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»

Выполнил:

ст. гр. ПИ–18б

Моргунов А.Г.

Руководитель:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_кафедры \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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 С++.

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

СОДЕРЖАНИЕ

[Реферат 2](#_Toc73462554)

[Введение 5](#_Toc73462555)

[1 РАзработка полигональной модели объекта 6](#_Toc73462556)

[1.1 Составляющие элементы объекта 7](#_Toc73462557)

[1.2 Триангуляция поверхности объекта 7](#_Toc73462558)

[2 Описание выбранных методов и алгоритмов визуализации 9](#_Toc73462559)

[2.1 Алгоритмы афинных преобразований 9](#_Toc73462560)

[2.2 Алгоритм удаления скрытых линий 9](#_Toc73462561)

[2.3 Алгоритмы проекционных преобразований 9](#_Toc73462562)

[2.3.1 Алгоритм параллельного проецирования 10](#_Toc73462563)

[2.3.2 Алгоритм центрального проецирования 10](#_Toc73462564)

[2.4 Алгоритм произвольных видовых преобразований (камеры) 11](#_Toc73462565)

[3 Разработка структур данных для хранения и описания объекта 12](#_Toc73462566)

[3.1 Описание структур данных 12](#_Toc73462567)

[3.2 Диаграммы UML взаимодействия классов 12](#_Toc73462568)

[3.3 Реализация масштабирования 15](#_Toc73462569)

[3.4 Реализация поворота 16](#_Toc73462570)

[3.5 Реализация перемещения объекта 17](#_Toc73462571)

[3.6 Реализация изменения параметров 18](#_Toc73462572)

[3.7 Реализация параллельной проекции 19](#_Toc73462573)

[3.8 Реализация перспективной проекции 20](#_Toc73462574)

[3.9 Реализация произвольных видовых преобразований (камеры) 21](#_Toc73462575)

[3.10 Реализация сохранения сцены 23](#_Toc73462576)

[3.11 Реализация удаления скрытых линий 26](#_Toc73462577)

[4 Тестирование программы 27](#_Toc73462578)

[Выводы 28](#_Toc73462579)

[Перечень ссылок 29](#_Toc73462580)

[Приложение А. Техническое задание 30](#_Toc73462581)

[Приложение Б. Проверка оригинальности 34](#_Toc73462582)

[Приложение В. Экранные формы 35](#_Toc73462583)

[Приложение Г. Руководство пользователя 40](#_Toc73462584)

[Приложение Д. Листинг 41](#_Toc73462585)

# 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.

# 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

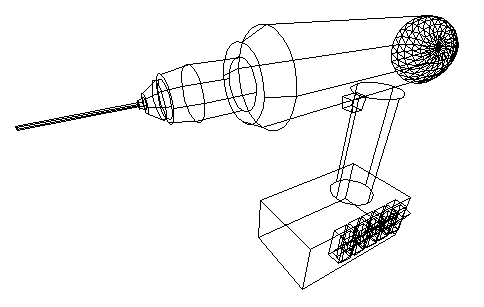


Рисунок 1.2 – Проволочное представление

## 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.

# 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 4х4.

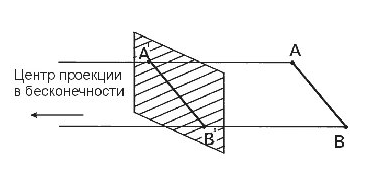


Рисунок 2.1 – Параллельная проекция

## 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)

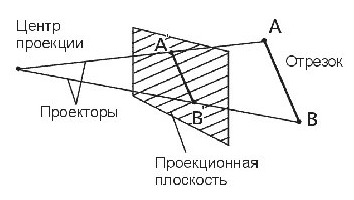


Рисунок 2.2 – Центральная проекция

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.

