МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ ДНР
ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту по дисциплине:
«Архитектура и проектирование графических систем»
тема курсового проекта:
«Разработка графического редактора для работы с параметризованными трёхмерными объектами»

Руководитель:
_____кафедры
_____

Выполнил:
ст. гр. ПИ–18б
Моргунов А.Г.

ДОНЕЦК – 2021

РЕФЕРАТ

Пояснительная записка к курсовому проекту содержит: 65 страниц, 27 рисунок, 4 источника, 5 приложений.

Целью курсового проектирования является проектирование и создание графического редактора, который может работать с трехмерными параметризованными объектами, определенного типа. В числе реализованных функций должны быть такие операции как: поворот, перенос, масштабирование объекта; панорамирование, зумирование; сохранение сцены в читаемую базу данных, работа с несколькими объектами, работа с камерой.

Для реализации цели курсового проекта необходимо: изучить алгоритмы аффинных преобразований, центрального и параллельного проецирования, алгоритмы работы с камерой и удаления невидимых линий, выбрать подходящие алгоритмы из существующих, выбрать программные средства реализации.

Результатом проекта является графический редактор для работы с трехмерными параметризованными объектами на языке C++.

ПОЛИГОН, КАМЕРА, АФИННЫЕ ПРЕОБРАЗОВАНИЯ, МАТРИЦЫ, ОБЪЕКТ, ТРИАНГУЛЯЦИЯ, Z-БУФЕР, ПРОЕЦИРОВАНИЕ, ПРОЕКЦИЯ

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Графический редактор — программа, позволяющая создавать, просматривать, обрабатывать и редактировать цифровые изображения на компьютере. В данном проекте мы остановимся на графическом редакторе, который умеет работать с объектами типа «Дрель», производить различные операции над камерой, сценой и объектами на сцене.

При выполнении курсового проекта были рассмотрены и проанализированы различные алгоритмы и подходы, их особенности. После изучения были выбраны и частично модифицированы удовлетворяющие нашему заданию по функционалу и производительности подходы и алгоритмы.

Объектом проекта является разработка алгоритмов визуализации трехмерных моделей и операции с камерой и объектами в пространстве.

Предметом проекта является графический редактор, который умеет работать с определенными объектами.

1 РАЗРАБОТКА ПОЛИГОНАЛЬНОЙ МОДЕЛИ ОБЪЕКТА

Полигональные (плоскостные) модели, или полигональные сетки, находят в компьютерной графике самое широкое применение. Они представляют поверхности геометрических объектов в виде набора состыкованных друг с другом плоских полигонов. Традиционное для компьютерной графики описание полигональной модели объекта является иерархическим и включает список вершин, список ребер и список полигонов объекта. [1]

В курсовом проекте разрабатывается полигональная модель объекта «Дрель». Полигональное и проволочное представления представлены на рисунках 1.1 и 1.2 соответственно.

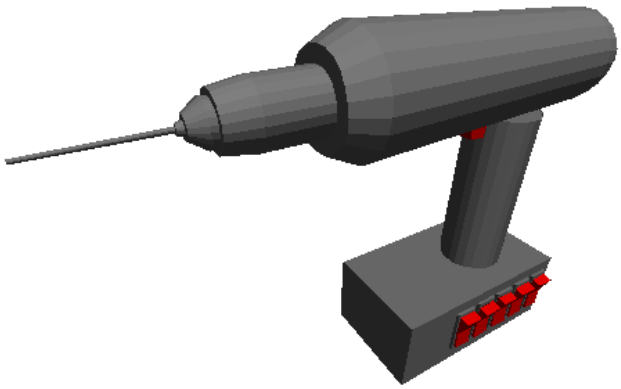


Рисунок 1.1 – Полигональное представление

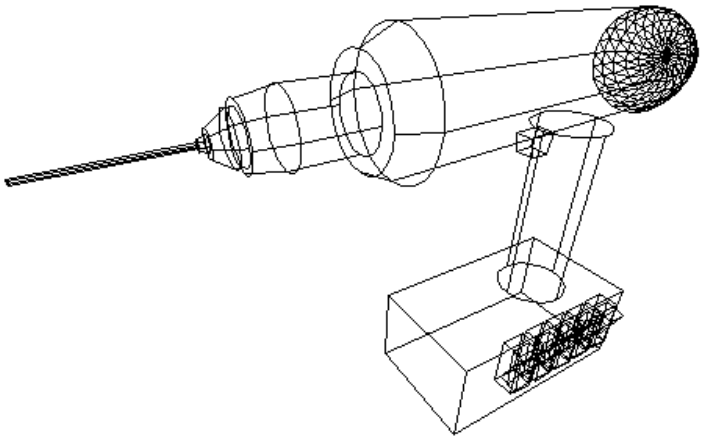


Рисунок 1.2 – Проволочное представление

1.1 Составляющие элементы объекта

Составляющими элементами модели являются такие фигуры как:

- полусфера;
- цилиндр;
- усеченный конус;
- усеченная пирамида;
- пирамида;
- прямоугольный параллелепипед.

Каждая из этих фигур разбивается на полигоны, которые и отрисовываются на сцене. Полусфера задается радиусом, центральной точкой и числом от 0 до 1, которое влияет на сечение окружности. Цилиндр задается радиусом, высотой, центральной точкой нижнего основания. Усеченный конус задается нижним радиусом, верхним радиусом, высотой, центральной точкой нижнего основания. Усеченная пирамида задается шириной и длиной нижнего основания, шириной и длиной верхнего основания, высотой, центральной точкой нижнего основания. Пирамида задается шириной и длиной нижнего основания, высотой, центральной точкой нижнего основания. Прямоугольный параллелепипед задается высотой, шириной, длиной, точкой основания.

1.2 Триангуляция поверхности объекта

Триангуляция – это разбиение объектов на треугольные полигоны. Преимуществом триангуляции является то, что для отрисовки объекта достаточно хранить список треугольников, из которых он состоит.

В курсовом проекте объекты триангулируются следующим образом. Прямоугольный параллелепипед разбивается на 6 граней, каждая из которых разбивается на 2 треугольника. Сфера разбивается на прямоугольники, каждый из которых разбивается на 2 треугольника. Усеченный конус и цилиндр разбиваются на 2 окружности, которые в свою очередь разбиваются

на треугольники, и боковую поверхность, которая также разбивается на треугольники. Пирамида разбивается на 4 треугольника и основание – четырехугольник. Усеченная пирамида разбивается на 4 треугольника и 2 четырехугольника.

2 ОПИСАНИЕ ВЫБРАННЫХ МЕТОДОВ И АЛГОРИТМОВ ВИЗУАЛИЗАЦИИ

2.1 Алгоритмы афинных преобразований

Преобразование плоскости называется аффинным, если оно непрерывно, взаимно однозначно и таким образом любой прямой является прямая. Частными случаями аффинных преобразований являются движение и масштабирование (сжатие и растяжение) объекта. Эти виды преобразований реализованы в программе. Шаги алгоритма аффинных преобразований:

1) Сформировать матрицу преобразования;
2) Если требуется, умножить ее справа на матрицу переноса объекта;
3) Если требуется, умножить справа получившуюся матрицу преобразования на матрицу обратного переноса;
4) Применить итоговую матрицу преобразования к каждой точке объекта.

2.2 Алгоритм удаления скрытых линий

Для реалистичного отображения объекта необходимо применять алгоритм удаления скрытых линий. В проекте используется алгоритм Z-буфера. Этот алгоритм заполняет массив, который хранит координаты z всех точек, которые не перекрываются другими точками.

В данном проекте Z-буфер был модифицирован, и теперь, помимо Z-координат точек, он хранит указатель на объект, которому принадлежит пиксель (для выделения объектов).

2.3 Алгоритмы проекционных преобразований

Для того чтобы увидеть на плоскости монитора трехмерное изображение, нужно уметь задать способ отображения трехмерных точек в двумерные. Способ перехода от трехмерных объектов к их изображениям на плоскости называется проекцией. Проекция трехмерного объекта

(представленного в виде совокупности точек) строится при помощи прямых проекционных лучей, называемых проекторами, проходящих через каждую точку объекта, пересекая картинную плоскость, образуя проекцию.[2]

2.3.1 Алгоритм параллельного проецирования

Параллельное проецирование можно рассматривать как частный случай центрального проецирования. Если центр проекции удален на бесконечность, то проекция – параллельная. [2]

В данном проекте используется перпендикулярное параллельное проецирование (рис. 2.1). Для него используется единичная матрица 4x4.

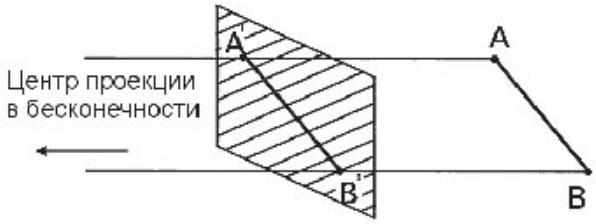


Рисунок 2.1 – Параллельная проекция

2.3.2 Алгоритм центрального проецирования

Центральная проекция приводит к визуальному эффекту перспективного укорачивания, когда размер проекции объекта изменяется обратно пропорционально расстоянию от центра проекции до объекта. [2]

В данном проекте реализована одноточечная центральная проекция объекта. (рис. 2.2)

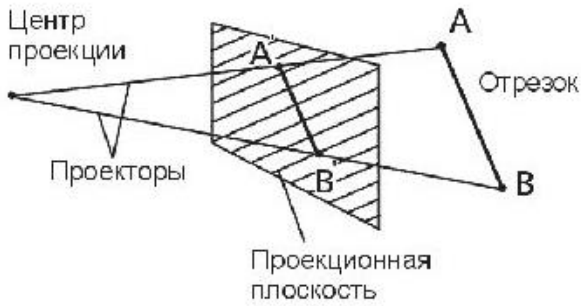


Рисунок 2.2 – Центральная проекция

Алгоритм:

1) Задается точка центра проецирования и точка наблюдения, а также два фокусных расстояния.

2) В зависимости от этих параметров формируется матрица проекции.

3) Каждая точка объекта преобразуется с помощью полученной матрицы.

Отсекаются объекты, расположенные ближе переднего и дальше дальнего фокуса – они невидимы наблюдателю.

2.4 Алгоритм произвольных видовых преобразований (камеры)

Камера задается точкой, в которой располагается камера, точкой, определяющей направление взгляда, вектором, который определяет угол поворота камеры относительно направления взгляда.

В проекте реализованы два вида перемещения камеры. Перемещение камеры с сохранением точки, в которую смотрит камера. Для этого меняется только положение камеры. Второй способ – это свободная камера. Особенностью свободной камеры является то, что при передвижении меняется не только координаты камеры, но и точка, в которую он смотрит. Также для поворота камеры относительно направления взгляда может изменяться вектор, который и задает угол поворота.

3 РАЗРАБОТКА СТРУКТУР ДАННЫХ ДЛЯ ХРАНЕНИЯ И ОПИСАНИЯ ОБЪЕКТА

3.1 Описание структур данных

Для описания объектов был реализован абстрактный класс Shape, который хранит в себе информацию обо всех параметрах объекта. Все классы, которые определяют объекты являются наследниками класса Shape (рис 3.1).

Все объекты имеют собственные параметры, но при отрисовке каждый объект представляется в виде треугольников.

```
std::optional<QRgb> color_{};
std::optional<QRgb> brush_{};
QMatrix4x4 modify_matrix_;
QVector3D base_point_;
Shape* parent_ = nullptr;

QVector4D i{1, 0, 0, 0};
QVector4D j{0, 1, 0, 0};
QVector4D k{0, 0, 1, 0};
```

Рисунок 3.1 Поля класса Shape

3.2 Диаграммы UML взаимодействия классов

В проектироуемой системе выделено 6 классов, отвечающих за внутреннее устройство системы, 1 абстрактный класс, а также 10 классов, определяющих объекты.

Системные классы:

– Camera - Класс, который хранит тип проекции.

– LookAtCamera – Класс, который хранит все параметры камеры

– GraphicsView – Класс, который выводит изображение на экран, а также хранит параметры изображения.

– Painter – Класс, который отрисовывает все объекты, хранит информацию о типе отображения (полигональный, проволочный).

– ZBuffer – Класс, описывающий Z-buffer

– GraphicsScene – Класс, описывающий сцену, в которох хранятся объекты

Абстрактный класс – Shape. Он является суперклассом для всех объектов.

Классы, обозначающие объекты:

– Box – Класс, описывающий прямоугольный параллелепипед

– Drill - Класс, описывающий дрель

– Ellipse - Класс, описывающий эллипс

– Ellipsoid - Класс, описывающий эллипсоид

– Frustum - Класс, описывающий конус, усеченный конус

– Line - Класс, описывающий линию

– Pyramid - Класс, описывающий пирамиду, усеченную пирамиду

– Quadrangle - Класс, описывающий четырехугольник

– Switch - Класс, описывающий переключатель

– Triangle - Класс, описывающий треугольник

Диаграммы классов (рис 3.2, 3.3) являются одной диаграммой, разделенной на 2 части. Из-за большого размера итоговой диаграммы было принято решение разделить ее на 2 части. Первая часть описывает общее устройство системы, а вторая часть иерархию классов, определяющих фигуры.

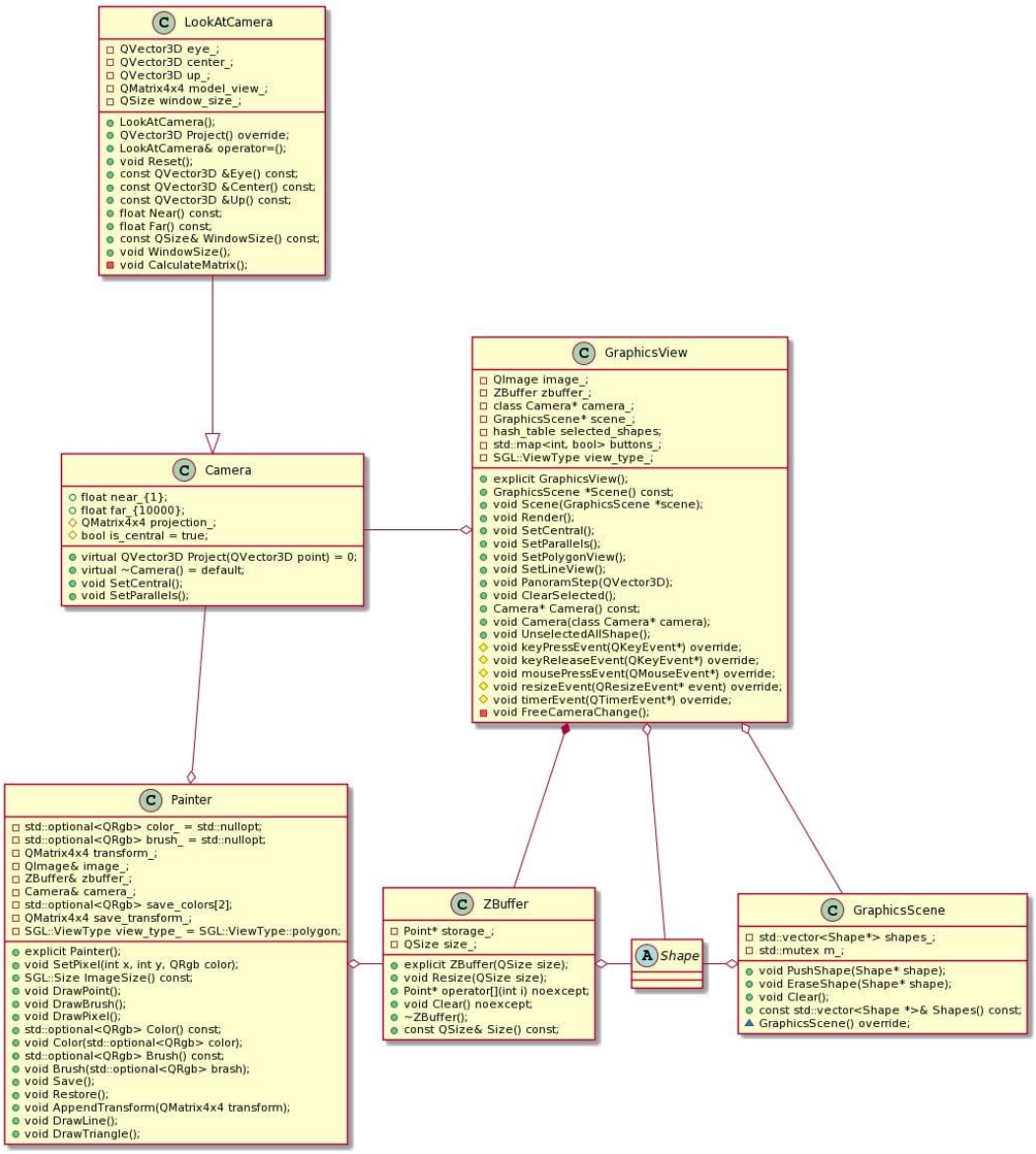


Рисунок 3.2 – Диаграмма классов системы

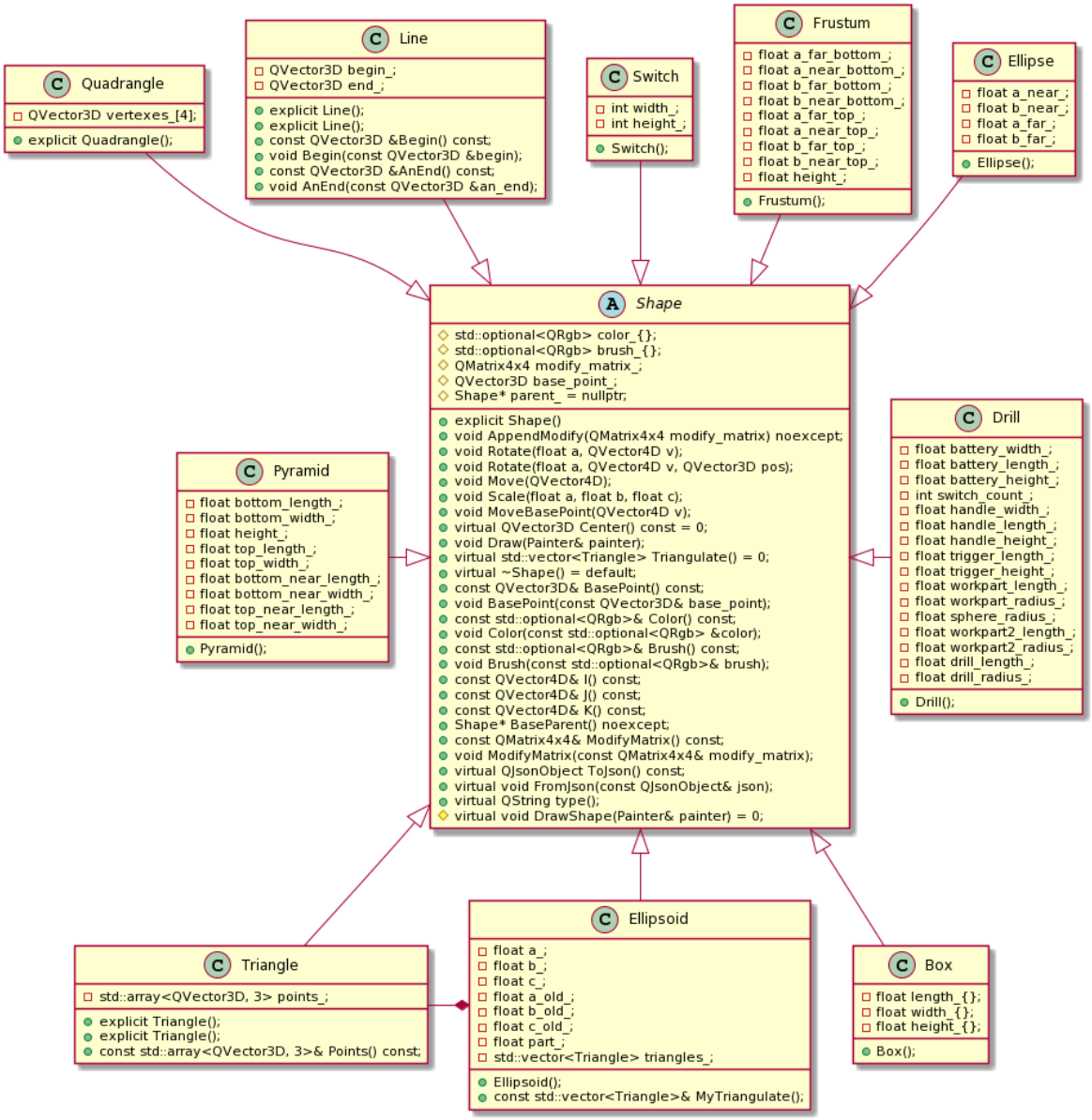


Рисунок 3.3 – Диаграмма классов для объектов

3.3 Реализация масштабирования

При масштабировании точки перемещаются на определенный коэффициент от начала координат. Если этот коэффициент больше 1, то точки отдаляются, если он меньше 1, то точки приближаются к началу координат.

