Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ

Студент Ляпина Наталья Викторовна
фамилия, имя, отчество

Группа ИУ7-42Б

Тип практики технологическая

Название предприятия НУК ИУ МГТУ им. Н.Э. Баумана

Студент _____ Ляпина Н. В.
подпись, дата *фамилия, и.о.*

Руководитель практики _____ Куров А. В.
подпись, дата *фамилия, и.о.*

Оценка

2022 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ7
_____ Рудаков И. В.
«30» июня 2022 г.

ЗАДАНИЕ
на прохождение производственной практики
(технологическая практика)

Студент 2 курса группы ИУ7-42Б
Ляпина Наталья Викторовна
в период
с 30.06.2022 г. по 20.07.2022 г.
Предприятие:
НУК ИУ МГТУ им. Н. Э. Баумана
Руководитель практики от кафедры:
Куров А. В.
Задание:
1. Изучить документацию инструментов для автоматизации и поддержки процессов непрерывной интеграции и непрерывного развертывания программного обеспечения в процессе разработки: Qt Test, Gitlab CI/CD, Docker.
2. Собрать материалы по применению средств и методов автоматизации процессов непрерывной интеграции и непрерывного развертывания программного обеспечения в процессе разработки.
3. Получить практические навыки автоматизации процессов интеграции и развёртывания программного обеспечения в процессе разработки.

Дата выдачи задания «30» июня 2022 г.

Руководитель практики от кафедры _____ **/ Куров А. В.** /

Студент _____ **/ Ляпина Н. В.** /

Содержание

Введение

В наше время построение трехмерных изображений является одной из множества задач компьютерной графики, но существующие алгоритмы, которые дают возможность получить реалистичное изображение, работают крайне медленно. Также алгоритмы, которые позволяют построить нагруженные сцены в реальном времени, не учитывают все особенности объектов изображения и реализуются при помощи графических процессоров.

Целью практики является получение навыков разработки программного обеспечения, реализующего отображение сцены, приобретение опыта самостоятельной разработки программного продукта, изучение средств автоматизации развёртывания, сборки и тестирования программы

Задачи, необходимые к выполнению для достижения данной цели:

1. Изучить документацию Gitlab CI/CD, Docker, Qt, Qt Test, Qt Widgets, ffmpeg

2. Создать программу, принимающую данные о сцене, содержащей объекты, и создающую изображение. Использовать алгоритм Z-буфера с тенями для отрисовки сцены.

3. Создать сценарий gitlab-ci.yml автоматизации сборки, тестирования, выполнения замерного эксперимента и получения данных будущего исследования.

4. Выбрать готовый подходящий для поставленных задач отрисовки образ docker.

5. Создать три сцены разной сложности (по числу полигонов, из которых они состоят) для демонстрации работоспособности всего конвейера.

6. Создать модульный тест для демонстрации работоспособности системы.

7. Создать сценарий исследования зависимости времени выполнения от количества полигонов.

1 Знакомство с документацией

Далее представлено краткое изложение изученной документации, необходимой для разработки программного продукта и автоматизации его развёртки. Указаны ссылки на использованные материалы.

1.1 Gitlab CI/CD

GitLab CI/CD — это инструмент для разработки программного обеспечения с использованием непрерывных методологий. CI (Continuous Integration) — непрерывная интеграция. Каждое изменение, представленное в приложении, даже в ветках разработки, фиксируется и тестируется автоматически и непрерывно. CD (Continuous Deployment) — непрерывная автоматическая развёртка приложения.

GitLab CI/CD использует ряд концепций для описания и выполнения сборки и развертывания [1]:

1. Pipeline (сборочная линия) — это компонент верхнего уровня, состоящий из:

   (a) Задания (job), которые определяют, что делать. Они могут присутствовать в любом количестве и иметь ограничения на выполнение

   (b) Стадии (stage), которые определяют, когда выполнять задания.

Листинг 1.1 – пример задачи «Generate image» из стадии «gen_image»

```
1       Generate image:
2         stage: gen_image
3         script:
4             - cd new_code
5             - bash gen_image.sh
```

Задания выполняются runner'ами. Несколько заданий на одной стадии выполняются параллельно, если имеется достаточное количество одновременно работающих runner'ов. Если все задания на этапе выполняются успешно, конвейер переходит к следующему этапу. Если какое-либо задание на этапе завершается сбоем, следующий этап (обычно) не выполняется, и конвейер завершается досрочно.

Для задания также можно объявить зависимости:

Листинг 1.2 – пример использования «needs»

```
    needs:
          - Unit tests
          - Func tests
```

В данном примере текущее задание будет выполнено лишь в случае успешного выполнения задания «Unit tests» и «Func tests»

2. Артефакты

   Задания могут оставлять архив файлов и каталогов — артефакт задания. Можно загружать артефакты с помощью пользовательского интерфейса GitLab. Чтобы создать артефакты заданий, нужно использовать ключевое слово «artifacts» в файле .gitlab-ci.yml

Листинг 1.3 – пример использования «artifacts»

```
    artifacts:
        paths:
            - new_code/img.png
      expire_in: 7 days
```

В примере после выполнения текущего задания к артефактам добавятся файлы, соответствующие пути «new_code/img.png», а время хранения этих артефактов составит 7 дней

3. gitlab-ci.yml Были изучены основные синтаксические конструкции и правила построения .gitlab-ci.yml файла. Были рассмотрены такие ключевые слова, как:

   (a) before_script (команды, выполняемые до основного скрипта)

   (b) image (клонирование образа docker)

   (c) stage (определение стадий сборочной линии)

   (d) script (команды основного скрипта)

   (e) needs (определение зависимостей и необходимых артефактов предыдущих стадий конвейера)

Листинг 1.4 – пример использования ключевых слов в .gitlab-ci.yml

```
        Timer:
            stage: calculate_time
          image: python
          before_script:
            - pip install matplotlib
          script:
            - cd new_code
            - bash gen_time.sh
          artifacts:
            paths:
              - new_code/graph.png
            expire_in: 7 days
          needs:
            - job: Release build
              artifacts: true
```

В данном примере описано задание “ Timer” из стадии “calculate_time”, которому необходим docker образ python. Перед основным скриптом происходит установка библиотеки matplotlib для построения графиков и диаграмм языка Python. Основной скрипт запускает Bash скрипт, предварительно переходя в директорию new_code, оставляя артефакты по указанному пути new_code/graph.png. Это задание выполнится только при условии успешного выполнения задания Release build и при наличии соответствующих этому этапу артефактов.