# 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.

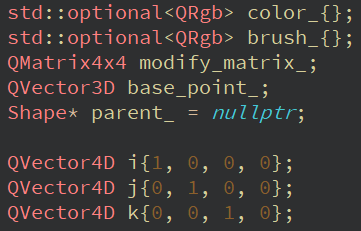


Рисунок 3.1 Поля класса Shape

## 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.

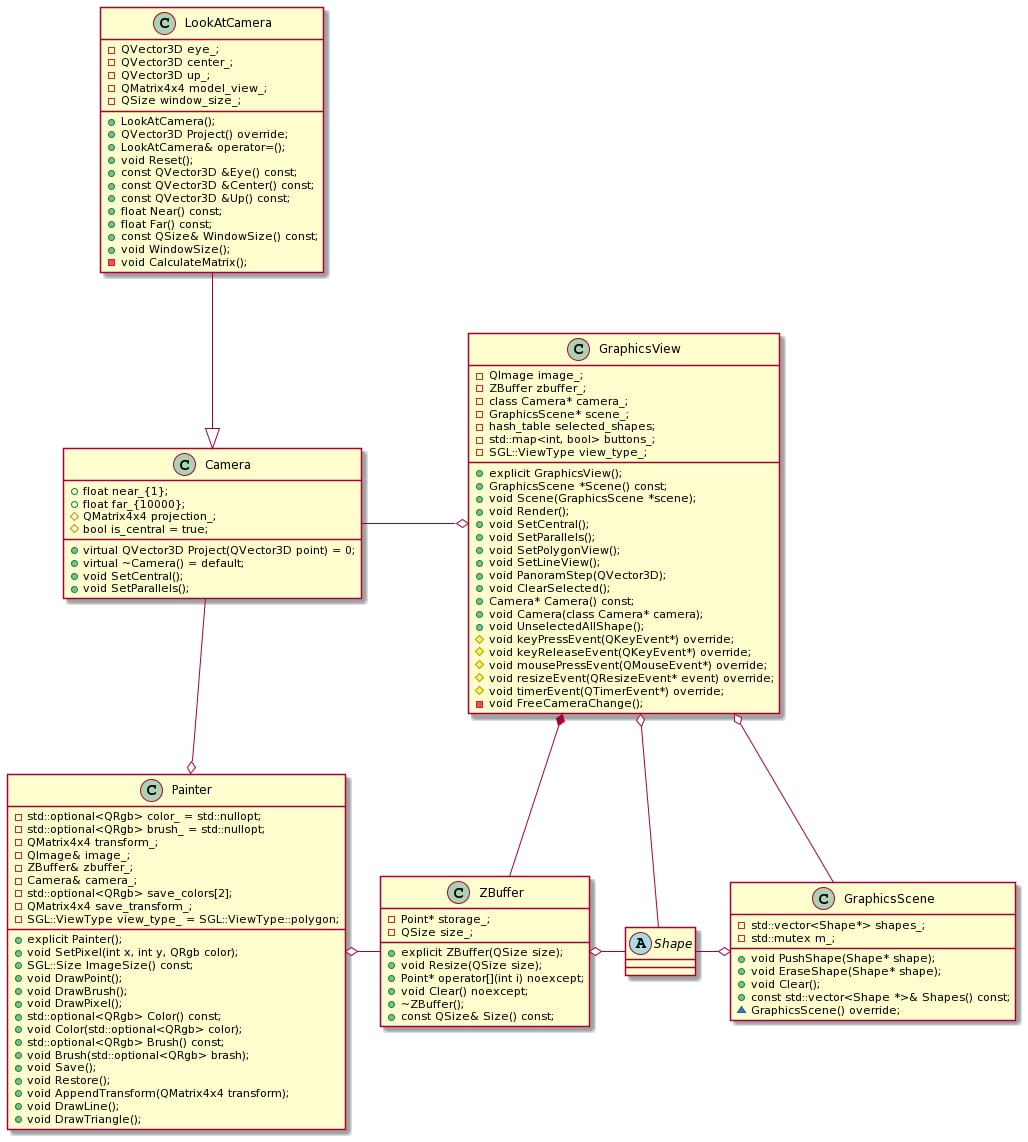


Рисунок 3.2 – Диаграмма классов системы

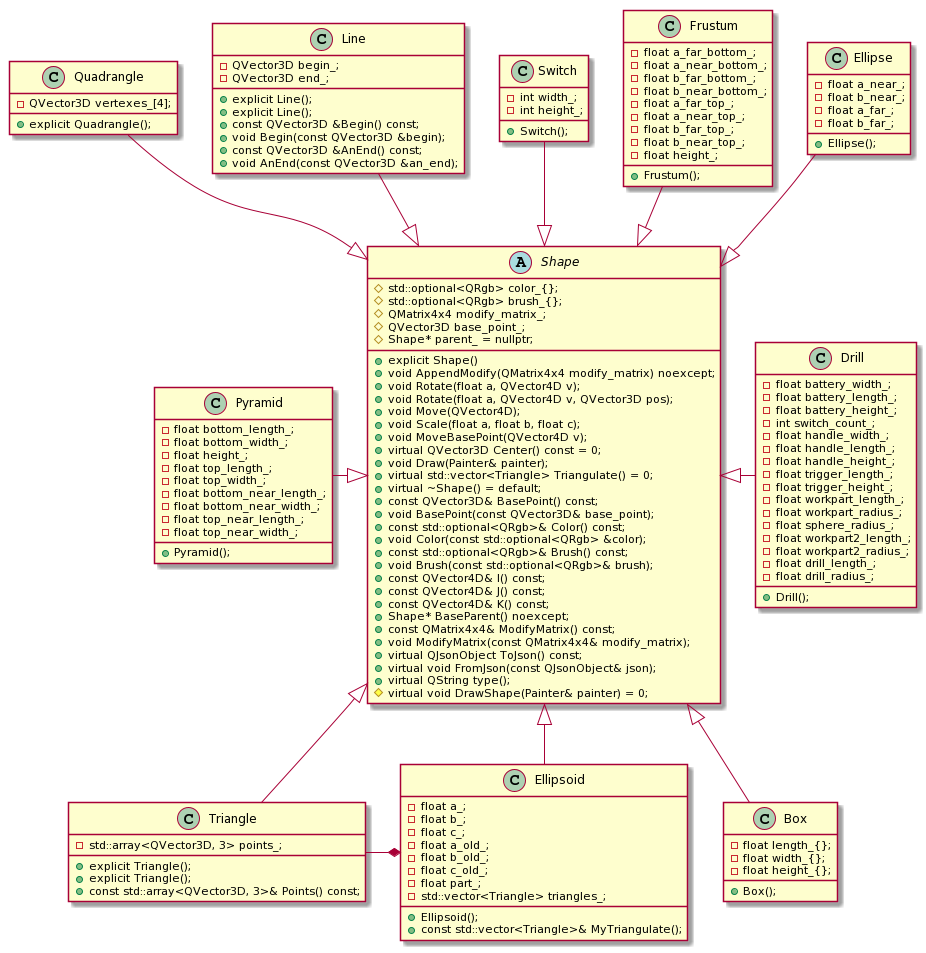


Рисунок 3.3 – Диаграмма классов для объектов

## 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

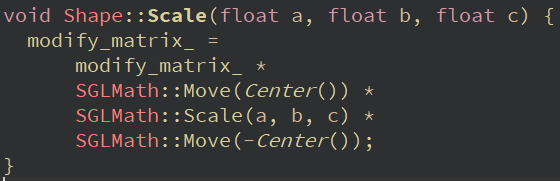


Рисунок 3.4 – Функция масштабирования объекта

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.

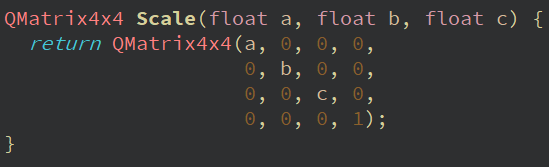


Рисунок 3.5 – Функция формирования матрицы масштабирования

## 3.4 Реализация поворота

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

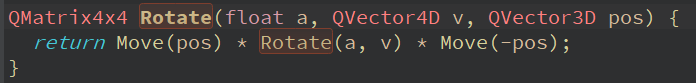


Рисунок 3.6 – Функция расчета матрицы поворота с учетом смещения

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.

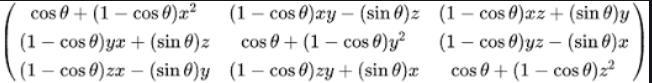


Рисунок 3.7 – Матрица поворота вокруг произвольной оси в общем виде

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

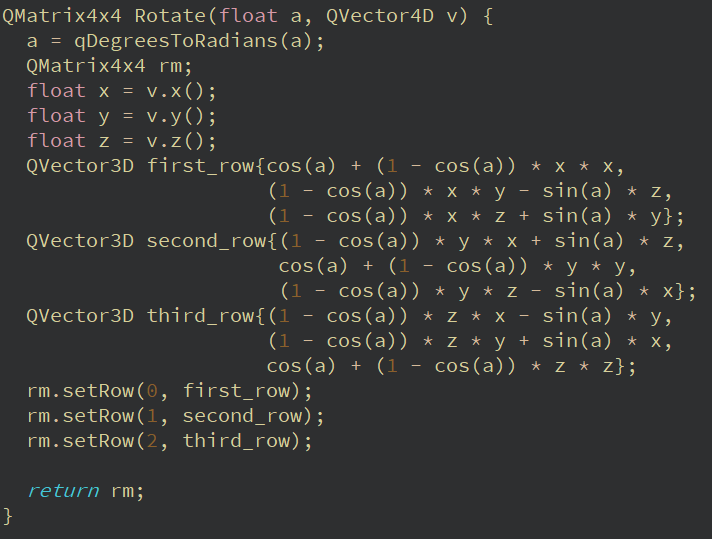


Рисунок 3.7 – Функция формирования матрицы поворота

## 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.

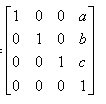


Рисунок 3.8 – Матрица переноса в общем виде

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

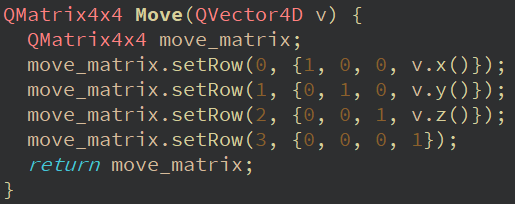


Рисунок 3.9 – Функция формирование матрицы переноса

## 3.6 Реализация изменения параметров

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



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.

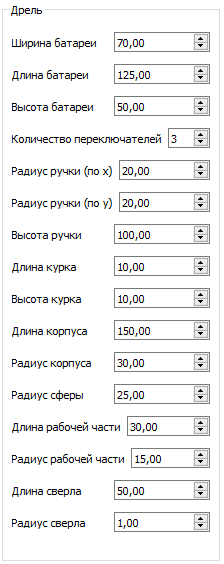


Рисунок 3.11 – Форма ввода параметров

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.

