«Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина)»
(СПбГЭТУ «ЛЭТИ»)

Направление	09.03.04 Программная инженерия
Профиль	Разработка программно-информационных систем
Факультет	КТИ
Кафедра	МО ЭВМ

К защите допустить

Зав. кафедрой Кринкин К.В.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

ТЕМА: Разработка инструмента упрощения 3D моделей.

Студент		_____ *подпись*	Ковалёв К.А.
Руководитель	к.т.н.	_____ *подпись*	Заславский М.М.
Консультанты		_____ *подпись*	Иванов А.Н.
	к.т.н.	_____ *подпись*	Заславский М.М.

Санкт-Петербург

2021

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Утверждаю
Зав. кафедрой МО ЭВМ
_____ Кринкин К.В.
«___»_____2021 г.

Студент Ковалёв К.А. Группа 7303

Тема работы: Разработка инструмента упрощения 3D моделей

Место выполнения ВКР: Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина)

Исходные данные (технические требования):

ОС Linux, Windows, Bash, CMD

Содержание ВКР:

Обзор предметной области, Выбор технологий и сред разработки, Проектирование инструмента упрощения, Реализация пользовательского интерфейса, Тестирование приложения, Безопасность жизнедеятельности.

Перечень отчетных материалов: пояснительная записка, иллюстративный материал.

Дополнительные разделы: Безопасность жизнедеятельности.

Дата выдачи задания «_22_»__апреля_____2021_ г.

Дата представления ВКР к защите «11»__июня___2021_ г.

Студент _____ Ковалёв К.А.

Руководитель к.т.н. _____ Заславский М.М.

КАЛЕНДАРНЫЙ ПЛАН ВЫПОЛНЕНИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Утверждаю
Зав. кафедрой МО ЭВМ
_____ Кринкин К.В.
«___»_____2021 г.

Студент Ковалёв К.А. Группа 7303

Тема работы: Разработка инструмента упрощения 3D моделей

№ п/п	Наименование работ	Срок выполнения
1	Обзор предметной области	01.04 – 05.04
2	Аналитическая часть	05.04 – 08.04
3	Разработка инструмента упрощения	08.04 – 12.04
4	Разработка пользовательского интерфейса	12.04 – 28.04
5	Тестирование приложения	28.04 – 13.05
6	Безопасность жизнедеятельности	13.05 – 23.05
7	Оформление пояснительной записки	23.05 – 01.06
8	Оформление иллюстративного материала	01.06 – 04.06
9	Предзащита	03.06

Студент _____ Ковалёв К.А.

Руководитель к.т.н. _____ Заславский М.М.

АННОТАЦИЯ

3D МОДЕЛИ. УПРОЩЕНИЕ ПОЛИГОНАЛЬНЫХ ОБЪЕКТОВ

Объектом исследования являются методы упрощения сетки полигонов 3D-модели.

Предметом исследования является создание инструмента для упрощения 3D моделей.

Цель работы – проектирование и разработка приложения по упрощению полигональных 3D моделей.

В настоящее время увеличивается скорость развития сенсоров для построения 3d-моделей. Полученные модели активно используются в картографии и системах дополненной реальности. В данной статье представлено описание и архитектура разработанного приложения для просмотра и упрощения полигональных 3D моделей. Было проведено сравнение существующих методов упрощения по поставленным приложением критериям. Также описаны затраты разработанного приложения по времени выполнения и изменению топологических свойств.

ABSTRACT

Currently, the speed of development of sensors for building 3d models is increasing. The resulting models are actively used in cartography and augmented reality systems. This article presents a description and architecture of the developed application for viewing and simplifying polygonal 3D models. A comparison of existing simplification methods was carried out according to the criteria set by the application. It also describes the testing of the developed application in terms of time costs and changes in the topological properties of models.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время активно развиваются технологии получения и обработки информации об объектах реального мира с помощью сенсоров машинного зрения. Полученные данные используются для создания трехмерных моделей объектов реального мира.

Созданные модели зачастую обладают высокой плотностью и большим количеством точек, из-за чего объем хранения данных моделей сильно увеличивается без заметного повышения качества или точности.

Несмотря на активное использование данных технологии до сих появляется необходимость в создании инструментов для уменьшения сложности данных моделей с сохранением линейный размеров и объемов. Подобные инструменты предоставляют пользователям возможность управления процессом упрощения и предоставляют статистические и визуальные данные после обработки. Данная работа посвящена созданию инструмента для упрощения полигональных 3D моделей.

1.1. Постановка задачи.

В ходе разработки приложения для упрощения полигональных 3D моделей необходимо реализовать необходимые структуры хранения данных, интерфейс пользователя для взаимодействия с обрабатываемой моделью, а также реализовать визуализацию в зависимости от варианта использования с поддержкой следующего функционала:

- Возможность загружать необходимую модель из файла.
- Сохранять обработанную модель в указанный файл.
- Обозревать обрабатываемую модель с возможностью перемещения камеры относительно модели.
- Поддерживать несколько режимов отображения данных.
- Упрощать модель в зависимости от параметров, задаваемых пользователем.

1.2. Цель, задачи, объект и предмет исследования работы

Цель работы: Проектирование и разработка приложения по упрощению полигональных 3D моделей.

Для достижения заданной цели необходимо решить следующие набор задач:

- Провести обзор предметной области.
- Определить стек технологий для разработки приложения.
- Провести обзор существующих алгоритмов уменьшения количества полигонов модели.
- Реализовать алгоритм упрощения полигональной модели.
- Спроектировать и реализовать приложения для обработки и визуализации модели.
- Провести тестирование разработанного приложения.

Объект исследования работы: Методы упрощения сетки полигонов 3D-модели.

Предмет исследования работы: Создание инструмента для упрощения 3D моделей.

1.3. Критерии к используемому алгоритму.

Так как разрабатываемое приложении будет использоваться для оценки геометрических характеристик модели, необходим алгоритм для уменьшения большого количества полигонов модели за относительно небольшое время, а также с сохранением топологии полученной модели. Следовательно, для используемого алгоритма можно выделить следующие критерии:

- Уменьшать количество полигонов модели.
- Сохранять целостность.
- Сохранять топологию.

- Обладать возможность останавливать работу алгоритма при достижении необходимого качества модели.

2. Обзор предметной области

На данный момент существует большое количество различных алгоритмов упрощения, или же для уменьшения количество полигонов поверхностей. Данные алгоритмы различаются по многим критерия, однако основным критерием, который будет рассмотрен далее, является способ удаления геометрии, по которому все алгоритмы можно разделить две группы: алгоритмы, использующие прореживания, или же выборку.

2.1. Методы удаления геометрии.

Прореживание - итеративный метод удаления геометрии полигональной модели. За одну итерацию объединяются вершины, удаляется вершина, ребро или треугольник с последующей триангуляцией для заполнения дырки при необходимости. При достижении целевого числа полигонов алгоритм завершается.

Существует несколько операций по удалению геометрии, использующиеся при прореживании:

1. Объединение вершин. В ходе данной операции происходит замена трех вершины одной с удалением всех граней и ребер, соединяющих данные вершины. После данной замены следует триангуляция для восстановления целостности поверхности после удаления граней.
2. Удаление вершины. В данном случае происходит удаление конкретной вершины со всеми зависимыми гранями и ребрами.
3. Схлопывание ребра. В ходе данной операции происходит объединения двух связных вершин с последующим удалением зависимых граней, а также связывающего данные вершины ребра.
4. Схлопывание пары. Данная операция работает похожим образом, однако допускает отсутствие ребра между объединяющимися вершинами.

5. Удаление треугольника. Данная операция завершается удалением треугольника со всеми связывающими его ребрами с последующей триангуляцией.

Проведя сравнение данных методов можно сделать следующие выводы: все операции уменьшают количество вершин обрабатываемой модели с сохранением её целостности. Методы 1 и 5 используют триангуляцию для восстановления целостности обрабатываемой поверхности, что может заметно увеличить время работы, а также данные методы сильно влияют на сохранении топологии модели. Методы 2, 3 и 4 работаю быстрее, однако методы 2 и 4 могут давать плохие результаты при сохранении качества.

Выборка – процесс быстрого уменьшения количества полигонов. Перед использованием происходит пересечение модели равномерной трехмерной сеткой, с помощью которой происходит разделение модели на ячейки. Далее для каждой ячейки происходит упрощение. Однако, из-за неоднородности модели данный способ может приводить к потере качества при упрощении.

Оценив описанные варианты можно составить следующие базовые критерии для реализуемого алгоритма:

- Использовать метод прореживая вершин.
- В качестве основной операции использовать схлопывание вершины, дающее наилучшие результаты при сохранении топологии.

2.2. Реализация алгоритма упрощения

Так в качестве основного метода обработки вершин выбрано прореживание, то реализуемый алгоритм должен работать итеративно, двигаясь по граням модели и обрабатывая входящие в их состав вершины. Также, исходя из критериев к данному алгоритму необходимо наличие параметра, указывающего на финальное количество граней модели после обработки, а также некоторого порогового значения(treshold), ограничивающего схлопывание граней с небольшой ошибкой. Благодаря

данном параметру первоначально будут удаляться грани, дающие наивысшую точность упрощения. В качестве ошибки используется квадратичный многочлен для нахождения квадратов расстояний от точки до плоскостей соседних треугольников и задающийся квадратной матрицей 4x4.