1.2 Docker

Была изучена документация Docker [2] и рассмотрен пример создания Dockerfile.

Листинг 1.5 – пример Dockerfile

```
FROM alpine:3.9 as builder

RUN apk update && apk add ca-certificates tzdata

FROM scratch

COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/
COPY --from=builder /usr/share/zoneinfo /usr/share/zoneinfo/

```

```
10      COPY main /
11      CMD ["/main"]
```

В данном примере:

1. FROM задаёт базовый (родительский) образ, основанный на alpine (легковесный дистрибутив Linux)
2. RUN используется для установки в образы дополнительных пакетов или их обновления (update)
3. COPY копирует файлы из одной директории в другую
4. CMD предоставляет Docker команду, которую нужно выполнить при запуске контейнера. Результаты выполнения этой команды не добавляются в образ во время его сборки. В одном файле Dockerfile может присутствовать лишь одна инструкция CMD

В этом примере также использована многоэтапная сборка, поэтому для удобства первому этапу дается понятное название — builder, иначе обращение велось бы по целочисленным номерам, что менее читаемо. На втором этапе используется «пустой» образ scratch. В нём нет файлов — это позволяет полностью контролировать содержимое контейнера Docker и размер образа. Артефакты первого этапа используются во втором (см. команду COPY).

Также было изучено использование Docker Hub, создание, удаление, клонирование репозиториев.

В работе использовались готовые образы с Docker Hub.

Листинг 1.6 – пример использования образа с Docker Hub из .gitlab-ci.yml файла

```
1       image: rabits/qt:5.15-desktop
```

В данном примере используется образ с Docker Hub, содержащий Qt.

1.3 Qt

Qt — это библиотека классов C++ и набор инструментального программного обеспечения для создания приложений с графическим интерфейсом. Бы-

ли изучены и использованы при реализации программного продукта интерфейсы следующих классов из библиотеки Qt [3]:

1. QGraphicsScene

   Данный класс используется для визуализации графических примитивов, таких как линии, плоские фигуры, текст и многих других. Он также предоставляет функционал, позволяющий эффективно определять расположение и видимость элементов сцены. Были использованы методы очистки сцены (clear), добавления pixmap (представление внеэкранного изображения, реализованное классом QPixmap) на сцену (addPixmap).

2. QPixmap

   Класс QPixmap является представлением закадрового изображения, которое можно использовать в качестве устройства для рисования. Этот класс разработан и оптимизирован для отображения изображений на экране.

3. QImage

   Класс QImage обеспечивает аппаратно-независимое представление изображения, которое обеспечивает прямой доступ к данным о пикселах и может использоваться в качестве устройства для рисования. QImage разработан и оптимизирован для прямой манипуляции пикселами изображения. Этот класс предоставляет набор функций, которые можно использовать для получения разнообразной информации об изображении и его трансформации. Функционал класса позволил заполнять изображение одним цветом (выполнять заливку — fill), устанавливать цвета конкретных пикселов (setPixel) и сохранять полученное изображение в необходимом формате PNG (save).

4. QString

   Класс QString предоставляет строку символов Unicode. Он хранит строку 16-битных QChars, где каждый QChar соответствует одной кодовой единице UTF-16

5. QVector3D

   Класс QVector3D представляет вектор или вершину в трехмерном про-

странстве. Векторы являются одним из основных строительных блоков трехмерного представления и рисования. Они состоят из трех координат, традиционно называемых x, y и z. Использованы методы нормализации вектора (normalize), скалярного (dotProduct) и векторного (crossProduct) произведений, расчёта длины (length).

6. QVector4D
   Класс QVector4D представляет вектор или вершину, состоящие из четырёх координат, традиционно называемых x, y, z и w, в четырехмерном пространстве.

7. QColor
   Класс QColor предоставляет цвета на основе значений RGB, HSV или CMYK. Цвет обычно задается в терминах компонентов RGB (красный, зеленый и синий), но также возможно указать его в терминах HSV (оттенок, насыщенность и значение) и CMYK (голубой, пурпурный, желтый и черный). Кроме того, цвет можно указать с помощью специального имени цвета.

1.4 QtWidgets

Модуль QtWidgets предоставляет набор элементов пользовательского интерфейса для создания классических пользовательских интерфейсов в стиле рабочего стола. Были использованы следующие классы [4]

1. QPushButton
   Виджет QPushButton представляет собой командную кнопку. Нажатие на кнопку даёт сигнал обработчику выполнить какое-либо действие. Командная кнопка имеет прямоугольную форму и обычно отображает текстовую метку, описывающую ее действие. Если произошло событие clicked, управление передаётся обработчику этого события. Использована для отрисовки сцены с заданным поворотом.

2. QGraphicsView
   QGraphicsView визуализирует содержимое QGraphicsScene в прокручиваемом окне просмотра. Использован для отображения сцены в окошке.

1.5 QtTest

Qt Test — это фреймворк для модульного тестирования приложений и библиотек на основе Qt [5].

Чтобы создать тест, необходимо создать подкласс QObject и добавить к нему один или несколько приватных слотов. Каждый приватный слот — это тестовая функция. QTest::qExec() может использоваться для выполнения всех тестовых функций в тестовом объекте. Были изучены следующие макросы:

1. QCOMPARE(actual, expected)
   Макрос QCOMPARE() сравнивает фактическое значение с ожидаемым, используя оператор равенства. Если фактическое и ожидаемое значения совпадают, выполнение продолжается. Если нет, то в журнал тестирования записывается сбой, и функция тестирования завершается без попыток каких-либо последующих проверок. При сравнении типов с плавающей точкой (float, double и т.д.) для сравнения значений используется функция qFuzzyCompare().

