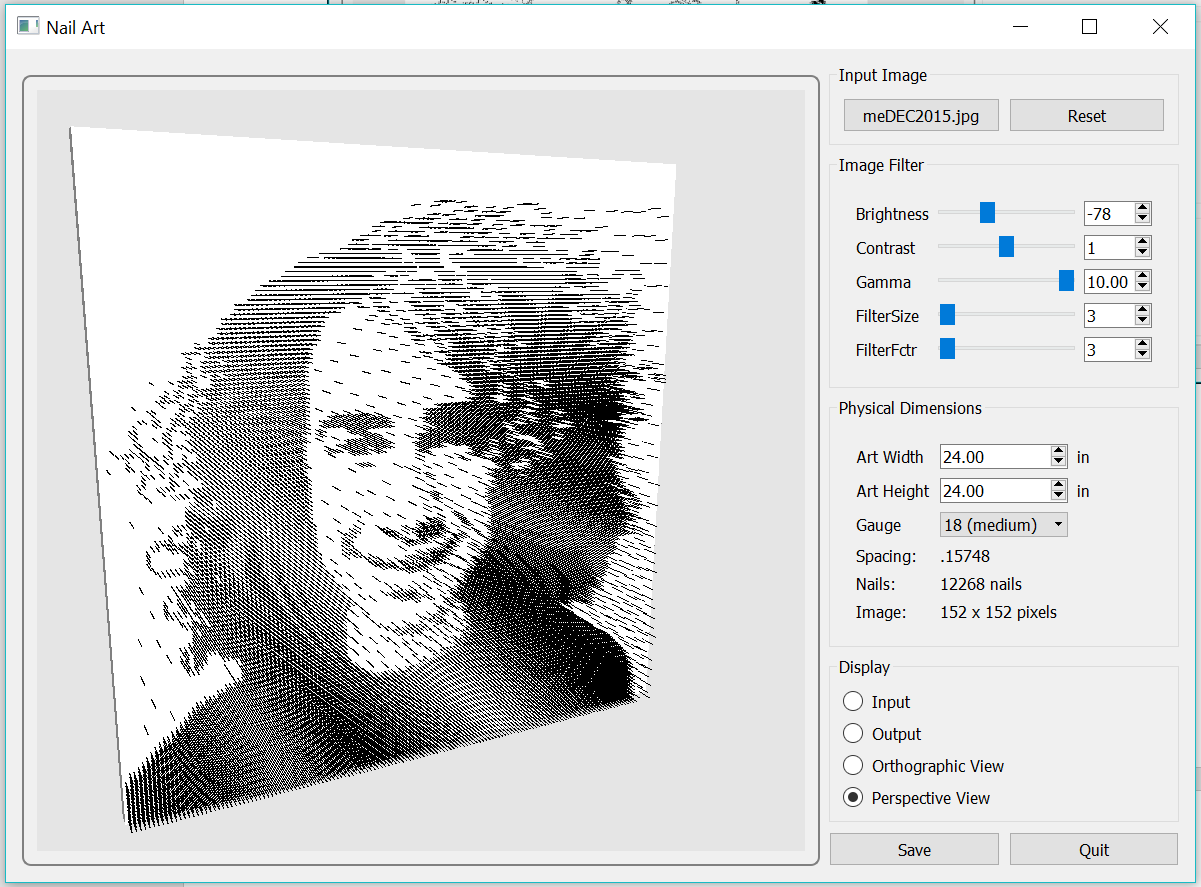
Senior Design II: The Nail Art Program

By Hafizur Rahman, Imer Jance, and Steven Eisinger

# Introduction

The goal of this semester was to create a program capable of designing and previewing (in both 2D and 3D) nail art based on an input image. Nail art, in this case, refers to nails inserted into a wooden board in such a way that the nails form an image. The idea is that this program could be connected to a machine which operates a nail gun to generate the nail art in real life quickly and easily, which we could sell to a client.



*Figure 1: ‘Nail Art’ program demonstration*

# Design and Functionality

The program is built in C++ and uses three essential libraries: Qt, Professor Wolberg’s IP library, and OpenGL. Qt is used to design the user interface. It works using the concept of signal and slot functions: user interface elements are organized into the arrangement we want, then a ‘slot’ function is called based on the ‘signal’ (the act of manipulating an interface element calls a ‘signal’ function) sent by the element. The user interface elements appear to be in a style native for the operating system they are compiled for, as Qt uses the libraries provided directly by the operating system to construct them. The buttons, sliders, spin boxes, combo boxes, radio buttons, and text labels seen in the right column of *Figure 1* are all provided in the Qt library.