Для того, чтобы предотвратить перемещение объекта при масштабировании в данном проекте перед масштабированием происходит перемещение объекта таким образом, чтобы его центр совпадал с началом координат. После масштабирования происходит обратный перенос, который

возвращает уже смасштабированный объект на свое место. Функция обеспечивающая масштабирование приведена на рисунке 3.3

```
void Shape::Scale(float a, float b, float c) {
  modify_matrix_ =
      modify_matrix_ *
      SGLMath::Move(Center()) *
      SGLMath::Scale(a, b, c) *
      SGLMath::Move(-Center());
}
```

Рисунок 3.4 – Функция масштабирования объекта

Параметры a, b, c – соответствуют масштабированию по осям x, y, z соответственно. Формирование матрицы происходит в методе SGLMath::Scale(), который приведен на рисунке 3.4.

```
QMatrix4x4 Scale(float a, float b, float c) {
  return QMatrix4x4(a, 0, 0, 0,
                    0, b, 0, 0,
                    0, 0, c, 0,
                    0, 0, 0, 1);
}
```

Рисунок 3.5 – Функция формирования матрицы масштабирования

3.4 Реализация поворота

В проекте реализован поворот объекта вокруг произвольной оси, проходящей через произвольную точку (рис. 3.5).

```
QMatrix4x4 Rotate(float a, QVector4D v, QVector3D pos) {
  return Move(pos) * Rotate(a, v) * Move(-pos);
}
```

Рисунок 3.6 – Функция расчета матрицы поворота с учетом смещения

Сначала применяется матрица смещения так, чтобы произвольная точка совпала с центром координат, затем производится поворот, и обратное

смещение в исходную точку. Матрица поворота вокруг произвольной оси в общем виде представлена на рисунке 3.6.

$$\begin{pmatrix} \cos\theta + (1-\cos\theta)x^2 & (1-\cos\theta)xy - (\sin\theta)z & (1-\cos\theta)xz + (\sin\theta)y \\ (1-\cos\theta)yx + (\sin\theta)z & \cos\theta + (1-\cos\theta)y^2 & (1-\cos\theta)yz - (\sin\theta)x \\ (1-\cos\theta)zx - (\sin\theta)y & (1-\cos\theta)zy + (\sin\theta)x & \cos\theta + (1-\cos\theta)z^2 \end{pmatrix}$$

Рисунок 3.7 – Матрица поворота вокруг произвольной оси в общем виде

Формирование матрицы поворота вокруг произвольной оси происходит в функции SGLMath::Rotate(), который приведен в рисунке 3.7.

```
QMatrix4x4 Rotate(float a, QVector4D v) {
  a = qDegreesToRadians(a);
  QMatrix4x4 rm;
  float x = v.x();
  float y = v.y();
  float z = v.z();
  QVector3D first_row{cos(a) + (1 - cos(a)) * x * x,
                      (1 - cos(a)) * x * y - sin(a) * z,
                      (1 - cos(a)) * x * z + sin(a) * y};
  QVector3D second_row{(1 - cos(a)) * y * x + sin(a) * z,
                       cos(a) + (1 - cos(a)) * y * y,
                       (1 - cos(a)) * y * z - sin(a) * x};
  QVector3D third_row{(1 - cos(a)) * z * x - sin(a) * y,
                      (1 - cos(a)) * z * y + sin(a) * x,
                      cos(a) + (1 - cos(a)) * z * z};
  rm.setRow(0, first_row);
  rm.setRow(1, second_row);
  rm.setRow(2, third_row);

  return rm;
}
```

Рисунок 3.7 – Функция формирования матрицы поворота

3.5 Реализация перемещения объекта

Перемещение объекта – это изменение всех его точек на определенное значение по координатным осям. Общая формула переноса точки на вектор представлена на рисунке 3.8.

$$= \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Рисунок 3.8 – Матрица переноса в общем виде

Формирование матрицы переноса производится в функции, которая показана на рисунке 3.9.

```
QMatrix4x4 Move(QVector4D v) {
  QMatrix4x4 move_matrix;
  move_matrix.setRow(0, {1, 0, 0, v.x()});
  move_matrix.setRow(1, {0, 1, 0, v.y()});
  move_matrix.setRow(2, {0, 0, 1, v.z()});
  move_matrix.setRow(3, {0, 0, 0, 1});
  return move_matrix;
}
```

Рисунок 3.9 – Функция формирование матрицы переноса

3.6 Реализация изменения параметров

Поскольку в курсовом проекте мы работаем с параметризованными объектами, была реализована функция изменения параметров объекта (рис. 3.10).

```
connect(ui->change_shape_2, &QPushButton::clicked, [this]() {
  for (auto[shape, brush] : ui->graphics_view->SelectedShapes()) {
    auto drill = dynamic_cast<Drill*>(shape);
    auto modify_matrix = drill->ModifyMatrix();
    *drill = Drill(ui->battery_width_sb->value(),
                   ui->battery_length_sb->value(),
                   ui->battery_height_sb->value(),
                   ui->switch_count_sb->value(),
                   ui->handle_width_sb->value(),
                   ui->handle_length_sb->value(),
                   ui->handle_height_sb->value(),
                   ui->trigger_length_sb->value(),
                   ui->trigger_height_sb->value(),
                   ui->workpart_length_sb->value(),
                   ui->workpart_radius_sb ->value(),
                   ui->sphere_radius_sb->value(),
                   ui->workpart2_length_sb->value(),
                   ui->workpart2_radius_sb->value(),
                   ui->drill_length_sb->value(),
                   ui->drill_radius_sb->value(),
                   drill->Color(),
                   drill->Brush(),
                   QVector3D(ui->shape_x_2->value(),
                             ui->shape_y_2->value(),
                             ui->shape_z_2->value()));
    drill->ModifyMatrix(modify_matrix);
  }
});
```

Рисунок 3.10 – Функция изменения параметров объекта

Эта функция считывает заданные в графическом интерфейсе поля (рис 3.11), и изменяет параметры объекта в соответствии с введенными значениями.

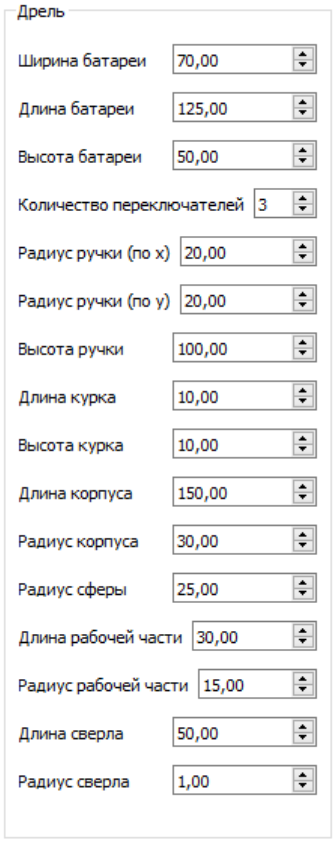


Рисунок 3.11 – Форма ввода параметров

Создание объекта с заданными параметрами происходит при помощи нажатия на кнопку «Добавить объект». Для модификации объектов предварительно необходимо выбрать один или несколько объектов, которые будут изменены после нажатия кнопки «Изменить выделенные объекты»

3.7 Реализация параллельной проекции

Параллельная проекция - это вид проекций, при котором сохраняются отношения сторон и их параллельность, однако углы между сторонами могут меняться. При параллельном проецировании матрица проекции является единичной. Функция, которая задает параллельный вид проецирования камеры приведена на рисунке 3.12.

```
void Camera::SetParallels() {
  projection_ = QMatrix4x4(1, 0, 0, 0,
                           0, 1, 0, 0,
                           0, 0, 1, 0,
                           0, 0, 0, 1);
  is_central = false;
}
```

Рисунок 3.12 – Функция переключения на параллельное проецирование

В процессе проецирования сначала происходит перевод трехмерных координат в двумерные, а затем перевод двумерных координат в координаты окна (рис. 3.13)

```
QVector3D project_point = projection_ * camera_point;
QMatrix4x4 window_matrix{
    1, 0, 0, width / 2,
    0, 1, 0, height / 2,
    0, 0, 1, 0,
    0, 0, 0, 1
};
QVector3D window_point = window_matrix * project_point;
```

Рисунок 3.13 – Перевод трехмерных точек в координаты окна

3.8 Реализация перспективной проекции

Перспективная проекция относится к центральным видам проекций, т.е. проекционные лучи направлены не параллельно друг другу, а сходятся в одной или нескольких точках. В данном проекте перспективная проекция имеет одну точку схода. Функция, которая задает центральный вид проецирования камеры приведена на рисунке 3.14.

```
void Camera::SetCentral() {
  const float a = far_ / (far_ - near_);
  const float b = far_ * near_ / (near_ - far_);
  projection_ = QMatrix4x4(near_, 0    , 0, 0,
                           0    , near_, 0, 0,
                           0    , 0    , a, b,
                           0    , 0    , 1, 0);
  is_central = true;
}
```

Рисунок 3.14 - Функция переключения на центральное проецирование

В процессе проецирования сначала происходит перевод трехмерных координат в двумерные, а затем перевод двумерных координат в координаты окна (рис. 3.15)

```
QVector3D project_point = projection_ * camera_point;
QMatrix4x4 window_matrix{
    width / 2, 0, 0, width / 2,
    0, height / 2, 0, height / 2,
    0, 0, 1, 0,
    0, 0, 0, 1
};
if (qAbs(project_point.x()) >= 1 ||
    qAbs(project_point.y()) >= 1) {
  throw std::runtime_error{"Bad point"};
}
result = window_matrix * project_point;
```

Рисунок 3.15 – Перевод трехмерных точек в координаты окна

3.9 Реализация произвольных видовых преобразований (камеры)

Для расчета матрицы, позволяющей перейти из глобальной системы координат в локальную систему координат камеры используется функция, представленная на рисунке 3.16.

```
void LookAtCamera::CalculateMatrix() {
  model_view_.setToIdentity();
  QVector3D z = (center_ - eye_).normalized();
  QVector3D x = QVector3D::crossProduct(up_, z).normalized();
  QVector3D y = QVector3D::crossProduct(z, x).normalized();
  QMatrix4x4 tr;
  model_view_.setRow(0, x);
  model_view_.setRow(1, y);
  model_view_.setRow(2, z);
  QVector4D last_row{-eye_, 1};
  tr.setColumn(3, last_row);
  model_view_ *= tr;
  if (is_central) {
    SetCentral();
  } else {
    SetParallels();
  }
}
```

Рисунок 3.16 – Функция расчета матрицы для камеры

Перемещение «свободной» камеры (рис. 3.17).

```
if (buttons_[Qt::Key_A]) {
  eye = eye - STEP * (x).normalized();
  center = center - STEP * (x).normalized();
}
if (buttons_[Qt::Key_W]) {
  eye = eye + STEP * (center - eye).normalized();
  center = center + STEP * (center - eye).normalized();
}
if (buttons_[Qt::Key_S]) {
  eye = eye - STEP * (center - eye).normalized();
  center = center - STEP * (center - eye).normalized();
}
if (buttons_[Qt::Key_D]) {
  eye = eye + STEP * (x).normalized();
  center = center + STEP * (x).normalized();
}
if (buttons_[Qt::Key_Z]) {
  eye = eye - STEP * (y).normalized();
  center = center - STEP * (y).normalized();
}
if (buttons_[Qt::Key_X]) {
  eye = eye + STEP * (y).normalized();
  center = center + STEP * (y).normalized();
}
```

Рисунок 3.17 – Перемещение «свободной» камеры

Вращение «свободной» камеры (рис. 3.18).

```
if (buttons_[Qt::Key_Down]) {
  rotate = SGLMath::Rotate(ANGLE_STEP, x, eye);
  center = rotate * center;
  up = SGLMath::Rotate(ANGLE_STEP, x) * up;
}
if (buttons_[Qt::Key_Up]) {
  rotate = SGLMath::Rotate(-ANGLE_STEP, x, eye);
  center = rotate * center;
  up = SGLMath::Rotate(-ANGLE_STEP, x) * up;
}
if (buttons_[Qt::Key_Left]) {
  rotate = SGLMath::Rotate(-ANGLE_STEP, y, eye);
  center = rotate * center;
  up = SGLMath::Rotate(-ANGLE_STEP, y) * up;
}
if (buttons_[Qt::Key_Right]) {
  rotate = SGLMath::Rotate(ANGLE_STEP, y, eye);
  center = rotate * center;
  up = SGLMath::Rotate(ANGLE_STEP, y) * up;
}
if (buttons_[Qt::Key_Q]) {
  rotate = SGLMath::Rotate(ANGLE_STEP, z, eye);
  center = rotate * center;
  up = SGLMath::Rotate(ANGLE_STEP, z) * up;
}
if (buttons_[Qt::Key_E]) {
  rotate = SGLMath::Rotate(-ANGLE_STEP, z, eye);
  center = rotate * center;
  up = SGLMath::Rotate(-ANGLE_STEP, z) * up;
}
```

Рисунок 3.18 - Вращение «свободной» камеры

Для демонстрации камеры, которая смотрит в одну точку был реализован функция панорамы, в которой камера вращается вокруг заданной точки (рис. 3.19).

```
void GraphicsView::PanoramStep(QVector3D center) {
  auto camera = dynamic_cast<LookAtCamera*>(camera_);
  assert(camera);
  auto eye = SGLMath::Rotate(1, {0, 0, 1, 0},
                             center) * camera->Eye();

  camera->Reset(eye, center, {0, 0, 1});
}
```

Рисунок 3.19 – Вращение камеры вокруг точки

3.10 Реализация сохранения сцены

Для сохранения состояния сцены в файл программа проходит по списку всех объектов сцены, и преобразовывает каждый из них в json формат (рис

3.10). Затем в файл записывается положение камеры и все объекты, находящиеся на сцене. (рис. 3.11)

```
QJsonObject Drill::ToJson() const {
  QJsonObject json;
  json["Тип"] = "Дрель";
  json["Shape"] = Shape::ToJson();
  json["Ширина батареи"] = battery_width_;
  json["Длина батареи"] = battery_length_;
  json["Высота батареи"] = battery_height_;
  json["Количество переключателей"] = switch_count_;
  json["Радиус ручки (по x)"] = handle_width_;
  json["Радиус ручки (по y)"] = handle_length_;
  json["Высота ручки"] = handle_height_;
  json["Длина курка"] = trigger_length_;
  json["Высота курка"] = trigger_height_;
  json["Радиус корпуса"] = workpart_radius_;
  json["Длина корпуса"] = workpart_length_;
  json["Радиус сферы"] = sphere_radius_;
  json["Радиус рабочей части"] = workpart2_radius_;
  json["Длина рабочей части"] = workpart2_length_;
  json["Длина сверла"] = drill_length_;
  json["Радиус сверла"] = drill_radius_;
  return json;
}
```

Рисунок 3.20 – Функция преобразования объекта в json

```
connect(ui->save_model, &QPushButton::clicked, [this]() {
  auto filename = QFileDialog::getSaveFileName(nullptr,
                                  "Выберите файл для сохранения");
  if (filename.isNull()) {
    return;
  }
  ui->graphics_view->UnselectedAllShape();
  QJsonDocument document ;
  QJsonObject json;
  QJsonObject json_camera;
  auto camera = dynamic_cast<LookAtCamera*>(ui->graphics_view->Camera());
  json_camera["Глаз"] = SGLMath::ToJson(camera->Eye());
  json_camera["Центр"] = SGLMath::ToJson(camera->Center());
  json_camera["Up"] = SGLMath::ToJson(camera->Up());
  QJsonArray json_shapes;
  for (Shape* shape : scene_.Shapes()) {
    json_shapes.push_back(shape->ToJson());
  }
  json["Камера"] = json_camera;
  json["Фигуры"] = json_shapes;
  document.setObject(json);
  SaveJson(document, filename);
});
```

Рисунок 3.21 – Функция сохранения сцены в файл

Для загрузки сцены из файла используется обратная операция. Сначала файл считывает параметры камеры (рис. 3.12), а затем параметры объектов, преобразуя их в объекты на сцене (рис. 3.13).

```
connect(ui->load_model, &QPushButton::clicked, [this]() {
  auto filename = QFileDialog::getOpenFileName(nullptr,
                                               "Выберите файл с фигурами");
  if (filename.isNull()) {
    return;
  }
  QJsonDocument document = LoadJson(filename);
  QJsonObject json = document.object();
  auto camera = dynamic_cast<LookAtCamera*>(ui->graphics_view->Camera());
  assert(camera);
  QVector4D eye = SGLMath::ToVector4D(json["Камера"]
      .toObject()["Глаз"].toArray());
  QVector4D center = SGLMath::ToVector4D(json["Камера"]
      .toObject()["Центр"].toArray());
  QVector4D up = SGLMath::ToVector4D(json["Камера"]
                                         .toObject()["Up"].toArray());
  camera->Reset(QVector3D(eye), QVector3D(center), QVector3D(up));

  QJsonArray json_shapes = json["Фигуры"].toArray();
  scene_.Clear();
  for (QJsonValue value : json_shapes) {    Missing reference in range-fo
    auto drill = new Drill;
    drill->FromJson(value.toObject());
    scene_.PushShape(drill);
  }
});
```

Рисунок 3.22 – Функция загрузки сцены

```
void Drill::FromJson(const QJsonObject& json) {
  Shape::FromJson(json["Shape"].toObject());
  battery_width_ = json["Ширина батареи"].toDouble();
  battery_length_ = json["Длина батареи"].toDouble();
   battery_height_ = json["Высота батареи"].toDouble();
   switch_count_ = json["Количество переключателей"].toInt();
   handle_width_ = json["Радиус ручки (по x)"].toDouble();
   handle_length_ = json["Радиус ручки (по y)"].toDouble();
   handle_height_ = json["Высота ручки"].toDouble();
   trigger_length_ = json["Длина курка"].toDouble();
   trigger_height_ = json["Высота курка"].toDouble();
   workpart_radius_ = json["Радиус корпуса"].toDouble();
   workpart_length_ = json["Длина корпуса"].toDouble();
   sphere_radius_ = json["Радиус сферы"].toDouble();
   workpart2_radius_ = json["Радиус рабочей части"].toDouble();
   workpart2_length_  = json["Длина рабочей части"].toDouble();
   drill_length_ = json["Длина сверла"].toDouble();
   drill_radius_ = json["Радиус сверла"].toDouble();
}
```

Рисунок 3.23 – Функция преобразования объекта из json

3.11 Реализация удаления скрытых линий

Для реализации закраски объекта и удаления скрытых линий в программе используется алгоритм z-буфера. Он определяет цвет пикселя в зависимости от отдаленности объекта от камеры и относительного положения объекта относительно остальных объектов (рис.3.24).

```
void Painter::DrawPixel(QVector3D point, QRgb color, Shape* parent) {
  QPoint p = point.toPoint();
  int i = (p.x());
  int j = (p.y());
  try {
    LookAtCamera &camera = dynamic_cast<LookAtCamera&>(camera_); // t
    if (zbuffer_[i][j].z >= point.z()) {
      zbuffer_[i][j].z = point.z();
      zbuffer_[i][j].parent = parent;
      SetPixel(i, j, color);
    }
  } catch(...) {

  }
}
```

Рисунок 3.24 – Реализация алгоритма z-буфера

4 ТЕСТИРОВАНИЕ ПРОГРАММЫ

Разработанная программная система отвечает всем требованиям технического задания, предоставляет пользователю возможность работы с полигональной моделью «Дрель», возможность параметризации объекта, поворота, перемещения, масштабирования, удаления и видовых преобразований. Программа имеет интуитивно понятный пользовательский интерфейс на русском языке.

При разработке пользовательского интерфейса был сделан упор на предотвращение ошибок, которые могут произойти со стороны пользователя. Например, если пользователь попытается удалить или модифицировать объекты, при этом не один объект не будет выбран, то ничего не произойдет. Если пользователь попытается ввести слишком большие значения параметров, то у него ничего не выйдет, т.к. поля для ввода числовых значений параметра имеют максимальное и минимальное значения.

ВЫВОДЫ

При выполнении курсового проекта было выполнено планирование собственного графического редактора для работы с определенными объектами.

В результате проекта был спроектирован графический редактор, который имеет такие функции как: возможность работы с несколькими объектами, добавление и удаление объектов, модификация объектов, поворот, перемещение, масштабирование объектов, перемещение камеры, параллельное и центральное проецирование, проволочный и полигональный вид камеры. Также графический редактор обладает модифицированным Z-буфером, который дополнительно хранит указатель на объект, которому принадлежит пиксель.

При продолжении работы над проектом в будущем можно будет получить намного большую функциональность. Например, добавление различных материалов, улучшение освещения, добавление теней, добавление анимаций.