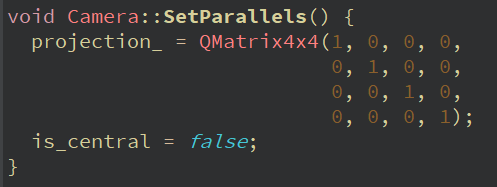


Рисунок 3.12 – Функция переключения на параллельное проецирование

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

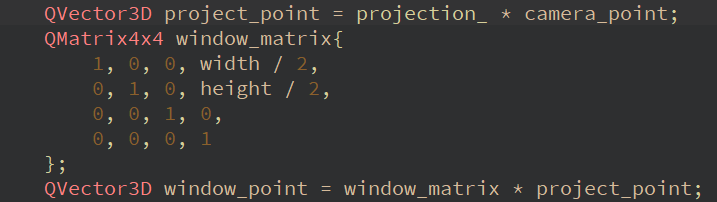


Рисунок 3.13 – Перевод трехмерных точек в координаты окна

## 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.

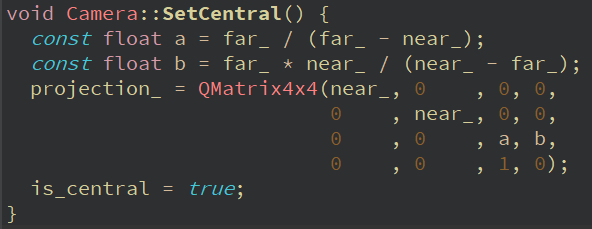


Рисунок 3.14 - Функция переключения на центральное проецирование

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

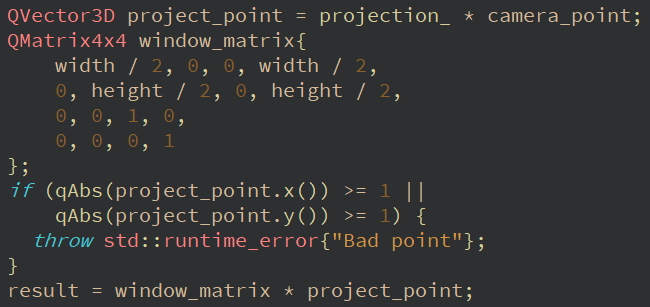


Рисунок 3.15 – Перевод трехмерных точек в координаты окна

## 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.

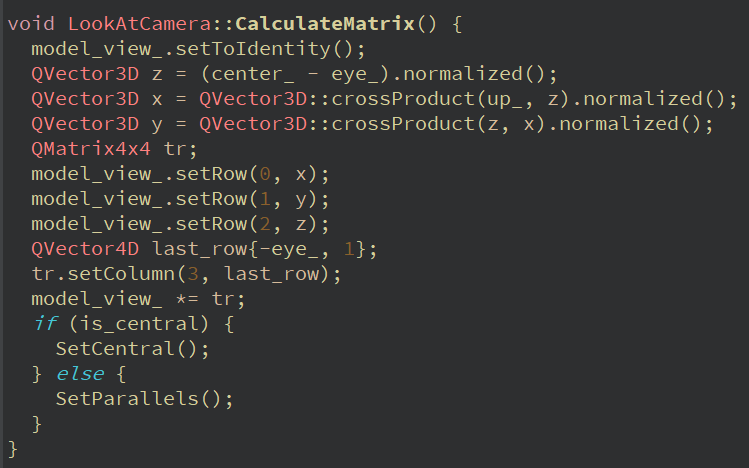


Рисунок 3.16 – Функция расчета матрицы для камеры

Перемещение «свободной» камеры (рис. 3.17).

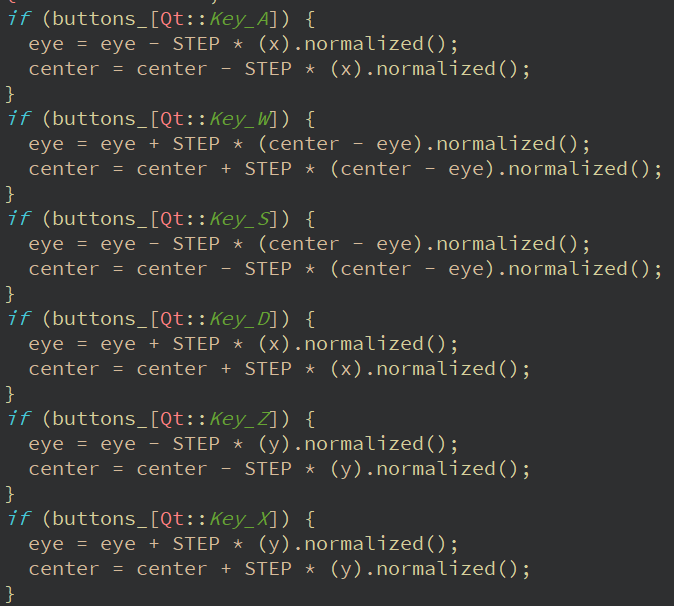


Рисунок 3.17 – Перемещение «свободной» камеры

Вращение «свободной» камеры (рис. 3.18).

