FlashPhoto Iteration #2

Michael Adkins, Cuiqing Li, Logan Seeley

GitHub Repository: **repo-group-10amGroup06**

Flashphoto has been correctly and completely implemented according to the specifications provided.

Two special filters were created, one based on single pixel editing to create a sepia tone and another that required a new image-wide mask filter type that creates a vignette effect.

Our major design choices for this iteration follow. These choices focus on the classes supporting filter application and external image handling. Although new tools were required, our previous design choices made their implementation relatively trivial.

Since Filters were the largest new feature to be added and had significant repetitive aspects, they received the majority of the design attention. It was clear from the start that there were two major types of Filters: those that act on a single pixel independent of the surrounding pixels (PositionIndependentFilters) and those that used convolution and a kernel to pull information from surrounding pixels to determine the value of a center pixel (ConvolutionBasedFilters). These two classes were implemented as children of a Filter class that served as a wrapper with a virtual apply function that received a pointer to the current PixelBuffer and a pointer to an array of float arguments pulled from the GUI. The movement of these classes into a wrapper allows dynamic casting into an array of Filters which makes Filter application a type independent process as in our Tool application.

The PositionIndependentFilter apply function iterates over all the pixels in the buffer (Figure 1), applying a different inner function to each one. Rather than placing a large switch inside this loop, the inner functions are passed into the class constructor via pointer. The PositionIndependentFilter is then ignorant of what function is being applied to the pixels, it just provides the surrounding algorithm. Similarly, the ConvolutionBasedFilter apply function works the same for every filter, convolving over the buffer. The only unique factor is the two dimensional filter matrix used for convolution. The class is constructed with a pointer to a function (filterGen) that given arguments will return a filter matrix and a filter size (by reference).

These function pointers provide an elegant isolation of the unique part of each algorithm (Figure 2). However, the functions must be stored somewhere. To simplify passing, they were created as static functions contained in a relevant namespace e.g. KernelFilterAlgorithm and PixelFilterAlgorithm. The history of static functions in C++ is convoluted as they've been deprecated then reinstated and is beyond the scope of this report, but this placement in a namespace seemed to be the best practice considering our goals. The static declaration of these functions not only allows easy passing into the Filter classes, but allows access from an outside source. This allows different surrounding algorithms to use the function with ease, e.g. a PixelFilterAlgorithm could be applied to only some of the pixels instead of all the pixels in a 'PartialImageFilter'. The namespaces also allow containment of support classes that aren't relevant outside the scope of the algorithms; a KernelFilter class was created to simplify work that recurred in many of the kernel filter algorithms, greatly simplifying their processes while still separated from the other code.

To separate the construction of the Filters from FlashPhotoApp, a FilterHandler class was made with an array of Filter objects constructed via a switch on an enumeration and a single-line apply function that calls the proper class by index (Figure 3). This approach provides a singular location for adding a new filter (not counting the GUI elements) requiring only addition to the enumeration and a case in the loop defining an object. For our special filters, this process was put into practice. Adding a sepia filter required the addition of one function in the PixelFilterAlgorithm namespace and four minor changes in FilterHandler. Since that was trivial, a vignette filter was created that required a new Filter child ImageMaskBasedFilter which applies a mask to the entire image and a new namespace to contain functions for image mask generation. Despite the significant difference in algorithmic application of this filter, its inclusion was straightforward and simple.

The creation of a tool that uses a filter requires a Filter object for the filter its implementing so the tool is blind to the implementation of the tool while reducing redundancy i.e. the convolution application code does not have to be repeated. This technique does sacrifice some performance demonstrating the tradeoffs of abstract minimal redundancy versus repetitive fully optimized solutions. However, since the inner algorithms of the Filters are separated, an optimal solution would be easily possible without sacrificing all redundancy elimination.

Overall, Filters used inheritance to create type indifferent application from FlashPhotoApp and function pointers to create algorithm indifferent application within each Filter. These features reduce repetitive code, increase readability of the codebase, and allow easy future extensions.

We had many options for a second design choice including the use of a customized stack for undo/redo, image saving and loading, and a deeper focus on one of the many sub-choices made in the Filter system e.g. KernelFilter support class. None of these design choices were "major" or heavily debated in the group as the solution seemed relatively straightforward. However, image saving and loading was a harder than expected task and its implementation underwent design changes before the final version of FlashPhoto was completed making it the prime candidate for this choice.