ПЕРЕЧЕНЬ ССЫЛОК

1. Вольхин К. А. Основы компьютерной графики : электронное учебное пособие для студентов [Электронный ресурс] / К. А. Вольхин ;Робачевский А. М. Операционная система UNIX. — СПб.: БХВ–Петербург, 2002. — 528 с.
2. Проецирование трехмерных объектов [Электронный ресурс] // Проецирование трехмерных объектов. – режим доступа: http://astro.tsu.ru/KGaG/text/5_1.html
3. Ламот, Андре. Программирование трехмерных игр для Windows. Советы профессионала по трехмерной графике и растеризации.: Пер. с англ. – М.: Издательский дом «Вильямс», 2004 – С.1424.
4. Сиденко Л.А. Компьютерная графика и геометрическое моделирование: Учебное пособие. – Спб.: Питер, 2009. – С. 224

ПРИЛОЖЕНИЕ А. ТЕХНИЧЕСКОЕ ЗАДАНИЕ

ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
КАФЕДРА «КОМПЬЮТЕРНОЕ МОДЕЛИРОВАНИЕ И ДИЗАЙН»

Дисциплина «Архитектура и проектирование графических систем».
Специальность «Программная инженерия»
Курс 3 Группа ПИ18 Б

ТЕХНИЧЕСКОЕ ЗАДАНИЕ
к курсовому проекту
по курсу «Графическое и геометрическое моделирование»
Моргунов Арсений Геннадьевич

ТЕМА ПРОЕКТА: Разработка графического редактора для работы с параметризованными трёхмерными объектами

СРОК СДАЧИ:

ЗАДАНИЕ: Создать графический редактор для работы с трёхмерным объектом, изображённым на рисунке А.1

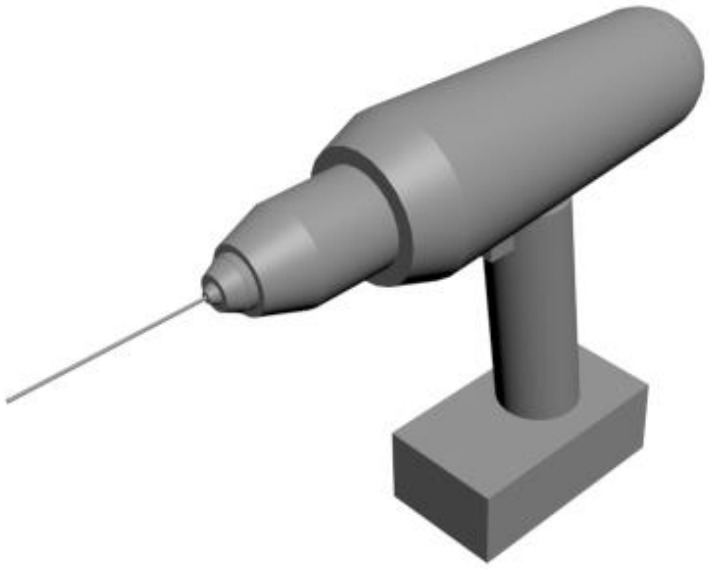


Рисунок А.1- Электрическая дрель

Объект задается следующими параметрами:

1 – ширина ручки;

2 – длина сверла;

3 – ширина батареи;

4 – длина батареи;

5 – высота батареи;

6 – Длина корпуса

7 – Радиус корпуса

8 – Радиус сверла

ww9 – Радиус рабочей части

10 – Длина рабочей части

11 – Длина курка

12 – Высота курка

13 – Радиус сферы

ТРЕБОВАНИЯ К ГРАФИЧЕСКОМУ РЕДАКТОРУ

1 Наличие графической базы данных:

Возможность сохранения сцены с объектами в файле.

1.1 Читабельность базы данных:

Файл сцены должен содержать данные модели в текстовом виде.

1.2 Возможность работы с несколькими объектами:

1.3 Обеспечить добавление на экран допустимого количества объектов, а также работу со всеми объектами (перемещение, панорамирование) и одним выбранным объектом

2 Обеспечить редактирование и параметризацию объектов:

Возможность изменения параметров любого объекта, а также его масштабирование, перенос, поворот и удаление

3 Обеспечить центральное и параллельное проецирование:

Возможность переключения с одного вида проецирования на другой

4 Задание всех параметров аппарата проецирования:

Обеспечить наличие “камеры”, задаваемой необходимыми параметрами (как минимум – точка зрения и точка цели), также возможность её перемещения вокруг объекта и поворота вокруг своей оси

5 Удаление невидимых частей объектов:

Обеспечить визуализацию объекта без его невидимых частей

при помощи алгоритма удаления невидимых линий

6 Разработать интуитивно понятный пользовательский интерфейс:

Программный продукт должен обеспечить пользователю максимально понятную и простую работу в редакторе за счёт оформления интерфейса, контекстных подсказок, горячих клавиш и предупреждений

7 При разработке графического редактора не использовать стандартные графические библиотеки.(Open GL, Direct X и т.п.)

СОДЕРЖАНИЕ ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ

- Разработка полигональной модели объекта
- Описание выбранных методов и алгоритмов визуализации
- Разработка структур данных для хранения описания объекта
- Программная реализация графического редактора
- Пример выполнения программы, иллюстрированный экранными формами

ДАТА ВЫДАЧИ ЗАДАНИЯ: __.__.2021__

Задание принял: студент группы ПИ18 Моргунов А. Г.

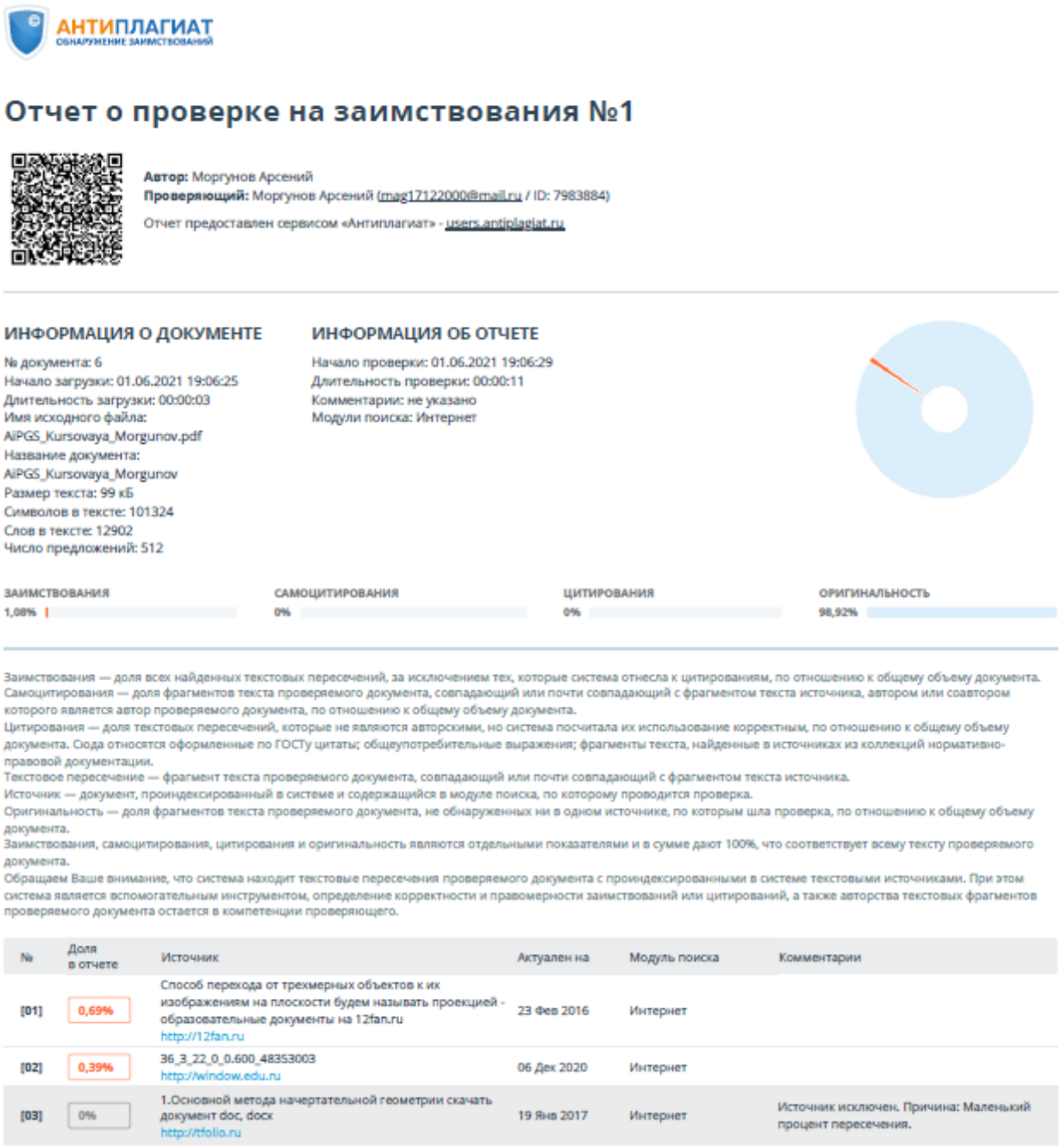
Руководители проекта доценты каф.КМД:

Карабчевский Виталий Владиславович_____

Боднар Алина Валерьевна _____

Доценко Георгий Васильевич _____

ПРИЛОЖЕНИЕ Б. ПРОВЕРКА ОРИГИНАЛЬНОСТИ



АНТИПЛАГИАТ
ОБНАРУЖЕНИЕ ЗАИМСТВОВАНИЙ

Отчет о проверке на заимствования №1

Автор: Моргунов Арсений
Проверяющий: Моргунов Арсений (mag17122000@mail.ru / ID: 7983884)

Отчет предоставлен сервисом «Антиплагиат» - users.antiplagiat.ru

ИНФОРМАЦИЯ О ДОКУМЕНТЕ

№ документа: 6
Начало загрузки: 01.06.2021 19:06:25
Длительность загрузки: 00:00:03
Имя исходного файла:
AiPGS_Kursovaya_Morgunov.pdf
Название документа:
AiPGS_Kursovaya_Morgunov
Размер текста: 99 кБ
Символов в тексте: 101324
Слов в тексте: 12902
Число предложений: 512

ИНФОРМАЦИЯ ОБ ОТЧЕТЕ

Начало проверки: 01.06.2021 19:06:29
Длительность проверки: 00:00:11
Комментарии: не указано
Модули поиска: Интернет

ЗАИМСТВОВАНИЯ	САМОЦИТИРОВАНИЯ	ЦИТИРОВАНИЯ	ОРИГИНАЛЬНОСТЬ
1,08%	0%	0%	98,92%

Заимствования — доля всех найденных текстовых пересечений, за исключением тех, которые система отнесла к цитированиям, по отношению к общему объему документа.
Самоцитирования — доля фрагментов текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника, автором или соавтором которого является автор проверяемого документа, по отношению к общему объему документа.
Цитирования — доля текстовых пересечений, которые не являются авторскими, но система посчитала их использование корректным, по отношению к общему объему документа. Сюда относятся оформленные по ГОСТу цитаты; общеупотребительные выражения; фрагменты текста, найденные в источниках из коллекций нормативно-правовой документации.
Текстовое пересечение — фрагмент текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника.
Источник — документ, проиндексированный в системе и содержащийся в модуле поиска, по которому проводится проверка.
Оригинальность — доля фрагментов текста проверяемого документа, не обнаруженных ни в одном источнике, по которым шла проверка, по отношению к общему объему документа.
Заимствования, самоцитирования, цитирования и оригинальность являются отдельными показателями и в сумме дают 100%, что соответствует всему тексту проверяемого документа.
Обращаем Ваше внимание, что система находит текстовые пересечения проверяемого документа с проиндексированными в системе текстовыми источниками. При этом система является вспомогательным инструментом, определение корректности и правомерности заимствований или цитирований, а также авторства текстовых фрагментов проверяемого документа остается в компетенции проверяющего.

№	Доля в отчете	Источник	Актуален на	Модуль поиска	Комментарии
[01]	0,69%	Способ перехода от трехмерных объектов к их изображениям на плоскости будем называть проекцией - образовательные документы на 12fan.ru http://12fan.ru	23 Фев 2016	Интернет	
[02]	0,39%	36_3_22_0_0.600_48353003 http://window.edu.ru	06 Дек 2020	Интернет	
[03]	0%	1.Основной метода начертательной геометрии скачать документ doc, docx http://tfolio.ru	19 Янв 2017	Интернет	Источник исключен. Причина: Маленький процент пересечения.

