REPORT

Face and Hand Filters using Haar Cascades detection

AUTHOR: OSKAR FORREITER MK. 2

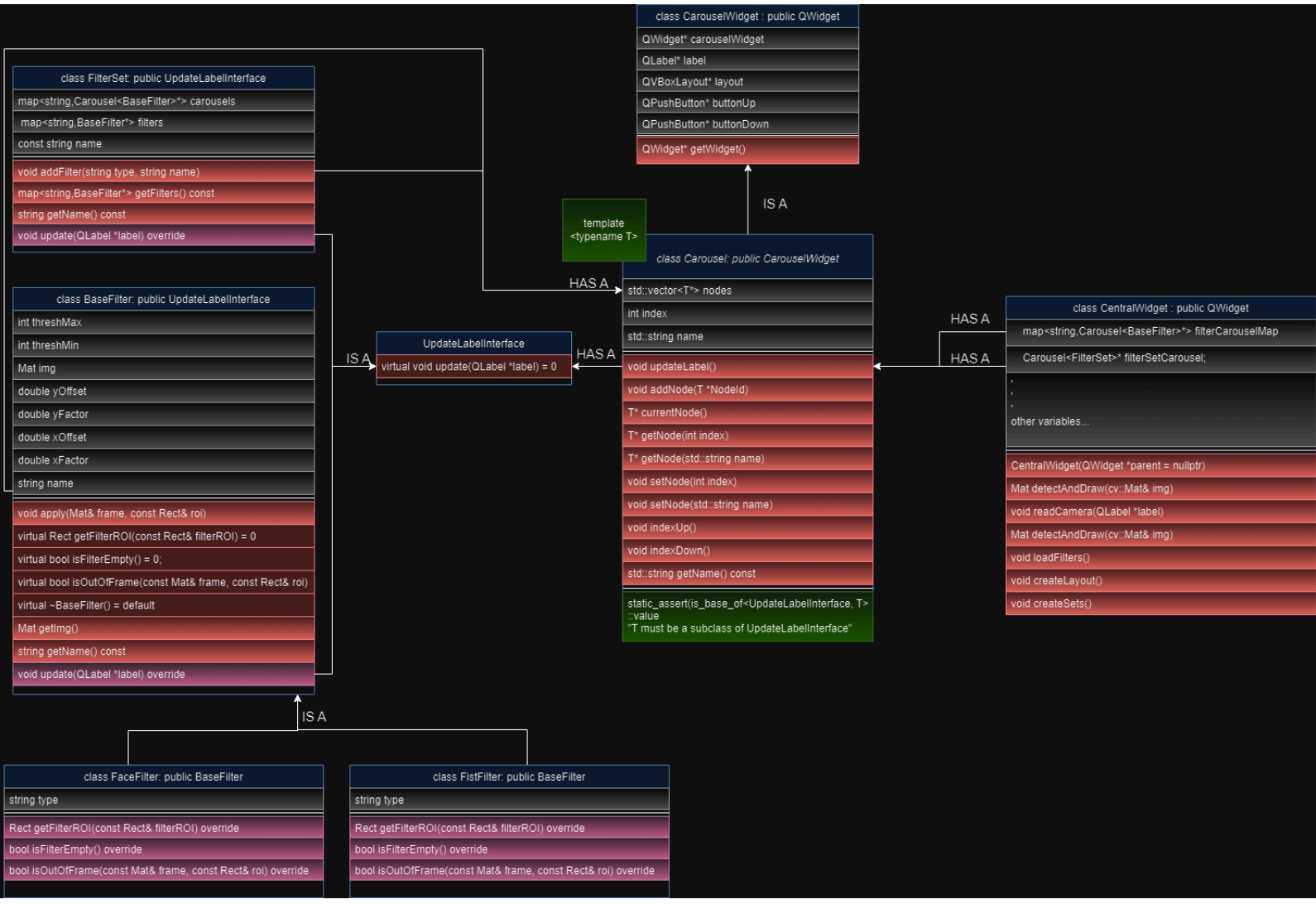
Task:

The program is a C++ desktop app with device camera input that detects faces and hands using Haar cascades and displays filters on them. Users can choose from various filters, such as glasses, hats, gloves, etc. They can also select a predefined filter set to apply a cohesive and aesthetically pleasing combination with a single click. The program utilises **QT** for the graphical user interface and **OpenCV** for image processing. Filters are loaded from the project directory: **/media/img**. To manage complexity, an object-oriented approach is employed, implementing classes like BaseFilter, Carousel, and FilterSet. Inheritance techniques and template typed classes with **static_assert** simulate interface behaviour. Exception handling is implemented for stability.

