Design Document

Design Documentation of API Convolutions

1 Goal and Predefined filters.

Provide developers a library that can apply different predetermined convolutional filters to input images. The predefined filters are listed in section 3.

2 Class Diagrams/Available API's

2.1 From the C++ Perspective

ConvoutionFilters

```
ConvoutionFilters(int rows, int cols, int channels, debug=0)
        void setInputImg(float *img, int size);
    float* applyFilterCPP(const char* filterName);
            float* getInputImgPtr();
        void setThreadCount(long num);
```

We exposed the above functionality to the C++ library for use. The user must know the size of the image they want to run convolutions on. Once the object is created the user must call `setInputImg()` to populate the class variable `inputImg,` this is done with a deep copy. Then `applyFilterCPP()` can be called to perform the convolution.

1. `setInputImg(float *img, int size)`

   This method will perform a deep copy from the float array img to the class float array inputImg.

2. `applyFilterCPP(const char* filterName)`

   Performs the filter that is passed in as a char*. The returned float* is a reference to the class variable inputImg and contains the image output.

3. `getInputImgPtr()`

   Returns a pointer to the data that contains either the input image, before applyFilterCPP is called or the pointer to the output image after applyFilterCPP is called. This is the same pointer that is returned by that function.

Extra Exposed C++ Functions

```
float* applyConv(float *kernel, int kernelSize, int channels)
        float* setInputImg(float* inputImg, int size)
              float* applySharpen(int channels)
                float* applyBlur(int channels)
             float* applyGaussBlur(int channels)
              float* edgeDetection(int channels)
               float* applyEmboss(int channels)
                 float* applyLoG(int channels)
```

1. `float* applyConv(float *kernel, int kernelSize, int channels)`

   creates a new float array that contains the output of applying kernel to the inputImg. It is the responsibility of the caller to free the returned pointer.

2. The rest of the functions listed make a call to `applyConv`, delete inputImg and replace it with the output of the `applyConv` removing the responsibility of deleting the output array.

2.2 Memory Concerns

`applyFilterCPP()` will return a float pointer that represents the resulting image. This pointer will be freed up when the object is deconstructed. If there is a need for the data to live beyond the object it will have to be copied to somewhere else. When `applySharpen()`, `applyBlur()`, `applyGaussBlur`, `edgeDetection()`, `applyEmboss()`, `applyLoG()`, the pointer will live as long as the object is around. However when using `applyConv()`, it is the caller responsibility to clean up the pointer returned. This function will not be exposed to the JavaScript Library.

2.3 From the JavaScript Library Perspective

```
ConvoutionFilters(int rows, int cols, int channels, debug=false)
                      long getInputImgPtr()
             long applyFilter(DOMString filterName)
                void setThreadCount(long count);
```

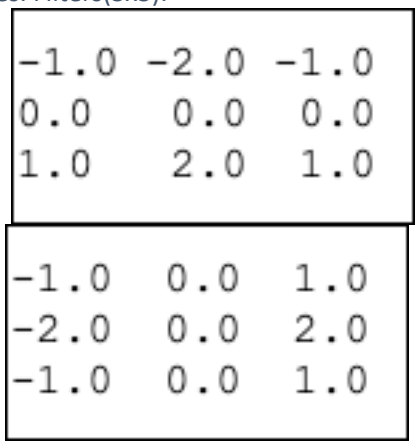
1. ConvolutionFilters(int rows, int cols, int channels, bool debug)
2. long getInputImgPtr()
3. long applyFilter(DOMString filterName)

The exposed JavaScript functions are limited with the intention of simplifying the process. To set everything complete the following steps:

1. Create an instance of `ConvoutionFilters()` with the specified size and channels.
2. Get the pointer to the input Img using getInputImgPtr()
3. Set the data in `inputImg` (using `Module.set(…)`) passing in a JavaScript Typed array of Floats

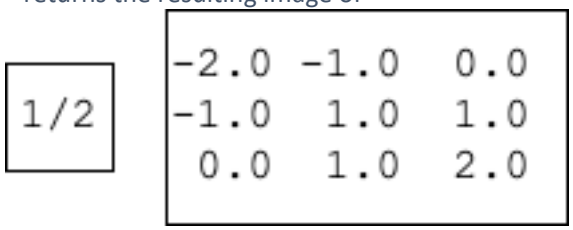
4. Call `applyFilter()` with the specified filter and maintain the pointer that is returned. It points to the image data we need to save.
5. Call setThreadCount() to set the number of threads you want to run. The default will be 8 unless this method is called before applyFilter()

3 Predefined Filters

- Edge Detection – returns a black and white image with high pixels representing the pixels that are edges. Filters(3x3):
  - 

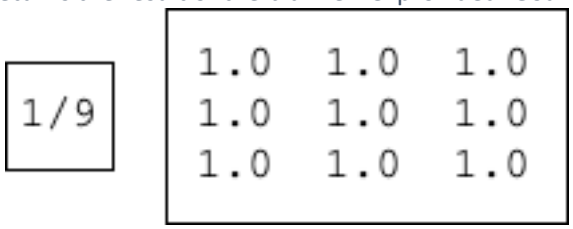
```
-1.0 -2.0 -1.0
0.0   0.0  0.0
1.0   2.0  1.0
```

```
-1.0  0.0  1.0
-2.0  0.0  2.0
-1.0  0.0  1.0
```

- Emboss – returns the resulting image of
  - 

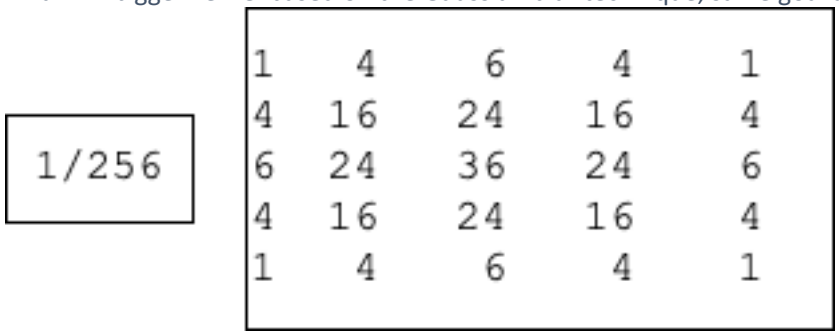
```
1/2
```

```
-2.0 -1.0  0.0
-1.0  1.0  1.0
 0.0  1.0  2.0
```

- Blur – Returns the result of the blur kernel provided. Goal is to blur edges of the image.
  - 

```
1/9
```

```
1.0  1.0  1.0
1.0  1.0  1.0
1.0  1.0  1.0
```

- Gaussian Blur – A bigger kernel based on the Gaussian blur technique, same goal as above.
  - 

```
1/256
```

```
1   4   6   4   1
4  16  24  16   4
6  24  36  24   6
4  16  24  16   4
1   4   6   4   1
```

- Laplacian of Gaussian – Referred to as LoG in the code. Runs a Laplacian filter then a Gaussian filter

```
 0.0 -1.0  0.0
-1.0 4.0 -1.0
 0.0 -1.0  0.0
```

```
-1.0 -1.0 -1.0
-1.0  8.0 -1.0
-1.0 -1.0 -1.0
```

- Sharpen – Applies a filter that will sharpen the edges of an image with the below filter:

```
 0.0 -0.5  0.0
-0.0  8.0 -0.5
 0.0 -0.0  0.0
```