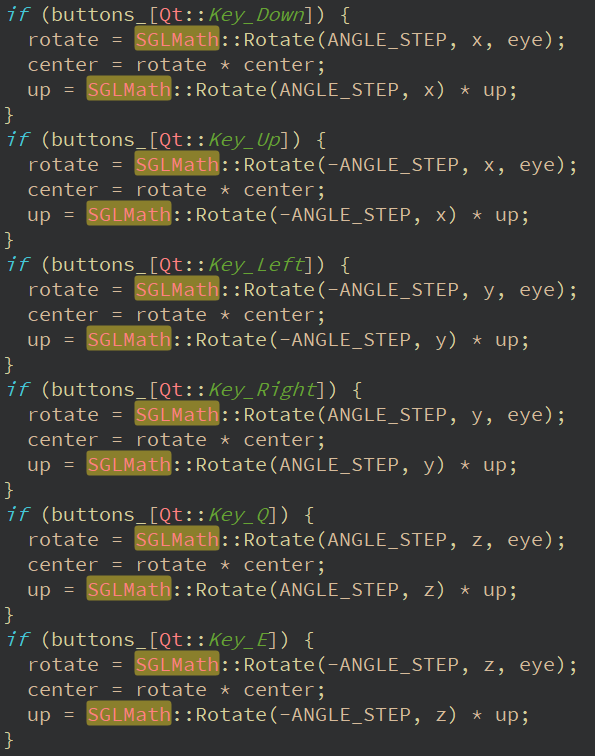


Рисунок 3.18 - Вращение «свободной» камеры

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

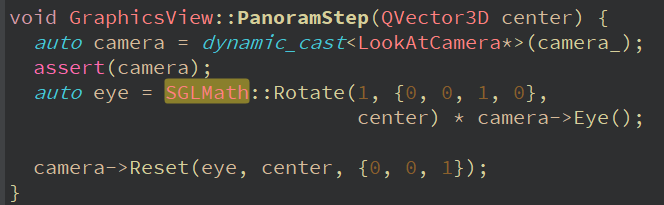


Рисунок 3.19 – Вращение камеры вокруг точки

## 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)

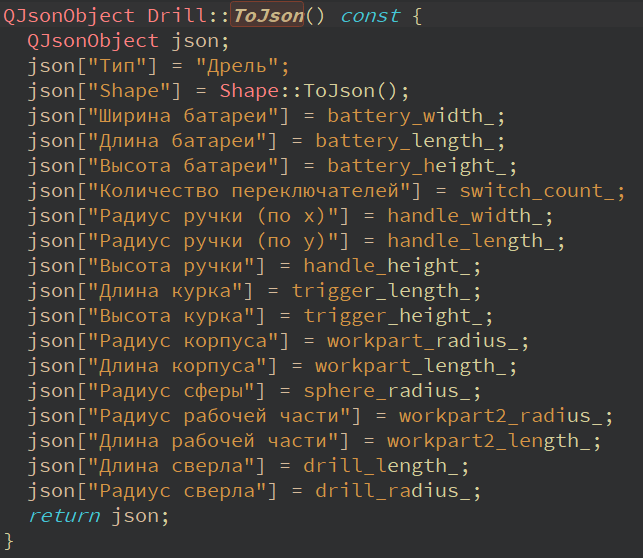


Рисунок 3.20 – Функция преобразования объекта в json

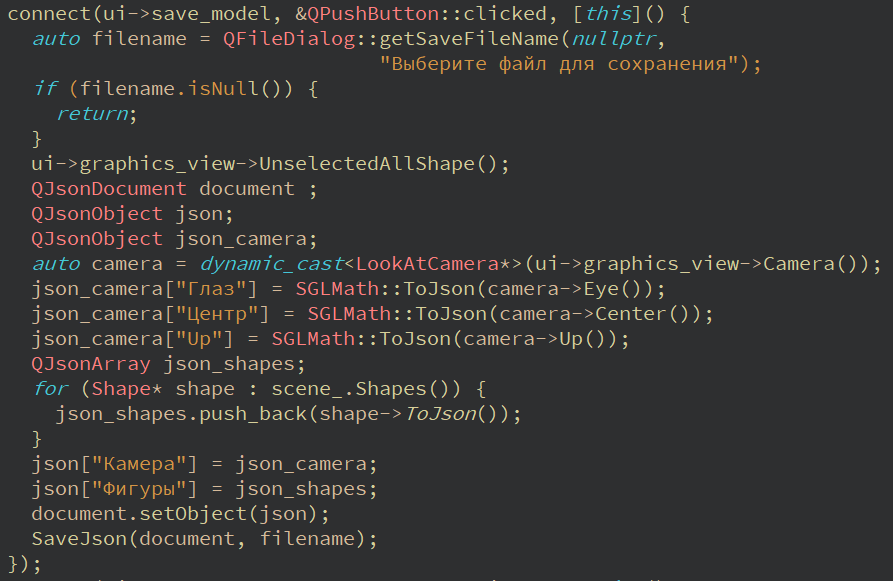


Рисунок 3.21 – Функция сохранения сцены в файл

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).