Рисунок Б1 – Проверка на оригинальность

ПРИЛОЖЕНИЕ В. ЭКРАННЫЕ ФОРМЫ

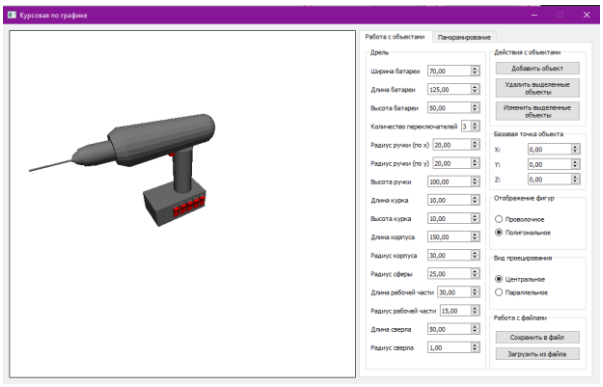


Рисунок В.1 – Полигональная модель с центральным проецированием

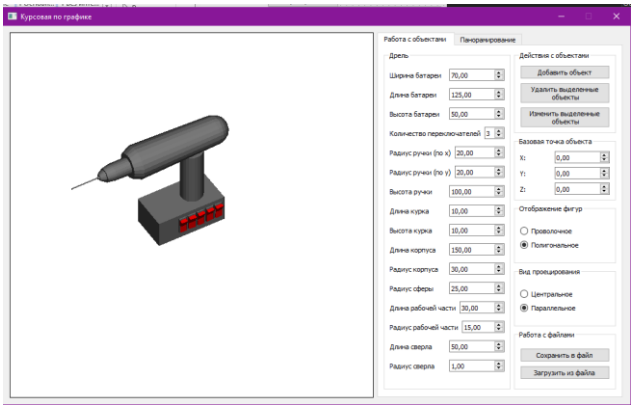


Рисунок В.2 – Полигональная модель с параллельным проецированием

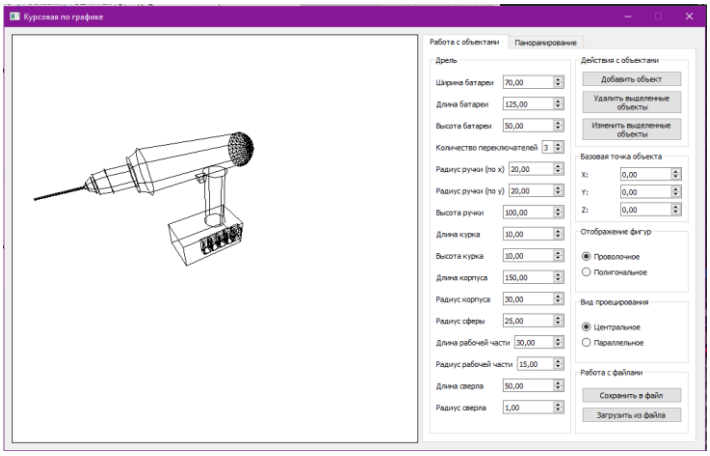


Рисунок В.3 – Проволочная модель

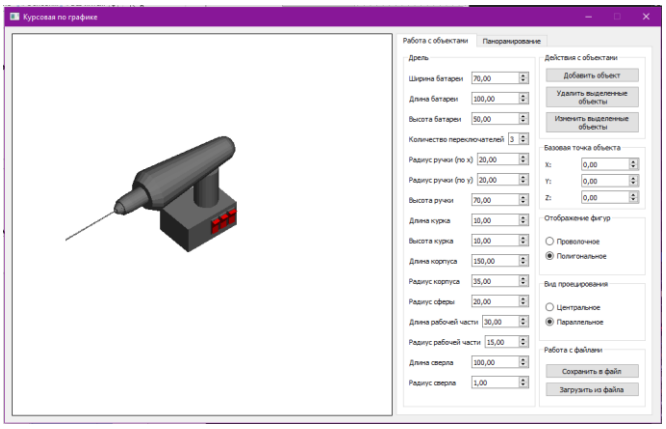


Рисунок В.4 – Изменение параметров объекта

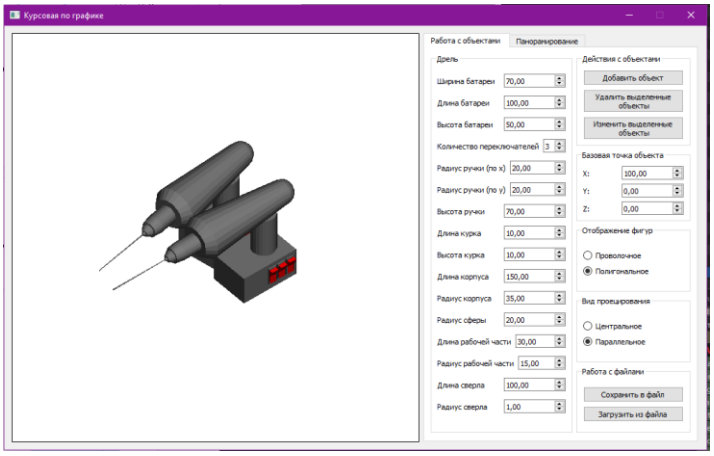


Рисунок В.5 – Создание нового объекта

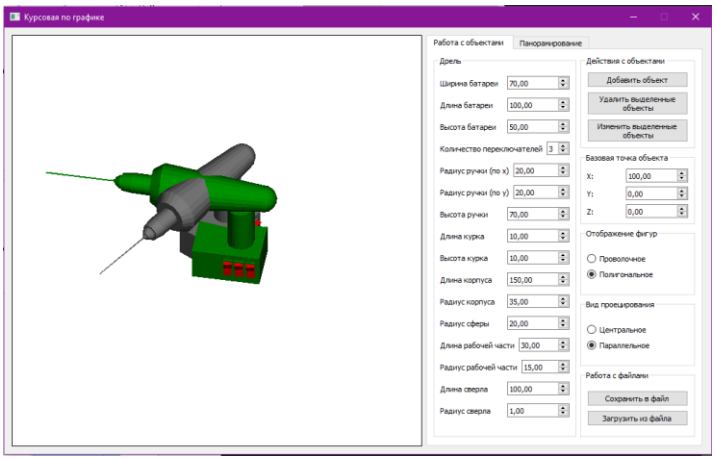


Рисунок В.6 – Поворот объекта

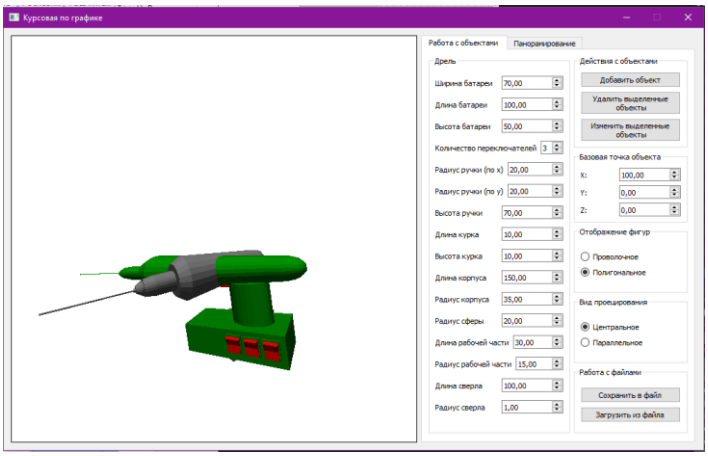


Рисунок В.7 – Изменение положения камеры

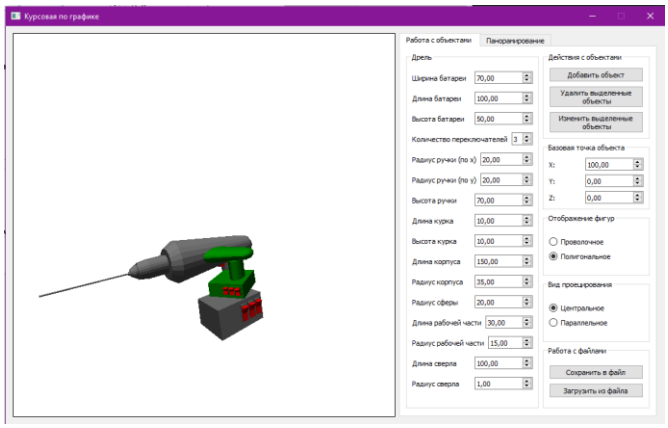


Рисунок В.8 – Масштабирование объектов

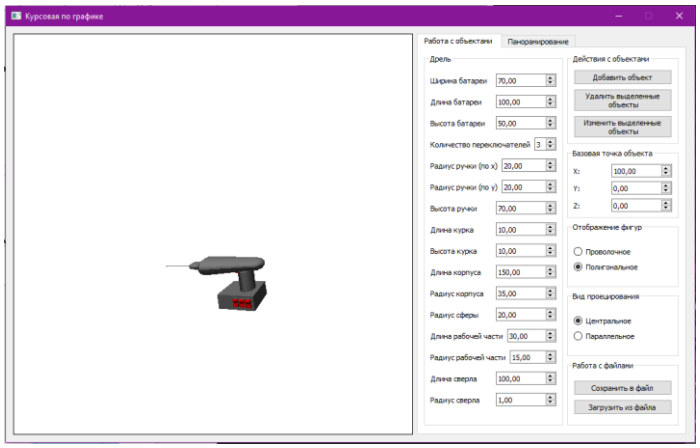


Рисунок В.9 – Удаление объекта

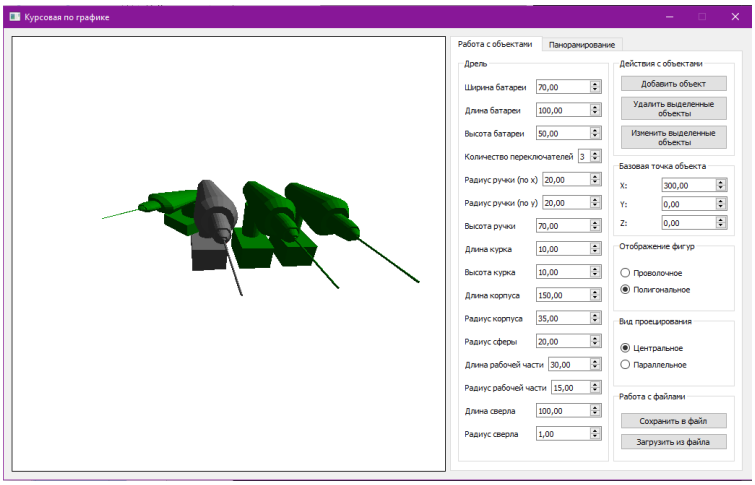


Рисунок В.10 – Выделение объектов

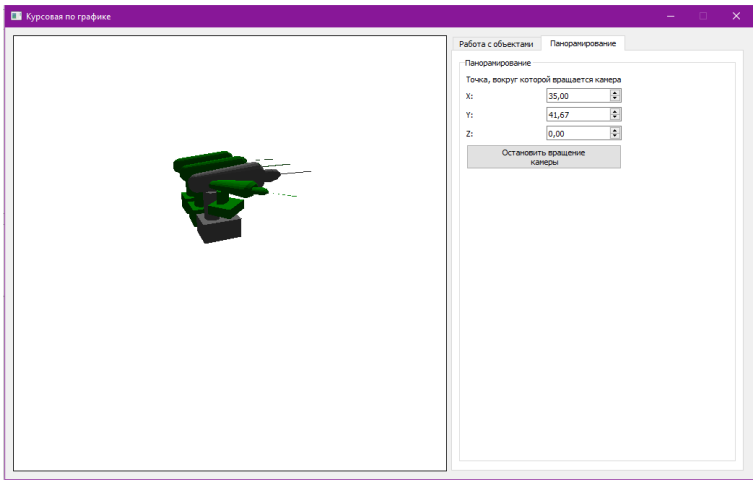


Рисунок В.11 – Панорамирование

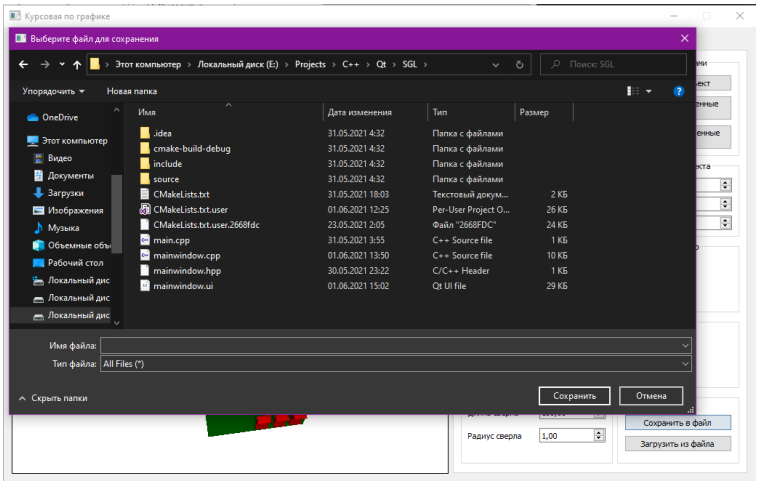


Рисунок В.12 – Сохранение в файл

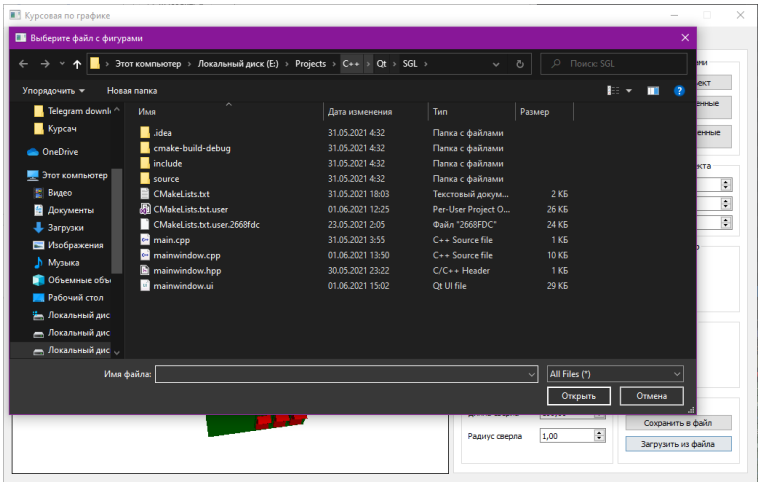


Рисунок В.13 – Загрузка из файла

ПРИЛОЖЕНИЕ Г. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

В разработанной программе управление камерой производится нажатием на клавиши клавиатуры. Для того, чтобы переместиться вперед нужно нажать W, назад – S, влево – A, вправо – D, чтобы повернуть камеру необходимо нажать стрелку в том направлении в котором необходимо повернуться. Поворот камеры по часовой и против часовой стрелки осуществляется клавишами E и Q соответственно. Чтобы подняться вверх нужно нажать X, а для спуска вниз – Z.

Масштабирование осуществляется клавишами R, F, T, G, Y, H. Вращение объектов происходит при помощи клавиш U, J, I, K, O, L.

ПРИЛОЖЕНИЕ Д. ЛИСТИНГ

Camera.hpp

```
#pragma once
#include <QVector3D>
#include <QMatrix4x4>

class Camera {
 public:
  virtual QVector3D Project(QVector3D point) = 0;
  virtual ~Camera() = default;
  void SetCentral();
  void SetParallels();

  float near_{1};
  float far_{10000};
 protected:
  QMatrix4x4 projection_;
  bool is_central = true;
};
```

GraphicsScene.hpp

```
#pragma once
#include <QObject>
#include "Shape.hpp"
#include <mutex>

class GraphicsScene : public QObject {
  Q_OBJECT

 public:
  void PushShape(Shape* shape);
  void EraseShape(Shape* shape);
  void Clear();
  [[nodiscard]] const std::vector<Shape *>&
Shapes() const;

  /*!
   * @brief delete for each shape in shapes_
   * for (Shape* shape : shapes_) {
   *   delete shape;
   * }
   */
  ~GraphicsScene() override;

 private:
  std::vector<Shape*> shapes_;
  std::mutex m_;
};
```

GraphicsView.hpp

```
#pragma once
#include <QtWidgets>
#include <QtGui>
#include "Painter.hpp"
#include "GraphicsView.hpp"
#include "GraphicsScene.hpp"
#include "ZBuffer.hpp"
#include "Camera.hpp"
#include <unordered_map>
#include "LookAtCamera.hpp"


class GraphicsView : public QLabel {
Q_OBJECT

 public:
  [[maybe_unused]] explicit GraphicsView(QWidget*
parent = nullptr,
                         GraphicsScene* scene =
nullptr);
  [[nodiscard]] GraphicsScene *Scene() const;
  void Scene(GraphicsScene *scene);
  void Render();
  void SetCentral();
  void SetParallels();
  void SetPolygonView();
  void SetLineView();
  void PanoramStep(QVector3D);
  [[nodiscard]] const
std::unordered_map<::Shape*,
                           std::optional<QRgb>>&
SelectedShapes() const;
  void ClearSelected();
  Camera* Camera() const;
  void Camera(class Camera* camera);
  void UnselectedAllShape();

 protected:
  void keyPressEvent(QKeyEvent*) override;
  void keyReleaseEvent(QKeyEvent*) override;
  void mousePressEvent(QMouseEvent*) override;
  void resizeEvent(QResizeEvent* event) override;
  void timerEvent(QTimerEvent*) override;


 private:
  QImage image_;
  ZBuffer zbuffer_;
  class Camera* camera_;
  GraphicsScene* scene_;
  bool is_free_camera_{true};
  std::unordered_map<::Shape*,
std::optional<QRgb>> selected_shapes;

  std::map<int, bool> buttons_;
  void FreeCameraChange();

  SGL::ViewType view_type_;
};
```

LookAtCamera.hpp

```
#pragma once
#include "Camera.hpp"
#include <QMatrix4x4>


class LookAtCamera : public Camera {
 public:
  LookAtCamera(QVector3D eye = {}, QVector3D
center = {}, QVector3D up = {});
  QVector3D Project(QVector3D point) override;
  LookAtCamera& operator=(const LookAtCamera&
rhs);
  void Reset(QVector3D eye, QVector3D center,
QVector3D up);
  const QVector3D &Eye() const;
  const QVector3D &Center() const;
  const QVector3D &Up() const;
  float Near() const;
  float Far() const;
  const QSize& WindowSize() const;
  void WindowSize(const QSize& window_size);


 private:
  QVector3D eye_;
  QVector3D center_;
  QVector3D up_;
  QMatrix4x4 model_view_;
  QSize window_size_;
  // bool central_parallels;
```

```
  /// \brief CalculateMatrix model_view_ with
current eye, center and up vector
  void CalculateMatrix();
};
```

Painter.hpp

```
#pragma once
#include <QtGui>
#include "Structs.hpp"
#include "ZBuffer.hpp"
#include <optional>
#include "Shape.hpp"
#include "Camera.hpp"

class Painter {
 public:

  explicit Painter(QImage& image, ZBuffer&
zbuffer, Camera& camera,
                   SGL::ViewType view_type);
  void SetPixel(int x, int y, QRgb color);
  [[nodiscard]] SGL::Size ImageSize() const;


  /// \brief DrawPixel with color is color_
  void DrawPoint(QVector3D point, Shape* shape =
nullptr);

  /// \brief DrawPixel with color is brush_
  void DrawBrush(QVector3D point, Shape* shape =
nullptr);

  /// \brief SetPixel with check zbuffer
  void DrawPixel(QVector3D point, QRgb color,
Shape* parent = nullptr);

  [[nodiscard]] std::optional<QRgb> Color()
const;
  void Color(std::optional<QRgb> color);
  [[nodiscard]] std::optional<QRgb> Brush()
const;
  void Brush(std::optional<QRgb> brash);

  void Save();
  void Restore();
  void AppendTransform(QMatrix4x4 transform);
  void DrawLine(QVector3D begin, QVector3D end,
Shape* parent);
  void DrawTriangle(std::array<QVector3D, 3>
points, Shape* parent);

 private:
  std::optional<QRgb> color_ = std::nullopt;
  std::optional<QRgb> brush_ = std::nullopt;
  QMatrix4x4 transform_;
  QImage& image_;
  ZBuffer& zbuffer_;
  Camera& camera_;
  std::optional<QRgb> save_colors[2];
  QMatrix4x4 save_transform_;
  SGL::ViewType view_type_;

  friend class Triangle;
  friend class Line;
};
```

SGLMath.hpp

```
#pragma once
#include <QtGui>
#include "Structs.hpp"
#include "ZBuffer.hpp"
#include <optional>
#include "Shape.hpp"
#include "Camera.hpp"

class Painter {
 public:

  explicit Painter(QImage& image, ZBuffer&
zbuffer, Camera& camera,
                   SGL::ViewType view_type);
  void SetPixel(int x, int y, QRgb color);
  [[nodiscard]] SGL::Size ImageSize() const;


  /// \brief DrawPixel with color is color_
  void DrawPoint(QVector3D point, Shape* shape =
nullptr);

  /// \brief DrawPixel with color is brush_
  void DrawBrush(QVector3D point, Shape* shape =
nullptr);

  /// \brief SetPixel with check zbuffer
  void DrawPixel(QVector3D point, QRgb color,
Shape* parent = nullptr);

  [[nodiscard]] std::optional<QRgb> Color()
const;
  void Color(std::optional<QRgb> color);
  [[nodiscard]] std::optional<QRgb> Brush()
const;
  void Brush(std::optional<QRgb> brash);

  void Save();
  void Restore();
  void AppendTransform(QMatrix4x4 transform);
  void DrawLine(QVector3D begin, QVector3D end,
Shape* parent);
  void DrawTriangle(std::array<QVector3D, 3>
points, Shape* parent);

 private:
  std::optional<QRgb> color_ = std::nullopt;
  std::optional<QRgb> brush_ = std::nullopt;
  QMatrix4x4 transform_;
  QImage& image_;
  ZBuffer& zbuffer_;
  Camera& camera_;
  std::optional<QRgb> save_colors[2];
  QMatrix4x4 save_transform_;
  SGL::ViewType view_type_;

  friend class Triangle;
  friend class Line;
};
```

Structs.hpp

```
#pragma once
#include <QtGui>
#include "Structs.hpp"
#include "ZBuffer.hpp"
#include <optional>
#include "Shape.hpp"
#include "Camera.hpp"

class Painter {
 public:

  explicit Painter(QImage& image, ZBuffer&
zbuffer, Camera& camera,
                   SGL::ViewType view_type);
  void SetPixel(int x, int y, QRgb color);
  [[nodiscard]] SGL::Size ImageSize() const;


  /// \brief DrawPixel with color is color_
  void DrawPoint(QVector3D point, Shape* shape =
nullptr);

  /// \brief DrawPixel with color is brush_
  void DrawBrush(QVector3D point, Shape* shape =
nullptr);
```

```
  /// \brief SetPixel with check zbuffer
  void DrawPixel(QVector3D point, QRgb color,
Shape* parent = nullptr);

  [[nodiscard]] std::optional<QRgb> Color()
const;
  void Color(std::optional<QRgb> color);
  [[nodiscard]] std::optional<QRgb> Brush()
const;
  void Brush(std::optional<QRgb> brash);

  void Save();
  void Restore();
  void AppendTransform(QMatrix4x4 transform);
  void DrawLine(QVector3D begin, QVector3D end,
Shape* parent);
  void DrawTriangle(std::array<QVector3D, 3>
points, Shape* parent);

 private:
  std::optional<QRgb> color_ = std::nullopt;
  std::optional<QRgb> brush_ = std::nullopt;
  QMatrix4x4 transform_;
  QImage& image_;
  ZBuffer& zbuffer_;
  Camera& camera_;
  std::optional<QRgb> save_colors[2];
  QMatrix4x4 save_transform_;
  SGL::ViewType view_type_;

  friend class Triangle;
  friend class Line;
};
```

ZBuffer.hpp

```
#pragma once
#include "Structs.hpp"
#include "Shape.hpp"
#include <QSize>

/*!
 * @brief Keep buffer for z coordinates
 */
class ZBuffer {
 public:
  explicit ZBuffer(QSize size);
  void Resize(QSize size);

  struct Point {
    float z{};
    Shape* parent{nullptr};
  };

  /*!
   * @brief For access to buffer which is storage
   * @param i Is index
   * @return return storage + i * width
   */
  Point* operator[](int i) noexcept;
  void Clear() noexcept;
  ~ZBuffer();

 private:
  Point* storage_;
  QSize size_;
 public:
  const QSize& Size() const;
};
```

Camera.cpp

```
#include "Camera.hpp"


void Camera::SetCentral() {
  const float a = far_ / (far_ - near_);
  const float b = far_ * near_ / (near_ - far_);
  projection_ = QMatrix4x4(near_, 0    , 0, 0,
                           0    , near_, 0, 0,
                           0    , 0    , a, b,
                           0    , 0    , 1, 0);
  is_central = true;
}


void Camera::SetParallels() {
  projection_ = QMatrix4x4(1, 0, 0, 0,
                           0, 1, 0, 0,
                           0, 0, 1, 0,
                           0, 0, 0, 1);
  is_central = false;
}
```

GraphicsScene.cpp

```
#include "GraphicsScene.hpp"
#include <algorithm>


GraphicsScene::~GraphicsScene() {
  for (Shape* shape : shapes_) {
    delete shape;
  }
}


void GraphicsScene::PushShape(Shape* shape) {
  std::lock_guard<std::mutex> lock(m_);
  auto it = std::find(shapes_.begin(),
shapes_.end(), shape);
  if (it != shapes_.end()) {
    throw std::runtime_error("Push shape twist");
  }
  shapes_.push_back(shape);
}


const std::vector<Shape*>&
GraphicsScene::Shapes() const {
  return shapes_;
}


void GraphicsScene::Clear() {
  shapes_.clear();
}


void GraphicsScene::EraseShape(Shape* shape) {
  auto it = std::find(shapes_.begin(),
shapes_.end(), shape);
  if (it != shapes_.end()) {
    shapes_.erase(it);
  }
}
```