2. QVERIFY(condition)
   Макрос QVERIFY() проверяет, верно условие или нет. Если условие верно, выполнение продолжается. Если нет, то в журнал тестирования заносится сбой и дальше тест выполняться не будет.

1.6 ffmpeg

Была изучена официальная документация утилиты ffmpeg, которая позволяет конвертировать несколько PNG изображений в MP4 видеофильм [6].

Листинг 1.7 – пример использования утилиты ffmpeg

```
1 ffmpeg -r 5 -f image2 -s 1600x900 -pattern_type glob -i '*.png' -codec:v
2 h264 -crf 25 -pix_fmt yuv420p my_film.mp4
```

В данном примере:

1. -r — установка частоты кадров в секунду;

2. -f — установка формата входного/выходного файлов;
3. -pattern_type — имя файла задаётся определённым форматом;
4. -pattern_type — имя файла задаётся определённым форматом;
5. -i — ввод имени входного файла;
6. -codec:v — настройки потоковой обработки;
7. -crf — фактор постоянного оценивания;
8. -pix_fmt — установка формата пикселей

2 Проектирование и реализация ПО

В этом разделе описаны этапы проектирования и реализации программного продукта, представлено изложение основных компонентов программы и обоснование выбора соответствующих методов решения поставленных задач.

Рисунок 2.1 – IDEF0 диаграмма работы реализованной программы

2.1 Формат входных и выходных данных и обоснование выбора

Для представления входных данных основным форматом был выбран формат файлов OBJ, так как:

1. Имеет широкую поддержку экспорта и импорта программного обеспечения САПР. Любое программное обеспечение САПР будет интерпретировать OBJ модель правильно и последовательно

2. Легко воспринимается пользователями без изучения дополнительных языков программирования и доступен для ручного изменения (не в бинарном формате)

3. В отличие от некоторых аналогов (например, STL) позволяет хранить текстуры (в отдельном MTL файле)

4. Согласно статье [7] формат файла OBJ, благодаря своей нейтральности, является одним из самых популярных форматов обмена для 3D-графики. Он также набирает обороты в индустрии 3D-печати, поскольку отрасль движется к полноцветной печати.

Для выходных данных был выбран формат файла PNG, так как:

1. Качество изображения формата PNG не меняется при любой степени сжатия.

2. Формат является платформонезависимым (в отличие от BMP файлов, удовлетворяющих только Windows).

3. Данный формат поддерживает максимальное возможное количество цветов.

4. Является совместимым с утилитой ffmpeg, используемой для создания видеофрагмента [6].

5. Согласно статье [8], среднеквадратическая ошибка (среднее различие в квадрате между идеальным и фактическим пиксельными значениями) для JPEG файлов на больших изображениях гораздо больше, чем у PNG файлов (данные в статье представлены для изображения размером 1024x1024 пиксела, в работе используется размер 1600x900 пикселов)

2.2 Реализуемые алгоритмы

2.2.1 Алгоритм Z-буфера

Данный алгоритм работает в пространстве изображений, используя буфер кадра для заполнения интенсивности каждого пикселя, здесь вводится некоторый Z-буфер (буфер глубины каждого пикселя).

Значение каждого нового пикселя, который нужно занести в буфер кадра, сравнивается с глубиной пикселя, занесенного в Z-буфер. Если сравнение

показывает, что новый пиксель расположен ближе к наблюдателю, то новое значение Z заносится в буфер и корректируется значение интенсивности.

Алгоритм, использующий Z-буфер крайне прост в своей реализации по сравнению с другими анализируемыми алгоритмами, также не тратится время на сортировку элементов сцены.

Но несмотря на его быстродействие увеличиваются затраты по памяти при использовании данного алгоритма, запоминается информация по каждому пикселю изображения.

Вычислительная сложность данного алгоритма равна $O(n \cdot m \cdot k)$, где $n \cdot m$ – количество пикселей в буфере кадра, k – количество полигонов.

Алгоритм

1. Всем элементам буфера кадра присвоить фоновое значение;
2. Инициализировать Z-буфер минимальными значениями глубины;
3. Выполнить растровую развертку каждого многоугольника сцены:
   (a) Для каждого пикселя, связанного с многоугольником вычислить его глубину $z(x, y)$;
   (b) Сравнить глубину пикселя со значением, хранимым в Z-буфере. Если $z(x, y) > z_b(x, y)$, то $z_b(x, y) = z(x, y)$ и цвет(x, y) = цветПикселя;
4. Отобразить результат;

Математическое содержание алгоритма

При известном уравнении плоскости, несущей конкретный многоугольник, вычисление глубины каждого пикселя могут быть произведены пошаговым способом.

При известном уравнении плоскости, несущей конкретный многоугольник, вычисление глубины каждого пикселя могут быть произведены пошаговым способом.

