IMAGE PROCESSING LIBRARY

Temi Oladipo
toladipo@smu.edu

IMAGE PROCESSING LIBRARY v0.0.1 Documentation

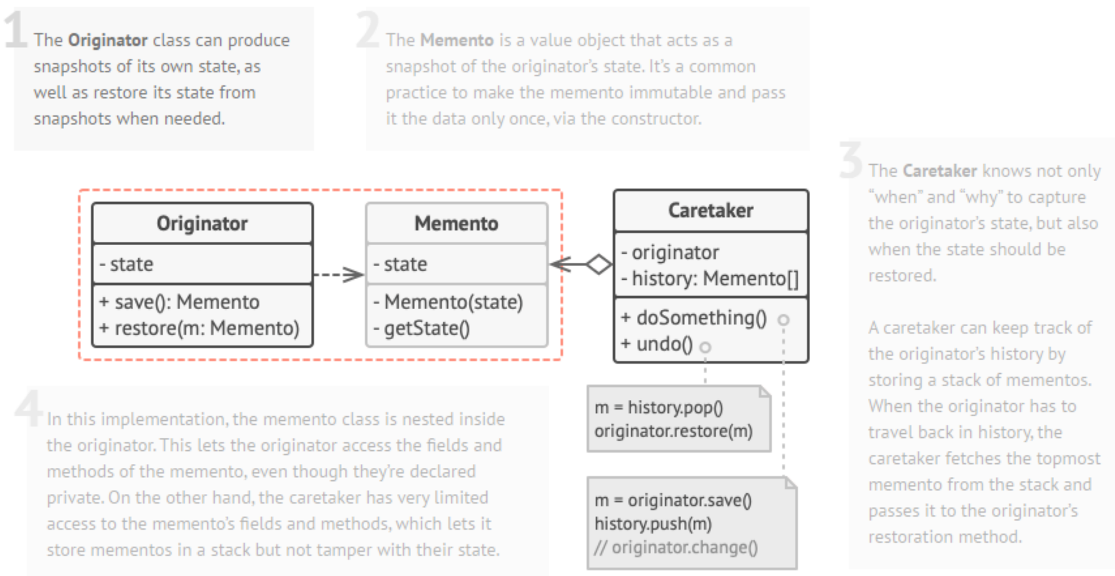
Version 0.0.1 - Stable:

Image processing JavaScript library (IPL) is a library that takes one or more comma separated image base64 representation as input, performs predefined image operations or processes on the images and return the updated images as comma separated image base64 representation. The library also can undo operations and restore the images to their original states. IPL uses ***stb_image libraries*** under the hood to extract image metadata such as height, width, channels, and pixels. IPL uses the metadata for image processing. This reduces the complexity of interpreting various image formats such as png, jpg, bmp, and so on. IPL also uses two base64 libraries for encoding and decoding images to and from base64 string respectively.