GraphicsView.cpp

```
#include "GraphicsView.hpp"
#include "Shape.hpp"
#include "LookAtCamera.hpp"
#include <QKeyEvent>
#include "SGLMath.hpp"
#include "Line.hpp"
#include "Triangle.hpp"
#include <algorithm>


void sleep(int n) {
  QEventLoop loop;
  QTimer::singleShot(n, &loop, SLOT(quit()));
  loop.exec();
}
```

```
GraphicsView::GraphicsView(QWidget* parent,
GraphicsScene* scene)
: QLabel{parent}
, image_{601, 601, QImage::Format_RGB888}
, zbuffer_{{601, 601}}
, scene_{scene} {
  auto look_at_camera = new LookAtCamera;
  look_at_camera->Reset({300, 200, 300}, {35,
41.67, 0}, {0, 0, 1});
  camera_ = look_at_camera;
  image_.fill(SGL::white);
  Scene(scene_);
  setFocusPolicy(Qt::StrongFocus);
  startTimer(1000/40);
}


void GraphicsView::Render() {
  if (!scene_) {
    return;
  }
  zbuffer_.Clear();
  image_.fill(SGL::white);
  for (int i = 0; i < image_.width(); ++i) {
    image_.setPixel(i, 0, SGL::black);
    image_.setPixel(i, image_.height() - 1,
SGL::black);
  }
  for (int i = 0; i < image_.height(); ++i) {
    image_.setPixel(0, i, SGL::black);
    image_.setPixel(image_.width() - 1, i,
SGL::black);
  }
  Painter painter{image_, zbuffer_, *camera_,
view_type_};
  for (::Shape* shape : scene_->Shapes()) {
    painter.Save();
    shape->Draw(painter);
    painter.Restore();
  }
  setPixmap(QPixmap::fromImage(image_));
}


GraphicsScene *GraphicsView::Scene() const {
  return scene_;
}


void GraphicsView::Scene(GraphicsScene *scene) {
  scene_ = scene;
  if (scene_) {
    Render();
  }
}


void GraphicsView::keyPressEvent(QKeyEvent*
event) {
  const std::vector<::Shape*> v = scene_-
>Shapes();
  ::Shape* shape;

  buttons_[event->key()] = true;

  switch (event->key()) {
    case Qt::Key_P:
      for (int i = 0; i < v.size(); ++i) {
        shape = v[i];
        if (dynamic_cast<::Line*>(shape)) {
          continue;
        }
        float angle = rand() % 30 - 15;
        QVector3D begin = shape->BasePoint();
        shape->MoveBasePoint(shape->J());
        QVector3D end = shape->BasePoint();
        if (shape->Color().has_value()) {
          scene_->PushShape(new Line(begin, end,
SGL::white));
        }

        if (rand() % 100 > 90) {
          angle = rand() % 90 - 45;
        }
        shape->Rotate(angle, {0, 0, 1, 0});
      }
      break;
    case Qt::Key_R:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Scale(1.1, 1, 1);
      }
      break;
    case Qt::Key_T:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Scale(1, 1.1, 1);
      }
      break;
    case Qt::Key_Y:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Scale(1, 1, 1.1);
      }
      break;
    case Qt::Key_F:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Scale(1 / 1.1, 1, 1);
      }
      break;
    case Qt::Key_G:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Scale(1, 1 / 1.1, 1);
      }
      break;
    case Qt::Key_H:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Scale(1, 1, 1 / 1.1);
      }
      break;
    case Qt::Key_U:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Rotate(5, shape->I());
      }
      break;
    case Qt::Key_I:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Rotate(5, shape->J());
      }
      break;
    case Qt::Key_O:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Rotate(5, shape->K());
      }
      break;
    case Qt::Key_J:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Rotate(-5, shape->I());
      }
      break;
    case Qt::Key_K:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Rotate(-5, shape->J());
      }
```

```
      break;
    case Qt::Key_L:
      for (auto[shape, brush] : selected_shapes)
{
        shape->Rotate(-5, shape->K());
      }
      break;
  }
  if (is_free_camera_) {
    FreeCameraChange();
  }

  QLabel::keyPressEvent(event);
}


void GraphicsView::FreeCameraChange() {
  auto camera =
dynamic_cast<LookAtCamera*>(camera_);
  QVector3D eye = camera->Eye();
  QVector3D center = camera->Center();
  QVector3D up = camera->Up();
  QVector3D z = (center - eye).normalized();
  QVector3D x = QVector3D::crossProduct(camera-
>Up(), z).normalized();
  QVector3D y = QVector3D::crossProduct(z,
x).normalized();
  constexpr int STEP = 5;
  constexpr int ANGLE_STEP = 5;
  QMatrix4x4 rotate;
  if (buttons_[Qt::Key_A]) {
    eye = eye - STEP * (x).normalized();
    center = center - STEP * (x).normalized();
  }
  if (buttons_[Qt::Key_W]) {
    eye = eye + STEP * (center -
eye).normalized();
    center = center + STEP * (center -
eye).normalized();
  }
  if (buttons_[Qt::Key_S]) {
    eye = eye - STEP * (center -
eye).normalized();
    center = center - STEP * (center -
eye).normalized();
  }
  if (buttons_[Qt::Key_D]) {
    eye = eye + STEP * (x).normalized();
    center = center + STEP * (x).normalized();
  }
  if (buttons_[Qt::Key_Z]) {
    eye = eye - STEP * (y).normalized();
    center = center - STEP * (y).normalized();
  }
  if (buttons_[Qt::Key_X]) {
    eye = eye + STEP * (y).normalized();
    center = center + STEP * (y).normalized();
  }
  if (buttons_[Qt::Key_Down]) {
    rotate = SGLMath::Rotate(ANGLE_STEP, x, eye);
    center = rotate * center;
    up = SGLMath::Rotate(ANGLE_STEP, x) * up;
  }
  if (buttons_[Qt::Key_Up]) {
    rotate = SGLMath::Rotate(-ANGLE_STEP, x,
eye);
    center = rotate * center;
    up = SGLMath::Rotate(-ANGLE_STEP, x) * up;
  }
  if (buttons_[Qt::Key_Left]) {
    rotate = SGLMath::Rotate(-ANGLE_STEP, y,
eye);
    center = rotate * center;
    up = SGLMath::Rotate(-ANGLE_STEP, y) * up;
  }
  if (buttons_[Qt::Key_Right]) {
    rotate = SGLMath::Rotate(ANGLE_STEP, y, eye);
    center = rotate * center;
    up = SGLMath::Rotate(ANGLE_STEP, y) * up;
  }
  if (buttons_[Qt::Key_Q]) {
    rotate = SGLMath::Rotate(ANGLE_STEP, z, eye);
    center = rotate * center;
    up = SGLMath::Rotate(ANGLE_STEP, z) * up;
  }
  if (buttons_[Qt::Key_E]) {
    rotate = SGLMath::Rotate(-ANGLE_STEP, z,
eye);
    center = rotate * center;
    up = SGLMath::Rotate(-ANGLE_STEP, z) * up;
  }
  if (buttons_[Qt::Key_2]) {
    camera->near_ += 0.3f;
  }
  if (buttons_[Qt::Key_1]) {
    camera->near_ -= 0.3f;
  }
  camera->Reset(eye, center, up);
  Render();
}

void GraphicsView::keyReleaseEvent(QKeyEvent*
event) {
  buttons_[event->key()] = false;
  QLabel::keyReleaseEvent(event);
}


void GraphicsView::mousePressEvent(QMouseEvent*
mouse) {
  if (mouse->button() == Qt::LeftButton) {
    int x = mouse->x();

    int y = height() - mouse->y() - 1;
    y = y - height() + image_.height() +
(height() - image_.height()) / 2;
    if (!((0 <= y && y < image_.height()) && (0
<= x && x < image_.width()))) {
      return;
    }
    ::Shape* shape = zbuffer_[x][y].parent;
    if (shape == nullptr) {
      UnselectedAllShape();
      Render();
      return;
    }

    if (mouse->modifiers() !=
Qt::KeyboardModifier::ControlModifier) {
      UnselectedAllShape();
    }
    auto[inserted_it, is_selected] =
        selected_shapes.emplace(shape, shape-
>Brush());

    if (!is_selected) {
      return;
    }

    shape->Brush(qRgb(0, 150, 0));
  }
  Render();
}


void GraphicsView::UnselectedAllShape() {
  for (auto[shape, brush] : selected_shapes) {
    shape->Brush(brush);
  }
  selected_shapes.clear();
}
```

```
void GraphicsView::resizeEvent(QResizeEvent*
event) {
  image_ = image_.scaled(event->size());
  setPixmap(QPixmap::fromImage(image_));
  zbuffer_.Resize({event->size().width(), event-
>size().height()});
  if (auto camera =
dynamic_cast<LookAtCamera*>(camera_); camera) {
    camera->WindowSize(event->size());
  }
  Render();
  QLabel::resizeEvent(event);
}


void GraphicsView::timerEvent(QTimerEvent* ) {
  FreeCameraChange();
}


void GraphicsView::SetCentral() {
  camera_->SetCentral();
}


void GraphicsView::SetParallels() {
  camera_->SetParallels();
}


void GraphicsView::SetPolygonView() {
  view_type_ = SGL::ViewType::polygon;
}


void GraphicsView::SetLineView() {
  view_type_ = SGL::ViewType::line;
}


void GraphicsView::PanoramStep(QVector3D center)
{
  auto camera =
dynamic_cast<LookAtCamera*>(camera_);
  assert(camera);
  auto eye = SGLMath::Rotate(1, {0, 0, 1, 0},
                             center) * camera-
>Eye();

  camera->Reset(eye, center, {0, 0, 1});
}

void GraphicsView::ClearSelected() {
  selected_shapes.clear();
}

const std::unordered_map<::Shape*,
                         std::optional<QRgb>>&
GraphicsView::SelectedShapes() const {
  return selected_shapes;
}


Camera* GraphicsView::Camera() const {
  return camera_;
}


void GraphicsView::Camera(class Camera* camera) {
  camera_ = camera;
}
```

LookAtCamera.cpp

```
#include "LookAtCamera.hpp"


QVector3D LookAtCamera::Project(QVector3D point)
{
  QVector3D result;
  float width = std::max(window_size_.width() -
1, 0);
  float height = std::max(window_size_.height() -
1, 0);
  QVector3D camera_point = model_view_ * point;

  if (!(near_ <= camera_point.z() &&
camera_point.z() <= far_)) {
    throw std::runtime_error{"Bad point"};
  }
  if (is_central) {
    QVector3D project_point = projection_ *
camera_point;
    QMatrix4x4 window_matrix{
        width / 2, 0, 0, width / 2,
        0, height / 2, 0, height / 2,
        0, 0, 1, 0,
        0, 0, 0, 1
    };
    if (qAbs(project_point.x()) >= 1 ||
        qAbs(project_point.y()) >= 1) {
      throw std::runtime_error{"Bad point"};
    }
    result = window_matrix * project_point;
  } else {
    QVector3D project_point = projection_ *
camera_point;
    QMatrix4x4 window_matrix{
        1, 0, 0, width / 2,
        0, 1, 0, height / 2,
        0, 0, 1, 0,
        0, 0, 0, 1
    };
    QVector3D window_point = window_matrix *
project_point;
    if (!((0 <= window_point.x() &&
window_point.x() < width) &&
        (0 <= window_point.y() &&
window_point.y() < height))) {
      throw std::runtime_error{"Bad point"};
    }
    result = window_point;
  }
  return result;
}


void LookAtCamera::Reset(QVector3D eye, QVector3D
center, QVector3D up) {
  eye_ = eye;
  center_ = center;
  up_ = up;
  CalculateMatrix();
}


LookAtCamera &LookAtCamera::operator=(const
LookAtCamera &rhs) {
  Reset(rhs.eye_, rhs.center_, rhs.up_);
  return *this;
}


LookAtCamera::LookAtCamera(QVector3D eye,
QVector3D center, QVector3D up)
: eye_{eye}
, center_{center}
, up_{up} {
  CalculateMatrix();
}
```

```
void LookAtCamera::CalculateMatrix() {
  model_view_.setToIdentity();
  QVector3D z = (center_ - eye_).normalized();
  QVector3D x = QVector3D::crossProduct(up_,
z).normalized();
  QVector3D y = QVector3D::crossProduct(z,
x).normalized();
  QMatrix4x4 tr;
  model_view_.setRow(0, x);
  model_view_.setRow(1, y);
  model_view_.setRow(2, z);
  QVector4D last_row{-eye_, 1};
  tr.setColumn(3, last_row);
  model_view_ *= tr;
  if (is_central) {
    SetCentral();
  } else {
    SetParallels();
  }
}


const QVector3D& LookAtCamera::Eye() const {
  return eye_;
}


const QVector3D& LookAtCamera::Center() const {
  return center_;
}


const QVector3D& LookAtCamera::Up() const {
  return up_;
}


float LookAtCamera::Near() const {
  return near_;
}


float LookAtCamera::Far() const {
  return far_;
}


const QSize& LookAtCamera::WindowSize() const {
  return window_size_;
}


void LookAtCamera::WindowSize(const QSize&
window_size) {
  window_size_ = window_size;
}
```

Painter.cpp

```
#include <algorithm>
#include <Box.hpp>
#include "LookAtCamera.hpp"
#include "Painter.hpp"
#include "Line.hpp"


void Painter::SetPixel(int x, int y, QRgb color)
{
  if ((0 < x && x < image_.width()) && 0 < y && y
< image_.height()) {
    y = image_.height() - y - 1;
    image_.setPixel(x, y, color);
  }
}


Painter::Painter(QImage &image, ZBuffer& zbuffer,
Camera& camera,
                 SGL::ViewType view_type)
: image_{image}
, zbuffer_{zbuffer}
, camera_{camera}
, view_type_{view_type} {
  ;
}


SGL::Size Painter::ImageSize() const {
  return {image_.width(), image_.height()};
}


void Painter::DrawLine(QVector3D begin, QVector3D
end, Shape* parent) {
  if (view_type_ == SGL::ViewType::polygon) {
    return;
  }
  begin = transform_ * begin;
  end = transform_ * end;
  try {
    begin = camera_.Project(begin);
    end = camera_.Project(end);
  } catch(std::runtime_error& error) {
    return;
  }
  begin.setX(static_cast<int>(begin.x() + 0.5));
  begin.setY(static_cast<int>(begin.y() + 0.5));
  end.setX(static_cast<int>(end.x() + 0.5));
  end.setY(static_cast<int>(end.y() + 0.5));
  if (!color_.has_value()) {
    return;
  }

  bool steep{false};
  if (std::abs(begin.x() - end.x()) <
std::abs(begin.y() - end.y())) {
    std::swap(begin[0], begin[1]);
    std::swap(end[0], end[1]);
    steep = true;
  }
  if (begin.x() > end.x()) {
    std::swap(begin, end);
  }

  int dx = end.x() - begin.x();
  int dy = end.y() - begin.y();
  const int derror = std::abs(dy) * 2;
  int error  = 0;
  int y = begin.y();
  float dz = end.z() - begin.z();
  for (int x = begin.x(); x <= end.x(); ++x) {
    float progress = (end.x() == begin.x()) ? 1 :
float(x - begin.x()) / (end.x() - begin.x());
    float z = begin.z() + dz * progress;
    if (steep) {
      DrawPoint(QVector3D(y, x, z), parent);
    } else {
      DrawPoint(QVector3D(x, y, z), parent);
    }
    error += derror;
    if (error > dx) {
      y += begin.y() < end.y() ? 1 : -1;
      error -= dx * 2;
    }
  }
}


std::optional<QRgb> Painter::Color() const {
  return color_;
}
```

```
void Painter::Color(std::optional<QRgb> color) {
  color_ = color;
}


std::optional<QRgb> Painter::Brush() const {
  return brush_;
}


void Painter::Brush(std::optional<QRgb> brash) {
  brush_ = brash;
}


void Painter::DrawTriangle(std::array<QVector3D,
3> points, Shape* parent) {
  if (view_type_ == SGL::ViewType::line) {
    Brush(std::nullopt);
  }
  auto camera_points = points;
  for (QVector3D& point : camera_points) {
    point = transform_ * point;
  }
  auto normal_vector = QVector3D::crossProduct(
      camera_points[2] - camera_points[0],
      camera_points[1] - camera_points[0]
      ).normalized();
  float intensity =
QVector3D::dotProduct(normal_vector,

QVector3D{-2, -1, -3}.normalized());
  std::optional<QRgb> old_brush = brush_;
  float real_intensity = std::max(0.25f,
intensity);
  if (brush_.has_value()) {
    QRgb new_brush = qRgb(qRed(brush_.value()) *
real_intensity,
                          qGreen(brush_.value())
* real_intensity,
                          qBlue(brush_.value()) *
real_intensity);
    Brush(new_brush);
  }
  for (QVector3D& point : camera_points) {
    try {
      point = camera_.Project(point);
    } catch(std::runtime_error& er) {
      return;
    }
    point.setX(static_cast<int>(point.x() +
0.5));
    point.setY(static_cast<int>(point.y() +
0.5));
    // point.setZ(static_cast<int>(point.z() +
0.5));
  }
  std::sort(camera_points.begin(),
camera_points.end(), [](QVector3D a, QVector3D b)
{
    return a.y() < b.y();
  });
  auto DrawCarcass = [this, &points, parent]() {
    Line l1(points[0], points[1], color_,
parent);
    l1.Draw(*this);
    Line l2(points[1], points[2], color_,
parent);
    l2.Draw(*this);
    Line l3(points[2], points[0], color_,
parent);
    l3.Draw(*this);
  };

  if (!brush_.has_value()) {
    DrawCarcass();
    return;
  }

  auto FillLine = [this, parent](QVector3D a,
QVector3D b, int y) {
    if (b.x() < a.x()) {
      std::swap(a, b);
    }
    const float dz = b.z() - a.z();
    for (int x = a.x() + 1; x <= b.x(); ++x) {
      float progress = b.x() == a.x() ? 1 :
static_cast<float>(x - a.x()) / (b.x() - a.x());
      // QVector3D point = a + (b - a) *
progress;
      float z = a.z() + dz * progress;
      DrawBrush(QVector3D(x, y, z), parent);
    }
  };

  int total_height = camera_points[2].y() -
camera_points[0].y();
  int segment_height = camera_points[1].y() -
camera_points[0].y();
  if (segment_height == 0) {
    segment_height = 1;
  }
  for (int y = camera_points[0].y(); y <
camera_points[1].y(); ++y) {
    float alpha = static_cast<float>(y -
camera_points[0].y()) / total_height;
    float beta = static_cast<float>(y -
camera_points[0].y()) / segment_height;

    QVector3D a = camera_points[0] +
(camera_points[2] - camera_points[0]) * alpha;
    QVector3D b = camera_points[0] +
(camera_points[1] - camera_points[0]) * beta;
    FillLine(a, b, y);
  }

  segment_height = camera_points[2].y() -
camera_points[1].y();
  if (segment_height == 0) {
    segment_height = 1;
  }
  for (int y = camera_points[1].y() + 0.5; y <
camera_points[2].y(); ++y) {
    float alpha = static_cast<float>(y -
camera_points[0].y()) / total_height;
    float beta = static_cast<float>(y -
camera_points[1].y()) / segment_height;
    QVector3D a = camera_points[0] +
(camera_points[2] - camera_points[0]) * alpha;
    QVector3D b = camera_points[1] +
(camera_points[2] - camera_points[1]) * beta;
    FillLine(a, b, y);
  }

  DrawCarcass();
  Brush(old_brush);
}


void Painter::DrawPoint(QVector3D point, Shape*
shape) {
  DrawPixel(point, color_.value(), shape);
}


void Painter::DrawBrush(QVector3D point, Shape*
shape) {
  if (brush_.has_value()) {
    DrawPixel(point, brush_.value(), shape);
  }
}
```

```
void Painter::DrawPixel(QVector3D point, QRgb
color, Shape* parent) {
  QPoint p = point.toPoint();
  int i = (p.x());
  int j = (p.y());
  try {
    LookAtCamera &camera =
dynamic_cast<LookAtCamera&>(camera_); // that's
perhaps throw exception
    if (zbuffer_[i][j].z >= point.z()) {
      zbuffer_[i][j].z = point.z();
      zbuffer_[i][j].parent = parent;
      SetPixel(i, j, color);
    }
  } catch(...) {

  }
}


void Painter::Save() {
  save_colors[0] = color_;
  save_colors[1] = brush_;
  save_transform_ = transform_;
}


void Painter::Restore() {
  color_ = save_colors[0];
  brush_ = save_colors[1];
  transform_ = save_transform_;
}


void Painter::AppendTransform(QMatrix4x4
transform) {
  transform_ =  transform_ * transform;
}
```

SGLMath.cpp

```
#include "SGLMath.hpp"
#include <QtMath>
#include <QJsonDocument>
#include <QJsonArray>
#include <QJsonObject>

namespace SGLMath {

QMatrix4x4 Rotate(float a, QVector4D v) {
  a = qDegreesToRadians(a);
  QMatrix4x4 rm;
  float x = v.x();
  float y = v.y();
  float z = v.z();
  QVector3D first_row{cos(a) + (1 - cos(a)) * x *
x,
                      (1 - cos(a)) * x * y -
sin(a) * z,
                      (1 - cos(a)) * x * z +
sin(a) * y};
  QVector3D second_row{(1 - cos(a)) * y * x +
sin(a) * z,
                       cos(a) + (1 - cos(a)) * y
* y,
                       (1 - cos(a)) * y * z -
sin(a) * x};
  QVector3D third_row{(1 - cos(a)) * z * x -
sin(a) * y,
                      (1 - cos(a)) * z * y +
sin(a) * x,
                      cos(a) + (1 - cos(a)) * z *
z};
  rm.setRow(0, first_row);
  rm.setRow(1, second_row);
  rm.setRow(2, third_row);

  return rm;
}


QMatrix4x4 Rotate(float a, QVector4D v, QVector3D
pos) {
  return Move(pos) * Rotate(a, v) * Move(-pos);
}


QMatrix4x4 Move(QVector4D v) {
  QMatrix4x4 move_matrix;
  move_matrix.setRow(0, {1, 0, 0, v.x()});
  move_matrix.setRow(1, {0, 1, 0, v.y()});
  move_matrix.setRow(2, {0, 0, 1, v.z()});
  move_matrix.setRow(3, {0, 0, 0, 1});
  return move_matrix;
}


QMatrix4x4 Scale(float a, float b, float c) {
  return QMatrix4x4(a, 0, 0, 0,
                    0, b, 0, 0,
                    0, 0, c, 0,
                    0, 0, 0, 1);
}


QJsonArray ToJson(QVector4D vector) {
  QJsonArray row;
  row.push_back(vector[0]);
  row.push_back(vector[1]);
  row.push_back(vector[2]);
  row.push_back(vector[3]);
  return row;
}


QJsonArray ToJson(QMatrix4x4 matrix) {
  QJsonArray json_matrix;
  json_matrix.push_back(ToJson(matrix.row(0)));
  json_matrix.push_back(ToJson(matrix.row(1)));
  json_matrix.push_back(ToJson(matrix.row(2)));
  json_matrix.push_back(ToJson(matrix.row(3)));
  return json_matrix;
}


QVector4D ToVector4D(QJsonArray json) {
  QVector4D v;
  for (int i = 0; i < json.size(); ++i) {
    v[i] = json[i].toDouble();
  }
  return v;
}


QMatrix4x4 ToMatrix4x4(QJsonArray json) {
  QMatrix4x4 result;
  for (int i = 0; i < json.size(); ++i) {
    result.setRow(i,
ToVector4D(json[i].toArray()));
  }
  return result;
}
} // SGLMath namespace
```

ZBuffer.cpp

```
#include "ZBuffer.hpp"


ZBuffer::ZBuffer(QSize size)
```

```
: storage_{new Point[size.width() *
size.height()]}
, size_{size} {
  Clear();
}


void ZBuffer::Resize(QSize size) {
  delete[] storage_;
  storage_ = new Point[size.width() *
size.height()];
  size_ = size;
}


ZBuffer::Point* ZBuffer::operator[](int i)
noexcept {
  return storage_ + i * size_.height();
}


ZBuffer::~ZBuffer() {
  delete[] storage_;
}


void ZBuffer::Clear() noexcept {
  for (int i = 0; i < size_.width() *
size_.height(); ++i) {
    storage_[i].z =
std::numeric_limits<float>::max();
    storage_[i].parent = nullptr;
  }
}


const QSize& ZBuffer::Size() const {
  return size_;
}
```

Box.hpp

```
#pragma once
#include "Shape.hpp"


class Box : public Shape {
 public:
  Box(QVector3D base_point, float length, float
width, float height,
  std::optional<QRgb> color = std::nullopt,
  std::optional<QRgb> brush = std::nullopt,
  Shape* parent = nullptr);
  std::vector<Triangle> Triangulate() override;
  [[nodiscard]] QVector3D Center() const
override;

 protected:
  void DrawShape(Painter &painter) override;

 private:
  float length_{};
  float width_{};
  float height_{};
};
```

Drill.hpp

```
#pragma once
#include "Shape.hpp"

class Drill : public Shape
{
public:
    Drill() = default;
    Drill(float battery_width,
          float battery_length,
          float battery_height,
          int switch_count,
          float handle_width,
          float handle_length,
          float handle_height,
          float trigger_length,
          float trigger_height,
          float workpart_length,
          float workpart_radius,
          float sphere_radius,
          float workpart2_length,
          float workpart2_radius,
          float drill_length,
          float drill_radius,
          std::optional<QRgb> color =
std::nullopt,
          std::optional<QRgb> brush =
std::nullopt,
          QVector3D base_point = {},
          Shape* parent = nullptr);

    QVector3D Center() const override;
    std::vector<Triangle> Triangulate() override;

    [[nodiscard]] QJsonObject ToJson() const
override;
    void FromJson(const QJsonObject& json)
override;
    QString type() override;

protected:
    void DrawShape(Painter& painter) override;

private:
    float battery_width_;
    float battery_length_;
    float battery_height_;
    int switch_count_;
    float handle_width_;
    float handle_length_;
    float handle_height_;
    float trigger_length_;
    float trigger_height_;
    float workpart_length_;
    float workpart_radius_;
    float sphere_radius_;
    float workpart2_length_;
    float workpart2_radius_;
    float drill_length_;
    float drill_radius_;

};
```

