CS 103 – Chroma Key

1 Introduction

In this assignment you will perform a green-screen or **chroma key** operation. You will be given an input image (e.g. Stephen Colbert) with a green background. First, you will identify which parts of that image form the foreground (the mask) and which are the background (the key). You will replace the chroma key portion of the image with the corresponding portions of a new background image (e.g. USC campus). This technique is frequently used in the TV, movie, and digital imaging industries.

Green-Screen Image[1]

Background

Chroma-Key Result

2 What you will learn

This assignment will expose you to simple image representation and manipulation techniques as well as familiarize you with C/C++ operations on 2D arrays.

1. Understand multiple file compilation units and their linkage.
2. Use command line arguments to provide input to the program vs. interactive user input
3. Understand image representation as a collection of pixels and understand the RGB color space
4. Apply knowledge of arrays (including multi-dimensional arrays) to implement an image-processing application
5. Create, develop, and evaluate your own processing approach to perform the operation

3 Background Information and Notes

Image processing is a major subfield of computer science and electrical engineering and to a lesser extent biomedical engineering. Graphics are usually represented via

[1] Image taken from http://www.comedycentral.com/colbertreport/videos.jhtml?videoId=79952

two methods: vector or bitmap. Vector graphics take a more abstract approach and use mathematical equations to represent lines, curves, polygons and their fill color. When a program opens the vector image it has to translate those equations and render the image. This vector approach is often used to represent clipart and 3D animations. Bitmap images take the opposite approach and simply represent the image as a 2D array of pixels. Each pixel is a small dot or square of color. The bitmap approach is used most commonly for pictures, video, and other images. In addition, bitmaps do not force us to translate the vector equations, and thus are simpler to manipulate for our purposes.

Color bitmaps can use one of several different color representations. The simplest of these methods is probably the RGB color space (HSL, HSV, & CMYK are others), where each pixel has separate red, green, and blue components that are combined to produce the desired color. Usually each component is an 8-bit (`unsigned char` in C) value. Given 3-values this yields a total of 24-bits for representing a specific color (known as 24-bit color = 2^{24} = 16 million unique colors). Storing a red, green and blue value for each pixel can be achieved using 3 separate 2D arrays (one for each RGB component) or can be combined into a 3D array with dimensions [256][256][3] as shown below.
[Note: BMP image files use a simple format like this one and thus will be the file format used in our lab.]

Color	RGB Value
White	255,255,255
Red	255,0,0
Yellow	255,255,0
Orange	255,128,0
Blue	0,0,255
Green	0,255,0
Purple	255,0,255
Black	0,0,0