Problem Analysis:

To address the task, QT's object hierarchy is utilised. The main module is the CentralWidget within a MainWindow. The class diagram illustrates the structure, emphasising the role of BaseFilter as a parent for FaceFilter and FistFilter. Both BaseFilter and FilterSet inherit from UpdateLabelInterface, allowing Carousel to be a generic class. The CarouselWidget class is introduced to overcome QT's limitation regarding the combination of QT_OBJECT with a template typed class.

Class Diagram:



External specification:

Upon program initiation, the QT Window with CentralWidget is initialised. The CentralWidget constructor sets up layouts for filter carousels, buttons, labels, and the camera display. Users can switch between filters and filter sets while the camera continuously displays frames with applied filters. When the users are finished with the program, they will simply exit, with a big smile on their face (a genuine one, not a filter).

Internal Specification:

Program is using 9 user defined classes (8 of them included in the diagram above, not including MainWindow):

```
class MainWindow : public QMainWindow
```

This class consists only of constructor in which we initialise CentralWidget and set it as a QCentralWidget of QT QMainWindow, we also maximise the window to enhance user enjoyment of the program.

```
class CentralWidget : public QWidget
```

This class is the main driver for our program. Through it's constructor we initialise all of the program layouts and objects responsible for the underlying logic. It consists of 5 functions:

1. Void createLayout() - this function is responsible for the creation of all the carousel objects (described later in the report), labels and any other Qt objects.
2. Void loadFilters() - this function is responsible for accessing all of the images in the media/img directory and properly filling all of the Carousel objects with BaseFilters
3. Void createSets() - this function is responsible for the creation of filterSets. In there we define and compose all of the filter sets which user will be able to access through Carousel<filterSet> object
4. Void readCamera() - this function is responsible for updating the camera label with a frame read from device camera with applied filters. To apply filters readCamera function calls detectAndDraw(cv::Mat& img) function passing to it current camera frame to draw filters on
5. cv::Mat detectAndDraw(cv::Mat& img) - this function is responsible for detecting faces/hands in the frame and drawing currently chosen filters on it. Detection is performed by Haar Cascades (one for face, one for fist(hand)). Filters are drawn by use of the apply() function contained in the BaseFilter class.

```
class CarouselWidget : public QWidget
```

This class as mentioned before is to allow the Carousel class to be a template class. All of the Carousel visual (QT) aspects are contained in there, namely:

1. QWidget* carouselWidget - this is a visual base for the Carousel object
2. QLabel* label - this is responsible for showing a current object chosen in the carousel, be it a filter image, or a filterSet name.
3. QPushButton* buttonUp - Button for incrementing index by 1
4. QPushButton* buttonDown - Button for decrementing index by 1
5. QVBoxLayout* layout - layout for the whole CarouselWidget

There is also one function for accessing carouselWidget object:

- QWidget* getWidget() { return carouselWidget; }

```
template <typename T>
class Carousel: public CarouselWidget
```

We already know what is a visual aspect of a Carousel object, but let's talk about how the Carousel logic is implemented. It's important to note that the Carousel name comes from the real life analogy of a spinning carousel which we can encounter at the festivals. They are spinning objects, containing few seats which we can spin in both directions. And regardless of which direction we spin the seating will eventually end up passing again and again. Carousel class as we mentioned earlier has buttons up and down for tampering with index value, which is a code analogy to a spinning carousel. Next thing to note is the fact that it's template class with typename T. In the class declaration we use static_assert:

```
static_assert(std::is_base_of<UpdateLabelInterface, T>::value,
              "T must be a subclass of UpdateLabelInterface");
```

To make sure that the template type we pass is_base_of UpdateLabelInterface. Because most of the logic implemented in this class relies on the ability to call the only function declared in the UpdateLabelInterface which is void update().

```
// Contract for updating a label
class UpdateLabelInterface {
public:
    virtual void update(QLabel *label) = 0;
};
```

By doing so we achieve something similar to Java Interface which allows us to not know what exactly update() function does because every class that inherits from UpdateLabelInterface will need to implement it by itself. All that Carousel class needs to know is that after calling this function the Carousel label will be properly updated with the right content.