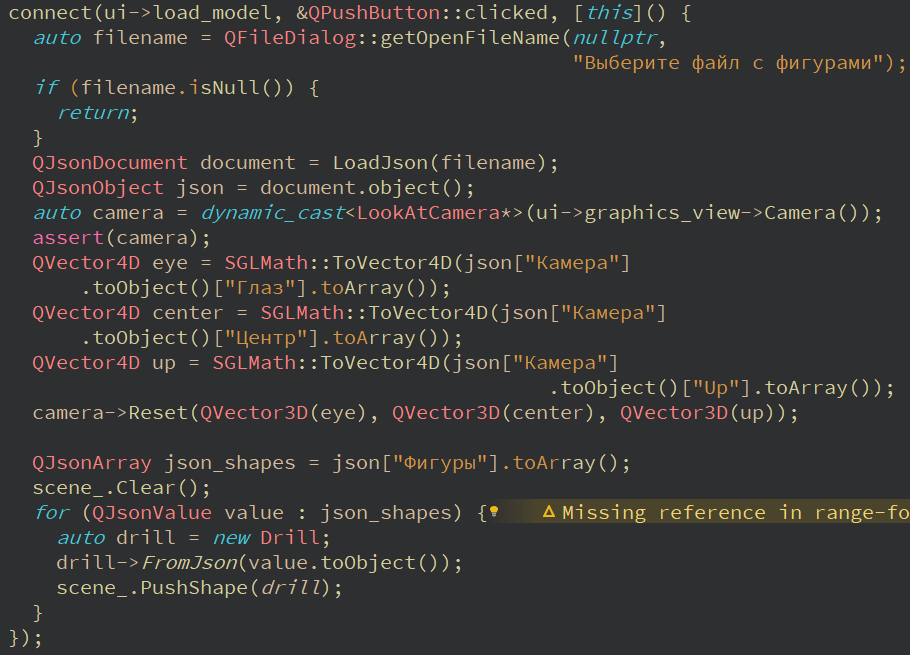


Рисунок 3.22 – Функция загрузки сцены

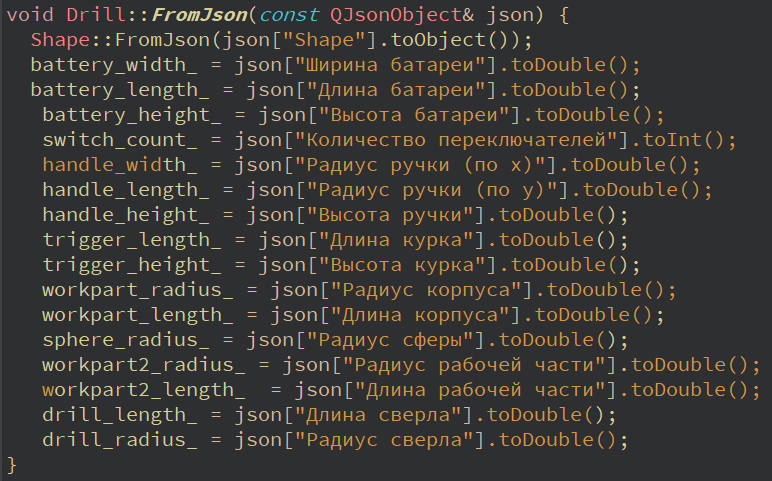


Рисунок 3.23 – Функция преобразования объекта из json

## 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).

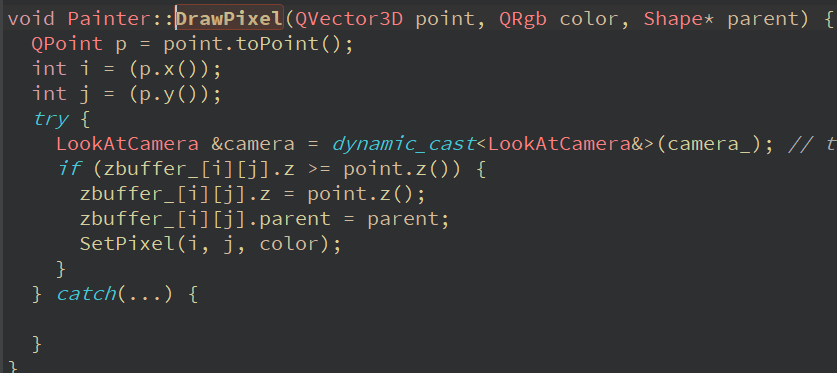


Рисунок 3.24 – Реализация алгоритма z-буфера

# 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

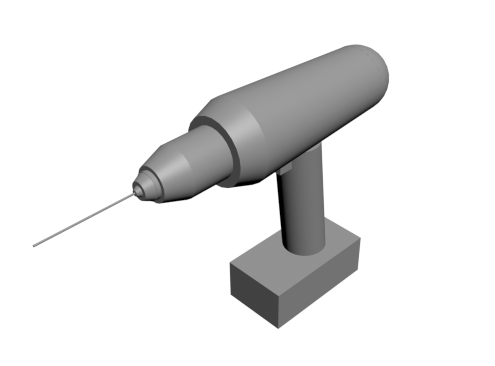


Рисунок А.1- Электрическая дрель

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