Professor Wolberg’s IP library provides all necessary functions for image processing. We use these functions to manipulate the input image in a desirable way to get an output image which we can use to calculate where the nails should go on the board.

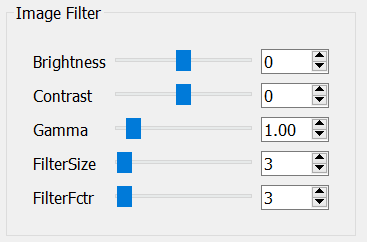


*Figure 2: Example input (left) and output (right) image from ‘Nail Art’*

The two most important uses of the IP library are converting the image to greyscale then half-toning it. Converting to greyscale is simple as one just has to multiply the red, green, and blue values by 0.2989, 0.5870, and 0.1140, respectively. Half-toning is a process which maps one or more pixels of various intensities to another group of pixels which are comprised of only black or white (no gray – in other words, color values {0, 0, 0} and {255, 255, 255}). Based on the concept that from a distance, a group of black and white pixels will be perceived as different intensities of gray, half-toning maps the lowest intensity pixel groups to show mostly white pixels in their corresponding area while the high intensity pixel groups map to areas with more black pixels. An illustration of this concept is shown in *Figure 3*. The result of half-toning on an input image in our program is shown in *Figure 2*.

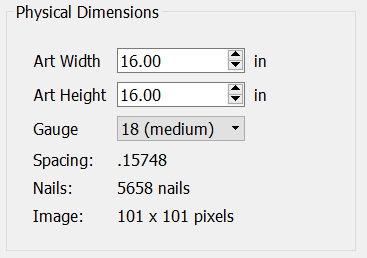
*Figure 3: Half-toning example*

Half-toning is crucial to the functionality of the program, but sometimes the output has black or white pixels in undesirable places. Our solution to this is allowing the user to change the brightness, contrast, gamma, and sharpness using sliders as shown in *Figure 4*. All of this functionality relies on the IP library, as well as Qt for the sliders. Related interface components are organized into group boxes as seen in *Figure 4, 5,* and *6*.



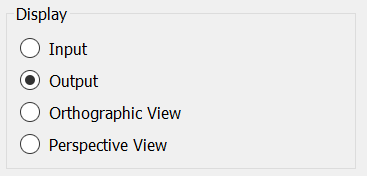
*Figure 4: Image Filter group box sliders (middle) and spin boxes (right)*

Since this program is designed to be used to model a real object, we have another set of parameters the user can manipulate to deal with the physical dimensions of the board and the gauge of the nails. This ‘Physical Dimensions’ group is shown in *Figure 5*. The dimensions of the board and the gauge of the nails can be changed using spin boxes and a combo box. The number of pixels in the output image, the number of nails required and the spacing between the nails are also displayed here. There are three gauges available: 16, 18, and 23. Spacing is predetermined based on what subjectively looked nicest based on the thickness of the nail.



*Figure 5: Physical Dimensions group box spin boxes (top) combo box (middle) and labels (bottom)*

There is a final group box (refer to *Figure 6*) consisting of radio buttons for selecting what you want to display in the large left section of the app as seen in *Figure 1*. There are four options: Input, Output, Orthographic View, and Perspective View. Input simply shows the original input image the user provided converted to greyscale. Output displays an image which is half-toned and scaled, taking into account the parameters specified in the Image Filters and Physical Dimensions group boxes (see *Figure 2* right). The last two buttons, Orthographic and Perspective View, use OpenGL to preview the board and nails in 3D in two different ways. Orthographic View (refer to *Figure 7*) shows the 3D object in space as if there was no vanishing point, meaning lines appear to be parallel (or orthogonal) to each other. Perspective View (refer to *Figure 8*) takes into account a vanishing point, meaning as we rotate the camera, different parts appear to be larger as they come closer to the viewer (also known as foreshortening). The 3D image for both of these views can be rotated by clicking the left mouse button within the OpenGL window (the left section of *Figure 1*) and dragging it.



