xoffset >= 0){
    xsize = (in1.getWidth() > in2.getWidth() + xoffset)
      ? in1.getWidth() : (in2.getWidth() + xoffset);
  }else{
    xsize = (in2.getWidth() > in1.getWidth() - xoffset)
      ? in2.getWidth() : (in1.getWidth() - xoffset);
  }
  if(yoffset >= 0){
    if(maxcommonstrip){
      ysize = ((in1.getHeight() - yoffset) < in2.getHeight())
? (in1.getHeight() - yoffset) : in2.getHeight();
    }else{
      ysize = (in1.getHeight() > in2.getHeight() + yoffset)
? in1.getHeight() : (in2.getHeight() + yoffset);
    }
  }else{
    if(maxcommonstrip){
      ysize = ((in2.getHeight() + yoffset) < in1.getHeight())
? (in2.getHeight() + yoffset) : in1.getHeight();
    }else{
      ysize = (in2.getHeight() > in1.getHeight() - yoffset)
? in2.getHeight() : (in1.getHeight() - yoffset);
    }
  }

  out.setSize(xsize, ysize);

  if(xoffset >= 0){
    if(yoffset >= 0){
      return topleft(out, in1, in2, xoffset, yoffset, maxcommonstrip);
    }else{
      return bottomleft(out, in1, in2, xoffset, -yoffset, maxcommonstrip);
```

```
    }
  }else{
    if(yoffset >= 0){
      return bottomleft(out, in2, in1, -xoffset, yoffset, maxcommonstrip);
    }else{
      return topleft(out, in2, in1, -xoffset, -yoffset, maxcommonstrip);
    }
  }

  return false;

}

void AbsoluteTransform::printOptions(){
  std::cout << "-x\tX offset of second image" << std::endl;
  std::cout << "-y\tY offset of second image" << std::endl;
}


void AbsoluteTransform::setIsHue(bool val){
  ishue = val;
}

bool AbsoluteTransform::topleft(Image &out, Image &in1, Image &in2,
 int xoffset, int yoffset, bool maxcommonstrip){
  int line1, line2, lineo;

  line1 = line2 = lineo = 0;

  if(maxcommonstrip){
    line1 += yoffset;
  }else{
    for(int i = 0; i < yoffset; i++){
      memcpy(out.getData() + 3 * lineo * out.getWidth(),
     in1.getData() + 3 * line1 * in1.getWidth(), 3 * in1.getWidth());
      memset(out.getData() + 3 * lineo * out.getWidth() + 3 * in1.getWidth(),
     0, 3 * (out.getWidth() - in1.getWidth()));

      lineo++;
      line1++;
    }
  }

  while(line1 < in1.getHeight() && line2 < in2.getHeight()){
    memcpy(out.getData() + 3 * lineo * out.getWidth(),
   in1.getData() + 3 * line1 * in1.getWidth(), 3 * xoffset);

    for(int i = xoffset; i < in1.getWidth(); i++){
      if(!ishue){
(out.getData() + 3 * lineo * out.getWidth() + 3 * i)[0] =
```

```
  ((in1.getData() + 3 * line1 * in1.getWidth() + 3 * i)[0]
   + (in2.getData() + 3 * line2 * in2.getWidth() + 3 * (i - xoffset))[0]) / 2;
(out.getData() + 3 * lineo * out.getWidth() + 3 * i)[1] =
  ((in1.getData() + 3 * line1 * in1.getWidth() + 3 * i)[1]
   + (in2.getData() + 3 * line2 * in2.getWidth() + 3 * (i - xoffset))[1]) / 2;
      }else{
//is hue

int foo = in1.getPixel(i, line1)[0] - in2.getPixel(i - xoffset, line2)[0];
if(foo > 90)
  foo = foo - 180;
foo = foo / 2 + in1.getPixel(i, line1)[0];
if(foo >= 180){
  foo -= 180;
}
if(foo < 0){
  foo += 180;
}

out.getPixel(i, lineo)[0] = foo;
(out.getData() + 3 * lineo * out.getWidth() + 3 * i)[1] =
  ((in1.getData() + 3 * line1 * in1.getWidth() + 3 * i)[1]
   + (in2.getData() + 3 * line2 * in2.getWidth() + 3 * (i - xoffset))[1]) / 2;
      }

