**4. РАЗРАБОТКА ПРОГРАМНЫХ МОДУЛЕЙ**

**4.1 Программный модуль рисования**

Рисование начинается с того, что пользователь выбирает текущую операцию рисования, будь то некоторая фигура или заливка. По умолчанию задана фигура – эллипс. При выборе фигуры срабатывает один из слотов-инициализаторов функции создания фигуры. Принадлежат эти слоты классу DrawingArea

void setCreateEllipse();

void setCreateRectangle();

void setCreateLine();

void setCreateCurve();

void setCreateEraser();

void setCreateFilledShape();

Каждый слот занимается тем, что инициализирует указатель на функцию создания фигуры, который содержится в DrawingArea (см. ПРИЛОЖЕНИЕ А), той функцией, что соответствует выбранной фигуре. К примеру, метод инициализации эллипсом

void DrawingArea::setCreateEllipse()

{

this->\_createShape = &createEllipse;

}

где

ShapePointer createEllipse(QImage\* image,

const QPoint &topLeft,

int penWidth,

const QColor &penColor)

{

return ShapePointer(new Ellipse(image, topLeft,

penWidth, penColor));

}

Как было упомянуто в разделе 3, все классы фигур имеют одинаковый список параметров инициализации, для упрощения работы с ними, поэтому и функции, создающие их, требуют те же самые параметры (см. ПРИЛОЖЕНИЕ Д) – отличаться будет только процесс рисования.

Теперь, когда указатель на функцию создания инициализирован можно приступить к рисованию. Когда пользователь нажимает мышью в рабочей зоне, то срабатывает событие mousePressEvent, создающее фигуру:

void DrawingArea::*mousePressEvent*(QMouseEvent \*event)

{

if (event->button() == Qt::LeftButton)

{

this->\_currentShape = \_createShape(&this->\_image,

event->pos(),

this->\_penWidth,

this->\_penColor);

update(this->\_currentShape->*rect*());

}

}

При движении мышью используются чисто виртуальные методы *rect*() и *update*() класса *Shape,* которые переопределены в классах наследниках. Эти методы нужны для определения размеров и позиции прямоугольника, в который будет вписана данная фигура:

void DrawingArea::*mouseMoveEvent*(QMouseEvent \*event)

{

if ((event->buttons() & Qt::LeftButton) &&

\_currentShape)

{

const QRect prevRect =

this->\_currentShape->*rect*();

this->\_currentShape->*update*(event->pos());

update(\_currentShape->*rect*().united(prevRect));

}

}

Как только пользователь прекращает рисовать и отпускает клавишу, срабатывает событие mouseReleaseEvent, которое передает команду рисования в стек команд, для возможности отката или повторения:

void DrawingArea::*mouseReleaseEvent*(QMouseEvent \*event)

{

if (event->button() == Qt::LeftButton && \_currentShape)

{

this->\_undoStack->push(new DrawCommand(this,

&this->\_image, \_currentShape));

this->\_currentShape = nullptr;

}

}

До этого момента рисование велось на самом виджете DrawingArea, который постоянно обновлялся с помощью базового метода QWidget::update(). Теперь же нужно перенести итоговую фигуру на изображение. В зависимости от типа фигуры будет срабатывать чисто виртуальный метод draw() переопределенный во всех классах-наследниках Shape, а происходить рисование будет в контексте метода класса DrawCommand redo():

void DrawCommand::*redo*()

{

QPainter painter(this->\_image);

this->\_shape->*draw*(painter);

this->\_drawingArea->update(this->\_shape->*rect*());

}

Если потребуется откат, то будет вызван метод класса DrawCommand undo():

void DrawCommand::*undo*()

{

const QRect rect = this->\_shape->*rect*();

QPainter painter(this->\_image);

painter.drawImage(rect, this->\_undoImage);

this->\_drawingArea->update(rect);

}

**4.2 Реализация модуля коррекции гаммы**

Рассмотрим алгоритм коррекции гаммы, используемый в данном проекте. Для выполнения операции необходимо обойти весь растовый массив изображения, при этом подвергая цвет каждого пикселя искажению при помощи математической операции (см. ПРИЛОЖЕНИЕ В). Ниже приведен код самого алгоритма

void CorrectionCommand::adjust()

{

int height = this->\_image->height();

int width = this->\_image->width();

for (int i = 0; i < height - 1; ++i)

{

QRgb \*line = reinterpret\_cast<QRgb\*>(this->\_image->scanLine(i));

for (int j = 0; j < width - 1; ++j)

{

QRgb pixelColor = line[j];

int r = qRed(pixelColor);

int g = qGreen(pixelColor);

int b = qBlue(pixelColor);

r = std::pow(double(r)/255, this->\_gamma) \* 255;

g = std::pow(double(g)/255, this->\_gamma) \* 255;

b = std::pow(double(b)/255, this->\_gamma) \* 255;

line[j] = qRgb(r,g,b);