*Figure 6: Display group box*



*Figure 7: Orthographic View*



*Figure 8: Perspective View*

There is one more group box seen at the top of the right column of *Figure 1* with two buttons. The left button loads the image to be manipulated by the program, and the right button (labeled ‘Reset’) resets all Image Filter parameters.

# Implementation

The entire program consists of six files: main.cpp, change.cpp, MainWindow.cpp, MainWindow.h, GLWidget.cpp, and GLWidget.h. We will focus on MainWindow.cpp, change.cpp, and GLWidget.cpp, as the other files just initialize variables or contain the app execution loop in the case of main.cpp. We use object-oriented programming for implementation, and the two classes we created to do this are MainWindow and GLWidget. MainWindow extends QWidget while GLWidget extends QGLWidget (both of which are part of the Qt library).

**MainWindow.cpp**

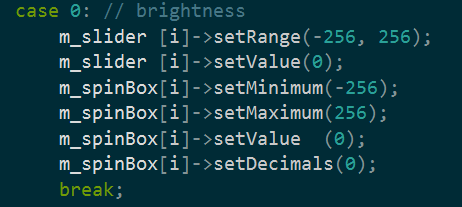
MainWindow.cpp is responsible for all user interface design and 2D image loading and manipulation. Each group box seen in the interface is constructed with its own function. There are five ‘createGroup’ functions: createGroupInput(), createGroupFilter(), createGroupDimensions(), createGroupDisplay(), and createGroupView().



*Figure 9: The simplest of the ‘createGroup’ functions, createGroupView*

All of them follow a similar format. A QGroupBox is initialized, then several varying widgets are initialized. The widgets are added to a box QHBoxLayout or QVBoxLayout, the layout is applied to the QGroupBox, the various widgets are connected to their corresponding slot functions, and the QGroupBox is returned. The QGroupBox is then placed in the correct location by the constructor.

Image Filter, seen in *Figure 4*, is created using a for loop, as all of its elements are similar arrangements of labels, sliders, and spin boxes. The sliders and labels were contained in arrays for simplicity, and a switch statement was used to initialize each slider and spin box. The QSlider class only supports integer values, while the QDoubleSpinBox supports double values, so some caution had to be observed when initializing the Gamma perameter.



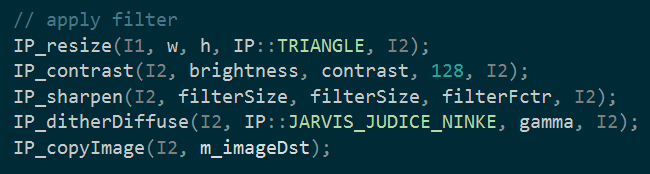
*Figure 10: Example case in the switch statement of createGroupDisplay()*

The Gamma parameter involved some conversion so that changing the slider position resulted in a 0.1 change in the spin box for each unit it was moved. This conversion was handled in the slot functions changeGammaI() (used for the slider) and ChangeGammaD() (used for the spin box) in change.cpp. The solution is simple, the slider is initialized to have ten times the number of points on it, and the value passed from the slider to the spin box is divided by ten. Likewise, the value passed from the spin box to the slider is multiplied by ten.

The most of the createGroup functions are very similar. createGroupDiminsions(), which is responsible for the Physical Dimensions group box, works very similarly to changeGroupFilter(), so there is little need to go into it in much detail. A marked difference is that some interface elements had to be placed without the aid of a for loop, as each row is comprised of different widgets. createGroupDisplay() uses a simple for loop to place all of its QRadioButtons in a QButtonGroup so that when one is selected, the others remain unselected. createGroupView() is notable in that it is comprised of a QStackedWidget. The QStacked widget has the ability to show different widgets in the same location based on the index it’s sent to. In the MainWindow class, we use the QStackedWidget to display the 2D or 3D images within the same group box as seen on the left side of *Figure 1*. The QStackedWidget in a sense consists of an array of pointers to different widgets, and displays the widget we set it to point to. We set our QStackedWidget to contain two QLabels and a GLWiget (which extends QGLWidget) of our design. The radio buttons in the Display group box are responsible for changing the widget which is displayed from the QStackedWidget by means of two functions found in MainWindow.cpp: display and displayGL().

