CPSC 217 Assignment #4: Image Decompression

Due: Friday, April 11, 2025, at 11:45pm
Weight: 7%
Sample Solution Length: Approximately 165 lines (including reasonable comments and the A+ portion of the assignment)

Individual Work:

All assignments in this course are to be completed individually. Use of large language models, such as ChatGPT, and/or other generative AI systems is prohibited. Students are advised to read the guidelines for avoiding plagiarism located on the course website. Students are also advised that electronic tools may be used to detect plagiarism.

Late Penalty:

Late assignments will not be accepted.

Submission Instructions:

Submit your .py file electronically to the Assignment 4 drop box in D2L. You don't need to submit the image files or SimpleGraphics.py – we already have them.

Description

Widely used image storage formats like PNG and JPEG use a variety of techniques to store image data compactly. While the small file sizes that these formats provide are desirable, this benefit comes at the cost of significant complexity in both the encoders and decoders. This makes them difficult to implement correctly and, at least in some cases, computationally expensive to run.

Dominic Szablewski created the Quite OK Image format which he first described in a blog post on November 24, 2021[1]. The format that he created uses some relatively straightforward techniques to quickly compress images. While the resulting files are not quite as small as well-compressed PNG files, the lack of complexity and better performance outweighs the file size penalty in some contexts. The Quite OK Image format also has the benefit that it is not encumbered by any patents, and the code for loading and saving images is distributed under a permissive license that allows it to be used in both personal and commercial projects without payment. A second blog post on December 20, 2021, described the finalized format and required only a single page to do so[2].

The Quite OK Image format is a binary file format. While binary files can be read and written with Python programs in much the same way that text files can, there are some additional details that need to be considered that we won't discuss in this course. As such, I have created a text file format for storing image data that is a variant of the Quite OK Image format for use in this assignment. I will refer to it as the OK Text Image (OKTI) format. While text file image formats are rare, they do exist. For example, the portable pixel map (PPM), portable gray map (PGB), and portable bit map (PBM) formats

[1] https://phoboslab.org/log/2021/11/qoi-fast-lossless-image-compression
[2] https://phoboslab.org/log/2021/12/qoi-specification

have both binary and text file variants which store full color, gray scale, and pure black and white images respectively.

In this assignment, you will write a function to decode images in the OKTI format. Then you will use that function to write a program that loads and displays an OKTI image. A detailed description of the image format can be found in the next section. Additional requirements for your program are described in later sections.

OKTI File Format

The OKTI format begins with a line containing nothing but the string "okti". This allows one to quickly determine whether or not a file contains OKTI data. The second line in the file contains two positive integers separated by a space. These positive integers are the width and height of the image, respectively.

The remainder of the file describes all of the pixels in the image. Pixels in the OKTI format are encoded in order from the upper left corner of the image to the lower right corner of the image, moving across each row in the image (from left to right) with the rows are encoded from top to bottom. Each pixel can be represented in one of 4 ways:

- As a full set of RGB values
- As the difference in RBG values from the immediately previous pixel
- As a run of several copies of the immediately previous pixel
- As a copy of a previously encountered pixel (not just the immediate predecessor)

Full RGB values

Pixels encoded as a full set of RGB values use 7 characters. The first character is always a lowercase ‘p’. This character is immediately followed by 6 hexadecimal digits. The first two encode the amount of red, the middle two encode the amount of green, and the final two encode the amount of blue. All three color components can range from 0 to (and including) 255 (which is denoted by ff). For example, a line in the file consisting of pff0000 represents one bright red pixel while p808080 represents a pixel that is middle gray colored.

Difference in RGB Values

When the current pixel is similar in color to the immediately previous pixel it is more efficient to encode it as a difference from the previous pixel rather than as a full set of RGB values. A pixel encoded in this manner begins with a lowercase ‘d’, followed by 3 hexadecimal digits. The first digit is the difference in the red component, the second digit is the difference in the green component, and the final digit is the difference in the blue component. Each of these values is stored with a bias of 8 meaning that the digit is 8 larger than the actual difference. For example, d08f is used to indicate that the current pixel’s red value is 8 smaller than the previous pixel’s (because 0 - 8 = -8), its green value is the same as the previous pixel’s (because 8 - 8 = 0), and the blue value is 7 larger than the previous pixel’s (because f - 8 = 7). Similarly, da27 is used to indicate that the current pixel’s red value is 2 larger than the previous pixel’s (because a - 8 = 2), the current pixel’s green value is 6 smaller than the previous pixel’s (because 2 - 8 = -6), and the blue value is 1 smaller than the previous pixel’s (because 7 - 8 = -1).