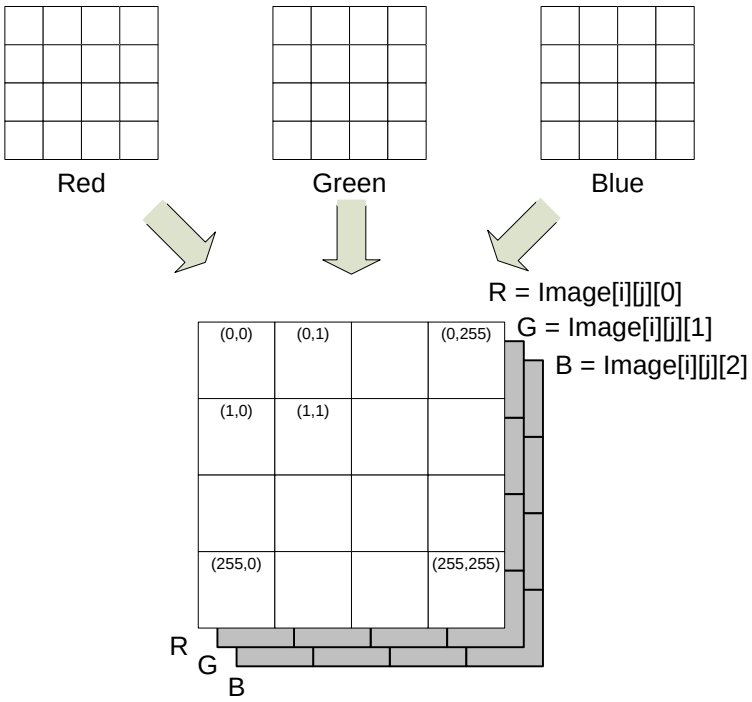

Figure 1 - Representation of a color bitmap as a 3D array

The primary task in performing the chroma key operation is to determine which pixels of the image should be classified as the chroma key (green). Part of the problem is that shadows and other artifacts may make the background pixels significantly lighter or darker shades of the key color, especially the closer they are to the foreground image. We would like an algorithm that is robust enough to classify these outlier pixels correctly without incorrectly classifying pixels of the foreground image. Whatever method we use, the goal is to create a 2D "mask" array that marks foreground image pixels (say with a 1) vs. chroma key pixels (with a 0). Once this mask is created, we can easily create the output image by using the mask to select pixels from the input or background image. This process is shown in Figure 2.

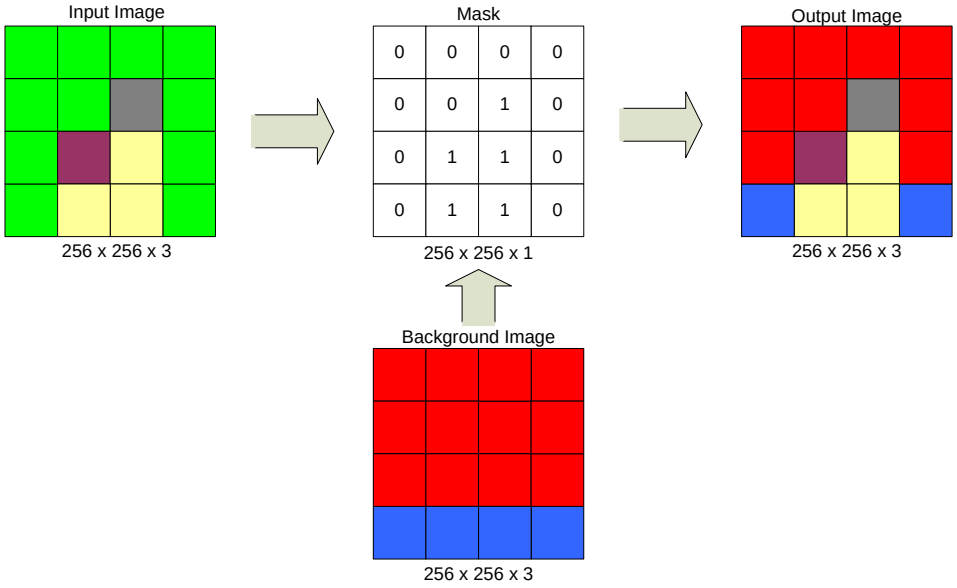

Figure 2 - Chroma key process

Many methods could be devised for determining whether a pixel is part of the chroma key. One may think of simply checking if a pixel's green component is above a certain value. However, usually this does not work because light colors like white and gray are represented as the superposition of all three RGB colors [i.e. white = (255,255,255)].

A better approach may be to think of each pixel's RGB components as a coordinate (vector) of a point in 3D space. Next, take some number of sample pixels that we would know to be the chroma key color (ones near the edges for example). We can average them to create a reference point that represents the average chroma key color. Then any pixel (RGB point) within a certain distance from the reference can be considered part of the chroma key while pixels (RGB points) further away will be considered part of the foreground image. This method can work for any chroma key but does require some trial and error to set the threshold distance for which pixel values will be determined to be the chroma key or foreground image. You will

need to experiment with this in your lab. [Note: The example image on the first page was created using this method.]

Many other methods could also be devised. One desirable feature that would make the algorithm more robust is automatic threshold determination (i.e. no user-interaction). An "adaptive" algorithm that can determine the parameters from the image values itself is desirable. Consider this feature when you create your own method.