Carousel class contains 3 class members:

1. std::vector<T*> nodes - these are the nodes of pointer to type T which in our program will be either BaseFIlter or FilterSet.
2. Int index - current position of the carousel
3. 3. Std::string name - name of the carousel which is used to identify which objects this carousel contains eg. ("Beard", "Glasses" or "Filter Set")

Responsible for the Carousel logic are 10 class methods:

1. void updateLabel() - all it does is it invokes update() function from UpdateLabelInterface at the current node[index], passing it a label on which the update() function will perform some customised display
2. void addNode(T *NodeId) - as the name suggests this function adds passed object to the nodes vector
3. T* getNode(int index) and T* getNode(std::string name) - these functions allow for accessing the node object either by index or by name (if the index or name are valid, otherwise this function throws an error)
4. void setNode(int index) and void setNode(std::string name) - similarly to getNode this function sets node either by index or by name

5. void indexUp() and void indexDown() - functions that are called after clicking buttons Up or Down to change the index. After changing the index they invoke updateLabel() function.
6. std::string getName() const { return name; } - function to access the name of the Carousel

```
class BaseFilter: public UpdateLabelInterface
```

We know how works the Carousel class works and how we ensure the proper functionality by static_assert. Now it's time to understand how one of the two classes that inherit from UpdateLabelInterface which is used in the Carousel. This class is BaseFIlter and as we will later learn it's a parent class of FaceFilter and FistFilter. This class contains some members responsible for proper filter placement and the filter img itself:

```
// Threshold values
    int threshMax;
    int threshMin;

    // Image for filter
    cv::Mat img;
    // Offsets and factors for filter placement
    double yOffset;
    double yFactor;
    double xOffset;
    double xFactor;
    // Name of the filter
    std::string name;
```

This class declares also some virtual methods such as:

```
virtual cv::Rect getFilterROI(const cv::Rect& filterROI) = 0; //
Returns the filter ROI
    virtual bool isFilterEmpty() = 0; // Returns true if the filter is
empty
    virtual bool isOutOfFrame(const cv::Mat& frame, const cv::Rect&
roi) = 0;
```

Which are used in the most important function of the class that is:

```
 void apply(cv::Mat& frame, const cv::Rect& roi)
```

This function contains the whole logic on how to properly place the filter on the frame
In this class there are also two getter methods:

```
cv::Mat getImg() { return img; } // Returns the filter image
    std::string getName() const { return name; } // Returns the name of
the filter
```

And the inherited update() method:

```
    void update(QLabel *label) override; // Updates the label with the
filter image
```

```
class FaceFilter: public BaseFilter
```

And

```
class FistFilter: public BaseFilter{
```

These Classes are responsible for implementing 3 BaseFIlter virtual methods mentioned earlier. One of the important aspects of these classes is initialisation of the BaseFIlter members that ensure proper placement based on which filter type it is, eg.:

```
    //values where obtained by trial and error
    if(type == "glasses")
    {
        yOffset = 1.0 / 10.0;
        yFactor = 1.0 / 2.0;
        xOffset = 0;
        xFactor = 1;
    } else if(type == "beard")
    {
        yOffset = 7 / 16.0;
        yFactor = 1.0 / 2.0;
        xOffset = 0;
        xFactor = 1;
```

These are class specific as there are different filters in FaceFilter class and FistFilter class

```
class FilterSet: public UpdateLabelInterface
```