Таким образом, можно составить следующую последовательность действий алгоритма:

- Итеративное прохождение граней меша.
- Определение квадратично ошибки для каждого ребра текущего треугольника и сравнение с пороговым значением.
- Расчет оптимальной позиции новой вершины.
- Удаление двух вершин с последующим пересчетом ошибки для всех связанных треугольников.

Стоит отметить, что подобный алгоритм не способен упрощать модель до полного удаления полигонов, так как реагирует на серьезные изменения в топологии модели. В таком случае, при достижении критического количества полигонов алгоритм может зайти в тупик, так как не сможет найти ребро для удаления, ошибка которого была бы меньше порогового значения, что сильно влияет на скорость работы алгоритма. Для этого было введено максимальное количество итераций для остановки алгоритма в подобных ситуациях. Данное значение можно регулировать в зависимости от нужд пользователя. Уменьшая пороговое значение ошибки можно добиться уменьшения количества полигонов до минимального значения с серьезной потерей качества модели. Однако данное поведение алгоритма заметно исключительно при малом количестве полигонов модели.

2.2.1. Расчет ошибки.

Для расчета ошибки каждая точка представляется как точка пересечения плоскостей, с которыми связана данная вершина. Для полученного набора

плоскостей находится квадрика, которая определяет удаленность заданной точки от оптимальной позиции. Далее ошибку вершины можно найти по следующей формуле:

$$\Delta(v) = \sum_{planes}(p^T v)^2,$$

Где p = $[a\ b\ c\ d]^T$ и представляет собой уравнение плоскости конкретного треугольника при $a^2 + b^2 + c^2 = 1$. Далее ошибка может быть записана в следующей квадратичной форме:

$$\Delta(v) = \sum_{planes}(v^T p)(p^T v) = \sum_{planes} v^T(p^T p)v,$$

Где $p^T p$ – квадрика K_p, которую можно записать матрицы.

Полученная квадрика позволяет найти расстояние любой точки в пространстве до плоскости p. Также стоит отметить, что полученная матрица является симметричной, что позволяет хранить только 10 чисел матрицы вместо 16.

В качестве расчета порогового значения используется следующая формула:

$$treshold = E * i^{agr},$$

Где E – множитель масштабируемости ошибки, в данном случае равный 10^{-9}, i – номер итерации, agr (*aggressiveness*) – скорость увеличения порогового значения.

2.2.2. Определение оптимальной позиции.

После схлопывания двух соседних вершина необходимо определить позицию для новой вершины. Получив квадратичную матрицу позицию новой вершины можно найти как произведение части обратной квадратично матрицы и вектора [0 0 0 1]. Так как позиция вершины является трехкомпонентной, то при расчете отбрасывается 4 строка матрицы.

Если же обратной матрицы не существует, то новая позиция вершины выбирается из трех следующих вариантов:

- $v = v_1$
- $v = v_2$
- $v = \frac{(v_1 + v_2)}{2}$

Конкретный вариант выбирается в зависимости от значения ошибки в каждой из данных точек.

3. Архитектура приложения

В данном разделе представлено описание архитектуры и примеры реализации разработанного приложения. Также дано описания стека технологий для разработки и визуализации.

3.1. Используемые технологии.

Так как разрабатываемое приложение должно обрабатывать большие объемы данных за короткое время, было принято решения взять C++ в качестве основного языка разработки. Также с использованием данного языка появляется возможность написания графического интерфейса на довольно низком уровне за короткое время с использованием готовых графических библиотек, а также GUI библиотек, что значительно ускоряет работу программы при визуализации значительного количества данных, в нашем случае полигонов.

В качестве основной графической библиотеки был выбран **OpenGL**, так как данная библиотека является наилучшим решением при разработке графики на платформе Linux при использовании языка разработки C++. Также для ускорения процесса разработки была выбрана библиотека **GLEW**, упрощающая запросы и загрузку расширений OpenGL.

В качестве дополнительной библиотеки для создания окон была использована библиотека GLFW, написанная на OpenGL и позволяющая создавать и инициализировать оконный контекст при помощи определенных

методов, что значительно упрощает разработку, ограничивая разработчика от создания системы обработки событий при взаимодействии пользователя с интерфейсом программы

Для создания интерфейса программы была использована библиотека **ImGui**. Данная библиотеки представляет удобные инструменты при создании окон и виджетов для удобного взаимодействия с приложением.

При разработке было принято решения отойти от использования кроссплатформенного IDE Qt из-за большого объема дополнительных библиотек, необходимых для создания приложения, также использование ImGui дает соразмерный результат при создании окон, использует меньшее количество сторонних библиотек, а также повышает скорость написания кода.

3.2. Основной программный цикл.

Для реализации основного программного цикла был разработан класс singleton класс Application с определенным методом run, внутри которого обрабатываются команды пользователя при взаимодействии с интерфейсом программы.

Данный классный является singleton классом, то есть объект данного класса может существовать в единственном экземпляре, который хранится как статическое поле класса и инициализируется после первой попытки создания объекта. Данное решение обусловлено тем, что для разрабатываемого приложения достаточно наличия единственного контекста работы, в котором обрабатываются события по обработке данных и визуализации полученных результатов.

Основные методы класса Application:

- Application() – Используется для инициализации хранимых данных, а также для создания контекста работы gui приложения и инициализации callback методов по обработки действий пользователя.

- run() – Запуск основного цикла работы программы, а также инициализация компонент для визуализации обрабатываемых данных, таких как шейдеры. В основном цикле происходит обработка событий от пользователя, таких как нажатие клавиш устройств ввода или изменения размеров рабочего окна.
- getInstance() – Используется для получения экземпляра объекта класса. Данный метод является публичный и используется для получения доступа к объекту от callback функций или gui составляющей приложения.
- resetInstance() – Освобождает экземпляр класса.

Остальные методы являются контекстно зависимыми и будут рассмотрены в следующих разделах.

После запуска приложения происходит инициализация и вызов соответствующего метода run, который запускает рабочий цикл программы.

3.3. Обработка событий.

Так как пользователь взаимодействует с приложением через графический интерфейс, необходима система обработки событий, таких как нажатие клавиши, движения мыши, изменения размеров окна и т.д. Используемая библиотеку GLFW способна обрабатывать данные события, однако для этого необходимо определение callBack функций, которые будут является обработчиками данных события. Однако, в ходе разработки следующая последовательность обработки событий:

- Вызов callBack функции при наступлении очередного события.
- определение подтипа события и сбор данных о событии.
- Передача полученной информации в метод обработчик.

В качестве обработчиков служат специальные методы, определенные в классе Apllication, что дает доступ к приватным данным данного класса.

Данное решение необходимо для получения доступ к данным о сцене, таким как камера, свет или меш. Также использование подобной системы позволяет наследоваться от класса Application и переопределять методы по обработке событий без создания собственных callBack функций, используя один и тот же формат данных.

3.3.1. Определение состояния.

Во время работы программы иногда необходимо знать состояние устройств ввода без привязки к конкретным событиям, так как генерация подобных события была бы слишком затратной в процессе выполнения программы. Для того, чтобы не хранить в памяти информацию о последних событиях, был разработан класс **Input**, предоставляющий данные о состоянии клавиатуры и мыши. Такими данными служат: позиция мыши на экране, а также информация о том, нажата ли конкретная клавиша мыши или клавиатуры.

Обращаясь к данному классу и получая данные о состоянии мыши была реализована система по перемещению и вращению камеры вокруг рассматриваемой модели. Подробнее про работы с камерой описано в разделе **GUI**.

3.4. Хранение данных.

В качестве обрабатываемых данных выступает полигональная сетка, для которой был разработан шаблонный класс ***template <typename TVertexComponents> class Mesh***, который хранит информацию о вершинах и гранях модели. Так как для данного алгоритма упрощения достаточно исключительно информации о вершинах и гранях, данные о ребрах или полу-ребрах(half-edge) было решено не хранить для экономии ram. Для использования функционала данного класса и его наследников необходимо, чтобы тип TVertexComponents был наследником класса VertexComponentsColored, в котором уже заложены основные компоненты вершин, которые будут описаны ниже.

Основные поля:

- m_vertices – список вершин сетки.
- m_faces – список граней сетки.

Данные поля являются приватным и предоставляются при помощи соответствующих методов vertices() и faces(), которые предоставляют доступ только на чтение для предотвращения их изменения вне основного функционала.

3.4.1. Данные о вершинах.

Для хранения информации о вершине используется шаблонный класс **template <typename TVertexComponents> class Vertex**, который содержит единственное публичное поле TVertexComponents components. Данный шаблон является структурой наследником абстрактного класса VertexComponents который хранит информацию о компонентах вершины, таких как позиция, цвет, нормаль и т.д. Благодаря наследованию появляется возможность расширять имеющийся набор данных, хранимый вершиной, тем самым ограничивая расходы памяти на хранения ненужных данных. Также появляется возможность определения поведения компоненты вершины при изменении её позиции при помощи виртуального метода virtual void interpolate(…), который будет описан ниже.

Методы обработки вершин:

- interpolate(…) - Является линейной интерполяцией компонент вершины, однако может переопределяется разработчиком для каждого наследника при необходимости, например, для отключения интерполяции для uv координат. Данный метод используется для пересчета компонент вершин при схлопывании.

Основными компонентами вершины, которые используются приложением являются:

- Позиция.
- Нормаль. Необходима для определения «направления» грани и используется для вычисления объема модели и освещения на сцене.
- Цвет. Используется как замена текстурам.
- Координаты текстур. Используется для определения текселя(пикселя текстуры).