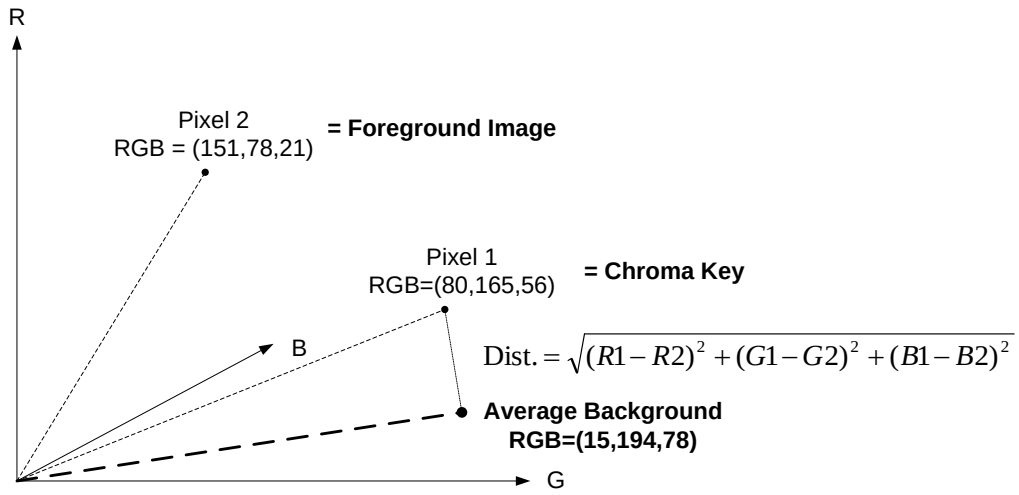

Figure 3 - RGB Color Space and Distance Determination

C/C++ Language Implementation

Some features of C that we will utilize include:

2- and 3-D Arrays: We will store the mask and images in 2- and 3-D arrays, respectively. Recall the declaration and access syntax:

```
int example2d[10][5];
unsigned char array3d[256][256][3];

x = example2d[9][4];    // accesses element in the bottom row, last column
pixel_red = array3d[255][255][0];  // access lower, left pixel's red value
```

Note that each of the RED, GREEN, and BLUE components of a pixel is an 8-bit value, logically between 0 and 255. Thus we will use the `unsigned char` type to represent these numbers.

Also, the library functions that we will provide use the convention that the **first index is the row** and the **second index is the column** (with the **third index being the R,G, or B value**.)

Math functions: You will likely need the ability to compute the square root of a number. This `sqrt` function is part of the `cmath` library.

BMP Library & File I/O: For this lab, we will provide you predefined functions that you can use to read in and write out .bmp image files. These functions are given in the compiled library `bmplib.o` and are prototyped in the header file `bmplib.h`. Be sure to include `bmplib.h` in your program by adding the following line with the other #include statements.

```
#include "bmplib.h"
```

The functions you you will use are "readRGBBMP" and "writeRGBBMP". Prototypes are shown below. You must pass each function a character array (text string) of the filename you wish to read/write and a 256x256x3 array of unsigned char's (8-bit values) that represent the data to be read/written.

```
int readRGBBMP(char filename[], unsigned char inputImage[][256][3]);
int writeRGBBMP(char filename[], unsigned char outputImage[][256][3]);
```

Note: These functions return 0 if successful and non-zero if they cannot open, read, or write the particular file.

For debugging purposes, you may also use the function
`void showRGBBMP(unsigned char outputImage[][256][3]);`
that displays an image. However, it may not work if you are not using the VM.

Command Line Arguments: Rather than prompting the user during your program to enter the files you will process, we will pass the filenames to your program as command line arguments. Command line arguments provide a way to pass a program some initial input values without having to prompt the user explicitly when your program executes. Most programs provide this kind of feature. E.g. in Windows from the Start..Run box, type "notepad mydoc.txt". This will start notepad and attempt to open a file named mydoc.txt without requiring you to use the GUI interface. Your OS provides this ability by parsing the command line when you start your program and passing the additional command line words as arguments: `int argc` and `char *argv[]` to the main() routine.

```
int main(int argc, char *argv[]) { ... }
```

The `argc` value is an integer indicating how many command line arguments were entered (note that the executable program name is included in the count, so argc will always be at least one.) The `argv` argument is an array of character strings. For example if we run

```
$ ./chromakey input.bmp bgnd.bmp 40.5 output1.bmp output2.bmp
```