The previous pixel should be initialized to black (r: 0, g: 0, b: 0) before loading begins. This initial value allows the first pixel in the image to be a difference from a previous pixel.

A Run of Several Pixels

There are two variations of pixel runs, both of which encode copies of the immediately preceding pixel. The first variation consists of a lowercase ‘r’, followed by a single hexadecimal digit that represents the number of additional copies of the previous pixel that should be included in the image. This allows up to 15 copies of the previous pixel to be represented. For example, r5 is used to indicate that 5 additional copies of the previous pixel should be included in the image while re indicates 14 additional copies. The second variation consists of an uppercase ‘R’, followed by two hexadecimal digits that represent the number of additional copies of the previous pixel. This allows up to 255 copies of the previous pixel to be represented. For example, R20 represents 32 copies of the previous pixel while Rfe represents 254 copies of such. Note that it is possible that a run will wrap around onto the next row of the image, and in extreme cases, a run could span several rows.

The previous pixel should be initialized to black (r: 0, g: 0, b: 0) before loading begins. This initial value allows the first pixel in the image to a run of previous pixels.

Copy of a Previous Pixel

Each time a pixel is added to the image its color should be inserted at the front of a list of previously encountered colors **only if it is not already present in that list**. (As a consequence, the list will never contain any duplicate values). This list will store a maximum of 256 previous colors. If adding an additional color at the front of the list will violate this restriction, then the last color in the list should be discarded as the new value is inserted so that its length remains 256.

Like runs of pixels, there are two variations for copying a previous pixel. The first variation consists of a lowercase ‘i', followed by a single hexadecimal digit which is the index into the list of previous colors. For example, i0 is used to indicate that the current pixel should have the color that is stored at index 0 in the list, while ib indicates the current pixel’s color can be found at index 11 in the list. The second variation consists of an uppercase ‘I’, followed by two hexadecimal digits that are the index into the list of previous colors. For example, I11 is used to indicate that the current pixel should have the color that is stored at index 17 in the list, while Ia2 indicates that the current pixel’s color is at index 162 in the list.

The list of previous colors should initially contain one entry representing black (r: 0, g: 0, b: 0). This initial value allows the first pixel in the image to be a copy of a previous pixel.

Decoder Requirements

You must create a function that loads an image stored in the OK Text Image format (as described in the previous section) from a file. This function will take one argument, which is the name of the file to load. Your function will return one result, which is a SimpleGraphics image object containing all of the pixels described in the file. Your function must handle invalid files by displaying a meaningful error message and quitting (by calling the quit function). The error message that is displayed must indicate which of the following errors was encountered:

- The requested file does not exist.
- The requested file is not an OKTI file because its first line is not "okti".
- The dimensions for the image are invalid because either the width or the height (or both) are less than or equal to 0.
- A pixel type is present which is something other than ‘p’, ‘d’, ‘r’, ‘R’, ‘i’ or ‘I’.

When the requested file does not exist, the error message should include the name of the file that could not be opened. Similarly, when the image does not begin with "okti" or has invalid dimensions, the incorrect value should be displayed. When an invalid pixel type is encountered, the error message that your program displays must include the line that contained the error and the line number within the file so that the error can be located and corrected.

Once your program has confirmed that the pixel type is valid, it can assume that the data for that pixel type is correct – you don't need to consider errors such as an invalid hexadecimal digit or fewer characters being provided than expected. You may also assume that the number of pixels stored in the file is equal to width * height.

Main Program

Create a main program that demonstrates your image decoder by reading the name of an image file from the user and displaying the image. If the user provides one command line argument when the program is run, then that image should be opened and displayed. If no command line arguments are provided, then your program should read the name of the file from the user using the input function. Otherwise, your program should report that too many command line arguments were provided and quit.

Additional Requirements:

- Include your name, student ID number, and a brief description of your program at the top of your .py file.
- All lines of code that you write must be inside functions (except for the function definitions themselves, any constant definitions or import statements, and the call to main). Make appropriate use of a main function.
- Close every file that you open.
- You may create additional functions beyond your decode function and main if you find it helpful to do so.
- Do **not** define one function inside of another function.
- Include appropriate comments on each of your functions. All of your functions (except for main) should begin with a comment that briefly describes the purpose of the function, along with every parameter and every return value.
- Your program must not use global variables (except for constant values that are never changed, and you may not need any constants in this program).
- Your program must use good programming style. This includes things like appropriate variable names, good comments, minimizing the use of magic numbers, etc.
- When your program displays an error message and quits it should close the SimpleGraphics window. This can be accomplished by calling the close function in the SimpleGraphics module before calling quit.
- Break and continue are generally considered bad form. As a result, you are NOT allowed to use them when creating your solution to this assignment. In general, their use can be avoided by using a combination of if statements and writing better conditions on your while loops.