Уравнение плоскости имеет следующий вид (выражено через z):

$$z = \frac{ax+by+d}{c} \neq 0$$

Для сканирующей строки $y = const$, глубина пикселя, у которого $x_1 = x + \triangle x$, поэтому равна:

$$z_1 - z = -\frac{ax_1+d}{c} + \frac{ax+d}{c} = \frac{a(x-x_1)}{c}$$

Отсюда получаем, что

$$z_1 = z - \frac{a}{c} \triangle x, \triangle x = 1, \text{ поэтому } z_1 = z - \frac{a}{c}$$

Поскольку в данной задаче для визуализации используется только один вид многоугольников, а именно треугольник, то нахождение абсцисс точек пересечения горизонтали со сторонами треугольника будет выглядеть следующим образом:

- Для каждой из сторон треугольника будут записываться параметрические уравнения вида:

$$x = x_H + (x_K - x_H)t$$
$$y = y_H + (y_K - y_H)t$$

- После этого для каждой стороны находится параметр tпри пересечении с горизонталью $y = c$:

$$c = y_H + (y_K - y_H)t, \text{ где } t = \frac{c-y_H}{y_K} - y_H$$

- Если $t \in (0, 1)$, то рассчитывается абсцисса точки пересечения горизонтали со стороной треугольника:

$$x = x_H + (x_K - x_H)\frac{c-y_H}{y_K-y_H}$$

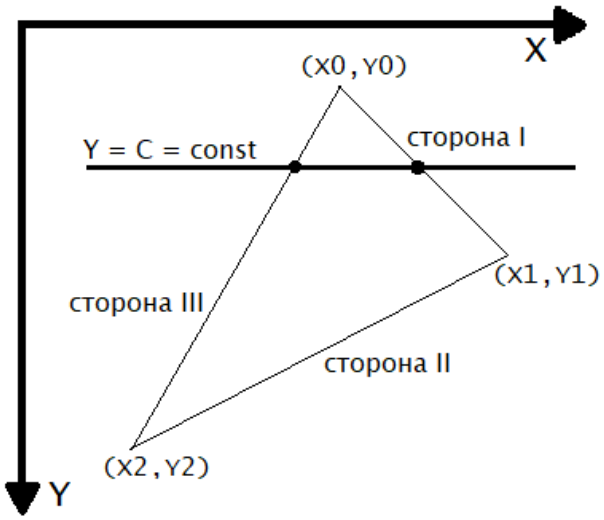

Рисунок 2.2 – Поиск абсцисс точек пересечения

2.2.2 Модифицированный алгоритм Z-буфера

1. Для каждого направленного источника света:
   (a) Инициализировать теневой z-буфер минимальным значением глубины;
   (b) Определить теневой z-буфер для источника;
2. Выполнить алгоритм z-буфера для точки наблюдения. При этом, если некоторая поверхность оказалась видимой относительно текущей точки наблюдения, то проверить, видима ли данная точка со стороны источников света. Для каждого источника света:
   (a) Координаты рассматриваемой точки (x, y, z) линейно преобразовать из вида наблюдателя в координаты (x', y', z') на виде из рассматриваемого источника света;
   (b) Сравнить значение $z_t(x', y')$ со значением $z'(x', y')$: Если $z'(x', y') < z_t(x', y')$, то пиксел высвечивается с учётом его затемнения, иначе точка высвечивается без затемнения;

2.2.3 Освещение

В данном проекте планеты не будут иметь свойств зеркального отражения, а, значит, в данном случае уместно только диффузное отражение, которое описывается по закону Ламберта: интенсивность отраженного света пропорциональна косинусу угла между направлением на точечный источник света и нормалью к поверхности.

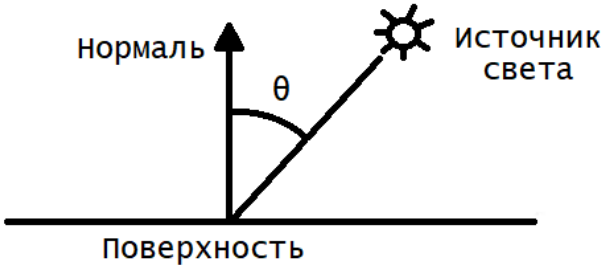

Рисунок 2.3 – Матовая поверхность

$I = I_i \cdot K \cdot \cos(\theta)$, где K – коэффициент свойства материалла поверхности, I_i – интенсивность источника света

Для определения цвета матовой поверхности определяется комбинацией собственного цвета поверхности и цвета излучения источника света. Также, как можно заметить, в формуле имеется косинус угла вектора источника света в пространстве между вектором нормали к самой поверхности, данную, величину можно определить при помощи скалярного произведения.

Пусть есть две точки – $A(x_A, y_A, z_A)$, принадлежащая поверхности, и точка $B(x_B, y_B, z_B)$, задающая положение источника в пространстве, также имеется вектор нормали $N(x_N, y_N, z_N)$, тогда вектор от точки поверхности к источнику света имеет следующие координаты:

$$V(x_B - x_A, y_B - y_A, z_B - z_A)$$

Отсюда следует, что:

$$N \cdot V = \mid N \mid \cdot \mid V \mid \cdot \cos(\theta)$$

После преобразований и подстановок получается следующее:

$$N \cdot V = x_N x_V + y_N y_V + z_N z_V$$

Следовательно с учетом того, что в программе будут использоваться единичные вектора нормали (уменьшает количество вычислений), получаем следующее:

$$\cos(\theta) = \frac{x_N(x_B - x_A) + y_N(y_B - y_A) + z_N(z_B - z_A)}{\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}} = \frac{x_N(x_B - x_A) + y_N(y_B - y_A) + z_N(z_B - z_A)}{|V|}$$

2.2.4 Метод закраски Гуро

Поскольку данный метод основан на билинейной интерполяции вектора нормали. Определение интерполированных значений удобно выполнять на моменте заполнения полигона, например, совместить с алгоритмом визуализации изображения. После этого рассматривается заполнение грани горизонталями в экранных координатах.