3.4.2. Данные о гранях

Для хранения информации о гранях используются классы Face или UVFace, в каждом из которых присутствуют три целочисленных поля для хранения индексов соответствующих вершин в вершинном списке.

Класс UVFace является наследником класса Face и хранит дополнительную информацию о текстурных координатах соответствующих вершин. Данная информация используется для инициализации вершинного буфера во время рендера, так как каждая вершина, являющаяся частью нескольких полигонов, может содержать разные текстурные координаты.

Таким образом список вершин и список граней с индексами соответствующих вершин позволяют хранить любую полигональную сетку.

3.5. Загрузка данных

Для загрузки меша из файла используется метод load_from_file, который принимает путь к исходному файлу. Внутри данного метода происходит обращение к классу OBJReader для открытия и считывания данного файла, передавая списки вершин и граней в качестве параметров. Подробнее считывание данных будет описано в разделе IO.

После считывания данных происходит расчет нормалей загруженной модели и инициализация материалов, если программа запущена в режиме с графическим интерфейсом. Подробнее про материалы будет сказано в разделе Визуализация.

3.6. Обработка данных.

В данном подразделе описаны методы по упрощению полигональной сетки, а также методы для оценки точности упрощения, такие как площадь, объем, а также значение квадратичной ошибки упрощенной модели.

3.6.1. Упрощение модели.

Для упрощения 3д моделей был написан блок функций в пространстве имен Simplify. Основой является шаблонная функция simplify_mesh, которая принимает указатель на объект класса Mesh и финальное количество граней модели. Полученных списков вершин и ребер из меша достаточно для запуска алгоритма, однако для работы алгоритма необходимо наличие дополнительных параметров для каждой вершины или ребра, таких как ошибка, список соседних граней и т.д. Для этого были добавлены отдельные классы для хранения вершины и ребра, которые представлены ниже.

- struct Vertex { vec3f p; int tstart, tcount; SymetricMatrix q; int border; };
- struct Triangle { int v[3]; double err[4]; int deleted, dirty; vec3f n; };

В начале алгоритма происходит копирование данных из меша в соответствующие списки вершин и граней с последующей инициализацией дополнительных параметров.

Также для хранения связей между вершинами и гранями добавляется список структур типа: struct Ref { int tid,tvertex; };

После инициализации всех данных алгоритм начинает работу, итеративно проходя по вершинам и удаляя те грани, значение ошибки которых меньше определенного значения(threshold). Как только удаляется очередное ребро, появляется необходимость определения позиции новой точки, а также изменение или установка компонент новой вершины. Позиция вершины находится исходя из значения с квадратичной ошибки в функции **calculate_error**, описание которой дано ниже. Для расчета значений дополнительных компонент вершины достаточно получить значение

весов(weights) для формулы линейной интерполяции, которое получается из отношения

$$t = \frac{|v - v_0|}{|v_1 - v_0|}$$

Далее для новой вершины достаточно вызвать метод v.**interpolate**(v_0, v_1, t), получив тем самым корректные значения компонент вершины.

После достижения указанного количества полигонов алгоритм прекращает работу и компонует списки вершин и граней. Стоит отметить, что никакого копирования значений из временного списка, необходимого для работы алгоритма, в исходный не происходит, так как все значения позиции вершин были получены из интерполяции, а индексы граней перезаписывались во время работы алгоритма одновременно с временными данными.

Для запуска алгоритма были реализованы две виртуальные функции, описание которых дано ниже:

- void simplify(float p = 0.5f) – Упрощает модель, оставляя долю граней, указанную в параметре p.
- void simplify(uint finalFaceCount) – Упрощает модель, оставляя указанное количество граней.

3.6.2. Расчет нормалей.

Нормаль можно определить по формуле:

$N = \left|\left|crossProduct(v_1 - v_0, v_2 - v_0)\right|\right|$, где v_0, v_1, v_2 – координаты конкретной вершины.

Для придания модели более сглаженного вида было принято решение хранить для каждой вершины только одну нормаль, которая является средним между всеми нормалями соединяющих её граней, что также позволяет избавиться от копий существующих вершин. Таким образом нормаль вершины определяется по следующей формуле:

$N = \left|\sum_i^k Ni\right|$, где Ni – нормаль связанной грани, которая вычисляется по формуле выше.

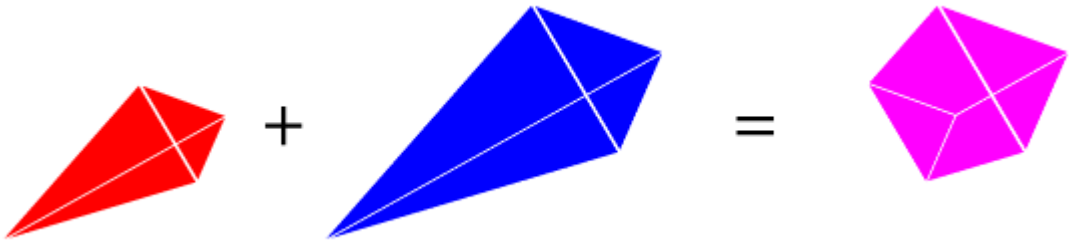
Данная формула реализуется в методе **calculate_normals** и используется после каждого загрузки или упрощения модели.

3.6.3. Площадь и объём модели.

Площадь модели оценивается исходя из суммы площадей всех её граней. Площадь отдельного же треугольника можно получить из формулы:

$S = |crossProduct(v_1 - v_0, v_2 - v_0)|$, где v_0, v_1, v_2 – вершины треугольника.

Объем модели можно оценить исходя из суммы знаковым объемов тетраэдров, образованных точками v0, v1, v2, O, где v0, v1, v2 – координаты вершин конкретного треугольника, а O – начало(origin) координат. Стоит отметить, что нас интересует именного знаковый объем, так как в нашем случае у каждого треугольника имеется конкретное направление.



Значение знакового объема можно получить из формулы:

$$\frac{dotProduct\big(v_0, crossProduct(v_1, v_2)\big)}{6}$$

Данные формулы реализованы в качестве публичных методов класса **Mesh**.

4. Визуализация.

В данном подразделе описаны методы для визуализации полигональной сетки. Дается общее представление об используемых технологиях, а также

информация о классах и методах, реализующих хранение, доступ и взаимодействие с данными для рендера.

4.1. Хранение данных.

После разработки структур для хранения данных о полигональной сетке необходимо позаботиться о хранении данных, которые будут использоваться для ее визуализации, так как простого списка для вершин и граней не хватает для корректного представления данных. Для этих целей был разработан шаблонный класс template **<typename TVertexComponents> RenderMesh**, который является публичным наследником класса **Mesh**.

Создание данного класса понадобилось для разделения данных меша, которые можно использовать без графической составляющей, и данных, которые необходимы **OpenGL** для рисования полигонов. В качестве таких данных выступают вершинные и индексные буферы, материалы и текстуры.

Так как визуальная составляющая меша не сильно важна в рамках данной статьи, в качестве материалов выступают исключительно текстуры или же цвет, которых достаточно для передачи пользователю образа модели.

4.1.1. Материалы и текстуры.

Инициализация данных для рендера осуществляется в приватном методе initMaterials, который вызывается после каждой загрузки нового меша. В данном методе происходит привязка буферов OpenGL к текущим данным о вершинах и гранях. Стоит отметить, что списки вершин укомплектованы достаточно хорошо, чтобы привязка происходила без необходимости копирования информации. Это осуществляется при помощи параметров OpenGL, которые позволяют указывать смещение и отступ от буфера с данными для каждой компоненты, тем самым позволяя использовать один и тот же список вершин для разных буферов. Однако из-за наличия у вершины разных текстурных координат перед рендером необходимо создавать копии уже имеющихся вершин, что немного увеличивает общий объем, о чем будет написано в разделе 5.1.3.

Также в методе initMaterials происходит загрузка в память используемых моделью текстур, пути к которым уже хранятся в специальной структуре внутри классов для загрузки модели из файла, о которых будет написано в разделе **IO**.

4.1.2. Рендер-группы.

После загрузки текстур происходит инициализации индексного буфера, который используется для указания порядка рендера вершин в созданном заранее вершинном буфере, а также создание рендер-групп.

Render-groups – разработанная структура группировки вершин по типу используемого материала. Данная структура необходима для рендера вершин, которые разделяют одну и ту же текстуру, что дополнительно позволяет уменьшить количество drawCall-ов, а как следствие, увеличить производительность.

4.1.3. Создание копий вершин.

Несколько связанных граней могут использовать различные участки текстуры, или же совершенно разные. По этой причине у конкретной вершины, разделяющей подобные грани, могу быть разные текстурные координаты, которые невозможно хранить в самой вершине из-за нарушения «однородности данных». Чтобы обойти данную проблему было реализовано создание копий существующих вершин для каждой координаты текстуры, а также привязка граней с созданным копиям. Данное решение незначительно увеличивает общий объем хранимых данных, порядка нескольких десятков килобайт, что не очень существенно по сравнению с хранением текстур, которые в лучшем случае занимают несколько мегабайт.

Однако такое решение вызывает проблемы при упрощении расширенной модели, так как копии вершин не являются реальной частью модели и не могут обрабатываться алгоритмом. Появляется необходимость в хранении для граней, указывающих на копии вершин, данных об их реальных вершинах и восстановлении этих данных перед каждым упрощением. Для

этого была реализована структура **FaceNativeData**, хранящая информацию о гране и ее родных вершинах, а также соответствующий список.