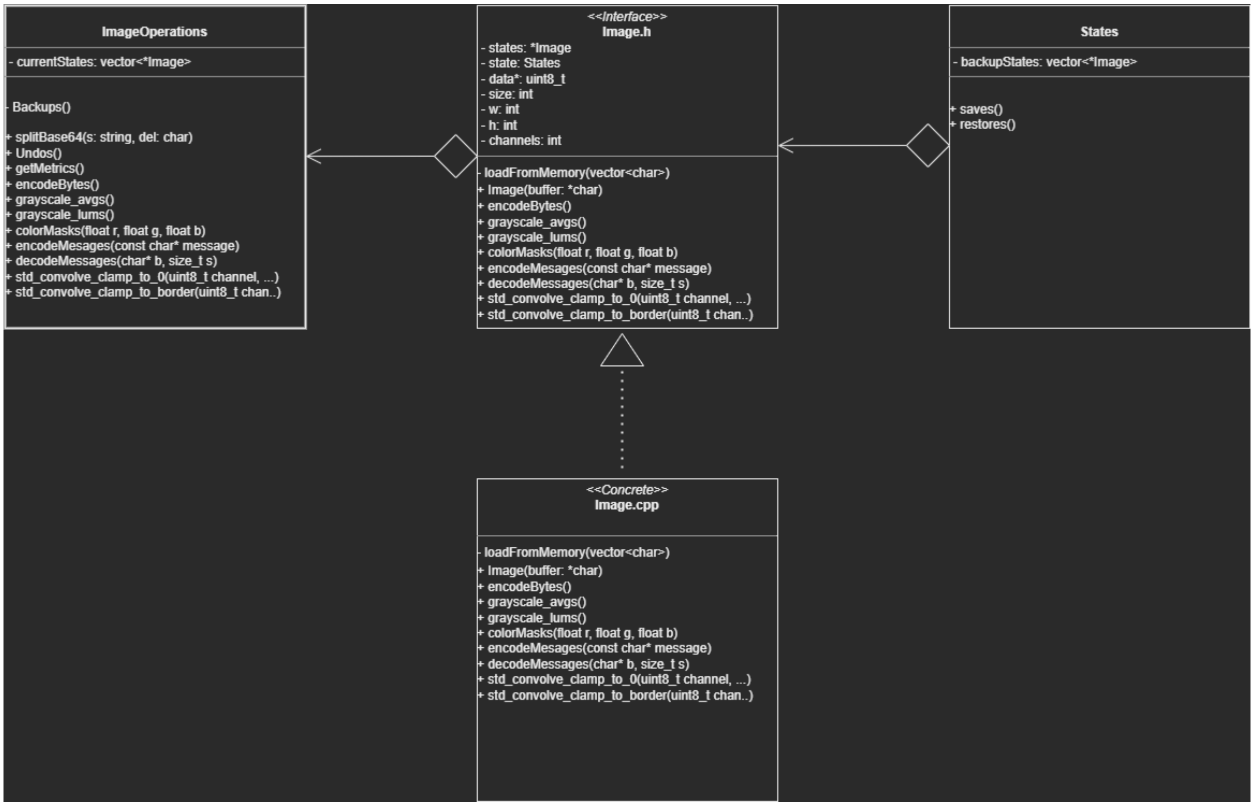
Design:

The library was designed using Momento Behavioral Design pattern. According to geeksforgeeks, "Memento pattern is a behavioral design pattern. Memento pattern is used to restore state of an object to a previous state. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later" (Geeks, 2021). Before any operation is performed on the image(s), the library creates a snapshot or states of the original images which can be restored at a later time. Consequently, images can be restored to any state within operational chain. For example, if operations A, B, C, and D was performed on images x and y, images x and y can be restored to states in the order C -> B -> A. Library Design is modeled after the design template below. However, there was a need to do a bit of consolidation because the stb_image libraries have a bunch of non-static methods and functions. These methods were cause issues during build time because they were being duplicated in every file that imports them. I am sure that the issues could have been resolved but for lack of time of C++ expertise.

Design Template:



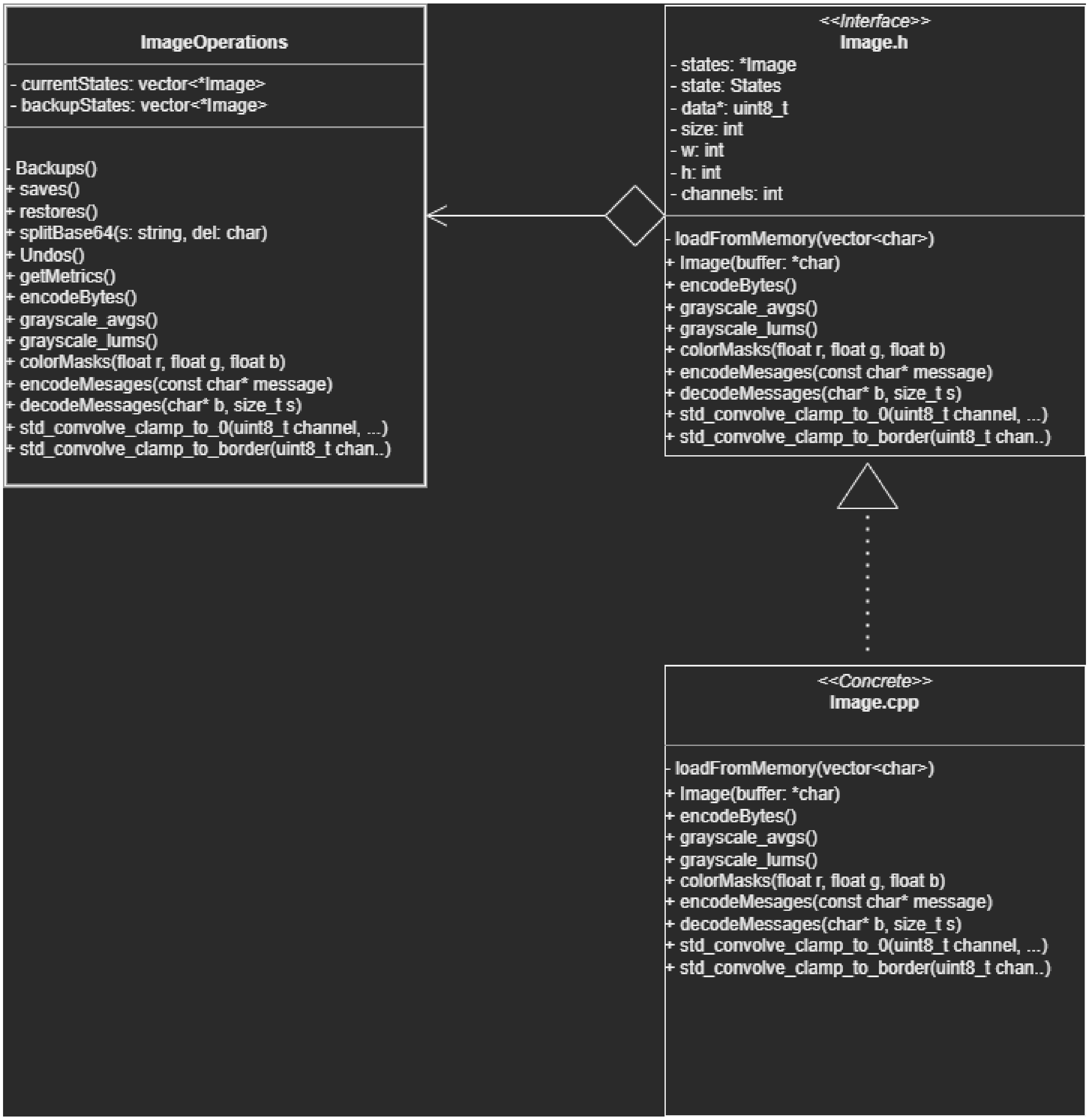
Initial Design:



Initial design has **Momento** as ***Image.h***, **Caretaker** as **States**, and **Originator** as ***ImageOperations***.

However, due to occurrence of multiple stb_image methods during compilation, the designed was altered to rectify the problem.

Altered or Consolidated Design



The altered design consolidates States with Image Operations. It delegates backups and restores responsibilities solely to the Originator, in this case ImageOperations.

Metrics class was later introduced to collects time metrics for every operation. The introduction did not necessarily change the original design.

API

Constructor

- Creates a single instance of Image Operation Class. Example:
- Example: let images = new Module.ImageOperations();

createImages(base64encodedString)

- function takes in base64encoded string (commas separated if multiple strings) as input parameter and creates instance(s) of Image class object(s).
- The base64encoded string must be valid else invalid base64 exception is thrown.
- Example: images.createImages(base64String);

grayscale_avgs()

- Converts colored image(s) to grayscale image(s).
- These images must have already been created by createsImages(base64encodingString) method.
- It adds all the red, blue, green channels in the image, divides the result by 3, and then sets the r, g, b values back to the averaged result.
- Function does not work on grayscaled image(s).
- Example: images.grayscale_avgs();

grayscale_lums()

- Converts colored image(s) to grayscale image(s).