Рассмотрим основные этапы данного алгоритма:

1. Вычисление векторов нормалей к каждой грани;
2. Вычисление векторов нормалей к каждой вершине грани через усреднение нормалей к граням;
3. Вычисление интенсивностей в вершинах граней;
4. Интерполяция интенсивности вдоль ребер грани;
5. Линейная интерполяция вдоль сканирующей строки;

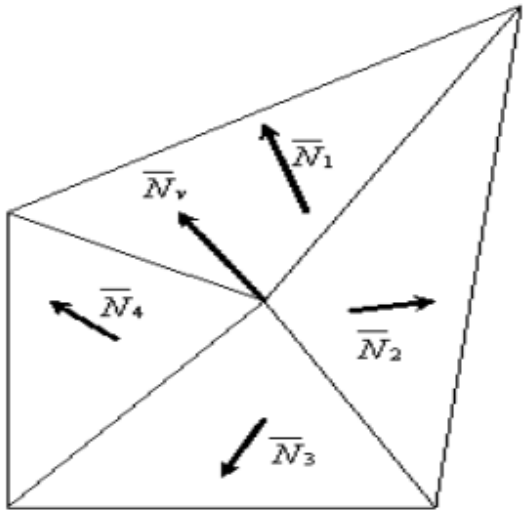

Рисунок 2.4 – Усреднение нормалей к граням

Интерполированная интенсивность I в точке (x, y) определяется из пропорции:

$\frac{I-I_1}{x-x_1} = \frac{I_2-I_1}{x_2-x_1}$, выражая отсюда I, получаем: $I = I_1 + \frac{(I_2-I_1)\cdot(x-x_1)}{(x_2-x_1)}$

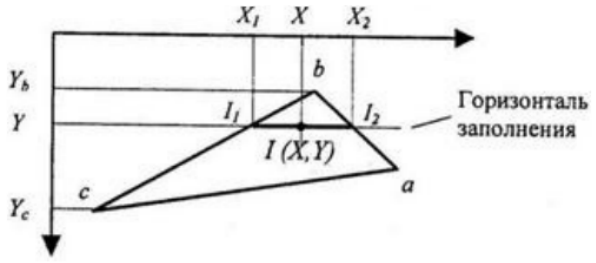

Рисунок 2.5 – Интерполяция значений интенсивности

Значения I_1 и I_2 на концах горизонтального отрезка является интерполяцией интенсивности в вершинах:

$\frac{l_1-l_b}{Y-Y_b} = \frac{l_c-l_b}{Y_c-Y_b}$, а также $\frac{l_2-l_b}{Y-Y_b} = \frac{l_a-l_b}{Y_a-Y_b}$, откуда получается, что

$$l_1 = l_b + \frac{(l_c-l_b)(Y-Y_b)}{Y_c-Y_b} \text{ и } l_2 = l_b + \frac{(l_a-l_b)(Y-Y_b)}{Y_a-Y_b}$$

2.3 Сценарий Gitlab CI/CD

На рисунке 2.5 продемонстрированы зависимости сценария CI/CD.

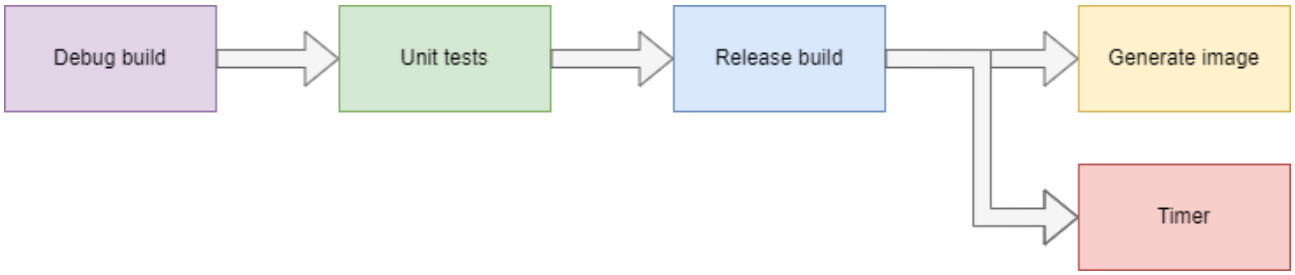

Рисунок 2.6 – диаграмма зависимостей в сценарии Gitlab CI/CD

Сценарий состоит из 5 стадий:

- build_debug — отладочная сборка программы В этой стадии одно задание — Debug build, в котором производится сборка отладочной версии программы.
- test — модульное тестирование программы В этой стадии одно задание — Unit tests, в котором осуществляется модульное тестирование функции SetAutoNormal() из заголовочного файла surface.h.
- build_release — сборка выпускаемой версии программы В этой стадии одно задание — Release build, в котором производится сборка версии программы для выпуска.
- gen_image – генерация изображения заданной сцены
- calculate_time – проведение замерного эксперимента и построение графиков

Описание зависимостей сценария CI/CD:

- Задание сборки отладочной версии программы не имеет зависимостей, так как отладочная сборка первична.

- Задание модульного тестирования программного модуля зависит от задания сборки отладочной версии программы, так как для запуска тестов необходимо наличие исполняемого файла программы, который может быть получен в результате отладочной сборки.

- Задание выпускаемой сборки программы зависит от задания модульного тестирования программы (соответственно, и от задания сборки отладочной версии программы, явно указывать косвенную зависимость в сценарии CI/CD не нужно), так как выпускаемая версия программы не должна содержать в себе ошибок.

- Задание проведения замерного эксперимента зависит от задания выпускаемой сборки, так как для исследования времени работы программы необходим исполняемый файл программы, для которого обеспечена защита от возникновения ошибочных ситуаций и оптимизация работы программы.

- Задания генерации изображения заданной сцены зависит от задания выпускаемой сборки, так как для генерации изображений сцены необходим исполняемый файл программы, для которого обеспечена защита от возникновения ошибочных ситуаций и оптимизация работы программы.