Working with Hexadecimal Values

The OKTI format makes heavy use of hexadecimal values. While you could create your own function for converting from a string containing a hexadecimal digit sequence to an integer, doing so is unnecessary because Python already includes this functionality. Python's int function is used to convert strings to integers. By default, the int function assumes that the string contains a sequence of characters representing a base 10 integer, but a second argument can be passed to int which specifies the numeric base of the digit sequence contained in the string. For example, if h is a variable that contains the string "a8", then calling int(h) will fail with a value error because "a8" is not a valid base 10 integer, but calling int(h, 16) will return 168 because "a8" is a valid hexadecimal digit sequence.

Other Reminders

Specific characters can be extracted from a string using indexing (for a single character) or slicing (for multiple characters). A slice of a string is formed by including two integers separated by a colon inside the square brackets that following the string. The first integer is the index of the first character to include in the slice. The second integer is the index of the first character that will **not** be included in the slice.

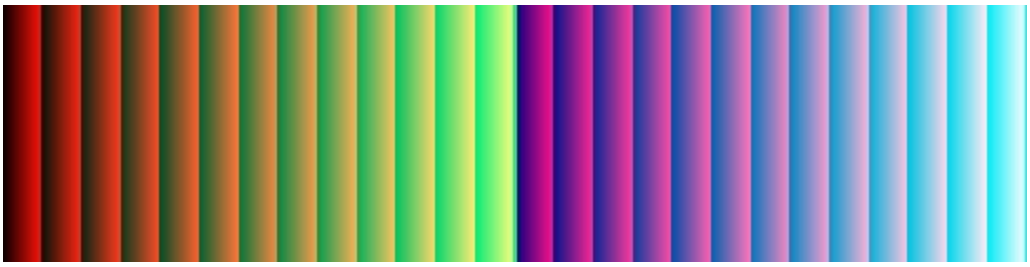
Hint:

Develop your function incrementally. Start by implementing support for only full RGB values. Once that works, add support for pixels that are represented by a difference from the previous pixel. Then add support for runs of identical pixels. Finally, add support for the list of previous pixels. Test images have been provided that only use full RGB values and full RGB values plus one other pixel type to help you focus on one type of pixel at a time. The sample images below use only a limited selection of pixel types so that you can focus on each pixel type individually.

small.okti

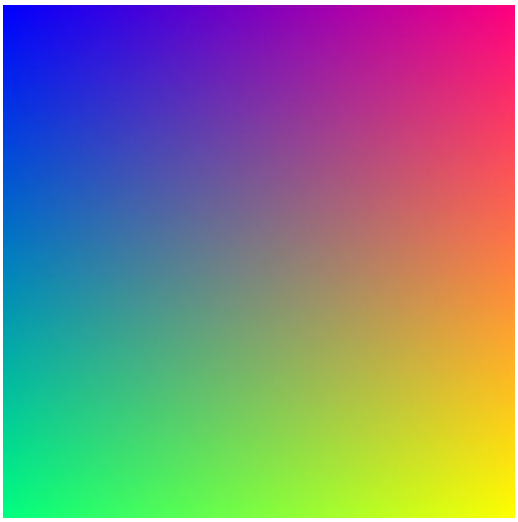
This 2x2 image consists of only 4 red pixels, all of which are represented as the 'p' pixel type. This is a good image to use if you are tracing values by hand, or if you have added print statements to your program for debugging which are generating an overwhelming amount of output for pixels.okti.

pixels.okti



This image is constructed only using pixels where a full set of RGB values are provided. It does not include any pixels represented as differences, runs, or indices into the list of previously encountered colors.

differences.okti



Two types of pixels are used in this image: full sets of RGB values and differences from the immediately previous pixel. It does not include any runs or indices into the list of previously encountered colors.