Then
`argc is 6`

```
argv[0] is "./chromakey"
argv[1] is "input.bmp"
argv[2] is "bgnd.bmp"
argv[3] is "40.5"
argv[4] is "output1.bmp"
argv[5] is "output2.bmp"
```

Note: Numeric arguments such as "40.5" are passed in as character (text) strings and need to be converted to the appropriate numeric types before operated upon. This can be accomplished with functions like "`atoi`" (ASCII to Integer) or "`atof`" (ASCII to floating point) which are defined in `<cstdlib>` and whose prototypes are shown below.

```
// returns the integer value of the number represented by the character string "string"
int atoi(char *string);

// returns the double floating point value of the number represented by the character string "string"
double atof(char *string);
```

As an example, the "40.5" argument can be converted by:

```
double x;
x = atof(argv[3]);
```

Multi-file compilation: Most real-world program contain more than one source code file and thus require the compiler to generate code and then link several files together to produce an executable. In this lab, we have provided `bmplib.h` and `bmplib.o`. `bmplib.o` is an "object" file (.o extension) representing the compiled (but not linked) functions to perform .BMP image I/O. It is not a text file but binary instructions and memory initialization commands.

An object file can be created from a C++ file by using the -c extension to the compiler.

```
$ compile -g -Wall -c bmplib.cpp
```

This command will create the `bmplib.o` file (if you have the .cpp source, which you don't).

`bmplib.h` is a header file that includes prototypes and other declarations that you can include into your C code that will allow you to call the functions in `bmplib.o`. To compile your code with the BMP functions and then link them together you could run:

```
$ compile -g -Wall -o chromakey bmplib.o chromakey.cpp
```

You can list any number of .o files and C files on the command line. The C files will be compiled and then linked together with all of the .o files specified producing an executable as output.

'make' and Makefiles: As more files become part of your program, you will not want to compile EVERY file again when you simply make a change to one file. However, keeping track of which files have changed and thus require recompilation can also become difficult. Enter the 'make' utility. This program takes as input a text file usually named 'Makefile' which includes command that identify the order dependencies of files on each other (i.e. if file 1 changes, then it may require re-compiling file 2 and file 3) and the commands to perform the compilation. It will then examine the modification times of the files and determine which files have changed and only compile those. It also acts as a *script* of compilation commands so you do not have to type in the 'gcc/g++' command line each time. For this program we will use make and provide you a Makefile shown below:

```
CC = g++
CFLAGS = -g -Wall
SRCS =  chromakey.cpp bmplib.cpp
OBJ = $(SRCS:.cpp=.o)
TARGET = chromakey

all: $(TARGET)

$(TARGET): $(OBJ)
	$(CC) $(CFLAGS) $(OBJ) -o $(TARGET)

.cpp.o:
	$(CC) $(CFLAGS) -c $<

bmplib.o : bmplib.h

clean:
	rm -f *.o $(TARGET) *~
```

In general, a Makefile is made of rules that indicate which files they are dependent upon. Each rule then has a set of commands that should be run when any dependency file changes.

```
rulename:  dependency1 dependency2 …
  <TAB>  command1
  <TAB>  command2…
```

Typing '`make`' at the command line will run the '`all`' target (this is the default rule to run) which depends on the `chromakey` target. The `chromakey` target depends on `chromakey.o` and `bmplib.h`. Thus, if those files have changed it will trigger the commands under the `chromakey` target to run. However a general rule (the `.cpp.o` or `.c.o` rules) exist to generate an .o file from .c or .cpp file if the .c/.cpp file is newer than the .o file. Thus, if we had changed our `chromakey.cpp` file, it would first be compiled to an object file using the `.cpp.o` rule and then the chromakey commands would be run to generate the executable. All of this happens by typing '`make`' and without your interaction. Finally, typing

'`make clean`' at the command prompt will run the rules associated with that '`clean`' target. The associated command will remove the files listed to "clean up" the directory. Having a '`clean`' rule is customary to remove all the intermediate compilation files from a directory.