      (out.getData() + 3 * lineo * out.getWidth() + 3 * i)[2] =
((in1.getData() + 3 * line1 * in1.getWidth() + 3 * i)[2]
 + (in2.getData() + 3 * line2 * in2.getWidth() + 3 * (i - xoffset))[2]) / 2;
    }

    memcpy(out.getData() + 3 * lineo * out.getWidth() + 3 * in1.getWidth(),
   in2.getData() + 3 * line2 * in2.getWidth() + 3 * (in1.getWidth() - xoffset),
   3 * (xoffset + in2.getWidth() - in1.getWidth()));

    line1++;
    line2++;
    lineo++;
  }

  if(!maxcommonstrip){
    while(line1 < in1.getHeight()){
      memcpy(out.getData() + 3 * lineo * out.getWidth(),
     in1.getData() + 3 * line1 * in1.getWidth(), 3 * in1.getWidth());
      memset(out.getData() + 3 * lineo * out.getWidth() + 3 * in1.getWidth(),
     0, 3 * (out.getWidth() - in1.getWidth()));

      lineo++;
      line1++;
    }
```

```
    while(line2 < in2.getHeight()){

      memset(out.getData() + 3 * lineo * out.getWidth(), 0, 3 * xoffset);
      memcpy(out.getData() + 3 * lineo * out.getWidth() + 3 * xoffset,
     in2.getData() + 3 * line2 * in2.getWidth(), 3 * in2.getWidth());
      lineo++;
      line2++;
    }
  }


  return true;
}

bool AbsoluteTransform::bottomleft(Image &out, Image &in1, Image &in2,
  int xoffset, int yoffset, bool maxcommonstrip){
int line1, line2, lineo;

  line1 = line2 = lineo = 0;

  if(maxcommonstrip){
    line2 = yoffset;
  }else{
    for(int i = 0; i < yoffset; i++){
      memset(out.getData() + 3 * lineo * out.getWidth(), 0, 3 * xoffset);
      memcpy(out.getData() + 3 * lineo * out.getWidth() + 3 * xoffset,
     in2.getData() + 3 * line2 * in2.getWidth(), 3 * in2.getWidth());

      lineo++;
      line2++;
    }
  }

  while(line1 < in1.getHeight() && line2 < in2.getHeight()){
    memcpy(out.getData() + 3 * lineo * out.getWidth(),
   in1.getData() + 3 * line1 * in1.getWidth(), 3 * xoffset);

    for(int i = xoffset; i < in1.getWidth(); i++){
if(!ishue){
(out.getData() + 3 * lineo * out.getWidth() + 3 * i)[0] =
  ((in1.getData() + 3 * line1 * in1.getWidth() + 3 * i)[0]
   + (in2.getData() + 3 * line2 * in2.getWidth() + 3 * (i - xoffset))[0]) / 2;
(out.getData() + 3 * lineo * out.getWidth() + 3 * i)[1] =
  ((in1.getData() + 3 * line1 * in1.getWidth() + 3 * i)[1]
   + (in2.getData() + 3 * line2 * in2.getWidth() + 3 * (i - xoffset))[1]) / 2;
      }else{
//is hue

int foo = in1.getPixel(i, line1)[0] - in2.getPixel(i - xoffset, line2)[0];
if(foo > 90)
```

```
  foo = foo - 180;
foo = foo / 2 + in1.getPixel(i, line1)[0];
if(foo >= 180){
  foo -= 180;
}
if(foo < 0){
  foo += 180;
}

out.getPixel(i, lineo)[0] = foo;
(out.getData() + 3 * lineo * out.getWidth() + 3 * i)[1] =
  ((in1.getData() + 3 * line1 * in1.getWidth() + 3 * i)[1]
   + (in2.getData() + 3 * line2 * in2.getWidth() + 3 * (i - xoffset))[1]) / 2;


      }

      (out.getData() + 3 * lineo * out.getWidth() + 3 * i)[2] =
((in1.getData() + 3 * line1 * in1.getWidth() + 3 * i)[2]
 + (in2.getData() + 3 * line2 * in2.getWidth() + 3 * (i - xoffset))[2]) / 2;