Last class to mention is FilterSet. This class goal is to provide a framework for organising filters in specific sets, that we can choose with one click of FilterSet Carousel button.
Class members are:

```
 std::map<std::string,Carousel<BaseFilter>*> carousels; // map
containing all off the button carousels
        std::map<std::string,BaseFilter*> filters; // map containing
all of the filters in a set
        const std::string name; // name of the set
```

Carousels map contains pointers to all of the filter carousels so that whenever we change filterSet we can update all of the filters accordingly. It also contains a filters map with pointers to BaseFilters which are the specific filters that are added to the set in the set creation process in the CentralWidget createSets() function. Last member is name which is a unique identifier for the FilterSet that is printed on the FilterSetCarousel label eg. “French Lord” or “Fancy Zulu”

There are two ways of adding filters to the set, we can either pass a map<string,string> that contains all of the filter categories with their chosen filter names, along with a map with pointers to BaseFilter Carousel. Or we can use a class method addFilter:

```
FilterSet(std::string name, std::map<std::string,Carousel<BaseFilter>*>
carousels,
                std::map<std::string,std::string>set) : name(name),
carousels(carousels)
```

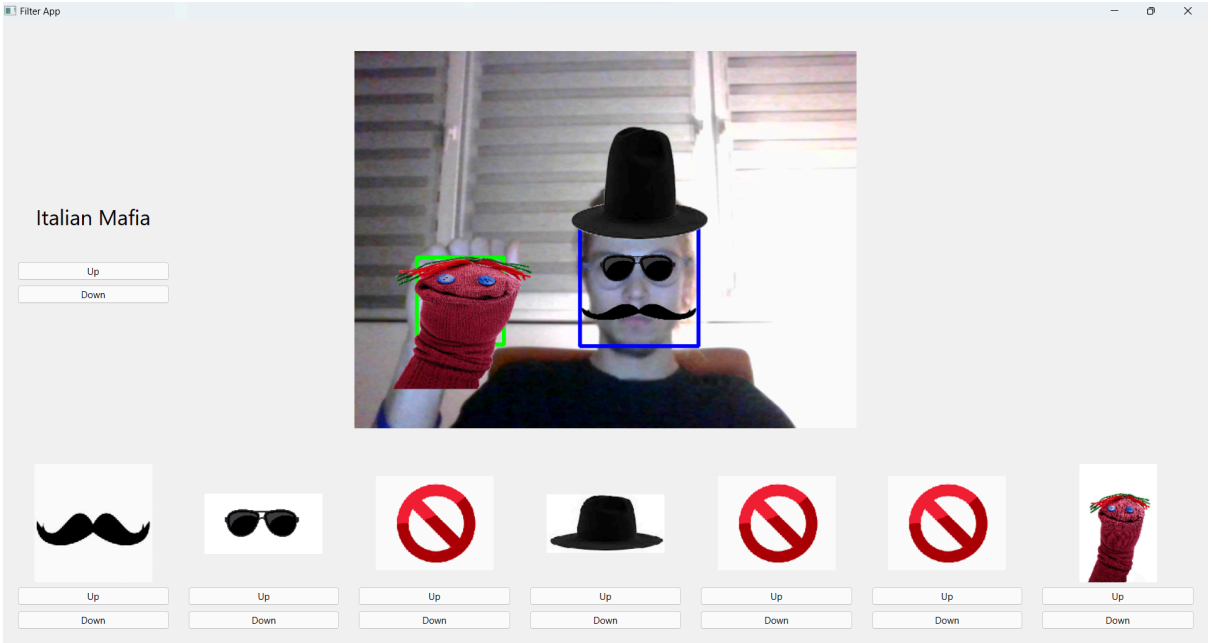
```
        {
            for (auto& filter : set)
            {
                addFilter(filter.first,filter.second);
            }
        }

        // Method to add a filter to the set
        void addFilter(std::string type, std::string name) {
            filters[type] = carousels[type]->getNode(name);
        }
```

There is also a getter for name:

```
// Method to get the name of the set
    std::string getName() const {
        return name;
    }
```