More information about the 'make' utility and Makefiles can be found at:
http://www.eng.hawaii.edu/Tutor/Make/
http://frank.mtsu.edu/~csdept/FacilitiesAndResources/make.htm

4 Prelab

[Perform item 1 of the Procedure to download the sample code & BMP files then complete the rest of this prelab.]

The first task will be to better understand an example image, `colbert_gs.bmp` that we provide. On your VM you can run the 'geeqie' program to view .BMP graphics by typing:

```
$ geeqie colbert_gs.bmp     <or any other image file>
```

After opening the picture of Stephen Colbert, use the Dropper (Pixel Color) tool in the bottom right corner of the window. Positioning the cursor arrow over any pixel and the bottom status bar will show you the row, column coordinates as well as the RGB value of a pixel on the status bar.

(If you are not using the VM, paint/pbrush on Windows has an identical tool, and Mac users can either download http://paintbrush.sourceforge.net/ or use the online tool http://apps.pixlr.com/editor/ All of these tools are also useful for doing more tests and resizing images to the 256-by-256 size we use.)

Position your mouse cursor over the top left pixel (green screen). Record its RGB value. Next pick any of the flesh colored pixels on Stephen's arm. Record its RGB value. Find the RGB values for a pixel on his dark vest and on his white shirt. Compute the distance between the chroma key pixel and these three pixels (arm, vest, shirt). This may help in identifying a threshold value.

Pixel	RGB Value	Distance from Chroma Key
Sample Chroma Key (Top-left)	69, 140, 20	
Arm	254, 155, 113	207.6030
Vest	28, 29, 24	118.3976
Shirt	251, 216, 248	301.4697

5 Methodology

You must implement two different methods for chroma key detection. Both of them use a common helper function that you will implement:

```
void replace(   bool mask[][SIZE],
                unsigned char inImage[][SIZE][RGB],
                unsigned char bgImage[][SIZE][RGB],
                unsigned char outImage[][SIZE][RGB]);
```

It does the replacement operation: using the mask `mask`, it takes the foreground of `inImage` and replaces its background with `bgImage`, putting the result in `outImage`.

You will write two functions for mask creation that use slightly different approaches. The first one,

```
void method1(   unsigned char inImage[][SIZE][RGB],
                bool mask[][SIZE], double threshold);
```

must take `inImage`, determine the background color by averaging sample points (assume a reasonable number of top rows, left columns, and right columns will be background), classify all points as either foreground or background, and write the resulting classification to `mask`. The classification of pixels must be according to their whether their RGB distance from the background color is greater or less than `threshold`, which corresponds to a command-line input the program takes.

Note that `method1()` just creates the mask, so `main()` will have to use `replace()` in conjunction with the `method1()` function.

The second function has a similar signature,

```
void method2(unsigned char inImage[][SIZE][RGB], bool mask[][SIZE]);
```

except that it must determine both the background color and a scheme for classification automatically (without the threshold input from the user). You can use any approach you like – just make sure to document it!

When you run your program it will run both methods and so produce two new `.bmp` files. Specifically, the command-line arguments will be

```
./chromakey input.bmp bgnd.bmp THRESH output1.bmp output2.bmp
```

with the arguments meaning the input file, the background file, the numerical threshold (e.g. 34.5 or 216), the first output file, and the second output file.

6 Procedure

1. Download the skeleton project:

```
$ cd cs103
$ mkdir chromakey
$ cd chromakey
$ wget http://bits.usc.edu/files/cs103/chromakey/cs103_chromakey.tar
$ tar xvf cs103_chromakey.tar
```

   This will download the files:

```
chromakey.cpp  [Source code skeleton for you to complete]
bmplib.h       [header file for you to include in chromakey.cpp]
bmplib.cpp     [Source code containing the .bmp read/write functions]
Makefile       [Script for you to compile your program]
colbert_gs.bmp  bird_gs.bmp    astro.bmp      faded.bmp
               [Green-screened images for you to test with]
campus1.bmp   village.bmp  [Background images]
readme.txt     [readme template for pre-lab and review, to submit]
```

2. Edit the `chromakey.cpp` file taking note of the comments embedded in it. You will implement two different methods. In addition to prototypes for the methods already mentioned, in main() we provide some starter code. When calling the read and write BMP file functions, note that we check the return value to ensure successful operation. If one of these functions ever returns a non-zero value, exit the program immediately using "`return 1;`".