- These images must have already been created by createsImages(base64encodingString) method.
- It uses weighted average of rgb channels to convert image(s) to grayscale in order to preserve the human perceived luminance of the original images.
- Function does not work on grayscaled image(s).
- Example: images.grayscale_lums();

colorMasks(float r, float g, float b)

- Masks colored image(s) red, blue, green channel(s) with parameters r, b, g respectively.
- These images must have already been created by createsImages(base64encodingString) method.
- Parameters r, b, g must be floating numbers between 0 and 1 for proper color masking. All deviating parameters will be ignored and have no effect.
- Image must have red, blue, green channels for the function to work.
- Example: images.colorMasks(0, 0, 1); will turn off red and green channels of the image leaving on only the blue channel. The resulting image will be blueish.

encodeMessages(string message)

- Encodes message in the image for later decoding.
- These images must have already been created by createsImages(base64encodingString) method.
- Length of message must not exceed to size of any of the image(s).
- Example:
    - Let message = “blab la bla”
    - images.encodeMessages(message);

string decodeMessagesLib();

- Returns the encoded message in the image or the encode message in the first image if multiple images are present since multiple images will consist of the same encoded message.
- These images must have already been created by createsImages(base64encodingString) method and must have already been encoded with encodeMessages(string message) function.

std_convolve_clamp_to_0(int channel, int ker_w, int ker_h, int cr, int cc)

- Uses convolution to apply Gaussian blur effect to an image.
- Please not that the actual gaussian kernel is hardcoded in the library. Future versions will allow custom kernels to be pass in as parameters.
- The function can only be performed on a single channel at a time.
- Channel affects either r, g, or b, ker_w is the kernel width, ker_h is the kernel height, cr and cc represents the coordinates of the center of the kernel. For more information, checkout out https://graphics.stanford.edu/courses/cs148-10-summer/docs/04_imgproc.pdf
- These images must have already been created by createsImages(base64encodingString) method.
- Example: images.std_convolve_clamp_to_0(1, 3, 3, 1, 1)

std_convolve_clamp_to_border(int channel, int ker_w, int ker_h, int cr, int cc)

- Same as std_convolve_clamp_to_0 with one exception.
- The exception is that corner cases at the image border are handled differently.
- Example images.std_convolve_clamp_to_border(0, 3, 3, 1, 1);

flipX()

- Flips image on the X-axis

- These images must have already been created by createsImages(base64encodingString) method.
- Example: images.flipX()

flipY()

- Flips image on the Y-axis
- These images must have already been created by createsImages(base64encodingString) method.
- Example: images.flipY()

string encodeBytes()

- Returns current state of all images in a comma separated base64 encoded string.
- These images must have already been created by createsImages(base64encodingString)
- Example: const base64_str = images.encodeBytes();
- Please note that custom method is needed to parse individual image string using comma and delimiter.

string getMetrics():

- Retries all time metrics for a single run as csv formatted string.
- The string can be saved to a file and processed later by any application capable of processing csv files.

NOTE: There are some functions defined in the C++ code that could not be exported out in the library due to emcripten file API limitations. For example, the native C++ native code can write images to file in according to their formats or extensions, but the transpiled JS library cannot do that using the same interface.

PERFORMANCE TESTING

Criteria was based on the aggregation of execution times for all operations in a single run. Unfortunately, the library's base64 comma delimiter parsing takes a lot of time because the strings are humongous. Consequently, performance test assumes that all images have already been parsed and the ImageOperations class object instantiated. Performance testing carried out for the native C++ application and the transpiled JavaScript application. The two applications uses the result from the getMetrics() to create a csv file which will be analyzed later. To facilitate a more meaningful test comparison, the two applications are tested with the same images.