Ellipse.hpp

```
#pragma once
#include "Shape.hpp"


class [[maybe_unused]] Ellipse : public Shape {
 public:
  [[maybe_unused]] Ellipse(float a_far, float
b_far,
                  std::optional<QRgb> color =
std::nullopt,
                  std::optional<QRgb> brush =
std::nullopt,
                  QVector3D base_point = {},
                  float a_near = 0, float b_near
= 0);

  [[nodiscard]] QVector3D Center() const
override;
  std::vector<Triangle> Triangulate() override;

 protected:
```

```
  void DrawShape(Painter& painter) override;

 private:
  float a_near_;
  float b_near_;
  float a_far_;
  float b_far_;
};
```

Ellipsoid.hpp

```
#pragma once
#include "Shape.hpp"


class Ellipsoid : public Shape {
 public:
  Ellipsoid(float a,
            float b,
            float c,
          std::optional<QRgb> color =
std::nullopt,
          std::optional<QRgb> brush =
std::nullopt,
          QVector3D base_point = {},
          Shape* parent = nullptr,
             float part = 1);
  [[nodiscard]] QVector3D Center() const
override;
  std::vector<Triangle> Triangulate() override;
  const std::vector<Triangle>& MyTriangulate();

 protected:
  void DrawShape(Painter& painter) override;

 private:
  float a_;
  float b_;
  float c_;
  float a_old_;
  float b_old_;
  float c_old_;
  float part_;
  std::vector<Triangle> triangles_;
};
```

Frustum.hpp

```
#pragma once
#include "Shape.hpp"


class Frustum : public Shape {
 public:
  Frustum(float a_far_bottom,
                  float b_far_bottom,
                  float a_far_top,
                  float b_far_top,
                  float height,
                  std::optional<QRgb> color =
std::nullopt,
                  std::optional<QRgb> brush =
std::nullopt,
                  QVector3D base_point = {},
                  float a_near_bottom = 0,
                  float b_near_bottom = 0,
                  float a_near_top = 0,
                  float b_near_top = 0,
                  Shape* parent = nullptr);

  [[nodiscard]] QVector3D Center() const
override;
  std::vector<Triangle> Triangulate() override;

 protected:
  void DrawShape(Painter& painter) override;

 private:
  float a_far_bottom_;
  float a_near_bottom_;
  float b_far_bottom_;
  float b_near_bottom_;
  float a_far_top_;
  float a_near_top_;
  float b_far_top_;
  float b_near_top_;
  float height_;
};
```

Line.hpp

```
#include <Structs.hpp>
#include "Shape.hpp"
#include <QVector3D>

class Line : public Shape {
 public:
  explicit Line(QVector3D begin = {}, QVector3D
end = {},
                std::optional<QRgb> color = {},
                Shape* parent = nullptr);
  [[maybe_unused]] explicit Line(int x0, int y0,
int z0, int x1, int y1, int z1,
                std::optional<QRgb> color = {},
                Shape* parent = nullptr);
  [[nodiscard]] std::vector<Triangle>
Triangulate() override;

  [[nodiscard]] QVector3D Center() const
override;
  [[maybe_unused]] [[nodiscard]] const QVector3D
&Begin() const;
  [[maybe_unused]] void Begin(const QVector3D
&begin);
  [[maybe_unused]] [[nodiscard]] const QVector3D
&AnEnd() const;
  [[maybe_unused]] void AnEnd(const QVector3D
&an_end);

 protected:
  void DrawShape(Painter& painter) override;

 private:
  QVector3D begin_;
  QVector3D end_;
};
```

Pyramid.hpp

```
#pragma once
#include "Shape.hpp"


class Pyramid : public Shape {
 public:
  Pyramid(float bottom_length, float
bottom_width, float height,
          std::optional<QRgb> color =
std::nullopt,
          std::optional<QRgb> brush =
std::nullopt,
          QVector3D base_point = {},
          float top_length = 0,
          float top_width = 0,
          float bottom_near_length = 0,
          float bottom_near_width = 0,
          float top_near_length = 0,
          float top_near_width = 0,
          Shape* parent = nullptr);

  [[nodiscard]] QVector3D Center() const
override;
  std::vector<Triangle> Triangulate() override;
```

```
 protected:
  void DrawShape(Painter& painter) override;

 private:
  float bottom_length_;
  float bottom_width_;
  float height_;
  float top_length_;
  float top_width_;
  float bottom_near_length_;
  float bottom_near_width_;
  float top_near_length_;
  float top_near_width_;
};
```

Quadrangle.hpp

```
#pragma once
#include "Shape.hpp"


class Quadrangle : public Shape {
 public:
  /*!
   * @param vertexes must order by<br>
   * <pre>
   * c is color<br>
   * b is brush<br>
   * 1ccccccccc2<br>
   * cbbbbbbbbbbc<br>
   * cbbbbbbbbbbc<br>
   * 3ccccccccccc4<br>
   * </pre>
   */
  explicit Quadrangle(std::array<QVector3D, 4>
vertexes,
              std::optional<QRgb> color =
std::nullopt,
              std::optional<QRgb> brush =
std::nullopt,
              QVector3D base_point = {},
              Shape* parent = nullptr);
  [[nodiscard]] std::vector<Triangle>
Triangulate() override;

  [[nodiscard]] QVector3D Center() const
override;

 protected:
  void DrawShape(Painter &painter) override;

 private:
  QVector3D vertexes_[4];
};
```

Shape.hpp

```
#pragma once
#include <vector>
#include <QRgb>
#include <optional>
#include <QMatrix4x4>

class Painter;
class Triangle;


class Shape {
 public:
  explicit Shape(QVector3D base_point = {},
std::optional<QRgb> color = std::nullopt,
        std::optional<QRgb> brush = std::nullopt,
        Shape* parent = nullptr)
  : base_point_{base_point}
  , color_{color}
  , brush_{brush}
  , parent_{parent} {

  }
  [[maybe_unused]] void AppendModify(QMatrix4x4
modify_matrix) noexcept;

  void Rotate(float a, QVector4D v);
  void Rotate(float a, QVector4D v, QVector3D
pos);
  void Move(QVector4D);
  void Scale(float a, float b, float c);
  void MoveBasePoint(QVector4D v);

  [[nodiscard]] virtual QVector3D Center() const
= 0;
  void Draw(Painter& painter);
  [[nodiscard]] virtual std::vector<Triangle>
Triangulate() = 0;
  virtual ~Shape() = default;

  [[nodiscard]] const QVector3D& BasePoint()
const;
  void BasePoint(const QVector3D& base_point);
  [[nodiscard]] const std::optional<QRgb>&
Color() const;
  void Color(const std::optional<QRgb> &color);
  [[nodiscard]] const std::optional<QRgb>&
Brush() const;
  void Brush(const std::optional<QRgb>& brush);

  [[maybe_unused]] [[nodiscard]] const QVector4D&
I() const;
  [[nodiscard]] const QVector4D& J() const;
  [[maybe_unused]] [[nodiscard]] const QVector4D&
K() const;

  Shape* BaseParent() noexcept;
  [[nodiscard]] const QMatrix4x4& ModifyMatrix()
const;
  void ModifyMatrix(const QMatrix4x4&
modify_matrix);
  [[nodiscard]] virtual QJsonObject ToJson()
const;
  virtual void FromJson(const QJsonObject& json);
  [[maybe_unused]] virtual QString type();

 protected:
  std::optional<QRgb> color_{};
  std::optional<QRgb> brush_{};
  QMatrix4x4 modify_matrix_;
  QVector3D base_point_;
  Shape* parent_ = nullptr;

  QVector4D i{1, 0, 0, 0};
  QVector4D j{0, 1, 0, 0};
  QVector4D k{0, 0, 1, 0};

  virtual void DrawShape(Painter& painter) = 0;
};
```

Switch.hpp

```
#pragma once
#include "Shape.hpp"

class Switch : public Shape
{
public:
    Switch(int width,
           int height,
           std::optional<QRgb> color =
std::nullopt,
           std::optional<QRgb> brush =
std::nullopt,
           QVector3D base_point = {},
           Shape* parent = nullptr);

    QVector3D Center() const override;
```

```
    std::vector<Triangle> Triangulate() override;


protected:
    void DrawShape(Painter& painter) override;

private:
    int width_;
    int height_;
};
```

Triangle.hpp

```
#pragma once
#include "Painter.hpp"
#include "Shape.hpp"

class Triangle : public Shape {
 public:
  explicit Triangle(std::array<QVector3D, 3>
points = {},
                    std::optional<QRgb> color =
std::nullopt,
                    std::optional<QRgb> brush =
std::nullopt,
                    QVector3D base_point = {},
                    Shape* parent = nullptr);
  explicit Triangle(QVector3D a, QVector3D b,
QVector3D c,
                    std::optional<QRgb> color =
std::nullopt,
                    std::optional<QRgb> brush =
std::nullopt,
                    QVector3D base_point = {},
                    Shape* parent = nullptr);
  [[nodiscard]] std::vector<Triangle>
Triangulate() override;
  [[maybe_unused]] [[nodiscard]] const
std::array<QVector3D, 3>& Points() const;
  [[nodiscard]] QVector3D Center() const
override;

 protected:
  void DrawShape(Painter &painter) override;

 private:
  std::array<QVector3D, 3> points_;
};
```

Box.cpp

```
#include "Box.hpp"
#include "Triangle.hpp"
#include "Quadrangle.hpp"
#include "SGLMath.hpp"
#include <algorithm>

template<typename T, typename U>
void AppendVector(std::vector<U>& to, const
std::vector<T>& from) {
  std::copy(from.begin(), from.end(),
std::back_inserter(to));
}

Box::Box(QVector3D base_point,
         float length,
         float width,
         float height,
         std::optional<QRgb> color,
         std::optional<QRgb> brush,
         Shape* parent)
: Shape(base_point, color, brush, parent)
, length_{length}
, width_{width}
, height_{height} {

}
```

```
void Box::DrawShape(Painter& painter) {
  Quadrangle face1{
    std::array<QVector3D, 4>{
        QVector3D{0, 1, 0} * height_,
        QVector3D{1, 0, 0} * length_ +
            QVector3D{0, 1, 0} * height_,
          QVector3D(0, 0, 0),
          QVector3D{1, 0, 0} * length_,

      },
      color_,
      brush_,
      {},
      BaseParent()
  };

  face1.Draw(painter);
  //
  Quadrangle face2{
      std::array<QVector3D, 4>{
          QVector3D{0, 0, width_},
          QVector3D{length_, 0, width_},
          QVector3D(0, height_, width_),
          QVector3D{length_, height_, width_},

      },
      color_,
      brush_,
      {},
      BaseParent()
  };

  face2.Draw(painter);

  Quadrangle face3{
      std::array<QVector3D, 4>{
         QVector3D{0, 0, 1} * width_,
         QVector3D{0, 0, 1} * width_
         + QVector3D{0, 1, 0} * height_,
         QVector3D(0, 0, 0),
         QVector3D{0, 1, 0} * height_
      },
      color_,
      brush_,
      {},
      BaseParent()
  };
  face3.Draw(painter);

  Quadrangle face4{
      std::array<QVector3D, 4>{
          QVector3D(length_, 0, 0),
          QVector3D{length_, height_, 0},
          QVector3D{length_, 0, width_},
          QVector3D{length_, height_, width_}

      },
      color_,
      brush_,
      {},
      BaseParent()
  };
  face4.Draw(painter);

  Quadrangle face5 {
      std::array<QVector3D, 4>{
          QVector3D(0, 0, 0),
          QVector3D{length_, 0, 0},
          QVector3D{0, 0, width_},
          QVector3D{length_, 0, width_},
      },
      color_,
      brush_,
      {},
```

```
        BaseParent()
    };
    face5.Draw(painter);
    //
    Quadrangle face6 {
        std::array<QVector3D, 4>{
            QVector3D{0, height_, width_},
            QVector3D{length_, height_, width_},
            QVector3D(0, height_, 0),
            QVector3D{length_, height_, 0},
        },
        color_,
        brush_,
        {},
        BaseParent()
    };
    face6.Draw(painter);
}
std::vector<Triangle> Box::Triangulate() {
    std::vector<Triangle> result;
    auto it = std::back_inserter(result);
    Quadrangle face1{
        std::array<QVector3D, 4>{
            QVector3D(0, 0, 0),
            QVector3D{1, 0, 0} * length_,
            QVector3D{0, 1, 0} * height_,
            QVector3D{1, 0, 0} * length_ +
                QVector3D{0, 1, 0} * height_,
        },
        color_,
        brush_,
        {},
        BaseParent()
    };

    Quadrangle face2 = face1;
    face2.Move(QVector3D{0, 0, 1} * width_);

    Quadrangle face3{
        std::array<QVector3D, 4>{
            QVector3D{0, 0, 1} * width_,
            QVector3D{0, 0, 1} * width_
                + QVector3D{0, 1, 0} * height_,
            QVector3D(0, 0, 0),
            QVector3D{0, 1, 0} * height_
        },
        color_,
        brush_,
        {},
        BaseParent()
    };
    //
    Quadrangle face4 = face3;
    face4.Brush(brush_);
    face4.Move(QVector3D{1, 0, 0} * length_);

    Quadrangle face5 {
        std::array<QVector3D, 4>{
            QVector3D(QVector3D{0, 0, 1}* width_),
            QVector3D(QVector3D{0, 0, 1}* width_ +
                QVector3D{1, 0, 0}* length_),
            QVector3D(0, 0, 0),
            QVector3D(QVector3D{1, 0, 0}* length_),
        },
        color_,
        brush_,
        {},
        BaseParent()
    };

    Quadrangle face6 = face5;
    face6.Move(QVector3D{0, 1, 0} * height_);

    AppendVector(result, face1.Triangulate());
    AppendVector(result, face2.Triangulate());
    AppendVector(result, face3.Triangulate());
    AppendVector(result, face4.Triangulate());
    AppendVector(result, face5.Triangulate());
    AppendVector(result, face6.Triangulate());
    return result;
}


QVector3D Box::Center() const {
    return QVector3D(length_ / 2, height_ / 2,
width_ / 2);
}
```