Далее приведено содержание файла .gitlab-ci.yml со сценарием CI/CD

Листинг 2.1 – содержание файла .gitlab-ci.yml со сценарием CI/CD

```
image: rabits/qt:5.15-desktop

stages:
  - build_debug
    - test
    - build_release
    - gen_image
    - calculate_time

Debug build:

   stage: build_debug

    script:
        - cd new_code
```

```
        - qmake CONFIG+=debug new_code.pro
        - make

  artifacts:
    paths:
            - new_code/new_code
        expire_in: 1 day

Unit tests:

    stage: test

    script:
        - cd new_code
        - ./new_code -test
    needs:
        - job: Debug build
          artifacts: true

Release build:

   stage: build_release

    script:
        - cd new_code
        - qmake CONFIG+=release new_code.pro
        - make

    artifacts:
        paths:
            - new_code/new_code
        expire_in: 1 day
    needs:
        - Unit tests
    when: manual


Generate image:

    stage: gen_image

    script:
        - cd new_code
        - bash gen_image.sh
    artifacts:
       paths:
            - new_code/img.png
        expire_in: 7 days
```

```
64      needs:
65          - job: Release build
66            artifacts: true
67
68  Timer:
69
70      stage: calculate_time
71    image: python
72      before_script:
73          - pip install matplotlib
74      script:
75          - cd new_code
76          - bash gen_time.sh
77      artifacts:
78          paths:
79              - new_code/graph.png
80          expire_in: 7 days
81      needs:
82          - job: Release build
83            artifacts: true
```

2.4 Docker

В сценарии Gitlab CI/CD были использованы образы docker с Docker Hub, удовлетворяющие задачам соответствующих заданий конвейера. Ниже приведено описание данных образов и обоснование их использования:

1. rabits/qt:5.15-desktop Образ rabits/qt:5.15-desktop представляет собой набор пакетов для сборки и исполнения программ в фреймворке Qt. Он использовался для всех заданий, кроме Timer, так как этому заданию требовалось иное специфичное окружение, а остальным была необходима сборка и исполнение программы, следовательно, библиотеки Qt. Зависимости образа:

    - unzip
    - git
    - openssh-client
    - ca-certificates
    - locales

- sudo
- curl
- make
- openjdk-8-jdk-headless
- openjdk-8-jre-headless
- ant
- libsm6
- libice6
- libxext6
- libxrender1
- libxkbcommon-x11-0
- libfontconfig
- libdbus-1-3
- libx11-xcb1
- libc6:i386
- libncurses5:i386
- libstdc++6:i386
- libz1:i386

2. python Образ python с необходимыми для запуска программ на языке Python пакетами был использован в задании Timer, так как в ходе этого задания было необходимо запустить скрипт для построения графика по полученным временным характеристикам с помощью библиотеки matplotlib языка Python. Зависимости образа:

    - gnupg
    - dirmngr
    - git
    - mercurial
    - openssh-client

- subversion
- procps
- autoconf
- automake
- bzip2
- dpkg-dev
- file
- g++
- gcc
- imagemagick
- libbz2-dev
- libc6-dev
- libcurl4-openssl-dev
- libdb-dev
- libevent-dev
- libffi-dev
- libgdbm-dev
- libglib2.0-dev
- libgmp-dev
- libjpeg-dev
- libkrb5-dev
- liblzma-dev
- libmagickcore-dev
- libmagickwand-dev
- libmaxminddb-dev
- libncurses5-dev
- libncursesw5-dev

- libpng-dev
- libpq-dev
- libreadline-dev
- libsqlite3-dev
- libssl-dev
- libtool
- libwebp-dev
- libxml2-dev
- libxslt-dev
- libyaml-dev
- make
- patch
- unzip
- xz-utils
- zlib1g-dev

2.5 Модульное тестирование

Было произведено тестирование одного из модулей программы с помощью библиотеки testlib и фреймворка Qt Test.

Для тестирования был выбран модуль Surface, из которого была выбрана функция SetAutoNormal для примера теста, как наиболее простая для понимания и подбора данных для тестирования. Создан класс Test, наследуемый от класса QObject. В публичном слоте находится конструктор класса, а в приватном — непосредственно создаваемые тесты.

Листинг 2.2 – Класс Test для тестирования модуля Surface

```
1 class Test : public QObject {
2     Q_OBJECT
3 public:
4     explicit Test(QObject* parent = nullptr);
5
```

```
6 private slots:
7     void test_01();
8 };
```

Таким образом, чтобы добавить новый тест, достаточно создать новый приватный слот. В данном случае добавление теста будет выглядеть следующим образом:

Листинг 2.3 – пример добавления нового модульного теста

```
1 private slots:
2     void test_01();
3     void test_02();
```

Ниже приведен листинг реализации конкретного теста проверки рассчета нормали к заданной поверхности. Проверка корректности полученного результата производится с помощью макросов фреймворка Qt Test. В данном примере используется макрос QCOMPARE, сравнивающий ожидаемое значение с полученным.

Листинг 2.4 – реализация модульного теста test_01

```
1  void Test::test_01()
2  {
3      QColor cl(255, 4, 33);
4      QRgb r = cl.rgb();
5
6      QVector3D p1(0.0, 10.0, 0.0);
7      QVector3D p2(0.0, 0.0, 0.0);
8      QVector3D p3(0.0, 0.0, 10.0);
9
10     Surface c(Polygon(p1 , p2 , p3 , r));
11
12     c.SetAutoNormal();
13     QVector3D n = c.GetNormal().getNorm();
14
15     QCOMPARE(qFuzzyCompare(n.x(), -100.0), true);
16     QCOMPARE(qFuzzyCompare(n.y(), 0.0), true);
17     QCOMPARE(qFuzzyCompare(n.z(), 0.0), true);
18 }
```

Функция qFuzzyCompare используется для корректного сравнения чисел с плавающей точкой, так как QCOMPARE сравнивает числа с помощью оператора «==». Такое сравнение не используется для чисел с плавающей точкой, но подходит для сравнения значений типа bool, что и показано в приведенном примере.

2.6 Реализация управления из командной строки

Для автоматизации работы программы и использования сценария было необходимо реализовать передачу в программу аргументов командной строки и их распознавание самой программой. Для этого была использована библиотека getopt [12], позволяющая выполнить «сканирование» аргументов командной строки и их обработку. Конкретно была использована одноименная функция getopt, которая последовательно перебирает переданные параметры в программу в соответствии с заданной строкой параметров, содержащей имена параметров и признак наличия передаваемого значения («:»). Перечень используемых параметров:

- Флаг «-f» — указание программе о том, что нужно отрисовать первую сцену и сохранить полученное изображение.
- Флаг «-s» — указание программе о том, что нужно отрисовать вторую сцену и сохранить полученное изображение.
- Флаг «-t» — указание программе о том, что нужно отрисовать третью сцену и сохранить полученное изображение.
- Флаг «-time» — указание программе о том, что нужно провести замер времени и вызвать python скрипт, который сгенерирует графики.
- Флаг «-test» — указание программе о том, что проводится модульное тестирование.