    }

    memcpy(out.getData() + 3 * lineo * out.getWidth() + 3 * in1.getWidth(),
   in2.getData() + 3 * line2 * in2.getWidth() + 3 * (in1.getWidth() - xoffset),
   3 * (xoffset + in2.getWidth() - in1.getWidth()));

    line1++;
    line2++;
    lineo++;
  }

  if(!maxcommonstrip){
    while(line1 < in1.getHeight()){
      memcpy(out.getData() + 3 * lineo * out.getWidth(),
     in1.getData() + 3 * line1 * in1.getWidth(), 3 * in1.getWidth());
      memset(out.getData() + 3 * lineo * out.getWidth() + 3 * in1.getWidth(),
     0, 3 * (out.getWidth() - in1.getWidth()));

      lineo++;
      line1++;
    }

    while(line2 < in2.getHeight()){

      memset(out.getData() + 3 * lineo * out.getWidth(), 0, 3 * xoffset);
      memcpy(out.getData() + 3 * lineo * out.getWidth() + 3 * xoffset,
     in2.getData() + 3 * line2 * in2.getWidth(), 3 * in2.getWidth());
      lineo++;
      line2++;
```

```
    }

  }

  return true;
}
```

hierarchical.cpp

```
#include <map>
#include <iostream>
#include <sstream>
#include <string.h>
#include <math.h>

#include "absolute.h"

#include "hierarchical.h"

HierarchicalTransform::HierarchicalTransform(){

}

HierarchicalTransform::~HierarchicalTransform(){

}

bool HierarchicalTransform::doTransform(Image &out, Image &in1, Image &in2,
int argc, char** argv){

  std::map<int, Image> images1;
  std::map<int, Image> images2;

  int level = 0;

  //args
  bool intermediate = false;
  bool verbose = false;
  int lowestchoice = 0;
  int errormetric = 0;
  int maxlevel = 40;

  int xminoverlap = 0;
  int xmaxoverlap = 0;
  int yminoverlap = 0;
  int ymaxoverlap = 0;

  bool mcs = false;
```

```
  for(int i = 0; i < argc; i++){
    if(strncmp(argv[i], "-i", 2) == 0){
      intermediate = true;
    }else if(strncmp(argv[i], "-V", 2) == 0){
      verbose = true;
    }else if(strncmp(argv[i], "-c", 2) == 0){
      lowestchoice = atoi(argv[i] + 2);
    }else if(strncmp(argv[i], "-e", 2) == 0){
      errormetric = atoi(argv[i] + 2);
    }else if(strncmp(argv[i], "-l", 2) == 0){
      maxlevel = atoi(argv[i] + 2);
    }else if(strncmp(argv[i], "-s", 2) == 0){
      mcs = true;
    }else if(strncmp(argv[i], "-o", 2) == 0){
      //set max and min overlap
      i++;
      xmaxoverlap = atoi(argv[i++]);
      xminoverlap = atoi(argv[i++]);
      yminoverlap = atoi(argv[i++]);
      ymaxoverlap = atoi(argv[i]);

    }else if(strncmp(argv[i], "-t", 2) == 0){
      if(argv[i][2] == '\0'){
//set max and min typical overlap
xmaxoverlap = 95; //5%
xminoverlap = 50; //50%
yminoverlap = -20; //20% above
ymaxoverlap = 20; //20% down
maxlevel = 4;
      }else if(argv[i][2] == 'e'){
//set max and min typical overlap
xmaxoverlap = 95; //5%
xminoverlap = -95; //-5%
yminoverlap = -95; //95% above
ymaxoverlap = 95; //95% down
maxlevel = 4;
      }
    }

  }

  Image ci1 = in1.toGray();
  Image ci2 = in2.toGray();

  while(ci1.getWidth() > 6 && ci1.getHeight() > 6 && ci2.getWidth() > 6
&& ci2.getHeight() > 6 && maxlevel > level){

    images1[level] = ci1;
    images2[level] = ci2;
```

```
    ci1 = ci1.scaleDown();
    ci2 = ci2.scaleDown();

    if(verbose){
      std::cout << "Image1 size " << ci1.getWidth() << " x "
<< ci1.getHeight() << std::endl;
      std::cout << "Image2 size " << ci2.getWidth() << " x "
<< ci2.getHeight() << std::endl;
    }

    if(intermediate){
      std::ostringstream formater;
      formater << level + 1;
      std::string slev = formater.str();

      ci1.write(std::string("i1l") + slev + std::string(".jpg"));
      ci2.write(std::string("i2l") + slev + std::string(".jpg"));
    }

    level++;

  }

  images1[level] = ci1;
  images2[level] = ci2;