Рисунок Б1 – Проверка на оригинальность

# Приложение В. Экранные формы

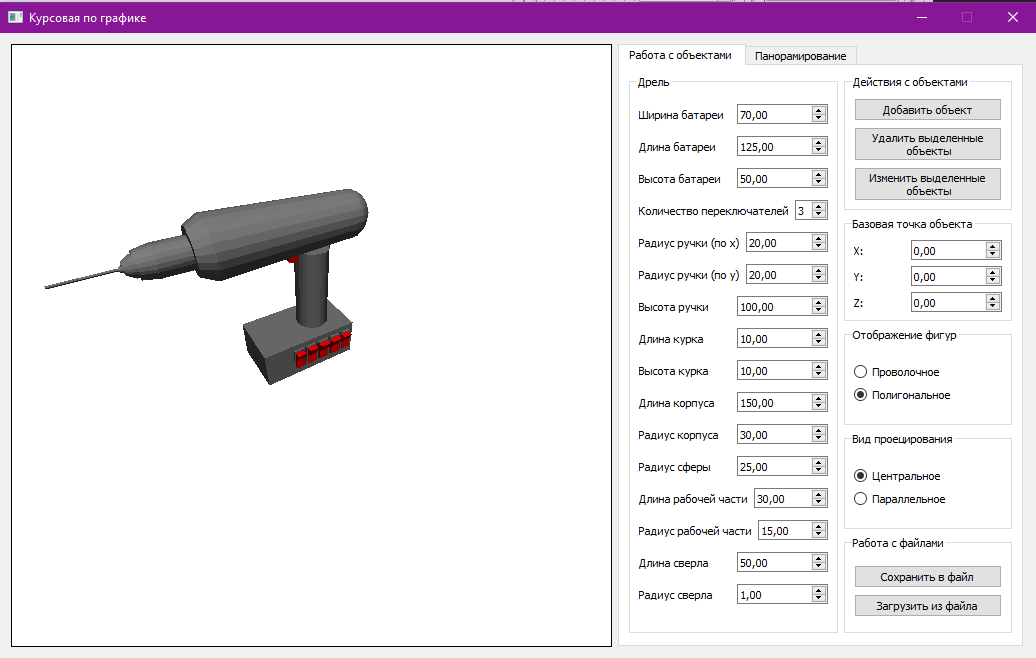


Рисунок В.1 – Полигональная модель с центральным проецированием

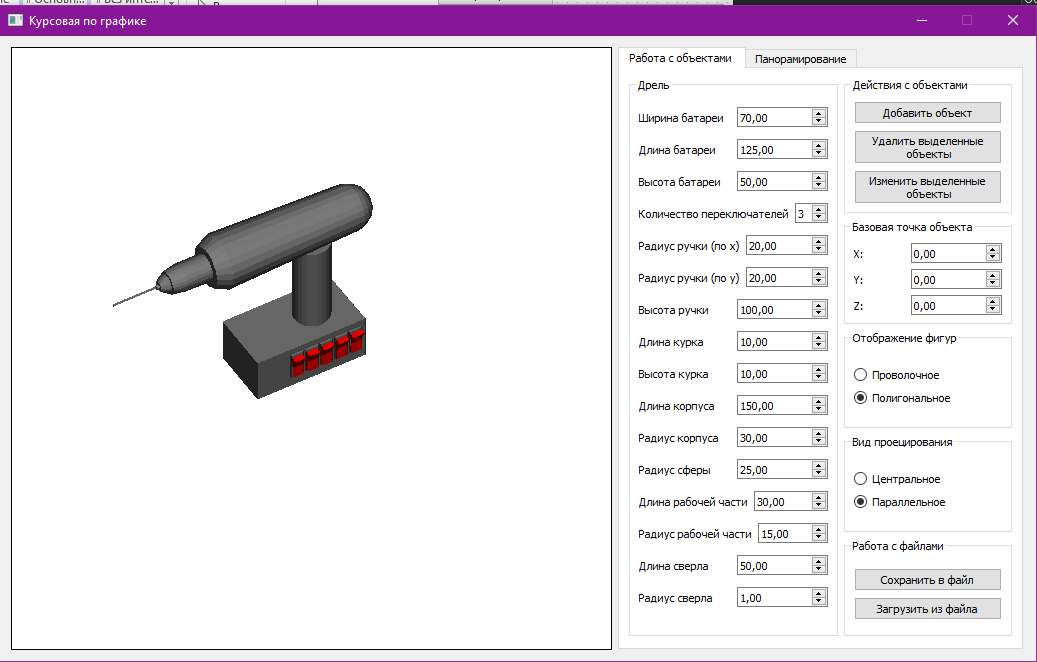


Рисунок В.2 – Полигональная модель с параллельным проецированием

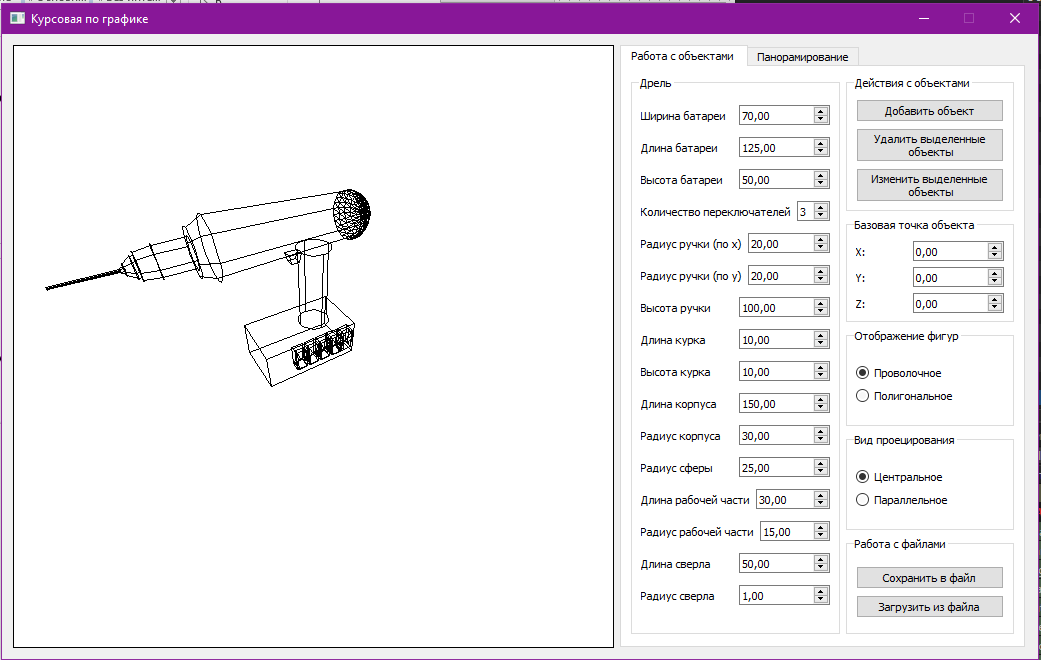


Рисунок В.3 – Проволочная модель

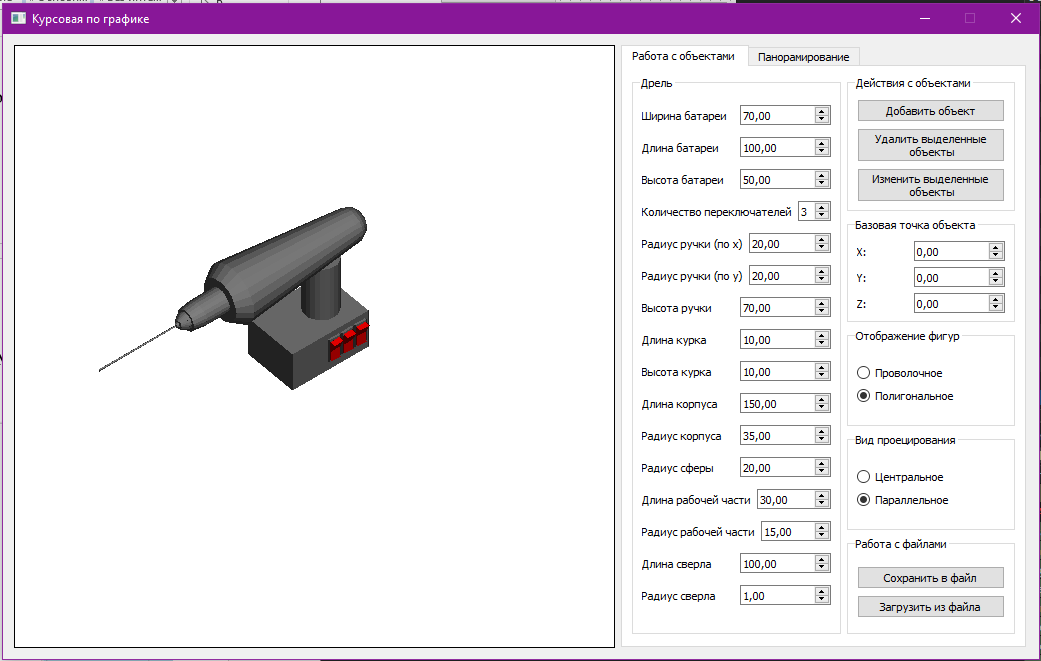


Рисунок В.4 – Изменение параметров объекта

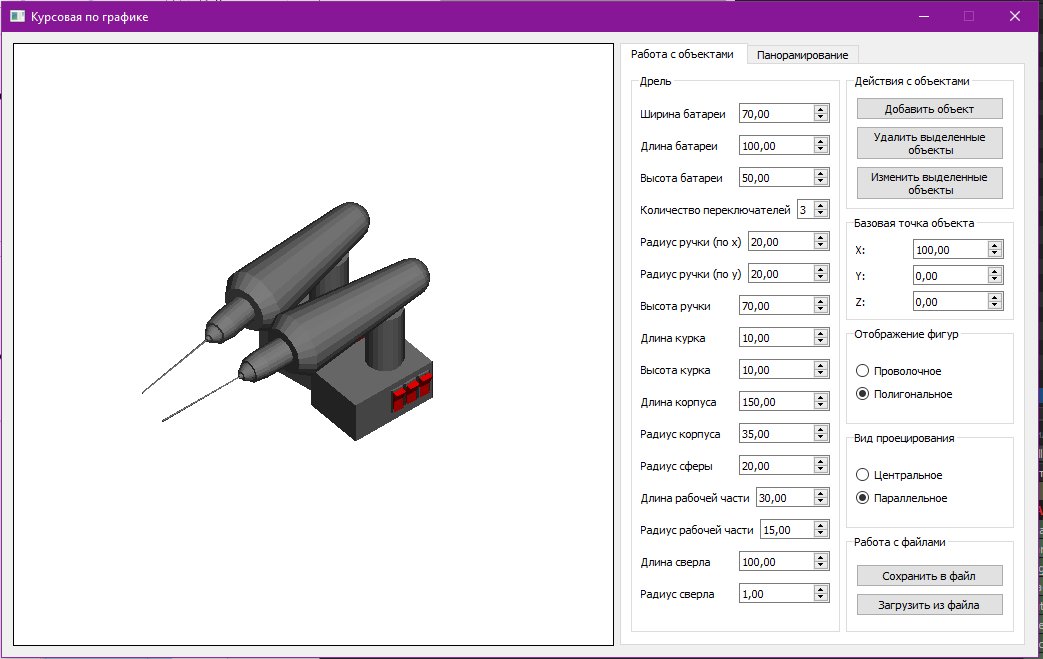


Рисунок В.5 – Создание нового объекта