Листинг 2.5 – пример вызова программы для замеров времени

```
1 #!/bin/bash
2
3 ./new_code -timer
```

2.7 Пример работы программы

На рисунках 2.7–2.9 представлены изображения, полученные в результате работы программы с тремя разными сценами, содержащими модели с различным количеством полигонов.

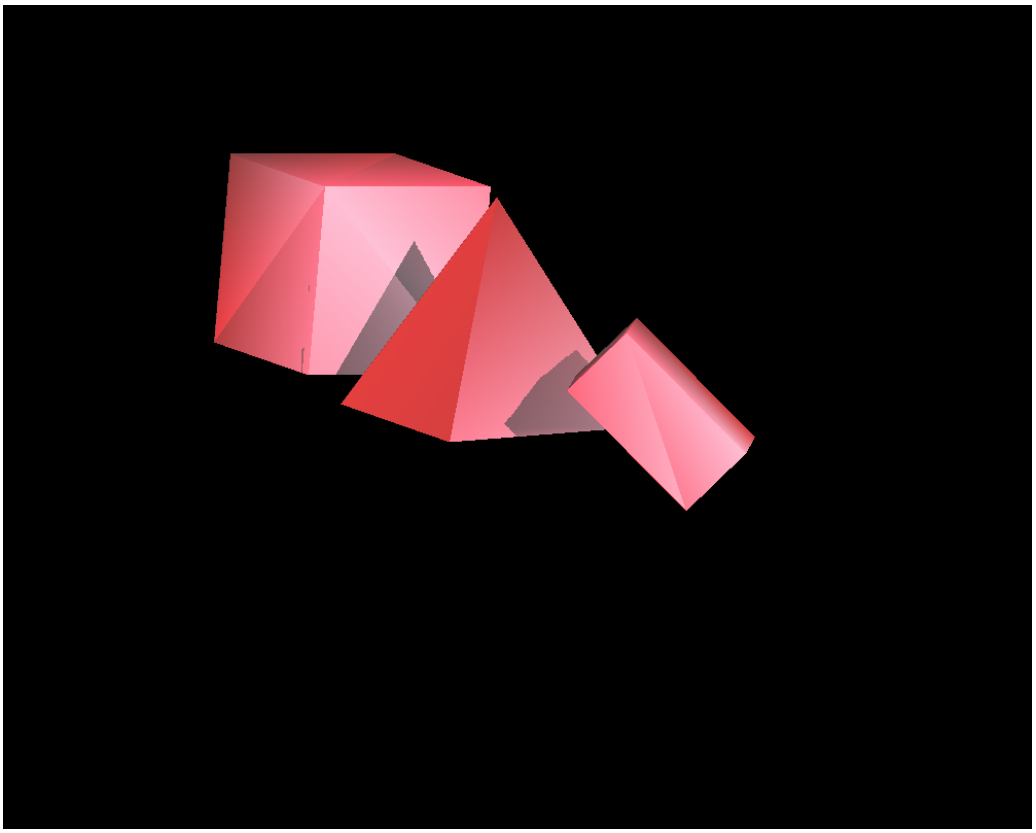

Рисунок 2.7 – пример №1 работы программы

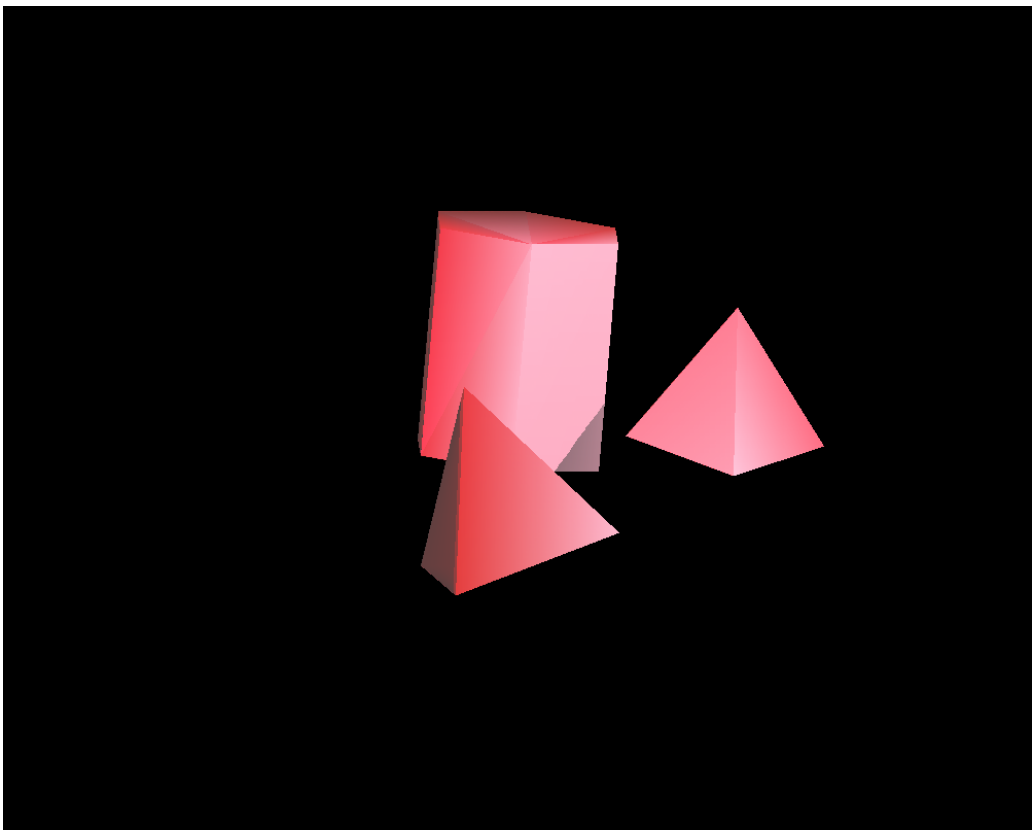

Рисунок 2.8 – пример №2 работы программы

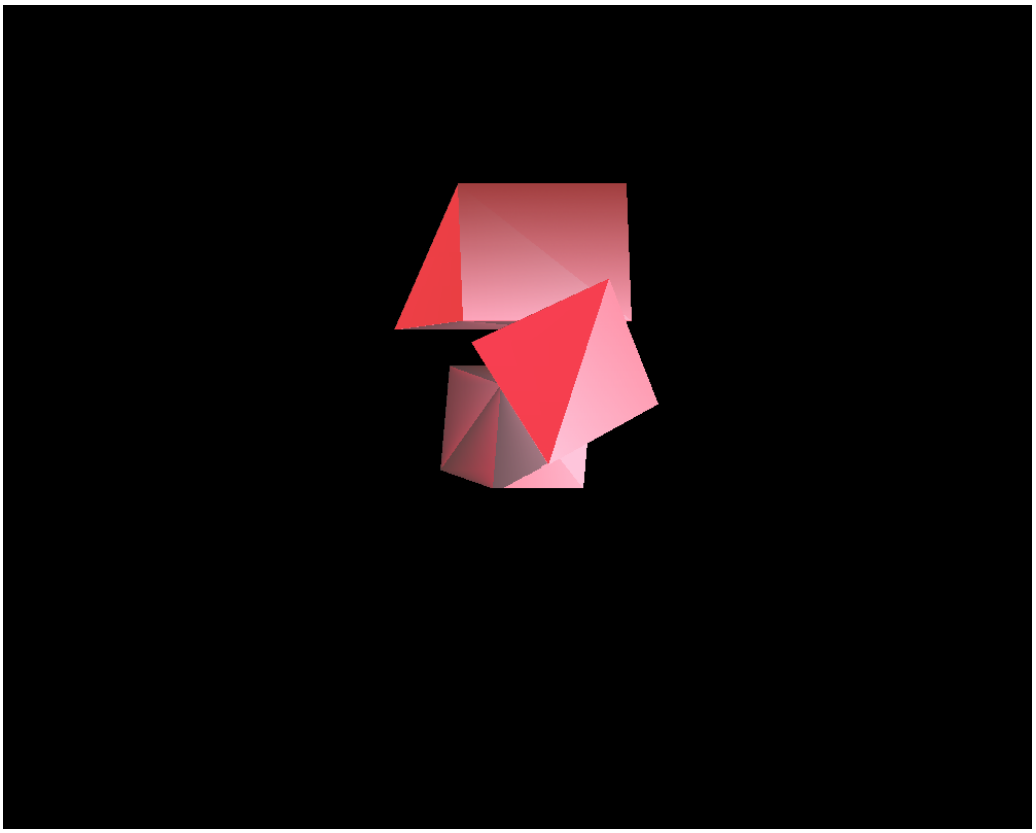

Рисунок 2.9 – пример №3 работы программы

Заключение

Во время выполнения поставленной задачи была изучена документация Qt, Qt Test, Qt Widgets, Gitlab CI/CD, Docker, ffmpeg. Были рассмотрены и реализованы алгоритмы удаления невидимых линий, построения теней, методы закрашивания и модели освещенности. Был создан сценарий автоматизации сборки, тестирования и получения данных будущего исследования. Было создано три сцены и проведено модульное тестирование для демонстрации работоспособности конвейера. Также был реализован замерный эксперимент зависимости времени выполнения программы от числа полигонов.

В ходе выполнения технологической практики были выполнены все поставленные цели и задачи.

Данная работа помогла закрепить полученные навыки в области компьютерной графики, приобрести опыт самостоятельной разработки программного продукта и проектирования программного обеспечения и изучить средства автоматизации развёртывания, сборки и тестирования программы.

Список литературы

1. Документация по Gitlab CI/CD (Электронный ресурс).