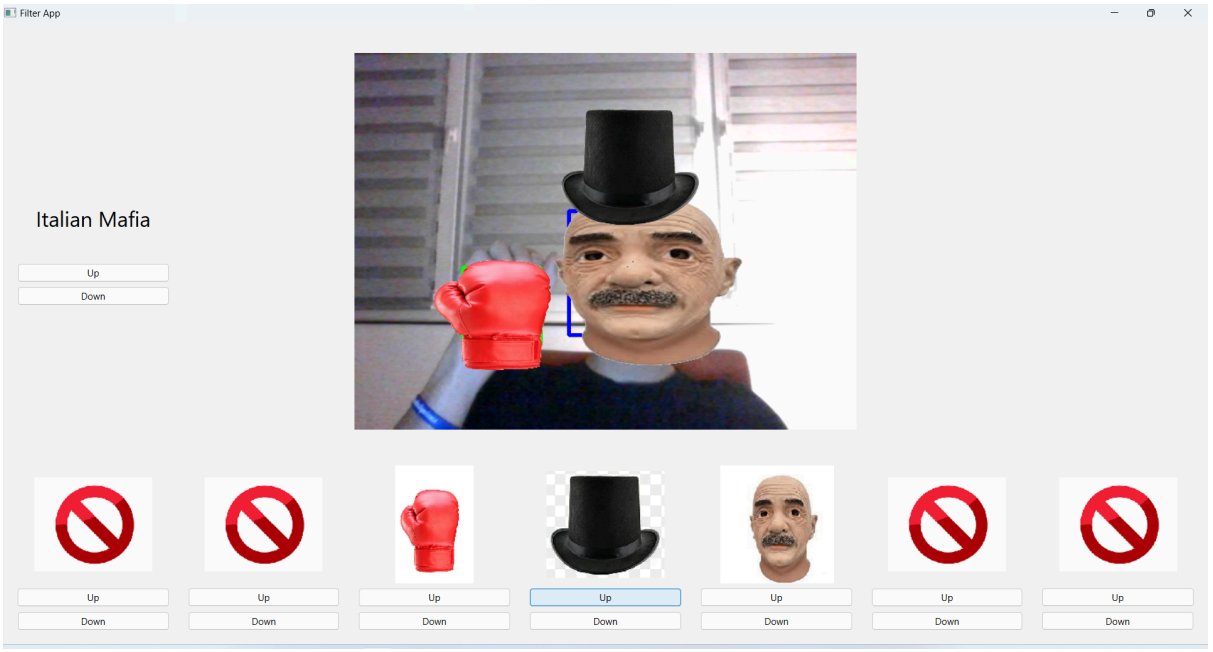
And method inherited from the UpdateLabelInterface(), namely update(). Whenever this function is called. We change the label of the FIlterSet to the current set name and recursively go through all of the BaseFilter Carousels and set their nodes to the filters contained in the set, which in turn invokes BaseFilter update() method and updates their labels. This may seem complicated at the first glance as the update call of one class initiates several update() calls in another class, but this behaviour is the most important aspect of the program class structure that allows us to do a lot of work with as little code as possible. Simple and elegant.

Test:

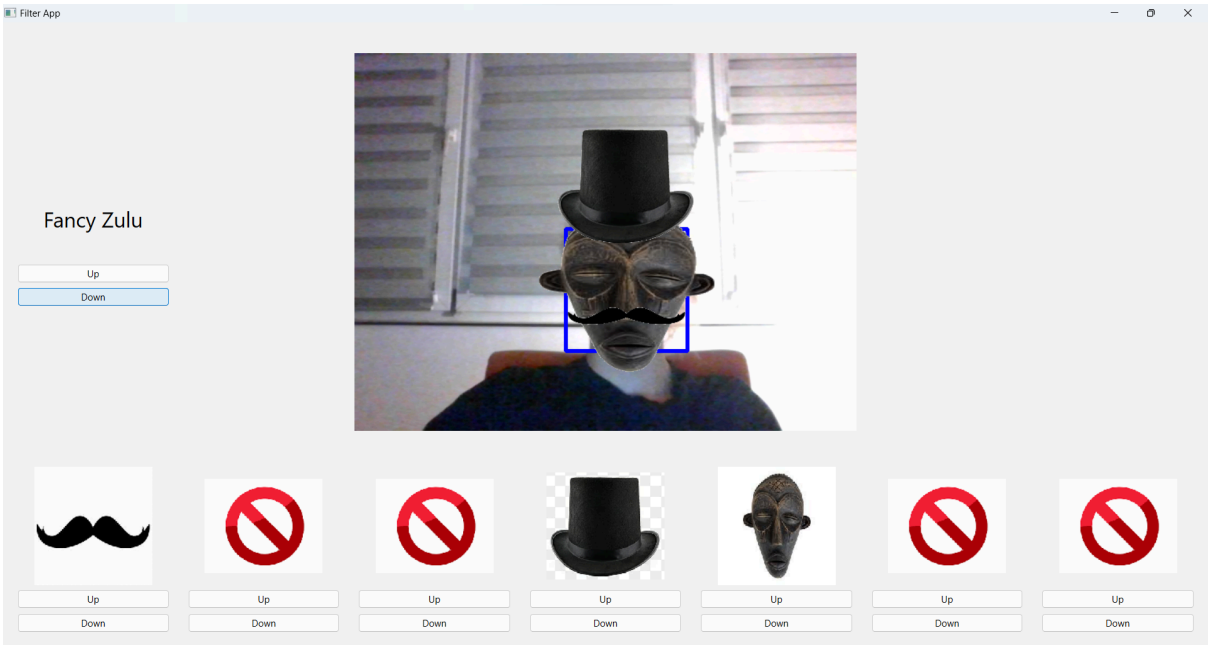


The program initiates with the "Italian Mafia" set. Detected face and hand display filters like moustache, glasses, a fancy fedora, and a sock puppet. User interactions with filters and sets yield diverse and enjoyable outcomes.

By changing the filters via the buttons user can end up with the results like this:



Or by changing the filter set with a result like this one:



The possibilities of this program are endless (or near endless) and the joy one can experience by the interaction with it is immeasurable.

Conclusions:

1. **Implementation Approach**: The chosen approach of using an object-oriented design with classes such as BaseFilter, Carousel, and FilterSet demonstrates a structured and modular design. The use of inheritance and templates enhances code extensibility and maintainability.
2. **Graphical User Interface (GUI):** The integration of QT for the graphical user interface provides a user-friendly experience. The carousel concept for selecting filters and filter sets is intuitive, mimicking real-life carousel interactions.
3. **Filter Variety:** The ability to load filters from a project directory and categorise them into sets allows for a diverse range of customization. Users can easily switch between different filters and sets, enhancing the program's entertainment value.
4. **Error Handling and Stability:** The consideration of exceptions in areas prone to errors suggests a focus on program stability. This is essential for providing a seamless user experience and preventing unexpected crashes.
5. **Haar Cascades Detection:** The use of Haar Cascades for face and hand detection is a robust choice, providing accurate results. The detection and application of filters in real-time contribute to the interactive nature of the application.
6. **Filter Placement Logic:** The BaseFilter class, along with its subclasses (FaceFilter and FistFilter), encapsulates the logic for proper filter placement on detected faces and hands. The implementation of virtual methods allows for flexibility and customization in filter behaviour.
7. **FilterSet Organization: T**he FilterSet class introduces a systematic way to organise filters into sets. The dynamic updating of filter carousels when switching sets showcases an efficient way to manage and apply different combinations of filters.
8. **Testing and Results:** The provided test scenarios demonstrate the program's capability to detect faces and hands, apply filters in real-time, and allow users to create diverse combinations. The visual outcomes showcase the program's potential for entertainment and creative expression.
9. **Room for Improvement:** While the current implementation appears robust, there may be opportunities for further refinement, such as optimising filter loading, enhancing the GUI aesthetics, or exploring additional features like user-defined filter creation.
10. **User Experience:** The program aims to provide a joyful and interactive user experience, evident in the variety of filters and sets. The real-time application of filters on detected faces and hands adds an element of fun and creativity.
11.

In conclusion, the program successfully combines image processing, GUI design, and object-oriented principles to create an interactive face and hand filter application. The elegant class structure simplifies code maintenance and promotes extensibility.