  //do hierarchical matching

  int xminoffset = - ci2.getWidth() + 1, xmaxoffset = ci1.getWidth();
  int yminoffset = - ci2.getHeight() + 1, ymaxoffset = ci2.getHeight();

  if(xminoverlap != xmaxoverlap){
    xminoffset = (ci1.getWidth() * xminoverlap / 100 < xminoffset)
      ? xminoffset : (ci1.getWidth() * xminoverlap / 100);
    xmaxoffset = (ci1.getWidth() * xmaxoverlap / 100 > xmaxoffset)
      ? xmaxoffset : (ci1.getWidth() * xmaxoverlap / 100);
  }
  if(yminoverlap != ymaxoverlap){
    yminoffset = (ci1.getHeight() * yminoverlap / 100 < yminoffset)
      ? yminoffset : (ci1.getHeight() * yminoverlap / 100);
    ymaxoffset = (ci1.getHeight() * ymaxoverlap / 100 > ymaxoffset)
      ? ymaxoffset : (ci1.getHeight() * ymaxoverlap / 100);
  }

  int newx = 0, newy = 0;

  while(level >= 0){


    std::cout << "xmim, xmax, ymin, ymax " << xminoffset << " "
```

```
      << xmaxoffset << " " << yminoffset << " "
      << ymaxoffset << std::endl;

    ci1 = images1[level];
    ci2 = images2[level];

    float error[ymaxoffset - yminoffset][xmaxoffset - xminoffset];

    for(int yoffset = yminoffset; yoffset < ymaxoffset; yoffset++){
      for(int xoffset = xminoffset; xoffset < xmaxoffset; xoffset++){
error[yoffset - yminoffset][xoffset - xminoffset] = 0;
int i = 0;
for(int y = (yoffset > 0) ? yoffset : 0;
    y < ci1.getHeight() && ci2.getHeight() > y - yoffset; y++){


  for(int x = (xoffset > 0) ? xoffset : 0;
      x < ci1.getWidth() && ci2.getWidth() > x - xoffset; x++){
    switch(errormetric){
    case 0:
      error[yoffset - yminoffset][xoffset - xminoffset] +=
abs(ci1.getData()[(y * ci1.getWidth() + x) * 3]
    - ci2.getData()[((y - yoffset) * ci2.getWidth() + x - xoffset) * 3]);
      break;

    case 1:
      error[yoffset - yminoffset][xoffset - xminoffset] +=
pow(ci1.getData()[(y * ci1.getWidth() + x) * 3]
    - ci2.getData()[((y - yoffset) * ci2.getWidth() + x - xoffset) * 3], 2);
      break;

    case 2:
      error[yoffset - yminoffset][xoffset - xminoffset] +=
fabs(pow(ci1.getData()[(y * ci1.getWidth() + x) * 3], 2.0)
     - pow(ci2.getData()[((y - yoffset) * ci2.getWidth() + x - xoffset) * 3], 2));
      break;

    default:
      std::cerr << "No error metric"  << std::endl;
      exit(0);
      break;
    }
    i++;

  }
}
if(i != 0)
  switch(errormetric){
  case 0:
    error[yoffset - yminoffset][xoffset - xminoffset] /= (float)i;
```

```
    break;

  case 1:
  case 2:
    error[yoffset - yminoffset][xoffset - xminoffset] =
      sqrt(error[yoffset - yminoffset][xoffset - xminoffset] / (float)i);
  }
else
  error[yoffset - yminoffset][xoffset - xminoffset] = 257.0;

      }

    }

    // find the smallest

    float smallerr = 256.0;
    newx = 0;
    newy = 0;
    Image errmap;

    if(intermediate){
      errmap.setSize(xmaxoffset - xminoffset, ymaxoffset - yminoffset);
    }

    for(int yoffset = yminoffset; yoffset < ymaxoffset; yoffset++){
      for(int xoffset = xminoffset; xoffset < xmaxoffset; xoffset++){
if(verbose)
  std::cout << "match at " << xoffset << " x " << yoffset
    << " with error " << error[yoffset - yminoffset][xoffset - xminoffset]
    <<std::endl;

if(intermediate){
  unsigned char* pixel =
    errmap.getPixel(xoffset - xminoffset, yoffset - yminoffset);
  unsigned int value =
    (unsigned int)(error[yoffset - yminoffset][xoffset - xminoffset])
    * ((errormetric != 2)? 4 : 2);
  if(value > 255)
    value = 255;
  pixel[0] = pixel[1] = pixel[2] = value;
}

switch(lowestchoice){
case 0:
  if(error[yoffset - yminoffset][xoffset - xminoffset] < smallerr){
    smallerr = error[yoffset - yminoffset][xoffset - xminoffset];
    newx = xoffset;
    newy = yoffset;
  }
```