Drill.cpp

```
#include "Drill.hpp"
#include "Box.hpp"
#include "Triangle.hpp"
#include "Switch.hpp"
#include "Frustum.hpp"
#include "Pyramid.hpp"
#include "Ellipsoid.hpp"

Drill::Drill(float battery_width,
             float battery_length,
             float battery_height,
             int switch_count,
             float handle_width,
             float handle_length,
             float handle_height,
             float trigger_length,
             float trigger_height,
             float workpart_length,
             float workpart_radius,
             float sphere_radius,
             float workpart2_length,
             float workpart2_radius,
             float drill_length,
             float drill_radius,
             std::optional<QRgb> color,
             std::optional<QRgb> brush,
             QVector3D base_point,
             Shape* parent)
    :Shape (base_point,
             color,
             brush,
             parent),
battery_width_{battery_width},
battery_length_{battery_length},
battery_height_{battery_height},
  switch_count_{switch_count},
  handle_width_{handle_width},
  handle_length_{handle_length},
  handle_height_{handle_height},
  trigger_length_{trigger_length},
  trigger_height_{trigger_height},
  workpart_length_{workpart_length},
  workpart_radius_{workpart_radius},
  sphere_radius_{sphere_radius},
  workpart2_length_{workpart2_length},
  workpart2_radius_{workpart2_radius},
  drill_length_{drill_length},
  drill_radius_{drill_radius}
{
    ;
}
void Drill::DrawShape(Painter &painter)
{
    Box block({}, battery_width_,
battery_height_, battery_length_, color_, brush_,
BaseParent());
    int padding = 5;
    int switch_height =
std::min(battery_height_/2,
                                (battery_length_
- padding * (switch_count_+1))/switch_count_ *
2);
    int switch_width = switch_height/2;
    block.Draw(painter);
```

```
    for (int i = 0; i < switch_count_; i++){
    Switch switch1(switch_width, switch_height,
color_, brush_,
                   {(float)battery_width_ -
switch_width + 2,
                    (float)switch_width * i +
padding * (i+1),
                    (float)switch_height/2},
                   BaseParent());
    switch1.Draw(painter);
    }


    Frustum handle(handle_width_, handle_length_,
               handle_width_, handle_length_,
               handle_height_, color_, brush_,
               {battery_width_/2.0f,
                battery_length_/3.0f,
                (float)battery_height_},
               {}, {}, {}, {}, BaseParent());
    handle.Draw(painter);


    float dy = 0;
    float dz = (float)battery_height_ +
            handle_height_ +
            std::min(workpart_radius_,
sphere_radius_);
    float dx = battery_width_/2.0f;

    Pyramid trigger(trigger_length_,
trigger_length_, trigger_height_,
               color_, SGL::red,
               {dx - trigger_length_/2,
                (float) (handle.BasePoint().y() +
handle_length_),
                (float)handle_height_ +
battery_height_ - trigger_height_},
               trigger_length_, trigger_length_ +
5, {}, {}, {}, {}, BaseParent());

    trigger.Draw(painter);

    dy += workpart_length_;
    Frustum workpart(workpart_radius_,
workpart_radius_,
                     sphere_radius_,
sphere_radius_,
                     workpart_length_, color_,
brush_,
                     {dx,
                      dy,
                      dz},
                     {}, {}, {}, {},
BaseParent());
    workpart.Rotate(90 ,{1, 0, 0, 0});
    workpart.Draw(painter);
    Ellipsoid sphere(sphere_radius_,
sphere_radius_, sphere_radius_,
                 color_, brush_,
                 {dx,
                  0.0f,
                  dz},
                 BaseParent(),
                 0.5);
    sphere.Rotate(90, {1, 0, 0, 0});
    sphere.Draw(painter);

    int between_workparts1_length = 15;
    dy += between_workparts1_length;
    Frustum between_workparts1((workpart2_radius_
+ workpart_radius_)/2,
                               (workpart2_radius_
+ workpart_radius_)/2,
                     workpart_radius_,
workpart_radius_,
                     between_workparts1_length,
color_, brush_,
                     {dx,
                      dy,
                      dz},
                     {}, {}, {}, {},
BaseParent());
    between_workparts1.Rotate(90 , {1, 0, 0, 0});
    between_workparts1.Draw(painter);

    dy += workpart2_length_;

    Frustum workpart2(workpart2_radius_,
workpart2_radius_,
                     workpart2_radius_,
workpart2_radius_,
                     workpart2_length_, color_,
brush_,
                     {dx,
                      dy,
                      dz},
                     {}, {}, {}, {},
BaseParent());
    workpart2.Rotate(90 , {1, 0, 0, 0});
    workpart2.Draw(painter);

    int between_workparts2_length = 15;
    int workpart3_radius = 10;
    dy += between_workparts2_length;
    Frustum between_workparts2((workpart2_radius_
+ workpart3_radius)/2,
                               (workpart2_radius_
+ workpart3_radius)/2,
                     workpart2_radius_,
workpart2_radius_,

between_workparts2_length, color_, {brush_},
                     {dx,
                      dy,
                      dz},
                     {}, {}, {}, {},
BaseParent());
    between_workparts2.Rotate(90 , {1, 0, 0, 0});
    between_workparts2.Draw(painter);

    int workpart3_length = 3;
    dy += workpart3_length;

    Frustum workpart3(workpart3_radius,
workpart3_radius,
                     workpart3_radius,
workpart3_radius,
                     workpart3_length, color_,
brush_,
                     {dx,
                      dy,
                      dz},
                     {}, {}, {}, {},
BaseParent());
    workpart3.Rotate(90 , {1, 0, 0, 0});
    workpart3.Draw(painter);

    int between_workparts3_length = 7;
    int between_workparts3_radius =
(drill_radius_ + workpart3_radius) / 2;
    dy += between_workparts3_length;
    Frustum
between_workparts3(between_workparts3_radius,

between_workparts3_radius,
                     workpart3_radius,
workpart3_radius,

between_workparts3_length, color_, {brush_},
                     {dx,
                      dy,
```

```
                     dz},
                    {}, {}, {}, {},
BaseParent());
    between_workparts3.Rotate(90 , {1, 0, 0, 0});
    between_workparts3.Draw(painter);

    int between_workparts4_length = 3;
    int between_workparts4_radius =
(drill_radius_ + between_workparts3_radius)/2;
    int between_workparts4_radius2 =
(drill_radius_ + between_workparts4_radius)/2;
    dy += between_workparts4_length;
    Frustum
between_workparts4(between_workparts4_radius2,

between_workparts4_radius2,
                    between_workparts4_radius,
between_workparts4_radius,

between_workparts4_length, color_, {brush_},
                    {dx,
                     dy,
                     dz},
                    {}, {}, {}, {},
BaseParent());
    between_workparts4.Rotate(90 , {1, 0, 0, 0});
    between_workparts4.Draw(painter);


    dy += drill_length_;
    Frustum drill(drill_radius_, drill_radius_,
                    drill_radius_,
drill_radius_,
                    drill_length_, color_,
brush_,
                    {dx,
                     dy,
                     dz},
                    {}, {}, {}, {},
BaseParent());
    drill.Rotate(90 , {1, 0, 0, 0});
    drill.Draw(painter);
}

QVector3D Drill::Center() const{

    return {battery_width_/2.0f,
    battery_length_/3.0f,
    (float)battery_height_ +
    handle_height_ +
    std::min(workpart_radius_, sphere_radius_)};
}

std::vector<Triangle> Drill::Triangulate(){
    return std::vector<Triangle> ();
}

QJsonObject Drill::ToJson() const {
  QJsonObject json;
  json["Тип"] = "Дрель";
  json["Shape"] = Shape::ToJson();
  json["Ширина батареи"] = battery_width_;
  json["Длина батареи"] = battery_length_;
  json["Высота батареи"] = battery_height_;
  json["Количество переключателей"] =
switch_count_;
  json["Радиус ручки (по x)"] = handle_width_;
  json["Радиус ручки (по y)"] = handle_length_;
  json["Высота ручки"] = handle_height_;
  json["Длина курка"] = trigger_length_;
  json["Высота курка"] = trigger_height_;
  json["Радиус корпуса"] = workpart_radius_;
  json["Длина корпуса"] = workpart_length_;
  json["Радиус сферы"] = sphere_radius_;
  json["Радиус рабочей части"] =
workpart2_radius_;
  json["Длина рабочей части"] =
workpart2_length_;
  json["Длина сверла"] = drill_length_;
  json["Радиус сверла"] = drill_radius_;
  return json;
}


void Drill::FromJson(const QJsonObject& json) {
  Shape::FromJson(json["Shape"].toObject());
  battery_width_ = json["Ширина
батареи"].toDouble();
  battery_length_ = json["Длина
батареи"].toDouble();
   battery_height_ = json["Высота
батареи"].toDouble();
   switch_count_ = json["Количество
переключателей"].toInt();
   handle_width_ = json["Радиус ручки (по
x)"].toDouble();
   handle_length_ = json["Радиус ручки (по
y)"].toDouble();
   handle_height_ = json["Высота
ручки"].toDouble();
   trigger_length_ = json["Длина
курка"].toDouble();
   trigger_height_ = json["Высота
курка"].toDouble();
   workpart_radius_ = json["Радиус
корпуса"].toDouble();
   workpart_length_ = json["Длина
корпуса"].toDouble();
   sphere_radius_ = json["Радиус
сферы"].toDouble();
   workpart2_radius_ = json["Радиус рабочей
части"].toDouble();
   workpart2_length_  = json["Длина рабочей
части"].toDouble();
   drill_length_ = json["Длина
сверла"].toDouble();
   drill_radius_ = json["Радиус
сверла"].toDouble();
}


QString Drill::type() {
  return "Drill";
}



Ellipse.cpp

#include "Ellipse.hpp"
#include "Triangle.hpp"
#include "Quadrangle.hpp"


QVector3D Ellipse::Center() const {
  return QVector3D(0, 0, 0);
}


std::vector<Triangle> Ellipse::Triangulate() {
  return std::vector<Triangle>();
}


void Ellipse::DrawShape(Painter& painter) {
  constexpr float ALPHA_STEP = 30;
  for (int i = 0; i < 360; i += ALPHA_STEP) {
    float angle = qDegreesToRadians(float(i));
    float angle_two = qDegreesToRadians(float(i +
ALPHA_STEP));
    QVector3D first_point = QVector3D(a_far_ *
cos(angle),
```

```
                                        b_far_ *
sin(angle), 0);
    QVector3D second_point = QVector3D(a_far_ *
cos(angle_two),
                                         b_far_ *
sin(angle_two), 0);
    QVector3D third_point = QVector3D(a_near_ *
cos(angle),
                                        b_near_ *
sin(angle), 0);
    QVector3D fourth_point = QVector3D(a_near_ *
cos(angle_two),
                                         b_near_ *
sin(angle_two), 0);

    Quadrangle q({first_point, second_point,
third_point, fourth_point},
                 std::nullopt,
                 brush_, {}, BaseParent());
    painter.Color(std::nullopt);
    q.Draw(painter);
    painter.Color(color_);
    painter.DrawLine(first_point, second_point,
BaseParent());
    if (a_near_ != 0) {
      painter.DrawLine(third_point, fourth_point,
BaseParent());
    }
  }
}


[[maybe_unused]] Ellipse::Ellipse(float a_far,
                                  float b_far,

std::optional<QRgb> color,

std::optional<QRgb> brush,
                                  QVector3D
base_point,
                                  float a_near,
                                  float b_near)
: Shape(base_point, color, brush)
, a_near_{a_near}
, b_near_{b_near}
, a_far_{a_far}
, b_far_{b_far} {

}

Ellipsoid.cpp //

#include <Box.hpp>
#include "Ellipsoid.hpp"
#include "Triangle.hpp"


QVector3D Ellipsoid::Center() const {
  return QVector3D(0, 0, 0);
}



const std::vector<Triangle>&
Ellipsoid::MyTriangulate() {
  if (a_old_ == a_ &&
  b_old_ == b_ &&
  c_old_ == c_ &&
  !triangles_.empty()) {
    return triangles_;
  }
  a_old_ = a_;
  b_old_ = b_;
  c_old_ = c_;
  triangles_.clear();
  constexpr float TETA_STEP = 15;
  constexpr float FI_STEP = 15;
  constexpr int FI_POINT_COUNT = 360 / FI_STEP;
  QVector3D first_points[FI_POINT_COUNT];
  QVector3D second_points[FI_POINT_COUNT];
  for (float teta = 0; teta < 180 * part_; teta
+= TETA_STEP) {
    float teta_angle = qDegreesToRadians(teta);
    float teta_next = qDegreesToRadians(teta +
TETA_STEP);
    for (float fi = 0; fi < 360; fi += FI_STEP) {
      float fi_angle = qDegreesToRadians(fi);
      first_points[int(fi / FI_STEP)] =
QVector3D(a_ * sin(teta_angle) * cos(fi_angle),

b_ * sin(teta_angle) * sin(fi_angle),

c_ * cos(teta_angle));

      second_points[int(fi / FI_STEP)] =
QVector3D(a_ * sin(teta_next) * cos(fi_angle),

b_ * sin(teta_next) * sin(fi_angle),

c_ * cos(teta_next));
    }

    for (int i = 0; i < std::size(first_points) -
1; ++i) {
      Triangle t1{first_points[i],
second_points[i], first_points[i + 1],
std::nullopt,
                  std::nullopt, {},
BaseParent()};
      Triangle t2{second_points[i],
second_points[i + 1], first_points[i + 1],
std::nullopt,
                  std::nullopt, {},
BaseParent()};
      triangles_.push_back(t1);
      triangles_.push_back(t2);
    }
    Triangle
t1{first_points[std::size(first_points) - 1],
second_points[std::size(first_points) - 1],
first_points[0], std::nullopt,
               std::nullopt, {}, BaseParent()};
    Triangle
t2{second_points[std::size(first_points) - 1],
second_points[0], first_points[0], std::nullopt,
               std::nullopt, {}, BaseParent()};
    triangles_.push_back(t1);
    triangles_.push_back(t2);
  }
  return triangles_;
}


void Ellipsoid::DrawShape(Painter& painter) {
  const auto& triangle_ = MyTriangulate();
  painter.Brush(brush_);
  painter.Color(color_);
  for (Triangle t : triangle_) {;
    t.Draw(painter);
  }
}


Ellipsoid::Ellipsoid(float a, float b, float c,
               std::optional<QRgb> color,
               std::optional<QRgb> brush,
               QVector3D base_point,
               Shape* parent,
                    float part)
: Shape{base_point, color, brush, parent}
, a_{a}
```

```
, b_{b}
, c_{c}
, a_old_{a}
, b_old_{b}
, c_old_{c}
, part_{part}{

}


std::vector<Triangle> Ellipsoid::Triangulate() {
  return std::vector<Triangle>();
}
```

Frustum.cpp

```
#include "Quadrangle.hpp"
#include "Frustum.hpp"
#include "Triangle.hpp"


Frustum::Frustum(float a_far_bottom,
                                 float
b_far_bottom,
                                 float a_far_top,
                                 float b_far_top,
                                 float height,

std::optional<QRgb> color,

std::optional<QRgb> brush,
                                 QVector3D
base_point,
                                 float
a_near_bottom,
                                 float
b_near_bottom,
                                 float
a_near_top,
                                 float
b_near_top,
                                 Shape* parent)
: Shape{base_point, color, brush, parent}
, a_far_bottom_{a_far_bottom}
, a_near_bottom_{a_near_bottom}
, b_far_bottom_{b_far_bottom}
, b_near_bottom_{b_near_bottom}
, a_far_top_{a_far_top}
, a_near_top_{a_near_top}
, b_far_top_{b_far_top}
, b_near_top_{b_near_top}
, height_{height} {

}


QVector3D Frustum::Center() const {
  return QVector3D(0, 0, height_ / 2);
}


std::vector<Triangle> Frustum::Triangulate() {
  return std::vector<Triangle>();
}


void Frustum::DrawShape(Painter& painter) {
  constexpr int ANGLE_STEP = 15;
  for (int i = 0; i < 360; i += ANGLE_STEP) {
    float angle = qDegreesToRadians(float(i));
    float angle_two = qDegreesToRadians(float(i +
ANGLE_STEP));
    QVector3D bottom_first_near =
QVector3D(a_near_bottom_ * cos(angle),

b_near_bottom_ * sin(angle),
                                      0);
    QVector3D bottom_second_near =
QVector3D(a_near_bottom_ * cos(angle_two),

b_near_bottom_ * sin(angle_two),
                                      0);
    QVector3D bottom_first_far =
QVector3D(a_far_bottom_ * cos(angle),

b_far_bottom_ * sin(angle),
                                    0);
    QVector3D bottom_second_far =
QVector3D(a_far_bottom_ * cos(angle_two),

b_far_bottom_ * sin(angle_two),
                                     0);

    QVector3D top_first_near =
QVector3D(a_near_top_ * cos(angle),

b_near_top_ * sin(angle),

height_);
    QVector3D top_second_near =
QVector3D(a_near_top_ * cos(angle_two),

b_near_top_ * sin(angle_two),

height_);
    QVector3D top_first_far =
QVector3D(a_far_top_ * cos(angle),

b_far_top_ * sin(angle),

height_);
    QVector3D top_second_far =
QVector3D(a_far_top_ * cos(angle_two),

b_far_top_ * sin(angle_two),

height_);
    Quadrangle face_bottom(
        {
            bottom_first_near,
            bottom_second_near,
            bottom_first_far,
            bottom_second_far
        },std::nullopt,brush_, {}, BaseParent());

    Quadrangle face_top(
        {
            top_first_far,
            top_second_far,
            top_first_near,
            top_second_near,
        },std::nullopt,brush_, {}, BaseParent());

    Quadrangle face_far(
        {
            bottom_first_far,
            bottom_second_far,
            top_first_far,
            top_second_far
        },std::nullopt, brush_, {},
BaseParent());

    Quadrangle face_near(
        {top_first_near,
         top_second_near,
         bottom_first_near,
         bottom_second_near
        },std::nullopt, brush_, {},
BaseParent());

    painter.Color(std::nullopt);
    face_bottom.Draw(painter);
    face_top.Draw(painter);
```

```
    face_far.Draw(painter);
    face_near.Draw(painter);


    painter.Color(color_);
    if (i % 90 == 0) {
      painter.DrawLine(bottom_first_far,
top_first_far, BaseParent());
      if (a_near_top_ != 0 || a_near_bottom_ !=
0) {
        painter.DrawLine(bottom_first_near,
top_first_near, BaseParent());
      }
      if (a_near_bottom_ != 0) {
        painter.DrawLine(bottom_first_near,
bottom_first_far, BaseParent());
      }
      if (a_near_top_ != 0) {
        painter.DrawLine(top_first_near,
top_first_far, BaseParent());
      }
    }
    painter.DrawLine(bottom_first_far,
bottom_second_far, BaseParent());
    if (a_near_bottom_ != 0) {
      painter.DrawLine(bottom_first_near,
bottom_second_near, BaseParent());
    }
    painter.DrawLine(top_first_far,
top_second_far, BaseParent());
    if (a_near_top_ != 0) {
      painter.DrawLine(top_first_near,
top_second_near, BaseParent());
    }
  }
}
```

Line.cpp

```
#include "Line.hpp"
#include "Triangle.hpp"
#include "SGLMath.hpp"

void Line::DrawShape(Painter& painter) {
  painter.Color(color_);
  painter.DrawLine(begin_, end_, BaseParent());
}


std::vector<Triangle> Line::Triangulate() {
  return {Triangle{begin_, end_, (begin_ + end_)
/ 2}};
}


Line::Line(QVector3D begin, QVector3D end,
std::optional<QRgb> color, Shape* parent)
: Shape{{}, color, std::nullopt, parent}
, begin_{begin}
, end_{end} {
  ;
}

[[maybe_unused]] const QVector3D &Line::Begin()
const {
  return begin_;
}

[[maybe_unused]] void Line::Begin(const QVector3D
&begin) {
  begin_ = begin;
}

[[maybe_unused]] const QVector3D &Line::AnEnd()
const {
  return end_;
}

[[maybe_unused]] void Line::AnEnd(const QVector3D
&an_end) {
  end_ = an_end;
}

[[maybe_unused]] Line::Line(int x0, int y0, int
z0, int x1, int y1, int z1,
           std::optional<QRgb> color,
           Shape* parent)
: Line{QVector3D(x0, y0, z0), QVector3D(x1, y1,
z1), color, parent} {
  ;
}


QVector3D Line::Center() const {
  return QVector3D((begin_ + end_) / 2);
}


Pyramid.cpp

#include <Line.hpp>
#include "Pyramid.hpp"
#include "Triangle.hpp"
#include "Quadrangle.hpp"


Pyramid::Pyramid(float bottom_length,
                 float bottom_width,
                 float height,
                 std::optional<QRgb> color,
                 std::optional<QRgb> brush,
                 QVector3D base_point,
                 float top_length,
                 float top_width,
                 float bottom_near_length,
                 float bottom_near_width,
                 float top_near_length,
                 float top_near_width,
                 Shape* parent)
: Shape{base_point, color, brush, parent}
, bottom_length_{bottom_length}
, bottom_width_{bottom_width}
, height_{height}
, top_length_{top_length}
, top_width_{top_width}
, bottom_near_length_{bottom_near_length}
, bottom_near_width_{bottom_near_width}
, top_near_length_{top_near_length}
, top_near_width_{top_near_width} {

}


QVector3D Pyramid::Center() const {
  return QVector3D(bottom_length_ /2,
                   bottom_width_ / 2, height_ /
2);
}


std::vector<Triangle> Pyramid::Triangulate() {
  return std::vector<Triangle>();
}


void Pyramid::DrawShape(Painter& painter) {
  const QVector3D i{1, 0, 0};
  const QVector3D j{0, 1, 0};
  const QVector3D k{0, 0, 1};
  QVector3D bottom_left_up_far{0, 0, 0};
  QVector3D bottom_right_up_far =
bottom_left_up_far + i * bottom_length_;
  QVector3D bottom_left_down_far =
bottom_left_up_far + j * bottom_width_;
```

```
  QVector3D bottom_right_down_far =
bottom_left_down_far + i * bottom_length_;

  QVector3D bottom_left_up_near{(bottom_length_ -
bottom_near_length_) / 2,
                                (bottom_width_ -
bottom_near_width_) / 2,
                                0};
  QVector3D bottom_right_up_near =
bottom_left_up_near +
      i * bottom_near_length_;
  QVector3D bottom_left_down_near =
bottom_left_up_near +
      j * bottom_near_width_;
  QVector3D bottom_right_down_near =
bottom_left_down_near +
      i * bottom_near_length_;

  QVector3D top_left_up_far{(bottom_length_ -
top_length_) / 2,
                            (bottom_width_ -
top_width_) / 2, height_};
  QVector3D top_right_up_far = top_left_up_far +
i * top_length_;
  QVector3D top_left_down_far = top_left_up_far +
j * top_width_;
  QVector3D top_right_down_far =
top_left_down_far + i * top_length_;

  QVector3D top_left_up_near{(bottom_length_ -
top_near_length_) / 2,
                                (bottom_width_ -
top_near_width_) / 2,
                                height_};
  QVector3D top_right_up_near = top_left_up_near
+
      i * top_near_length_;
  QVector3D top_left_down_near = top_left_up_near
+
      j * top_near_width_;
  QVector3D top_right_down_near =
top_left_down_near +
      i * top_near_length_;

  Quadrangle bottom_left_base{
      {
          bottom_left_down_far,
          bottom_left_down_near,
          bottom_left_up_far,
          bottom_left_up_near
      }, std::nullopt, brush_, {}, BaseParent()
  };
  bottom_left_base.Draw(painter);

  Quadrangle bottom_up_base{
      {
          bottom_left_up_far,
          bottom_left_up_near,
          bottom_right_up_far,
          bottom_right_up_near
      }, std::nullopt, brush_, {}, BaseParent()
  };
  bottom_up_base.Draw(painter);

  Quadrangle bottom_right_base{
      {
          bottom_right_down_near,
          bottom_right_down_far,
          bottom_right_up_near,
          bottom_right_up_far
      }, std::nullopt, brush_, {}, BaseParent()
  };
  bottom_right_base.Draw(painter);

  Quadrangle bottom_down_base{
      {
          bottom_left_down_far,
          bottom_right_down_far,
          bottom_left_down_near,
          bottom_right_down_near
      }, std::nullopt, brush_, {}, BaseParent()
  };
  bottom_down_base.Draw(painter);

  Quadrangle top_left_base{
      {
          top_left_up_far,
          top_left_up_near,
          top_left_down_far,
          top_left_down_near
      }, std::nullopt, brush_, {}, BaseParent()
  };
  top_left_base.Draw(painter);

  Quadrangle top_up_base{
      {
          top_right_up_far,
          top_right_up_near,
          top_left_up_far,
          top_left_up_near
      }, std::nullopt, brush_, {}, BaseParent()
  };
  top_up_base.Draw(painter);

  Quadrangle top_right_base{
      {
          top_right_up_near,
          top_right_up_far,
          top_right_down_near,
          top_right_down_far
      }, std::nullopt, brush_, {}, BaseParent()
  };
  top_right_base.Draw(painter);

  Quadrangle top_down_base{
      {
          top_left_down_near,
          top_right_down_near,
          top_left_down_far,
          top_right_down_far
      }, std::nullopt, brush_, {}, BaseParent()
  };
  top_down_base.Draw(painter);

  Quadrangle left_far_face{
      {
          bottom_left_down_far,
          top_left_down_far,
          bottom_left_up_far,
          top_left_up_far
      }, std::nullopt, brush_, {}, BaseParent()
  };
  left_far_face.Draw(painter);

  Quadrangle up_far_face{
      {
          top_left_up_far,
          top_right_up_far,
          bottom_left_up_far,
          bottom_right_up_far
      }, std::nullopt, brush_, {}, BaseParent()
  };
  up_far_face.Draw(painter);

  Quadrangle right_far_face{
      {
          bottom_right_down_far,
          top_right_down_far,
          bottom_right_up_far,
          top_right_up_far
      }, std::nullopt, brush_, {}, BaseParent()
  };
```

```
  right_far_face.Draw(painter);

  Quadrangle down_far_face{
      {
          top_left_down_far,
          top_right_down_far,
          bottom_left_down_far,
          bottom_right_down_far
      }, std::nullopt, brush_, {}, BaseParent()
  };
  down_far_face.Draw(painter);

  Quadrangle left_near_face{
      {
          bottom_left_down_near,
          top_left_down_near,
          bottom_left_up_near,
          top_left_up_near
      }, std::nullopt, brush_, {}, BaseParent()
  };
  left_near_face.Draw(painter);

  Quadrangle up_near_face{
      {
          top_left_up_near,
          top_right_up_near,
          bottom_left_up_near,
          bottom_right_up_near
      }, std::nullopt, brush_, {}, BaseParent()
  };
  up_near_face.Draw(painter);

  Quadrangle right_near_face{
      {
          bottom_right_up_near,
          top_right_up_near,
          bottom_right_down_near,
          top_right_down_near
      }, std::nullopt, brush_, {}, BaseParent()
  };
  right_near_face.Draw(painter);

  Quadrangle down_near_face{
      {
          bottom_left_down_near,
          bottom_right_down_near,
          top_left_down_near,
          top_right_down_near
      }, std::nullopt, brush_, {}, BaseParent()
  };
  down_near_face.Draw(painter);

  painter.Color(color_);
  Line(bottom_left_up_far, bottom_right_up_far,
color_, BaseParent()).Draw(painter);
  Line(bottom_right_up_far,
bottom_right_down_far, color_,
BaseParent()).Draw(painter);
  Line(bottom_right_down_far,
bottom_left_down_far, color_,
BaseParent()).Draw(painter);
  Line(bottom_left_down_far, bottom_left_up_far,
color_, BaseParent()).Draw(painter);

  Line(bottom_left_up_far, top_left_up_far,
color_, BaseParent()).Draw(painter);
  Line(bottom_right_up_far, top_right_up_far,
color_, BaseParent()).Draw(painter);
  Line(bottom_left_down_far, top_left_down_far,
color_, BaseParent()).Draw(painter);
  Line(bottom_right_down_far, top_right_down_far,
color_, BaseParent()).Draw(painter);

  if (top_length_ != 0) {
    Line(top_left_up_far, top_right_up_far,
color_, BaseParent()).Draw(painter);
    Line(top_right_up_far, top_right_down_far,
color_, BaseParent()).Draw(painter);
    Line(top_right_down_far, top_left_down_far,
color_, BaseParent()).Draw(painter);
    Line(top_left_down_far, top_left_up_far,
color_, BaseParent()).Draw(painter);
  }

  if (bottom_near_length_ != 0) {
    Line(bottom_left_up_near,
bottom_right_up_near, color_,
BaseParent()).Draw(painter);
    Line(bottom_right_up_near,
bottom_right_down_near, color_,
BaseParent()).Draw(painter);
    Line(bottom_right_down_near,
bottom_left_down_near, color_,
BaseParent()).Draw(painter);
    Line(bottom_left_down_near,
bottom_left_up_near, color_,
BaseParent()).Draw(painter);
  }

  if (top_near_length_ != 0) {
    Line(top_left_up_near, top_right_up_near,
color_, BaseParent()).Draw(painter);
    Line(top_right_up_near, top_right_down_near,
color_, BaseParent()).Draw(painter);
    Line(top_right_down_near, top_left_down_near,
color_, BaseParent()).Draw(painter);
    Line(top_left_down_near, top_left_up_near,
color_, BaseParent()).Draw(painter);
  }

  if (top_near_length_ != 0 ||
bottom_near_length_ != 0) {
    Line(bottom_left_up_near, top_left_up_near,
color_, BaseParent()).Draw(painter);
    Line(bottom_right_up_near, top_right_up_near,
color_, BaseParent()).Draw(painter);
    Line(bottom_left_down_near,
top_left_down_near, color_,
BaseParent()).Draw(painter);
    Line(bottom_right_down_near,
top_right_down_near, color_,
BaseParent()).Draw(painter);
  }
}
```