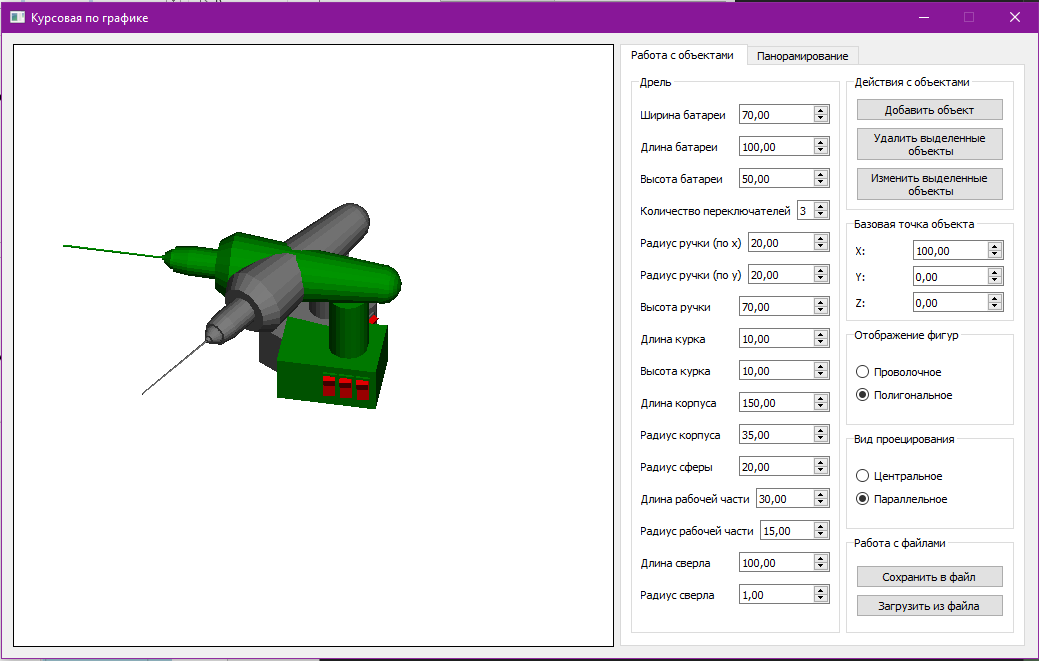


Рисунок В.6 – Поворот объекта

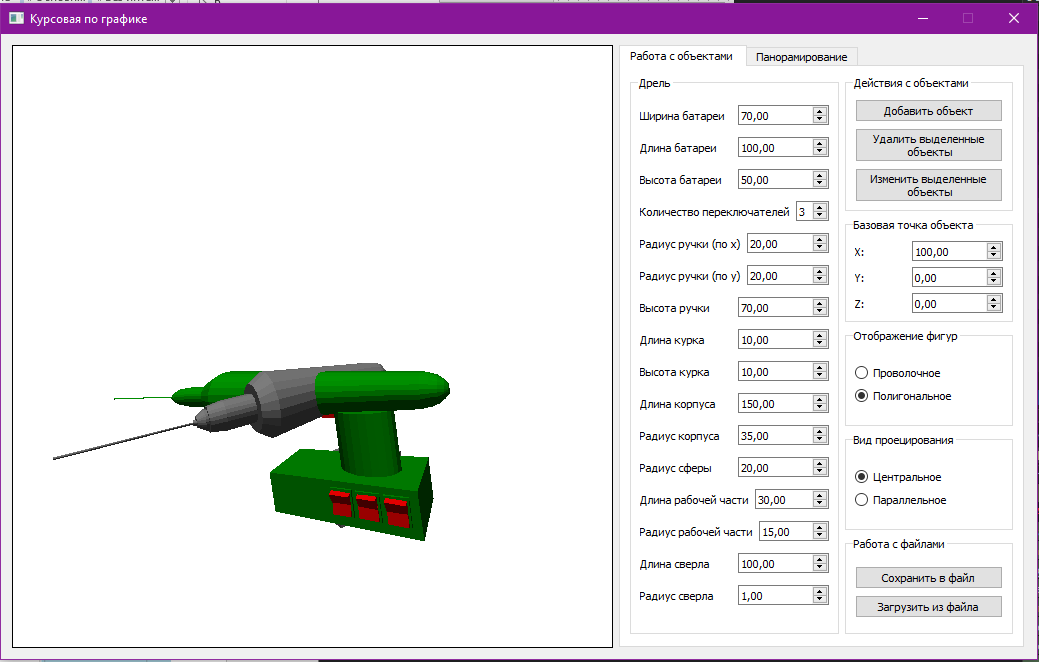


Рисунок В.7 – Изменение положения камеры

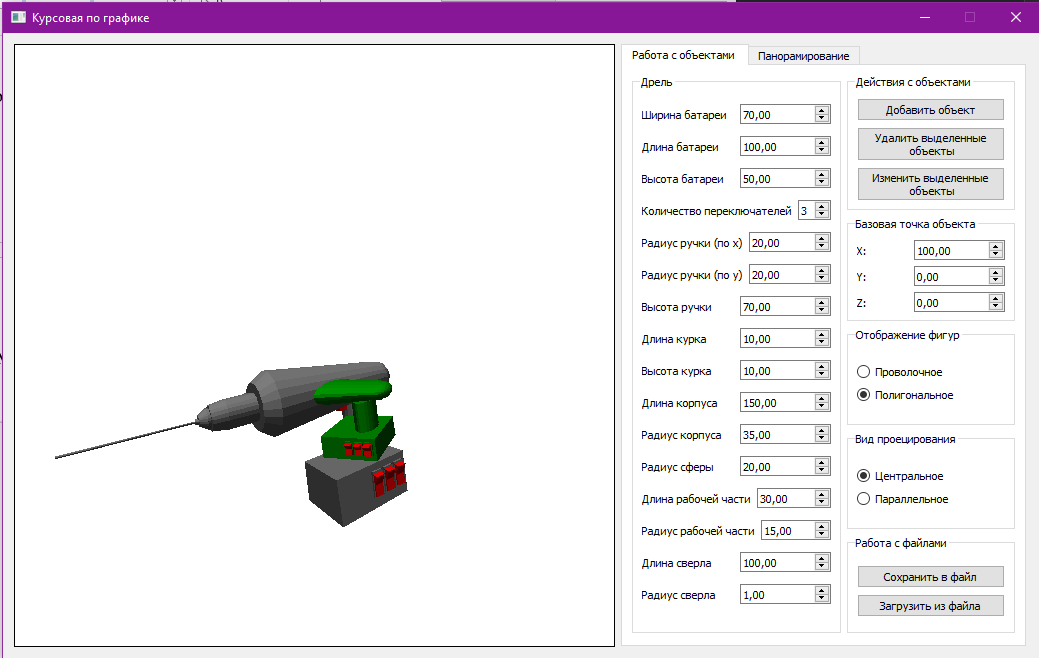


Рисунок В.8 – Масштабирование объектов

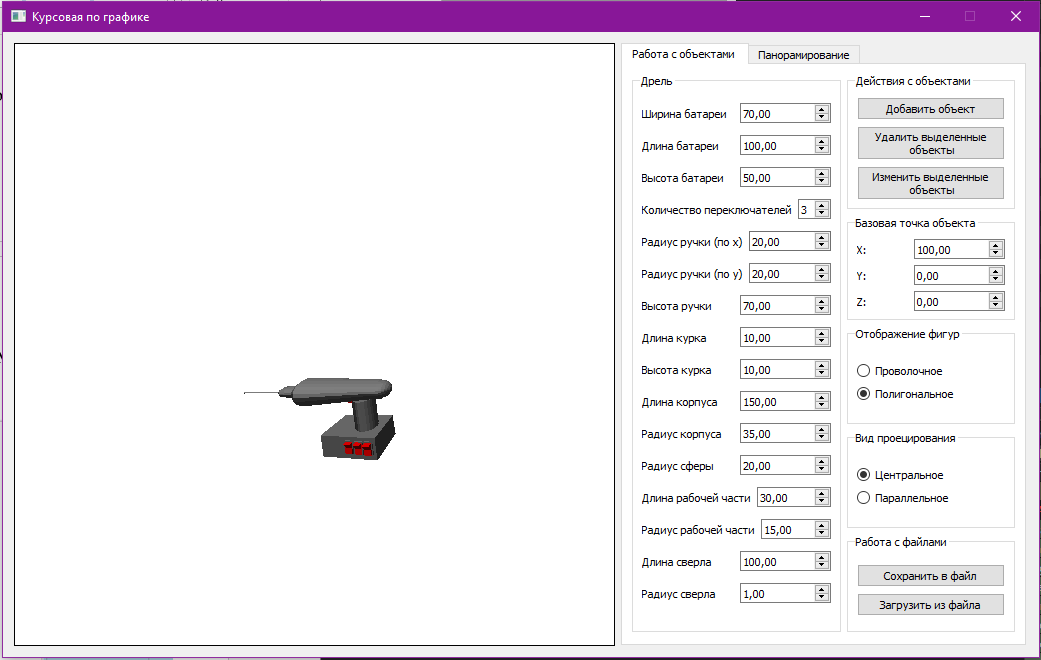


Рисунок В.9 – Удаление объекта

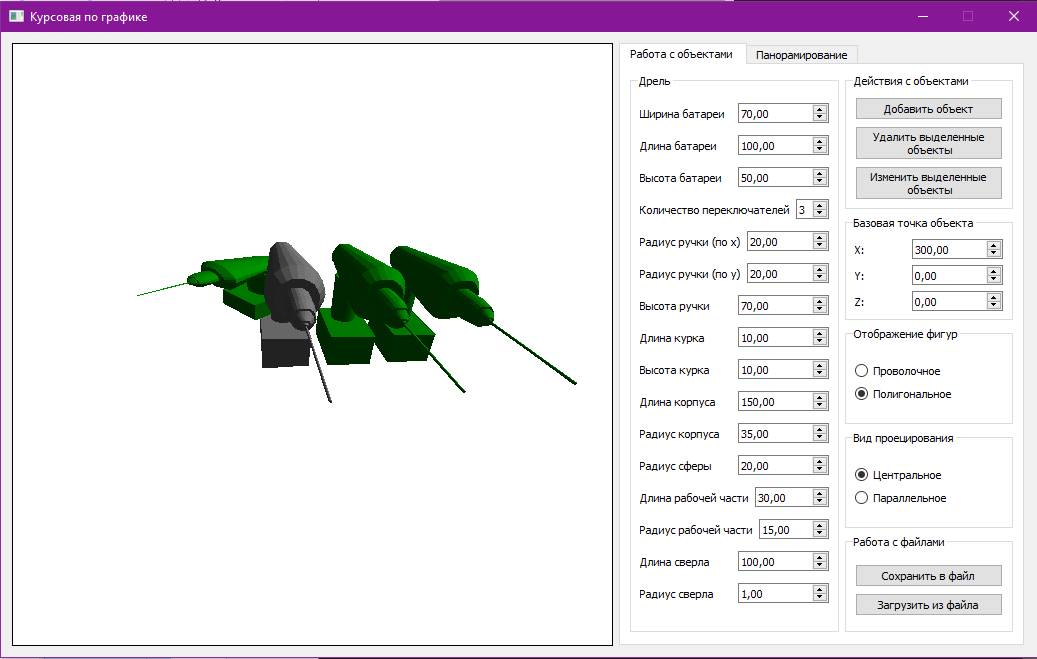


Рисунок В.10 – Выделение объектов

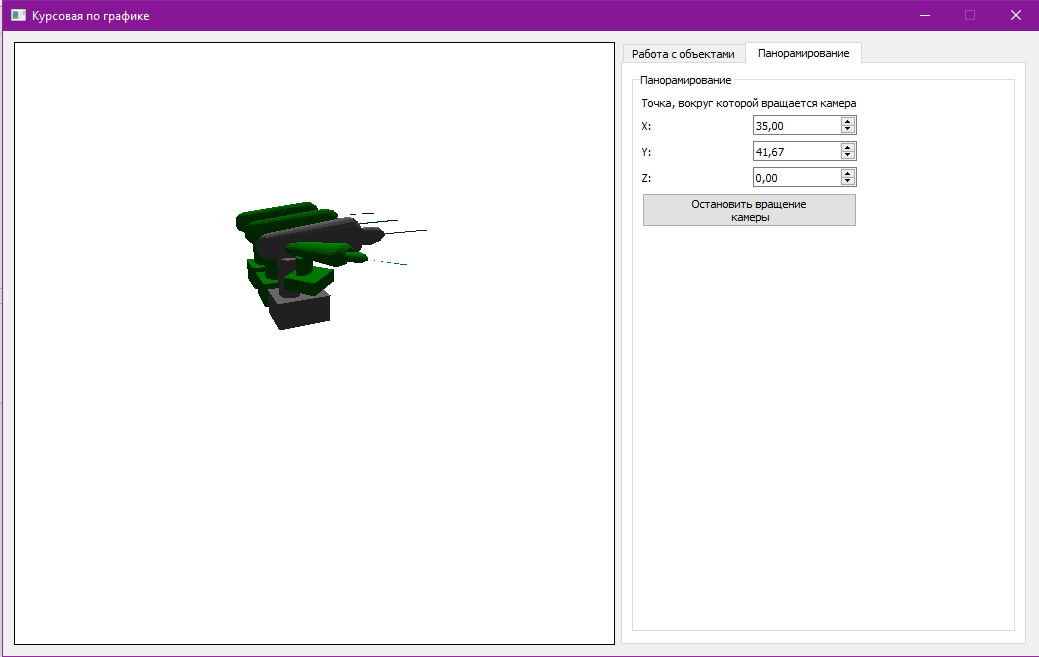


Рисунок В.11 – Панорамирование

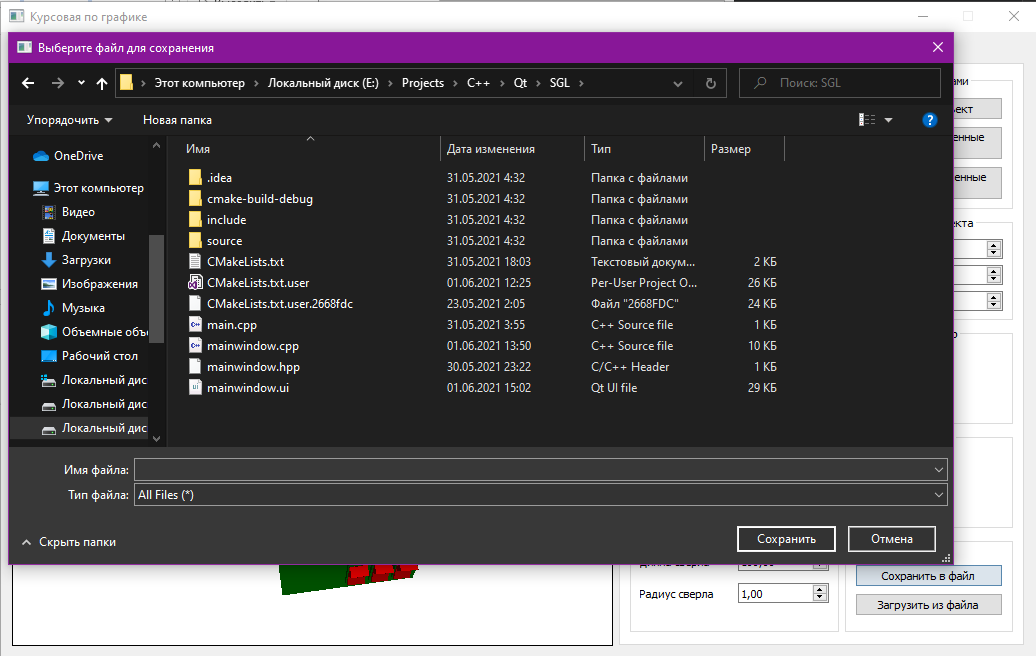