```
  break;

case 1:
  {
    float localerr = error[yoffset - yminoffset][xoffset - xminoffset];
    int errcount = 1;
;
    if(yoffset > yminoffset){
      localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset];
      errcount++;
      if(xoffset > xminoffset){
localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset - 1];
errcount++;
      }
      if(xoffset < xmaxoffset - 1){
localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset + 1];
errcount++;
      }
    }

    if(yoffset < ymaxoffset - 1){
      localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset];
      errcount++;
      if(xoffset > xminoffset){
localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset - 1];
errcount++;
      }
      if(xoffset < xmaxoffset - 1){
localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset + 1];
errcount++;
      }
    }

    if(xoffset > xminoffset){
       localerr += error[yoffset - yminoffset][xoffset - xminoffset - 1];
errcount++;
    }

    if(xoffset < xmaxoffset - 1){
      localerr += error[yoffset - yminoffset][xoffset - xminoffset + 1];
errcount++;
    }

    if(localerr / (float)errcount < smallerr){

      smallerr = localerr / (float)errcount;
      newx = xoffset;
      newy = yoffset;
    }
```

```
  }
  break;

case 2:
  {
    float localerr = error[yoffset - yminoffset][xoffset - xminoffset];
    int errcount = 1;
;
    if(yoffset > yminoffset){
      localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset];
      errcount++;
      if(xoffset > xminoffset){
localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset - 1];
errcount++;
      }
      if(xoffset < xmaxoffset - 1){
localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset + 1];
errcount++;
      }
    }

    if(yoffset < ymaxoffset - 1){
      localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset];
      errcount++;
      if(xoffset > xminoffset){
localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset - 1];
errcount++;
      }
      if(xoffset < xmaxoffset - 1){
localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset + 1];
errcount++;
      }
    }

    if(xoffset > xminoffset){
       localerr += error[yoffset - yminoffset][xoffset - xminoffset - 1];
errcount++;
    }

    if(xoffset < xmaxoffset - 1){
      localerr += error[yoffset - yminoffset][xoffset - xminoffset + 1];
errcount++;
    }

    float distmod =
      pow(sqrt(yoffset * yoffset + xoffset * xoffset)
  / sqrt(ci1.getWidth() * ci1.getWidth() + ci1.getHeight() * ci1.getHeight()),
  3) + 1;

    if(localerr * distmod / (float)errcount < smallerr){
```

```
      smallerr = localerr * distmod / (float)errcount;
      newx = xoffset;
      newy = yoffset;
    }

  }
  break;

default:
  std::cerr << "No lowest choice made, bailing" << std::endl;
  exit(0);
  break;
}
      }
    }

    if(intermediate){
      std::ostringstream formater;
      formater << level + 1;
      std::string slev = formater.str();

      errmap.write(std::string("err-map") + slev + std::string(".jpg"));

    }


    std::cout << "At level " << level << " the best match is at "
      << newx << " x " << newy << " with error " << smallerr <<std::endl;

    yminoffset = newy * 2 - 5;
    if(yminoffset < -images2[level - 1].getHeight() + 1)
      yminoffset = -images2[level - 1].getHeight() + 1;
    ymaxoffset = newy * 2 + 5;
    if(ymaxoffset > images1[level - 1].getHeight())
      ymaxoffset = images1[level - 1].getHeight();

    xminoffset = newx * 2 - 5;
    if(xminoffset < - images2[level - 1].getWidth() + 1)
      xminoffset = - images2[level - 1].getWidth() + 1;
    xmaxoffset = newx * 2 + 5;
    if(xmaxoffset > images1[level - 1].getWidth())
      xmaxoffset = images1[level - 1].getWidth();

    level--;
  }

  AbsoluteTransform trans;

  return trans.doTransform(out, in1, in2, newx, newy, mcs);
```

```
}

void HierarchicalTransform::printOptions(){

}
```

huematchequ.cpp

```
#include <map>
#include <iostream>
#include <sstream>
#include <string.h>
#include <math.h>