Quadrangle.cpp

```
#include "Quadrangle.hpp"
#include "Triangle.hpp"
#include "SGLMath.hpp"
#include <vector>
#include <Line.hpp>


void Quadrangle::DrawShape(Painter& painter) {
  painter.Brush(brush_);
  painter.Color(std::nullopt);
  for (Triangle triangle : Triangulate()) {
    triangle.Draw(painter);
  }

  painter.Color(color_);
  Line(vertexes_[0],    vertexes_[1],    color_,
BaseParent()).Draw(painter);
  Line(vertexes_[1],    vertexes_[3],    color_,
BaseParent()).Draw(painter);
  Line(vertexes_[3],    vertexes_[2],    color_,
BaseParent()).Draw(painter);
  Line(vertexes_[2],    vertexes_[0],    color_,
BaseParent()).Draw(painter);
}


std::vector<Triangle> Quadrangle::Triangulate() {
  std::vector<Triangle> result;
  result.emplace_back(vertexes_[0], vertexes_[1],
vertexes_[2], std::nullopt,
                      brush_, QVector3D{},
                      BaseParent());
```

```
  result.emplace_back(vertexes_[3], vertexes_[2],
vertexes_[1], std::nullopt,
                      brush_, QVector3D{},
                      BaseParent());
  return result;
}


Quadrangle::Quadrangle(std::array<QVector3D,   4>
vertexes,
                       std::optional<QRgb> color,
                       std::optional<QRgb> brush,
                       QVector3D base_point,
                       Shape* parent)
: Shape(base_point, color, brush, parent) {
  for (int i = 0; i < vertexes.size(); ++i) {
    vertexes_[i] = vertexes[i];
  }
}


QVector3D Quadrangle::Center() const {
  QVector3D sum_vertex;
  for (QVector3D vertex : vertexes_) {
    sum_vertex += vertex;
  }
  return sum_vertex / 4;
}
Shape.cpp

#include "Shape.hpp"
#include "SGLMath.hpp"
#include "Painter.hpp"
#include "Triangle.hpp"



[[maybe_unused]] void
Shape::AppendModify(QMatrix4x4 modify_matrix)
noexcept {
  modify_matrix_ = modify_matrix *
modify_matrix_;
}


void Shape::Rotate(float a, QVector4D v) {
  auto modify_matrix = SGLMath::Rotate(a,  v);
  modify_matrix_ = modify_matrix *
modify_matrix_;
  i = (modify_matrix * i).normalized();
  j = (modify_matrix * j).normalized();
  k = (modify_matrix * k).normalized();
}


void Shape::MoveBasePoint(QVector4D v) {
  base_point_ = SGLMath::Move(v) * base_point_;
}


void Shape::Rotate(float a, QVector4D v,
QVector3D pos) {
  auto modify_matrix = SGLMath::Rotate(a, v,
pos);
  modify_matrix_ = modify_matrix *
modify_matrix_;
  i = (modify_matrix * i).normalized();
  j = (modify_matrix * j).normalized();
  k = (modify_matrix * k).normalized();
}


const QVector3D& Shape::BasePoint() const {
  return base_point_;
}


void Shape::BasePoint(const QVector3D&
base_point) {
  base_point_ = base_point;
}


const std::optional<QRgb> &Shape::Color() const {
  return color_;
}


void Shape::Color(const std::optional<QRgb>
&color) {
  color_ = color;
}


const std::optional<QRgb> &Shape::Brush() const {
  return brush_;
}


void Shape::Brush(const std::optional<QRgb>
&brush) {
  brush_ = brush;
}


void Shape::Draw(Painter& painter) {

painter.AppendTransform(SGLMath::Move(base_point_
));
  painter.AppendTransform(modify_matrix_);
  DrawShape(painter);

painter.AppendTransform(modify_matrix_.inverted()
);
  painter.AppendTransform(SGLMath::Move(-
base_point_));
}

[[maybe_unused]] const QVector4D &Shape::I()
const {
  return i;
}


const QVector4D &Shape::J() const {
  return j;
}


const QVector4D &Shape::K() const {
  return k;
}


void Shape::Move(QVector4D v) {
  modify_matrix_ = SGLMath::Move(v) *
modify_matrix_;
}


Shape* Shape::BaseParent() noexcept {
  if (parent_ == nullptr) {
    return this;
  } else {
    return parent_->BaseParent();
  }
}

void Shape::Scale(float a, float b, float c) {
  modify_matrix_ =
      modify_matrix_ *
      SGLMath::Move(Center()) *
      SGLMath::Scale(a, b, c) *
      SGLMath::Move(-Center());
}


const QMatrix4x4& Shape::ModifyMatrix() const {
```

```
  return modify_matrix_;
}


void Shape::ModifyMatrix(const QMatrix4x4&
modify_matrix) {
  modify_matrix_ = modify_matrix;
}


QJsonObject Shape::ToJson() const {
  QJsonObject json;
  if (color_.has_value()) {
    QJsonObject json_color;
    json_color["Красный"] = qRed(color_.value());
    json_color["Зелёный"] =
qGreen(color_.value());
    json_color["Синий"] = qBlue(color_.value());
    json["Цвет"] = json_color;
  } else {
    json["Цвет"] = "Без цвета";
  }
  if (brush_.has_value()) {
    QJsonObject json_brush;
    json_brush["Красный"] = qRed(brush_.value());
    json_brush["Зелёный"] =
qGreen(brush_.value());
    json_brush["Синий"] = qBlue(brush_.value());
    json["Зарисовка"] = json_brush;
  } else {
    json["Зарисовка"] = "Без зарисовки";
  }
  json["Матрица модификаций"] =
SGLMath::ToJson(modify_matrix_);
  json["Базовая точка"] =
SGLMath::ToJson(base_point_);
  json["i"] = SGLMath::ToJson(i);
  json["j"] = SGLMath::ToJson(j);
  json["k"] = SGLMath::ToJson(k);
  return json;
}


void Shape::FromJson(const QJsonObject& json) {
  i = SGLMath::ToVector4D(json["i"].toArray());
  j = SGLMath::ToVector4D(json["j"].toArray());
  k = SGLMath::ToVector4D(json["k"].toArray());

  base_point_ =
QVector3D(SGLMath::ToVector4D(json["Базовая
точка"].toArray()));
  if (json["Зарисовка"].toString() != "Без
зарисовки") {
    QJsonObject brush =
json["Зарисовка"].toObject();
    brush_ = qRgb(brush["Красный"].toInt(),
                  brush["Зелёный"].toInt(),
                  brush["Синий"].toInt());
  } else {
    brush_ = std::nullopt;
  }

  if (json["Цвет"].toString() != "Без цвета") {
    QJsonObject color = json["Цвет"].toObject();
    color_ = qRgb(color["Красный"].toInt(),
                  color["Зелёный"].toInt(),
                  color["Синий"].toInt());
  } else {
    color_ = std::nullopt;
  }

  modify_matrix_ =
SGLMath::ToMatrix4x4(json["Матрица
модификаций"].toArray());
}
```

```
QString Shape::type() {
  return "Shape";
}
```

Switch.cpp

```
#include "Switch.hpp"
#include "Box.hpp"
#include "Triangle.hpp"
#include "Quadrangle.hpp"
#include "Pyramid.hpp"

Switch::Switch(int width,
               int height,
               std::optional<QRgb> color,
              std::optional<QRgb> brush,
              QVector3D base_point,
              Shape* parent)
    :Shape (base_point,
              color,
              brush,
              parent),
width_{width},
height_{height}{
    ;
}



void Switch::DrawShape(Painter &painter)
{
    Box block(QVector3D(0, 0, 0),
              width_,
              height_ + height_/5.f,
              width_ + width_/5.f,
              color_, brush_, this);

    Pyramid p1(width_, width_, height_,
SGL::black, SGL::red,
               {(width_/3.f), (width_/10.f),
(height_/10.f)}, 0, width_, {}, {}, {}, {},
this);
    Pyramid p2(width_, width_, height_,
SGL::black, SGL::red,
               {(width_/3.f), (width_/10.f),
(height_/10.f)}, 0, width_, {}, {}, {}, {},
this);
    p2.Rotate(180, {0, 1, 0, 0}, p1.Center());
    p2.Rotate(10, {0, 1, 0, 0}, p1.Center());
    p1.Rotate(10, {0, 1, 0, 0}, p1.Center());

    p1.Draw(painter);
    p2.Draw(painter);
    block.Draw(painter);

}

QVector3D Switch::Center() const{
    return QVector3D(width_ /2, width_ / 2,
height_ / 2);
}

std::vector<Triangle> Switch::Triangulate(){
    return std::vector<Triangle> ();
}
```

Triangle.cpp

```
#include "Triangle.hpp"
#include "Painter.hpp"
#include "SGLMath.hpp"
#include <vector>


void Triangle::DrawShape(Painter& painter) {
```

```
  if (!painter.Color().has_value()) {
    painter.Color(color_);
  }
  if (!painter.Brush().has_value()) {
    painter.Brush(brush_);
  }
  painter.DrawTriangle(points_, BaseParent());
}


std::vector<Triangle> Triangle::Triangulate() {
  return {*this};
}


Triangle::Triangle(std::array<QVector3D, 3>
points,
                   std::optional<QRgb> color,
                   std::optional<QRgb> brush,
                   QVector3D base_point,
                   Shape* parent)
: Shape{base_point, color, brush, parent}
, points_{points} {

}


Triangle::Triangle(QVector3D a, QVector3D b,
QVector3D c,
                   std::optional<QRgb> color,
                   std::optional<QRgb> brush,
                   QVector3D base_point,
                   Shape* parent)
: Triangle({a, b, c}, color, brush, base_point,
parent) {

}

[[maybe_unused]] const std::array<QVector3D, 3>&
Triangle::Points() const {
  return points_;
}


QVector3D Triangle::Center() const {
  return (points_[0] + points_[1] + points_[2]) /
3;
}
```

mian.cpp

```
#include <QApplication>
#include "mainwindow.hpp"


int main(int argc, char *argv[]) {
  QApplication a(argc, argv);
  MainWindow w;
  w.show();
  return QApplication::exec();
}
```

mainwindow.cpp

```
// You may need to build the project (run Qt uic
code generator) to get "ui_MainWindow.h" resolved

#include <Box.hpp>
#include <SGLMath.hpp>
#include "mainwindow.hpp"
#include "ui_MainWindow.h"
#include <QFileDialog>
#include <Line.hpp>
#include "Triangle.hpp"
#include "Drill.hpp"


void SaveJson(const QJsonDocument& document,
              const QString& filename) {
  QFile file(filename);
  file.open(QFile::WriteOnly);
  file.write(document.toJson());
}


QJsonDocument LoadJson(const QString& filename) {
  QFile file(filename);
  file.open(QFile::ReadOnly);
  return QJsonDocument::fromJson(file.readAll());
}


MainWindow::MainWindow(QWidget* parent)
: QWidget(parent)
, ui(new Ui::MainWindow) {
  ui->setupUi(this);
  SetupUi();

  scene_.PushShape(new Drill(70, 125, 50,
                             5,
                             20, 20, 100,
                             10, 10,
                             150, 30, 25,
                             30, 15,
                             50, 1, SGL::black,
SGL::gray));

  ui->graphics_view->Scene(&scene_);
}


MainWindow::~MainWindow() {
  delete ui;
}


void MainWindow::keyPressEvent(QKeyEvent* event)
{
  QWidget::keyPressEvent(event);
}


void SystemCoordinates(GraphicsScene& scene) {
  scene.PushShape(new Line({-300, 0, 0},
                           QVector3D{300, 0, 0},
                           SGL::red));
  scene.PushShape(new Triangle({300, 0, 0},
                               QVector3D{300 -
30, -10, 0},
                               QVector3D{300 -
30, 10, 0}, SGL::white, SGL::red));

  scene.PushShape(new Line({0, -300, 0},
                           QVector3D{0, 300, 0},
                           SGL::green));
  scene.PushShape(new Triangle({0, 300, 0},
                               QVector3D{0, 300 -
30, -10},
                               QVector3D{0, 300 -
30, 10}, SGL::white, SGL::green));

  scene.PushShape(new Line({0, 0, -300},
                           QVector3D{0, 0, 300},
                           SGL::blue));
  scene.PushShape(new Triangle({0, 0, 300},
                               QVector3D{-10, 0,
300 - 30},
                               QVector3D{10, 0,
300 - 30}, SGL::white, SGL::blue));
}


void MainWindow::SetupUi() {
  // SystemCoordinates(scene_);
```

```
  ui->central->setChecked(true);
  ui->graphics_view->SetCentral();

  ui->polygon_view->setChecked(true);
  ui->graphics_view->SetPolygonView();

  connect(ui->central, &QRadioButton::clicked,
[this](bool clicked) {
    ui->graphics_view->SetCentral();
    ui->graphics_view->Render();
  });
  connect(ui->parallels, &QRadioButton::clicked,
[this](bool clicked) {
    ui->graphics_view->SetParallels();
    ui->graphics_view->Render();
  });
  connect(ui->polygon_view,
&QRadioButton::clicked, [this](bool clicked) {
    ui->graphics_view->SetPolygonView();
    ui->graphics_view->Render();
  });
  connect(ui->line_view, &QRadioButton::clicked,
[this](bool clicked) {
    ui->graphics_view->SetLineView();
    ui->graphics_view->Render();
  });
  connect(ui->panoram_start,
&QPushButton::clicked,
          this, &MainWindow::PanoramStart);

  connect(ui->push_shape_2,
&QPushButton::clicked,
          [this]() {
      scene_.PushShape(new Drill(ui-
>battery_width_sb->value(),
                                 ui-
>battery_length_sb->value(),
                                 ui-
>battery_height_sb->value(),
                                 ui-
>switch_count_sb->value(),
                                 ui-
>handle_width_sb->value(),
                                 ui-
>handle_length_sb->value(),
                                 ui-
>handle_height_sb->value(),
                                 ui-
>trigger_length_sb->value(),
                                 ui-
>trigger_height_sb->value(),
                                 ui-
>workpart_length_sb->value(),
                                 ui-
>workpart_radius_sb ->value(),
                                 ui-
>sphere_radius_sb->value(),
                                 ui-
>workpart2_length_sb->value(),
                                 ui-
>workpart2_radius_sb->value(),
                                 ui-
>drill_length_sb->value(),
                                 ui-
>drill_radius_sb->value(),
                                 SGL::black,
                                 SGL::gray,
                                 QVector3D(ui-
>shape_x_2->value(),
                                           ui-
>shape_y_2->value(),
                                           ui-
>shape_z_2->value())));

  });

  connect(ui->remove_shape_2,
&QPushButton::clicked, [this]() {
    for (auto[shape, brush] : ui->graphics_view-
>SelectedShapes()) {
      scene_.EraseShape(shape);
    }
    ui->graphics_view->ClearSelected();
  });

  connect(ui->change_shape_2,
&QPushButton::clicked, [this]() {
    for (auto[shape, brush] : ui->graphics_view-
>SelectedShapes()) {
      auto drill = dynamic_cast<Drill*>(shape);
      auto modify_matrix = drill->ModifyMatrix();
      *drill = Drill(ui->battery_width_sb-
>value(),
                     ui->battery_length_sb-
>value(),
                     ui->battery_height_sb-
>value(),
                     ui->switch_count_sb-
>value(),
                     ui->handle_width_sb-
>value(),
                     ui->handle_length_sb-
>value(),
                     ui->handle_height_sb-
>value(),
                     ui->trigger_length_sb-
>value(),
                     ui->trigger_height_sb-
>value(),
                     ui->workpart_length_sb-
>value(),
                     ui->workpart_radius_sb -
>value(),
                     ui->sphere_radius_sb-
>value(),
                     ui->workpart2_length_sb-
>value(),
                     ui->workpart2_radius_sb-
>value(),
                     ui->drill_length_sb-
>value(),
                     ui->drill_radius_sb-
>value(),
                     drill->Color(),
                     drill->Brush(),
                     QVector3D(ui->shape_x_2-
>value(),
                               ui->shape_y_2-
>value(),
                               ui->shape_z_2-
>value()));
      drill->ModifyMatrix(modify_matrix);
    }
  });

  connect(ui->save_model, &QPushButton::clicked,
[this]() {
    auto filename =
QFileDialog::getSaveFileName(nullptr,
                                 "Выберите файл
для сохранения");
    if (filename.isNull()) {
      return;
    }
    ui->graphics_view->UnselectedAllShape();
    QJsonDocument document ;
    QJsonObject json;
    QJsonObject json_camera;
    auto camera = dynamic_cast<LookAtCamera*>(ui-
>graphics_view->Camera());
    json_camera["Глаз"] = SGLMath::ToJson(camera-
>Eye());
```

```
    json_camera["Центр"] =
SGLMath::ToJson(camera->Center());
    json_camera["Up"] = SGLMath::ToJson(camera-
>Up());
    QJsonArray json_shapes;
    for (Shape* shape : scene_.Shapes()) {
      json_shapes.push_back(shape->ToJson());
    }
    json["Камера"] = json_camera;
    json["Фигуры"] = json_shapes;
    document.setObject(json);
    SaveJson(document, filename);
  });
  connect(ui->load_model, &QPushButton::clicked,
[this]() {
    auto filename =
QFileDialog::getOpenFileName(nullptr,

"Выберите файл с фигурами");
    if (filename.isNull()) {
      return;
    }
    QJsonDocument document = LoadJson(filename);
    QJsonObject json = document.object();
    auto camera = dynamic_cast<LookAtCamera*>(ui-
>graphics_view->Camera());
    assert(camera);
    QVector4D eye =
SGLMath::ToVector4D(json["Камера"]
        .toObject()["Глаз"].toArray());
    QVector4D center =
SGLMath::ToVector4D(json["Камера"]
        .toObject()["Центр"].toArray());
    QVector4D up =
SGLMath::ToVector4D(json["Камера"]

.toObject()["Up"].toArray());
    camera->Reset(QVector3D(eye),
QVector3D(center), QVector3D(up));

    QJsonArray json_shapes =
json["Фигуры"].toArray();
    scene_.Clear();
    for (QJsonValue value : json_shapes) {
      auto drill = new Drill;
      drill->FromJson(value.toObject());
      scene_.PushShape(drill);
    }
  });
}


void MainWindow::timerEvent(QTimerEvent* event) {
  ui->graphics_view->PanoramStep(QVector3D(ui-
>center_x->value(),
                                   ui->center_y-
>value(),
                                   ui->center_z-
>value()));
  QObject::timerEvent(event);
}


void MainWindow::PanoramStart() {
  ui->panoram_start->disconnect();
  ui->panoram_start->setText("Остановить
вращение\n камеры");
  timer_id_ = startTimer(60);
  connect(ui->panoram_start,
          &QPushButton::clicked,
          this, &MainWindow::PanoramStop);
}


void MainWindow::PanoramStop() {
  ui->panoram_start->disconnect();
  killTimer(timer_id_);
  ui->panoram_start->setText("Начать вращение\n
камеры");
  connect(ui->panoram_start,
          &QPushButton::clicked,
          this, &MainWindow::PanoramStart);
}
```