["../../Data/base64_images/tiger.txt", "../../Data/base64_images/tiger.txt"]

An array of two images

Before running the test, follow the following installation guide. Before running the test, follow the following installation guide. The Makefile ***in lab-2-multithreading-eltopus/Code/build*** has the following environment variables that must be modified accordingly.

```
IMAGE_OPS = /home/kali/LABS/lab-2-multithreading-
eltopus/Code/image_operations.cpp
IMAGE_H = /home/kali/LABS/lab-2-multithreading-eltopus/Code/image.h
IMAGE_CPP = /home/kali/LABS/lab-2-multithreading-eltopus/Code/image.cpp
IMAGE_ORIGINATOR = /home/kali/LABS/lab-2-multithreading-
eltopus/Code/originator.cpp
IMAGE_GLUE_WRAPPER = /home/kali/LABS/lab-2-multithreading-
eltopus/Code/image_glue_wrapper.cpp
OUTPUT_JS = /home/kali/LABS/lab-2-multithreading-eltopus/Code/ipl.js
OUTPUT_JS_MIN = /home/kali/LABS/lab-2-multithreading-eltopus/Code/ipl_min.js
GLUE_JS = /home/kali/LABS/lab-2-multithreading-eltopus/Code/glue.js
EMSDK_ROOT_FOLDER = /home/kali/emsdk/upstream/emscripten
REPO_CODE_DIR = /home/kali/LABS/lab-2-multithreading-eltopus/Code
WASM_OUTPUT = /home/kali/LABS/lab-2-multithreading-eltopus/Code/ipl.wasm
WASM_MIN_OUTPUT = /home/kali/LABS/lab-2-multithreading-eltopus/Code/ipl_min.wasm
IMAGE_MAIN = /home/kali/LABS/lab-2-multithreading-eltopus/Code/main.cpp
NODE = /home/kali/LABS/lab-2-multithreading-eltopus/Code/app.js
METRICS = /home/kali/LABS/lab-2-multithreading-eltopus/Code/metrics.cpp
```

Installation

- Download imp.js and imp.wasm.
- Import as need in NodeJS projects or Plain JavaScript projects.

Building from Source:

- IMP.js was built using Emscripten WebIdl interface.
- Building from source requires:
    - C++17 compiler
    - Emscripten emsdk
    - Cmake
- Clone repository.
- Change directory to /LABS/lab-2-multithreading-eltopus/Code/build
- Run make glue.
- Run make build.

Running C++ native code test execution:

- Change directory to /LABS/lab-2-multithreading-eltopus/Code/build
- Run make
- Execute ./main

Running JavaScript Code test execution:

- Change directory to /LABS/lab-2-multithreading-eltopus
- Run npm install
- Run node app.js

C++ Test Execution

```
int main() {
    runOperations(100);
}
```

runOperations function accepts integer which accounts for the number of times a test suite is ran. After test suite is completed, the getMetrics function is called and result written to csv file. File can be found at

Lab-2-multithreading-eltopus/Code/build/cPlusMetrics.csv

Test suite definition looks like this:

```
std::vector<std::string> files{ "../../Data/base64_images/tiger.txt",
"../../Data/base64_images/tiger.txt"};

ImageOperations* ops = new ImageOperations();
    ops->createImages(buffer);

    for (int i =0; i < n; i++){

        // Perform operations
        ops->grayscale_avgs();
        ops->Undos();
        ops->grayscale_lums();
        ops->Undos();
        ops->colorMasks(1, 0, 0);
        ops->Undos();
        ops->encodeMessages("Some
loooooooooooooooooooooooooooooooooooooooooooooooooooon
strinnnnnnnnnnnnnnnnnnnnnnnnnng");
        std::string decodedString = ops->decodeMessagesLib();
        ops->Undos();
        ops->std_convolve_clamp_to_0(0, 3, 3, 1, 1);
        ops->Undos();
        ops->std_convolve_clamp_to_border(2, 3, 3, 1, 1);
        ops->Undos();
        ops->flipX();
        ops->Undos();
        ops->flipY();
        ops->Undos();
        std::string encodedBytes = ops->encodeBytes();
}
```

```
char* metrics = ops->getMetrics();

writeMetrics(metrics);

delete ops;
```