An ImageHandler class was written to handle all of the operations associated with saving and loading PNG and JPEG images in FlashPhotoApp; this class serves to isolate each of these processes from the rest of FlashPhotoApp. A single ImageHandler is instantiated by `FlashPhotoApp::intializeImageLibs()` during initialization of FlashPhotoApp for use during the application's lifetime. Since image saving/loading is heavily dependent on the image formats and libraries used, it is desirable to move this code out of FlashPhotoApp. This is beneficial for four main reasons:

- It hides implementations and encapsulates complex code.
- It makes image loading and saving from within FlashPhotoApp practically effortless (Figure 4). Since ImageHandler provides straightforward functions `loadImage()` and `saveImage()`, very little work must be done in FlashPhoto itself (see image below). Moreover, if the load and save algorithms needed to be changed,, nothing would need to be changed within FlashPhotoApp
- It makes it relatively easy to add support for other image libraries if desired; since all filetype-specific operations occur within ImageHandler, this is the only class that needs to be edited to provide support for more libraries. Given the multitude of image formats available, this is a key point for future extensibility.
- It allows the same image loading code to be used for loading both the canvas and the stamp tool, rather than requiring separate processes for each.

An alternative approach to image saving and loading would be to place the image handling code directly in the `loadImageToCanvas()`, `loadImageToStamp()`, and `saveCanvasToFile()` functions within FlashPhotoApp. While this is the most direct solution to the problem, it has a couple drawbacks.

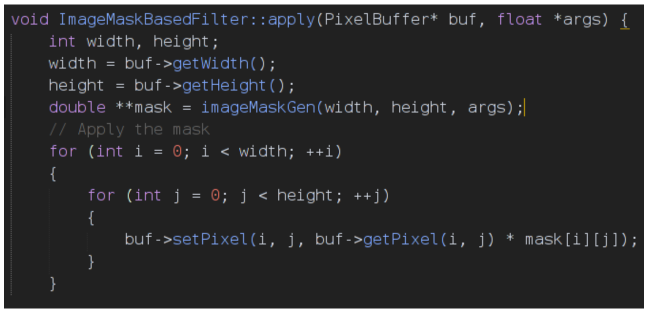
- FlashPhotoApp would quickly become unruly, making it more cumbersome to read and understand.
- There will be extensive creation of redundant code, since `loadImageToCanvas()` and `loadImageToStamp()`follow roughly the same process.

While at least the second drawback could be resolved by writing a helper function, such functions are beyond the scope of FlashPhotoApp which is dedicated primarily to the GUI and callbacks. The helper function would still detract from the readability of the main application.

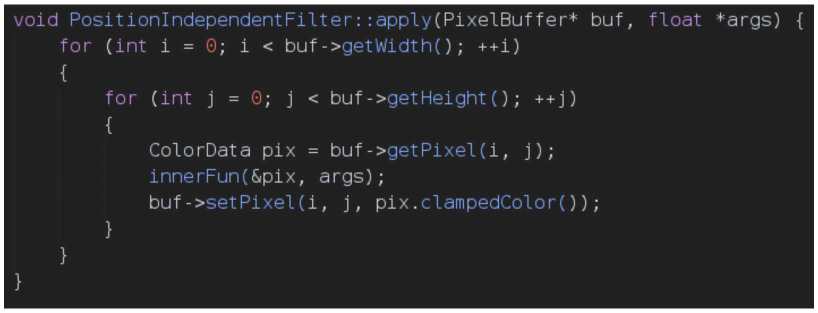
Another alternative to the implemented solution would be to further abstract the image loading and saving procedures to filetype specific classes. This solution has two primary advantages: it encapsulates the differences between the ways images are processed using different libraries and isolates the private variables used for each library. While this solution has merit, it is too much overhead for the current version of FlashPhoto since only two image libraries are currently in use.. If many more image libraries were to be implemented in the future, separating the library-specific code into individual classes might be more practical, but has the potential drawbacks of maintaining additional class files.