4.2. Обработка данных

Так как класс RenderMesh добавляет дополнительные данные в текущий список вершин необходимо переопределить методы по обработке данных и выводе информации о модели. Для этого вводится новая приватная переменная класса m_vertices_count, хранящая реальное количество вершин.

Также переопределяются методы для упрощения исходной модели simplify(…), в которых перед упрощением происходит восстановление оригинальных данных модели, а также повторная инициализации буферов OpenGL, так как предыдущие данные были искажены удалением вершин.

4.2.1. Проблема сохранения текстур.

При упрощении модели с использованием текстур возникает проблема сохранения однородности текстур. Так как текстуры связаны с вершинами значением текстурных координат, хранящихся в компонентах самой вершины, возникает необходимость определение среднего значения при интерполяции (см. 3.6.1. – упрощение модели). Однако полученное значение может оказаться некорректным ввиду формата хранения текстур. Большинство текстур хранится в виде текстурного атласа, где на единственной текстуре хранятся несколько областей, соответствующих разным текстурам. Следовательно, при усреднении текстурных координат из разных областей получается значение, совершенно не относящееся к исходным областям, что вызывает артефакт, представленный на рисунке 4.2.

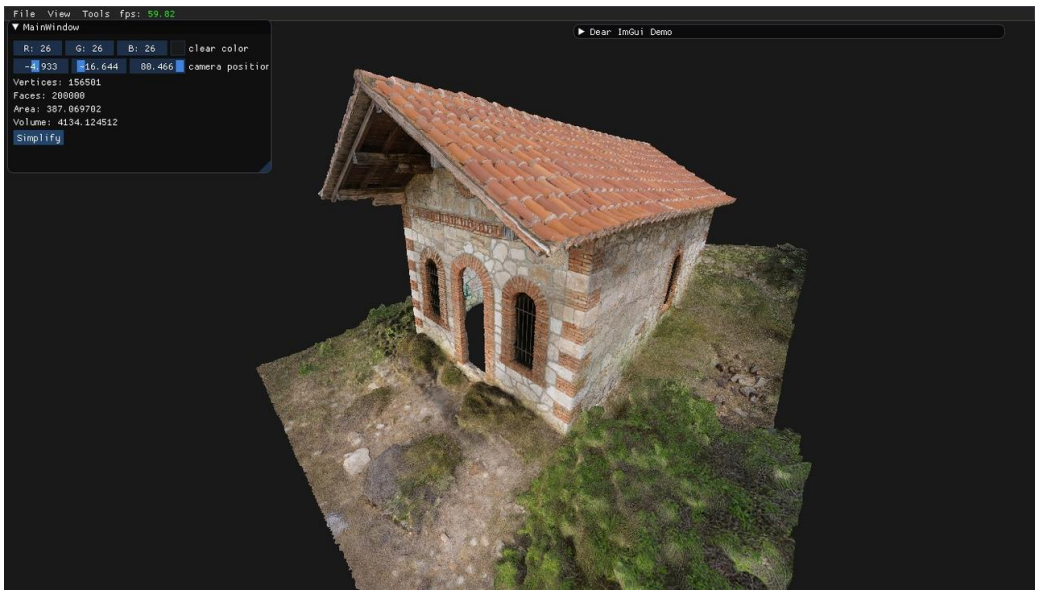


Рис 4.1. – Исходная модель.

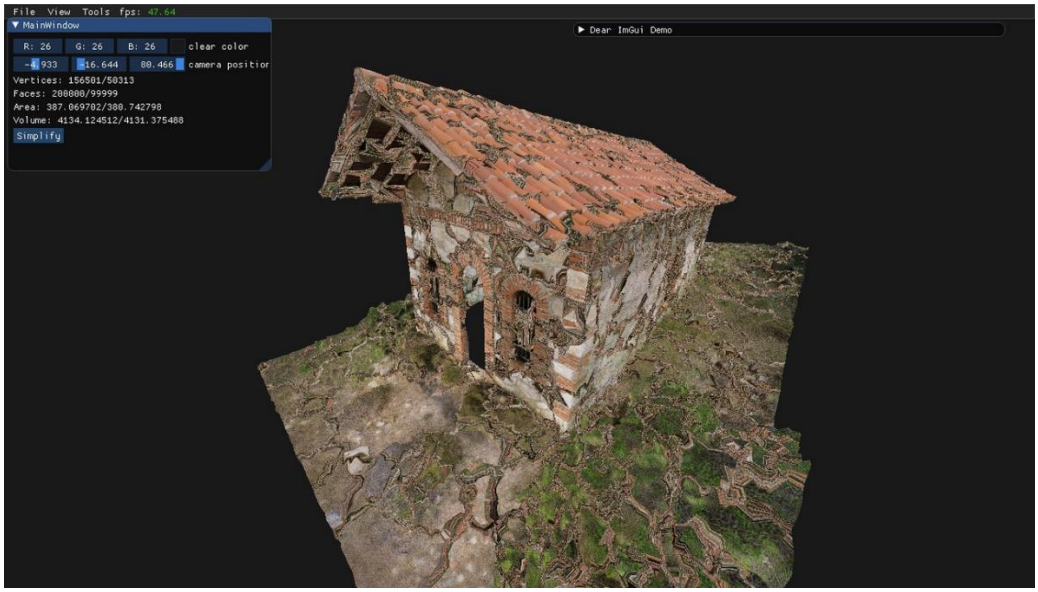


Рис 4.2. – Артефакты упрощенной модели.

Таким образом, сохранение однородности текстурных координат для текущей задачи невозможно, так хранимые в текстурном атласе области никак не разграничены.

Для разрешения данной проблемы можно предложить следующие варианты:

- Отказ от интерполяции текстурных координат в компонентах вершины. Данное решение позволит избавиться от разрывов в

текстурах, однако при сильном упрощении модели текстуры некоторых полигонов начнут преобладать и расползаться по поверхности модели. Данное решение позволяет сохранить использование текстур упрощенной модели. Пример реализации представлен на рисунке 4.3.

- Отказ от текстур в пользу цветов. Данное решение предполагает замену всех текстур соответствующими им цветами. Для каждого полигона возможно определить усредненное значение цветов используемой им части текстуры, после чего определить цвета составляющих данный полигон вершин. Так как конкретная вершина может входить в состав нескольких треугольников, разделяющих разные текстуры, то для вычисления цвета вершины следует взять взвешенное среднее значение цветов соседних треугольников в зависимости от их площади. Использование цветов может давать иной визуальный результат, однако это позволит корректно изменять цвета треугольников при упрощении модели и избавиться от описанных выше артефактов. Пример возможного результата представлен на рисунке 4.4.

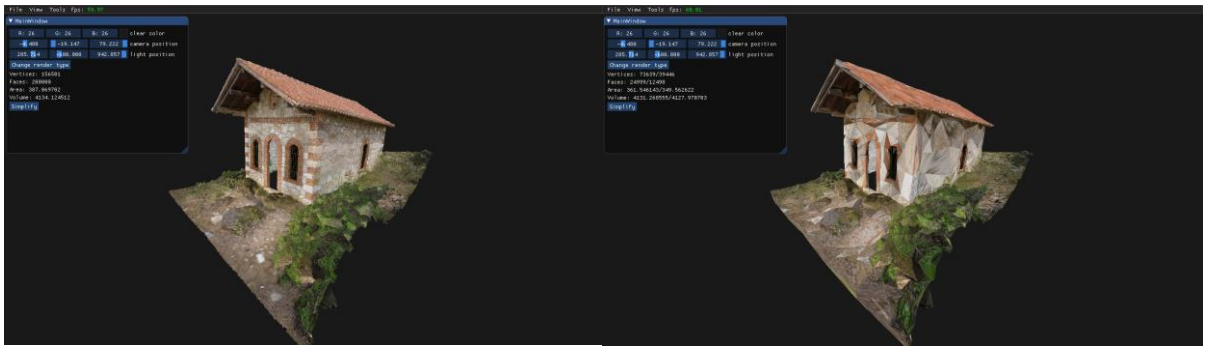


Рис. 4.3. – Использование текстур без интерполяции.

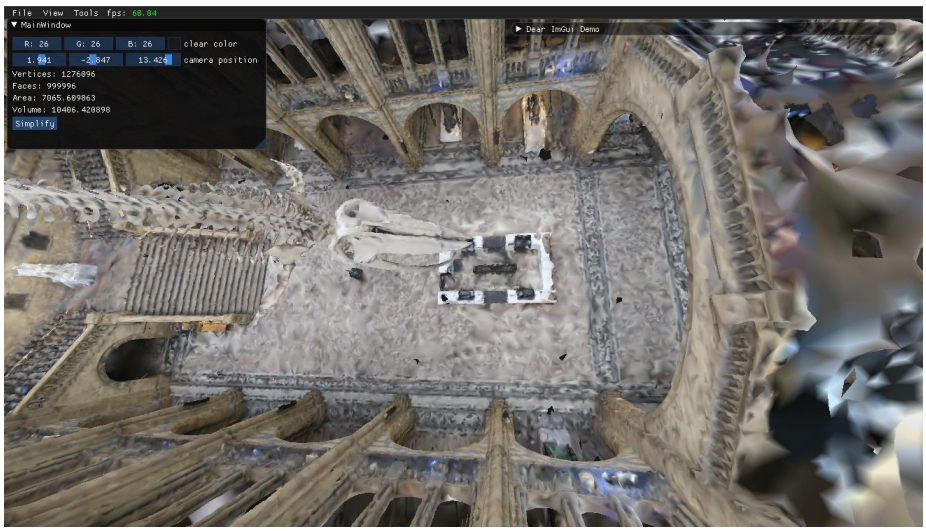


Рис 4.3. – Представление цветов.