JavaScript Test Execution

```
async function runOperations(n) {
  try{

    const files = ["../../Data/images/tiger.jpg", "../../Data/images/tiger.jpg"]
    let base64Combined = await combinesBase64Images(files);
    let images = new Module.ImageOperations();
    images.createImages(base64Combined);

    if (n < 1){
      n = 1;
    }

    for (i = 0; i < n; i++){
      images.grayscale_avgs();
      images.Undos();
      images.grayscale_lums();
      images.Undos();
      images.colorMasks(1, 0, 0);
      images.Undos();
      images.encodeMessages("Some
looooooooooooooooooooooooooooooooooooooooooooooooon
strinnnnnnnnnnnnnnnnnnnnnnnnnng");
      const decodedString = images.decodeMessagesLib();
      images.Undos();
      images.std_convolve_clamp_to_0(0, 3, 3, 1, 1);
      images.Undos();
      images.std_convolve_clamp_to_border(2, 3, 3, 1, 1);
      images.Undos();
      images.flipX();
      images.Undos();
      images.flipY();
      images.Undos();
      const base64Response2 = images.encodeBytes();
      //await splitsAndWriteBase64Images(base64Response2, outputfilesOriginal);

    }

    const metrics = images.getMetrics();
    // console.log(metrics);
    writeMetrics(metrics);
  }catch (error) {
```

```
    console.log(`An error occurred during processing: ${error}`);
  }

};
```

Run operations method takes the number of runs as parameter n and then run through the test suite n times. The result is collected from getMetrics() and then written to file. File can be found in:

lab-2-multithreading-eltopus/Code/build/jsMetrics.csv

Optimized JavaScript was built using -O2 flag :

```
emcc $(IMAGE_OPS) $(METRICS) $(IMAGE_CPP) $(IMAGE_GLUE_WRAPPER) --post-js
$(GLUE_JS) -O2 -o $(OUTPUT_JS) -s ALLOW_MEMORY_GROWTH=1 -s
EXPORTED_RUNTIME_METHODS=["ccall, cwrap"] -s EXPORTED_FUNCTIONS=["_free"]
```

TEST RESULT ANALYSIS

Applications	**Average Execution Time in seconds**	**Runs**
C++ Native	0.1309	500
JavaScript	0.0733	500
Optimized JavaScript	0.0590	500

500-run was chosen because that is the max the JavaScript library can handle before it throws an exception. Average execution time was calculated by adding all execution times and dividing by execution runs.

Test results shows that optimized JavaScript is the fastest, followed by unoptimized JavaScript, and lastly, C++ native.

FYI:

- The library expects a well-formed or valid base64 encoding of image(s).
- Template images are placed in **lab-2-multithreading-eltopus/Data/images**
- Images operated on by C++ code are placed in **lab-2-multithreading-eltopus/Data/cPlusOutput**
- Images operated on by JavaScript library are placed in **lab-2-multithreading-eltopus/Data/jsOutput**
- Template base64 images are placed in **lab-2-multithreading-eltopus/Data/base64_images**
- Metrics from C++ code, Javacript code, and Optimized JavaScript code are placed in **lab-2-multithreading-eltopus/Data/metrics** as *cPlusMetrics.csv*, *jsMetrics.csv*, and *jsMetrics_min.csv* respectively.
- The test application on the JavaScript side uses a library called "sharp" to load images into a buffer before converting them to base64 encoding. Loading images directly from NodeJS file system and then encoding them to base64 did not work for this test. The reason is currently unknown.
- Although sharp has other image processing capabilities, those capabilities were never used during the test.
- The library has a limit to the amount of base64 encoded images it can run. This limit is probably based on NODEJS execution memory limit. Not sure though.