Regarding MainWindow.cpp, there are five functions we must concern ourselves with to understand what is displayed in our View group box to the left of the application: load(), preview(), applyFilter(), display(), and displayGL(). load() brings up the dialogue box to pix a file to upload, but also uses the IP library functions IP\_readImage() and IP\_castImage() to make the image into an ImagePtr which we can more easily manipulate later, and to make it black and white before we perform half-toning operations. load() also sets the output image size to the default 16 and computes the aspect ratio of the image for later use. preview() is called by every function in change.cpp; it simply updates whatever is being displayed on the left side of the app immediately when a parameter is changed within one of the group boxes by calling applyFilter() and either display() or displayGL() with the correct argument depending on which radio button is checked in the Display group box.

applyFilter() is the function which takes every parameter provided by the sliders, spin boxes, and combo box, then applies them to the output image. As seen in *Figure 11*, the input image I1 is passed through IP\_resize(), then its output I2 is put through brightness, sharpness, and dither functions before finally being assigned to m\_imageDest which is where the image we are displaying in the View group box is stored.



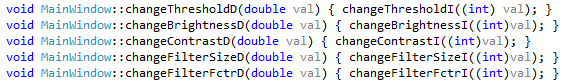
*Figure 11: IP library use in applyFilter() in MainWindow.cpp*

display() and displayGL() are actually slot functions designed for the radio buttons. display() takes a flag as an argument, either 0 or 1 depending on whether input (0) or output (1) is pressed. It sets the QStackedWidget index to the correct one then assigns the input image (in the case of 0) or the output image (in the case of 1) to the QLabel which the stacked widget points to. The IP library is used here to convert our ImagePtr to a QImage so we can use it easily with Qt. The functions displayIn() and displayOut() simply call display(0) and display(1), respectively.

displayGL() does something similar to display(), but handles the GLWidget. displayGL() changes the stacked widget index to 3 if either Perspective or Orthographic View is selected, but then calls a function defined in the GLWidget class called glSetOrtho() which takes a flag as an argument – 1 for orthographic view, 0 for perspective view. We get this flag by passing it as an argument to displayGL(). displayPersp() and displayOrtho() call displayGL(0) and displayGL(1), respectively.

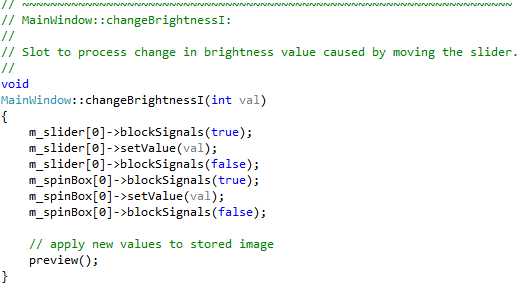
**change.cpp**

In the change.cpp file, we begin defining the change function which takes double value. Thereafter, each of them call its identical function that takes an integer value. The code is given below.



For example, if we provide 2.0 to the **chnageBrighnessD()** function, it will call the corresponding **changeBrightnessI()** with an input of 2. Since the **chnageBrighnessI()** function takes an integer value.

It will be clear if we take a look at the **changeBrighnessI()** function.

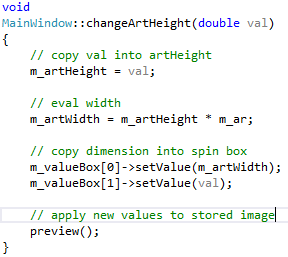
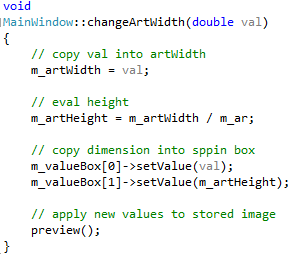


When we move the slider the brightness value changes accordingly. As we notice that we use the **blockSignals()** which suppresses the QT signals in order to allow us to set the value explicitly. Then we use the **setValue()** function which requires an integer argument to change the slider value. We use the same strategy for the rest of the sliders and spin boxes as well. At the end, we call the preview function which calls the **applyFilter()** and checks which display method is selected to produce the updated view. The same process is then repeated for all the other change functions.

In the case of **changeGauge()**, we have three different types of nail gauge size and nail spaces. We have thin, medium and thick gauge option. In the MainWindow class we have a signal and slot function that provides the argument of the **chageGauge()**. At the end of this function we also call the preview function which call **applyFilter()** and checks which display method is selected to produce the updated output image.