#include "absolute.h"

#include "huematchequ.h"


bool HueMatchEquTransform::doTransform(Image &out, Image &in1, Image &in2,
       int argc, char** argv){

  std::map<int, Image> images1;
  std::map<int, Image> images2;

  int level = 0;

  //args
  bool intermediate = false;
  bool equ = false;
  int lowestchoice = 0;
  int errormetric = 0;
  int maxlevel = 40;

  int xminoverlap = 0;
  int xmaxoverlap = 0;
  int yminoverlap = 0;
  int ymaxoverlap = 0;

  bool mcs = false;

  for(int i = 0; i < argc; i++){
    if(strncmp(argv[i], "-i", 2) == 0){
      intermediate = true;
    }else if(strncmp(argv[i], "-q", 2) == 0){
      equ = true;
    }else if(strncmp(argv[i], "-c", 2) == 0){
      lowestchoice = atoi(argv[i] + 2);
```

```
    }else if(strncmp(argv[i], "-e", 2) == 0){
      errormetric = atoi(argv[i] + 2);
    }else if(strncmp(argv[i], "-l", 2) == 0){
      maxlevel = atoi(argv[i] + 2);
    }else if(strncmp(argv[i], "-s", 2) == 0){
      mcs = true;
    }else if(strncmp(argv[i], "-o", 2) == 0){
      //set max and min overlap
      i++;
      xmaxoverlap = atoi(argv[i++]);
      xminoverlap = atoi(argv[i++]);
      yminoverlap = atoi(argv[i++]);
      ymaxoverlap = atoi(argv[i]);

    }else if(strncmp(argv[i], "-t", 2) == 0){
      //set max and min typical overlap
      xmaxoverlap = 95; //5%
      xminoverlap = 50; //50%
      yminoverlap = -20; //20% above
      ymaxoverlap = 20; //20% down
      maxlevel = 4;
    }

  }

  Image ci1 = in1.toHSV();
  Image ci2 = in2.toHSV();

  if(intermediate){
    ci1.write("testhvs1.jpg");
    ci2.write("testhvs2.jpg");

    Image cb1 = ci1.fromHSV();
    Image cb2 = ci2.fromHSV();

    cb1.write("testhvs3.jpg");
    cb2.write("testhvs4.jpg");
  }

  while(ci1.getWidth() > 6 && ci1.getHeight() > 6 && ci2.getWidth() > 6
&& ci2.getHeight() > 6 && maxlevel > level){

    images1[level] = ci1;
    images2[level] = ci2;

    ci1 = ci1.scaleDownHSV();
    ci2 = ci2.scaleDownHSV();

    if(intermediate){
      std::ostringstream formater;
```

```
      formater << level + 1;
      std::string slev = formater.str();

      ci1.write(std::string("i1l") + slev + std::string(".jpg"));
      ci2.write(std::string("i2l") + slev + std::string(".jpg"));
    }

    level++;
  }


  images1[level] = ci1;
  images2[level] = ci2;

  //do hierarchical matching

  int xminoffset = - ci2.getWidth() + 1, xmaxoffset = ci1.getWidth();
  int yminoffset = - ci2.getHeight() + 1, ymaxoffset = ci2.getHeight();

  if(xminoverlap != xmaxoverlap){
    xminoffset = (ci1.getWidth() * xminoverlap / 100 < xminoffset)
      ? xminoffset : (ci1.getWidth() * xminoverlap / 100);
    xmaxoffset = (ci1.getWidth() * xmaxoverlap / 100 > xmaxoffset)
      ? xmaxoffset : (ci1.getWidth() * xmaxoverlap / 100);
  }
  if(yminoverlap != ymaxoverlap){
    yminoffset = (ci1.getHeight() * yminoverlap / 100 < yminoffset)
      ? yminoffset : (ci1.getHeight() * yminoverlap / 100);
    ymaxoffset = (ci1.getHeight() * ymaxoverlap / 100 > ymaxoffset)
      ? ymaxoffset : (ci1.getHeight() * ymaxoverlap / 100);
  }

  int newx = 0, newy = 0;

  while(level >= 0){

    std::cout << "xmim, xmax, ymin, ymax " << xminoffset << " "
      << xmaxoffset << " " << yminoffset << " " << ymaxoffset << std::endl;

    ci1 = images1[level];
    ci2 = images2[level];

    float error[ymaxoffset - yminoffset][xmaxoffset - xminoffset];

    for(int yoffset = yminoffset; yoffset < ymaxoffset; yoffset++){
      for(int xoffset = xminoffset; xoffset < xmaxoffset; xoffset++){
error[yoffset - yminoffset][xoffset - xminoffset] = 0;
int i = 0;
for(int y = (yoffset > 0) ? yoffset : 0;
    y < ci1.getHeight() && ci2.getHeight() > y - yoffset; y++){
```