Supplemental Figures

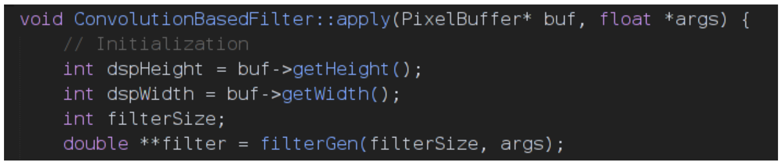
Figure 1: Snippets of Filter application demonstrating use of inner function pointers



```
void ImageMaskBasedFilter::apply(PixelBuffer* buf, float *args) {
    int width, height;
    width = buf->getWidth();
    height = buf->getHeight();
    double **mask = imageMaskGen(width, height, args);
    // Apply the mask
    for (int i = 0; i < width; ++i)
    {
        for (int j = 0; j < height; ++j)
        {
            buf->setPixel(i, j, buf->getPixel(i, j) * mask[i][j]);
        }
    }
```

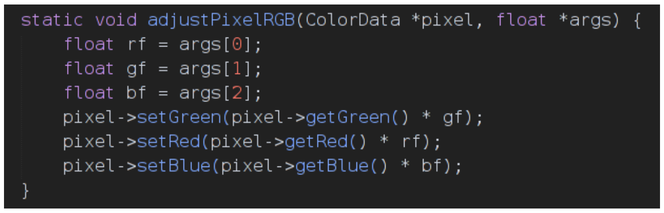


```
void PositionIndependentFilter::apply(PixelBuffer* buf, float *args) {
    for (int i = 0; i < buf->getWidth(); ++i)
    {
        for (int j = 0; j < buf->getHeight(); ++j)
        {
            ColorData pix = buf->getPixel(i, j);
            innerFun(&pix, args);
            buf->setPixel(i, j, pix.clampedColor());
        }
    }
}
```

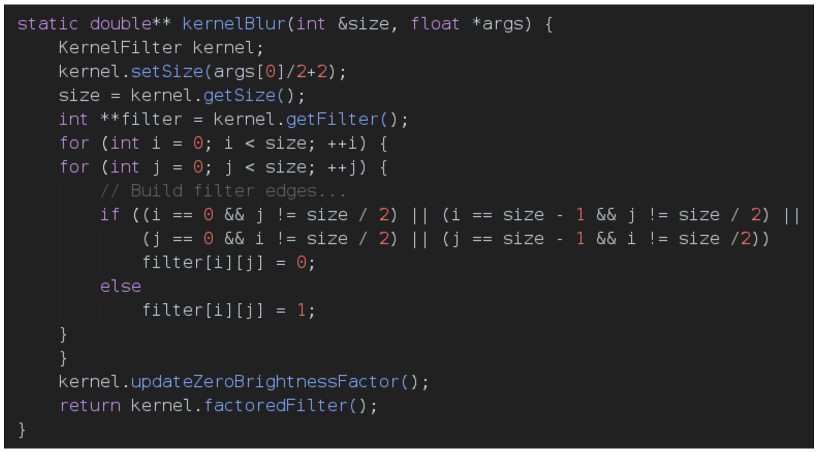


```
void ConvolutionBasedFilter::apply(PixelBuffer* buf, float *args) {
    // Initialization
    int dspHeight = buf->getHeight();
    int dspWidth = buf->getWidth();
    int filterSize;
    double **filter = filterGen(filterSize, args);
```

Figure 2: Sample static filter algorithm functions

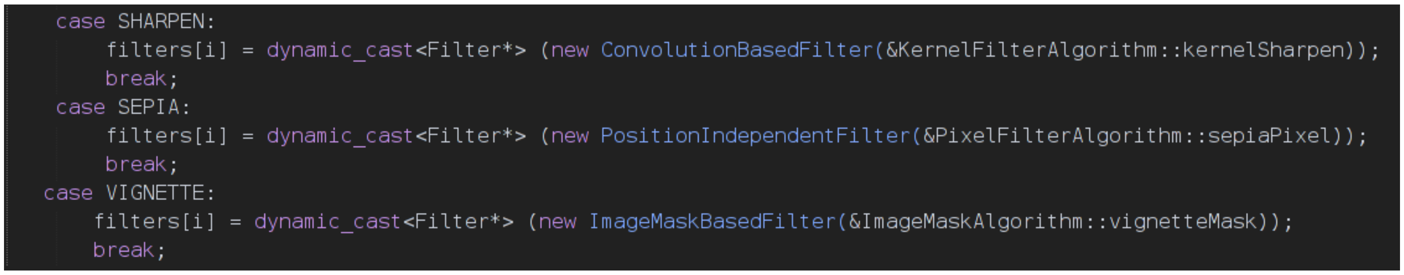


```
static void adjustPixelRGB(ColorData *pixel, float *args) {
    float rf = args[0];
    float gf = args[1];
    float bf = args[2];
    pixel->setGreen(pixel->getGreen() * gf);
    pixel->setRed(pixel->getRed() * rf);
    pixel->setBlue(pixel->getBlue() * bf);
}
```