В разработанном приложении был реализован первый вариант решения проблемы, так как является менее затратным и сложным в реализации, а также показывает хорошие результаты при упрощении модели в 5 – 10 раз.

4.3. Использование шейдеров

При использовании современных графических библиотек нельзя обойтись без написания шейдеров, которые являются удобным решением для полного контроля отображаемой информации.

Шейдер - программа, предназначенная для обработки данных для рендера процессорами видеокарты.

Существует большое количество различных шейдеров, каждый из которых предназначен для обработки конкретной информации. В рамках данной работы будут описаны только два используемых вида шейдеров: вершинный и пиксельный.

Вершинный шейдер - шейдер для обработки геометрических данных выводимого объекта. На вход данному шейдеру подается список примитивов для обработки, в нашем случае в качестве такого примитива выступает треугольник. После получения данных происходит расчет позиции вершин треугольника, а также подготовка компонент данного примитива для

обработки пиксельным шейдером.

Пиксельный шейдер - шейдер для вычисления цвета конкретного выводимого пикселя, который вычисляется за счет дополнительных параметров примитивов, таких как нормали, цвет, текстуры и т.д.

Для написания шейдеров был использован язык GLSL версии 1.30, который используется в OpenGL.

4.3.1. Освещение.

Освещение модели необходимо для возможности визуально определять геометрию обрабатываемой модели, так как может возникнуть ситуация с отсутствием цветов или текстур у загруженной модели. Освещение позволит изменять оттенок цвета полигона при разных его расположениях относительно источника, что повышает различимость пользователем соседних полигонов.

В рамках данной работы была реализована базовая модель освещения, базирующаяся на вычисления угла между направлением нормали вершины и направлением на источник света, значение которого заносится в промежуток от 0 до 1. Полученное же значение умножается на значение цвета вершины.

C = S * C, где C - цвет вершины, S - значение угла между нормальную и направлением на источник освещения.

S = max(0, dot(N, L)), где N - направление нормали, L - направление на источник освещения.

Стоит сказать, что для получения значения в пределах от 0 до 1 необходимо, чтобы используемые векторы были нормализованными.

Пример использования освещения представлен на рисунках 1 и 2.

4.3.2. Режимы отображения данных.

Использование шейдеров позволят определить разнообразные режимы отображения 3D модели, которые необходимы для анализа состояния модели.

В рамках данного решения были реализованы следующие режимы:

- Режим модели - отображение меша без использования цветов или текстур с использованием освещения, пример представлен на рисунке 4.4.

- Режим нормалей - отображение меша с использованием цветов, в качестве которых выступают нормали, что позволяет определить ориентированность конкретного участка меша. Так как значение нормали может варьироваться в пределах от -1 до 1, необходимо перевести эти значения в интервал от 0 до 1, что рассчитывается по формуле C = 0.5 * N + 0.5, где N - трехкомпонентное значение нормали, C - значение цвета для вывода. Пример отображения в данном режиме представлен на рисунке 4.5.

- Режим текстур - отображения меша с использованием текстур, или же цвета, в зависимости от хранимых моделью информации. Пример отображения представлен на рисунке 4.6.

- Режим ошибки - отображение меша с использованием в качестве цвета значения квадратичной ошибки модели. Пример отображения представлен на рис 4.7.

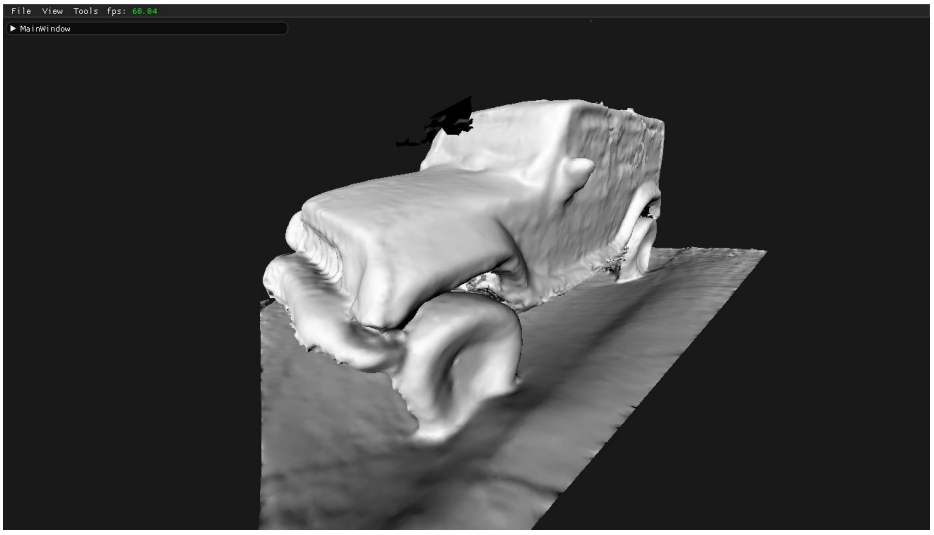


Рис 4.4. – Режим модели.

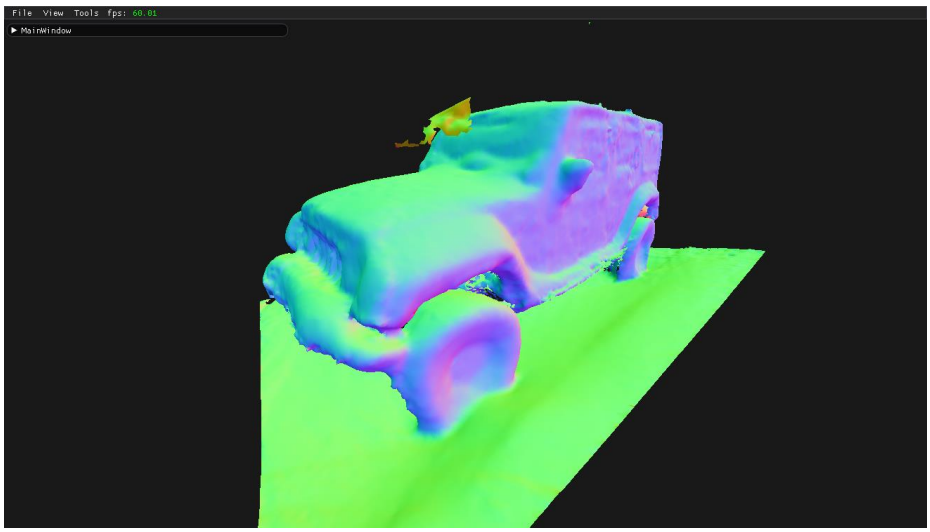


Рис 4.5. – Режим нормалей.

Рис 4.6. – Режим текстур.

5. Графический интерфейс пользователя.

В данном разделе описана реализация и архитектура графического интерфейса пользователя, а также способы взаимодействия. Дано описание различный элементов интерфейса, связанного с представлением, обработкой, и загрузке данных.

5.1. Создание GUI.

Для создания графического интерфейса используется библиотека с открытым исходным кодом ImGui использующая концепцию Immediate mode GUI. Способ создания окон при помощи предоставляемых библиотекой

функций отличается от способов, которые используются в Qt или же в GTK. Создание и описание окна происходит непосредственно в основном программном цикле с вызовом соответствующих функций, что позволяет связывать хранимые данные о модели с использующими их окнами во время создания.

Для использования вышеописанных функций необходимо наличие контекста, в котором будут создаваться необходимые окна. Данный контекст, а также главное окно приложения создаются с помощью OpenGL и функций библиотеки GLFW во время инициализации приложения. Также данный контекст используется для визуализации модели, однако не ограничиваются им.

Таким образом, интерфейс пользователя реализуется следующим образом:

- Создание главного окна и инициализация контекста работы.
- Передача контекста и инициализация ImGui.
- Отображение основных данных в главном окне.
- Отображение интерфейса пользователя.

5.2. Основные элементы.

Для представления данных о модели, а также для доступа методам и информации по обработке было создано несколько модальных окон:

- Основным окном для отображения модели является окно, созданное OpenGL, которое в свою очередь является базовым для размещения дочерних.
- Окно для предоставления информации по обрабатываемой модели, а также для сравнения характеристик модели до и после обработки.
- Окно загрузки и сохранения модели.
- Окно для инициализации процесса упрощения. В данном окне устанавливаются основные значения для упрощения модели, такие как

финальное количество граней и точность упрощения(threshold).

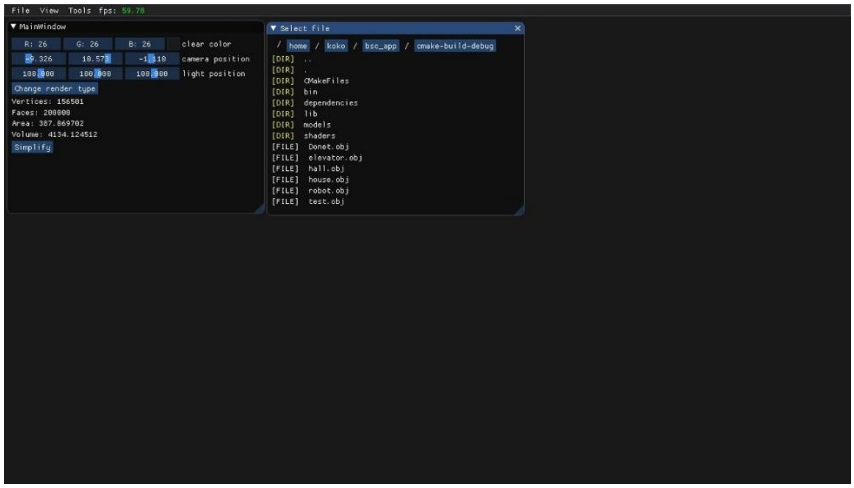


Рис 5.1. – Пример графического интерфейса.