```
  for(int x = (xoffset > 0) ? xoffset : 0;
      x < ci1.getWidth() && ci2.getWidth() > x - xoffset; x++){
    switch(errormetric){
    case 0:

      {
int foo = abs(ci1.getPixel(x, y)[0]
      - ci2.getPixel(x - xoffset, y - yoffset)[0]);
if(foo > 90)
  foo = 180 - foo;
error[yoffset - yminoffset][xoffset - xminoffset] += foo;
      }
      break;

    case 1:

      {
float x1, x2, y1, y2;
x1 = (float)(ci1.getPixel(x, y)[1])
  * cos(M_PI / 90.0 * (double)ci1.getPixel(x, y)[0]);
y1 = (float)ci1.getPixel(x, y)[1]
  * sin(M_PI / 90 * (double)ci1.getPixel(x, y)[0]);
x2 = (float)ci2.getPixel(x - xoffset, y - yoffset)[1]
  * cos(M_PI / 90.0 * (double)ci2.getPixel(x - xoffset, y - yoffset)[0]);
y2 = (float)ci2.getPixel(x - xoffset, y - yoffset)[1]
  * sin(M_PI / 90.0 * (double)ci2.getPixel(x - xoffset, y - yoffset)[0]);
error[yoffset - yminoffset][xoffset - xminoffset]
  += (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
      }
      break;

    case 2:
      error[yoffset - yminoffset][xoffset - xminoffset] +=
abs(ci1.getData()[(y * ci1.getWidth() + x) * 3]
    - ci2.getData()[((y - yoffset) * ci2.getWidth() + x - xoffset) * 3])
+ abs(ci1.getPixel(x, y)[1] - ci2.getPixel(x, y)[1]);
      break;

    default:
      std::cerr << "No error metric"  << std::endl;
      exit(0);
      break;
    }
    i++;

  }
}
if(i != 0)
```

```
  switch(errormetric){
  case 0:
  case 1:
    error[yoffset - yminoffset][xoffset - xminoffset] /= (float)i;
    break;

  case 2:
    error[yoffset - yminoffset][xoffset - xminoffset] =
      sqrt(error[yoffset - yminoffset][xoffset - xminoffset] / (float)i);
  }
else
  error[yoffset - yminoffset][xoffset - xminoffset] = 25700.0;

      }

    }

    // find the smallest

    float smallerr = 25600.0;
    newx = 0;
    newy = 0;

    for(int yoffset = yminoffset; yoffset < ymaxoffset; yoffset++){
      for(int xoffset = xminoffset; xoffset < xmaxoffset; xoffset++){
std::cout << "match at " << xoffset << " x " << yoffset << " with error "
  << error[yoffset - yminoffset][xoffset - xminoffset] <<std::endl;
switch(lowestchoice){
case 0:
  if(error[yoffset - yminoffset][xoffset - xminoffset] < smallerr){
    smallerr = error[yoffset - yminoffset][xoffset - xminoffset];
    newx = xoffset;
    newy = yoffset;
  }
  break;

case 1:
  {
    float localerr = error[yoffset - yminoffset][xoffset - xminoffset];
    int errcount = 1;
;
    if(yoffset > yminoffset){
      localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset];
      errcount++;
      if(xoffset > xminoffset){
localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset - 1];
errcount++;
      }
      if(xoffset < xmaxoffset - 1){
localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset + 1];
```

```
errcount++;
      }
    }

    if(yoffset < ymaxoffset - 1){
      localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset];
      errcount++;
      if(xoffset > xminoffset){
localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset - 1];
errcount++;
      }
      if(xoffset < xmaxoffset - 1){
localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset + 1];
errcount++;
      }
    }

    if(xoffset > xminoffset){
       localerr += error[yoffset - yminoffset][xoffset - xminoffset - 1];
errcount++;
    }

    if(xoffset < xmaxoffset - 1){
      localerr += error[yoffset - yminoffset][xoffset - xminoffset + 1];
errcount++;
    }

    if(localerr / (float)errcount < smallerr){

      smallerr = localerr / (float)errcount;
      newx = xoffset;
      newy = yoffset;
    }