Another important aspect that the code had to ensure that it was maintaining the aspect ratio (width divided by height) every time the width or height are changed. That was done through the **changeArtWidth()** and **changeArtHeight()**. Both functions copy the argument value into width or height value and then evaluate the width or height based on the function call. Then we assign the new value to the spin boxes and finally call the preview function to apply the new values to stored image. The **changeArtWidth()** and **changeArtHeight()** are given below.

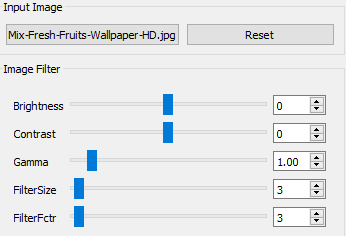
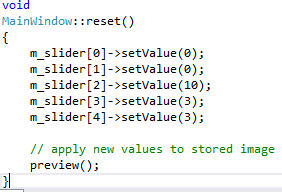


For example, if we load a wider image into our program, our program converts the image into grayscale and then the **changeArtWidth()** preserves the aspect ratio and evaluate the new height value. Then the updated value will be applied to the spin boxes and the number of nails and image pixel value will come from our display function.



As we can see in the above figure that a wider image becomes 16’’ by 10’’ based on the aspect ratio and the rest of the information are calculated from our **changeGauge()** and display function.

Lastly, to cancel all the changes we have the function **reset()** which brings the sliders (and the spinBoxes, by causation) to the default values that we choose.



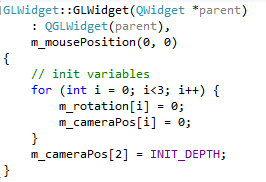
**GLWidget Class**

1. ***glWidget.cpp***

In this function we included the mainWidow.h header file in order access all the member variables of the MainWindow class. Here we begin by defining the initial depth of the camera at 3.

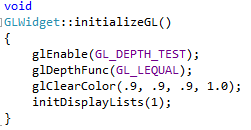


Then, we define the constructor in which we initialize the mouse position to (0,0) and the initialize the rotation array and the camera position array to (0,0,0) and (0,0,3) respectively.



In this way our camera is going to be positioned at the positive z direction but looking in the negative z-axis direction of the board.

After defining the destructor function which causes to delete the resulting lists and clear up the memory, we implement the **initializeGL()** function which sets up the scene before the display loop starts. It enables some GL capabilities that eventually allow to update the depth buffer eventually. It passes the incoming depth value if it is less than or equal to the previously stored value. After that it calls the **initDisplayList()** which initializes the display lists.



1. ***PaintGL():***

PaintGL() is one of the main functions in our nail art program. We need to move the camera to (x,y,z) then clip the z value so that it is always greater than 1 to be infront of the nail art. After getting the camera coordinates from the camera position array, namely m\_cameraPos[], we invoke the gluLookAt() to position the eye, camera and decide the up direction. The gluLookAt() function details is given below.

*gluLookAt (m\_globalEyeX, m\_globalEyeY, m\_globalEyeZ,*

*0, 0, 0,*

*0.0, 0.0, 1.0);*

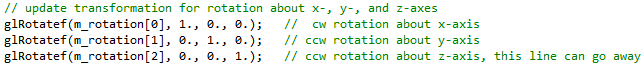
This function is the main contributor to view to the view in which we see the images. The parameters and their significances are given below.

***gluLookAt ( eyeX, eyeY, eyeZ*** *// specifies the position of the eye*

***cenX, cenY, cenZ*** *// specifies the position of the reference point*

***upX, upY, upZ);***  *// specifies the direction of the z vector*

We make all the necessary rotation if any element of the rotation array has modified.



At the end of the definition of PaintGL(), we call initDisplayList() and execute the display list (m\_nailsList).

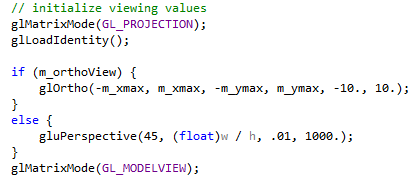
1. ***ResizeGL()***

In the **ResizeGL()**, we start by setting the view port with **glViewPort()** which takes four parameters. The first two are the x and y coordinates of the left corner of the viewport rectangle. The rest of the parameters represent the width and length of that viewport rectangle.



The width and height parameters are passed as arguments to **resizeGL().**  Then we use them to upadate our window width and height. After that we calculate the aspect ratio by dividing the width by height. Later on we calculate the m\_xmax and m\_ymax, which set the maximum x-value and y-value of the cubic field of the view.