5.2.1. Окно файловой системы.

Для загрузки и сохранения моделей необходим быстрый доступ к файловой системе. Для этого были разработаны соответствующее окно и модель файловой системы, использующаяся для поиска необходимых для загрузки файлов.

Данное окно позволяет перемещаться по дереву файловой системы в любых направлениях, а также выбирать файлы для загрузки или сохранения модели.

Для доступа к файловой системы были разработаны следующие функции:

- getCurrentDirectory – возвращает полный путь файла текущей программы.
- parseDirectory – функция для считывания всех файлов директории, передаваемой в качестве первого параметра. Возвращает список имен файлов, а также соответствующие типы файлов, описанные в перечисление FileType. В рамках текущей задачи искомыми типами файлов являются регулярные файлы и файлы директорий.

Данные функции используются для отображения файлов директории в текущем окне.

5.3. Просмотр модели.

Для более точного визуального анализа модели была реализована система для изменения положения камеры во время просмотра. Для этого реализуется структура Camera, объект которой является приватным членом класса Application и хранящая информацию о позиции и направлении камеры. Также создается вспомогательная структура CameraUpdateInfo, хранящая информацию о предыдущих положениях мыши на экране, а также реализующий методы по обновлению на их основе позиции и направления камеры.

Структура CameraUpdateInfo описывает следующие поля и методы:

- beginPosition и endPosition – являются двумерными векторами и хранят информацию о текущем и предыдущем положении мыши на экране. Данные поля используются для расчета смещения камеры вокруг или относительно рассматриваемой модели.
- rotate – Реализует вращение камеры относительно точки, используя вышеописанные данные.
- translate – Реализует перемещение камеры.

Данные структуры обновляются при наступлении определенных событий, связанных с обновлением позиции мыши или нажатием клавиши. Благодаря этому у пользователя появляется возможность перемещения и вращения камеры вокруг просматриваемой модели с использованием мыши.

5.3.1. Вращение камеры.

Вращение камеры происходит при зажатой левой клавиши мыши с одновременным движением мыши по экране, что проверяется следующими условиями при наступлении события перемещения мыши:

```
if (!ImGui::IsAnyWindowFocused()) {
```

```
        if (Input::mouseButtonPressed(GLFW_MOUSE_BUTTON_LEFT)) {
            cameraUpdateInfo.rotate(mMainCamera);
        } else if (Input::mouseButtonPressed(GLFW_MOUSE_BUTTON_RIGHT)) {
            cameraUpdateInfo.translate(mMainCamera);
        }
    }
```

В данном условии происходит разделение на вращение и перемещение камеры в зависимости от нажатой клавиши мыши.

Стоит отметить, что вращение или перемещение камеры необходимо исключительно при фокусировке мыши на главном окне, так как в противном случае изменение позиции камеры будет происходит при работе с дочерними окнами. Данное ограничение реализуется проверкой условия ImGui::IsAnyWindowFocused().

Вычисление новой позиции камеры производится при вызове функции cameraUpdateInfo.rotate(mMainCamera).

Для получение новой позиции камеры вычисляются углы поворота в сферической системе координат и переводящиеся в декартовы после вычислений. Процесс вычисление позиции камеры производится следующим образом:

- Вычисляется относительное смещение мыши на экране в диапазоне от 0 до 1 и сохраняется в переменную vector. Данное значение показывает, на какую долю экрана переместилась мышь за последний кадр.
- На основе полученного значения вычисляются соответствующие смещения углов с некоторым коэффициентом, которые в данном случае является некоторым аналогом скорости вращения.
- Вычисляются значения углов текущего положения камеры, с

последующим вычислением новых значений углов. Текущие значения углов находятся через вычисленный заранее вектор направления на камеру и вычисление арктангенса данного вектора в двух плоскостях.

- Производится вычисления позиции в декартовой системе координат, используя следующие формулы:

$$x = R * \sin(\theta) * \sin(\varphi)$$
$$y = R * \cos(\theta)$$
$$z = R * \sin(\theta) * \cos(\varphi)$$

где R – радиус сферы, по которой движется камера, который в данном случае является длиной вектора направления на камеру, θ и φ – значения углов в сферической системе координат.

5.3.2. Перемещение камеры

Перемещение камеры происходит при перемещении мыши с зажатой правой клавишей мыши и вычисляется в функции cameraUpdateInfo.translate(mMainCamera). Для определения новой позиции камеры необходимо, как и в случае с вращением, получение смещение позиции мыши на экране. Используя значение смещения на экране вычисляется смещение камеры в видовом трехмерном пространстве, т.е. в пространстве камеры. Однако для перемещения камеры в трехмерном пространстве необходим перевод полученного вектора перемещения из видового пространства в мировое. Для получения данных векторов строится обратная транспонированная матрица перехода, с помощью которой вычисляет нужный вектор:

```
glm::mat3 inverseViewMatrix = glm::inverse(glm::lookAtLH(camera.position, camera.lookPosition, { 0, 1, 0 }));

glm::vec3 moveVector(mouseVector * glm::length(camera.position - camera.lookPosition), 0.0f);

moveVector = inverseViewMatrix * moveVector;
```

5.3.3. Зум.

Зум реализуется с помощью приближения камеры к рассматриваемой модели и активируется при наступлении события использования колесика мыши. В зависимости от расстояния от камеры до модели вычисляется скорость приближения. Благодаря данной зависимости появляется возможность уменьшить скорость приближения при близком расположении камеры относительно модели

6. Импорт и экспорт модели.

В данном разделе описывается загрузка и сохранение модели из файла. Описывается процесс обработки файлов с данными о модели, а также файлов с материалами и текстурами.

6.1. Чтение данных.

Для загрузки моделей из файлов был разработан класс OBJReader, предоставляющий методы по обработке файлов типа .obj с сохранением данных в указанные структуры и записью модели в файл. Также в данном классе описаны основные структуры для временного хранения данных во время чтения, а также структуры для предоставления информации о порядке хранения данных.

6.1.1. Основные методы и структуры данных.

Ниже представлен список основных методов класса:

- read – осуществляет чтение модели из файла в указанные списки вершин и граней. Для записи данных в элементы данных списков использует Layout с указанием размера и смещения для каждого поля файла. Подробнее в layout будет сказано ниже.
- write – сохраняет модель в указанный файл в формате .obj.
- get_field_type – определяет тип поля считываемого файла для последующей обработки.

Также для считывания данных были реализованы следующие структуры и

перечисления:

- FieldType – перечисление для определения типа считываемых данных, таких как данные о вершинах, гранях, цветах и так далее.
- ParseMaterialInfo – структура для хранения информации о материалах после считывания. Данная структура используется для получения списка материалов в любое время после последнего считывания, так как информация о материалах или текстурах может использоваться выборочно, в зависимости от необходимости отображения считываемых данных. Благодаря этому считывание может производится для каждого из вышеописанных вариантов использования программы.
- Layout – специальная структура для указания формата сохранения считываемых данных в передаваемые списки. Так как структуры для хранения данных о вершинах или гранях могут быть разными, необходимо явно указывать размер и смещения для каждой из поддерживаемых данным форматом компоненты вершины. Базовыми компонентами данной структуры служат: позиция, нормаль, цвет и текстурные координаты. Для отказа от считывания определенных компонент вершин достаточно выставить смещение соответствующего поля лайаута в -1. Использование данной структуры позволяет отказаться от зависимости структуры хранимых данных от особенностей считывающего алгоритма.

6.1.2. Парсинг входных данных.

Чтение данных происходит построчно из-за особенностей используемого формата. Для каждой строки происходит определение типа хранимой информации, используя метод get_field_type, после чего, если в данной строке хранятся данные о компонентах вершины, для каждого типа данных происходит считывание и сохранение данных в соответствующие

структуры на основе переданного лайаута, в противном случае в текущей строке может храниться информация о материалах или текстурах, в данном случае происходит инициализация буферов для хранения данных о материалах с последующим считыванием вспомогательного файла, содержащего информацию об всех используемых моделью материалах, после чего для каждой грани в процессе считывания указывается имя используемого материала. Стоит отметить, что все считываемые материалы не возвращаются из в функции в качестве результата работы, а сохраняются в буфер типа ParseMaterialInfo для дальнейшего использования в процессе работы программы.

Чтение материалов происходит в специальной структуре MTLReader аналогичным способом, как и вышеописанные методы, так как файлы с данными модели и материалами имеют похожую структуру и формат хранимых данных. Также в пространстве имен данного класса определяются следующие структуры хранения информации о материалах:

- mtl::Material – Структура для связи названия материла и индекса используемой материалом текстуры в текстурном буфере. Стоит отметить, что данная структура хранит ссылку исключительно на текстуру, не учитывая другие данные, связанные с материалом, так как они не требуются в рамках данной задачи.
- mtl::MaterialInfo – Структура для связи конкретного материала с последовательностью граней, разделяющих данный материал. Даная структура хранит индексы начала и конца последовательности граней в общем списке граней данной модели, а также индекс используемого материала.
- mtl::Data – Структура для хранения связей материалов, текстур и разделяющих данные материалы граней. В данной структуре присутствуют следующие списки:
    1. textures – список с именами путей всех используемых

текстур.