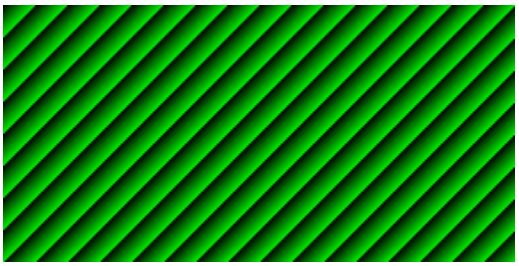
shortruns.okti

Only pixels represented as a full set of RGB values and runs of the immediately previous pixel are used in this image. All of the runs are less than 16 pixels in length which allows their length to be represented using a single hexadecimal digit.

longruns.okti

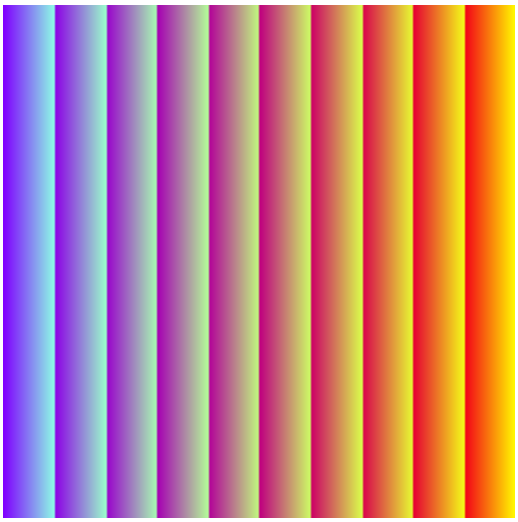
Like the previous image, this one only uses pixels where a full set of RGB values are provided and runs of the immediately previous pixel, but instead of using runs of 1 to 15 pixels it uses only runs that are at least 16 pixels in length (and as such, the length of the run is specified using two hexadecimal digits).

smallindices.okti



All of the pixels in this image are represented as either a full set of RGB values or a copy of a pixel that has been encountered previously. When a pixel is represented as a copy of a previous pixel, it is always a pixel that is among the 16 most recently encountered pixels which allows its position in the list of previous pixels to be represented by a single hexadecimal digit.

indices.okti



Most of the pixels in this image are represented as indices into the list of previously encountered in pixels. A few of these pixels use a single digit for the index into the list of pixels while most use a two-digit index. This image does not include any pixels represented as a run or a difference from the immediately previous pixel.

Additional images are available on the course website which use all of the pixel types. Ensure that you test your program with some (or all) of those images in addition to the images shown in this document.

Grading

This assignment will be graded on a combination of functionality and style. A base grade will be determined from the general level of functionality of the program (Does the decoder handle each of the required pixel types? Does the main program read the file name from the user successfully? Is the necessary error checking performed? Etc.). The base grade will be recorded as a mark out of 12.

Style will be graded on a subtractive scale from 0 to -3. For example, an assignment which receives a base grade of 12 (A), but has several stylistic problems (such as magic numbers, missing comments, etc.) resulting in a -2 adjustment will receive an overall grade of 10 (B+). Fractional marks will be rounded to the closest integer.

Total Score (Out of 12)	Letter Grade
12	A
11	A-
10	B+

9	B
8	B-
7	C+
6	C
5	C-
4	D+
3	D
0-2	F

For an A+:

Like the Quite OK Image format developed by Dominic Szablewski, my OK Text Image format is designed to provide some compression with relatively little complexity, rather than aggressively minimizing the size of the compressed data. One aspect of that minimal complexity for the OKTI format includes having each pixel (or group of pixels in the case of a run) stored on its own line. While this makes the files particularly easy to read, it imposes a space overhead of as much as one end-of-line marker for each pixel. When these end-of-line markers are a single character they represent at least 12.5% of the image data stored in the file. In the worst case (two-character end-of-line markers and a file that is stored almost entirely as 'i' and 'r' pixels) they represent almost 50% of the data stored in the file. However, it isn't actually necessary to store each pixel (or group of pixels in the case of a run) on its own line because all of the encodings used to represent the pixels are fixed length, and as such, the end-of-line marker isn't needed to identify the end of one pixel and the beginning of the next.

For an A+, update your loading function so that it works correctly on files with or without newlines separating each pixel (or group of pixels in the case of a run), including files that have a newline character between some pixels but not others. Several files that omit (some of) the newlines are available on the course website.

Including only the end-of-line characters necessary to keep the lines in the file to less than 80 characters reduces the space overhead for the newline characters to less than 3% of the image data in the worst case, while still producing files that can be opened in most text editors without any of the lines wrapping.