Considering the viewing mode option, we either call the **glOrtho()** or **gluPerspective().**



The function **glOrtho()** takes the arguments as the dimensions of the cubic rectangle (left, right, bottom, top, near and far respectively). In other words, it multiplies the identity matrix by the orthographic matrix.

In the **gluPerspective()** has four parameters: the view angle in degrees, the aspect ratio, the near z-value and the far z-value with respect to the viewer.

At the end we call the **glMatrixMode()** with GL\_MODELVIEW as input to apply matrix operations to the modelview of the matrix stack.

We used other functions to implement any mouse event. For example, when the mouse is moved, pressed or released our event handler function take care those options.

1. ***InitDisplayLists()***

For the very first time, when we call this function we pass the flag ‘1’ to it. This makes a display list for the board to be drawn. And this function is calls for the first time only.

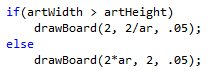


The **glGenLists(n)** generates a adjacent set of n empty display lists while glNewList() create or replaces a display list. Therefore, we are making space for one display list and placing m\_boardList in that place.

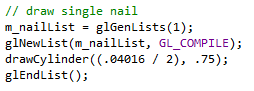
In order to get the aspect ratio and nail spacing and art dimensions we use the **getParams()**as follows.



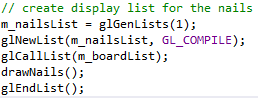
Then based on the aspect ratio (width divided by the height), we call the **drawBoard().** We use the aspect ratio, to make sure our board does not exceed 2 in height nor width. So our goal is to set that in between [-1, 1].



We use the similar procedure to draw single nail by calling the **draCylinder()** using radius and height of the cylinder as argument.

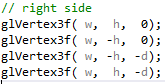
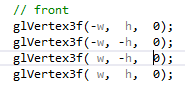
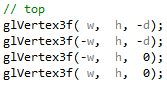


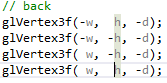
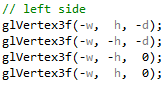
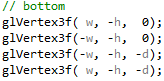
In the case of flag ‘0’ or ‘1’, a display nail list is generated and m\_board list is executed by calling the **glCallList().**  Then we call the **drawNails()** to draw nails on the board.

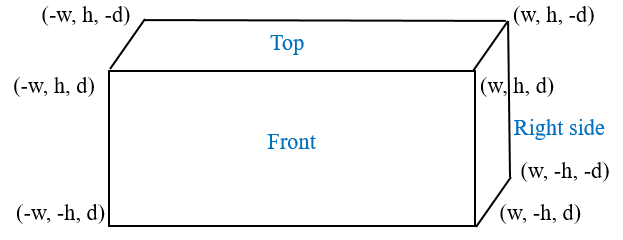


1. ***DrawBoard()***

The **drawBoard()** function was quite simple to implement in this project. We just need to visualize the board in 3-dimensions, and set the four vertices for each of its six quadrilaterals for the six sides of the board.

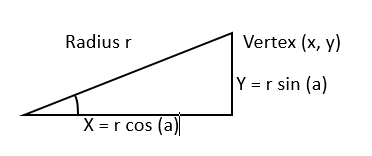


1. ***DrawCylinder()***

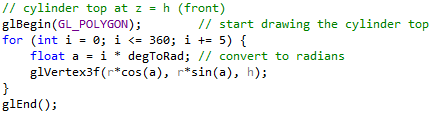
In order to draw the nails, we will consider each of them to be a cylinder of a radius r and height of h. This function takes care of drawing a cylinder. In our cylinder has no head so it’s top and bottom has the same radius. Since the function’s sine and cosine need an argument in radian, we first calculate the conversion factor from degrees to radian.



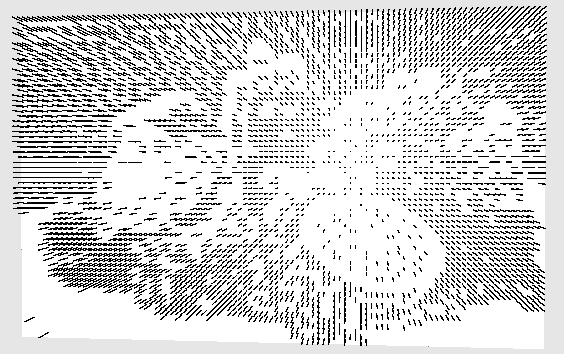
Then we set the color to black. Since our board is in white we draw the top of the cylinder of the coordinates on 72 vertices which are the distance from the origin.