2. materials – список объектов типа Material. Данный список используется для определения текстуры по индексу конкретного материала. В свою очередь, индекс материала получается из дополнительно созданного словаря material_map.
3. material_map – Специальный словарь для хранения связи между именем материала и его индексом в списке materials. Данный словарь необходим, так как при чтении входного файла модели используемые материалы ссылаются на место своего хранения не индексом, а названием.

Объекты структур mtl::Data и mtl::MaterialInfo являются частью структуры ParseMaterialInfo ссылаясь друг на друга. Таким образом появляется возможность получения данных конкретного материала как по его названию, так и по индексу в списке материалов. Получение данных по индексу необходимо для построения соответствия между списком материалов, полученным после считывания, и сгенерированного списка материалов класс RenderMesh, который преобразует полученные данные о материалах для удобного хранения и использования.

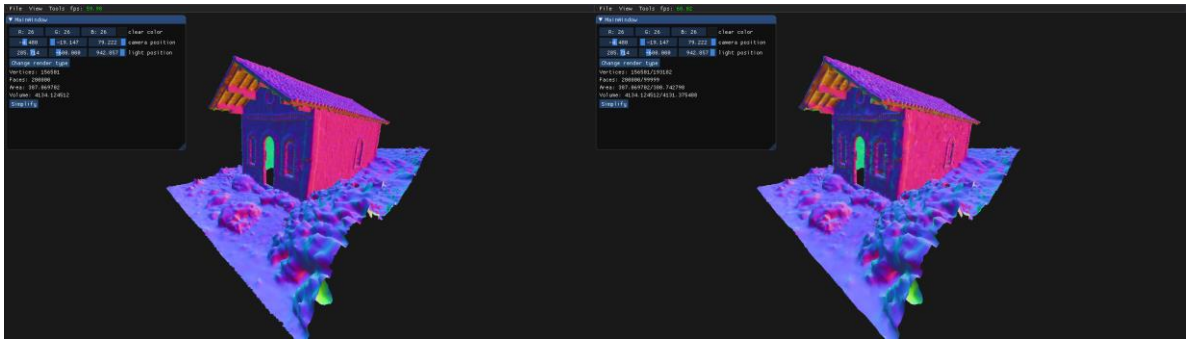
7. Результаты работы приложения.

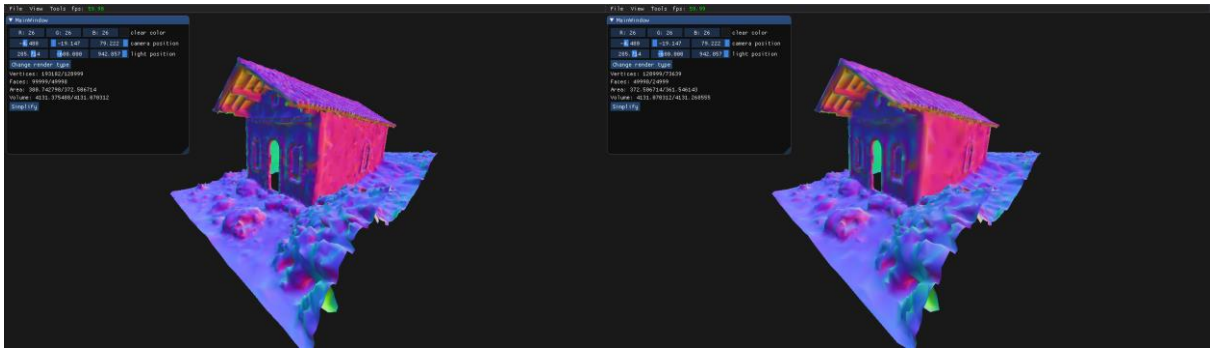
В данном разделе описываются результаты работы разработанного приложения. Приводятся сравнения затраченных на работу алгоритма времени и памяти на разных наборах данных, а также приводятся примеры работы при использовании различных режимов отображения данных. Производиться анализ сохранения топологии упрощаемых моделей для разного количества удаляемых граней. Также дается сравнение работы алгоритма с графическим интерфейсом пользователя и без.

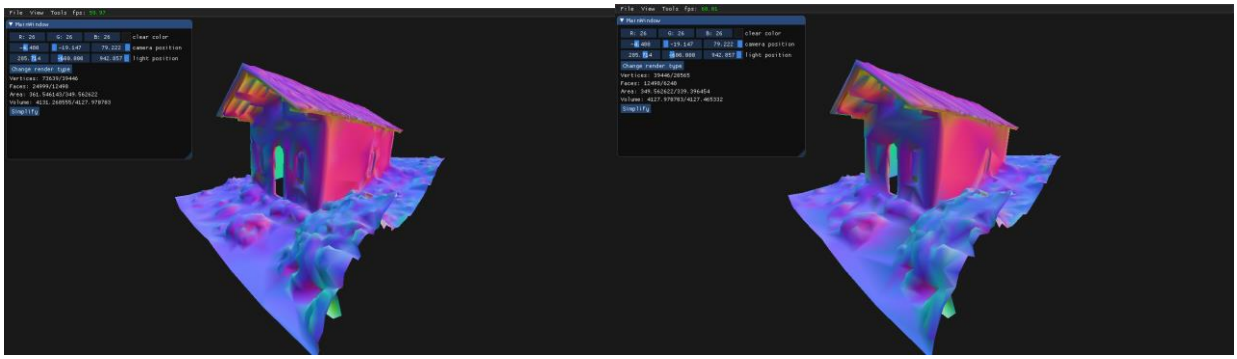
7.1. Примеры работы.

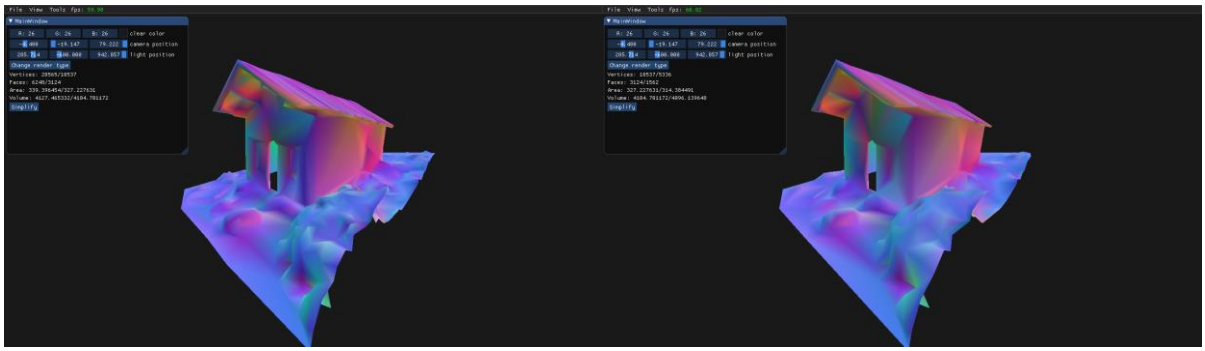
В данном подразделе представлены результаты упрощения модели при различных режимах отображения данных. Количество полигонов тестируемой модели уменьшалось в 2 раза при каждом запуске алгоритма. Также приведена таблица с параметрами модели, такими как количество граней, а также количество итераций, необходимых для завершения упрощения.

На следующих рисунках представлены результаты упрощения модели в 2 раза для 9 запусков алгоритма:









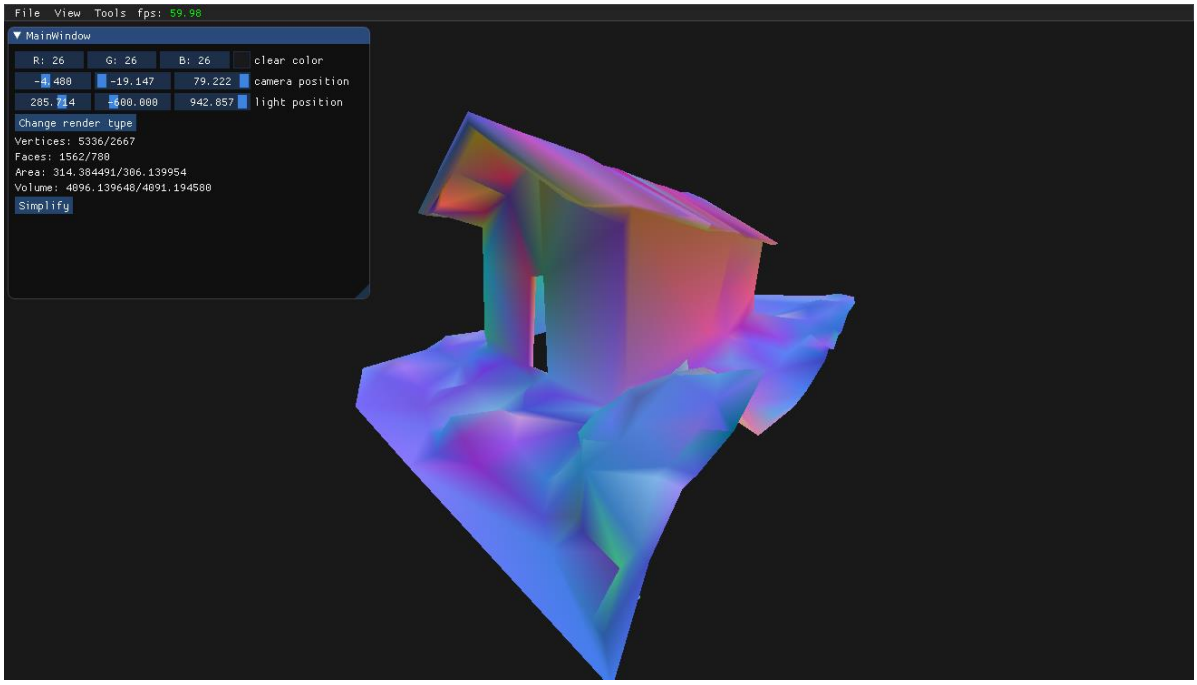


Таблица 7.1. - Характеристики модели

Номер запуска алгоритма	Количество граней	Площадь	Объем
1	200000	387.07	4134.12
2	99999	380.74	4131.38
3	49998	372.51	4131.07
4	24999	361.55	4131.27
5	12498	349.56	4127.97
6	6248	339.40	4127.47
7	3124	327.23	4104.70
8	1562	314.38	4096.14
9	780	306.14	4091.19

7.2. Временные затраты.