3. Look up the `cmath` library functions to take the power of a number and the square root. This is useful for computing RGB distance.

4. Review how to pass an array to a function. You will need this when you write your calls to method1(), method2(), and replace().

5. For method 1 you must implement the distance method described earlier in the lab. **First determine which set of pixels (location of pixels) you will "assume" are background in any green-screen image.** You should write code to iterate over these and find the "chromakey" (average R, G, B) values. Then iterate over the entire image computing the distance between each pixel and the chromakey value, and setting the mask array appropriately. Use the distance threshold that the user entered at the command line. You will need to determine a good threshold value by repeatedly re-running your program and examining the resulting `output1.bmp` (or `colbert1.bmp / bird1.bmp`) file.

6. At this point you can complete the replace() function and main() function, test whether your method1 worked, and see the effects of different distance thresholds.

7. For method 2 you will need to devise another method that can perform the chromakey operation without using any user-defined input (i.e. the threshold) from the user (i.e. automatic threshold / chromakey determination). You can either use method 1 but with your new automatic threshold determination or devise a completely new approach (but it should not require user input). You are NOT allowed to hard code a constant that was determined by examining pixels on a specific image. You may use a constant (scale factor or fudge factor) if it would likely work well for any image that might be input. Maximum points will be given to methods that perform well (visually) and that do not require user input (additional command line arguments) or trial and error runs like the previous method.

8. Be sure you ALWAYS compile your program by using 'make' since it will need to compile together both bmplib.cpp as well as chromakey.cpp.

9. Test your program with the following command line arguments:

```
$ chromakey colbert_gs.bmp campus1.bmp threshold_num colbert1.bmp colbert2.bmp
```

   Other tests can be performed by using the 'bird_gs.bmp' file and 'astro.bmp'. **Note:** 'astro.bmp' is much **HARDER** due to irregularity in the background. You don't have to handle it perfectly to get a full score.

   To view the output BMP files you can:

```
$ eog colbert1.bmp &
```

7 Troubleshooting

Before you post a question or seek help make sure you:

1. Can view the images you downloaded. If you have problems with gqview make sure you are in the right directory where your images are located.

2. Try printing out some pixel values to ensure you are reading in the input images correctly. Just be warned if you want to print out the numeric values of pixels, you'll need to cast to an integer first, as in:

```
cout << (int)inputImage[0][0][0] << endl;
```

   This is because otherwise, the image arrays are unsigned chars which will print out as actual ASCII characters.

3. Try using cout to print out the chromakey R,G,B value you find to make sure it is sensible.

4. Try using a debugger to run through and examine your values. When you want to debug a program that needs command line arguments (like ours which requires you to pass it the name of the input files, the threshold, etc.) when you start the program, as in:

```
$ ./chromakey colbert_gs.bmp campus1.bmp 40.5 output1.bmp output2.bmp
```

You start gdb normally with just the program name:

```
$ gdb chromakey
```

Then when gdb starts up, after `run`, you also type in the command line argument:

```
run colbert_gs.bmp campus1.bmp 40.5 output1.bmp output2.bmp
```

8 Review

In your "readme.txt" file answer the following questions.

1. Discuss and justify which pixels you chose to compute the average background value for method 1.

2. For method 1, go back and do some experiments to find the range of distance thresholds for which the program seems to operate the best for both Colbert and the Bird image. You may use different thresholds for the two different images. (i.e. threshold values that are too low will cause certain portions of the chroma key to not be filtered; values that are too high will cause parts of the foreground image to be filtered.)

3. For method 2, explain what you implemented and why. How did you go about implementing it (i.e. walk us through a high-level description of what you did in your code)?

   What other alternatives did you consider either before or after your implemented your selected method? What are the relative pros and cons of these methods? [This question is open-ended and we would like to see some discussion from you and a well-thought out response.]

9 Submission

Submit your 'chromakey.cpp' and 'readme.txt' files to the class website.