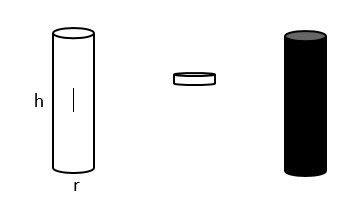


The small distance of the diameter of the cylinder (0.04016 in) draws a polygon of 72 vertices looks exactly the same as drawing a circle.

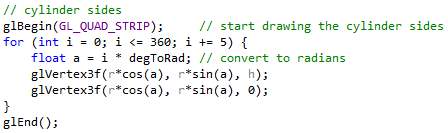


We use the same strategy to draw both top and bottom of our cylinder since both of them has same radius. As we mentioned earlier we don’t have nail head.





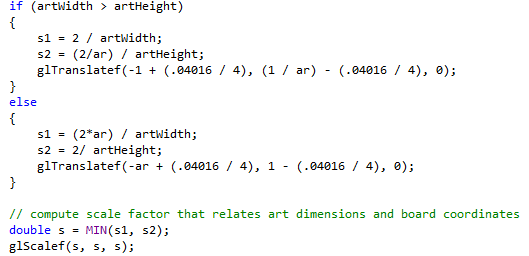
Then the sides of the cylinder are formed by adding two vertices with every iteration and GL\_QUAD\_STRIP connect them to form a connected group of quadrilaterals each time.



1. ***DrawNails()***

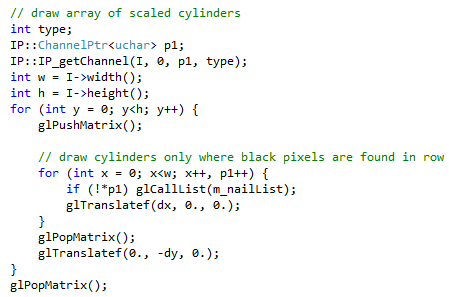
When we designed how to the nails (cylinders) will be drawn, it is easy then to draw many of them. Here we need to map them to each dark pixel of our black-and white image and place them on the board. This is done in the **drawNails()** function.

After updating the spacing value, width and height we scale the image and translate it to align with the board.

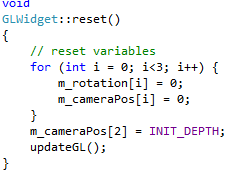


As you notice that in the glTranslatef() we consider the radius so the nails will aligned properly with the edges of the board.

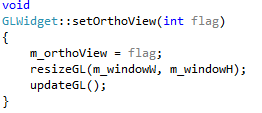
Then we use a loop to navigate through the image pixel by pixel from left to right, top to bottom. Every time it finds a black pixel, we draw a nail by calling the **m\_nailList** list.



A **reset()**  function is defined to reset the rotation angles **m\_rtotaion[]** and the camera coordinates **m\_cameraPos[]** to (0,0,0) and (0,0,3) respectively and at the end call the **updateGL()** to apply those changes to the widget.



We define the **setOrthoView()** to the end of GLWidget.cpp file. In this function we simply pass the value of the flag to our member variable **m\_orthoView**, then call the **resizeGL()** to update the window dimensions and finally call the **updateGL()** to apply those changes.



# Future Improvements

Future improvements for the program include adding manual image manipulation control, a working save button, more mouse or touch actions to manipulate the GLWidget, and connecting the app to a device which will actually print the nail art on a real board with real nails.

The limitation of using the image filter sliders is that sometimes the output isn’t ideal, even after playing around with all options. In this scenario, it would be great to have the ability for the user to manually place or remove a nail. We also have the limitation of being locked into a grid format, because the image is stored in this way. Because of this, our output might not look quite as nice as hand-made nail art.

Currently, the save button doesn’t do anything, but it could be programmed to save the image or parameters which were selected by the user so they can load it back into the program to do further manipulation later.

There are more mouse and touch actions available in openGL which we could use to add zoom functionality. We can also use functions to enable touch input for rotating and zooming.

Connecting the program to an actual machine to print the art is the most daunting task, as that is a feat of electrical and mechanical engineering. Designing that would be a project of its own.