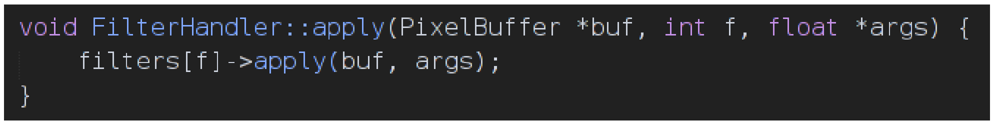


```
static double** kernelBlur(int &size, float *args) {
    KernelFilter kernel;
    kernel.setSize(args[0]/2+2);
    size = kernel.getSize();
    int **filter = kernel.getFilter();
    for (int i = 0; i < size; ++i) {
    for (int j = 0; j < size; ++j) {
        // Build filter edges...
        if ((i == 0 && j != size / 2) || (i == size - 1 && j != size / 2) ||
            (j == 0 && i != size / 2) || (j == size - 1 && i != size /2))
            filter[i][j] = 0;
        else
            filter[i][j] = 1;
    }
    }
    kernel.updateZeroBrightnessFactor();
    return kernel.factoredFilter();
}
```

Figure 3: Filter handler construction snippet and apply function

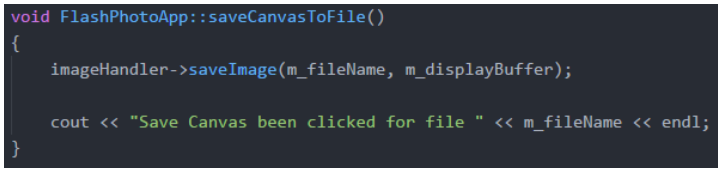


```
    case SHARPEN:
        filters[i] = dynamic_cast<Filter*> (new ConvolutionBasedFilter(&KernelFilterAlgorithm::kernelSharpen));
        break;
    case SEPIA:
        filters[i] = dynamic_cast<Filter*> (new PositionIndependentFilter(&PixelFilterAlgorithm::sepiaPixel));
        break;
   case VIGNETTE:
       filters[i] = dynamic_cast<Filter*> (new ImageMaskBasedFilter(&ImageMaskAlgorithm::vignetteMask));
       break;
```



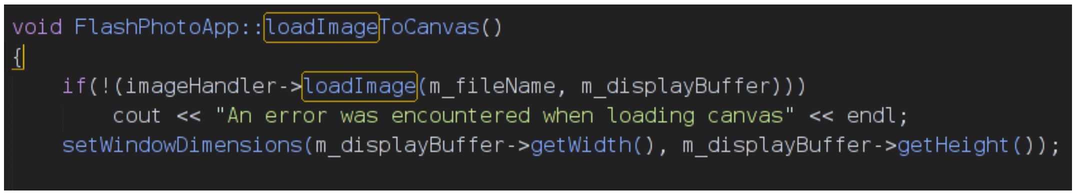
```
void FilterHandler::apply(PixelBuffer *buf, int f, float *args) {
    filters[f]->apply(buf, args);
}
```

Figure 4: Loading and saving images using ImageHandler



```
void FlashPhotoApp::saveCanvasToFile()
{
    imageHandler->saveImage(m_fileName, m_displayBuffer);

    cout << "Save Canvas been clicked for file " << m_fileName << endl;
}
```



```
void FlashPhotoApp::loadImageToCanvas()
{
    if(!(imageHandler->loadImage(m_fileName, m_displayBuffer)))
        cout << "An error was encountered when loading canvas" << endl;
    setWindowDimensions(m_displayBuffer->getWidth(), m_displayBuffer->getHeight());
```