}

}

}

**4.3 Реализация модуля заливки**

Алгоритм заливки в данном проекте основан на алгоритме обхода графа в ширину. Имеется некоторая очередь пикселей, в голове который стоит пиксель, на который нажал пользователь. При рассмотрении пикселя из головы, в том случае, если его цвет совпадает со значением цвета, который надо залить, мы меняем его цвет, удаляем его из очереди, а потом добавляем в очередь всех его соседей из окрестности фон Неймана, таким образом обходя всю область, где значение цвета пикселей совпадает с тем, что требуется залить (см. ПРИЛОЖЕНИЕ Г). Ниже приведен код реализации алгоритма.

FilledShape::floodFill(const QPoint &pos, const QRgb &newColor)

{

std::vector<QPoint> modified;

QImage image = this->\_image->copy();

QRgb oldColor = image.pixel(pos);

if (oldColor == newColor)

{

return modified;

}

std::queue<QPoint> nodeQ;

nodeQ.push(QPoint(pos.x(), pos.y()));

while(!nodeQ.empty())

{

QPoint currNode = nodeQ.front();

nodeQ.pop();

if(image.pixel(currNode) == oldColor)

{

image.setPixel(currNode.x(),currNode.y(), newColor);

modified.emplace\_back(currNode);

if(currNode.x() > 0)

nodeQ.push(QPoint(currNode.x()-1, currNode.y()));

if(currNode.x() < (image.width() - 1))

nodeQ.push(QPoint(currNode.x()+1, currNode.y()));

if(currNode.y() > 0)

nodeQ.push(QPoint(currNode.x(), currNode.y()-1));

if(currNode.y() < (image.height() - 1))

nodeQ.push(QPoint(currNode.x(), currNode.y()+1));

}

}

return modified;

}

**4.4 Реализация модуля медианной фильтрации**

Алгоритм медианной фильтрации заключается в обходе растового массива изображения некоторой апертурой – пикселя и его окрестности Мура. Т.е. в данном случае апертура состоит из 9 пикселей (или из 4, если мы находимся в крайних положениях изображения). Апертура строится для каждого пикселя изображения, и на каждом шаге ее движения по растру происходи следующее: красная, зеленая и синяя составляющие каждого пикселя из апертуры заносятся в соответствующий по цвету массив (красный, синий, зеленый). В каждом массиве определяется среднее значение цвета, после чего все 3 составляющие опять комбинируются вместе, но теперь присваиваются каждому пикселю в апертуре. Ниже приведен код алгоритма.

int FiltraringCommand::findMedian(int\* Array, const int size)

{

int i;

int j;

int median\_index;

int temp;

int median;

for (i = 1; i < size; i++)

{

j = i - 1;

temp = Array[i];

while (j >= 0 && temp < Array[j])

{

Array[j + 1] = Array[j];

j--;

}

Array[j + 1] = temp;

}

median\_index = (int)(size / 2);

median = Array[median\_index];

return median;

}

void FiltraringCommand::filtrate()

{

const int array\_size = 9;

int row = 0;

int coloumn = 0;

int size = 0;

int row\_limit = 3;

int coloumn\_limit = 3;

int red\_array[array\_size];

int green\_array[array\_size];

int blue\_array[array\_size];

for (int i = 0; i < this->\_image->height(); i++)

{

for (int j = 0; j < this->\_image->width(); j++)

{

row = i - 1;

row\_limit = 3;

if (i == this->\_image->height() - 1 || i == 0)

{

row\_limit = 2;

if(i == 0) row = i;

}

for (int y = row, k = 0; row\_limit--; y++)

{

coloumn = j - 1;

coloumn\_limit = 3;

if (j == this->\_image->width() - 1 || j == 0)

{

coloumn\_limit = 2;

if (j == 0) coloumn = j;

}

for (int x = coloumn; coloumn\_limit--; x++)

{

red\_array[k] = qRed(this->\_image->pixel(x, y));

green\_array[k] = qGreen(this->\_image->pixel(x, y));

blue\_array[k] = qBlue(this->\_image->pixel(x, y));

k++;

size = k;

}

}

int r = findMedian(red\_array, size);

int g = findMedian(green\_array, size);

int b = findMedian(blue\_array, size);

QRgb pixelColor = qRgb(r, g ,b);

this->\_image->setPixel(j, i, pixelColor);

memset(red\_array, 0, array\_size \* sizeof(int));

memset(green\_array, 0, array\_size \* sizeof(int));

memset(blue\_array, 0, array\_size \* sizeof(int));

size = 0;

}

}

}