Для тестирования были выбраны три модели разной степени сложности и содержащие различное количество полигонов: от 10^3 до 10^6. Для каждой модели были проведено упрощение в 2, 10. 100 и 1000 раз, после чего были измерены затраты по времени и памяти для каждого упрощения. Данное тестирование проводилось с использованием графического интерфейса пользователя для визуализации полученных результатов упрощения.

В качестве максимального количества итераций алгоритма было выбрано значение в 100 итераций. Использование данного значения показывает хорошие результаты при упрощении, так как большая часть полигонов удаляется на первых нескольких итерациях.

Значение aggressiveness для расчета порогового значения было установлено в 7.

По заданным тестовым данным было проведено упрощение заданных тестовых моделей. По результатам вычислений были получены результаты, представленные в таблице 7.2.

Таблица 7.2. – Время работы алгоритма.

Номер модели	Масштаб упрощения			
	2	10	100	1000
1	0.860	1.227	1.310	1.401
2	0.867	1.194	1.305	1.382
3	4.613	6.270	6.897	> 24

7.3. Сравнение скорости работы без GUI.

При работе алгоритма с графическим интерфейсом и без нет разницы в скорости работы алгоритма упрощения, так как все данные, необходимые для работы алгоритма идентичны в обоих случаях. Однако при запуске программы

без графического интерфейса в вершинах не хранятся данные, требующиеся для визуализации модели, такие как нормали, цвет и текстурные координаты. Следовательно, алгоритм тратит меньше времени для их обработки при схлопывании ребра, что значительно уменьшает время работы. Также при работе без графического интерфейса программа не тратит время для подготовки данных, необходимых для работы алгоритма, а также на их восстановление после обработки. Таким образом, для оценки времени работы алгоритма необходимо для обоих случаев:

- Измерить время работы алгоритма упрощения.
- Измерить время выполнения подготовки и восстановления данных для работы алгоритма.

Значение времени работы алгоритма с графическим интерфейсом было рассчитано в предыдущем подразделе.

Для расчета времени были взяты модели, предложенные ранее.

7.3.1. Измерение времени подготовки и восстановления данных.

Отсчет времени начинается в данном случае начинается при вызове метода RenderMesh::simplify() и заканчивается при завершении его работы. Как было описано выше, перед вызовом алгоритма упрощения в методе RenderMesh::simplify() происходит восстановление оригинальной полигональной сетки с удалением дополнительных вершин, после чего, после завершения процедуры упрощения, происходит их восстановление.

Результаты измерений времени работы алгоритма с учетом времени подготовки и восстановления данных представлены в таблице 7.4.

Таблица 7.4. – Расширенные границы времени работы алгоритма

Номер модели	Масштаб упрощения			
	2	10	100	1000

1	2.044	2.377	2.442	2.549
2	3.390	3.615	3.756	3.791
3	7.246	8.798	9.303	>24

7.3.2. Выводы.

Проведя сравнение полученных данных можно сделать следующие выводы:

- Время работы алгоритма зависит от начального количества граней исходной модели.
- Время работы алгоритма слабо зависит от масштаба упрощения, так как большая часть полигонов удаляется на первых итерациях.
- Значительная часть времени работы алгоритма тратится на подготовку и восстановление данных модели для визуализации. В худшем случае время работы алгоритма без учета подготовки данных составляет 25% от общего времени работы.

7.4. Расчет используемой памяти.

Объем используемой памяти во время работы алгоритма фиксированный и зависит исключительно от количества вершин и граней обрабатываемой модели. Как было сказано выше, перед запуском алгоритма происходит копирование данных о вершинах и гранях в дополнительные буферы, так как в процессе работы алгоритма для обрабатываемых данных необходимы дополнительные параметры. Также в процессе работы алгоритма происходит удаление вершин и граней при схлопывании, тем самым освобождая некоторое количество памяти один раз за несколько итераций для предотвращения замедления работы программы при работе с памятью. Данный процесс называется компоновкой данных.

Для определения объема используемой дополнительной памяти достаточно определить объем структур данных для хранения вершин и граней. Полученный объем далее достаточно умножить на количество вершин и

граней соответственно. Таким образом, количество используемой памяти можно вычислить по следующей формуле:

$$V = V_v * N_v + V_p * N_p + V_{ref} * 3 * N_p,$$

где V_v и V_p – объемы структур данных для хранения данных о вершине и гране, N_v и N_p – количество вершин и граней соответственно, V_{ref} – объем структуры для хранения ссылок вершин на треугольники, в которые входит конкретная вершина.

Используя данную формулы были рассчитаны расходы памяти для представленных выше моделей. Результаты вычислений представлены в таблице 4.

Таблица 4- объем расходуемой памяти

Номер модели	Объем данных, МБ
1	26.70
2	16.93
3	161.81

8. Безопасность жизнедеятельности

ЗАКЛЮЧЕНИЕ

В ходе данной работы были проанализированы существующие методы упрощения полигональных 3D моделей, а также были выставлены критерии для используемого алгоритма и разрабатываемого приложения.

Для достижения поставленной цели был реализован алгоритм для упрощения полигональных 3D моделей на основе нахождения квадратичной ошибки. Также были разработаны структуры данных для хранения и обработки данных модели.

Было спроектировано и разработано приложения с графическим интерфейсом пользователя для визуализации результатов работы алгоритма, а также для предоставления пользователю возможности настраивать параметры алгоритма. Также были реализованы несколько режимов отображения данных.

Была проведена оценка времени выполнения реализованного алгоритма для разных вариантов использования. Также было определено количество затрачиваемой алгоритмом памяти и изменение геометрических характеристик модели при упрощении.

Таким образом все поставленные задачи в ходе данной работы были выполнены, а цель квалификационной выпускной работы была достигнута.

Дальнейшее исследование может быть нацелено на использование мощностей графический процессоров для обработки моделей, а также распараллеливание вычислений для достижения большей скорости работы алгоритма.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Hjelmervik J., Leon J. "GPU–accelerated shape simplification for mechanical based applications" Shape modeling international, pp. 91–102, IEEE Computer Society, 2007.
2. S. Shontz, D. Nistor, "CPU–GPU algorithms for triangular surface mesh simplification," in Proceedings of the 21st international meshing roundtable, pp. 475– 492, Springer, Berlin, 2013.
3. J. Vad'ura, "Parallel mesh decimation with GPU," 2011
4. A. Papageorgiou, N. Platis, "Triangular mesh simplification on the GPU," NASAGEM Geometry Processing Workshop, Computer Graphics International, 2013.
5. C. DeCoro, N. Tatarchuk, "Real–time Mesh Simplification Using GPU," in Proceedings of the symposium on interactive 3D graphics and games, 2007.
6. Surface Simplification Using Quadric Error Metrics https://www.ri.cmu.edu/pub_files/pub2/garland_michael_1997_1/garland_michael_1997_1.pdf
7. Measuring error on simplified surfaces https://www.researchgate.net/publication/227611629_METRO_Measuring_error_on_simplified_surfaces
8. Обзор методов упрощения полигональных моделей на графическом процессоре https://cyberleninka.ru/article/n/obzor-metodov-uproscheniya-poligonalnyh-modeley-na-graficheskom-protsessore/viewer
9. Метод геометрического упрощения 3D полигональных объектов https://cyberleninka.ru/article/n/metod-geometricheskogo-uproscheniya-3d-poligonalnyh-obektov/viewer
10. Mesh Simplification using Quadric Error Metrics https://users.csc.calpoly.edu/~zwood/teaching/csc570/final06/jseeba/#:~:text=Quadric%20Error%20Metrics%20are%20a,is%20from%20an%20ideal%20spot.&text=Before%20any%20mesh%20simplification%20happens,is%20evaluated%20using%20the%20quadric.

11. Dear ImGui open source library https://github.com/ocornut/imgui
12. OpenGL (Open Graphics Library) https://en.wikipedia.org/wiki/OpenGL
13. The OpenGL Extension Wrangler Library http://glew.sourceforge.net/
14. GLFW. Open Source, multi-platform library for OpenGL https://www.glfw.org/