  }
  break;

case 2:
  {
    float localerr = error[yoffset - yminoffset][xoffset - xminoffset];
    int errcount = 1;
;
    if(yoffset > yminoffset){
      localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset];
      errcount++;
      if(xoffset > xminoffset){
localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset - 1];
errcount++;
      }
      if(xoffset < xmaxoffset - 1){
```

```
localerr += error[yoffset - yminoffset - 1][xoffset - xminoffset + 1];
errcount++;
      }
    }

    if(yoffset < ymaxoffset - 1){
      localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset];
      errcount++;
      if(xoffset > xminoffset){
localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset - 1];
errcount++;
      }
      if(xoffset < xmaxoffset - 1){
localerr += error[yoffset - yminoffset + 1][xoffset - xminoffset + 1];
errcount++;
      }
    }

    if(xoffset > xminoffset){
       localerr += error[yoffset - yminoffset][xoffset - xminoffset - 1];
errcount++;
    }

    if(xoffset < xmaxoffset - 1){
      localerr += error[yoffset - yminoffset][xoffset - xminoffset + 1];
errcount++;
    }

    float distmod =
      pow(sqrt(yoffset * yoffset + xoffset * xoffset) /
  sqrt(ci1.getWidth() * ci1.getWidth()
       + ci1.getHeight() * ci1.getHeight()), 3) + 1;

    if(localerr * distmod / (float)errcount < smallerr){

      smallerr = localerr * distmod / (float)errcount;
      newx = xoffset;
      newy = yoffset;
    }

  }
  break;

default:
  std::cerr << "No lowest choice made, bailing" << std::endl;
  exit(0);
  break;
}
      }
    }
```

```
    std::cout << "At level " << level << " the best match is at " << newx
      << " x " << newy << " with error " << smallerr <<std::endl;

    yminoffset = newy * 2 - 5;
    if(yminoffset < -images2[level - 1].getHeight() + 1)
      yminoffset = -images2[level - 1].getHeight() + 1;
    ymaxoffset = newy * 2 + 5;
    if(ymaxoffset > images1[level - 1].getHeight())
      ymaxoffset = images1[level - 1].getHeight();

    xminoffset = newx * 2 - 5;
    if(xminoffset < - images2[level - 1].getWidth() + 1)
      xminoffset = - images2[level - 1].getWidth() + 1;
    xmaxoffset = newx * 2 + 5;
    if(xmaxoffset > images1[level - 1].getWidth())
      xmaxoffset = images1[level - 1].getWidth();

    level--;
  }

  if(equ){
    int sumdiff = 0;
    int pcount = 0;
    for(int y = (newy > 0) ? newy : 0; y < ci1.getHeight()
  && ci2.getHeight() > y - newy; y++){
      for(int x = (newx > 0) ? newx : 0; x < ci1.getWidth()
    && ci2.getWidth() > x - newx; x++){
sumdiff += ci1.getPixel(x, y)[2]
  - ci2.getPixel(x - newx, y - newy)[2];
pcount++;
      }
    }
    std::cout << "Equ: sumdiff = " << sumdiff << ", pcount = "
      << pcount << ", meandiff = " << (sumdiff / pcount) << std::endl;

    int meandiff = (sumdiff / pcount);
    if(meandiff != 0){

      for(int y = 0; y < ci2.getHeight(); y++){
for(int x = 0; x < ci2.getWidth(); x++){
  int newval = ci2.getPixel(x, y)[2] + meandiff;
  if(newval > 255){
    newval = 255;
  }
  if(newval < 0){
    newval = 0;
  }
  ci2.getPixel(x, y)[2] = newval;
}
```

```
      }
      if(intermediate){
ci2.write("int_correct.jpg");
      }
    }

  }

  AbsoluteTransform trans;
  Image foo;

  trans.setIsHue(true);

  bool rtn = trans.doTransform(foo, ci1, ci2, newx, newy, mcs);

  out = foo.fromHSV();
  //out = foo;

  return rtn;

}

void HueMatchEquTransform::printOptions(){

}
```