   Режим доступа: https://docs.gitlab.com/ee/ci (дата обращения: 16.07.2022).

2. Документация по Docker (Электронный ресурс).
   Режим доступа: https://docs.docker.com (дата обращения: 16.07.2022).

3. Документация библиотеки Qt (Электронный ресурс).
   Режим доступа: https://doc.qt.io (дата обращения: 01.07.2022).

4. Документация библиотеки Qt Widgets (Электронный ресурс).
   Режим доступа: https://doc.qt.io/qt-6/qtwidgets-index.html (дата обращения: 01.07.2022).

5. Документация библиотеки Qt Test (Электронный ресурс).
   Режим доступа: https://doc.qt.io/qt-6/qttest-index.html (дата обращения: 15.07.2022).

6. Документация по ffmpeg (Электронный ресурс).
   Режим доступа: https://ffmpeg.org/documentation.html (дата обращения: 18.07.2022).

7. The Most Common 3D File Formats in 2022 (Электронный ресурс)
   Режим доступа: https://all3dp.com/2/most-common-3d-file-formats-model/
   (дата обращения: 18.07.2022)

8. Comparison of different image formats using LSB Steganography (Электронный ресурс).
   Режим доступа: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=
   (дата обращения 15.07.2022)

9. Д. Роджерс. Алгоритмические основы машинной графики./Д. Роджерс. – М.: Мир, 1989. – 512с

10. Peter Shirley and Steve Marschner. Fundamentals of Computer Graphics - 3rd Edition, 2009. 782c.

11. Куров А.В., Курс лекций по дисциплине «Компьютерная графика»

12. Документация по библиотеке getopt. (Электронный ресурс).
Режим доступа: https://ru.manpages.org/getopt/3 (дата обращения: 05.07.202