Рисунок В.12 – Сохранение в файл

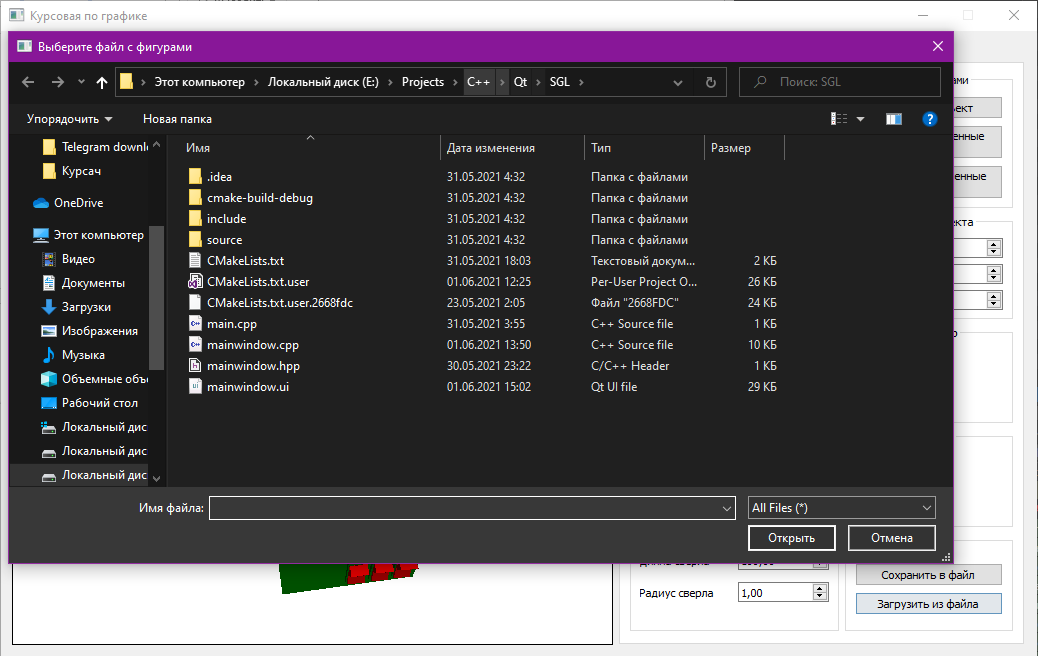


Рисунок В.13 – Загрузка из файла

# 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>***

***\**** ***cbbbbbbbbbbbc<br>***

***\**** ***3cccccccccccc4<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 х)"] = handle\_width\_;

json["Радиус ручки (по у)"] = 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 х)"].toDouble();

handle\_length\_ = json["Радиус ручки (по у)"].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);

}